Query Processing in Sensor Networks

Smart sensors are small wireless computing devices that sense information such as light and humidity at extremely high resolutions. A smart sensor query-processing architecture using database technology can facilitate deployment of sensor networks.

Johannes Gehrke
Cornell University

Samuel Madden
MIT

Recent advances in computing technology have led to the production of a new class of computing devices: the wireless, battery-powered, smart sensor. Traditional sensors deployed throughout buildings, labs, and equipment are passive devices that simply modulate a voltage on the basis of some environmental parameter. These new sensors are active, full-fledged computers, capable of not only sampling real-world phenomena but also filtering, sharing, and combining sensor readings with each other and nearby Internet-equipped end points.

Smart-sensor technology enables a broad range of ubiquitous computing applications. Their low cost, small size, and untethered nature lets them sense information at previously unobtainable resolutions. Animal biologists can monitor the movements of hundreds of animals simultaneously, receiving updates of both location and ambient environmental conditions every few seconds. Vineyard owners can place sensors on all their plants to capture an exact picture of how light and moisture levels vary in the microclimates around each vine. Supervisors of manufacturing plants, temperature-controlled storage warehouses, and computer server rooms can monitor each piece of equipment, automatically dispatching repair teams or shutting down problematic equipment in localized areas where temperature spikes or other faults occur.

Despite hardware and domain-specific differences, these deployments share a substantial collection of software functionality: They all collect and periodically transmit information from some set of sensors, and they all must carefully manage limited power and radio bandwidth to ensure that essential information is collected and reported in a timely fashion. To that end, we've designed and implemented an architecture on which we can rapidly develop such data collection applications. Users specify the data they want to collect through simple, declarative queries, and the infrastructure efficiently collects and processes the data within the sensor network. Unlike traditional, embedded-C-based programming models in which each device is treated as a separate computational unit, these queries are high-level statements of logical interests over an entire network. The database system manages data collection and processing details, freeing the user from these concerns. Although other researchers have proposed using database technology to manage networks of smart devices,[1] it's only recently that systems providing this functionality have appeared. At Berkeley and Cornell, we've built two prototype sensor network query processors (SNQPs)—TinyDB (http://telegraph.cs.berkeley.edu/tinydb) and Cougar (http://cougar.cs.cornell.edu)—that run on a variety of sensor platforms.

Published by the IEEE CS and IEEE ComSoc ■ 1536-1268/04/$20.00

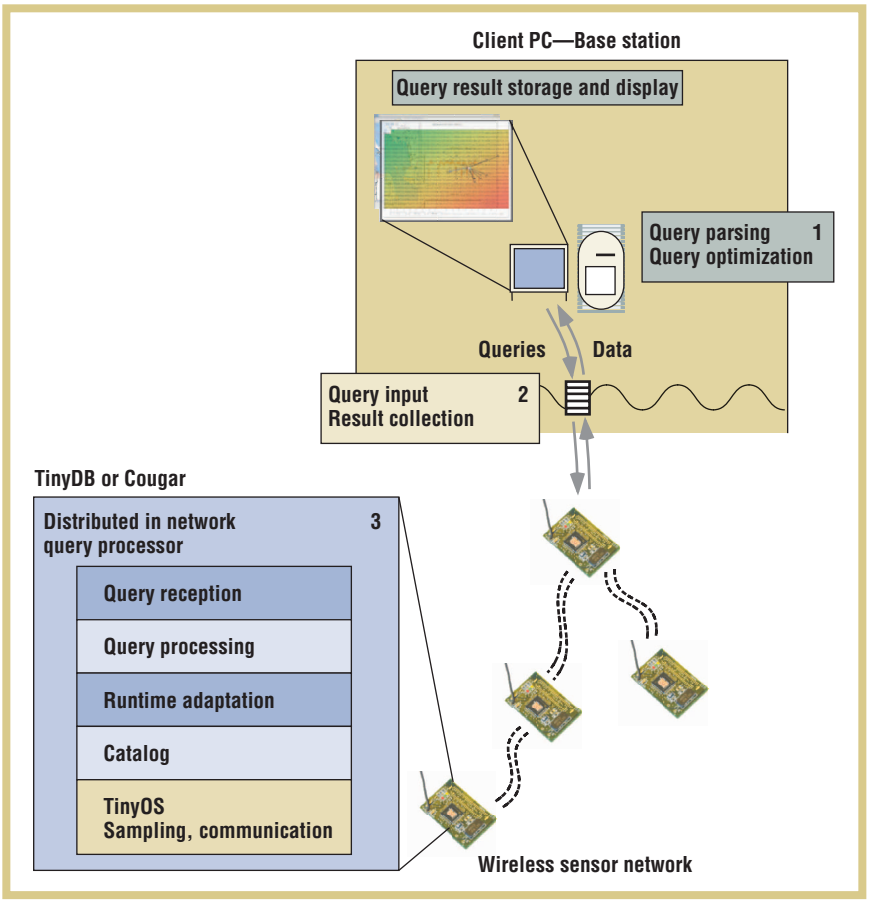


Figure 1. Architecture of a sensor network query processor. Numbers indicate the sequence of steps involved in processing a query.

Our query-processing-based approach can also dramatically improve the energy efficiency—the typical measure of performance in sensor networks—of data-collection applications. This article describes our experiences designing the TinyDB and Cougar query processors. Although we focus on networks composed of homogeneous collections of Mica motes, our work is general enough to be applicable outside this regime. Indeed, we implemented initial versions of Cougar on nodes from Sensoria Corporation running both Windows CE and Linux.

Query processing architecture

Sensor networks provide a surprisingly challenging programming and computing environment: the devices are small and crash-prone, and the operating system running on them provides no benefits (such as fault isolation) to help mitigate such failures. Debugging is usually done via a few LEDs on the device. Programs are highly distributed and must carefully manage energy and radio bandwidth while sharing information and processing.

Because of limitations imposed by this impoverished computing environment, data collection systems in sensor networks must support an unusual set of software requirements. For example,

- They must carefully manage resources, particularly power. Communication and sensing tend to dominate power consumption given the data size and operation complexity feasible on sensor networks. Furthermore, Moore's law suggests that the energy cost per CPU cycle will continue to fall as transistors get smaller and lower voltage, whereas fundamental physical limits and trends in battery technology suggest that the energy to transmit data via radio will continue to be more expensive than the energy density of batteries.
- They must be aware of and manage the transient nature of sensor networks: nodes come and go, signal strengths between devices vary as batteries run low and interference patterns change, but data collection should be interrupted as little as possible.
- They must reduce and summarize data online while providing storage, logging, and auditing facilities for offline analysis. Transmitting all the raw data out of the network in real time is often prohibitively expensive (in terms of energy) or impossible given data collection rates and limited radio bandwidth. Instead, they can provide small summaries or aggregates (such as averages, moments, histograms, or statistical summaries) in real time.
- They must provide an interface substantially simpler than TinyOS's (an operating system especially suited to mote capabilities)[2] embedded-C-based programming model; the interface must also let users collect information and process it in useful ways.
- They must provide users with tools to manage and understand the status of a network of deployed sensors and to easily add nodes with new types of sensors and capabilities.

Each of these points represents a dissertation's worth of research, much of which is incomplete. This article describes at a high level the software and languages we've developed to address these challenges.

Overview

Figure 1 shows a simple block diagram of an architecture for query processing in sensor networks. The architecture has two main parts:

- Server-side software running on the user's PC—the *base station*. In its most basic form, the software parses queries, delivers them into the network, and collects results as they stream out of the network. A more detailed discussion of server-side query processing is available elsewhere.[3]
- Sensor-side software running on the motes. As the "Distributed in network query processor" detail box in Figure 1 shows, this software consists of several components built on top of TinyOS. One of the motes—the *root*—communicates with the base station.

In our architecture, users input queries at the server in a simple, SQL-like language that describes the data they wish to collect and how they wish to combine, transform, and summarize it.

Introducing queries and query optimization

In our architecture, users input queries at the server in a simple, SQL-like language that describes the data they wish to collect and how they wish to combine, transform, and summarize it. Our SQL variant differs most significantly from traditional SQL in that its queries are continuous and periodic. That is, users register an interest in certain kinds of sensor readings (for example, "temperatures from sensors on the fourth floor every five seconds"), and the system streams the results to the user. Each period in which a result is produced is an *epoch*. The epoch duration, or sample period of a query, refers to the amount of time between successive samples (five seconds in this example).

As in traditional database systems, queries describe a logical set of data that the user is interested in, but don't describe the actual algorithms and software modules or operators the system uses to collect the answer set. Typically, the system can choose from several plans and operator orderings for any given logical query. For example, to find the average temperature of the fourth-floor sensors, the system might collect readings from every sensor, then filter the list for fourth-floor sensors and compute the average. Alternatively, it might request that only fourth-floor sensors provide their temperatures, and then average the values it collects. In a sensor network, the latter plan will be a better choice because it requires only sensors on the fourth floor to collect and report their temperatures.

The process of selecting the best possible plan is called *query optimization*. At a very high level, query optimizers work by enumerating a set of possible plans, assigning a cost to each plan based on estimated costs of each of the operators, and choosing the lowest-cost plan. In sensor networks, query optimization, because it can be computationally intensive, occurs as much as possible on the server-side PC. However, because the server might have imperfect state about the sensor network's status, and because costs used to optimize queries might initially change over their lifetimes, adapting running query plans after they've been sent into the network is sometimes necessary.

Query language

As in SQL, queries in Cougar and TinyDB consist of `SELECT-FROM-WHERE-GROUPBY-HAVING` blocks to support selection, join, projection, aggregation, and grouping. The systems also include explicit support for windowing and subqueries (via materialization points in TinyDB). TinyDB also explicitly supports sampling. In queries, we view sensor data as a single virtual table with one column per sensor type. The systems append tuples to the table at well-defined intervals specified as query parameters. The time between sample intervals is the epoch. Epochs provide a convenient mechanism for structuring computation to minimize power consumption.

For example, the query (`SELECT nodeid, light, temp, FROM sensors, SAMPLE PERIOD 1s FOR 10s`) specifies that each sensor should report its own identifer (id), light, and temperature readings once per second for 10 seconds. The virtual table `sensors` contains one column for every attribute available in the catalog and one row for every possible instant in time. The term "virtual" means that these rows and columns are not actually materialized—the systems only generate the attributes and rows referenced in active queries.

Results of this query stream to the network root via the multihop topology, where they can be logged or output to the user. The output consists of an ever-growing sequence of tuples clustered into 1-second time intervals. Each tuple includes a time stamp indicating when it was produced.

Conceptually, the `sensors` table is an unbounded, continuous data stream of values. As with other streaming and online systems, certain blocking operations (such as sort and symmetric join) aren't allowed over such streams unless the user specifies a bounded subset of the stream or window. Windows in TinyDB are fixed-size materialization points over the sensor streams that accumulate a small buffer of data for use in other queries. Similarly, Cougar's view nodes can store intermediate query results much like materialized views in relational database systems: sensors push data to view

nodes, where interactive queries pull them or periodically push them to other view nodes or a base station.

Consider, as an example, the following query (in TinyDB syntax):

```
CREATE STORAGE POINT recentlight
SIZE 8 seconds
AS (SELECT nodeid, light
    FROM sensors
    SAMPLE PERIOD 1s)
```

This statement provides a shared, local (that is, single-node) location to store a streaming view of recent data similar to materialization points in other streaming systems, such as Aurora[4] or Stream,[5] or materialized views in conventional databases. Joins are allowed between two storage points on the same node, or between a storage point and the sensors relation, in which case sensors serves as the outer relation in a nested-loops join. That is, when a sensors tuple arrives, it joins tuples in the storage point. This is effectively a landmark query,[6] common in streaming systems. For example, the following query outputs a stream of counts indicating the number of recent light readings (from 0 to 8 samples) that were brighter than the current reading.

```
SELECT COUNT(*)
FROM sensors AS s, recentLight AS rl
WHERE rl.nodeid = s.nodeid
AND s.light < rl.light
SAMPLE PERIOD 10s
```

TinyDB and Cougar also support grouped aggregation queries. Aggregation reduces the quantity of data that must be transmitted through the network. Thus, it can reduce energy consumption and bandwidth use by replacing more expensive communication operations with cheaper computation operations, extending the sensor network's life significantly. TinyDB also includes a mechanism for user-defined aggregates and a metadata management system that supports optimizations over them.

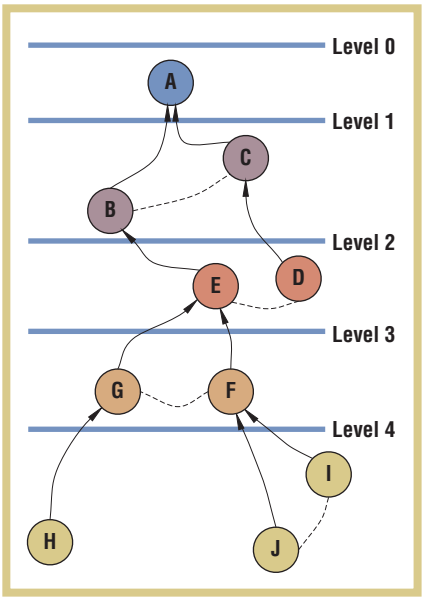


Figure 2. Sensor network topology with routing tree overlay. Solid arrows indicate parent nodes; dotted lines indicate nodes that can hear each other but don't route through each other.

Aggregation is a powerful paradigm with applicability extending far beyond simple averaging. For example, the Cougar system supports object tracking: nodes have a signal-processing layer that generates signatures for objects in a sensor's vicinity. Cougar implements a tracking operator as an aggregation over a region of sensor nodes whose detections are aggregated into an estimation of a track containing an object's estimated speed and direction. Overlap between regions ensures an accurate track at all times.

In addition to aggregates over values produced during the sample interval (for an example, as in the COUNT query just described), users want to be able to perform temporal operations. For example, in a monitoring system for conference rooms, users can detect occupancy by measuring maximum sound volume over time and reporting the volume periodically.

When a user issues a query in TinyDB or Cougar, the system assigns the query an id, which it returns to the user. Using the id, the user can stop a query via a STOP QUERY id command. Alternatively, the user can set queries to run for a specific time period via a FOR clause, or include a stopping condition as a triggering condition or event.[4]

Query dissemination and result collection

After optimizing a query, the system disseminates it into the network. A routing tree is a communication primitive rooted at either the base station or a storage point. The routing tree is formed as nodes forward the query to other nodes in the network: The network root transmits the query, and all child nodes hearing the query process it and forward it to their children, who forward it to their children, and so on until the entire network has heard the query.

Each radio message contains a hop-count, or level indicating the distance from the broadcaster to the root. To determine its own level, each node picks a parent node that is (by definition) one level closer to the root than the node is. The parent will be responsible for forwarding the node's (and its children's) query results to the base station. If nodes keep track of multiple parents, the network can have several routing trees, which can support several simultaneous queries with different roots. This type of communication topology, known as *tree-based routing*, is common within the sensor network community.

Figure 2 shows an example sensor network topology and routing tree. Solid arrows indicate parent nodes, and dotted lines indicate nodes that can hear each other but don't route through each other. A node can generally choose a parent from several possible nodes; a simple approach is to choose the ancestor node at the lowest level. In practice, choosing the proper parent is quite important in

TABLE 1
Common sensor network query-processing operators.

Operator	Description
Data acquisition	Acquire a reading (field) from a sensor or an internal device attribute, such as a light sensor reading or free RAM in the dynamic heap.
Select	Reject readings that don't satisfy a particular Boolean predicate. For example, the predicate temp > 80°F rejects readings under 80°F.
Aggregate	Combine readings according to an aggregation function. For example, AVG(light) computes the average light value over each mote.
Join	Concatenate two readings when some join predicate is satisfied. For example, the predicate mat-point.light > sensors.light joins (concatenates) all the historical tuples in mat-point with current sensor readings for any pair of tuples in which the current light value exceeds the historical value.

terms of communication and data collection efficiency. Moreover, network topologies are much less regular and more complex than you might expect. Unfortunately, the details of the best-known techniques for forming trees in real networks are complicated and outside this article's scope. For a more complete discussion of these and other issues, see, for example, recent work from the TinyOS group at UC Berkeley.[7]

A plethora of work on routing in ad hoc and sensor networks exists,[8,9] including energy-aware routing[10] and special MAC protocols.[11] Our goal here is different: Instead of using a general-purpose routing layer, we disseminate information from sensors to the root, leveraging knowledge about our communication patterns.

Each node in a completed routing tree has a connection to the tree's root that's just a few radio hops long. We can use this tree to collect data from sensors by having them forward query results up this path. In both TinyDB and Cougar, the routing tree evolves over time as new nodes come online, interference patterns change, or nodes run out of power. In TinyDB, nodes maintain the tree locally by keeping a set of candidate parents and an estimate of the quality of the communications link with each of them. When the quality of the link to the current parent is sufficiently worse than the quality to another candidate parent, the node takes the new parent.

A simple routing structure such as routing trees is well suited to our scenario: Sensor network query processors impose communication workloads on the multihop communication network that differ from those mobile nodes impose on traditional ad hoc networks. Because the sensor network is programmed only through queries, regular communication patterns exist, mainly consisting of the collection of sensor readings from a region at a single node or the base station. A query workload with more than a few destinations requires routing structures other than routing trees because the overlay of several routing trees neglects any sharing between trees and leads to performance decay. The discussion of such routing algorithms is beyond this article's scope, but we've begun to explore such issues.

Query processing

After a query has been disseminated, each node begins processing it. Processing is a simple loop: once per epoch, a special acquisition operator at each node acquires readings, or samples, from sensors corresponding to the fields or attributes referenced in the query. The query processor routes this set of readings, or tuple, through the query plan built in the optimization phase. The plan consists of a number of operators applied in a fixed order. Each operator can pass the tuple to the next operator, reject it, or combine it with one or more other tuples. A node transmits tuples that successfully pass the plan up the routing tree to the node's parent, which can, in turn, forward the result or combine it with its own data or data collected from other children. Table 1 describes some common query processing operators used in SNQPs.

The data acquisition operator uses a catalog of available attributes to map names referenced in queries to low-level operating system functions that can be invoked to provide their values. This catalog abstraction lets sophisticated users extend the sensor network with new kinds of sensors and provides support for sensors accessed via different software interfaces. For example, in the TinyDB system, users can run queries over sensor attributes such as light and temperature but can also query attributes that reflect the state of the device or operating system, such as the free RAM in the dynamic-memory allocator.

Figure 3 illustrates query processing for the simple aggregate query, "Tell me the average temperature on the fourth floor once every five seconds." Here, the query plan contains three operators: a data acquisition operator, a select operator that checks whether the value of the floor attribute equals 4, and an aggregate operator that computes the temperature attribute average from the local mote and the average temperature values of any of the mote's descendents that are on the fourth floor. Each sensor applies this plan once per epoch, and the data stream produced at the root node is the answer to the query. We represent the partial computation of averages as {sum,count} pairs, which are merged at each intermediate node in the query plan to compute a running average as data flows up the tree.

Several implementation details must be resolved for this scheme to work: sensors must wait to hear from their chil-

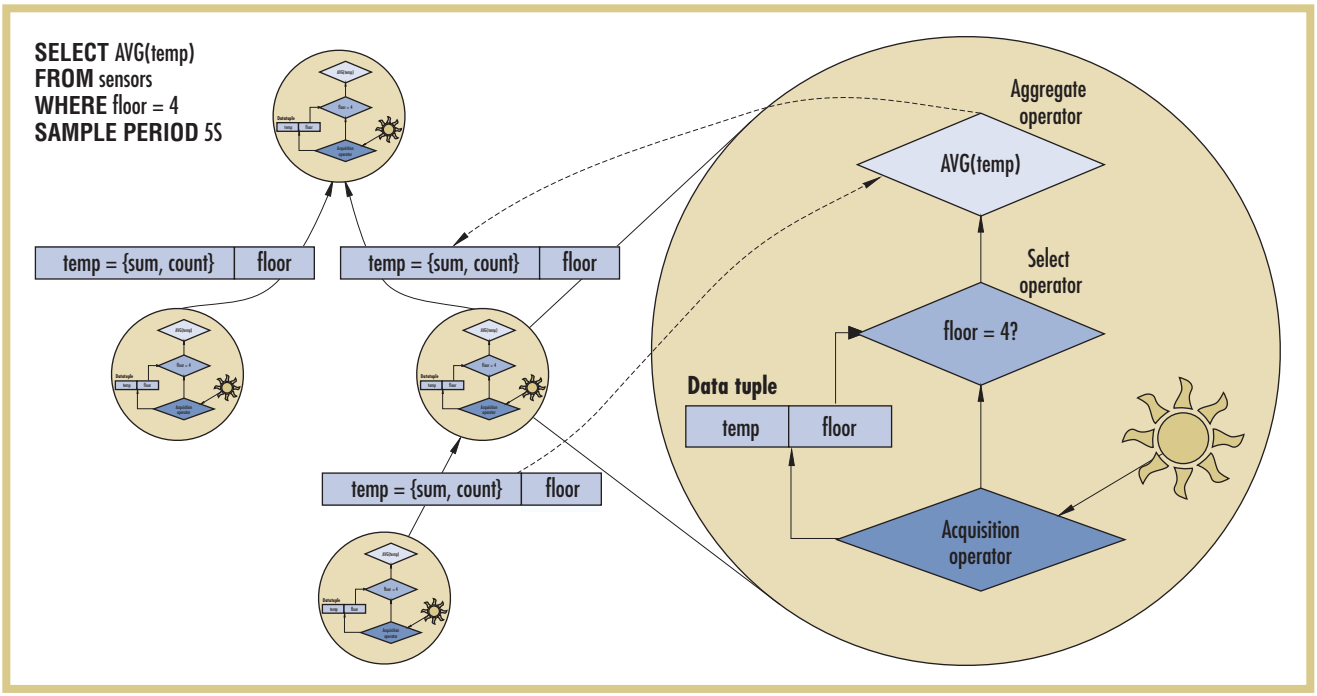


Figure 3. Sensor network executing a simple aggregate query.

dren before reporting their own averages, and average records must be represented in such a way that they can be combined as they flow up the tree (in this case, as a sum and a count, rather than a simple average).

Finally, for this example, if the nodes are known to be immobile, the predicate over **floor** will be constant valued, meaning that nodes on floors other than the fourth will never produce values for the query. TinyDB includes the notion of a *semantic routing tree*,[4] which lets nodes opt out of queries with nonsatisfiable predicates over constant-valued attributes. However, such nodes might still have to forward packets on behalf of other nodes that satisfy the predicate.

Sensor-network-specific techniques and optimizations

A number of unusual optimizations and query processing techniques arise in the SNQP context.

Lifetime clause

In lieu of an explicit **sample period** clause, we let users specify a **query lifetime** via a query lifetime <x> clause, where <x> is a duration in days, weeks, or months. Specifying a lifetime lets the user reason about power consumption more intuitively. In environmental monitoring scenarios in particular, scientific users might not be especially concerned with small adjustments to the sample rate, nor do they understand how such adjustments influence power consumption. Such users, however, are very concerned with the lifetime of the network executing the queries.

For example, the query (**SELECT nodeid, accel, FROM sensors, LIFETIME 30 days**) specifies that the network should run for at least 30 days, sampling light and acceleration sensors at as fast a rate possible while still satisfying this goal. To satisfy a lifetime clause, the SNQP applies *lifetime estimation*, which computes a sampling and transmission rate given a number of joules of remaining energy (which can usually be estimated from the mote's battery voltage) and a specific query or set of queries to run. As with query optimization, lifetime estimation can be performed when a query is initially issued at the PC, or applied periodically within the network as the query runs. We've currently implemented the former approach in TinyDB, but the latter approach will be more effective, especially in a network with a lot of nodes communicating in unpredictable ways.

To illustrate this estimation's effectiveness, we inserted a lifetime-based query (**SELECT voltage, light FROM sensors LIFETIME x**) into a sensor with a new pair of AA batteries and asked it to run for 24 weeks. The result was a sample rate of 15.2 seconds. We measured the remaining voltage on the device nine times over 12 days. The first two readings were outside the mote voltage detector's range (they read 1,024—the maximum value). On the basis of experiments with our test mote connected to a power supply, we expected the device to stop functioning when its voltage reached 350. Figure 4 shows the measured lifetime, linear fit, and expected voltage at each point in time, which we computed using a simple cost model. The resulting voltage linear fit is near the expected voltage. The linear fit reaches V = 350 about five days after the expected voltage line.

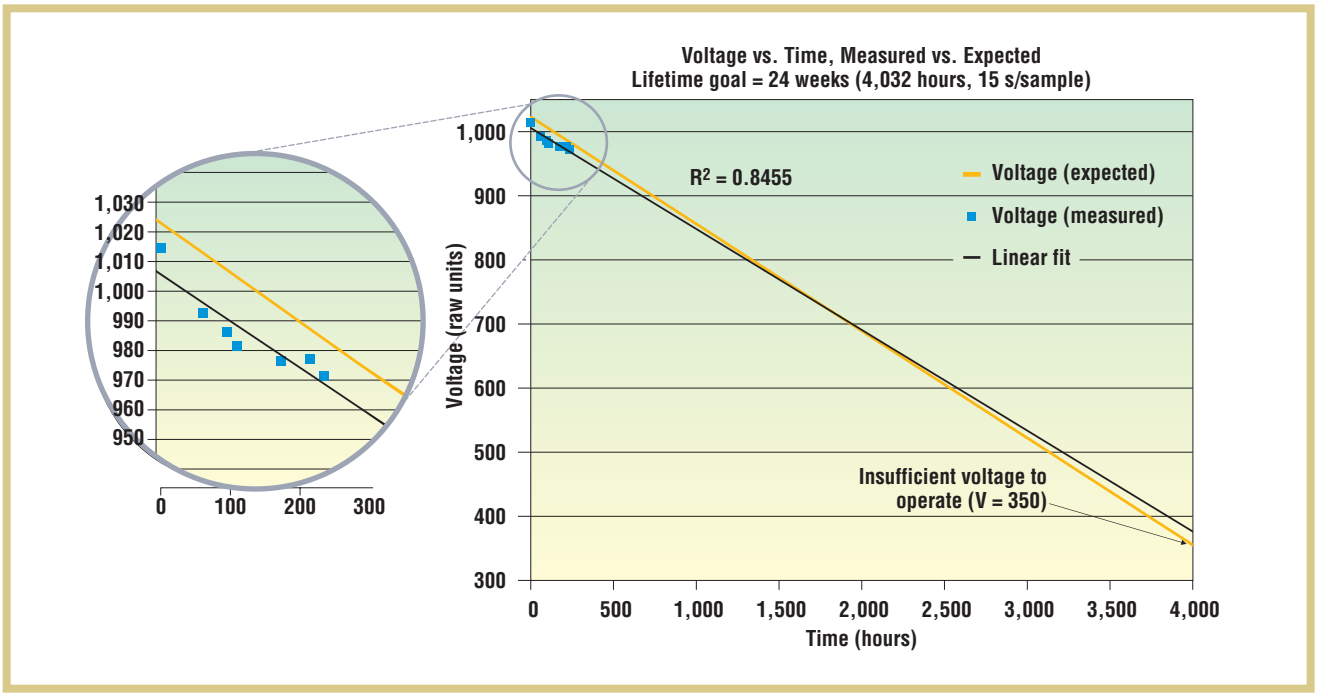


Figure 4. Predicted versus actual lifetime for a requested 24-week (168-day) lifetime.

Lifetime estimation is a simple optimization technique that the sensor network can apply to give users a more useful, expressive way of interacting with a network of sensors.

Pushing computation

Among the most general techniques for query optimization in sensor network database systems is *pushing computation*, or moving processing into the network, toward the origin of the data being processed.

We divide a query plan for a simple aggregate query into two components. Because queries require data from spatially distributed sensors, we must deliver records from a set of distributed nodes to a central destination node for aggregation by setting up suitable communication structures for delivering sensor records within the network. This is the communication component of a query plan. The query plan's computation component computes the aggregate at the network root and potentially computes already partial aggregates at intermediate nodes.

We describe two simple schemes for pushing computation; more sophisticated push-based approaches are also possible.[12,13]

Partial aggregation. For aggregates that can be incrementally maintained in constant space (or, in database terminology, for distributive and algebraic aggregate operators), we push computation from the root node down to intermediate nodes. Intermediate sensor nodes compute partial results containing sufficient statistics to compute the final result. We can use this scheme to distribute the aggregate AVERAGE, which has constant intermediate state. The example in Figure 3 illustrates the concept of pushing partial aggregation into the network.

Packet merging. Because sending multiple small packets is more expensive than sending one large packet (considering the cost of reserving the channel and the packet header payload), we merge several records into a large packet and only pay the packet overhead once. For exact query answers with aggregate operators that don't have a compact incremental state representation such as Median (that is, *holistic aggregates*), packet merging is the only way to reduce the number of bytes transmitted.

Cross-layer interactions

These in-network aggregation techniques require internal nodes to intercept data packets passing through them. However, with traditional network-layer send-and-receive interfaces, only the routing tree root receives the data packets. The network layer on an internal node automatically forwards the packets to the next hop toward the destination, with the upper layers unaware of data packets traveling through the node. Thus a node must be able to intercept packets routed through it, and the sensor network query processor needs a way to communicate to the network layer when it wants to intercept packets destined for another node.

Cougar uses network filters to implement this interception. With filters, the network layer passes a packet through a set of registered functions that can mod-

ify it (and possibly even delete it). In the query layer, if a node *n* is scheduled to aggregate data from all children nodes, it intercepts all data packets received from its children and caches the aggregated result. At a specific time, *n* generates a new data packet representing the incremental aggregation of it and its children's data and sends it toward the network root. All this happens transparently to the network layer.

TinyDB implements this interception by collapsing the network stack and merging the routing layer with the application layer. In this case, the application controls the routing layer completely, and the application-level routing layer handles each packet routed through a node.

Both approaches are instances of cross-layer interactions. To preserve resources, we believe future sensor network generations will take an integrated approach to system architecture design, cross-cutting the data management and communication (routing and MAC) layers using one of two approaches:

- In the top-down approach, we design and adapt communication protocols and their interfaces to the communication needs of the sensor network query processor. Adding filters to existing routing protocols is a top-down technique. Because in-network query processing must intercept each message at each node for possible aggregation, we adapt the routing-layer interface to let the application layer filter messages routed through a node.
- In the bottom-up approach, we design new communication patterns optimized for query-processing workloads (for example, a routing tree).

Cross-layer interactions are a fertile area of sensor network research, and TinyDB and Cougar have only made preliminary steps in this direction.

Data collection experiments

We've studied the performance and behavior of our SNQP implementations, both in simulation to demonstrate the potential of our algorithms and approaches and in real-world environments to observe their overall effectiveness.

Berkeley Botanical Garden deployment

In June and July 2003, we deployed the TinyDB software in the Berkeley Botanical Garden, located near the University of California, Berkeley, campus. The deployment sought to monitor environmental conditions in and around coastal redwood trees (the microclimate) in the garden's redwood grove, which consists of several hundred new-growth redwoods. Botanists at UC Berkeley actively study these microclimates with a particular interest in the role the trees play in regulating and controlling their environment, especially how they affect the humidity and temperature of the forest floor on warm, sunny days.[14]

The initial sensor deployment consisted of 11 Mica2 sensors on a single 36-meter redwood tree. Each sensor was equipped with a weather board providing light, temperature, humidity, solar radiation, photosynthetically active radiation, and air pressure readings. We clustered the sensors at different altitudes throughout the tree. We placed the processor and battery in a watertight PVC enclosure, with the sensors exposed on the outside. A loose-fitting hood covered the bottom of the sensors to protect humidity and light sensors from rain. We sealed the light and radiation sensors on the top of the assembly against moisture and thus left them exposed.

Sensors on the tree run a simple selection query that retrieves a full set of sensor readings every 10 minutes and sends them to the base station, which is attached to an antenna on a nearby field station's roof, about 150 feet from the tree. The field station is connected to the Internet, so users can easily log results into a PostgreSQL database for analysis and observation. The sensors have been running continuously for about three weeks.

Each sensor was equipped with a weather board providing light, temperature, humidity, solar radiation, photosynthetically active radiation, and air pressure readings.

Figure 5 shows data from five of the sensors, collected during the second week of July 2003. Sensor 101 was at a height of 10 meters, sensor 104 at 20 meters, 109 at 30 meters, 110 at 33 meters, and 111 at 34 meters. We exposed 110 and 111 but kept the others shaded in the forest canopy.

The periodic bumps in the graph correspond to daytime readings; at night, the temperature drops significantly and humidity becomes very high as fog rolls in. Notice that 7 July was a cool day, below 18 degrees Celsius and likely overcast. On such days, all sensors recorded approximately the same temperature and humidity. On warmer days, however, the temperature was as much as 10 degrees cooler at the bottom of the tree and 30 percent more humid.

Although this fairly basic deployment runs only a simple query, we programmed the sensors to begin data collection in just a few minutes. By far the most time-consuming aspects of the deployment in-

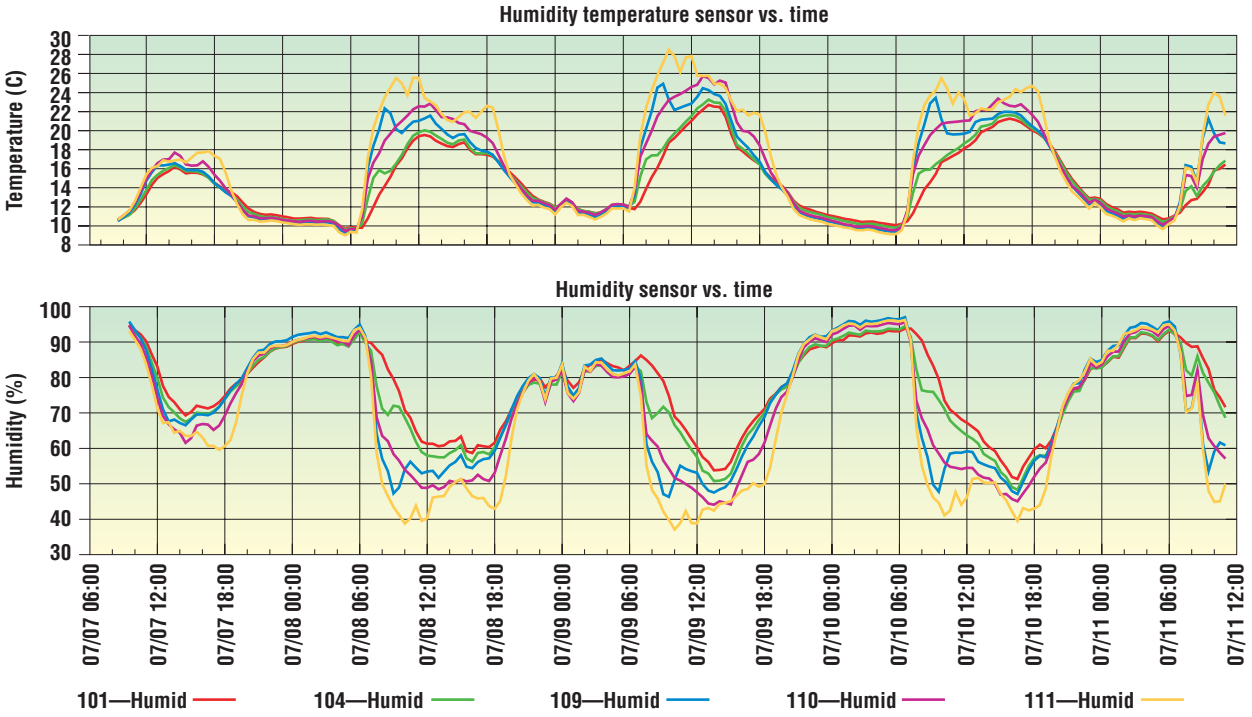


Figure 5. Humidity and temperature readings from five sensors in the Berkeley Botanical Garden.

volved packaging the devices, obtaining access to the various spaces, and climbing the tree to place the sensors.

One interesting future direction related to this deployment involves tracking correlations between sensors and using those correlations to improve query-processing efficiency. In Figure 5, temperature and humidity are highly correlated; thus, knowing the humidity and sensor number lets us predict the temperature to within a few degrees Celsius. This observation suggests an interesting query optimization possibility: We can evaluate queries containing predicates over temperature by instead looking at humidity. This could be an energy-saving alternative if we need a humidity sample for other purposes or the energy costs of acquiring a humidity sample are low.

Simulation experiments

Several simulations of our approach showed that it works well in a controlled environment (often, simulation is the only way to get repeatable results out of noisy, lossy sensor networks). We have a prototype of Cougar's query-processing layer running in the ns-2 network simulator.[15] Ns-2 is a discrete-event simulator aiming to simulate network protocols to highest fidelity. Because of the strong interaction between the network layer and our proposed query layer, we simulate the network layer to a high degree of precision, including collisions at the MAC layer and detailed energy models developed by the networking community.

In our experiments, we used IEEE 802.11 as the MAC layer, setting the communication range of each sensor to 50 meters and assuming bidirectional links. In our energy model, the receive and transmit power dissipation is 395 and 660 milliwatts. We assumed energy usage in the idle state is negligible. We ran a simple query that computes the average sensor value over all sensor nodes every 10 seconds for 10 continuous rounds.

Figure 6 illustrates the benefit of the in-network (push-down) aggregation approaches for a simple network topology—a uniform distribution of sensors in a square region with a gateway sensor in the top-left corner. We set the average sensor node density to eight sensors per 100 m^2, while increasing the region size to fit 40 to 240 sensors. Partial aggregation and packet merging dramatically reduced the amount of energy used by reducing the amount of data sent to the gateway node.

Database approaches to sensor network data management are promising, as initial experiences with users of our technology show. Declarative queries offer both an easy-to-interface and energy-efficient execution substrate. Fur-

thermore, our approach has unearthed a plethora of interesting research problems in this domain. Future challenges include

- *Multiquery optimization.* At any time, several long-running queries from multiple users might run over a sensor network. How can we share resources among these queries to balance and minimize overall resource usage?
- *Storage placement.* Storage points or view nodes provide an abstraction for in-network sensor data storage. Where should we place view nodes to balance and minimize resource usage? What type of fault tolerance can prevent the query from losing data if a view node fails?
- *Heterogeneous networks.* So far we've only considered relatively homogeneous sensor networks in which all nodes are equally powerful. Future networks will likely have several tiers of nodes with different performance characteristics. How can sensor network query processors take advantage of this heterogeneity?

These problems suggest that sensor network database research will continue to be a rich and exciting field for many years to come.

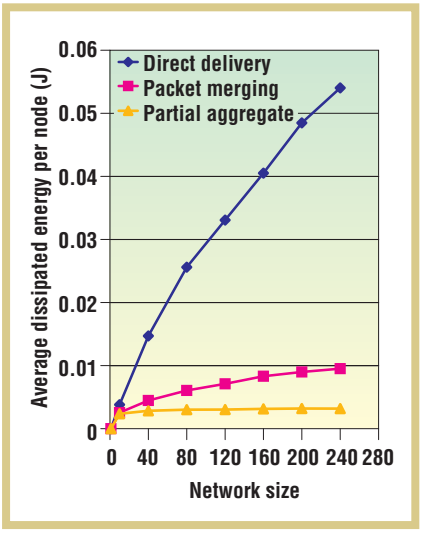
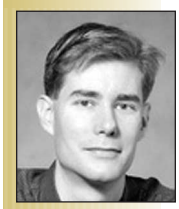


Figure 6. Simulation results of different approaches for answering an aggregate query.

REFERENCES

1. P. Bonnet and P. Seshadri, "Device Database Systems," *Proc. Int'l Conf. Data Eng.* (ICDE), IEEE CS Press, 2000, pp. 194.
2. J. Hill et al., "System Architecture Directions for Networked Sensors," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), ACM Press, 2000, pp. 93–104.
3. S. Madden and M.J. Franklin, "Fjording the Stream: An Architecture for Queries over Streaming Sensor Data," *Proc. Int'l Conf. Data Eng.* (ICDE), IEEE CS Press, 2002, pp. 555–566.
4. S. Madden et al., "The Design of an Acquisitional Query Processor for Sensor Networks," *Proc. ACM Sigmod*, ACM Press, 2003, pp. 491–502.
5. R. Motwani et al., "Query Processing, Approximation and Resource Management in a Data Stream Management System," *Proc. Conf. Innovative Data Systems Research* (CIDR), ACM Press, 2003.
6. J. Gehrke, F. Korn, and D. Srivastava, "On Computing Correlated Aggregates over Continual Data Streams," *Proc. ACM SIGMOD Conf. Management of Data*, ACM Press, 2001, pp. 13–24.
7. A. Woo, T. Tong, and D. Culler, "Taming the Underlying Challenges for Reliable Multihop Routing in Sensor Networks," *Proc. ACM SenSys*, ACM Press, 2003.
8. C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks, *Proc. Int'l Conf. Mobile Computing and Networking* (MobiCom), ACM Press, 2000, pp. 56–67.
9. C.E. Perkins, "Ad Hoc on Demand Distance Vector (AODV) Routing," Internet draft, work in progress, Oct. 1999; http://moment.cs.ucsb.edu/AODV/aodv.html.
10. G.J. Pottie and W.J. Kaiser, "Embedding the Internet: Wireless Integrated Network Sensors," *Comm. ACM*, vol. 43, no. 5, May 2000, pp. 51–58.
11. W. Ye, J. Heidemann, and D. Estrin, "An Energy-Efficient MAC Protocol for Wireless Sensor Networks," *Proc. IEEE Infocom*, IEEE CS Press, 2002, pp. 1567–1576.
12. S. Madden et al., "TAG: A Tiny AGgregation Service for Ad Hoc Sensor Networks," *Proc. Symp. Operating System Design and Implementation* (OSDI), Usenix, 2002.
13. Y. Yao and J. Gehrke, "Query Processing in Sensor Networks," *Proc. 1st Biennial Conf. Innovative Data Systems Research* (CIDR), ACM Press, 2003.
14. T. Dawson, "Fog in the California Redwood Forest: Ecosystem Inputs and Use by Plants," *Oecologia*, vol. 117, 1998, pp. 476–485.
15. L. Breslau et al., "Advances in Network Simulation," *Computer*, vol. 33, no. 5, May 2000, pp. 59–67.

the AUTHORS



Johannes Gehrke is an assistant professor in the Department of Computer Sciences at Cornell University. His research interests include data mining, data-stream processing, and distributed query processing. He has a PhD in computer science from the University of Wisconsin-Madison. He is a member of the ACM and the IEEE. Contact him at 4105B Upson Hall, Cornell Univ., Ithaca, NY 14850; johannes@cs.cornell.edu.

Samuel Madden is an assistant professor at the Massachusetts Institute of Technology. His research interests include database system design, query processing and optimization, and sensor networks. He has a PhD from the University of California, Berkeley. Contact him at NE43-532, MIT CSAIL, Technology Square, Cambridge, MA 02139; madden@csail.mit.edu.