

Dynamic Join Optimization in Multi-Hop Wireless Sensor Networks*

Svilen R. Mihaylov

Marie Jacob

Zachary G. Ives

Sudipto Guha

Computer and Information Science Department, University of Pennsylvania
Philadelphia, PA, U.S.A.

{svilen, majacob, zives, sudipto}@cis.upenn.edu

ABSTRACT

To enable smart environments and self-tuning data centers, we are developing the Aspen system for *integrating* physical sensor data, as well as stream data coming from machine logical state, and database or Web data from the Internet. A key component of this system is a query processor optimized for limited-bandwidth, possibly battery-powered devices with *multiple hop* wireless radio communications. This query processor is given a portion of a data integration query, possibly including joins among sensors, to execute.

Several recent papers have developed techniques for computing joins in sensors, but these techniques are static and are only appropriate for specific join selectivity ratios. We consider the problem of *dynamic join optimization* for sensor networks, developing solutions that employ cost modeling, as well as adaptive learning and self-tuning heuristics to choose the best algorithm under real and variable selectivity values. We focus on *in-network* join computation, but our architecture extends to other approaches (and we compare against these). We develop basic techniques assuming selectivities are uniform and known in advance, and optimization can be done on a pairwise basis; we then extend the work to handle joins between multiple pairs, when selectivities are not fully known. We experimentally validate our work at scale using standard datasets.

1. INTRODUCTION

A new class of monitoring and control applications is emerging, which integrates data from *multiple* networked sensor devices and Internet sources, to obtain high-level status information and ultimately support complex monitoring and control logic. Examples include hospitals automatically guiding visitors and physicians to patients or equipment, environmental monitors helping data centers optimize their energy consumption, or power grids using weather and usage forecasts to optimize electricity generation. Today's forerunners to these sensor systems run a single application and have limited extensibility to new devices or data types; the challenge in tomorrow's world will be supporting multiple applications and achieving seamless integration with data from other sensor systems, databases, feeds, streams, and Web services. To achieve this,

*Funded in part by NSF grants IIS-0477972, IIS-0713267, CNS-0721541, and DARPA HRO1107-1-0029.

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. Articles from this volume were invited to present their results at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

Copyright 2010 VLDB Endowment 2150-8097/10/09... \$ 10.00.

an easy-to-adopt, yet sophisticated and extensible development infrastructure is needed — which features a uniform interface for posing queries and defining views over all available data in the system (remote or local). The Aspen system (see overview in Appendix A) develops such an acquisition and integration architecture, with a particular emphasis on distributed sensor data.

Aspen uses a federated query processing model based on *Stream SQL*, where some query processing subsystems operate on server-type hardware, and others “scale down” to more limited devices and networks (e.g., ad-hoc wireless networks of Crossbow motes or PDAs). Our target applications require *joining* data from different types of sensors, and we support this task with a sensor query subsystem that is the focus of this paper. Numerous approaches have been proposed for computing joins in sensor networks. Yet, to the best of our knowledge, no prior work *optimizes* joins in a multi-hop wireless sensor network, taking into account operator selectivities to minimize message traffic (and thus network load, chance of dropped messages, congestion, latency, and power consumption). Moreover, in case of multiple concurrent queries, minimizing resource consumption is even more critical.

Join optimization in a sensor network differs from traditional distributed query optimization in several ways: (1) the *cost* minimized is often network utilization (thus power and congestion indirectly), rather than CPU or disk usage; (2) individual tuples are partitioned across different sensors, making access cost (in network messages) variable; (3) low bandwidth often necessitates decentralized optimization, with limited information at each node; (4) groups of sensors (not just pairs) may join, leading to possible optimizations; (5) data characteristics may vary across regions and over time. These differences require a new approach to the query optimization problem. To illustrate the types of queries we aim to answer (and optimize), we consider energy monitoring in a data center.

Query R. Consider an instrumented data center. A meter monitors the energy used by each machine, and temperature sensors monitor the area around the machines. Sensors are wireless to avoid dependence on a crashed or overheated server. When energy or temperature exceed a threshold, readings from adjacent sensors of both types should be paired up and reported to the base station. Low latencies allow the base station to immediately reduce the work allocated to overheated machines.

Query P. Given many racks of machines with temperature sensors, it may be convenient to use a mesh (multi-hop wireless) network to relay data from these sensors. An event should be triggered (i.e., a join output should be produced) if the difference between any pair of sensors in different regions exceeds a threshold.

These queries may not produce large amounts of output (since they are primarily event detectors). However, fast response times are essential (e.g., to support load balancing), and it is vital not to

drop join results due to congestion. Moreover, the data properties (selectivities, transmission rates) may vary across regions, making it essential to *adapt* to local and temporal variations.

In this paper we develop core techniques for addressing the challenges of query optimization in a highly distributed, decentralized multi-hop wireless sensor network, whether the sensor devices are primitive motes or powerful PCs. Our contributions are as follows:

- We develop *decentralized* algorithms for *cost-based query optimization* of joins, using distributed coordination.
- We develop a strategy that dynamically learns producer and operator selectivities and uses them for continuous query optimization in the presence of both spatial and temporal changes.
- Through extensive experiments using synthetic and real-world data, we experimentally demonstrate the effectiveness of our join optimization strategies versus previous approaches.

Our query optimization techniques address a number of important factors: non-uniform distribution of values, leading to different selectivities over different sets of sources; overlapping computation due to tuples that join with multiple other tuples; and dynamic changes, e.g., as devices fail or environmental conditions change. We assume that sensors are embedded within the environment (hence stationary) and that queries are *long-lived* and focused on *event detection* (where events are relatively rare). Our goals are to minimize message transmissions both in the long run and during an event (which translates into reduced congestion and battery usage) and reduce path lengths (which results in less latency).

Section 2 describes our problem setting. We then consider how to optimize joins when selectivities are known in advance (Section 3, evaluated in Section 4). Section 5 considers more complex joins, and Section 6 shows how to *learn* selectivities and re-optimize for them. Section 7 studies failures. We review related work in Section 8 and conclude in Section 9.

2. SETTING AND PROBLEM STATEMENT

We focus on executing a single windowed join computation (selected by the federated optimizer) over data streaming from a plethora of wireless sensor devices. We seek to be platform agnostic, developing a solution that can scale down to the popular Crossbow mote architecture and up to powerful PC nodes on 802.11 networks. (We assume a reader relatively familiar with ad hoc wireless sensor networks, although we review the basics in Appendix C.)

As in prior work on declarative sensor data management systems, we abstract groups of sensors into conceptual relations, based on sensor types, administrative domains, or other similar criteria. We adopt the *windowed* join [3] model of computation over streaming data: we are given an operation $S \bowtie_{\theta} T$, where S and T represent two (possibly overlapping) collections of sensors and θ represents a predicate over the (scalar) attributes of pairs of tuples ($s \in S, t \in T$). As sensors sample new readings, they send these readings to participate in the join. In a “push”-based manner, the join buffers new tuples arriving from S and joins them with buffered tuples from T , and vice versa. The join query typically specifies a time or size window over each source stream, which defines a bound on the size of the buffer for each source: each newly arriving tuple is to be joined against the contents of the opposite buffer (each s tuple with the buffered T tuples, and vice versa). We assume relations are *partitioned* into sets of independent windows based on grouping attributes — the query maintains the last k samples, or k time units’ samples, for *each* partition key value — to avoid the need for global window coordination across nodes.

If *all* attributes are dynamic in the system, then the only feasible strategy is to perform a join at the base station: no consistent pruning mechanism exists. Fortunately, many attributes in a sensor network are actually *static*: e.g. node IDs, coordinates, or other

types of identifiers (name, group ID, capabilities) After converting the query into **conjunctive normal form**, we pre-evaluate clauses that refer exclusively to static attributes. Pre-evaluating a selection clause determines each node’s eligibility to participate in the query; pre-evaluating a join clause establishes that a node might join with others in any given cycle (depending on the dynamic attributes).

Query processing in a sensor network consists of four main tasks: (1) find promising paths for computing the join; (2) retain the best paths and place join nodes along them using a cost model; (3) ensure that all relevant source and join nodes are informed of the decision; (4) begin execution, i.e., sampling data and computing join results. We now describe how these steps are optimized.

2.1 Join Optimization Problem

Depending on the type of sensor network, different optimization goals might be formulated: latency, network congestion, energy consumption, or some combination thereof. While our optimization methods are **agnostic to the cost model**, we implemented a specific cost model instance, whose benefits we experimentally demonstrate for both battery and AC-powered devices: we focus on reducing *overall traffic* and *congestion or hot spots*. Optimizing these provides secondary benefits in reducing energy consumption and traffic at the most stressed nodes, and it reduces overall latency.

Distributed database systems use extensions of System-R’s dynamic programming algorithm to determine an efficient join strategy. They assume each table or table fragment has (approximately) uniform access cost, and only a few fragments exist for each table. The optimizer chooses an order of evaluation (and shipment) among a set of joins, while picking a specific algorithm to compute each join expression. Being centralized, the optimizer has enough memory to use dynamic programming (or memorization) to explore alternative join strategies, and it has reliable access cost and data distribution information.

In our context, those assumptions do not hold. Data (with characteristics often not known in advance) is partitioned across a multi-hop network, making the cost of join dependent on its distribution, network topology, and number of alternate routes between nodes. Depending on the substrate, even communication cost to different nodes is non-uniform: in a tree-based sensor network routing substrate, it is typically less expensive to route to the root node than to another equally distant node¹.

Goals. We have several desiderata for our join optimization solution: (1) *locality awareness*, exploiting variations in data distributions, (2) *decentralized* query optimization, with self-optimizing nodes not relying on the base station to perform the join computation², (3) *adaptivity*, tracking changes in data distributions over time, (4) *scaling up* to hundreds of nodes, matching the expectation that some sensor network deployments might be dense — yet *scaling down* to nodes with 10s of KB of RAM (as in a mote device), and (5) *balance* between minimizing network transmissions and path lengths on one hand, and avoiding significant congestion in any single area of the network on the other.

Challenges. Join optimization in a sensor network is limited by resource constraints, distributed knowledge, and dynamicity. In general, it is extremely expensive to acquire full connectivity information, even at the base. Instead each node has limited knowledge about its surrounding network (nodes within one radio hop) and its position in the primary routing tree (depth, parent and children). It can route to its parent or children. Some sensor networks also sup-

¹Every node knows its parent, so messages do not need to specify a path to the root, as they would for any other destination.

²This results in less flooding of the root, less latency in making decisions, and less overall state transmission in the system.

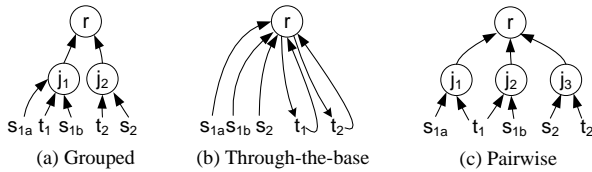


Figure 1: Different classes of join algorithms.

port routing to a destination node at a position (e.g., GPSR [13]), or to nodes holding a static value (*BestRoute* [11], which indices values at the network nodes). Once destination nodes are located, they may return a message back to the initiator containing a path vector, which is used to directly route subsequent messages.

The optimization task consists of determining whether a given node is eligible to participate in any joins with other nodes in the system, and if so, determining a *join strategy* for that node.

2.2 Common Join Strategies

Executing a join requires (i) determining *which pairs* will send data to be joined; (ii) deciding *which paths* data is to be sent along in order to accomplish the join; (iii) deciding *where to buffer* the data and perform the computation, at some *join node*. The join node does the actual windowed join computation and sends results to the base station if necessary.

The join node can be the root node (we term this *join at the base*). Conversely, given a join with at least one static-valued key, suppose join computations are performed asymmetrically: source nodes $s \in S$ “search” for target nodes $t \in T$ that might (in any given sampling cycle) join with them based on this key. Then, the join node is placed along the path connecting the (s, t) pair. We term this strategy as *pairwise*, since each (s, t) pair chooses its own join node. In the case of GHT, which allows for geographic routing to a destination based on its hash value, we can perform *grouped* joins, where all nodes with the same join key use the same join node. (Joining at the base is also a grouped join, with a single group.) Finally, we might send all data from the S nodes “through” the base, from where it is re-routed to the T nodes, which in turn perform join computations and return answers back to the base. We see these options visualized in Figure 1. To make the discussion more concrete, we summarize the actual algorithms for each class that we study in this paper. Appendix D presents detailed cost formulas for each algorithm. In the appendix and the remainder of this section, we refer to any pre-processing work as *initiation* (and its cost in bytes transferred as *initiation cost*), per-cycle processing of dynamic state as *computation* (with bytes transferred as *computation cost*), and overall memory usage as *storage cost*.

Grouped Join: At the Base. This straightforward scheme pushes down selection conditions, then sends all satisfying source tuples to the base station, which performs the join computation. All join operations are grouped and applied at the base. An advantage of this **Naive** algorithm (as we refer to this basic strategy) is the lack of per-query setup (excluding initial routing tree construction). Disadvantages are high memory consumption at the base (potentially a buffer for *every* sensor node), congestion near the base, and high computation cost. One refinement is adding a pre-computation step for static join clauses, thus eliminating source nodes which cannot participate in any joins. **Base** uses this strategy, trading costlier initiation for cheaper computation.

Grouped Join: DHT/GHT. Instead of involving the base, we might spread the grouped join computation across multiple nodes. In a wireless IP-based network, this can be achieved with a distributed hash table (DHT); and in a mote-based network with a geographic hashing table (GHT)³. Both strategies route to a join

node based on the hash value of the join key. For DHT this node is the one with hashed IP value closest to the hashed key; for GHT, it is the node with location closest to the key. A GHT-based strategy places all computation for a given key at the same node whose placement (and therefore query execution cost) is unpredictable, as it may be arbitrarily distant from the source nodes.

Through-the-Base Join. The algorithm of Yang+07 [16] reduces storage cost at the base station by sending data from the source nodes *through* the base, and back down to the destination nodes. Those nodes perform the join and return data to the base. This strategy often has higher computation cost than joining at the base.

Pairwise Join: Innet. In recent work [11] we proposed to place join nodes anywhere on a path between the source nodes. The ideal location of the join point depends on selectivities, as we show in this paper. In [11] we provide details of how routing is achieved for the **Innet** algorithm using a combination of multiple trees that share the same source nodes. To summarize, we construct the initial network using the standard routing tree construction algorithm of [10]. To create successive trees, we choose a new root node *furthest from any existing roots*, then build the new tree using the same algorithm. During tree construction, certain attribute information (particularly join keys) is indexed using summary structures like Bloom filters or R-trees in the trees’ routing tables. Routing employs several techniques to reduce message traffic and avoid cyclic paths: it employs an extension of semantic routing trees (supporting Bloom filters and multidimensional R-trees over data); it emphasizes exploring from a node *down* its subtrees, but for completeness also searches *up* each subtree. A search *ascending* a subtree can then search *downwards* from each node, but never go upwards again. After finding paths between nodes satisfying all static predicates, join nodes are placed and the computation begins.

3. BASIC OPTIMIZATION STRATEGIES

In this section we describe a scheme for performing join optimization in a sensor network. For now, we develop a model under the assumptions that (1) selectivities are uniform across the network, (2) they are constant and known in advance to the optimizer, and (3) each sensor node from relation S joins with at most one node from T (for simplicity we will denote this a 1:1 join, even though some sensors may not participate in the join). We relax all of these assumptions later. Our optimization method works for a single join between relations S and T . We refer to a node $s \in S$ as a *producer*, and likewise for $t \in T$.

When Aspen receives a query, it converts it to CNF and disseminates it to all nodes. Then pre-computation and network exploration are performed in order to minimize the per-cycle cost of query execution. Each node computes all selection predicates exclusively referring to static attributes, and any is unsatisfied the node will not generate data for the query. Next, static *join predicates* are computed: exploration discovers pairs of nodes with attributes satisfying the predicates, and communication paths are established between the nodes. Matching our discussion above, exploration can follow several approaches:

1. **Grouped:** sensors from both S and T send to a common node based on their join key, relying either on GHT, or on routing to the base station.
2. **Through-the-base:** sensors from S route to the base station, which then routes to T nodes based on the join key.
3. **Pairwise:** sensors from S route in a multicast style to find T nodes with a matching join key.

Each of these strategies results in a series of unicast or multicast message transmissions. When an exploration message is forwarded, a *path vector* may be used to record visited nodes. When

³Unlike other substrates GHT requires geographical information.

the target is reached, the path vector can be reversed and a response to the source sent without further exploration. (We assume symmetric communication links.) While in tree-structured networks the target is reachable along only one path, if using full connectivity graphs (as in GHT) or multiple trees (as in [11]) the target may receive *multiple* messages from the same source, along different paths. At this point (except for the through-the-base or join-at-base strategies), our query optimizer:

- Determines *which paths* to use.
- Determines where to place join nodes along the paths, given the restrictions of the routing substrate.
- Checks if joining at the base station is cheaper.

The first two steps are broadly analogous to the dynamic programming algorithm used in a conventional optimizer, while the last corresponds to join algorithm selection. Next we describe the process of cost-based selection of paths and placement of join nodes.

3.1 Join Cost Estimate for Window Size w

In the case of **Innet**, as the network is explored, not only are paths between (s, t) pairs found, but additional information is recorded for optimizing the join node placement. For each node j on the prospective path P from s to t , an integer h denoting j 's number of hops to the base station (the resulting array is delta encoded for compression). Let $D_{n_1 n_2}$ be the number of hops between nodes n_1 and n_2 . For **Innet** we can write a cost expression for a pair of s and t nodes and any j on P as: $\sigma_s D_{sj} + \sigma_t D_{tj} + (\sigma_s + \sigma_t) w \sigma_{st} D_{jr}$

This considers the likelihood that each producer sends data to j (with rates σ_s and σ_t respectively), and the likelihood that this data produces join results to be forwarded to the base station. Data from either s or t , in expectation, produces $w \sigma_{st}$ result tuples when joining in the window maintained by j . In addition to considering j 's on P , t also considers performing the pairwise join at the base station. In this case the cost is computed as $\sigma_s D_{sr} + \sigma_t D_{tr}$.

For the through-the-base, the cost differs slightly because messages are forwarded from each s node through the root to a set of t nodes, regardless of t 's selectivity: $\sigma_s D_{sr} + (\sigma_s + (\sigma_s + \sigma_t) w \sigma_{st}) D_{tr}$.

3.2 Join Algorithm Selection

Based on a comparison of the cost estimates — both for different join node placements and different available algorithms — node t should choose the **best scheme** among the alternatives. In practice, it is straightforward to take either **GHT** or **Innet** and compare versus joining at or through the base: every node knows its distance from the base and can estimate the cost to forward there. However, **GHT** and **Innet** are not comparable, as **GHT** is a grouped strategy.

Once the t node chooses a join node j , it sends a *nomination message* to j , containing the triple (sourceID, targetID, sequence). Node j in turn notifies s that it will be performing this pairwise (s, t) join.⁴ Due to the explicit minimization, our strategy is **never more expensive than joining at the base station**, provided we use the same initiation strategy to discover the joining pairs. This claim is true because the pairs are independent. We do not claim that this scheme can match full global coordination, which can consider, e.g., how the results of several joins might be merged into the same packets. (However, if the selectivities are real numbers, such a global solution encodes NP-Hard Knapsack.) After finishing optimization, query execution is quite straightforward. The designated join node buffers source tuples from nodes it joins — maintaining a join window of size w .⁵ New tuples are enqueued into the window (evicting expired ones) and joined the other relation's windows.

⁴While better paths are discovered, t continues to nominate new join nodes for the (s, t) pair, until the end of initiation.

⁵Here we assume tuple-based windows. For time-based windows, we maintain space sufficient for the maximum expected data rate.

Table 1: Attributes used in queries

| Static attributes | |
|--------------------|---|
| id : | unique identifier. |
| x : | $[7, 60]$ exponential spatial distrib., center has higher values. |
| y : | $[0, 10]$ uniform random distrib. |
| cid and rid : | column and row numbers in a 4 by 4 grid. |
| pos : | real-life position (256m by 256m grid). |
| Dynamic attributes | |
| u : | $[0, [1/\sigma_{st}]]$ uniform random distrib. |
| v : | real-life humidity. |

Table 2: Queries used in experiments

| |
|--|
| 1:1, Join with random endpoints (Query 0) |
| $(\sigma_{id=random \wedge h_S(u)S}) \bowtie_{S,u=T,u} (\sigma_{id=random \wedge h_T(u)T})$ where $h_P(u)$ is short for $(hash(u) \% [1/\sigma_P] = 0)$. |
| Non-1:1, uniform join endpoints (Query 1) |
| $(\sigma_{id < 25 \wedge h_S(u)S}) \bowtie_{S,x=T,y+5 \wedge S,u=T,u} (\sigma_{id > 50 \wedge h_T(u)T})$ |
| Join at perimeter (Query 2, based on Query P) |
| $(\sigma_{rid=0 \wedge h_S(u)S}) \bowtie_{S,cid=T.cid \wedge S.id \% 4 = T.id \% 4 \wedge S,u=T,u} (\sigma_{rid=3 \wedge h_T(u)T})$ |
| Region-based join (Query 3, based on Query R) |
| $S \bowtie_{D_{st} < 5m \wedge s.id < t.id \wedge abs(s.v - t.v) > 1000} T$ |

4. EVALUATION OF JOIN OPTIMIZATION

The mote implementation amounted to 15,100 lines of nesC, generating 101KB of IRIS code which uses 4.5KB of RAM. We run TOSSIM [9], which models **radio errors and retransmissions**, on a cluster of 20 2.4GHz Core 2 Quad workstations. We support select-project-single join queries with predicates over 16-bit integer attributes (common for most hardware). At a join node j we maintain: (1) a list containing (s, t) pairs to join; (2) paths to producer nodes; (3) window of values from each producer.

4.1 Experimental Workload

We study several network topologies generated with different deployment densities (6, 7, 8 and 13 neighbors on average), and one from the Intel Research-Berkeley dataset.⁶ In the paper we focus on the 7-neighbor and Intel dataset topologies; Appendix C shows the properties of the **Innet** routing substrate for all topologies.

Our query workload considers both spatially correlated and uncorrelated data, with attributes listed in Table 1. Though pos is not required by the **Innet** substrate, it can be used for region-based queries. Specifically, pos allows us to compute the Euclidean distance D_{st} for a (s, t) pair. Regarding u , though our algorithms are not distribution dependent, to simplify analysis we pick integer σ_{st} s.t. for any u_1 and u_2 $Prob[u_1 = u_2] = \sigma_{st}$. Values for attribute v were obtained from the Intel dataset. For x, y, cid, rid and id we build Bloom filter summary structures; and for pos , an R-tree.

Table 2 shows a diverse query workload. For Query 0 no producer joins with more than one other producer (termed a 1:1 join). Query 1 allows producers to join with multiple other producers (termed an $m:n$ join), with producers well distributed throughout the network. Query 2 is an $m:n$ join at the perimeter. Queries 0–2 are run using synthetic data, with given σ_s, σ_t and σ_{st} varied across experiments. Query 3 is region-based join, using real-life data. For Queries 0–2, each producer generates as much as 800 u samples for runs consisting of up to 800 *sampling cycles*. Each sampling cycle itself consists of 100 *transmission cycles*. For Query 3 producers generate 65535 v samples. Experiments are averaged across 9 runs and 95% confidence intervals are provided. For more detail about supported query types and implementation refer to Appendix B.

4.2 Join Algorithm Performance

To show potential for optimization, we experimentally compare the performance of the basic join algorithms — expected to vary considerably as the relative selectivities change. Figures 2 and 3 are

⁶db.csail.mit.edu/labdata/labdata.html

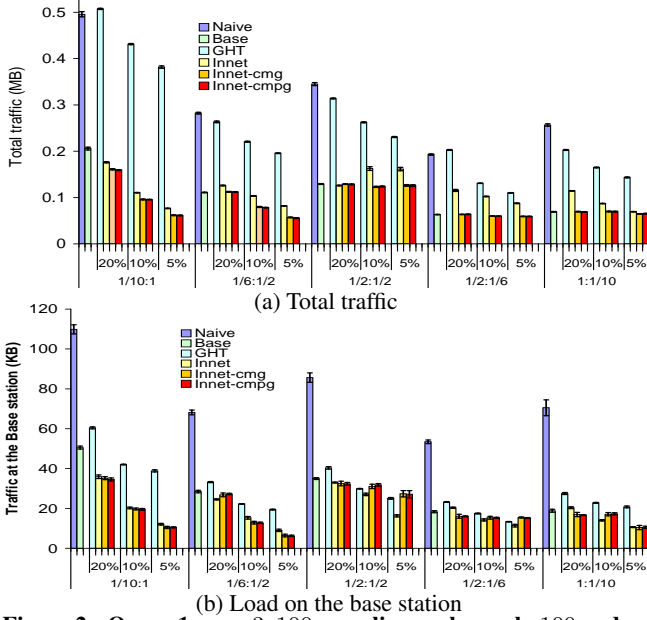


Figure 2: Query 1, $w = 3$, 100 sampling cycles each, 100 nodes

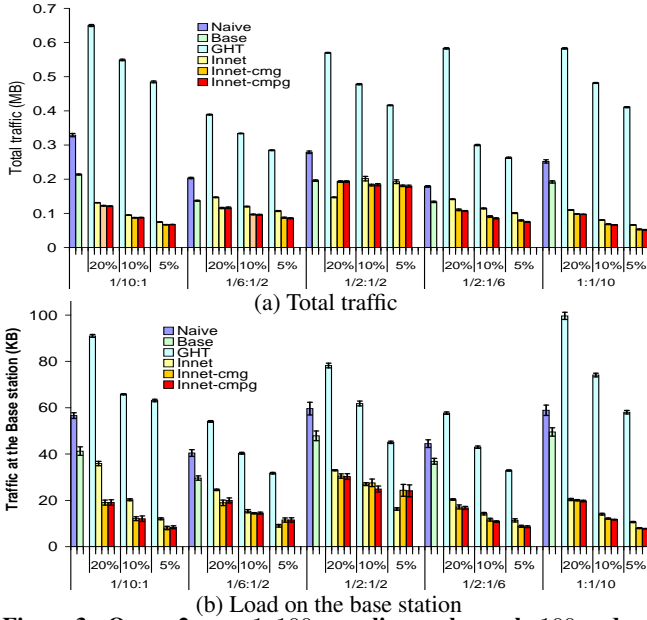


Figure 3: Query 2, $w = 1$, 100 sampling cycles each, 100 nodes

for queries Query 1 and 2, respectively, and are structured as follows. Across the x-axis we vary the relative selectivity ratios $\sigma_s:\sigma_t$ between source relations S and T , in a series of stages: $\frac{1}{10}:1$, $\frac{1}{6}:\frac{1}{2}$, $\frac{1}{2}:\frac{1}{2}$, $\frac{1}{2}:\frac{1}{6}$, $1:\frac{1}{10}$, respectively. For each such ratio we vary the selectivity of the join, σ_{st} . The **Naive** and **Base** algorithms, which are unaffected by join selectivity, are shown as the first two bars. Then we see, for each join selectivity value (20%, 10%, 5%) the relative performance between **GHT** and the **Innet** algorithms. (For now, ignore the two improvements explained in Section 5: **Innet-cmg** and **Innet-cmpg**. We also attempted to compare with **Yang07+**, but for the parameters we tested it did not complete even a single run on the synthetic topologies, as its routing queues overflow almost immediately. Even increasing them by a factor of 10 was not helpful. We were able, however, to compare in Section 6.2.)

Naive incurs high traffic and maximum load (being preferable only for short queries, as it has no initiation cost). **Base** is significantly better — and the most efficient basic algorithm for Query

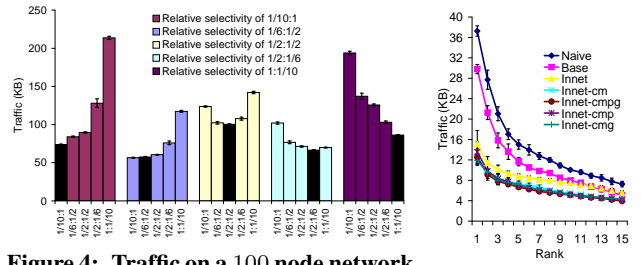


Figure 4: Traffic on a 100 node network, Query 0, $\sigma_{st} = 20\%$, $w = 3$. Data has $\sigma_s:\sigma_t$ selectivities (in dark), but we optimize for different selectivities

Figure 5: Load distribution for the 15 most loaded nodes

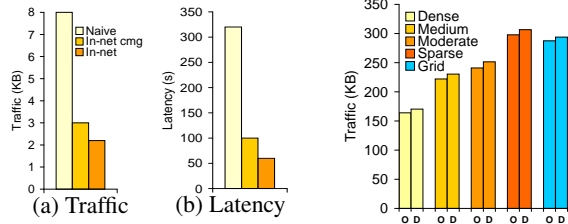


Figure 6: Centralized and distributed initiation

Figure 7: Optimal (O) and distributed (D) computation

1 when σ_s is high (S nodes send the majority of the data) and σ_{st} is high (but worse than the improvements **Innet-cmg** and **-cmpg**). **GHT** always does poorly due to its long routing paths. Finally, **Innet** provides the best performance in all cases of Query 2, and in Query 1 if σ_s is low. For higher σ_s **Base** becomes the better choice. In Figure 5 all strategies exhibit similar load distribution profiles.

An in-depth examination of **Innet** on Query 1 reveals an important drawback of the strategy: the pairwise cost model does not take into account that a single s tuple might be joining with multiple t tuples — and that this computation could be shared if the join node was placed at the base station. This explains why **Base**, a *grouped* strategy, sometimes works better. Motivated by need for further improvements, in Section 5 we describe **Innet-cmg** and **Innet-cmpg**.

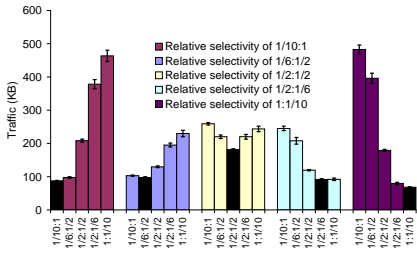
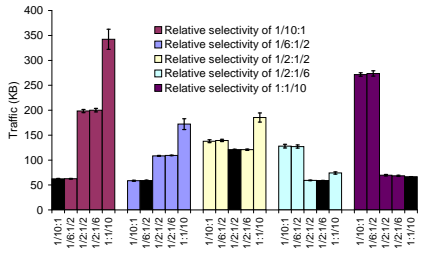
We show experimental validation on mesh networks in Appendix F.

4.3 Centralized vs. Distributed Optimization

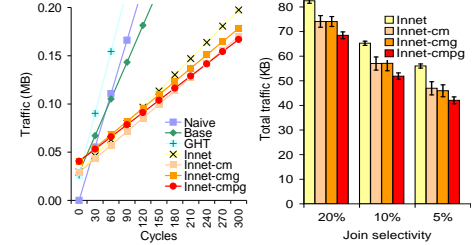
We next show the benefits of our distributed optimization scheme as compared to a centralized one: namely, that centralized optimization causes high congestion near the base when it collects the information (connectivity, static attribute values) it requires. Figure 6(a) shows that our distributed initiation (optimization) scheme is up to 3 times more efficient at the base than centralized optimization, even when ignoring the extra traffic required to distribute the query plan back to the network. Moreover, if we periodically re-optimize to adapt to new conditions (Section 6), this overhead is incurred each time. Importantly, Figure 6(b) shows that the centralized case incurs a latency up to 5 times greater than the decentralized case. Furthermore, Figure 7 shows that the decentralized computation yields traffic levels within 3% of the optimum centralized case, independent of network topology. These results are for a query consisting of 1:1 joins between 10 random pairs of nodes, with $\sigma_s = 1$ and $\sigma_t = \sigma_{st} = 0$.

4.4 Cost Model Performance

Finally, we seek to determine whether the cost model places the join node appropriately, in the absence of shared computation. We study Query 0 in which each S tuple joins with at most one T tuple. In Figure 4, again we group the runs based on relative source selectivity ratios: the **true** ratios of $\sigma_s:\sigma_t$, starting from the left, were $\frac{1}{10}:1$, $\frac{1}{6}:\frac{1}{2}$, $\frac{1}{2}:\frac{1}{2}$, $\frac{1}{2}:\frac{1}{6}$, and $1:\frac{1}{10}$. We ran **Innet** with join nodes opti-



(a) Uniform non1:1: Query 1, $\sigma_{st} = 5\%$, $w = 3$ (b) Perimeter: Query 2, $\sigma_{st} = 10\%$, $w = 1$
Figure 8: Traffic on a 100 node network. Data has $\sigma_s:\sigma_t$ selectivities highlighted in dark, but we optimize for different selectivities



(a) Method vs. duration (b) Traffic, 1000 cycles
Figure 9: Query 2, $w = 1$, comparison of different MPO techniques.

Algorithm 1 GROUPOPT()

- 1: **if** a new pair is found by p **then** recompute cost difference ΔC_p
- 2: **if** ΔC_p has changed, or a new G_c is found as a result of the new pair **then** send ΔC_p to G_c
- 3: **if** G_c has received ΔC_p for each $p \in G$ **then**
- 4: set decision $G_c.D$ to be *innet join* if $\sum_{p \in G} \Delta C_p < 0$ and *join at base* otherwise
- 5: Give it next sequence number $G_c.D.seq$ and send to each p
- 6: **end if**
- 7: At some $p \in G$, let G_c' be the coordinator that sent $G_c.D$.
- 8: **if** $G_c'.id < G_c.id$ or $(G_c'.id = G_c.id$ and $G_c'.D.seq > G_c.D.seq)$ **then** accept $G_c'.D$ and set $G_c = G_c'$

mized for each of the different selectivities. Our algorithm should provide the best performance when it is given the true selectivities, i.e., the dark bar will be the lowest in each group, and this is the case. **Innet** did not do as well for the other queries, because of shared computation. We conclude that the pairwise algorithm, **Innet**, with optimization techniques used to choose a join node, performs well when multiple tuples do not mutually join. Grouped techniques such as **Base** work better when there is more sharing.

5. MULTI-JOIN-PAIR OPTIMIZATION (MPO)

In this section we develop a set of techniques that choose between a pairwise algorithm and grouped algorithm, using a fully distributed strategy that can make a different choice for each set of join keys. We focus specifically on the **Innet** and **Base** algorithms, respectively, as good-performing exemplars of the two classes.

5.1 Network-Level Resource Sharing

We implement several techniques in our query engine to make multi-pair computation more efficient. These are described in detail in Appendix E, but we briefly summarize two major features. For each producer p we build a *multicast tree* T rooted, using paths established between p and other producer nodes with which p joins. The multicast tree maintains state at each internal node in the tree, enabling packet transmissions’ path vectors to be compressed. Additionally, we implement a *path collapse* feature as follows. Suppose producer p sends data values to two join nodes, j_1 and j_2 , using two node-disjoint (except for p) paths, $P_1: [p \dots n_1 \dots j_1]$ and $P_2: [p \dots n_2 \dots j_2]$. If for some pair of nodes (n_1, n_2) there exists a link between n_1 and n_2 , the two paths can be *collapsed* into a multicast tree that is rooted at p , passes through n_1 and n_2 , and has j_1 and j_2 as leaf nodes. We let nodes along P_1 and P_2 opportunistically *snoop* on messages traveling on neighboring paths, and they notify p if an optimization opportunity is discovered.

5.2 Group-based Optimization

We focus on join queries where the predicates are commutative and transitive: if we put all $s \in S$ nodes on one side of a bipartite graph and all $t \in T$ nodes on the other, and add edges between each (s, t) pair that joins, then each node will be part of a *complete bipartite subgraph* (i.e., if s_1 joins with t_1 and t_2 , and t_2 joins with

s_2 , then t_1 also joins with s_2). An example of this is an equijoin. We term each complete bipartite subgraph a *group*, and separately determine for each group whether it should compute a series of pairwise joins, or a single grouped join (at the base station).

For each group of participating producer nodes G , let us designate a unique *group coordinator* G_c to be the node with the smallest ID in G . We now rewrite the traffic cost expression in a relative way for each producer p (from S or T) that participates in the group, as a *difference* between performing a fully in-network computation, and computation at the base station (N_{pj} represents the number of pairs that join node j is handling between p and other producers):

$$\Delta C_p = \sigma_p \sum_j (D_{pj} + w \sigma_{st} N_{pj} D_{jr}) - \sigma_p D_{pr}$$

Each producer p sends its own cost difference ΔC_p to G_c . Based on $\sum_{p \in G} \Delta C_p$, G_c determines whether for G as a whole it is cheaper to perform a fully in-network join, or a join at the base; it notifies all nodes in G about its decision D . Thus, at the expense of a slightly higher optimization cost, we can achieve a lower computation cost than in the case of a straightforward application of the pairwise cost model. We point out that in our algorithm each group arrives at a join strategy independently of the others: there is no flow of data values between groups. Also, the algorithm is decentralized: each group elects its own G_c . Figure 1 shows pseudocode for the algorithm. Most of its complexity lies in managing consistency with respect to G_c and decisions. With minor changes, the algorithm can also be used to handle symmetric and transitive predicates, with joining nodes forming a *complete graph*.

5.3 Performance of Multi-Pair Optimization.

We revisit Figures 2 and 3 for Queries 1 and 2 results for **Innet-cmg** (with added multicasting and group optimization) and **Innet-cmpg** (which also adds path collapsing, as this technique was particularly useful in combination with group optimization). The MPO techniques match or beat the standard **Base** and **Innet** (and best all of the other algorithms). **Innet-cmpg** is never worse than **Innet-cmg** and for Query 2 it gives slight improvement.

Cost Model Validation and Comparisons. Section 4.4 showed the basic optimization strategy was appropriate for Query 0, but not for the other queries. Figure 8, however, shows the MPO strategy making good decisions. For each group of selectivities, we ran **Innet-cmpg** for 5 sets of actual selectivities. MPO uses correct selectivity estimates to generate better plans. Interestingly, even **ballpark estimates** give reasonable performance, whereas very inaccurate estimates can be expensive, making the strategy both **useful** and **robust**.

Breakdown of MPO Contributions. Figure 9(a) compares algorithm performance against running time, measured in sampling cycles. Figure 9(b) varies join selectivity from 5% to 20%. For queries running for 30 sampling cycles or more, **Naive** is not competitive in traffic (and never in maximum load). **Innet-cm** is the best choice for queries 30-240 cycles in duration, and for even longer queries, **Innet-cmpg** achieves up to 25% additional gain.

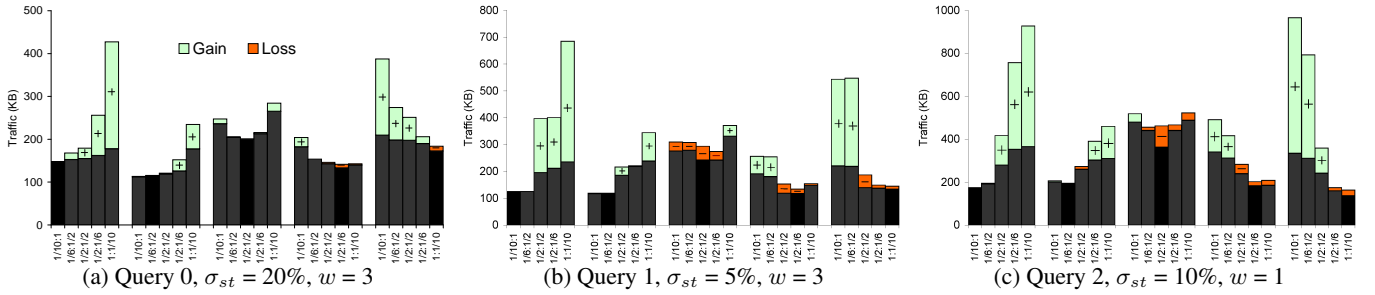


Figure 10: Traffic when data has selectivities σ_s, σ_t highlighted in black. Effect of learning is shown as + (gain) and - (loss)

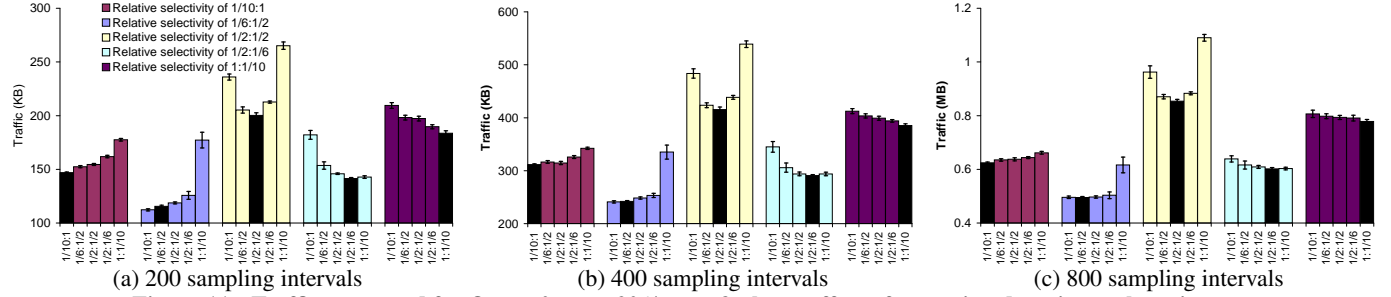


Figure 11: Traffic generated for Query 0 $\sigma_{st} = 20\%$, $w = 3$, shows effect of execution duration on learning

6. ADAPTIVE RE-OPTIMIZATION

So far we have assumed knowledge of the correct values of σ_s , σ_t and σ_{st} (except when testing the performance of the cost model), and their uniformity and stability across producer nodes. We now relax those assumptions. Our cost-based optimization mechanism assigns a join node during *join initiation* for every pair of producer nodes (s, t) . We extend this mechanism to trigger a *new join node assignment* for a given pair when we detect that cost model parameters have changed significantly, in order to keep computation running optimally. If a change of join node occurs, the tuples in the old join window are transferred to the one in the new join node, resuming query computation seamlessly without loss of results.

σ_s , σ_t and σ_{st} can be determined at a join node j , which tracks the number of tuples N_s and N_t received from every s and t node it handles, along with the number of join results N_{st} produced for the (s, t) pair. According to a pre-specified time interval, j re-estimates σ_{st} ; for every tuple from either producer $w\sigma_{st}$ results are generated: $\sigma_{st} = N_{st}/(w(N_s + N_t))$. Producer selectivities are determined using the number of values received: $\sigma_p = N_p/T$, where T is the number of sampling cycles since j became the join node for the producer pair. We trigger a new join node placement when the current parameter estimates diverge by more than 33% from their previous values. (We experimentally found this ratio was a good compromise between maintaining near-optimal execution and low adaptivity overhead.) We also trigger MPO by recomputing the traffic cost expression ΔC_p . N_s , N_t , N_{st} and T are periodically reset to 0 to allow learning within a local time span.

6.1 Validation on Synthetic Data

Learning the Correct Selectivity. Here we start executing with wrong estimates, which do not change for the entire execution run. Figure 10 repeats the experiment of Figure 8 using **Innet-*cmpg*** for Queries 0–2, but for 200 sampling cycles, with and without learning. The groups of bars, as before, represent actual selectivities; the individual bars, provided estimates. The upper segment of each bar indicates the difference between running **Innet-*cmpg*** with learning turned off or on. For example, for Query 1 the fifth column of the first group shows that when we initially optimize for $\sigma_s:\sigma_t = 1:\frac{1}{10}$ and run with actual ratio of $\frac{1}{10}:1$, we can reduce the computation traffic from 685KB to 235KB if we learn the selectivities (upper

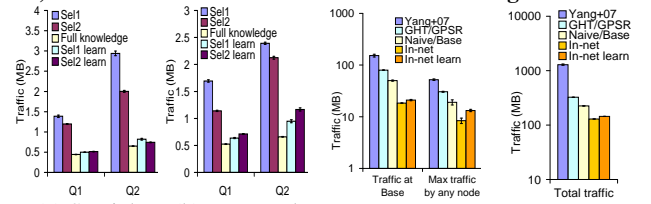


Figure 12: Spatial and temporal learning: traffic at base, total and per node. Note the log scale.

bar marked with '+'/green). Occasionally, when the provided estimates are correct, we experience slightly higher traffic caused by the learning overhead ('-/red). Under incorrect initial selectivities, however, we always observe large gains. Figure 11 shows that as we increase the number of sampling cycles from 200 to 800, the performance under incorrect initial estimates approaches the performance under correct estimates, thus largely removing the need to know the correct selectivities ahead of time for sufficiently long queries. The $\frac{1}{2}:\frac{1}{2}$ result is interesting because given two paths of equal length, unless the distances to the base station vary, there is no difference in the analytical cost model expression for their traffic. The learning algorithm quickly settles to a local optimum: some paths which were rejected during join initiation due to erroneous cost estimates may have provided better solutions. The limited amount of state we maintain does not allow us to remember all paths for a given pair of joining nodes.

Adjusting to Skewed Data and Correlated Predicates. We next consider the case where every producer node can have a different selectivity. Figure 12(a) shows an experiment where half of the nodes generate values according to *Sel1*: $\sigma_s = 10\%$, $\sigma_t = 100\%$ and $\sigma_{st} = 5\%$ and the other half under *Sel2*: $\sigma_s = 100\%$, $\sigma_t = 10\%$ and $\sigma_{st} = 20\%$. Computation proceeds for 800 sampling cycles. Columns **Sel1** and **Sel2** show traffic when we initially optimize for all nodes using *either* Sel1 or Sel2, respectively. Column **Full knowledge** shows an oracle aware of the actual selectivities at each node. Columns **Sel1 learn** and **Sel2 learn** show performing initialization for all nodes with Sel1 and Sel2 respectively — but learning each node's correct selectivities during computation. The learning schemes approach the oracle, reducing traffic by up to 70%.

Adapting to Changing Selectivities. Finally, we study behav-

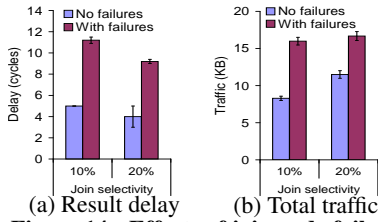


Figure 14: Effects of join node failure

ior when we start executing with the right selectivities, but those change in the middle of the run. Figure 12(b) shows an experiment in which for the first 400 sampling cycles the query computation proceeds according to Sel1 and for the second 400 — under Sel2. For column Sel1 we perform query initiation using estimates for Sel1, with no foreknowledge that those will change halfway through the execution run. (Similarly for Sel2.) Column Full knowledge shows the strategy of an oracle anticipating the change. For Sel1 learn and Sel2 learn, we start identically to Sel1 and Sel2, respectively, but we learn and adjust to the correct selectivities. As in the previous experiment, with learning we can approach the oracle’s performance, gaining as much as 50%.

6.2 Learning and Real-Life Data

Figure 13 shows the performance of the learning model on real-life data, running query Query 3 initially optimized for $\sigma_s = \sigma_t = 100\%$ and $\sigma_{st} = 100\%$. During initiation, those parameters caused all join nodes to be placed at the base station, executing identically to Naive/Base. As join computation proceeded and selectivity estimates became available, the join nodes were gradually transferred from the base station to in-network nodes, and the computation proceeded in a fashion identical to Innet full knowledge (itself running with correct parameters $\sigma_s = \sigma_t = 100\%$ and $\sigma_{st} = 20\%$). We get within 10% of the total traffic of Innet full knowledge, retaining similar base station and maximum node load, making Innet learn the most attractive strategy due to its load profiles and its ability to dynamically adapt.

7. NODE FAILURE

Trivial failures such as intermittently dropped packets are easily by our communication layer. For permanent failures (e.g., signal obstruction, battery depletion, or node crash), we provide best-effort recovery, avoiding data loss whenever possible, with minimal loss of performance. Node failures are repaired transparently using a limited-exploration repair strategy described in [11]. If failure occurs when trying to reach a join node and the repair fails, then the producer switches to joining at the base station (forwarding its last w tuples, enabling the base to reconstruct the join window). If the base station is unreachable, the node will wait for the routing trees to be rebuilt. We experimentally study a simple query consisting of only one join pair, with $\sigma_{st} = 10\%$ and 20% . As a baseline, we let the computation proceed without failure, and then we fail the join node at times varying from 45% to 55% into the run, averaging the results. Figure 14 shows the delay increases by only about 6 cycles, and the traffic behaves similarly to joining at the base. We discuss mobility issues in Appendix G.

8. RELATED WORK

Our work is closely related to operator placement in distributed databases [4, 12, 15], though there bandwidth and power constraints with servers are less severe and relations are less fragmented. Our sensor query system resembles that of TinyDB [10] and Cougar [7], but we support join across heterogeneous sensors and interface to a federated query processor. In addition to [16] (discussed previously), join in sensor networks has been considered in specialized settings. Synopsis joins [17] propagate synopses to prune messages

that cannot contribute to the final join answers. The work of [1] assumes one of the sources is a static table rather than a stream. The work of [5] assumes disjoint regions for the source nodes to be joined, then computes a region “on the way to the base station” for nodes that join, then distributes a table snapshot among those nodes; [6] similarly focuses on sources in disjoint regions, but provides cost models. These works do not easily generalize to the type of setting we describe in our introduction. [18] returns “top- k ” answers in a ranked join model.

9. CONCLUSIONS AND FUTURE WORK

We have demonstrated a cost model based optimization scheme for distributed join computation in multi-hop networks. We presented various fully distributed optimization techniques to further extend the performance, scalability and flexibility of the model. We have validated the performance for mesh network deployments. As future work we plan to implement our model on Crossbow IRIS and Imote2 hardware and implement building monitoring applications. We will extend our framework to support multi-join queries, mobile data producers and destinations for join results, frequent changes in node membership, and node failure recovery, while maintaining our current low memory requirements and low in-network traffic.

10. 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] V. Chowdhary and H. Gupta. Communication-efficient implementation of join in sensor networks. In *DASFAA*, 2005.
- [6] A. Coman and M. A. Nascimento. A distributed algorithm for joins in sensor networks. In *SSDBM*, 2007.
- [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. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.
- [9] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Sensys*, 2003.
- [10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.
- [11] S. R. Mihaylov, M. Jacob, Z. G. Ives, and S. Guha. A substrate for in-network sensor data integration. In *DMSN*, August 2008.
- [12] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.
- [13] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensor networks with GHT, a geographic hash table. *Mob. Netw. Appl.*, 8(4), 2003.
- [14] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, Nov. 2001.
- [15] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.
- [16] X. Yang, H.-B. Lim, M. T. Özsu, and K.-L. Tan. In-network execution of monitoring queries in sensor networks. In *SIGMOD*, 2007.
- [17] H. Yu, E.-P. Lim, and J. Zhang. On in-network synopsis join processing for sensor networks. In *MDM*, 2006.
- [18] 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.

APPENDIX A. ASPEN SYSTEM OVERVIEW

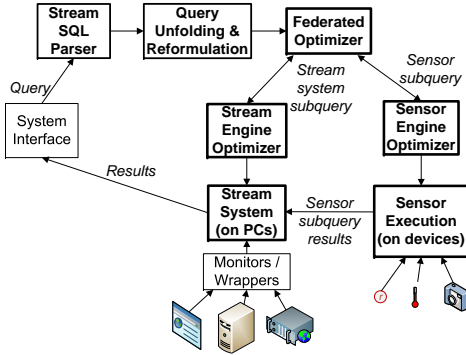


Figure 15: Aspen system architecture

The overall Aspen system consists of a single query and data integration interface over a federated query processor, as shown in Figure 15. Query processing subsystems run on wired Internet hosts (based on a DHT for communication) or on wireless mesh networks (based on our substrate from [11]); and on low-powered wireless devices like Crossbow motes. This paper has focused on the sensor components, and in the appendices of this paper we show that our techniques also scale to wireless mesh networks.

Using a single language and programming model, developers can define *mediated schemas* that can be queried, and *schema mappings* describing how to translate data from sources (sensors, “soft sensors,” and databases) to the mediated schema. A query reformulation component (based on algorithms from [8]) is used to compose the query and the mappings, and the result is fed into the federated query optimizer. Based on a combination of heuristics and cost modeling, the optimizer divides the query plan into components that can be separately executed in the stream or sensor query engines. In turn, each of these engines separately optimizes and executes its portion of the query. Data may be fed from the sensor subsystem to the stream query engine.

B. SUPPORTED QUERY TYPES

Our sensor subsystem supports StreamSQL-style queries consisting of selection and join predicates over two sensor relations. Query 1, for example, is expressed with the following query:

```
SELECT S.id, T.id, S.time
FROM S, T [windowsize=3 sampleinterval=100]
WHERE S.id < 25 AND hash(S.u) % 2 = 0
      AND T.id > 50 AND hash(T.u) % 2 = 0
      AND S.x = T.y + 5 AND S.u = T.u
```

Sensor relations S and T are pre-defined and include a schema with 28 attributes. 18 of the attributes are populated with actual physical sensor measurements (e.g., temperature, light, humidity, battery level, RFID being detected, ADC values) or *soft* readings (e.g., memory available at the mote, local time, etc.). The remaining attributes can be updated by user request from the base station using directed multi-hop flooding (e.g., each mote can be assigned a role, room number, or 3D location). The update rate for physical sensors is specified as part of the query, as are an assortment of other parameters (query start and end times, join window size, etc.)

Selection and join predicates can include not only standard comparisons and Boolean operations, but also the standard arithmetic operators and a handful of utility functions (e.g., hash functions and random value generators). When a query is posed at the base station node, the query preprocessor first separates the predicates in the query into selections and joins. Then, predicates from each

group are separated into static and dynamic subgroups, depending of their attributes being exclusively static or not. Each static join predicate is further fed into a pattern matcher, which, given a collection of summaries built on various static attributes, decides whether the predicate is suitable for content routing using our substrate. In essence, the pattern matcher separates the primary join predicates usable for routing from the remaining secondary join predicates, evaluated after the routing stage has completed. Following this, the parsed query is disseminated in the sensor network.

C. WIRELESS NETWORK SUBSTRATE

We assume a sensor network in which communication is over multiple ad hoc wireless “hops,” which covers a wide range of technologies including citywide mesh networks, many building networks, and most types of ad hoc sensor networks. Query results should be routed to a *base station*, which is powerful enough to process the data. (Typically the base station will be a PC being used either to collect the data or to relay it across the Internet.) We assume that every node knows how far away it is from the base station, and how to route to a neighbor that is one hop closer to the base station. We also assume that there exists a *content-addressable routing substrate* over which messages can be sent based on a *key* attribute. This substrate might be geographic hashing (GHT [13]) over a mote network, a distributed hash table (DHT [14]) over an 802.11 network, or better yet the multi-tree routing substrate we proposed in [11]. GHT and DHTs randomly assign a *single* node to be the destination of a given key, and ignore locality. The multi-tree substrate of [11] is based on a generalization of semantic routing trees [10] and the Generalized Search Tree (GiST) [2]: here we pre-index static attributes at each node, and route to the node(s) holding a particular index value.

Not technically a part of query optimization, the initial construction of sensor routing trees nonetheless plays an important role. Specifically, as described in [11] we assume the ability to do point-to-point (or point-to-multipoint) routing within the network, based on particular attribute values. This is achieved by having *routing tables* at each node, and assigning each node to multiple overlapping trees. For each indexed attribute (which must be static or slowly changing), there is a routing table at every node for each for each tree: this table describes the values of the indexed attribute that exist in the subtree rooted at the current node. Our specific implementation uses a generalization of the semantic routing trees of [10], which can encode 1-dimensional intervals (as in TinyDB) as well as rectangles, Bloom filters, or histograms — each of these structures may be useful for particular datatypes and value ranges.

We also allow for nodes to be *extended* with static attributes during tree construction, e.g., we could add a **floor** attribute to building sensors, and flood the network with a series of messages mapping each sensor node ID to a particular value. These extended attributes become part of the sensor’s schema.

We briefly illustrate the performance characteristics of our substrate in finding paths between nodes that may join: this governs the performance of the join algorithms in the paper. We study several different topologies: random with 6, 7, 8, and 13 average neighbors per node (“sparse random,” “moderate,” “medium,” and “dense random”); and grid with an average of 7 neighbors (“grid”). Experiments were conducted on 3 different sensor layouts for each random topology. See [11] for more details. Figure 16 reproduces the key result from that work: in terms of the path and load performance in a mote network, we perform significantly better than a single-tree routing scheme, or a scheme using hash-based routing to geographical coordinates (GHT/GPSR).

As mentioned previously, we want our algorithm to scale beyond

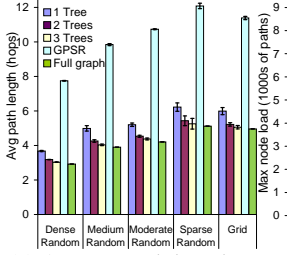


Figure 16: Path quality, 100 node mote network

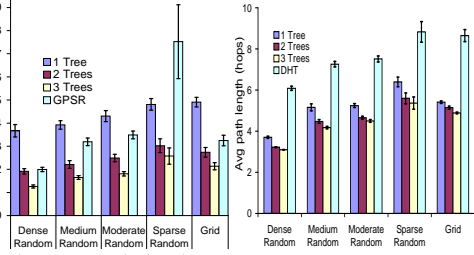


Figure 17: Path quality, 100 node mesh net

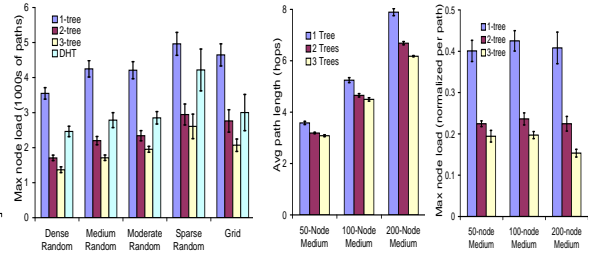


Figure 18: Mesh net scaleup

motes, e.g., to powerful devices over a multi-hop mesh network (e.g., a citywide network). To show that our routing/path-finding algorithm indeed scales appropriately, we developed a Java-based implementation of the same algorithms over a simulated 802.11 mesh network. The overall trends, shown in Figures 17 and 18 look very similar to the mote results, except that the hash-based scheme (based on a distributed hash table [14]) produces slightly better path lengths than GPSR/GHT. This is because the DHT algorithms do not require traversing the boundary of a connectivity gap (see [13]); but as a consequence the maximum load increases. Finally, our results scale nicely, as shown by Figure 18, which examines performance in 50- through 200-node networks. (This second experiment produces similar results when conducted over motes [11].) This establishes the versatility of our communication substrate, and that it produces high-quality paths. In the rest of the paper we consider how to make use of these paths in join optimization and execution.

D. DETAILED COST MODEL

In Table 3 we describe the cost model we use for each join method. Assume a join key k , and that there exist c_s nodes from relation S with k , and c_t nodes from relation T .

Let us assume that costs are defined in terms of the number of tuples sent.⁷ As is standard in cost based query-optimization, our initial cost-based model for joins requires estimates for the rates at which producers send data. Let $s \in S$ and $t \in T$ be a pair of sensor nodes that might join. Let σ_s and σ_t be the probability that s and t will send data for a given sampling interval, thereby defining their production rates (σ_p being the rate of an arbitrary producer p). Define σ_{st} as the probability that two values sent by a pair of producer nodes will join and form a resulting tuple. Intuitively, σ_p is the probability that some (possibly internal) selection predicate over the producer is true, and σ_{st} — that the join predicate is true. Finally, let w be a query-specific window size.

E. NETWORK RESOURCE SHARING

This section describes several network-level techniques we use to improve communication and join computation efficiency.

Multicast Trees: We build a multicast tree T rooted at some producer p using paths established between p and in-network join nodes, where data from p joins with other producer nodes. T is used to efficiently deliver data values to the join nodes. For best performance, we maintain state at each internal node i of T having more than one child. The state encodes T 's subtree rooted at i . In this way the transmission overhead is reduced, as p needs to address only a few i nodes, rather than send encoded representation of T along with every data value. In case T changes, p pushes an updated copy and all i nodes update their state. This technique was especially helpful when $\sigma_s \neq \sigma_t$.

⁷The cost metric can be easily modified to instead consider bytes or packets sent, or to scale by nodes' remaining battery life.

Notation: r = root node (base station); σ_s = probability of $s \in S$ nodes satisfying selection predicates (resp. for σ_t , $t \in T$); $\phi_{s \rightarrow t}$ = selectivity of $s \in S$ that satisfy static selection and pre-filter conditions (similarly $\phi_{t \rightarrow s}$); c_s = number of $s \in S$ nodes with the same join key (resp. for c_t); set of all $j =$ join nodes selected by algorithm; D_{sj} (and D_{tj}) = hops between nodes s (t , resp.) and associated join node; D_{ar} (D_{st}) = hops between node a and root (s and t , resp.). Cost is in the form *contribution of sources + contribution of targets + contribution of join*.

| Naive (grouped at base) | |
|---|---|
| Initiation: | 0 (done by initial routing tree construction) |
| Computation: | $\sigma_s \sum_s D_{sr} + \sigma_t \sum_t D_{tr} + 0$ |
| Storage: | $w(\sigma_s S + \sigma_t T)$ values at base |
| Base (grouped at base) | |
| Initiation: | $2(\sigma_s \sum_s D_{sr} + \sigma_t \sum_t D_{tr})$ |
| Computation: | $\sigma_s \sum_s (\phi_{s \rightarrow t} D_{sr}) + \sigma_t \sum_t (\phi_{t \rightarrow s} D_{tr}) + 0$ |
| Storage: | $w(\sigma_s \phi_{s \rightarrow t} S + \sigma_t \phi_{t \rightarrow s} T)$ values at base |
| Yang+07 (through-the-root) | |
| Initiation: | 0 (done by initial routing tree construction) |
| Computation: | $\sigma_s \sum_s D_{sr} + 0 + (\sigma_s S / T + (\sigma_s + \sigma_t) w \sigma_{st}) \sum_t D_{tr}$ |
| Storage: | $ S $ values at base |
| GHT (grouped by key at join node $j = \text{hash}(\text{key})$) | |
| Initiation: | $\geq \sigma_s \sum_s D_{sj} + \sigma_t \sum_t D_{tj}$ |
| Computation: | $\sigma_s \sum_s D_{sj} + \sigma_t \sum_t D_{tj} + (\sigma_s + \sigma_t) c_s c_t w \sigma_{st} \sum_j D_{jr}$ |
| Storage: | $c_s c_t w$ per join node |
| In-Net (pairwise along $s \rightarrow t$ path; j chosen using a cost model) | |
| Initiation: | $\geq \sum_s D_{st}$ for each pair of nodes satisfying static selection and pre-filtering conditions |
| Computation: | $\sigma_s \sum_s D_{sj} + \sigma_t \sum_t D_{tj} + (\sigma_s + \sigma_t) c_s c_t w \sigma_{st} \sum_j D_{jr}$ |
| Storage: | $c_s c_t w$ per join node |

Table 3: Join algorithm costs, assuming uniform value distribution and spatial distribution, and independent predicates

It is notable that creating an optimal multicast tree by itself is a challenging problem.

THEOREM 1. *If each node has an arbitrary list of neighbors, then choosing the number of broadcasts to multicast a tuple from one source to a subset of nodes is as hard as the set cover problem.*

The intuition of the proof is as follows. In a distributed setting, a node cannot assume specific information other than what it discovers from its neighbors. This forces us to not assume any property of the graph, making the problem NP-hard. We omit the detailed proof of the reduction; but given a set-cover instance we represent it as a bipartite graph where each vertex L on the left corresponds to a given set and each element corresponds to a vertex on the right. A set A is a neighbor of an element a iff $a \in A$. There are no other edges. We add a source s which has all the set-vertices as a neighbor. Note that the multicast from s to all the elements would require a subset of L to relay the tuples and this subset must define a set cover. To see the context in case of joins, suppose that $s \in S$ and all the elements define R . If the rate at which s produces a tuple is less than $1/2$ the rate at which any of the elements produce a tuple; and the join selectivity is ≈ 0 (for example if the query is an intrusion detection query which is usually false) it is straightforward to see that the number of tuples sent/relayed is exactly the size of a set

Algorithm 2 PATHCOLLAPSEDETECT(*This*, *Nbr*, *Next*, *Src*, *Dest*, *PathV*)

Input

This: current node, where the algorithm is executed.
Nbr: neighboring node whose data message is being snooped.
Next: the node to which *Nbr* is sending.
Src: source node where the data message originated.
Dest: destination node, where the message is ultimately headed.
PathV: path vector used to forward the message. (might only contain portion of path from *Nbr* to *Dest*)

Output

T: tuple describing optimization opportunity.

```

1: if This ≠ Next and This ≠ Dest and Dest ≠ Base station
   and FlowExists(Src, Dest, Nbr) = FALSE then
2:   for all entries F in FlowBuffer do
3:     if F.Dest ≠ Base station and F.Prev ≠ This and
       F.Prev ≠ Nbr and F.Next ≠ Nbr then
4:       if F.Src = Src and IsNeighbor(Next) = FALSE and
         F.Dest < Dest then
5:         Set tuple T = (This, Nbr, F.Dest, Dest, {})
6:         if T does not occur in PathCollapseBuffer then
7:           Add T to PathCollapseBuffer
8:           Send T to producer F.Src
9:         end if
10:      else if F.Dest = Dest and |PathV| > 1 and F.hops >
        |PathV| and F.Next ≠ This and F.Src > Src then
11:        Set tuple T = (This, Nbr, Dest, Dest, PathV)
12:        if T does not occur in PathCollapseBuffer then
13:          Add T to PathCollapseBuffer
14:          Send T to producer F.Src
15:        end if
16:      end if
17:    end if
18:  end for
19: end if

```

cover. Thus minimizing the number of tuples sent corresponds to minimizing the set-cover which is NP-hard and has no constant factor approximation. This serves to justify our decision to implement a lightweight heuristic based construction for multicast trees.

Path collapsing: Suppose producer p sends data values to two join nodes, j_1 and j_2 , using two node-disjoint (except for p) paths, $P_1: [p \dots n_1 \dots j_1]$ and $P_2: [p \dots n_2 \dots j_2]$. If for some pair of nodes (n_1, n_2) there exists a link between n_1 and n_2 , the two paths can be *collapsed* into a multicast tree that is rooted at p , passes through n_1 and n_2 in some order, and has j_1 and j_2 as leaf nodes. The relative ordering of n_1 versus n_2 depends on their distances to p , as well as the other path lengths. To achieve this optimization, nodes along P_1 and P_2 opportunistically *snoop* on data value messages on neighboring paths, and if they discover an optimization opportunity, they notify p . This can also be achieved by periodic *gossip* messages broadcast by the nodes on the path. p considers all possible optimizations received and computes the best multicast tree. We also perform a similar optimization for paths which share the same destination join node, instead of the same producer node. In this case, we aim to improve the probability of two data values traversing the same intermediate nodes and be combined into one physical packet using the opportunistic merging described below.

Algorithms 2 and 3 show implementation details related to the path collapsing scheme. *PathCollapseDetect* is being executed by every node in the network upon the interception of any data message. If a tuple T (encoding an optimization) is generated, it is sent to the respective producer node, which in turn executes *PathCollapseApply* to make necessary modifications to paths to join nodes and potentially to the multicast tree used to send data values to

Algorithm 3 PATHCOLLAPSEAPPLY(*This*, N_1 , N_2 , $Dest_1$, $Dest_2$, *PathV*)

Input

This: producer receiving optimization opportunity.
 N_1, N_2 : nodes between which a link exists.
 $Dest_1$: destination of path on which N_1 is located.
 $Dest_2$: destination of path on which N_2 is located.
PathV: path segment from to N_2 to $Dest_2$. (might be empty)

```

1: Set Swapped to FALSE
2: repeat
3:   repeat
4:     Set Changed to FALSE
5:     Save current paths to all join nodes
6:     if exists a join node  $J_1$  s.t.  $Dest_1$  appears after  $N_1$  on the path
        $P_1$  to  $J_1$  and  $N_2$  is not on  $P_1$  then
7:       if |PathV| = 0 then
8:         if exists a join node  $J_2$  s.t.  $Dest_2$  appears after  $N_2$  on the
           path  $P_2$  to  $J_2$  and  $N_1$  is not on  $P_2$  then
9:           Update path to  $J_1$  by concatenating segment from This
             to  $N_2$  and segment from  $N_1$  to  $J_1$ 
10:          Set Changed to TRUE
11:         end if
12:       else
13:         Update path to  $J_1$  by concatenating segment from This
           to  $N_1$  and PathV
14:         Set Changed to TRUE
15:       end if
16:     end if
17:   if Changed then
18:     Build updated multicast tree  $T_{new}$ 
19:     Compute cost  $C_{new}$  of the updated multicast  $T_{new}$ 
20:     if  $C_{new} < C_{best}$  then
21:       Set  $T_{best} = T_{new}$ 
22:       if  $1.1 * C_{new} < C_{send}$  then
23:         Set  $T_{send} = T_{new}$ 
24:       end if
25:     else
26:       Restore changes made to join nodes
27:     end if
28:   end if
29: until Changed = FALSE
30: if |PathV| = 0 and Swapped = FALSE then
31:   Set Swapped to TRUE
32:   Swap  $N_1$  and  $N_2$ 
33:   Swap  $Dest_1$  and  $Dest_2$ 
34:   continue
35: end if
36: until |PathV| > 0 or Swapped

```

those join nodes.

Since we assume symmetric communication links, any pair of nodes can mutually snoop on their messages and thus *PathCollapseDetect* is written in a way to allow only one of the neighboring nodes to issue an optimization decision. This is achieved with the help of comparison of the unique identifiers of the nodes, such as those in lines 4 and 10. Such tiebreakers allows the algorithms to converge and be communication efficient.

Several data structures are used in the algorithms, some of which are a core part of the implementation of our system. *IsNeighbor(Node)* is a boolean function which returns *TRUE* if the current node can directly communicate with *Node*, and *FALSE* otherwise. *FlowExists(Src, Dest, Next)* is a boolean function which checks the existence of a given *data flow* through the current node (as specified by a source node, destination node and next node on the path). *FlowExists* uses a *data flow buffer* data structure, which is essentially soft state also used for a variety of other purposes, such as opportunistically traversing the network without a path vec-

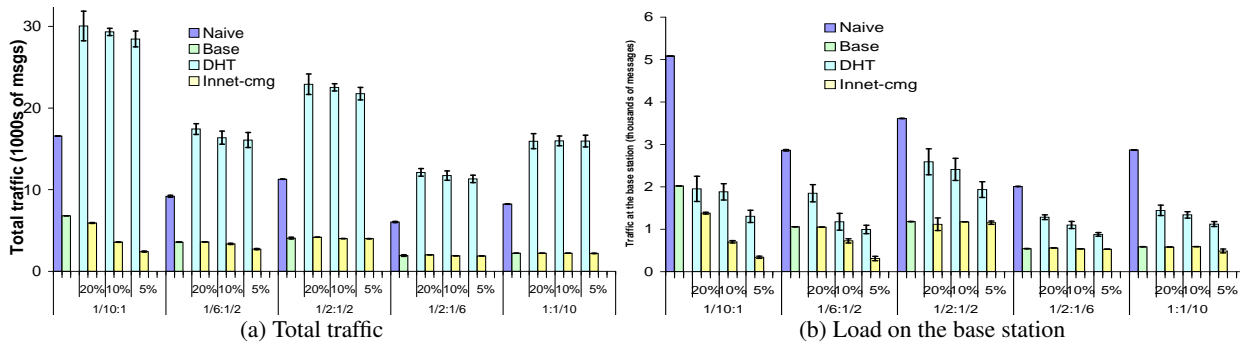


Figure 19: Query 1, $w = 3$ on 100 node mesh networks, 100 sampling cycles, averaged across 9 runs

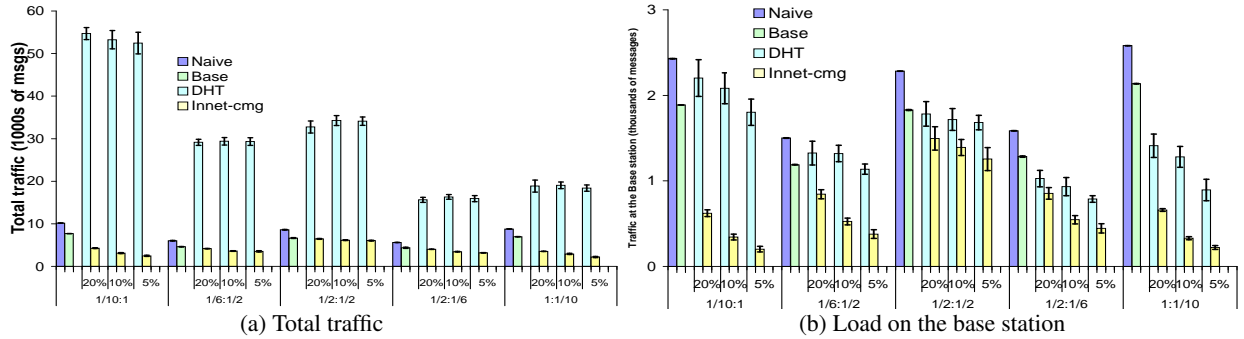


Figure 20: Query 2, $w = 1$ on 100 node mesh networks, 100 sampling cycles, averaged across 9 runs

tor, path repair and load balancing.

If a tuple T is generated, it is sent to the source node without a path vector, using only the data flow buffer. Being part of an opportunistic optimization, T is not always guaranteed to reach the source node. For example, if a given intermediate node handles too many data flows, it might evict some of the older entries from its data flow buffer. In practice however, our routing substrate achieves good load balancing and thus we need no more than five entries in the data flow buffer to achieve significant optimizations. *PathCollapseBuffer* is also a soft state data structure, one exclusively used by *PathCollapseDetect*. Its sole purpose is to avoid sending identical optimizations multiple times to the same producer node whenever possible, thus helping reduce network traffic. If, however, a duplicate optimization is being received, it will have no effect as it will not lead to an improvement of the cost of the multicast tree.

With regard to *PathCollapseApply* it is worth noting that while we do accept optimizations as long as they improve on the cost of the best tree T_{best} found so far ($C_{new} < C_{best}$), we would not necessarily use the new multicast tree T_{new} to send messages until it becomes significantly better (at least 10% lower cost) than the current multicast tree T_{send} we use. In other words, the following must hold: $C_{new} \leq C_{best} \wedge 1.1 * C_{new} \leq C_{send}$. This 10% threshold is desirable because we use *multicast caching*, and thus any update to T_{send} necessitates pushing the updated multicast tree into the network, at an added communication cost. As seen in lines 30 to 33, in the case of common source, the algorithm also tries swapping the two neighboring nodes and re-applying the optimization, doubling the number of explored multicast trees per optimization tuple.

Other opportunistic techniques: We “merge” data values originating from different nodes, but traveling to the same destination node via some common intermediate node n . Node n periodically checks if its buffer of outgoing messages contains data values sharing the same destination, and if so merges them into one physical packet to reduce transmission overhead. This technique is used for data values sent by multiple producers to the same join node or for results sent by join nodes and producers to the base station. We

explicitly do not wait for merges, and no additional propagation delays are introduced. In fact, in some circumstances we may actually reduce the delay due to decreased traffic and congestion around n . This is a generalization of a technique used in TinyDB.

F. MORE POWERFUL NETWORKS

In the main paper, we focused on validating performance on mote networks. However, the goal of Aspen is to “scale up” to more complex devices as well. For multi-hop wireless networks such as mesh networks, short paths and low bandwidth are important just as they are in a mote network — primarily to minimize latency and dropped packets. Figures 19 and 20 show total traffic on a mesh network with exactly the same topologies, source data traces and duration as Figures 2 and 3, respectively. However, we count *messages* rather than *bytes* transferred, since 802.11 link layer and TCP header overhead dominates packet data size. We also do not modify the 802.11 link layer and hence do not perform path collapsing. As in the mote case, the MPO-optimized **Innet-cmg** outperforms all of the other schemes, with **Base** being the next best scheme (versus **DHT** and **Naive**). Relative performance matches closely with the mote results, i.e., our techniques and conclusions **generalize**.

G. MOBILE NODES

While our target domain is sensor and stream nodes embedded within an environment, we studied whether our approach can accommodate a limited number of mobile nodes (e.g., PDAs) moving at a moderate rate. We constrain such nodes to be leaf nodes in the network topology, to avoid significant updates whenever they move. In an experiment we forced a node to move within the medium random topology by picking a new set of parent nodes, and measured the propagation delay in updating all routing trees. On average, 19.4 cycles (seconds in real-time) are needed for the system to completely propagate updates of the summary structures of all the affected nodes. The total amount of network traffic generated was 1195 bytes. If an average node had 10m of radio range, this would support continuous connectivity with a movement rate of approximately 10m per 20 seconds, or 0.5m per second. For faster rates the network will catch up to the object as it slows.