Beyond Average: Toward Sophisticated Sensing with Queries

Joseph M. Hellerstein[1,2], Wei Hong[2], Samuel Madden[1], and Kyle Stanek[1]

[1] UC Berkeley
{jmh,madden}@cs.berkeley.edu, kyles@uclink.berkeley.edu
[2] Intel Research, Berkeley
whong@intel-research.net

Abstract. High-level query languages are an attractive interface for sensor networks, potentially relieving application programmers from the burdens of distributed, embedded programming. In research to date, however, the proposed applications of such interfaces have been limited to simple data collection and aggregation schemes. In this paper, we present initial results that extend the TinyDB sensornet query engine to support more sophisticated data analyses, focusing on three applications: topographic mapping, wavelet-based compression, and vehicle tracking. We use these examples to motivate the feasibility of implementing sophisticated sensing applications in a query-based system, and present some initial results and research questions raised by this agenda.

1 Introduction

Sensor networks present daunting challenges to potential application developers. Sensornet programming mixes the complexities of both distributed *and* embedded systems design, and these are often amplified by unreliable network connections and extremely limited physical resources. Moreover, many sensor network applications are expected to run unattended for months at a time.

These challenges have motivated research into higher-level programming interfaces and execution environments, which try to relieve programmers from many of the burdens of distributed and embedded programming (e.g. [13, 24]). In our own work, we have designed a framework called TAG [17] for sensornet data aggregation via an SQL-like language. More recently we have implemented the TAG framework in a system called TinyDB [18] that runs in networks of TinyOS-based Berkeley motes [11].

We have received initial feedback indicating that TinyDB's SQL-based interface is very attractive to a number of users interested in distributed sensing. However, we have also heard concerns about apparent limits to the functionality of simple SQL queries. This feedback resulted in part from our early work, which performed fairly traditional SQL queries for relatively simple tasks: periodically collecting raw readings, and computing simple summarizations like averages and counts.

In this paper, we present a status report on our efforts to do deploy more complex sensing tasks in TinyDB. Our intention is both to illustrate TinyDB's potential as a vehicle for complex sensing algorithms, and to highlight some of the unique features and constraints of embedding these sensing algorithms in an extensible, declarative query framework.

In the paper we review and extend the TAG framework [17], and show how it can be used to implement three sensing applications that are relatively distant from vanilla database queries:

1. *Distributed Mapping:* One commonly cited [4] application for sensor networks is to produce contour maps based on sensor readings. We present simple topographic extensions to the declarative query interface of TAG that allow it to efficiently build maps of sensor-value distributions in space. Our approach is based on finding *isobars*: contiguous regions with approximately the same sensor value. We show how such maps can be built using very small amounts of RAM and radio bandwidth, remaining useful in the face of significant amounts of missing information (e.g. dropped data or regions without sensor nodes.) Results from an initial simulation are included.
2. *Multiresolution Compression and Summarization:* Traditional SQL supports only simple aggregates for summarizing data distributions. We develop a more sophisticated wavelet-based aggregation scheme for compressing and summarizing a set of readings. Our technique also has the ability to produce results of increasing resolution over time. We describe a hierarchical wavelet encoding scheme that integrates naturally into the standard TAG framework, and is tuned to low-function devices like Berkeley motes. We also discuss a number of open research questions that arise in this context.
3. *Vehicle Tracking:* Several research papers have investigated distributed sensornet algorithms that track moving objects [2]. We show how a declarative, event-based query infrastructure can serve as a framework for such algorithms, and discuss how the TAG approach can be extended to allow sensor nodes to remain idle unless vehicles are near to them. This is work in progress: we have yet to instantiate this infrastructure with a sophisticated tracking algorithm, but hope that this framework will seed future efforts to combine intelligent tracking with the other ad-hoc query facilities afforded by a full-function sensornet query process like TinyDB.

The remainder of this paper is organized as follows: Section 2 summarizes the TAG approach to in-network aggregation, which we extend in the remaining sections of the paper. Section 3 discusses some new language features we have added since the publication of TAG[17], which enable our tracking scenario. Section 4 discusses the distributed mapping problem; Section 5 discusses techniques for compressing and summarizing; and Section 6 discusses vehicle tracking. Finally, Section 7 discusses related work and Section 8 concludes with some discussion of future work.

2 Background

In this section, we describe the declarative, SQL-like language we have developed for querying sensor networks. We also describe the processing of queries in a sensor network, with a focus on aggregation queries.

2.1 A Query Language for Sensor Networks

TAG [17] presented a simple query language for sensor networks, which we have implemented in TinyDB [18]. We present a basic overview of the scheme here. In TinyDB, queries are posed at a powered *basestation*, typically a PC, where they are parsed into a simple binary representation, which is then flooded to sensors in the network.

As the query is flooded through the network, sensors organize into a *routing tree* that allows the basestation to collect query results. The flooding works as follows: the basestation injects a query request at the *root* sensor, which broadcasts the query on its radio; all *child* nodes that hear the query process it and re-broadcast it on to their children, and so on, until the entire network has heard the query[3].

[3] Schemes to prune the query flooding process are presented in [16].

Each request contains a hop-count (or *level*), indicating the distance from the broadcaster to the root. To determine their own level, nodes pick a *parent* node that is (by definition) one level closer to the root than they are. This parent will be responsible for forwarding the node's query results (and its children's results, recursively) to the basestation. Also note that each node may have several possible choices of parent; for the purposes of our discussion here, we assume that a single parent is chosen uniformly and at random from the available parents. In practice, more sophisticated schemes can be used for parent selection, but this issue will not impact our discussion here.

Queries in TinyDB have the following basic structure:

```
SELECT expr1, expr2, ...
  FROM sensors
  WHERE pred1 [AND | OR] pred2 ...
  GROUP BY groupExpr1, groupExpr2, ...
  SAMPLE PERIOD t
```

The `SELECT` clause lists the fields (or *attributes*) to retrieve from the sensors; $expr_n$ specifies a transform on a single field. Each transform may be a simple arithmetic expression, such as `light + 10`, or an aggregate function, which specifies a way in which readings should be combined across nodes or over time (aggregation is discussed in more detail in the following section.) As in standard SQL, aggregates and non-aggregates may not appear together in the `SELECT` clause unless the non-aggregate fields also appear in the `GROUP BY` clause.

The `FROM` clause specifies the table from which data will be retrieved; in the language presented in [17], there is only one table, `sensors`, which contains one attribute for each of the types of sensors available to the devices in the network (e.g. light, acceleration, or temperature). Each device has a small *catalog* which it uses to determine which attributes are locally available; the catalog also includes cost information and other metadata associated with accessing the attribute, and a pointer to a function that allows TinyDB to retrieve the value of the attribute.

The (optional) `WHERE` clause filters out readings that do not satisfy the boolean combination of predicates. Predicates in TinyDB are currently restricted to simple boolean and arithmetic operations over a single attribute, such as `light / 10 > 25`.

The (optional) `GROUP BY` clause is used in conjunction with aggregate expressions. It specifies a partitioning of the input records before aggregation, with aggregates in the `SELECT` clause being computed on each partition. In the absence of a `GROUP BY` aggregates are computed over the entire set of sensors; a `GROUP BY` partitions the sensors into groups whose group expressions each have the same value. For example, the query fragment:

```
SELECT roomNumber, AVG(light)
  GROUP BY roomNumber
  ...
```

partitions sensors into groups according to the value of the `roomNumber` attribute, and computes the average light reading within each group.

Finally, the `SAMPLE PERIOD` clause specifies the time between successive samples or *epochs*. Each node samples its sensors once per epoch and applies its query processing operators to that sensor.

2.2 Aggregation in Sensor Networks

Given this basic description of the query language, we now discuss how TinyDB processes queries, focusing on how aggregate queries are handled.

Structure of Aggregates Recall that an aggregation expression may be specified in the SELECT clause of a query. In standard SQL, that expression contains one of a few basic aggregation functions: MIN, MAX, AVERAGE, COUNT, or SUM. As in TAG, TinyDB provides an *extensible* mechanism for registering new aggregates, derived from literature on extensible database languages. In TinyDB, aggregates are implemented via three functions: a merging function f, an initializer i, and an evaluator, e. In general, f has the following structure:

$$< z >= f(< x >, < y >)$$

where $< x >$ and $< y >$ are multi-valued *partial state records* (PSRs), computed over one or more sensor values, representing the intermediate state of the aggregation processing based on those values. $< z >$ is the partial-state record resulting from the application of function f to $< x >$ and $< y >$. For example, if f is the merging function for AVERAGE, each partial state record will consist of a pair of values: SUM and COUNT, and f is specified as follows, given two state records $< S_1, C_1 >$ and $< S_2, C_2 >$:

$$f(< S_1, C_1 >, < S_2, C_2 >) =< S_1 + S_2, C_1 + C_2 >$$

The initializer i is needed to specify how to instantiate a state record for a single sensor value; for an AVERAGE over a sensor value of x, the initializer $i(x)$ returns the tuple $< x, 1 >$. Finally, the evaluator e takes a partial state record and computes the actual value of the aggregate. For AVERAGE, the evaluator $e(< S, C >)$ simply returns S/C.

Processing Aggregate Queries Aggregate queries produce one result per group per epoch. Once a query has been disseminated as described above, each leaf node in the routing tree produces a single tuple of sensor readings each epoch, applies the initialization function to the approriate column, and forwards the initialized result to its parent. On the next epoch, the parent merges its own PSR from the previous epoch with PSRs from its children in the previous epoch, and forwards that result on to its parent. Results propagate up the tree, epoch-by-epoch, until a complete PSR from d epochs ago arrives at the root of the routing tree (where d is the depth of the tree). Depending on the sample period, there may be enough time in each epoch to send aggregates up multiple levels of the routing tree; see TAG[17] for more information. Once a result has arrived at the root, the basestation applies the evaluation function to it to produce a complete aggregate record and outputs the result to the user.

GROUP BY queries are processed in a similar way. When a partial state record is initialized, it is tagged with a unique group ID. Parents combine readings if they come from the same group, and propagate a separate PSR for each group they heard about during the previous epoch (even if they did not contribute to one or more of the groups.) Thus, at the root, one state record arrives for each of the n groups, and the evaluation function is applied n times.

Temporal Aggregates All the aggregates that we described above aggregate sensor values sampled from multiple nodes at the same epoch. We have extended this framework to support *temporal aggregates* which aggregate sensors values across multiple consecutive epochs from the same or different nodes. Temporal aggregates typically take two extra arguments: *window size* and *sliding distance*. Window size specifies the number of consecutive epochs the temporal aggregate operates on, and the sliding distance specifies the number of epochs to skip over for the next window of samples. One frequently used temporal aggregates in TinyDB is the running average aggregate `winavg(window_size, sliding_dist, arg)`. It is typically used to reduce noise in sensor signals. For example, `winavg(10, 1, light)` computes the 10-sample running average of light sensor readings. It accumulates light readings from 10 consecutive epochs, averages them, then replaces the oldest value in the average

window with the latest light sensor reading and keeps on computing averages over the window of samples. In addition to `winavg`, TinyDB also supports similar temporal aggregates such as `winmin`, `winmax`, `winsum`, etc. More sophisticated custom temporal aggregates such as one that computes the trajectory of a moving vehicle can be developed using the same extensible aggregate framework described above.

In TinyDB, by default all aggregates operate over sensor values from the entire network. `GROUP BY nodeid` must be specified to limit temporal aggregates to operate on values from individual nodes only.

3 New Language Features

The query language described above provides a foundation for many kinds of simple monitoring queries. However, as sensor networks become more autonomous, the language needs to move beyond passive querying: rather than simply monitoring the environment and relaying results, the sensors will need to detect and initiate automatic responses to nearby events. Furthermore, sensor networks will need to collect and store information locally, since it is not always possible or advantageous to get data out of the network to a powered, storage-rich PC. We introduce two extensions to our query language to handle these situations.

3.1 Events

Events provide a mechanism for initiating data collection in response to some external stimulus. Events are generated explicitly, either by another query, by software in the operating system, or by specialized hardware on the node that triggers the operating system. Consider the following query for monitoring the occupancy of bird nests:

```
ON EVENT bird-detect(loc):
  SELECT AVG(light), AVG(temp)
  FROM sensors AS s
  WHERE dist(s.loc, event.loc) < 10m
  SAMPLE INTERVAL 2 s FOR 30 s
```

When a bird is detected in a nest (e.g. via a pressure switch in the nest), this query is executed to collect the average light and temperature level from sensors near the nest, and send these results to the root of a network. (Alternatively, the results could be stored locally at the detecting node, using the storage point mechanism described in the next section.) The semantics of this query are as follows: when a `bird-detect` event occurs, the query is issued from the detecting node and the average light and temperature are collected from nearby nodes (those nodes that are 10 or less meters from the collecting node) every 2 seconds for 30 seconds.

3.2 Storage Points

Storage points accumulate a small buffer of data that may be referenced in other queries. Consider, as an example:

```
CREATE
  STORAGE POINT recentlight SIZE 5s
  AS (SELECT nodeid, light
  FROM sensors
  SAMPLE INTERVAL 1s)
```

This `STORAGE POINT` command provides a shared global location to store a streaming view of recent data, similar to materialized views in conventional databases. Note that this data structure is accessible for read or write from any node in the network; its exact location within the network is not fixed – that is, it can be moved as an optimization. Typically, these storage points are partitioned by `nodeid`, so that each sensor stores its own values locally.

The specific example here stores the previous five seconds worth of light readings (taken once per second) from all of the nodes in the network.

In this paper, we use storage points as a mechanism for storage and offline delivery of query results. Queries that select all of the results from a storage point, or that compute an aggregate of a storage point, are allowed; consider, for example:

```
SELECT MAX(light)
  FROM recentLight
```

This query selects the maximum light reading from the `recentLight` storage point defined above. The storage point is continually updated; this query returns the maximum of the values at the time the query is posed.

4 Isobar Mapping

In this section, we explore the problem of building a topographic (contour) map of a space populated by sensors. Such maps provide an important way to visualize sensor fields, and have applications in a variety of biological and environmental monitoring scenarios [4]. We show how TinyDB's aggregation framework can be leveraged to build such maps. Conceptually, the problem is similar to that of computing a `GROUP BY` over both space and quantized sensor readings – that is, our algorithms partition sensors into *isobars* that are contiguous in space and approximately equal in sensor value. Using in-network aggregation, the storage and communication costs for producing a topographic map are substantially less than the cost of collecting individual sensor readings and building the map centrally. We discuss three algorithms for map-building: a centralized, *naive* approach, an exact, *in-network* approach, and an approximate, *lossy* approach. Figure 1 illustrates the general process of aggregation to build a topological map: each sensor builds a small representation of its local area, and sends that map to its parent, where it is combined with the maps from neighbors and ancestors and eventually becomes part of a complete map of the space at the root of the tree.

To support topographic operations on sensors, we require a few (very) simple geometric operators and primitives. To determine adjacency in our maps, we impose a rectangular grid onto the sensors, and assign every sensor into a cell in that grid. Our goal is to construct isobars, which are orthogonal polygons with holes; we need basic operations to determine if two such polygons overlap and to find their union. Such operations can be performed on any polygon in $nlog(n)$ time (where n is the number of edges in the polygon) using the Leonov-Nitkin algorithm [15]. There are a number of free libraries which implement such functionality [14].

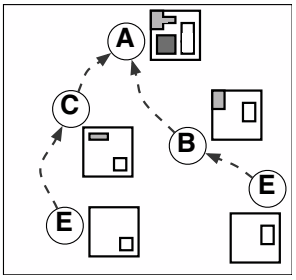


Fig. 1. Aggregation of contours as data flows up a routing tree.

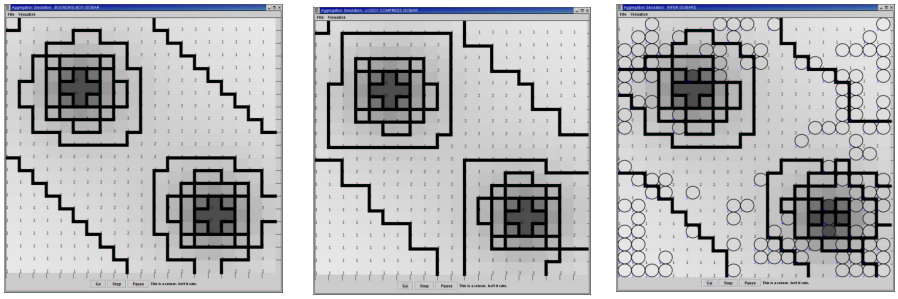
We begin with a discussion of the three algorithms, assuming that every cell in the grid is occupied. We reserve a discussion of mapping sparse grids for Section 4.4.

4.1 Naive Algorithm

In the naive algorithm, we run an aggregate-free query in the network, e.g.:

```
SELECT xloc, yloc, attr
  FROM sensors
  SAMPLE PERIOD 1s
```

This query returns the location and attribute value of all of the sensors in the network; these results are combined via code outside the network to produce a map. We implemented this



(a) Lossless Isobars (b) Lossy Isobars (c) Missing Information

Fig. 2. Screenshots of a visualization of isobars imposed on a grid of sensors. Each cell represents a sensor, the intensity of the background color indicates sensor value, and black lines frame isobars. (a) shows the isobars as computed by the in-network and naive algorithms. (b) shows a lossy approximation of isobars, where each approximate polygon is the bounding box of original polygon with 4 maximally sized "cuts". (c) shows a visualization of a topological map with incomplete information about all of the squares. Black circles indicate nodes whose value was missing and was inferred by the algorithm.

approach in a simulation and visualization, as shown in figure Figure 2(a). In this first simulation, sensors were arranged in a grid, and could communicate losslessly with their immediate neighbors. The isobars were aggregated by the node at the center of the network. The network consisted of 400 nodes in a depth 10 routing tree. In the screenshot, the saturation of each grid cell indicates the sensor value, and the thick black lines show isobars.

4.2 In-Network Algorithm

In the in-network approach, we define an aggregate, called *contour-map* where each partial state record is a set of isobars, and each isobar is a container polygon (with holes, possibly) and an attribute value, which is the same for all sensors in the isobar. The structure of an isobar query is thus:

```
SELECT contour-map(xloc,yloc,floor(attr/k))
  FROM sensors
```

where k defines the width (in attribute-space) of each of the isobars. We can then define the three aggregation functions, i, f, and e, as follows:

- i: The initialization function takes an `xloc, yloc`, and `attr`, and generates as a partial state record the singleton set containing an isobar with the specified `attr` value and a container polygon corresponding to the grid cell in which the sensor lies.
- f: The merging function combines two sets of isobars, I_1 and I_2 into a new isobar set, I_3, where each element of I_3 is a disjoint polygon that is the union of one or more polygons from I_1 and I_2. This new set may have several non-contiguous isobars with the same attribute value. Conversely, merging can cause such disjoint isobars in I_1 to be joined when an isobar from I_2 connects them (and vice-versa.) Figure 3 shows an example of this happening as two isobar sets are merged together.
- e: The evaluation function generates a topographic map of contiguous isobars, each labeled with their attribute value.

4.3 Lossy Algorithm

The lossy algorithm works similarly to the in-network algorithm, except that the number of vertices v used to define the bounding polygon of each isobar is limited by a parameter of the

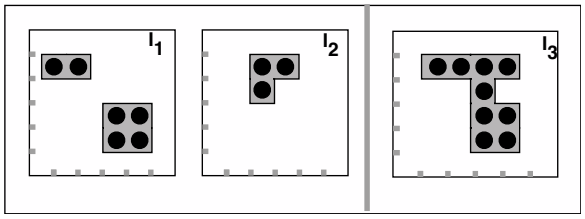


Fig. 3. Two isobar sets, I_1 (with two elements) and I_2 (with one element) being merged into a new isobar set, I_3 (also with one element).

aggregate. This reduces the communication cost of the approach, but makes it possible for isobars to overlap, as they will no longer perfectly trace out the edges of the contours.

In the lossy algorithm, i is the same as in the in-network case. For f, we compute I_3 as above, but we do not use it as the partial state record. Instead, for the containing polygon p in each set of I_3, we compute a bounding box, b, and then take from b a number of maximally sized rectangular "cuts" that do not overlap p. We continue taking cuts until either b contains v vertices, or the next cut produces a polygon with more than v vertices. We omit the details of how we compute maximal cuts; because our polygons are orthogonal, this can be done via a scan of the vertices of p. We use these cut-bounding-boxes as approximations of the containing polygons in the isobars of the PSRs resulting from our merge function. Figure 4 shows a containing polygon approximated by a bounding rectangle with a single cut.

In the lossy evaluation function e, one or more isobars in the final aggregate state record may overlap, and so some policy is needed to choose which isobar to assign to a particular cell. We use a simple "containment" principle: if one isobar completely contains another, we assume the true value of the cell is that specified by the innermost isobar. When the containment principle does not apply, we assign grid cells to the nearest isobar (in terms of number of grid cells), breaking ties randomly.

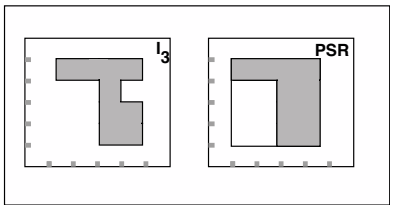


Fig. 4. A lossy approximation of a containing polygon (I_3) as a bounding box with a single cut (PSR).

We simulated this lossy algorithm for the same sensor value distribution as was shown in Figure 2(a), using a maximum of 4 "cuts" per isobar. The results are shown in Figure 2(b); notice that the shape of the isobars is preserved.

We compared the total amount of data transmitted by our simulation of the lossy, in-network, and naive algorithms for the isobars shown in Figure 2(a) and 2(b), and found that the naive algorithms algorithm used a factor of four more communication than the lossy algorithm and about 40% more communication than the in-network algorithm.

4.4 Sparse Grids

Finally, we consider the case of sparse grids, where sensors do not exist at every cell in the grid. In sparse grids, the lossy algorithm described above can be used to infer an isobar for missing points. Since the merging function no longer tracks exact contours but uses bounding boxes, cells without sensors will often end up as a part of an isobar. Cells that aren't assigned an isobar as a part of merging can be assigned using the nearest-isobar method described in the lossy algorithm.

A similar situation arises in dense topologies with network loss, when some sensor values are not be reported during a particular epoch. We implemented this sparse grid approach

and used it to visualize isobars with a high-loss radio model, where the probability that two sensors can communicate with each other falls off with the distance between the sensors. For adjacent sensors, loss rates are about 5%; for sensors that are three cells away (the maximum communication range), loss rates are about 20%. The result is shown in Figure 2(c), with black circles on the nodes whose values were lost during the epoch being visualized. Notice that, despite the large number of losses, the shape of the isobars is largely preserved.

5 Wavelet Histograms via Hierarchical Aggregation

SQL's built-in aggregates provide some basic statistical information about the distribution of a set of readings. But in many cases it is useful to get a richer representation of the distribution, e.g. a histogram. In the sensornet environment, we would like to have a *multiresolution* histogram, which can optionally provide additional resolution of "buckets" at the expense of additional communication. To that end, we explore using *wavelet histograms*, since wavelets are one of the best-known and most effective multiresolution coding techniques.

In this section, we sketch a TAG aggregate function for encoding a set of readings in a sensor network using Haar wavelets, the simplest and most widely-used wavelet encoding[4]. Our discussion here focuses on *wavelet histograms* [20], which capture information about the statistical distribution of sensor values, without placing significance on any ordering of the values. We drop coefficients with low absolute values ("threshholding") to keep the communication costs down, but always retain the value of coefficient 0; in Haar wavelets, the 0th coefficient represents the average of the values, and hence is often of interest to users.

Our wavelet compression setting here is somewhat unique. First, recall that aggregates in TinyDB are computed incrementally as data is passed up the network communication tree. Hence we will be computing wavelets piecewise, combining pairs of wavelets as we go, without access to the complete underlying set of values. Second, our processors do not have floating-point arithmetic and are generally rather slow, so we will use integer wavelets [1], do as much as possible *in place* to minimize copies, and devise techniques to process wavelets without the need to decompress/recompress. Finally, since we are constrained in both memory and bandwidth, we will be dropping low coefficients, and using a sparse array representation for the coefficients we keep.

The core of our logic is in the merging function f, which takes the PSRs from two subtrees (which are themselves wavelets) and combines them into a new PSR (another wavelet). Our wavelet PSR will be a sparse array represented by $2N + 2$ short integers. In order to maintain wavelet properties, N must be a power of 2 ($N = 8$ in our current implementation.) The first $2N$ values capture the non-zero elements of the sparse array: N array `offsets`, and N `data` values of the coefficients at those offsets. The next short integer is the `count`, which tracks the number of actual sensor readings rolled up into this wavelet. One additional short, called `loglen`, represents the $\log_2$ of the number of array entries in the (possibly zero-padded) wavelet.

The merging function considers 4 cases for merging two state records, `r1` and `r2`[5]

1. `r1.count` + `r2.count` $< N$: In this case, we do not compress, but simply store all the values. We concatenate the values from `r2.data` to the end of `r1.data`, and

[4] In the interest of brevity, we do not overview wavelets here; the interested reader is referred to [23] for a good practical overview of wavelets, or to [20] for a simple introduction to Haar wavelets.

[5] Note that the choice of ordering `r1` before `r2` is rather arbitrary: for now, we assume that the network topology and scheduling determines which input is first, and which is second.

update the `offsets` and `count` of `r1` accordingly. The `loglen` variable remains at the initialization value of $\log_2 N$.

2. `r1.count` $< N$ *and* `r2.count` $< N$, *but their sum is* $> N$: In this case we need to compress the output. Conceptually, we think of the two input arrays as one array of length $2^{\texttt{loglen}+1}$, and use the *lifting* scheme [22] to wavelet-compress the double-length array in place. We then keep only the top N coefficients by overwriting `r1`'s `data` and `offsets` fields appropriately. We add `r2.count` to the value in `r1.count`, and increment the `r1.loglen` variable to reflect the effective doubling of the array.
3. *Both inputs have* `count` $> N$: In this case, we need to merge two wavelets.
   Our merge technique will assume that both inputs have the same `loglen`. If one input has a smaller `loglen` than the other, we need to zero-pad the smaller to match the larger. For example, if `r1.loglen` $<$ `r2.loglen`, we zero-pad `r1` until it is of equal length. Pseudocode for efficiently doubling a Haar wavelet with 0's is given in Figure 5.
   Once both wavelets have the same `loglen`, we need to merge `r2` into `r1` to form a wavelet with twice as many coefficients. We then run the pseudocode given in Figure 6 to merge `r2` into `r1` without decoding and re-encoding. Finally we copy the top N coefficients of the result into `r1.data`, update `r1`'s `offsets` appropriately, add `r2.count` to `r1.count`, and increment `r1.loglen` to reflect the doubling of the array.
4. *Exactly one input has* `count` *larger than* N: In this case, we zero-pad the smaller array to be N entries, and convert it to a wavelet of N coefficients. Then we invoke Case 3.

```
// for all coefficients i except 0th, bump up offsets carefully
for i from 1 to N-1
    offsets[i] += 2^(floor(log_2(offsets[i])));
    // keep track of min coefficient, too
    if (abs(data[i]) < min) then { min = abs(data[i]);  minpos = i; }
// New 1st coefficient is 0 - (old 0th coefficient).
// If it's in the top N, make room for it in data and offsets arrays
if (abs(data[0]) > min)
    move offsets[1] through offsets[minpos-1] one position rightward;
    move data[1] through data[minpos-1] one position rightward;
    offsets[1] = 1; data[1] = 0 - data[0];
// overall average is halved, reflecting the 0-padding.
data[0] >>= 2; // i.e. data[0] = floor(data[0] / 2);
loglen++; // we doubled the size
```

Fig. 5. Double a Haar wavelet of N coefficients by zero-padding in place, without decoding/recoding. The result should have $N + 1$ coefficients; we drop the lowest of these other than the 0th coefficient, which we always keep in position 0 in the arrays. Note that $\lfloor (log_2(n)) \rfloor$ can be computed efficiently via bit-shifting, and that we use the *floor* of the average for data[0] in integer wavelets [1].

At the top of the aggregation tree, this technique produces a wavelet that lossily represents the concatenation of all the readings in the network, along with a large number of padded 0's. Given the `count` and `loglen` variables, a PC at the root of the network can discard the extraneous 0's, and perform the appropriate normalization to recreate both the overall average, and somewhat finer approximations of the densities of values.

Note that the coefficients produced by the recursive application of the merge procedure are not the top N coefficients of a Haar wavelet on the full array of readings. In particular, the $N + 1$'st coefficient of one network subtree will be discarded even though it may be much larger than the top N coefficients of another subtree. The effect of such an error may be spread across higher-order coefficients as further merges happen. We are investigating heuristics for improving this situation, including probabilistic updating schemes from [20], rank-merging techniques based on [5], and coefficient confidence intervals based on [7].

```
// Double r1 and r2, but bump r2 rightward by an extra factor of 2.
for i from 1 to N-1
    r1.offsets[i] += 2^(floor(log_2(r1.offsets[i])));
    r2.offsets[i] += 2^(floor(log_2(r2.offsets[i])) + 1);
// merge r1's {offsets,data} pairs with r2's, sorted by offset
cursor1 = 1; cursor2 = 1;
for k from 2 to (N*2) - 1
    if (cursor1 < N && r1.offsets[cursor1] <= r2.offsets[cursor2])
         { smaller = r1; curs = cursor1; }
    else { smaller = r2; curs = cursor2; }
    wtmp.offsets[k] = smaller.offsets[curs];
    wtmp.data[k] = smaller.data[curs];
    curs++;
// 0th coefficient of wtmp is avg of old 0-coefs, 1st is diff
wtmp.offsets[0] = 0; wtmp.offsets[1] = 1;
wtmp.data[0] = floor((r1.data[0] + r2.data[0])/2);
wtmp.data[1] = r1.data[0] - r2.data[0];
// pack top N coefficients of wtmp into first N slots
// of wtmp.data, update wtmp.offsets appropriately,
topN_coeffs(wtmp);
copy N wtmp.{data,offsets} into r1.{data,offsets}
r1.count += r2.count;   r1.loglen++;
```

Fig. 6. Given two Haar wavelets `r1` and `r2` of N non-zero coefficients, merges them without decoding/recoding. We copy to a temporary wavelet `wtmp` of size $2N$, but this logic can also be done in place by replacing the merge step with a quicksort that spans r1 and r2 and coordinates the data and offset arrays appropriately.

5.1 MultiResolution Snapshots, Temporal Queries

In the spirit of image coding and *online aggregation* in databases [9, 10], we might want the answer to a snapshot query to improve with additional rounds of communication. In order to achieve this, we can augment the logic above so that at the lowest point in the tree where the merge function would have dropped coefficients, it sends the second highest set of N coefficients on round 2. At the top of the tree, the second round of coefficients needs to be merged into the previous coefficients from right to left in order to spread the updates correctly. This process can be repeated for additional rounds. In this scheme, the low valued coefficients can either be stored, or can be recommunicated and recomputed from the base snapshot readings. Given the relative costs of storage and communication in modern sensor networks, we expect to store the coefficients – in practice, storage limitations will dictate a bound on the number of rounds we can support.

Multiresolution snapshot queries are complicated when we consider change in the time dimension. Online aggregation as described in [9] is targeted at traditional databases, where snapshot semantics are guaranteed via transactional mechanisms. Since online aggregation requires multiple rounds, it is quite possible that the sensor readings will change before much data can be propagated to the output.

Continuous queries with time-varying results are supported in TinyDB by buffering the state of aggregates from multiple epochs within the network, and delivering better estimations for prior epochs alongside new estimations [17]. However, this increases the storage overhead in the network by a factor of the depth of the network.

We are exploring ideas for intelligently managing the total storage across both time and space. The mix of multiresolution results and time-varying data raises a number of questions with respect to both the encoding (which may be analogous to work on video), and to human-computer issues and performance metrics. A driving question for performance metrics may be to consider different possible interfaces for users to specify their desires by fixing resources in one or both dimensions. Of course, in principle there is some pareto-optimal set of strategies across these dimensions, but naive users are unlikely to be able to reason in that fashion.

One can imagine fairly natural temporal controls like "animation speed" sliders and spatial controls in terms of visual selection, zoom, or foveation. One can also imagine that the dependency across dimensions could be demonstrated by having adjustments in one dimension be reflected in the controls of the other dimension. We hope to explore these inter-disciplinary issues in future work.

6 Vehicle Tracking

In this section, we provide a rough illustration of TinyDB's support for a vehicle tracking application, where a fixed field of nodes detects the magnetic field, sound, or vibration of a vehicle moving through them. We choose the tracking application because it is a representative Collaborative Signal Processing (CSP) application for sensor networks and because it demonstrates the relative ease with which such applications can be expressed in TinyDB. As will become clear, our focus to date has not been on sophisticated algorithms for tracking, but rather on extending our platform to work reasonably naturally for collaborative signal processing applications.

Target tracking via a wireless sensor network is a well-researched area [2]. There are different versions of the tracking problem with varying degrees of complexities. For ease of illustration, in our discussion we only deal with a very simple version of the tracking problem, based on the following assumptions and constraints:

- There is only a single target to track.
- The target is detected when the running average of the magnetometer sensor readings go over a pre-defined threshold.
- The target location at any point in time is reported as the node location with the largest running average of the sensor reading at that time.
- The application expects to receive a time series of target locations from the sensor network once a target is detected.

We believe that more sophisticated versions of tracking can also be supported in TinyDB, using more sophisticated signal processing logic for dynamic threshold adjustment, signal strength based localization, multiple targets, etc.

There are some clear advantages to implementing tracking applications on top of TinyDB. First, TinyDB's generic query language is available as a resource, allowing applications to mix and match existing spatial-temporal aggregates and filters in a query. Applications can also run multiple queries in the sensor network at the same time, for example one tracking query and one network health monitoring query. Second, TinyDB takes care of many of sensor-network systems programming issues such as multi-hop routing, coordination of node sleeping, query and event diseminations, etc. Third, by registering tracking subroutines as user-defined aggregates in TinyDB, they become reusable in other TinyDB queries in a natural way. Fourth, we are optimistic that TinyDB's query optimization techniques [17] can benefit tracking queries. For example, each node can "snoop" the messages from its neighboring nodes and suppress its output if any neighbor has detected a stronger sensor signal.

We will describe below two implementations of the tracking application in TinyDB with increasing levels of query complexity for better energy efficiency. We describe these implementations in TinyDB's SQL-like query language[6]. In all the TinyDB SQL statements, mag is a TinyDB attribute for the magnetometer reading, $time$ is an attribute that returns the current timestamp as an integer. We assume the sensor nodes are time synchronized within 1

[6] Some of the language features used in this section are not available in TinyDB 1.0.

millisecond using protocols like [3]. $nodeid$ is a TinyDB attribute for the unique identifier of each node. We assume the target is detected when the magnetometer reading goes over a constant value, *threshold*. $winavg(10, 1, mag)$ is for the 10-sample running average for the magnetometer readings. $max2(arg1, arg2)$ is another TinyDB aggregate that returns the value of $arg2$ corresponding to the maximum value of $arg1$. $max2(avgmag, nodeid)$ is used in our implementations to find the $nodeid$ with the largest average magnetometer reading. As mentioned above, we use this to represent the location of our target and assume that the basestation is capable of mapping $nodeid$ to some spatial coordinate. $max2$ is really a place holder that can be replaced with much more sophiscated target localization aggregates. In both implementations, we need to apply $max2$ to group of values with the same timestamp. Values are grouped by $time/10$ to accomodate minor time variations between nodes.

6.1 The Naive Implementation

Figure 7 shows the TinyDB queries that implement our initial tracking application. In this implementation, each sensor node samples the magnetometer every 100 miliseconds and computes the 10-sample running average of the magnetometer readings. If the running average of magnetometer readings is over the detection threshold, the current time, nodeid and average value of the magnetometer are inserted into the storage point $running\_avg\_sp$.

Recall that storage points in TinyDB provide temporary in-network storage for query results and facilitate applications to issue nested queries. The second query in Figure 7 is a query that runs over the storage point $running\_avg\_sp$ every second and computes the target locations using the $max2$ aggregate.

6.2 The Query-Handoff Implementation

The problem with the naive implementation is that all sensor nodes must continuously sample the magnetometer every 100 milliseconds, and magnetometers typically consume substantial power per sample. For example, the magnetometer on the Berkeley motes consumes 15mW of power per sample while the light sensor only consumes 0.9mW per sample. Assuming the sensor nodes are spread over a wide area, at every point in time, the target can only be detected by a small number of nodes. Thus, for a large percentage of nodes, the energy spent sampling the magnetometer is wasted.

Ideally, we would like to only start the target tracking query on a node when the target is near it and stop the query when the target moves away. This means that we need a TinyDB event to trigger the tracking query. The query-handoff implementation that we are about to describe requires some special standalone hardware such as a motion detector that detects the possible presence of the target, interrupts the mote processor, and pulls it out of sleep mode. $target\_detected$ is the TinyDB event corresponding to this external interrupt. It is unrealistic to require this special hardware be installed with every node. However it might be feasible to only install them on a small number nodes near the possible entry points for the target to enter the sensor field (e.g. endpoints of a line of sensors along a road). These nodes will be woken up by the $target\_detected$ event and start sampling the magnetometer to determine the current target locations. At the same time, they also try to predict the possible locations the target may move to next via a custom aggregate $next\_location$ and signal a remote event $target\_approaching$ on nodes at these locations to alert them to start sampling their magnetometers and tracking the incoming target. Nodes that receive the $target\_approaching$ event will basically do the same. The TinyDB queries for this implementation is shown in Figure 6.2. We call this the *query-handoff* implementation because the node hands the tracking queries off from one set of nodes to another set of nodes following the target movement.

Query handoff is probably the most unique query processing feature required by tracking applications, and one that at first we expected to provide via low-level network routing infrastructure. However, we were pleased to realize that event-based queries and storage points allow handoff to be expressed reasonably simply at the query language level. This bodes well for prototyping other application-specific communication patterns as simple queries. An ongoing question in such work will be to decide when these patterns are deserving of a more efficient, low-level implementation inside of TinyDB.

```
// Create storage point holding 1
// second worth of running average
// of magnetometer readings with a
// sample period of 100 miliseconds
// and filter the running average with
// the target detection threshold.
CREATE STORAGE POINT running_avg_sp
SIZE 1s AS
(SELECT time,
  nodeid,
  winavg(10, 1, mag) AS avgmag
FROM sensors
GROUP BY nodeid
HAVING avgmag > threshold
SAMPLE PERIOD 100ms);

// Query the storage point every second
// to compute target location for
// each timestamp.
SELECT time, max2(avgmag, nodeid)
FROM running_avg_sp
GROUP BY time/10
SAMPLE PERIOD 1s;
```

Fig. 7. Naive Implementation

```
// Create an empty storage point
CREATE STORAGE POINT running_avg_sp
SIZE 1s (time, nodeid, avgmag);

// When the target is detected, run query to
// compute running average.
ON EVENT target_detected DO
SELECT time, nodeid, winavg(10, 1, mag) AS avgmag
INTO running_avg_sp
FROM sensors group by nodeid
HAVING avgmag > threshold
SAMPLE PERIOD 100ms
UNTIL avgmag <= threshold;

// Query the storage point every
// sec. to compute target location;
// send result to base and signal
// target_approaching to the possible
// places the target may move next.
SELECT time, max2(avgmag, nodeid)
FROM running_avg_sp GROUP BY time/10
SAMPLE PERIOD 1s
OUTPUT ACTION
SIGNAL EVENT target_approaching
WHERE location IN
(SELECT next_location(time, nodeid, avgmag)
      FROM running_avg_sp ONCE);

// When target_approaching event is
// signaled, start sampling and
// inserting results into the storage point.
ON EVENT target_approaching DO
SELECT time, nodeid, winavg(8, 1, mag) AS avgmag
INTO running_avg_sp
FROM sensors GROUP BY nodeid
HAVING avgmag > threshold
SAMPLE PERIOD 100ms
UNTIL avgmag <= threshold;
```

Fig. 8. Handoff Implementation

7 Related Work

Several groups have proposed high-level or declarative interfaces for sensor-networks [24, 17, 13]. There has also been some work on aggregation-like operations in sensor networks, such as [25, 12, 8]. Neither of these bodies of work specifically addresses any of the more sophisticated types of agggreates or queries we discuss in this paper.

Building contour maps is a frequently mentioned target application for sensor networks; see, for example, [4], though, to our knowledge, no one has previously described a viable algorithm for constructing such maps using sensor networks. There is a large body of work on building contour maps in the image processing and segmentation literature – see [19] for an excellent overview of the state of the art in image processing. These computer vision algorithms are subtantially more sophisticated than those presented here, but assume a global view where the entire image is at hand.

Wavelets have myriad applications in data compression and analysis; a practical introduction is given in [23]. Wavelet histograms have been proposed for summarizing database tables in a number of publications, e.g. [20, 7]. In the sensor network environment, a recent short position paper proposed using wavelets for in-network storage and summarization [6]. This work is related to ours in spirit, but different in focus at both the system architecture and coding level. It sketches a routing-level approach for relatively power-rich devices, focused on encoding regularly-gridded, spatial wavelets over timeseries. By contrast, we focus on highly-constrained devices, and integrate with the multi-purpose TinyDB query execution framework. We also provide efficient algorithms for hierarchically encoding Haar wavelets, with a focus on wavelet histograms.

The query handoff implementation for the tracking application in Section 6 is based on the single-target tracking problem discussed in [2]. The tracking algorithms described in [2] is implemented on top of UW-API [21] which is a location-centric API for developing collaborative signal processing applications in sensor networks. UW-API is implemented on top of Directed Diffusion [13] focusing on routing of data and operations based on dynamically created geographical regions. While TinyDB can shield application developers from the complexities of using such a lower level API, it can potentially leverage this work to do location-based event and query disemination.

Our use of event-based queries to implement query handoff resembles a content-based routing scheme in some ways, not dissimilar to the basic ideas of Directed Diffusion [13]. Of course the two schemes represent different design points: TinyDB is a relatively high-level system intended to shield application writers from network considerations, while Directed Diffusion is a lower-level multi-hop routing scheme that exposes content-based routing policies to users. These distinctions of ”high-level” and ”low-level” can become blurry, however, as illustrated by the routing-like handoff in TinyDB, and the query-like aggregation examples from Directed Diffusion. There are clearly open questions regarding possible synergies and tradeoffs between the two approaches in different settings; these merit further investigation.

8 Future Work and Conclusions

Many potential users of sensor networks are not computer scientists. In order for these users to develop new applications on sensor networks, high-level languages and corresponding execution environments are desirable. We are optimistic that a query-based approach can be a good general-purpose platform for application development.

The work described here attempts to justify this optimism with some non-trivial applications outside the realm of traditional SQL queries. In addition to pursuing the work here further, we also hope to continue this thrust by collaborating with domain experts in the development of new applications; this includes both application experts outside computing, and experts in other aspects of computing including collaborative signal processing and robotics. Our intent is for TinyDB to serve as an infrastructure that allows these experts to focus on issues within their expertise, leaving problems of data collection and movement in the hands of TinyDB’s adaptive query engine. As with traditional database systems, we do not necessarily expect a TinyDB-based implementation to always be as efficient as a hand-coded implementation, but we hope the ease of use and additional functionality of TinyDB will justify any modest performance overheads.

Acknowledgments

We thank Kannan Ramchandran and Michael Franklin for helpful discussions.

Bibliography

[1] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. Wavelet transforms that map integers to integers. *Applied and Computational Harmonic Analysis (ACHA)*, 5(3):332–369, 1998.

[2] Y. H. H. Dan Li, Kerry Wong and A. Sayeed. Detection, classification and tracking of targets in distributed sensor networks. *IEEE Signal Processing Magazine*, 19(2), Mar 2002.

[3] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI (to appear)*, 2002.

[4] D. Estrin. Embedded networked sensing for enviromental monitoring. Keynote, circuits and systems workshop. Slides available at http://lecs.cs.ucla.edu/ estrin/talks/CAS-JPL-Sept02.ppt.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, Santa Barbara, CA, May 2001.

[6] D. Ganesan, D. Estrin, and J. Heidemann. Dimensions: Why do we need a new data handling architecture for sensor networks? In *Proceedings of the First Workshop on Hot Topics In Networks (HotNets-I)*, Princeton, New Jersey, Oct. 2002.

[7] M. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *Proc. ACM SIGMOD 2002*, pages 476–487, Madison, WI, June 2002.

[8] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *SOSP*, October 2001.

[9] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis with CONTROL. *IEEE Computer*, 32(8), August 1999.

[10] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD*, pages 171–182, Tucson, AZ, May 1997.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, and D. C. K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

[12] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. Submitted for Publication, ICDCS-22, November 2001.

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCOM*, Boston, MA, August 2000.

[14] M. Leonov. Comparison of the algorithms for polygon boolean operations. Web Page: http://home.attbi.com/ msleonov/pbcomp.html.

[15] M. V. Leonov and A. G. Nitikin. An efficient algorithm for a closed set of boolean operations on polygonal regions in the plane. Technical report, A.P. Ershov Institute of Informatics Systems, 1997. Preprint 46 (In Russian.) English translation available at http://home.attbi.com/ msleonov/pbpaper.html.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. Submitted for publication., 2002.

[17] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI (to appear)*, 2002.

[18] S. Madden, W. Hong, J. M. Hellerstein, and M. Franklin. TinyDB web page. http://telegraph.cs.berkeley.edu/tinydb.

[19] J. Malik, S. Belognie, T. Leung, and J. Shi. Contour and texture analysis for image segmentation. *International Journal of Computer Vision*, 43(1):7–27, 2001.

[20] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD*, pages 448–459, Seattle, Washington, June 1998.

[21] P. Ramanathan, K. Saluja, K.-C. Wang, and T. Clouqueur. UW-API: A Network Routing Application Programmer's Interface. Draft version 1.0, January 2001.

[22] W. Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM J. Math. Anal.*, 29(2):511–546, 1997.

[23] W. Sweldens and P. Schröder. Building your own wavelets at home. In *Wavelets in Computer Graphics*, pages 15–87. ACM SIGGRAPH Course notes, 1996. `http://cm.bell-labs.com/who/wim/papers/athome.pdf`.

[24] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. In *SIGMOD Record*, September 2002.

[25] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. Technical Report 02-773, USC, September 2003.