INVITED PAPER

# Database Abstractions for Managing Sensor Network Data

*This paper surveys several research systems designed to manage sensor data by using database-like abstractions; these systems are designed to organize data collection and to clean and smooth data.*

By Samuel Madden

**ABSTRACT** | Sensor networking hardware, networking, and operating system software has matured to the point that the major challenges facing the field now have to do with storing, cleaning, and querying the data such networks produce. In this paper, we survey several research systems designed for managing sensor data using declarative database-like abstractions from the database community and specifically the Massachusetts Institute of Technology (MIT, Cambridge) database group. The systems we discuss are designed to help prioritize data collection in the face of intermittent bandwidth, clean and smooth data using statistical models stored inside the database, and run declarative queries over probabilistic data.

**KEYWORDS** | Database management systems; database query processing; intelligent sensors

## I. INTRODUCTION

Sensor networks—wireless collections of small devices that capture data about the world around us—have been a popular research topic for the past decade. They offer the potential to collect information about cities, natural habitats, industrial equipment, etc., at an unprecedented scale and granularity. For example, in our work on the CarTel[1] project at the Massachusetts Institute of Technology (MIT, Cambridge), we have been using a network of sensor-equipped cars to monitor traffic, road surface conditions (potholes, etc.), and driver behavior. Fig. 1 shows a screenshot of average historical traffic delays in Boston, MA and Cambridge, MA on weekdays at 5 P.M., as collected by a CarTel network running on a fleet of 30 taxi cabs for the past three years.

There have been a variety of different sensor networking hardware platforms developed. Perhaps the best known is the Berkeley Mote platform. Though there are many mote variants, they all feature a low-power microprocessor running at tens or hundreds of megahertz, a few tens of kilobytes of RAM, and up to a few megabytes of Flash storage. They can typically run for a few days at full power on a pair of AA batteries, and several months or even a few years when power is carefully managed. They employ low-power, low-range radios that use about an order of magnitude less energy per packet than an 802.11 (WiFi) radio, but transmit at rates up to only a few hundred kilobits per second.

In the CarTel project, we use more conventional off-the-shelf hardware. Our current device is based on a WiFi access point, with a 802.11b/g radio, 64 MB of RAM, and 400-MHz processor. We have extended this device with a global positioning system (GPS), an accelerometer, an interface to the on-board diagnostic network in cars, and a low-power microprocessor that can wake the device when motion is detected.

These hardware platforms are relatively mature, as is the basic operating system software that runs on them. For example, our CarTel nodes run Linux, while the Berkeley Motes typically run TinyOS,[2] a low-level, event-driven operating system that provides interfaces for capturing sensor data and sending messages.

Techniques for dealing with the data produced by these devices are less mature than the basic hardware and OSes. Although there are many systems that can capture raw sensor data and perform simple processing on it, e.g.,

Manuscript received December 8, 2009; revised June 6, 2010; accepted July 16, 2010. Date of publication September 16, 2010; date of current version October 20, 2010.

The author is with the Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology (MIT), Cambridge, MA 02139 USA (e-mail: madden@csail.mit.edu).

Digital Object Identifier: 10.1109/JPROC.2010.2063691

[1]http://cartel.csail.mit.edu.

[2]http://www.tinyos.net/.

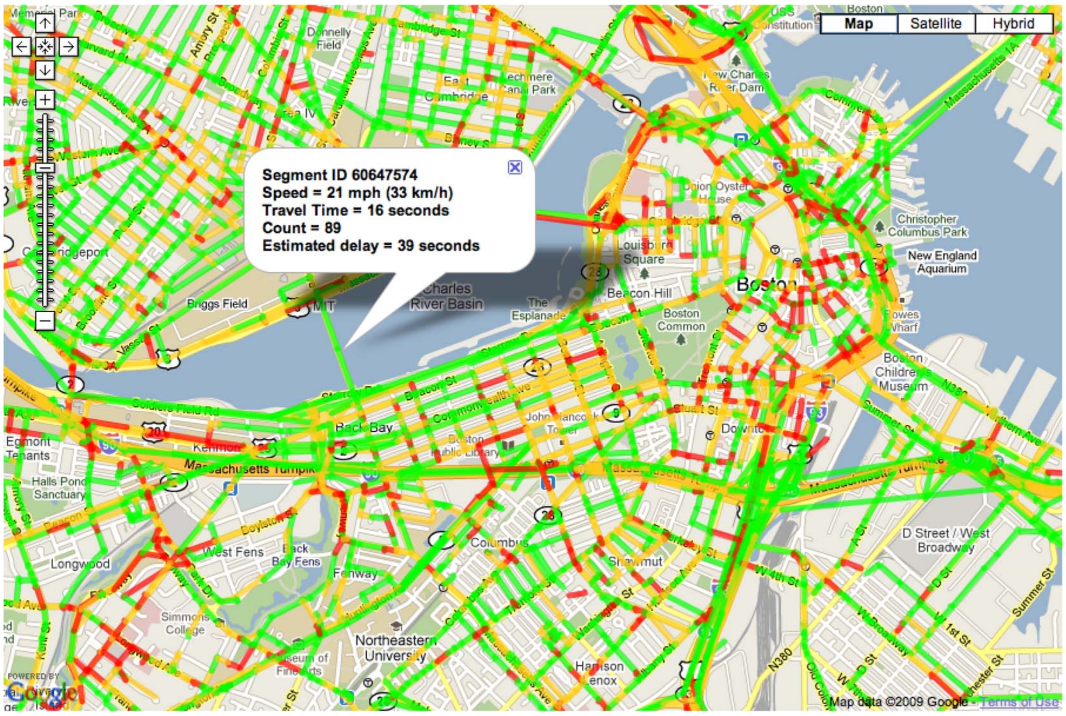

**Fig. 1.** ***Historical traffic in the Boston, MA area at 5 p.m. on a weekday; selected segments indicate traffic delays on the Harvard Bridge.***

sensor network query processors like TinyDB [1], Cougar [2], and SwissQM [3], as well as mote-based data collection protocols like CTP [4] and many off-the-shelf tools for Linux, there are a number of higher level data processing features that these systems lack. In particular:

- they do not provide users control over what data should be delivered in the presence of network congestion; it is often the case that sensor networks have more data to deliver than the network has bandwidth to handle;
- they do not provide facilities to allow users to interpolate missing readings, discard outliers, or otherwise model and clean their data;
- they do not provide high level query interfaces that allow users to pose queries over uncertain data produced by sensors and modeling processes.

In this paper, we discuss recent research from the database community, and in particular our group at MIT to address these shortcomings using relational database techniques. Although relational databases are not the only way to interact with data, they do provide a high level interface that many programmers find familiar and convenient, and are widely used by scientists and other researchers as a low-level interface for storing and extracting raw sensor data. We focus specifically on three research systems and areas.

1) Tools for data collection and prioritization: Specifically, we discuss ICEDB, a declarative query language that specifies what data to collect in the face of intermittent and limited connectivity.
2) Model-based views and FunctionDB: A system that allows database systems to fit models to data inside of the database, and to answer queries over that modeled data.
3) Uncertain data processing: Techniques for running queries when data has uncertainty as a result of the modeling process or underlying sensors.

In the next section, we describe the general outline of the rest of the paper, beginning with an overview of how these data management components work together.

## II. ARCHITECTURE FOR SENSOR NETWORK DATA MANAGEMENT

Fig. 2 shows how these three components interact in a generalized sensor network data management system. Raw data are collected from sensors and typically processed by an on-node data management system (ICEDB, in our case); this raw data are streamed to some central "portal" that loads the data into a server-based database. These data are cleaned (using model-based views, or some other technique) and then made available to external users for querying; users can specify how they would like to model their data (e.g., what functions to fit to it). Though conventional SQL may be an appropriate query language, data from sensors, or output by model-based views, are often probabilistic in nature, requiring extensions to SQL to deal with the additional complexity of uncertain data.

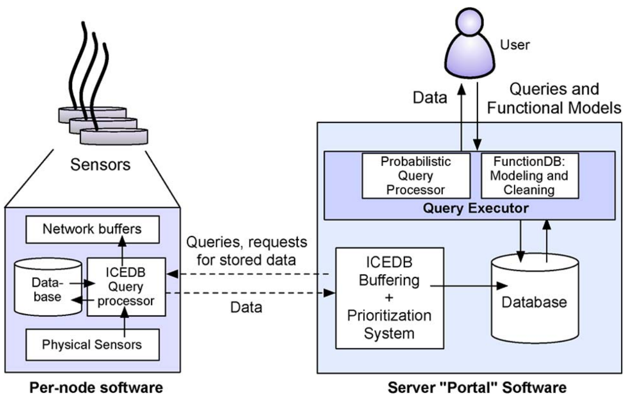
User
Data
Queries and Functional Models
Sensors
Probabilistic Query Processor
FunctionDB: Modeling and Cleaning
Query Executor
Network buffers
Queries, requests for stored data
Data-base
ICEDB Query processor
Data
ICEDB Buffering + Prioritization System
Database
Physical Sensors
Per-node software
Server "Portal" Software

**Fig. 2.** ***Software architecture of an idealized sensor network data processing system.***

In the remainder of this paper, we discuss these components in more detail, describing our ICEDB system for data collection and prioritization in Section III, our approach to cleaning and smoothing noisy data collected from such a system in Section IV, and several related projects that have looked at running queries on top of such cleaned data when it involves probabilistic or uncertain values in Section V.

## III. DATA COLLECTION AND PRIORITIZATION

One of the challenges of sensor networks is that sensors can produce *lots* of data—sometimes thousands of samples per second—and network bandwidth can be quite variable. In some deployments, such as our CarTel system, nodes have high bandwidth WiFi radios that can deliver tens of megabits per second, but are intermittently connected to other nodes or basestations and so must buffer data. In other cases, such as mote networks, radios are much lower bandwidth—delivering just hundreds of kilobits per second. In mote networks, interference from other nodes, especially in dense networks, can mean that each node sees even less bandwidth than this.

To address issues related to variable bandwidth in a data collection system, we developed ICEDB [5]. ICEDB is a delay-tolerant distributed continuous query processor. To support variable bandwidth, it provides two key features: first, queries can be annotated with one of several prioritization clauses that dictate which query results are most important; second, query results and queries are buffered and delivered when connectivity is available.

In ICEDB, user applications define data sources and express declarative queries in a SQL dialect at a centralized *portal*. ICEDB distributes these data sources and queries to "local" query processors running on each mobile node, such that all nodes operate on the same data sources and run the same queries. The nodes gather the required data, process it locally, and deliver it to the portal whenever network connectivity is available. Queries and control messages also flow from the portal to the remote nodes during these opportunistic connections.

A data source consists of a physical sensor attached to the node, software on the node that converts raw sensor data to a well-defined schema, and the schema itself. These data sources produce tuples at a certain rate and store them into a local database on each node, with one table per data source. Continuous and snapshot queries sent from the portal are then executed over this database.

The main difference between ICEDB and traditional continuous query processors (e.g., Aurora [6], TelegraphCQ [7], and STREAM [8]) is that the results of continuous queries are not immediately sent over the network, but instead are staged in an *output buffer*. The total size of each raw sensor data store and output buffer is limited by the size of the node's durable storage. Queries and data sources are prioritized, and we use a policy that evicts the oldest data from the lowest prioritized buffers or tables first. Buffers are drained using a network layer tuned for intermittent and short-duration wireless connections. As results arrive at the portal, tuples are partitioned into tables according to the name of the source query and the remote node ID.

In general, each node produces many more tuples than it can transmit to the portal at any time. The main advantage of buffering is that it allows an ICEDB node to select an order in which it should transmit data from among currently available readings when connectivity is present, rather than simply transmitting data in the order produced by the queries. This allows us to reduce the priority of old results when new, higher priority data are produced, or to use feedback from the portal to extract results most relevant to the current needs or users.

As result tuples flow into the output buffer from the continuous and *ad hoc* queries, they are placed into separate buffers. Each query (and corresponding buffer) can specify a `PRIORITY`. The node's network layer empties these buffers in priority order. Tuples from queries of the same priority are by default processed in a round-robin fashion.

The `PRIORITY` clause alone is insufficient to address all prioritization issues because the amount of data produced by a single query could still be large. To order data within a query's buffer, queries may include a `DELIVERY ORDER BY` clause, which causes the node to assign a "score" to each tuple in the buffer and deliver data in score order.

For example, the query

```
SELECT gps.speed FROM gps, road_seg
  WHERE gps.insert_time > cqtime − 5 AND
  road_seg.id = lookup(gps.lat, gps.lon)
  EVERY 5-s BUFFER IN gpsbuf
  DELIVERY ORDER BY gps.speed−
     road_seg.speed_limit DESC
```

requests that speed readings from cars that most exceed the speed limit be delivered first. Many more sophisticated prioritization schemes are possible using DELIVERY ORDER BY, as described in [5].

ICEDB also provides a centralized way for the portal to tell nodes what is most valuable to it, using the option SUMMARIZE AS clause in queries. Using this clause, nodes generate a low-resolution summary of the results present in the corresponding query's output buffer. When a node connects to the portal, it first sends this low-resolution summary. The portal then uses the summary to rank the node's results, and sends the ranking to the node. The node then orders the data in that query's buffer according to the ranking. This enables the portal, for example, to ask different cars to prioritize data from different geographic locations, avoiding redundant reports.

These prioritization mechanisms are run continuously, maintaining a buffer of data that will be delivered when a network connection is available. When a node does connect to the portal, several different rounds of communication occur. First, the portal sends a *changelog* of updates (adds, removes, modifications) to queries, data sources, and prioritization functions that have occurred since the node last connected (this information is maintained in a local database on the portal). Simultaneously, the node sends any summaries generated by SUMMARIZE AS queries and the portal sends back orderings for results based on these summaries. Once the summarization process is complete, the node drains its output buffers using an output iterator in the following order: 1) in order of buffer priority, using weights among equal priority buffers; 2) within each buffer, in the rank order specified in the summaries (if the query uses SUMMARIZE AS); and 3) within each "summary segment," in order of the score assigned by the DELIVERY ORDER BY clause.

A similar set of ideas for prioritizing data delivery were explored in Lance [9]; both Lance and ICEDB provide techniques for allowing applications to specify what data they would like to deliver next, and for assigning priorities to data items using both local and basestation-assigned priorities. One significant difference is that Lance is not coupled with a declarative data collection system; this may make it more flexible and capable of being embedded in existing mote-based applications, but also makes it harder to use as it requires a mote-based application to be coded in low-level nesC. In contrast, for applications whose data collection needs can be effectively expressed as SQL queries, ICEDB requires no additional programming for the sensor node.

## IV. MODEL-BASED VIEWS

ICEDB and Lance provide a way to collect specific, high-value data from nodes when connectivity is limited. Once data are collected, however, users typically need to perform several cleaning and processing steps before that data can be directly used. Performing this cleaning is the goal of our work on *model-based views* [10].

Model-based views are similar to views in a traditional database, except that instead of simply selecting out a portion of an underlying relational data set, model-based views fit a *model* (the view) to an underlying collection of data and then allow relational queries to be run over that model. For example, a simple model might represent a collection of temperature readings from a sensor using interpolation such that there are no gaps of larger than 1 s; this would be expressed as a model-based view as follows:

```
CREATE VIEW
IntView(time[ 0::1],sensorid[ ::1],temp)
AS INTERPOLATE temp USING tm, sensorid
FOR EACH sensorid M
TRAINING_DATA SELECT temp, tm, sensorid
FROM raw-temp-readings
WHERE raw − temp − readings.sensorid = M
```

This view says that temperature should be interpolated from time, with a different view for each sensor.

Model-based views have widespread utility in sensor applications. In particular, we have spent some time looking at applications of views that fit continuous functions (i.e., apply regression) to underlying sensor data [11].

In some cases, continuous functions emerge naturally from the data—for example, a vehicle trajectory can be represented as a series of road segments derived from geographic data. In others, functions arise from discrete data using some form of regression (curve fitting). In many applications, posing queries in the curve or function domain yields more natural and/or accurate answers compared to querying raw data, for one or both of the following reasons.

*Continuous Data:* For many applications, a set of discrete points is an *inherently* incomplete representation of the data. For example, if a number of sensors are used to monitor temperature in a region, it is necessary to interpolate sensor readings to predict temperature at locations where sensors are not physically deployed.

*Noisy Data:* Raw data can also contain measurement errors, in which case simple interpolation does not work. For example, in a sensor network, sensors can occasionally fail, or malfunction and report garbage values due to low batteries/other anomalies; when using a prioritization system such as ICEDB, some data may be dropped due to bandwidth shortages. In such situations, it is preferable to query values predicted by a regression function fit to the data, rather than the raw data itself.

As a simple example of a case where model-based views are needed, consider Fig. 3. Here a trace of GPS points from one of the taxis in our CarTel deployment is shown.

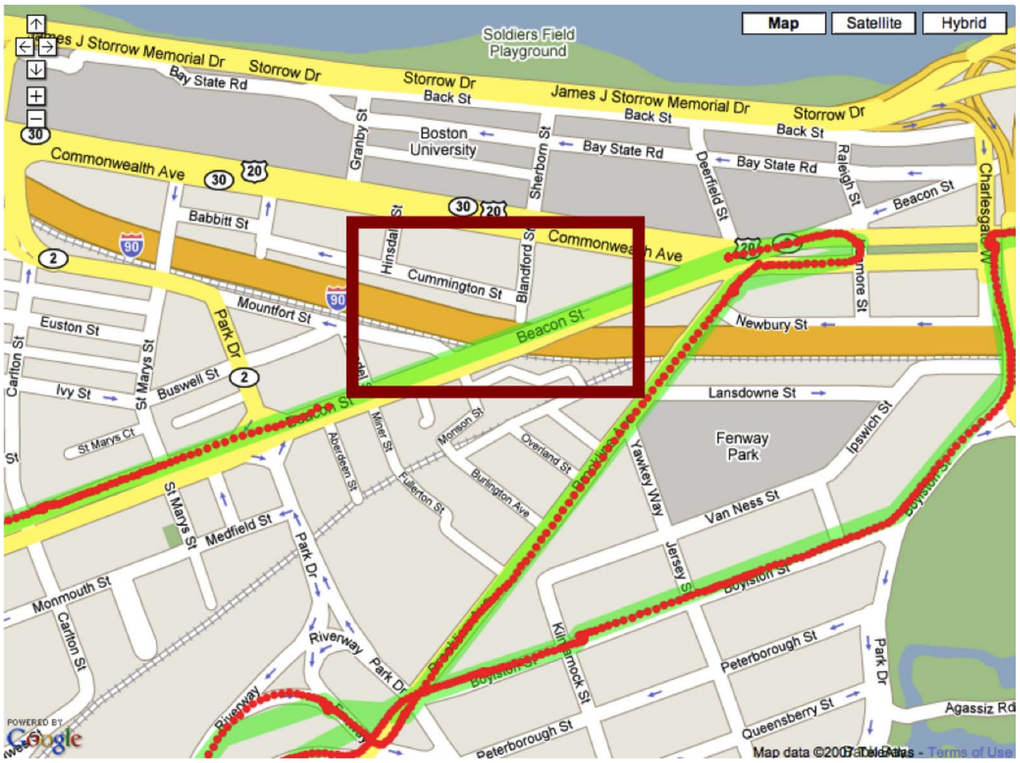

**Fig. 3.** ***Example showing missing GPS points along a roadway and a query rectangle over the region where data are missing. Model-based views allow the database system to interpolate missing values inside this region.***

Notice that there is a gap in the data (inside the dark rectangle), where there are no red dots. Now imagine the user asks for the average speed in, or the time when the cab passed through, the dark rectangle. A conventional database system would simply report no answer—since there are no points inside the rectangle. However, it is clear to a person that the car did in fact pass through the rectangle. Model-based views make it possible to use interpolation or regression to make the database system able to "understand" this fact as well. Model-based views can annotate interpolated values with probabilities or uncertainties to indicate to the user that they were introduced by the view; these probabilities can be queries using techniques for probabilistic querying described in Section V.

The problem of querying continuous functions is not adequately addressed by existing tools. Packages like MATLAB support fitting data with regression functions, but do not support declarative queries. Databases excel at querying discrete data, but do not support functions as first-class objects. They force users to store continuous data as a set of discrete points, which has two drawbacks—it fails to provide accurate answers for some queries, as discussed, and for many other queries, it is inefficient because a large number of discrete points are required for accurate results.

In our work on FunctionDB, we build a model-based view system that allows users to query the data represented by a set of continuous functions, for addressing applications like the "missing data" scenario in Fig. 3. By pushing support for functions into the database, rather than requiring an external tool like MATLAB, users can manage these models just like any other data, providing the benefits of declarative queries and integration with existing database data. FunctionDB supports special tables that can store mathematical functions, in addition to standard tables with discrete data. The system provides users with tools to input functions, either by fitting raw data using regression, or directly if data are already continuous (e.g., road segments). For example, a user might use FunctionDB to represent the points $(t=1, x=5)$, $(t=2, x=7)$, $(t=3, x=9)$ as the function $x(t)=2t+3$, defined over the domain $t \in [1,3]$.

In addition to creating and storing functions, FunctionDB allows users to pose familiar relational queries over the data represented by these functions. The key novel feature of FunctionDB is an *algebraic* query processor that executes relational queries using symbolic algebra whenever possible, rather than converting functions to discrete points for query evaluation. Relational operations become algebraic manipulations in the functional domain. For example, to answer a selection query that finds the time when the temperature of a sensor whose value is described by the equation $x(t)=2t+3$ equals 5, the system must solve the equation $2t+3=5$ to find $t=1$.

Algebraic query processing is challenging because queries over functions of multiple variables can generate systems of inequalities for which closed-form solutions are intractable. Consider $\texttt{temp} = f(x, y)$, a model of temperature as a function of $x$ and $y$ location, and suppose a user writes a query like `SELECT AVG(temp) WHERE` $x^2 + y^2 < 20$, which requests the average temperature in a circular region. This requires computing the integral of $f(x, y)$ over the region. For arbitrary functions, or regions defined by arbitrary `WHERE` predicates, closed-form solutions either do not exist, or are expensive to compute. Hence, FunctionDB chooses a middle ground between brute-force discretization and a fully symbolic system. It is able to efficiently approximate a broad class of regions—those defined by arbitrary systems of *polynomial* constraints (such as the region in the `WHERE` clause above) by a collection of hypercubes.

In [11], we describe the details of how we efficiently evaluate queries over regions enclosed by polynomial constraints by approximating them with hypercubes, while preserving the semantics of query results. Our results from evaluating FunctionDB on two real data sets (data from 54 temperature sensors in an indoor deployment, and traffic traces from cars) show that FunctionDB achieves order of magnitude ($10\times$–$100\times$) better performance for aggregate queries and $2\times$–$10\times$ savings for selective queries, compared to approaches that represent functions as discrete points on a grid (as we did in [10]). FunctionDB is also more accurate than gridding, which results in up to 15%–30% discretization error. Our current implementation of FunctionDB provides support for multivariate regression over polynomials of arbitrary degree; the model is easily extensible to other types of regression and curve fitting (e.g., logistic regression, fitting to differential equations [12]).

Other work on model-based views has looked at the use of probabilistic graphical models [13] and well as time series and differential equations [12].

## V. QUERYING UNCERTAIN DATA

After modeling with a system like FunctionDB, users need a way to pose queries over their data. Though FunctionDB can eliminate noise and outliers by fitting functions to data, sensor data can still be *uncertain*. Uncertain data arise due to the following factors.

- Inherent uncertainty in sensors: Physical sensors are inherently inaccurate, producing discrete values that effectively sample some distribution around the "true" value that they measure. For example, a GPS sensor typically reports a position estimate that is accurate to within about 5 m of the true position.
- Limited temporal or spatial resolution: Sensors only capture data about a limited geographic extent at some maximum rate. Bandwidth, money, and energy further restrict the number and maximum delivery rate for data. To correct for these "gaps," some form of interpolation or regression is typically used, introducing uncertainty in the times and areas where no data are available.

The database community has recently focused on the problem of querying uncertain data, that is, over probability distributions, or over discrete values that may or may not be present in the database. A complete treatment of all the issues that arise when processing queries over this kind of uncertain data is beyond the scope of this paper, but to give a simple example, consider a temperature sensor that produces readings drawn from some distribution. A natural way to represent this in tabular form might be to take some samples from this distribution, to produce a discrete table of samples at different times (assuming each time is independent of the previous time)

| Sensor | Time | Temperature | Probability |
|---|---|---|---|
| 1 | 1:00 | 28 °C | 0.25 |
| 1 | 1:00 | 27 °C | 0.5 |
| 1 | 1:00 | 25 °C | 0.25 |
| 1 | 1:30 | 29 °C | 0.15 |
| 1 | 1:30 | 26 °C | 0.7 |
| 1 | 1:30 | 23 °C | 0.15 |

Though this may look like a standard relational table, there are many queries over it that are not easy to answer in the relational framework. For example, consider the question "At what time did sensor 1 have the highest temperature?" At time 1:30, it has temperature 29 °C with probability 0.15, but at time 1:00, it had temperature 28 °C with probability 0.25, so it might be tempting to produce one of these results. However, it is also quite likely the maximum temperature was neither 29 °C or 28 °C, but something less. Hence, answering this question is actually quite complex.

One popular way to answer these kinds of questions is using so-called "possible world semantics" [14]. The idea is to define all possible values that the maximum could take, and to compute the probabilities for each such possible value. For this query, the possible worlds (along with the probability of each) work out to

| Max(Temp) | Probability |
|---|---|
| 29 °C | 0.15 |
| 28 °C | $(1-0.15)\times0.25=0.22$ |
| 27 °C | $(1-0.15-0.22)\times0.5/0.75=0.42$ |
| 26 °C | $(1-0.15-0.22-0.42)\times0.7/0.85=0.17$ |
| 25 °C | $(1-0.15-0.22-0.42-0.17)\times0.25/0.25=0.04$ |

This means that, for example, the largest value is 29 °C with probability 0.15, and that the "most probable" largest value is 27 °C.

Obviously, answering such queries requires a relatively sophisticated query processor that understands how to manipulate probabilities in a way that a standard query processor does not. Unfortunately, in general, this approach leads to a number of possible worlds that are exponential in the number of input tuples. To see this, suppose, for example, that we had asked for the times when sensor 1's value exceeded 25 °C; if there are $n$ different times in the database, each of which has a nonzero probability of the sensor's value either exceeding or not exceeding 25 °C, then there are $2^n$ possible worlds.

One solution to this problem might just be to eliminate the uncertainty, for example, to replace each reading by its average value. However, this may be nonideal in some applications. If our temperature data are used to sound an alarm when the temperature goes above 28 °C, it may be important to know that there is a 15% chance the temperature is at 29% appropriate action can be taken.

There are several research groups actively engaged in a variety of more sophisticated ways to represent and query uncertain data in database systems, and to deal with queries of the sort described above (see, e.g., [14]–[19], along with many others.) Researchers take different approaches, for example, some seek to use a restricted model of uncertainty to make query processing tractable [16]; others look at continuous distributions and simple approximations of them, instead of discrete values as in the above case [17], [19].

At MIT (along with researchers from Brown University, Providence, RI) we have worked on a small part of this problem, namely the problem of answering top-$k$ queries over uncertain data [20]. These queries are similar to the maximum query given above, except that the answer is a length-$k$ vector instead of just a single result. Clearly, just as there many possible maximum values, there are many possible top-$k$ vectors. In fact, for a large data set, there are exponentially many such vectors (containing essentially every possible permutation of values), which would be too expensive to compute or even to send to the user. Instead, we devised a method to effectively sample this space of top-$k$ vectors to compute $c$ *typical* top-$k$ vectors. Our algorithm is based on dynamic programming; it works by building up top-1 vectors, extending those to be top-2 vectors, and so on, until it has found a set of candidate top-$k$ vectors. The details, along with an evaluation on real data from the CarTel project, are given in [20].

In addition to answering top-$k$ queries, the probabilistic query processor is capable of answering more conventional queries with probabilistic thresholds, e.g., finding all of the sensor readings with probability above some threshold.

## VI. CONCLUSION

In this paper, we discussed the problem of data management in sensor networks, focusing on new problems that have arisen now that the basic sensor networking hardware and software have matured to the point that real applications are being built. In particular, we discussed problems various research groups, and especially our group at MIT, have tackled in this area in the past few years, including the following.

- The problem of prioritizing data that are collected based on application needs. Systems like Lance and the ICEDB system we have developed at MIT allow users to prioritize the data they collect as a part of queries.
- The problem of cleaning data, using models, in a relational database-like framework that provide declarative queries, transactional semantics, and efficient operation over disk resident data, and the FunctionDB system we have built to do this.
- The problem of querying uncertain data, with probabilities, using SQL-like queries, and the range of solutions developed in the database community for processing such queries.

Together, these problems demonstrate the rich set of data management challenges presented by sensor networks, with research challenges in networking, machine learning, databases, and operating systems. ■

## REFERENCES

[1] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *ACM SIGOPS Operat. Syst. Rev.*, vol. 36, no. SI, pp. 131–146, Winter 2002.

[2] Y. Yao and J. Gehrke, "Query processing in sensor networks," in *Proc. Conf. Innovative Data Syst. Res.*, 2003.

[3] R. Müller, G. Alonso, and D. Kossmann, "A virtual machine for sensor networks," in *Proc. EuroSys Conf.*, Lisbon, Portugal, 2007, pp. 145–158.

[4] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proc. 7th ACM Conf. Embedded Net. Sensor Syst.*, Berkeley, CA, Nov. 2009, pp. 1–14.

[5] Y. Zhang, B. Hull, H. Balakrishnan, and S. Madden, "ICEDB: Intermittently connected continuous query processing," in *Proc. Int. Conf. Data Eng.*, 2007, pp. 166–175.

[6] D. Carney, U. Centiemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams—A new class of data management applications," in *Proc. 28th Int. Conf. Very Large Data Bases*, Hong Kong, 2002, pp. 215–226.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Proc. Conf. Innovative Data Syst. Res.*, 2003.

[8] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Data, C. Olston, J. Rosenstein, and R. Varma, "Query processing, approximation and resource management in a data stream management system," in *Proc. Conf. Innovative Data Syst. Res.*, 2003.

[9] G. Werner-Allen, S. Dawson-Haggerty, and M. Welsh, "Lance: Optimizing high-resolution signal collection in wireless sensor networks," in *Proc. 6th ACM Conf. Embedded Net. Sensor Syst.*, Raleigh, NC, 2008, pp. 169–182.

[10] A. Deshpande and S. Madden, "Mauvedb: Supporting model-based user views in database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Chicago, IL, 2006, pp. 73–84.

[11] A. Thiagarajan and S. Madden, "Querying continuous functions in a database system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Vancouver, BC, Canada, 2008, pp. 791–804.

[12] Y. Ahmad, O. Papaemmanouil, U. Çetintemel, and J. Rogers, "Simultaneous equation systems for query processing on continuous-time data streams," in *Proc. IEEE Int. Conf. Data Eng.*, 2008, pp. 666–675.

[13] B. Kanagal and A. Deshpande, "Efficient query evaluation over temporally correlated probabilistic streams," in *Proc. IEEE Int. Conf. Data Eng.*, 2009, pp. 1315–1318.

[14] N. N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," *Int. J. Very Large Data Bases*, vol. 16, no. 4, pp. 523–544, Oct. 2007.

[15] N. Dalvi and D. Suciu, "Answering queries from statistics and probabilistic views," in *Proc. 31st Int. Conf. Very Large Data Bases*, Trondheim, Norway, 2005, pp. 805–816.

[16] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom, "ULDBs: Databases with uncertainty and lineage," in *Proc. 32nd Int. Conf. Very Large Data Bases*, Seoul, Korea, 2006, pp. 953–964.

[17] T. Tran, C. Sutton, R. Cocci, Y. Nie, Y. Diao, and P. J. Shenoy, "Probabilistic inference over RFID streams in mobile environments," in *Proc. IEEE Int. Conf. Data Eng.*, 2009, pp. 1096–1107.

[18] D. Barbará, H. Garcia-Molina, and D. Porter, "The management of probabilistic data," *IEEE Trans. Knowl. Data Eng.*, vol. 4, no. 5, pp. 487–502, Oct. 1992.

[19] P. Sen, A. Deshpande, and L. Getoor, "Representing tuple and attribute uncertainty in probabilistic databases," in *Proc. 7th Int. Conf. Data Mining Workshops*, 2007, pp. 507–512.

[20] T. Ge, S. Zdonik, and S. Madden, "Top-k queries on uncertain data: On score distribution and typical answers," in *Proc. 35th SIGMOD Int. Conf. Manage. Data*, Providence, RI, 2009, pp. 375–388.

## ABOUT THE AUTHOR

**Samuel Madden** received the Ph.D. degree from the University of California at Berkeley, Berkeley, in 2003, where he worked on the TinyDB system for data collection from sensor networks.

He is an Associate Professor of Electrical Engineering and Computer Science at the Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology (MIT), Cambridge. His research interests include databases, sensor networks, and mobile computing. Research projects include the C-Store column-oriented database system, and the CarTel mobile sensor network system.

Dr. Madden was named one of *Technology Review*'s Top 35 Under 35 in 2005, and is the recipient of several awards, including a National Science Foundation (NSF) CAREER Award in 2004, a Sloan Foundation Fellowship in 2007, best paper awards in VLDB 2004 and 2007, and a best paper award in MobiCom 2006.