TinyDB: An Acquisitional Query Processing System for Sensor Networks[1]

SAMUEL R. MADDEN
Massachusetts Institute of Technology
and
MICHAEL J. FRANKLIN and JOSEPH M. HELLERSTEIN
UC Berkeley
and
WEI HONG
Intel Research, Berkeley

General Terms: Experimentation,Performance

Additional Key Words and Phrases: Query Processing, Sensor Networks, Data Acquisition

1. INTRODUCTION

In the past few years, smart sensor devices have matured to the point that it is now feasible to deploy large, distributed networks of such devices [Pottie and Kaiser 2000; Hill et al. 2000; Mainwaring et al. 2002; Cerpa et al. 2001]. Sensor networks are differentiated from other wireless, battery-powered environments in that they consist of tens or hundreds of autonomous nodes that operate without human interaction (*e.g.*,configuration of network routes, recharging of batteries, or tuning of parameters) for weeks or months at a time. Furthermore, sensor networks are often embedded into some (possibly remote) physical environment from which they must monitor and collect data. The long-term, low-power nature of sensor networks, coupled with their proximity to physical phenomena, leads to a significantly altered view of software systems compared to more traditional mobile or distributed environments.

In this article, we are concerned with query processing in sensor networks. Researchers have noted the benefits of a query processor-like interface to sensor networks and the need for sensitivity to limited power and computational resources [Intanagonwiwat et al. 2000; Madden and Franklin 2002; P.Bonnet et al. 2001; Yao and Gehrke 2002; Madden et al. 2002]. Prior systems, however, tend to view query processing in sensor networks simply as a power-constrained version of traditional query processing: given some set of data, they

[1] This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.

strive to process that data as energy-efficiently as possible. Typical strategies include minimizing expensive communication by applying aggregation and filtering operations inside the sensor network – strategies that are similar to push-down techniques from distributed query processing that emphasize moving queries to data.

In contrast, we advocate *acquisitional query processing* (ACQP), where we focus not only on traditional techniques but also on the significant new query processing opportunity that arises in sensor networks: the fact that smart sensors have control over where, when, and how often data is physically acquired (*i.e.,sampled*) and delivered to query processing operators. By focusing on the locations and costs of acquiring data, we are able to significantly reduce power consumption compared to traditional passive systems that assume the *a priori* existence of data. Acquisitional issues arise at all levels of query processing: in query optimization, due to the significant costs of sampling sensors; in query dissemination, due to the physical co-location of sampling and processing; and, most importantly, in query execution, where choices of when to sample and which samples to process are made. We will see how techniques proposed in other research on sensor and power-constrained query processing, such as pushing down predicates and minimizing communication are also important alongside ACQP and fit comfortably within its model.

We have designed and implemented a query processor for sensor networks that incorporates acquisitional techniques called TinyDB (for more information on TinyDB, see the TinyDB Home Page [Madden et al. 2003]). TinyDB is a distributed query processor that runs on each of the nodes in a sensor network. TinyDB runs on the Berkeley *mote* platform, on top of the TinyOS [Hill et al. 2000] operating system. We chose this platform because the hardware is readily available from commercial sources [Crossbow, Inc.] and the operating system is relatively mature. TinyDB has many of the features of a traditional query processor (*e.g.,*the ability to select, join, project, and aggregate data), but, as we will discuss in this paper, also incorporates a number of other features designed to minimize power consumption via acquisitional techniques. These techniques, taken in aggregate, can lead to orders of magnitude improvements in power consumption *and* increased accuracy of query results over non-acquisitional systems that do not actively control when and where data is collected.

We address a number of questions related to query processing on sensor networks, focusing in particular on ACQP issues such as:

(1) *When should samples for a particular query be taken?*
(2) *What sensor nodes have data relevant to a particular query?*
(3) *In what order should samples for this query be taken, and how should sampling be interleaved with other operations?*
(4) *Is it worth expending computational power or bandwidth to process and relay a particular sample?*

Of these issues, question (1) is uniquely acquisitional. We show how the remaining questions can be answered by adapting techniques that are similar to those found in traditional query processing. Notions of indexing and optimization, in particular, can be applied to answer questions (2) and (3), and question (4) bears some similarity to issues that arise in stream processing and approximate query answering. We will address each of these questions, noting the unusual kinds of indices, optimizations, and approximations that are required under the specific constraints posed by sensor networks.

Figure 1 illustrates the basic architecture that we follow throughout this paper – queries are submitted at a powered PC (the *basestation*), parsed, optimized and sent into the sensor network, where they are disseminated and processed, with results flowing back up the routing tree that was formed as the queries propagated. After a brief introduction to sensor networks in Section 2, the remainder of the paper discusses each of these phases of ACQP: Section 3 covers our query language, Section 4 highlights optimization issues in power-sensitive environments, Section 5 discusses query dissemination, and finally, Section 6 discusses our adaptive, power-sensitive model for query execution and result collection.

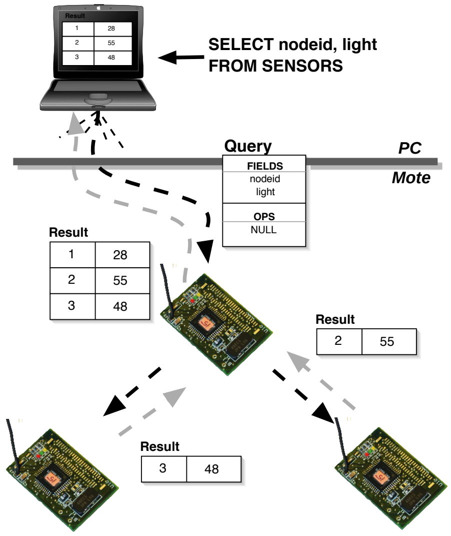


Fig. 1. A query and results propagating through the network.

2. SENSOR NETWORK OVERVIEW

We begin with an overview of some recent sensor network deployments, and then discuss properties of sensor nodes and sensor networks in general, providing specific numbers from our experience with TinyOS motes when possible.

A number of recent deployments of sensors have been undertaken by the sensor network research community for environmental monitoring purposes: on Great Duck Island [Mainwaring et al. 2002], off the coast of Maine, at James Reserve [Cerpa et al. 2001], in Southern California, at a vineyard in British Columbia [Brooke and Burrell 2003], and in the Coastal Redwood Forests of California [Madden 2003]. In these scenarios, motes collect light, temperature, humidity, and other environmental properties. On Great Duck Island, during the Summer of 2003, about 200 motes were placed in and around the burrows of Storm Petrels, a kind of endangered sea bird. Scientists used them to monitor burrow occupancy and the conditions surrounding burrows that are correlated with birds coming or going. Other notable deployments that are underway include a network for earthquake monitoring [UC Berkeley 2001] and a network for building infrastructure monitoring and control [Lin et al. 2002][2].

[2] Even in indoor infrastructure monitoring settings, there is great interest in battery powered devices, as running power wire can cost many dollars per device.

Each of these scenarios involves a large number of devices that need to last as long as possible with little or no human intervention. Placing new devices, or replacing or recharging batteries of devices in bird nests, earthquake test sites, and heating and cooling ducts is time consuming and expensive. Aside from the obvious advantages that a simple, declarative language provides over hand-coded, embedded C, researchers are particularly interested in TinyDB's ability to acquire and deliver desired data while conserving as much power as possible and satisfying desired lifetime goals.

We have deployed TinyDB in the redwood monitoring project [Madden 2003] described above, and are in the process of deploying it in Intel fabrication plants to collect vibration signals that can be used for early detection of equipment failures. Early deployments have been quite successful, producing months of lifetime from tiny batteries with about 1/2 the capacity of a single AA cell.

2.1 Properties of Sensor Devices

A sensor node is a battery-powered, wireless computer. Typically, these nodes are physically small (a few cubic centimeters) and extremely low power (a few tens of milliwatts versus tens of watts for a typical laptop computer)[3]. Power is of utmost importance. If used naively, individual nodes will deplete their energy supplies in only a few days[4]. In contrast, if sensor nodes are very spartan about power consumption, months or years of lifetime are possible. Mica motes, for example, when operating at 2% duty cycle (between active and sleep modes) can achieve lifetimes in the 6 month range on a pair of AA batteries. This duty cycle limits the active time to 1.2 seconds per minute.

There have been several generations of motes produced. Older, *Mica* motes have a 4 Mhz, 8 bit Atmel microprocessor. Their RFM TR1000 [RFM Corporation] radios run at 40 Kbits/second over a single shared CSMA/CA (carrier-sense multiple-access, collision avoidance) channel. Newer *Mica2* nodes use a 7 Mhz processor and a radio from ChipCon [ChipCon Corporation] which runs at 38.4 Kbits/sec. Radio messages are variable size. Typically about 20 50-byte messages (the default size in TinyDB) can be delivered per second. Like all wireless radios (but unlike a shared EtherNet, which uses the collision detection (CD) variant of CSMA), both the RFM and ChipCon radios are half-duplex, which means that they cannot detect collisions because they cannot listen to their own traffic. Instead, they try to avoid collisions by listening to the channel before transmitting and backing off for a random time period when it is in use. A third mote, called the *Mica2Dot*, has similar hardware as the Mica2 mote, but uses a slower, 4 Mhz, processor. A picture of a Mica and Mica2Dot mote are shown in Figure 2. Mica motes are visually very similar to Mica2 motes and are exactly the same form factor.

Motes have an external 32kHz clock that the TinyOS operating system can synchronize with neighboring motes to approximately +/- 1 ms. Time synchronization is important in a variety of contexts; for example: to ensure that readings can be correlated, to schedule communication, or to coordinate the waking and sleeping of devices.

[3] Recall that 1 Watt (a unit of power) corresponds to power consumption of 1 Joule (a unit of energy) per second. We sometimes refer to the current load of a device, because current is easy to measure directly; note that power (in Watts) = current (in Amps) * voltage (in Volts), and that motes run at 3V.

[4] At full power, a Berkeley Mica mote (see Figure 2) draws about 15 mA of current. A pair of AA batteries provides approximately 2200 mAh of energy. Thus, the lifetime of a Mica2 mote will be approximately $2200/15 = 146$ hours, or 6 days.

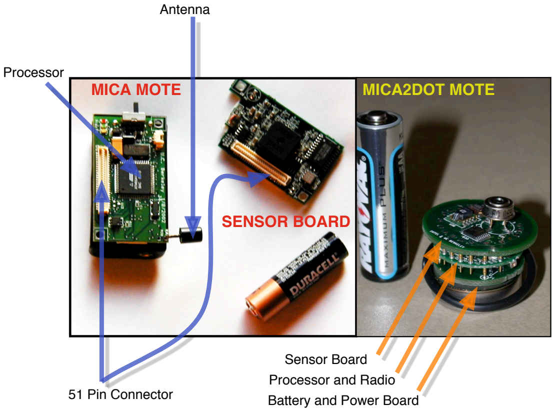


Fig. 2. Annotated Motes

2.1.1 *Power Consumption in Sensor Networks.* Power consumption in sensor nodes can be roughly decomposed into phases, which we illustrate in Figure 3 via an annotated capture of an oscilloscope display showing current draw (which is proportional to power consumption) on a Mica mote running TinyDB. In “Snoozing” mode, where the node spends most of its time, the processor and radio are idle, waiting for a timer to expire or external event to wake the device. When the device wakes it enters the “Processing” mode, which consumes an order of magnitude more power than snooze mode, and where query results are generated locally. The mote then switches to a “Processing and Receiving” mode, where results are collected from neighbors over the radio. Finally, in the “Transmitting” mode, results for the query are delivered by the local mote – the noisy signal during this period reflects switching as the receiver goes off and the transmitter comes on and then cycles back to a receiver-on, transmitter-off state.

Theses oscilloscope measurements do not distinguish how power is used during the active phase of processing. To explore this breakdown, we conducted an analytical study of the power utilization of major elements of sensor network query processing; the results of this study are given in Appendix A. In short, we found that in a typical data collection scenario, with relatively power-hungry sensing hardware, about 41% of energy goes to communicating or running the CPU while communicating, with another 58% going to

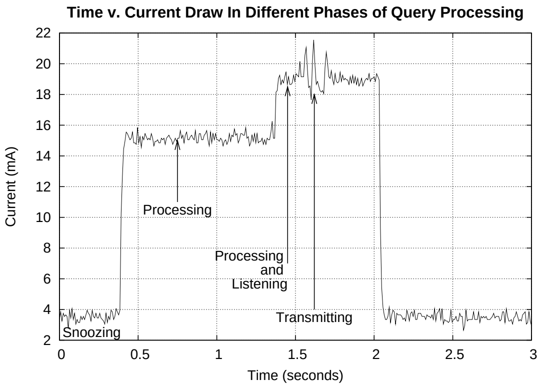


Fig. 3. Phases of Power Consumption In TinyDB

the sensors or to the CPU while sensing. The remaining 1% goes to idle-time energy consumption.

2.2 TinyOS

TinyOS consists of a set of components for managing and accessing the mote hardware, and a “C-like” programming language called nesC. TinyOS has been ported to a variety of hardware platforms, including UC Berkeley’s Rene, Dot, Mica, Mica2, and Mica2Dot motes, the Blue Mote from Dust Inc. [Dust Inc.], the MIT Cricket [Priyantha et al. 2000].

The major features of TinyOS are:

(1) A suite of software designed to simplify access to the lowest levels of hardware in an energy-efficient and contention-free way, and

(2) A programming model and the nesC language designed to promote extensibility and composition of software while maintaining a high degree of concurrency and energy efficiency. Interested readers should refer to [Gay et al. 2003].

It is interesting to note that TinyOS does not provide the traditional operating system features of process isolation or scheduling (there is only one application running at time), and does not have a kernel, protection domains, memory manager, or multi-threading. Indeed, in many ways, TinyOS is simply a library that provides a number of convenient software abstractions, including components to modulate packets over the radio, read sensor values for different sensor hardware, synchronize clocks between a sender and receiver, and put the hardware into a low-power state.

Thus, TinyOS and nesC provide a useful set of abstractions on top of the bare hardware. Unfortunately, they do not make it particularly easy to author software for the kinds of data collection applications considered in the beginning of Section 2. For example, the initial deployment of the Great Duck Island software, where the only behavior was to periodically broadcast readings from the same set of sensors over a single radio hop, consisted of more than 1,000 lines of embedded C code, excluding any of the custom software components written to integrate the new kinds of sensing hardware used in the deployment. Features such as reconfigurability, in-network processing, and multihop routing, which are needed for long-term, energy-efficient deployments, would require thousands of lines of additional code.

Sensor networks will never be widely adopted if every application requires this level of engineering effort. The declarative model we advocate reduces these applications to a few short statements in a simple language; the acquisitional techniques discussed allow these queries to be executed efficiently. [5]

2.3 Communication in Sensor Networks

Typical communication distances for low power wireless radios such as those used in motes and Bluetooth devices range from a few feet to around 100 feet, depending on transmission power and environmental conditions. Such short ranges mean that almost all real deployments must make use of multi-hop communication, where intermediate nodes relay information for their peers. On Mica motes, all communication is broadcast. The oper-

[5] The implementation of TinyDB consists of about 20,000 lines of C code, approximately 10,000 of which are for the low-level drivers to acquire and condition readings from sensors – none of which is the end-user is expected to have to modify or even look at. Compiled, this uses 58K of the 128K of available code space on current generations Motes.

ating system provides a software filter so that messages can be addressed to a particular node, though if neighbors are awake, they can still *snoop* on such messages (at no additional energy cost since they have already transferred the decoded message from the air.) Nodes receive per-message, link-level acknowledgments indicating whether a message was received by the intended neighbor node. No end-to-end acknowledgments are provided.

The requirement that sensor networks be low maintenance and easy to deploy means that communication topologies must be automatically discovered (*i.e.,ad-hoc*) by the devices rather than fixed at the time of network deployment. Typically, devices keep a short list of neighbors who they have heard transmit recently, as well as some routing information about the connectivity of those neighbors to the rest of the network. To assist in making intelligent routing decisions, nodes associate a link quality with each of their neighbors.

We describe the process of disseminating queries and collecting results in Section 5 below. As a basic primitive in these protocols, we use a *routing tree* that allows a *basestation* at the root of the network to disseminate a query and collect query results. This routing tree is formed by forwarding a routing request (a query in TinyDB) from every node in the network: the root sends a request, all *child* nodes that hear this request process it and forward it on to their children, and so on, until the entire network has heard the request.

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 (and its children's) query results to the basestation. We note that it is possible to have several routing trees if nodes keep track of multiple parents. This can be used to support several simultaneous queries with different roots. This type of communication topology is common within the sensor network community [Woo and Culler 2001].

3. ACQUISITIONAL QUERY LANGUAGE

In this section, we introduce our query language for ACQP focusing on issues related to when and how often samples are acquired. Appendix B gives a complete syntactic specification of the language; here, we rely primarily on example queries to illustrate the different language features.

3.1 Data Model

In TinyDB, sensor tuples belong to a table `sensors` which, logically, has one row per node per instant in time, with one column per attribute (*e.g.*,, light, temperature, *etc.*) that the device can produce. In the spirit of acquisitional processing, records in this table are materialized (*i.e.,* acquired) only as needed to satisfy the query, and are usually stored only for a short period of time or delivered directly out of the network. Projections and/or transformations of tuples form the `sensors` table may be stored in *materialization points* (discussed below).

Although we impose the same schema on the data produced by every device in the network, we allow for the possibility of certain devices lacking certain physical sensors by allowing nodes to insert NULLs for attributes corresponding to missing sensors. Thus, devices missing sensors requested in a query will produce data for that query anyway, unless NULLs are explicitly filtered out in the `WHERE` clause.

Physically, the `sensors` table is partitioned across all of the devices in the network, with each device producing and storing its own readings. Thus, in TinyDB, to compare readings from different sensors, those readings must be collected at some common node, *e.g.*,the root of the network.

3.2 Basic Language Features

Queries in TinyDB, as in SQL, consist of a `SELECT-FROM-WHERE-GROUPBY` clause supporting selection, join, projection, and aggregation.

The semantics of `SELECT, FROM, WHERE, and GROUP BY` clauses are as in SQL. The `FROM` clause may refer to both the *sensors* table as well as stored tables, which we call materialization points. Materialization points are created through special logging queries, which we describe below. They provide basic support for sub-queries and windowed stream operations.

Tuples are produced at well-defined *sample intervals* that are a parameter of the query. The period of time between the start of each sample period is known as an *epoch*. Epochs provide a convenient mechanism for structuring computation to minimize power consumption. Consider the query:

```
SELECT nodeid, light, temp
  FROM sensors
  SAMPLE PERIOD 1s FOR 10s
```

This query specifies that each device should report its own id, light, and temperature readings (contained in the virtual table `sensors`) once per second for 10 seconds. Results of this query stream to the root of the network in an online fashion, via the multi-hop topology, where they may be logged or output to the user. The output consists of a stream of tuples, clustered into 1s time intervals. Each tuple includes a time stamp corresponding to the time it was produced.

Nodes initiate data collection at the beginning of each epoch, as specified in the `SAMPLE PERIOD` clause. Nodes in TinyDB run a simple time synchronization protocol to agree on a global time base that allows them to start and end each epoch at the same time[6].

When a query is issued in TinyDB, it is assigned an identifier (id) that is returned to the issuer. This identifier can be used to explicitly stop a query via a “`STOP QUERY id`” command. Alternatively, queries can be limited to run for a specific time period via a `FOR` clause (shown above,) or can include a stopping condition as an event (see below.)

Note that because the `sensors` table is an unbounded, continuous data stream of values, certain blocking operations (such as sort and symmetric join) are not allowed over such streams unless a bounded subset of the stream, or *window*, is specified. Windows in TinyDB are defined via materialization points over the sensor streams. Such materialization points accumulate a small buffer of data that may be used in other queries. Consider, as an example:

```
CREATE
  STORAGE POINT recentlight SIZE 8
  AS (SELECT nodeid, light FROM sensors
  SAMPLE PERIOD 10s)
```

This statement provides a local (*i.e.*,single-node) location to store a streaming view of recent data similar to materialization points in other streaming systems like Aurora, TelegraphCQ, or STREAM [Carney et al. 2002; Chandrasekaran et al. 2003; Motwani et al. 2003], or materialized views in conventional databases. Multiple queries may read a ma-

[6] We use a time-synchronization protocol that is quite similar to the one described in work by [Ganeriwal et al. 2003]; typical time synchronization error in TinyDB is about 10ms.

terialization point.

Joins are allowed between two storage points on the same node, or between a storage point and the `sensors` relation, in which case `sensors` is used as the outer relation in a nested-loops join. That is, when a `sensors` tuple arrives, it is joined with tuples in the storage point at its time of arrival. This is effectively a *landmark query* [Gehrke et al. 2001] common in streaming systems. Consider, as an example:

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

This query outputs a stream of counts indicating the number of recent light readings (from 0 to 8 samples in the past) that were brighter than the current reading. In the event that a storage point and an outer query deliver data at different rates, a simple rate matching construct is provided that allows interpolation between successive samples (if the outer query is faster), via the `LINEAR INTERPOLATE` clause shown in Appendix B. Alternatively, if the inner query is faster, the user may specify an aggregation function to combine multiple rows via the `COMBINE` clause shown in Appendix B.

3.3 Aggregation Queries

TinyDB also includes support for grouped aggregation queries. Aggregation has the attractive property that it reduces the quantity of data that must be transmitted through the network; other sensor network research has noted that aggregation is perhaps the most common operation in the domain ([Intanagonwiwat et al. 2000; Yao and Gehrke 2002]). TinyDB includes a mechanism for user-defined aggregates and a metadata management system that supports optimizations over them, which we discuss in Section 4.1.

The basic approach of aggregate query processing in TinyDB is as follows: as data from an aggregation query flows up the tree, it is aggregated in-network according to the aggregation function and value-based partitioning specified in the query.

3.3.1 *Aggregate Query Syntax and Semantics.* Consider a user who wishes to monitor the occupancy of the conference rooms on a particular floor of a building. She chooses to do this by using microphone sensors attached to motes, and looking for rooms where the average volume is over some threshold (assuming that rooms can have multiple sensors). Her query could be expressed as:

```
SELECT AVG(volume),room FROM sensors
  WHERE floor = 6
  GROUP BY room
  HAVING AVG(volume) > threshold
  SAMPLE PERIOD 30s
```

This query partitions motes on the 6th floor according to the room where they are located (which may be a hard-coded constant in each device, or may be determined via some localization component available to the devices.) The query then reports all rooms where the average volume is over a specified threshold. Updates are delivered every 30 seconds. The query runs until the user deregisters it from the system. As in our earlier discussion of TinyDB's query language, except for the `SAMPLE PERIOD` clause, the semantics of this statement are similar to SQL aggregate queries.

Recall that the primary semantic difference between TinyDB queries and SQL queries is that the output of a TinyDB query is a stream of values, rather than a single aggregate value (or batched result). For these streaming queries, each aggregate record consists of one <*group id,aggregate value*> pair per group. Each group is time-stamped with an epoch number and the readings used to compute an aggregate record all belong to the same the same epoch.

3.3.2 *Structure of Aggregates*. TinyDB structures aggregates similarly to shared-nothing parallel database engines (*e.g.*,, [Bancilhon et al. 1987; Dewitt et al. 1990; Shatdal and Naughton 1995]). The approach used in such systems (and followed in TinyDB) is to implement agg 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*, computed over one or more sensor values, representing the intermediate state over those values that will be required to compute an aggregate. $< z >$ 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.

These three functions can easily be derived for the basic SQL aggregates; in general, the only constraint is that the merging function be commutative and associative.

TinyDB includes a simple facility for allowing programmers to extend the system with new aggregates by authoring software modules that implement these three functions.

3.4 Temporal Aggregates

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

```
SELECT WINAVG(volume, 30s, 5s)
  FROM sensors
  SAMPLE PERIOD 1s
```

will report the average volume over the last 30 seconds once every 5 seconds, sampling once per second. This is an example of a *sliding-window* query common in many streaming systems [Motwani et al. 2003; Chandrasekaran et al. 2003; Gehrke et al. 2001]. We note that the same semantics are available by running an aggregate query with `SAMPLE PERIOD 5s` over a 30s materialization point; temporal aggregates simply provide a more concise way of expressing these common operations.

3.5 Event-Based Queries

As a variation on the continuous, polling based mechanisms for data acquisition, TinyDB supports *events* as a mechanism for initiating data collection. Events in TinyDB are generated explicitly, either by another query or by a lower-level part of the operating system (in which case the code that generates the event must have been compiled into the sensor node[7]) For example, the query:

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

could be used to report the average light and temperature level at sensors near a bird nest where a bird has just been detected. Every time a `bird-detect` event occurs, the query is issued from the detecting node and the average light and temperature are collected from nearby nodes once every 2 seconds for 30 seconds. In this case, we expect that bird-detection is done via some low-level operating system facility – *e.g.,*a switch that is triggered when a bird enters its nest.

Such events are central in ACQP, as they allow the system to be dormant until some external conditions occurs, instead of continually polling or blocking on an iterator waiting for some data to arrive. Since most microprocessors include external interrupt lines than can wake a sleeping device to begin processing, events can provide significant reductions in power consumption, shown in Figure 4.

This figure shows an oscilloscope plot of current draw from a device running an event-based query triggered by toggling a switch connected to an external interrupt line that causes the device to wake from sleep. Compare this to plot at the bottom of Figure 4, which shows an event-based query triggered by a second query that polls for some condition to be true. Obviously, the situation in the top plot is vastly preferable, as much less energy is spent polling. TinyDB supports such externally triggered queries via events, and such support is integral to its ability to provide low power processing.

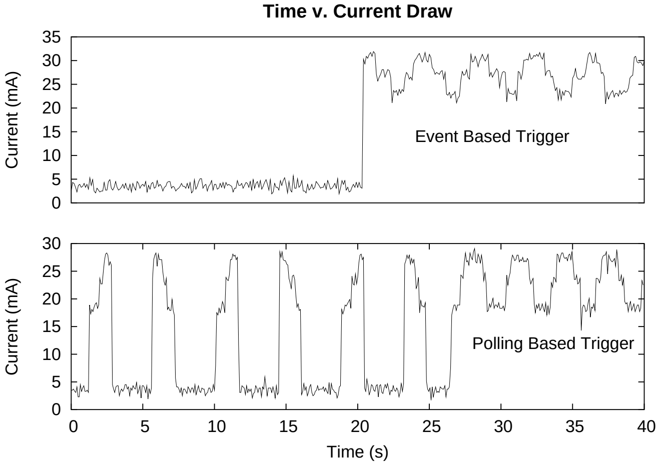


Fig. 4. External interrupt driven event-based query (top) vs. Polling driven event-based query (bottom).

Events can also serve as stopping conditions for queries. Appending a clause of the form `STOP ON EVENT(param) WHERE cond(param)` will stop a continuous

[7] TinyDB provides a special API for generating events; it is described in the TinyOS/TinyDB distribution as a part of the TinySchema package. As far as TinyDB is concerned. This API allows TinyDB to treat OS-defined events as black-boxes that occur at any time; for example, events may periodically sample sensors using low-level OS APIs (instead of TinyDB) to determine if some condition is true.

query when the specified event arrives and the condition holds.

Besides the low-level API which can be used to allow software components to signal events (such as the `bird-detect` event above), queries may also signal events. For example, suppose we wanted to signal an event whenever the temperature went above some threshold; we can write the following query:

```
SELECT nodeid,temp
  WHERE temp > thresh
  OUTPUT ACTION SIGNAL hot(nodeid,temp)
  SAMPLE PERIOD 10s
```

Clearly, we lose the power-saving advantages of having an event fired directly in response to a low-level interrupt, but we still retain the programmatic advantages of linking queries to the signaling of events. We describe the `OUTPUT ACTION` clause in more detail in Section 3.7 below.

In the current implementation of TinyDB, events are only signaled on the local node – we do not currently provide a fully distributed event propagation system. Note, however, that queries started in response to a local event may be disseminated to other nodes (as in the example above).

3.6 Lifetime-Based Queries

In lieu of an explicit `SAMPLE PERIOD` clause, users may request a specific query lifetime via a `QUERY LIFETIME <x>` clause, where `<x>` is a duration in days, weeks, or months. Specifying lifetime is a much more intuitive way for users to reason about power consumption. Especially in environmental monitoring scenarios, scientific users are not particularly concerned with small adjustments to the sample rate, nor do they understand how such adjustments influence power consumption. Such users, however, are very concerned with the lifetime of the network executing the queries. Consider the query:

```
SELECT nodeid, accel
  FROM sensors
  LIFETIME 30 days
```

This query specifies that the network should run for at least 30 days, sampling light and acceleration sensors at a rate that is as quick as possible and still satisfies this goal.

To satisfy a lifetime clause, TinyDB performs lifetime estimation. The goal of lifetime estimation is to compute a sampling and transmission rate given a number of Joules of energy remaining. We begin by considering how a single node at the root of the sensor network can compute these rates, and then discuss how other nodes coordinate with the root to compute their delivery rates. For now, we also assume that sampling and delivery rates are the same. On a single node, these rates can be computed via a simple cost-based formula, taking into account the costs of accessing sensors, selectivities of operators, expected communication rates and current battery voltage[8]. We show below a lifetime computation for simple queries of the form:

```
SELECT a_1, ...  , a_numSensors
  FROM sensors
  WHERE p
  LIFETIME l hours
```

To simplify the equations in this example, we present a query with a single selection predicate that is applied after attributes have been acquired. The ordering of multiple

[8] Throughout this section, we will use battery voltage as a proxy for remaining battery capacity, as voltage is an easy quantity to measure.

Table I. Parameters used in lifetime estimation

Parameter	Description	Units
l	Query lifetime goal	hours
c_{rem}	Remaining Battery Capacity	Joules
E_n	Energy to sample sensor n	Joules
E_{trans}	Energy to transmit a single sample	Joules
E_{rcv}	Energy to receive a message	Joules
σ	Selectivity of selection predicate	
C	# of children routing through node	

predicates and interleaving of sampling and selection are discussed in detail in Section 4. Table I shows the parameters we use in this computation (we do not show processor costs since they will be negligible for the simple selection predicates we support, and have been subsumed into costs of sampling and delivering results.)

The first step is to determine the available power p_h per hour,p_h = c_{rem} / l.

We then need to compute the energy to collect and transmit one sample, e_s, including the costs to forward data for its children:

$e_s = (\sum_{s=0}^{numSensors} E_s) + (E_{rcv} + E_{trans}) \times C + E_{trans} \times \sigma$

The energy for a sample is the cost to read all of the sensors at the node, plus the cost to receive results from children, plus the cost to transmit satisfying local and child results. Finally, we can compute the maximum transmission rate, T (in samples per hour), as :

$T = p_h/e_s$

To illustrate the effectiveness of this simple estimation, we inserted a lifetime-based query (`SELECT voltage, light FROM sensors LIFETIME x`) into a sensor (with a fresh pair of AA batteries) and asked it to run for 24 weeks, which resulted in a sample rate of 15.2 seconds per sample. We measured the voltage on the device 9 times over 12 days. The first two readings were outside the range of the voltage detector on the mote (*e.g.*,they read "1024" – the maximum value) so are not shown. Based on experiments with our test mote connected to a power supply, we expect it to stop functioning when its voltage reaches 350. Figure 5 shows the measured lifetime at each point in time, with a linear fit of the data, versus the "expected voltage" which was computed using the cost model above. The resulting linear fit of voltage is quite close to the expected voltage. The linear fit reaches V=350 about 5 days after the expected voltage line.

Given that it is possible to estimate lifetime on a single node, we now discuss coordinating the transmission rate across all nodes in the routing tree. Since sensors need to sleep between relaying of samples, it is important that senders and receivers synchronize their wake cycles. To do this, we allow nodes to transmit only when their parents in the routing tree are awake and listening (which is usually the same time they are transmitting.) By transitivity, this limits the maximum rate of the entire network to the transmission rate of the root of the routing tree. If a node must transmit slower than the root to meet the lifetime clause, it may transmit at an integral divisor of the root's rate.[9] To propagate this rate through the network, each parent node (including the root) includes its transmission rate in queries that it forwards to its children.

The previous analysis left the user with no control over the sample rate, which could be a problem because some applications require the ability to monitor physical phenomena

[9] One possible optimization, which we do not explore, would involve selecting or reassigning the root to maximize transmission rate.

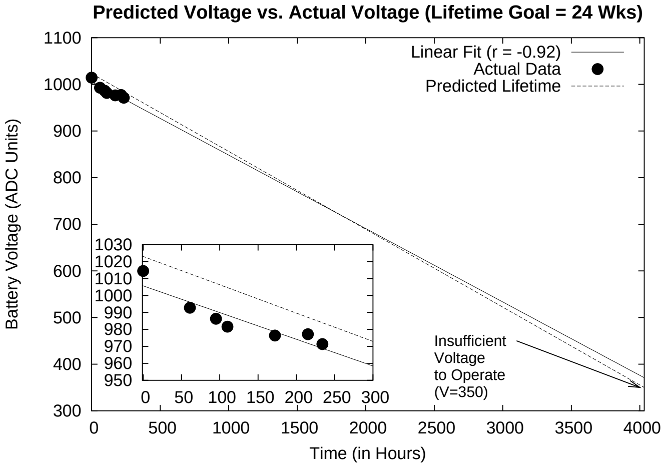


Fig. 5. Predicted versus actual lifetime for a requested lifetime of 24 weeks (168 days)

at a particular granularity. To remedy this, we allow an optional `MIN SAMPLE RATE` r clause to be supplied. If the computed sample rate for the specified lifetime is greater than this rate, sampling proceeds at the computed rate (since the alternative is expressible by replacing the `LIFETIME` clause with a `SAMPLE PERIOD` clause.) Otherwise, sampling is fixed at a rate of r and the prior computation for transmission rate is done assuming a different rate for sampling and transmission. To provide the requested lifetime and sampling rate, the system may not be able to actually transmit all of the readings – it may be forced to combine (aggregate) or discard some samples; we discuss this situation (as well as other contexts where it may arise) in Section 6.3.

Finally, we note that since estimation of power consumption was done using simple selectivity estimation as well as cost-constants that can vary from node-to-node (see Section 4.1) and parameters that vary over time (such as number of children, C), we need to periodically re-estimate power consumption. Section 6.4.1 discusses this runtime re-estimation in more detail.

3.7 Types of Queries in Sensor Networks

We conclude this section with a brief overview of some of the other types of queries supported by TinyDB.

- *Monitoring Queries*: Queries that request the value of one or more attributes continuously and periodically – for example, reporting the temperature in bird nests every thirty seconds; these are similar to the queries shown above.

- *Network Health Queries*: Meta-queries over the network itself. Examples include selecting parents and neighbors in the network topology or nodes with battery life less than some threshold. These queries are particularly important in sensor networks due to their dynamic and volatile nature. For example, the following query reports all sensors whose current battery voltage is less than k:

```
SELECT nodeid,voltage
 WHERE voltage < k
 FROM sensors
 SAMPLE PERIOD 10 minutes
```

- *Exploratory Queries*: One-shot queries examining the status of a particular node or set of nodes at a point in time. In lieu of the `SAMPLE PERIOD` clause, users may specify the keyword *ONCE*. For example:

```
SELECT light,temp,volume
 WHERE nodeid = 5
 FROM sensors
 ONCE
```

- *Nested Queries*: Both events and materialization points provide a form of nested queries. The TinyDB language does not currently support SQL-style nested queries, because the semantics of such queries are somewhat ill-defined in a streaming environment: it is not clear when should the outer query be evaluated given that the inner query may be a streaming query that continuously accumulates results. Queries over materialization points allow users to choose when the query is evaluated. Using the `FOR` clause, users can build a materialization point that contains a single buffer's worth of data, and can then run a query over that buffer, emulating the same effect as a nested query over a static inner relation. Of course, this approach eliminates the possibility of query rewrite based optimizations for nested queries [Pirahesh et al. 1992], potentially limiting query performance.

- *Actuation Queries*: Users want to able to take some physical action in response to a query. We include a special `OUTPUT ACTION` clause for this purpose. For example, users in building monitoring scenarios might want to turn on a fan in response to temperature rising above some level:

```
SELECT nodeid,temp
 FROM sensors
 WHERE temp > threshold
 OUTPUT ACTION power-on(nodeid)
 SAMPLE PERIOD 10s
```

The `OUTPUT ACTION` clause specifies an external `command` that should be invoked in response to a tuple satisfying the query. In this case, the `power-on` command is a low-level piece of code that pulls an output pin on the microprocessor high, closing a relay circuit and giving power to some externally connected device. Note that a separate query could be issued to `power-off` the fan when the temperature fell below some other threshold. The `OUTPUT ACTION` suppresses the delivery of messages to the basestation.

- *Offline Delivery*: There are times when users want to log some phenomenon that happens faster than the data can be transmitted over the radio. TinyDB supports the logging of results to EEPROM for offline, non-real time delivery. This is implemented through the materialization point mechanism described above.

Together, the these query types provide users of TinyDB with the mechanisms they need to build data collection applications on top of sensor networks.

4. POWER-BASED QUERY OPTIMIZATION

Given our query language for ACQP environments, with special features for event-based processing and lifetime queries, we now turn to query processing issues. We begin with a discussion of optimization, and then cover query dissemination and execution. We note that, based on the applications deployed so far, single table queries with aggregations seem to be the most pressing workload for sensor networks, and hence we focus primarily in this section on optimizations for acquisition, selection, and aggregation.

Queries in TinyDB are parsed at the basestation and disseminated in a simple binary format into the sensor network, where they are instantiated and executed. Before queries are disseminated, the basestation performs a simple query optimization phase to choose the correct ordering of sampling, selections, and joins.

We use a simple cost-based optimizer to choose a query plan that will yield the lowest overall power consumption. Optimizing for power allows us to subsume issues of processing cost and radio communication, which both contribute to power consumption and so will be taken into account. One of the most interesting aspects of power-based optimization, and a key theme of acquisitional query processing, is that the cost of a particular plan is often dominated by the cost of sampling the physical sensors and transmitting query results, rather than the cost of applying individual operators. For this reason, we focus in this section on optimizations that reduce the number and costs of data acquisition.

We begin by looking at the types of metadata stored by the optimizer. Our optimizer focuses on ordering joins, selections, and sampling operations that run on individual nodes.

4.1 Metadata Management

Each node in TinyDB maintains a catalog of metadata that describes its local attributes, events, and user-defined functions. This metadata is periodically copied to the root of the network for use by the optimizer. Metadata is registered with the system via static linking done at compile time using the TinyOS C-like programming language. Events and attributes pertaining to various operating system and TinyDB components are made available to queries by declaring them in an interface file and providing a small handler function. For example, in order to expose network topology to the query processor, the TinyOS `Network` component defines the attribute `parent` of type integer and registers a handler that returns the id of the node's parent in the current routing tree.

Event metadata consists of a name, a signature, and a frequency estimate that is used in query optimization (see Section 4.3 below.) User-defined predicates also have a name and a signature, along with a selectivity estimate which is provided by the author of the function.

Table II summarizes the metadata associated with each attribute, along with a brief description. Attribute metadata is used primarily in two contexts: information about the cost, time to fetch, and range of an attribute is used in query optimization, while information about the semantic properties of attributes is used in query dissemination and result processing. Table III gives examples of power and sample time values for some actual sensors – notice that the power consumption and time to sample can differ across sensors by several orders of magnitude.

The catalog also contains metadata about TinyDB's extensible aggregate system. As with other extensible database systems [Stonebraker and Kemnitz 1991] the catalog includes names of aggregates and pointers to their code. Each aggregate consists of a triplet of functions, that initialize, merge, and update the final value of partial aggregate records as they flow through the system. As in the TAG[Madden et al. 2002] paper, aggregate authors must provide information about functional properties. In TinyDB, we currently require two: whether the aggregate is *monotonic* and whether it is *exemplary* or *summary*. `COUNT` is a monotonic aggregate as its value can only get larger as more values are aggregated. `MIN` is an exemplary aggregate, as it returns a single value from the set of aggregate values, while `AVERAGE` is a summary aggregate because it computes some property over the entire set of values.

TinyDB also stores metadata information about the costs of processing and delivering data, which is used in query-lifetime estimation. The costs of these phases in TinyDB were shown in Figure 3 – they range from 2 mA while sleeping, to over 20 mA while transmitting and processing. Note that actual costs vary from mote to mote – for example, with a small sample of 5 motes (using the same batteries), we found that the average current with processor active varied from 13.9 to 17.6 mA (with the average being 15.66 mA).

4.2 Technique 1: Ordering of Sampling And Predicates

Having described the metadata maintained by TinyDB, we now describe how it is used in query optimization.

As shown in Section 2, sampling is often an expensive operation in terms of power. However, a sample from a sensor s must be taken to evaluate any predicate over the attribute `sensors.s`. If a predicate discards a tuple of the `sensors` table, then subsequent predicates need not examine the tuple – and hence the expense of sampling any attributes referenced in those subsequent predicates can be avoided. Thus these predicates are "expensive", and need to be ordered carefully. The predicate ordering problem here is somewhat different than than in the earlier literature (*e.g.,*[Hellerstein 1998]) because (a) an attribute may be referenced in multiple predicates, and (b) expensive predicates are only on a single table, `sensors`. The first point introduces some subtlety, as it is not clear which predicate should be "charged" the cost of the sample.

To model this issue, we treat the sampling of a sensor t as a separate "job" τ to be scheduled along with the predicates. Hence a set of predicates $P = \{p_1, \ldots, p_m\}$ is rewritten as a set of operations $S = \{s_1, \ldots, s_n\}$, where $P \subset S$, and $S - P = \{\tau_1, \ldots, \tau_{n-m}\}$ contains one sampling operator for each distinct attribute referenced in P. The selectivity of sampling operators is always 1. The selectivity of selection operators is derived by assuming that attributes have a uniform distribution over their range (which is available in the catalog.) Relaxing this assumption by, for example, storing histograms or time-dependent functions per-attribute remains an area of future work. The cost of an operator (predicate

[8] Scientists are particularly interested in monitoring the micro-climates created by plants and their biological processes. See [Delin and Jackson 2000; Cerpa et al. 2001]. An example of such a sensor is Figaro Inc's H_2S sensor [Figaro, Inc.].

Table II. Metadata fields kept with each attribute

Metadata	Description
Power	Cost to sample this attribute (in J)
Sample Time	Time to sample this attribute (in s)
Constant?	Is this attribute constant-valued (*e.g.,*id)?
Rate of Change	How fast the attribute changes (units/s)
Range	Dynamic range of attribute values (pair of units)

Table III. Summary of Power Requirements of Various Sensors Available for Motes

Sensor	**Time per Sample (ms)**	**Startup Time (ms)**	**Current (mA)**	**Energy Per Sample, mJ**
Weather Board Sensors				
Solar Radiation [TAOS, Inc. 2002]	500	800	0.350	.525
Barometric Pressure [Intersema 2002]	35	35	0.025	0.003
Humidity [Sensirion 2002]	333	11	.500	0.5
Surface Temp [Melexis, Inc. 2002]	0.333	2	5.6	0.0056
Ambient Temp [Melexis, Inc. 2002]	0.333	2	5.6	0.0056
Standard Mica Mote Sensors				
Accelerometer [Analog Devices, Inc.]	0.9	17	0.6	0.0048
(Passive) Thermistor [Atmel Corporation]	0.9	0	0.033	0.00009
Magnetometer [Honeywell, Inc.]	.9	17	5	.2595
Other Sensors				
Organic Byproducts[10]	.9	> 1000	5	> 15

or sample) can be determined by consulting the metadata, as described in the previous section. In the cases we discuss here, selections and joins are essentially “free” compared to sampling, but this is not a requirement of our technique.

We also introduce a partial order on S, where τ_i must precede p_j if p_j references the attribute sampled by τ_i. The combination of sampling operators and the dependency of predicates on samples captures the costs of sampling operators and the sharing of operators across predicates.

The partial order induced on S forms a graph with edges from sampling operators to predicates. This is a simple *series-parallel* graph. An optimal ordering of jobs with series-parallel constraints is a topic treated in the Operations Research literature that inspired earlier optimization work [Ibaraki and Kameda 1984; Krishnamurthy et al. 1986; Hellerstein 1998]; Monma and Sidney present the *Series-Parallel Algorithm Using Parallel Chains* [Monma and Sidney 1979], which gives an optimal ordering of the jobs in $O(|S| \log |S|)$ time.

Besides predicates in the `WHERE` clause, expensive sampling operators must also be ordered appropriately with respect to the `SELECT`, `GROUP BY`, and `HAVING` clauses. As with selection predicates, we enhance the partial order such that τ_i must precede any aggregation, `GROUP BY`, or `HAVING` operator that uses i. Note that projections do not require access to the value of i, and thus do not need to be included in the partial order. Thus, the complete partial order is:

(1) acquisition of attribute $a \prec$ any operator that references a
(2) selection $\prec$ aggregation, `GROUP BY`, and `HAVING`
(3) GROUP BY $\prec$ aggregation and `HAVING`
(4) aggregation $\prec$ `HAVING`

Of course, the last three rules are also present in standard SQL. We also need to add the operators representing these clauses to S with the appropriate costs and selectivities; the process of estimating these values been well-studied in the database query optimization and cost estimation literature.

As an example of this process, consider the query:

```
SELECT accel,mag
 FROM sensors
 WHERE accel > c1
 AND mag > c2
 SAMPLE PERIOD .1s
```

The order of magnitude difference in per-sample costs shown in Table III for the accelerometer and magnetometer suggests that the power costs of plans for this query having different sampling and selection orders will vary substantially. We consider three possible plans: in the first, the magnetometer and accelerometer are sampled before either selection is applied. In the second, the magnetometer is sampled and the selection over its reading (which we call S_{mag}) is applied before the accelerometer is sampled or filtered. In the third plan, the accelerometer is sampled first and its selection (S_{accel}) is applied before the magnetometer is sampled.

This interleaving of sampling and processing introduces an additional issue with temporal semantics: in this case, for example, the magnetometer and accelerometer samples are not acquired at the same time. This may be problematic for some queries, for example, if one is trying to temporally correlate high-frequency portions of signals from these two sensors. To address this concern, we include in our language specification a `NO INTERLEAVE` clause, which forces all sensors to be turned on and sampled simultaneously at the beginning of each epoch (obviating the benefit of the acquisitional techniques discussed in this section.) We note that this clause may not lead to perfect synchronization of sampling, as different sensors take different amounts of time to power up and acquire readings, but will substantially improve temporal synchronization.

Figure 6 shows the relative power costs of the latter two approaches, in terms of power costs to sample the sensors (we assume the CPU cost is the same for the two plans, so do not include it in our cost ratios) for different selectivity factors of the two selection predicates S_{accel} and S_{mag}. The selectivities of these two predicates are shown on the X and Y axis, respectively. Regions of the graph are shaded corresponding to the ratio of costs between the plan where the magnetometer is sampled first (*mag-first*) versus the plan where the accelerometer is sampled first (*accel-first*). As expected, these results show that the *mag-first* plan is almost always more expensive than *accel-first*. In fact, it can be an order of magnitude more expensive, when S_{accel} is much more selective than S_{mag}. When S_{mag} is highly selective, however, it can be cheaper to sample the magnetometer first, although only by a small factor.

The maximum difference in relative costs represents an absolute difference of 255 uJ per sample, or 2.5 mW at a sample rate of ten samples per second – putting the additional power consumption from sampling in the incorrect order on par with the power costs of running the radio or CPU for an entire second.

4.2.1 *Exemplary Aggregate Pushdown.* There are certain kinds of aggregate functions where the same kind of interleaving of sampling and processing can also lead to a performance savings. Consider the query:

```
SELECT WINMAX(light,8s,8s)
 FROM sensors
 WHERE mag > x
 SAMPLE PERIOD 1s
```

In this query, the maximum of eight seconds worth of light readings will be computed, but only light readings from sensors whose magnetometers read greater than x will be considered. Interestingly, it turns out that, unless the `mag >` x predicate is *very* selective, it will be cheaper to evaluate this query by checking to see if each new `light` reading is greater than the previous reading and then applying the selection predicate over `mag`, rather than first sampling `mag`. This sort of reordering, which we call *exemplary aggregate*

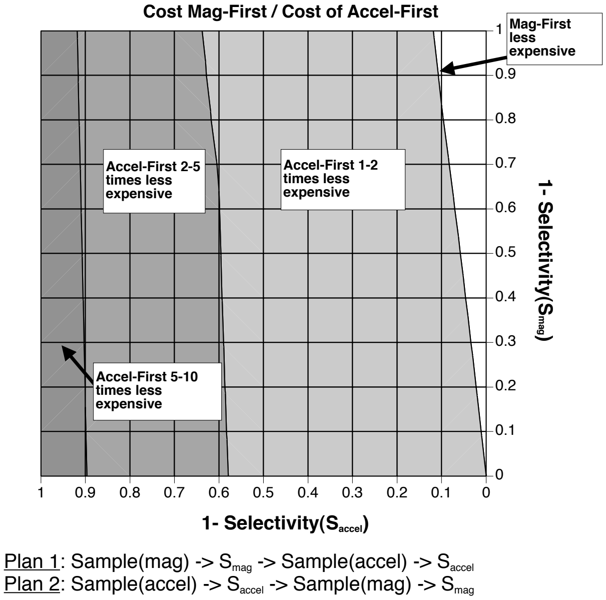


Fig. 6. Ratio of costs of two acquisitional plans over differing-cost sensors.

pushdown can be applied to any exemplary aggregate (*e.g.*,MIN, MAX). Similar ideas have been explored in the deductive database community by Sudarshan *et al*. [Sudarshan and Ramakrishnan 1991].

The same technique can be used with non-windowed aggregates when performing in-network aggregation. Suppose we are applying an exemplary aggregate at an intermediate node in the routing tree; if there is an expensive acquisition required to evaluate a predicate (as in the query above), then it may make sense to see if the local value affects the value of the aggregate before acquiring the attribute used in the predicate.

To add support for exemplary aggregate pushdown, we need a way to evaluate the selectivity of exemplary aggregates. In the absence of statistics that reflect how a predicate changes over time, we simply assume that the attributes involved in an exemplary aggregate (such as light in the query above) are sampled from the same distribution. Thus, for MIN and MAX aggregates, the likelihood that the second of two samples is less than (or greater than) the first is 0.5. For n samples, the likelihood that the nth is the value reported by the aggregate is thus $1/.5^{n-1}$. By the same reasoning, for bottom (or top)-k aggregates, assuming $k < n$, the nth sample will be reported with probability $1/.5^{n-k-1}$.

Given this selectivity estimate for an exemplary aggregate, $S(a)$, over attribute a with acquisition cost $C(a)$, we can compute the benefit of exemplary aggregate pushdown. We assume the query contains some set of conjunctive predicates with aggregate selectivity P over several expensive acquisitional attributes with aggregate acquisition cost K. We assume the values of $S(a)$, $C(a)$, and K, and P are available in the catalog. Then, the cost of evaluating the query without exemplary aggregate pushdown is:

$(1) K + P * C(a)$

and with pushdown it becomes:

$(2) C(a) + S(a) * K$

When (2) is less than (1), there will be an expected benefit to exemplary aggregate pushdown, and it should be applied.

4.3 Technique 2: Event Query Batching to Conserve Power

As a second example of the benefit of power-aware optimization, we consider the optimization of the query:

```
ON EVENT e(nodeid)
  SELECT a1
  FROM sensors AS s
  WHERE s.nodeid = e.nodeid
  SAMPLE PERIOD d FOR k
```

This query will cause an instance of the internal query (`SELECT ...`) to be started *every time* the event e occurs. The internal query samples results every d seconds for a duration of k seconds, at which point it stops running.

Note that, according to this specification of how an `ON EVENT` query is processed, it is possible for multiple instances of the internal query to be running at the same time. If enough such queries are running simultaneously, the benefit of event-based queries (*e.g.,*not having to poll for results) will be outweighed by the fact that each instance of the query consumes significant energy sampling and delivering (independent) results. To alleviate the burden of running multiple copies of the same identical query, we employ a multi-query optimization technique based on rewriting. To do this, we convert external events (of type e) into a stream of events, and rewrite the entire set of independent internal queries as a sliding window join between `events` and `sensors`, with a window size of k seconds on the event stream, and no window on the sensor stream. For example:

```
SELECT s.a1
  FROM sensors AS s, events AS e
  WHERE s.nodeid = e.nodeid
  AND e.type = e
  AND s.time - e.time <= k AND s.time > e.time
  SAMPLE PERIOD d
```

We execute this query by treating it as a join between a materialization point of size k on `events` and the `sensors` stream. When an `event` tuple arrives, it is added to the buffer of events. When a `sensor` tuple s arrives, events older than k seconds are dropped from the buffer and s is joined with the remaining events.

The advantage of this approach is that only one query runs at a time no matter how frequently the events of type e are triggered. This offers a large potential savings in sampling and transmission cost. At first it might seem as though requiring the sensors to be sampled every d seconds irrespective of the contents of the event buffer would be prohibitively expensive. However, the check to see if the the event buffer is empty can be pushed before the sampling of the sensors, and can be done relatively quickly.

Figure 7 shows the power tradeoff for event-based queries that have and have not been rewritten. Rewritten queries are labeled as *stream join* and non-rewritten queries as *async events*. We measure the cost in mW of the two approaches using a numerical model of power costs for idling, sampling and processing (including the cost to check if the event queue is non-empty in the event-join case), but excluding transmission costs to avoid complications of modeling differences in cardinalities between the two approaches. The expectation was that the asynchronous approach would generally transmit many more results. We varied the sample rate and duration of the inner query, and the frequency of events. We chose the specific parameters in this plot to demonstrate query optimization tradeoffs; for much faster or slower event rates, one approach tends to always be preferable. In this case, the stream-join rewrite is beneficial as when events occur frequently; this might be the case if, for example, an event is triggered whenever a signal goes above or below a threshold with a signal that is sampled tens or hundreds of times per second; vibration monitoring applications tend to have this kind of behavior. Table IV summarizes the parameters used in this experiment; "derived" values are computed by the model below. Power consumption numbers and sensor timings are drawn from Table III and the Atmel 128 data sheet [Atmel Corporation].

The cost in milliwatts of the asynchronous events approach, mW_{events}, is modeled via the following equations:

$$\begin{aligned}
t_{idle} = t_{sample} - n_{events} \times dur_{event} \times ms_{sample}/1000 \\
mJ_{idle} = mW_{idle} \times t_{idle} \\
mJ_{sample} = mW_{sample} \times ms_{sample}/1000 \\
mW_{events} = (n_{events} \times dur_{event} \times mJ_{sample} + mJ_{idle})/t_{sample}
\end{aligned}$$

The cost in milliwatts of the Stream Join approach, $mW_{streamJoin}$, is then:

$$\begin{aligned}
t_{idle} = t_{sample} - (ms_{check} + ms_{sample})/1000 \\
mJ_{idle} = mW_{idle} \times t_{idle} \\
mJ_{check} = mW_{proc} \times ms_{check}/1000 \\
mJ_{sample} = mW_{sample} \times ms_{samples}/1000 \\
mW_{streamJoin} = (mJ_{check} + mJ_{sample} + mJ_{idle})/t_{sample}
\end{aligned}$$

For very low event rates (fewer than 1 per second), the asynchronous events approach is sometimes preferable due to the extra overhead of empty-checks on the event queue in the stream-join case. However, for faster event rates, the power cost of this approach increases rapidly as independent samples are acquired for each event every few seconds. Increasing the duration of the inner query increases the cost of the asynchronous approach as more queries will be running simultaneously. The maximum absolute difference (of about .8mW) is roughly comparable to 1/4 the power cost of the CPU or radio.

Table IV. Parameters used in Async. Events vs. Stream Join Study

Parameter	Description	Value
t_{sample}	Length of sample period	1/8 s
n_{events}	Number of events per second	0 - 5 (x axis)
dur_{event}	Time for which events are active (FOR clause)	1, 3, or 5 s
mW_{proc}	Processor power consumption	12 mW
ms_{sample}	Time to acquire a sample, including processing and ADC time	.35 ms
mW_{sample}	Power used while sampling, including processor	13 mW
mJ_{sample}	Energy per sample	Derived
mW_{idle}	Milliwatts used while idling	Derived
t_{idle}	Time spent idling per sample period (in seconds)	Derived
mJ_{idle}	Energy spent idling	Derived
ms_{check}	Time to check for enqueued event	.02 ms (80 instrs)
mJ_{check}	Energy to check if an event has been enqueued	Derived
mW_{events}	Total power used in *Async Events* mode	Derived
$mW_{streamJoin}$	Total power used in *Stream Join* mode	Derived

Finally, we note that there is a subtle semantic change introduced by this rewriting. The initial formulation of the query caused samples in each of the internal queries to be produced relative to the time that the event fired: for example, if event e_1 fired at time t, samples would appear at time $t + d, t + 2d, ...$ If a later event e_2 fired at time $t + i$, it would produce a different set of samples at time $t + i + d, t + i + 2d, ...$ Thus, unless i were equal to d (*i.e.,*the events were *in phase*), samples for the two queries would be offset from each other by up to d seconds. In the rewritten version of the query, there is only one stream of sensor tuples which is shared by all events.

In many cases, users may not care that tuples are out of phase with events. In some situations, however, phase may be very important. In such situations, one way the system could improve the phase accuracy of samples while still rewriting multiple event queries into a single join is via *oversampling*, or acquiring some number of (additional) samples every d seconds. The increased phase accuracy of oversampling comes at an increased cost of acquiring additional samples (which may still be less than running multiple queries simultaneously.) For now, we simply allow the user to specify that a query must be phase-aligned by specifying `ON ALIGNED EVENT` in the event clause.

Thus, we have shown that there are several interesting optimization issues in ACQP systems; first, the system must properly order sampling, selection, and aggregation to be truly low power. Second, for frequent event-based queries, rewriting them as a join between an event stream and the `sensors` stream can significantly reduce the rate at which a sensor must acquire samples.

5. POWER SENSITIVE DISSEMINATION AND ROUTING

After the query has been optimized, it is disseminated into the network; dissemination begins with a broadcast of the query from the root of the network. As each node hears the query, it must decide if the query applies locally and/or needs to be broadcast to its children in the routing tree. We say a query q *applies* to a node n if there is a non-zero probability that n will produce results for q. Deciding where a particular query should run is an important ACQP-related decision. Although such decisions occur in other distributed query

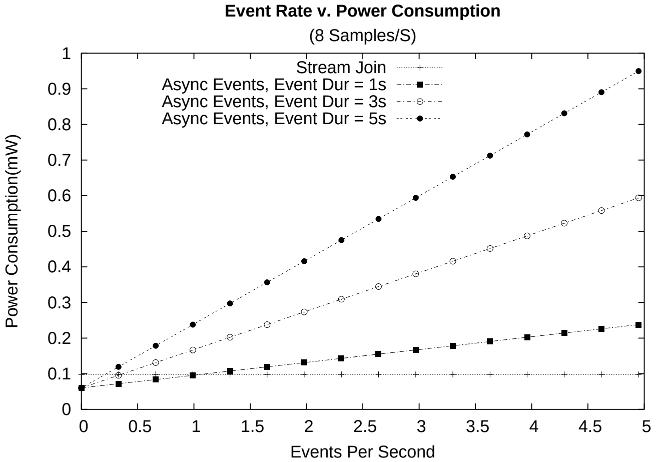


Fig. 7. The cost of processing event-based queries as asynchronous events versus joins.

processing environments, the costs of incorrectly initiating queries in ACQP environments like TinyDB can be unusually high, as we will show.

If a query does not apply at a particular node, and the node does not have any children for which the query applies, then the entire subtree rooted at that node can be excluded from the query, saving the costs of disseminating, executing, and forwarding results for the query across several nodes, significantly extending the node's lifetime.

Given the potential benefits of limiting the scope of queries, the challenge is to determine when a node or its children need not participate in a particular query. One situation arises with constant-valued attributes (*e.g.,*`nodeid` or `location` in a fixed-location network) with a selection predicate that indicates the node need not participate. We expect that such queries will be very common, especially in interactive workloads where users are exploring different parts of the network to see how it is behaving. Similarly, if a node knows that none of its children currently satisfy the value of some selection predicate, perhaps, because they have constant (and known) attribute values outside the predicate's range, it need not forward the query down the routing tree. To maintain information about child attribute values (both constant and changing), we propose a data structure called a *semantic routing tree* (SRT). We describe the properties of SRTs in the next section, and briefly outline how they are created and maintained.

5.1 Semantic Routing Trees

An SRT is a routing tree (similar to the tree discussed in Section 2.3 above) designed to allow each node to efficiently determine if any of the nodes below it will need to participate in a given query over some constant attribute A. Traditionally, in sensor networks, routing tree construction is done by having nodes pick a parent with the most reliable connection to the root (highest *link quality*.) With SRTs, we argue that the choice of parent should include some consideration of semantic properties as well. In general, SRTs are most applicable when there are several parents of comparable link quality. A link-quality-based parent selection algorithm, such as the one described in [Woo and Culler 2001], should be used in conjunction with the SRT to prefilter parents made available to the SRT.

Conceptually, an SRT is an index over A that can be used to locate nodes that have data relevant to the query. Unlike traditional indices, however, the SRT is an overlay on the network. Each node stores a single unidimensional interval representing the range of A values beneath each of its children. When a query q with a predicate over A arrives at a node n, n checks to see if any child's value of A overlaps the query range of A in q. If so, it prepares to receive results and forwards the query. If no child overlaps, the query is not forwarded. Also, if the query also applies locally (whether or not it also applies to any children) n begins executing the query itself. If the query does not apply at n or at any of its children, it is simply forgotten.

Building an SRT is a two phase process: first the *SRT build request* is flooded (retransmitted by every mote until all motes have heard the request) down the network. This request includes the name of the attribute A over which the tree should be built. As a request floods down the network, a node n may have several possible choices of parent, since, in general, many nodes in radio range may be closer to the root. If n has children, it forwards the request on to them and waits until they reply. If n has no children, it chooses a node p from available parents to be its parent, and then reports the value of A to p in a *parent selection message*. If n *does* have children, it records the child's value of A along with its id. When it has heard from all of its children, it chooses a parent and sends a selection message indicating the range of values of A which it and its descendents cover. The parent records this interval with the id of the child node and proceeds to choose its own parent in the same manner, until the root has heard from all of its children. Because children can fail or move away, nodes also have a timeout which is the maximum time they will wait to hear from a child; after this period is elapsed, the child is removed from the child list. If the child reports after this timeout, it is incorporated into the SRT as if it were a new node (see Section 5.2 below).

Figure 8 shows an SRT over the X coordinate of each node on an Cartesian grid. The query arrives at the root, is forwarded down the tree, and then only the gray nodes are required to participate in the query (note that node 3 must forward results for node 4, despite the fact that its own location precludes it from participation.)

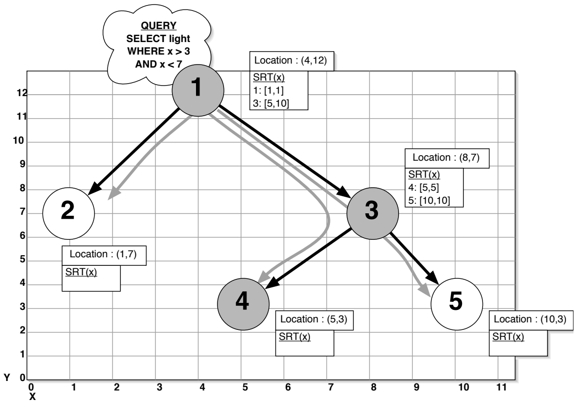


Fig. 8. A semantic routing tree in use for a query. Gray arrows indicate flow of the query down the tree, gray nodes must produce or forward results in the query.

SRTs are analogous to indices in traditional database systems; to create one in TinyDB, the `CREATE SRT` command can be used – its syntax is essentially similar to the `CREATE INDEX` command in SQL:

```
CREATE SRT loc ON sensors (xloc,yloc) ROOT 0
```

Where the `ROOT` annotation indicates the nodeid where the SRT should be rooted from – by default, the value will be 0, but users may wish to create SRTs rooted at other nodes to facilitate event-based queries that frequently radiate from a particular node.

5.2 Maintaining SRTs

Even though SRTs are limited to constant attributes, some SRT maintenance must occur. In particular, new nodes can appear, link qualities can change, and existing nodes can fail.

Both node appearances and changes in link quality can require a node to switch parents. To do this, the node sends a parent selection message to its new parent, n. If this message changes the range of n's interval, it notifies its parent; in this way, updates can propagate to the root of the tree.

To handle the disappearance of a child node, parents associate an *active query id* and *last epoch* with every child in the SRT (recall that an epoch is the period of time between successive samples.) When a parent p forwards a query q to a child c, it sets c's active query id to the id of q and sets its last epoch entry to 0. Every time p forwards or aggregates a result for q from c, it updates c's last epoch with the epoch on which the result was received. If p does not hear c for some number of epochs t, it assumes c has moved away, and removes its SRT entry. Then, p sends a request asking its remaining children to retransmit their ranges. It uses this information to construct a new interval. If this new interval differs in size from the previous interval, p sends a parent selection message up the routing tree to reflect this change. We study the costs of SRT maintenance in Section 5.4 below.

Finally, we note that, by using these maintenance rules, it is possible to support SRTs over non-constant attributes, although if those attributes change quickly, the cost of propagating interval-range changes could be prohibitive.

5.3 Evaluation of Benefit of SRTs

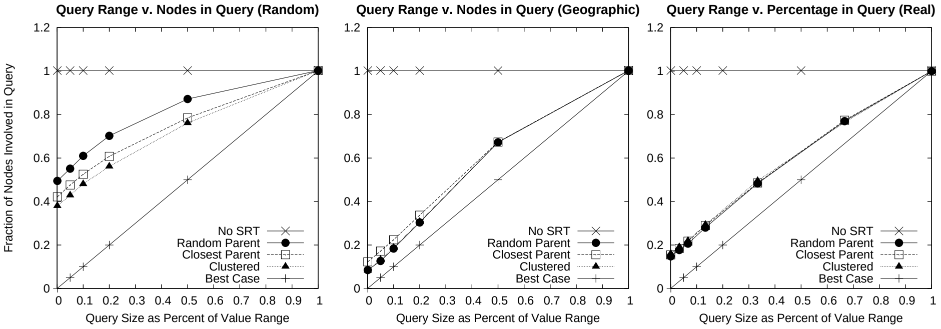
The benefit that an SRT provides is dependent on the quality of the clustering of children beneath parents. If the descendents of some node n are clustered around the value of the index attribute at n, then a query that applies to n will likely also apply to its descendents. This can be expected for location attributes, for example, since network topology is correlated with geography.

We simulate the benefits of an SRT because large networks of the type where we expect these data structures to be useful are just beginning to come online, so only a small-number of fixed real-world topologies are available. We include in our simulation experiments using a connectivity data file collected from one such real-world deployment. We evaluate the benefit of SRTs in terms of number of active nodes; inactive nodes incur no cost for a given query, expending energy only to keep their processors in an idle state and to listen to their radios for the arrival of new queries.

We study three policies for SRT parent selection. In the first, *random* approach, each node picks a random parent from the nodes with which it can communicate reliably. In the second, *closest-parent* approach, each parent reports the value of its index attribute with the SRT-build request, and children pick the parent whose attribute value is closest to their own. In the *clustered* approach, nodes select a parent as in the closest-parent approach, except, if a node hears a sibling node send a parent selection message, it *snoops* on the message to determine its siblings parent and value. It then picks its own parent (which could be the same as one of its siblings) to minimize the spread of attribute values underneath all of its available parents.

We studied these policies in a simple simulation environment – nodes were arranged on an nxn grid and were asked to choose a constant attribute value from some distribution (which we varied between experiments.) We used a perfect (lossless) connectivity model where each node could talk to its immediate neighbors in the grid (so routing trees were n nodes deep), and each node had 8 neighbors (with 3 choices of parent, on average.) We compared the total number of nodes involved in range queries of different sizes for the three SRT parent selection policies to the *best-case* approach and the *no SRT* approach. The *best-case* approach would only result if exactly those nodes that overlapped the range predicate were activated, which is not possible in our topologies but provides a convenient lower bound. In the *no SRT* approach, all nodes participate in each query.

We experimented with several of sensor value distributions. In the *random* distribution, each constant attribute value was randomly and uniformly selected from the interval [0,1000]. In the *geographic* distribution, (one-dimensional) sensor values were computed based on a function of a node's x and y position in the grid, such that a node's value tended to be highly correlated to the values of its neighbors.



(a)Random (b)Geographic (c)Real

Fig. 9. Number of nodes participating in range queries of different sizes for different parent selection policies in a semantic routing tree (20x20 grid, 400 nodes, each point average of 500 queries of the appropriate size.) The three graphs represent three different sensor-value distributions; see the text for a description of each of these distribution types.

Finally, for the *real* distribution, we used a network topology based on data collected from a network of 54 motes deployed throughout the Intel-Research, Berkeley lab. The SRT was built over the node's location in the lab, and the network connectivity was derived by identifying pairs of motes with a high probability of being able to successfully communicate with each other[11].

Figure 9 shows the number of nodes that participate in queries over variably-sized query intervals (where the interval size is shown on the X axis) of the attribute space in a 20x20 grid. The interval for queries was randomly selected from the uniform distribution. Each point in the graph was obtained by averaging over five trials for each of the three parent selection policies in each of the sensor value distributions (for a total of 30 experiments).For each interval size s, 100 queries were randomly constructed, and the average number of nodes involved in each query was measured.

For all three distributions, the clustered approach was superior to other SRT algorithms, beating the random approach by about 25% and the closest parent approach by about 10% on average. With the geographic and real distributions, the performance of the clustered approach is close to optimal: for most ranges, all of the nodes in the range tend to be co-located, so few intermediate nodes are required to relay information for queries in which they themselves are not participating. The fact that the results from real topology closely matches the geographic distribution, where sensors' values and topology are perfectly correlated, is encouraging and suggests that SRTs will work well in practice.

Figure 10 shows several visualizations of the topologies which are generated by the *clustered* (Figure 10(a)) and random (Figure 10(b)) SRT generation approaches for an 8x8 network. Each node represents a sensor, labeled with its ID and the distribution of the SRT subtree rooted underneath it. Edges represent the routing tree. The gray nodes represent the nodes that would participate in the query $\texttt{400} < \texttt{A} < \texttt{500}$. On this small grid, the two approaches perform similarly, but the variation in structure which results is quite evident – the random approach tends to be of more uniform depth, whereas the clustered approach

[11] The probability threshold in this case was 25%, which is the same as the probability the TinyOS/TinyDB routing layer use to determine if a neighboring node is of sufficiently high quality to be considered as a candidate parent.

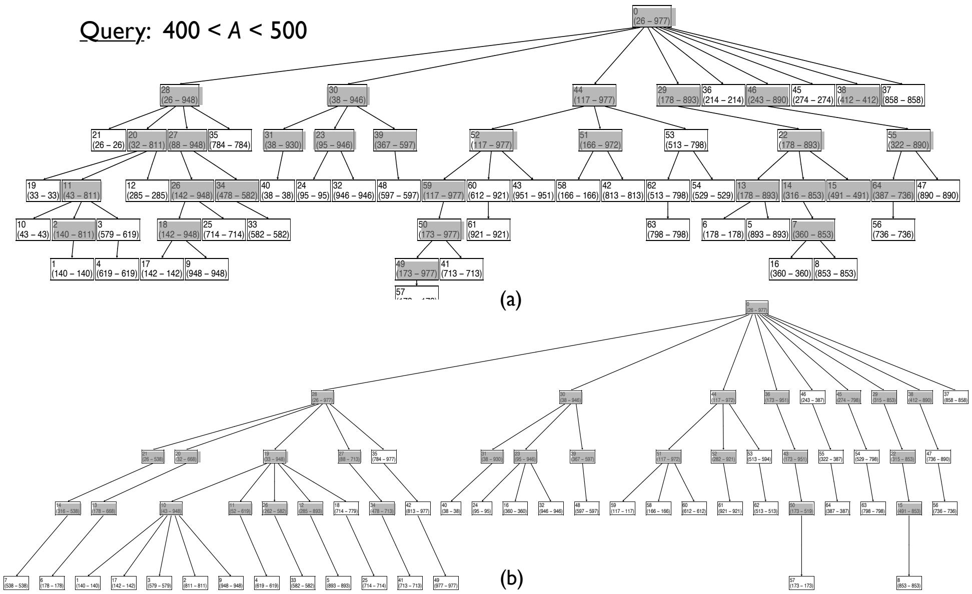


Fig. 10. Visualizations of the (a) clustered and (b) random topologies, with a query region overlaid on top of them. Node 0, the root in Figures (a) and (b), is at the center of the graph.

leads to longer sequences of nodes with nearby values. Note that the labels in this figure are not intended to be readable – the important point is the overall pattern of nodes that are explored by the two approaches.

5.4 Maintenance Costs of SRTs

As the previous results show, the benefit of using an SRT can be substantial. There are, however, maintenance and construction costs associated with SRTs, as discussed above. Construction costs are comparable to those in conventional sensor networks (which also have a routing tree), but slightly higher due to the fact that parent selection messages are explicitly sent, whereas parents do not always require confirmation from their children in other sensor network environments.

We conducted an experiment to measure the cost of selecting a new parent, which requires a node to notify its old parent of its decision to move and send its attribute value to its new parent. Both the new and old parent must then update their attribute interval information and propagate any changes up the tree to the root of the network. In this experiment, we varied the probability with which any node switches parents on any given epoch from .001 to .2. We did not constrain the extent of the query in this case – all nodes were assumed to participate. Nodes were allowed to move from their current parent to an arbitrary new parent, and multiple nodes could move on a given epoch. The experimental parameters were the same as above. We measured the average number of maintenance messages generated by movement across the whole network. The results are shown in Figure 11. Each point represents the average of 5 trials, and each trial consists of 100 epochs. The three lines represent the three policies; the amount of movement varies along the X axis, and the number of maintenance messages per epoch is shown on the Y axis.

Without maintenance, each active node (within the query range) sends one message per epoch, instead of every node being required to transmit. Figure 11 suggests that for low movement rates, the maintenance costs of the SRT approach are small enough that it remains attractive – if 1% of the nodes move on a given epoch, the cost is about 30 messages, which is substantially less than the number of messages saved by using an SRT for most query ranges. If 10% of the nodes move, the maintenance cost grows to about 300, making the benefit of SRTs less clear.

To measure the amount of movement expected in practice, we measured movement rates in traces collected from two real-world monitoring deployments; in both cases, the nodes were stationary but employed a routing algorithm that attempted to select the best parent over time. In the 3 month, 200 node Great Duck Island Deployment nodes switched parents between successive result reports with a .9% ($\sigma = .9\%$) chance, on average. In the 54 node Intel-Berkeley lab dataset, nodes switched with a 4.3% ($\sigma = 3.0\%$) chance. Thus, the amount of parent switching varies markedly from deployment to deployment. One reason for the variation is that the two deployments use different routing algorithms. In the case of the Intel-Berkeley deployment, the algorithm was apparently not optimized to minimize the likelihood of switching.

Figure 11 also shows that the different schemes for building SRTs result in different maintenance costs. This is because the average depth of nodes in the topologies varies from one approach to the other (7.67 in Random, 10.47 in Closest, and 9.2 in Clustered) and because the spread of values underneath a particular subtree varies depending on the approach used to build the tree. A deeper tree generally results in more messages being sent up the tree as path lengths increase. The closest parent scheme results in deep topologies because no preference is given towards parents with a wide spread of values, unlike the clustered approach which tends to favor selecting a parent that is a member of a preexisting, wide interval. The random approach is shallower still because nodes simply select the first parent that broadcasts, resulting in minimally deep trees.

Finally, we note that the cost of joining the network is strictly dominated by the cost of moving parents, as there is no old parent to notify. Similarly, a node disappearing is dominated by this movement cost, as there is no new parent to notify.

5.5 SRT Observations

SRTs provide an efficient mechanism for disseminating queries and collecting query results for queries over constant attributes. For attributes that are highly correlated amongst neighbors in the routing tree (*e.g.*,location), SRTs can reduce the number of nodes that must disseminate queries and forward the continuous stream of results from children by nearly an order of magnitude. SRTs have the substantial advantage over a centralized index structure in that they do not require complete topology and sensor value information to be collected at the root of the network, which will be quite expensive to collect and will be difficult to keep consistent as connectivity and sensor values change.

SRT maintenance costs appear to be reasonable for at least some real-world deployments. Interestingly, unlike traditional routing trees in sensor networks, there is a substantial cost (in terms of network messages) for switching parents in an SRT. This suggests that one metric by which routing layer designers might evaluate their implementations is rate of parent-switching.

For real world deployments, we expect that SRTs will offer substantial benefits. Although there are no benchmarks or definitive workloads for sensor network databases, we

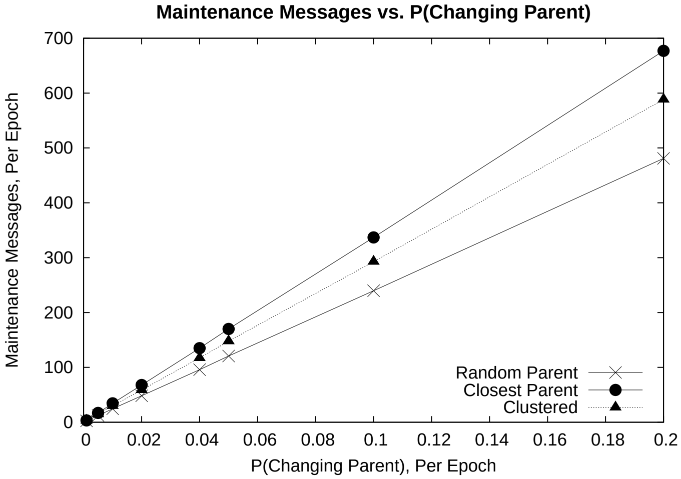


Fig. 11. Maintenance costs (in measured network messages) for different SRT parent selection policies with varying probabilities of node movement. Probabilities and costs are per epoch. Each point is the average of 5 runs, and where each run is 100 epochs long.

anticipate that many queries will be over narrow geographic areas – looking, for example, at single rooms or floors in a building, or nests, trees, or regions, in outdoor environments as on Great Duck Island; other researchers have noted the same need for constrained querying [Yao and Gehrke 2002; Mainwaring et al. 2002]. In a deployment like the Intel-Berkeley lab, if queries are over individual rooms or regions of the lab, Figure 9 shows that substantial performance gains can be had. For example, the 2 of the 54 motes are in the main conference room; 7 of the 54 are in the seminar area; both of these queries can be evaluated using less that 30% of the network.

We note two promising future extensions to SRTs. First, rather than storing just a single interval at every subtree, a variable number of intervals could be kept. This would allow nodes to more accurately summarize the range of values beneath them, and increase the benefit of the approach. Second, when selecting a parent, even in the clustered approach, nodes do not currently have access to complete information about the subtree underneath a potential parent, particularly as nodes move in the network or come and go. It would be interesting to explore a continuous SRT construction process, where parents periodically broadcast out updated intervals, giving current and potential children an option to move to a better subtree and improve the quality of the SRT.

6. PROCESSING QUERIES

Once queries have been disseminated and optimized, the query processor begins executing them. Query execution is straightforward, so we describe it only briefly. The remainder of the section is devoted to the ACQP-related issues of prioritizing results and adapting sampling and delivery rates. We present simple schemes for prioritizing data in selection queries, briefly discuss prioritizing data in aggregation queries, and then turn to adaptation. We discuss two situations in which adaptation is necessary: when the radio is highly contened and when power consumption is more rapid than expected.

6.1 Query Execution

Query execution consists of a simple sequence of operations at each node during every epoch: first, nodes sleep for most of an epoch; then they wake, sample sensors and apply operators to data generated locally and received from neighbors, and then deliver results to their parent. We (briefly) describe ACQP-relevant issues in each of these phases.

Nodes sleep for as much of each epoch as possible to minimize power consumption. They wake up only to sample sensors and relay and deliver results. Because nodes are time synchronized, parents can ensure that they awake to receive results when a child tries to propagate a message[12]. The amount of time, t_{awake} that a sensor node must be awake to successfully accomplish the latter three steps above is largely dependent on the number of other nodes transmitting in the same radio cell, since only a small number of messages per second can be transmitted over the single shared radio channel. We discuss the communication scheduling approach in more detail in the next section.

TinyDB uses a simple algorithm to scale t_{awake} based on the neighborhood size, which is measures by snooping on traffic from neighboring nodes. Note, however, that there are situations in which a node will be forced to drop or combine results as a result of the either t_{awake} or the sample interval being too short to perform all needed computation and communication. We discuss policies for choosing how to aggregate data and which results to drop in Section 6.3.

Once a node is awake, it begins sampling and filtering results according to the plan provided by the optimizer. Samples are taken at the appropriate (current) sample rate for the query, based on lifetime computations and information about radio contention and power consumption (see Section 6.4 for more information on how TinyDB adapts sampling in response to variations during execution.) Filters are applied and results are routed to join and aggregation operators further up the query plan.

Finally, we note that in event-based queries, the `ON EVENT` clause must be handled specially. When an event fires on a node, that node disseminates the query, specifying itself as the query root. This node collects query results, and delivers them to the basestation or a local materialization point.

6.1.1 *Communication Scheduling and Aggregate Queries.*

When processing aggregate queries, some care must be taken to coordinate the times when parents and children are awake, so that parent nodes have access to their children's readings before aggregating. The basic idea is to subdivide the epoch into a number of intervals, and assign nodes to intervals based on their position in the routing tree. Because this mechanism makes relatively efficient use of the radio channel and has good power consumption characteristics, TinyDB uses this scheduling approach for all queries (not just aggregates).

In this *slotted approach*, each epoch is divided into a number of fixed-length time intervals. These intervals are numbered in reverse order such that interval 1 is the last interval in the epoch. Then, each node is assigned to the interval equal to its level, or number of hops from the root, in the routing tree. In the interval preceding their own, nodes listen to their radios, collecting results from any child nodes (which are one level below them in the tree, and thus communicating in this interval.) During a node's interval, if it is aggregat-

[12] Of course, there is some imprecision in time synchronization between devices. In general, we can tolerate a fair amount of imprecision by introducing a buffer period, such that parent s wake up several milliseconds before and stay awake several milliseconds longer than their children.

ing, it computes the partial state record consisting of the combination of any child values it heard with its own local readings. After this computation, it transmits either its partial state record or raw sensor readings up the network. In this way, information travels up the tree in a staggered fashion, eventually reaching the root of the network during interval 1.

Figure 12 illustrates this in-network aggregation scheme for a simple COUNT query that reports the number of nodes in the network. In the figure, time advances from left to right, and different nodes in the communication topology are shown along the Y axis. Nodes transmit during the interval corresponding to their depth in the tree, so H, I, and J transmit first, during interval 4, because they are at level 4. Transmissions are indicated by arrows from sender to receiver, and the numbers in circles on the arrows represent COUNTs contained within each partial state record. Readings from these three nodes are combined, via the COUNT merging function, at nodes G and F, both of which transmit new partial state records during interval 3. Readings flow up the tree in this manner until they reach node A, which then computes the final count of 10. Notice that motes are idle for a significant portion of each epoch so they can enter a low power sleeping state. A detailed analysis of the accuracy and benefit of this approach in TinyDB can be found in [Madden 2003].

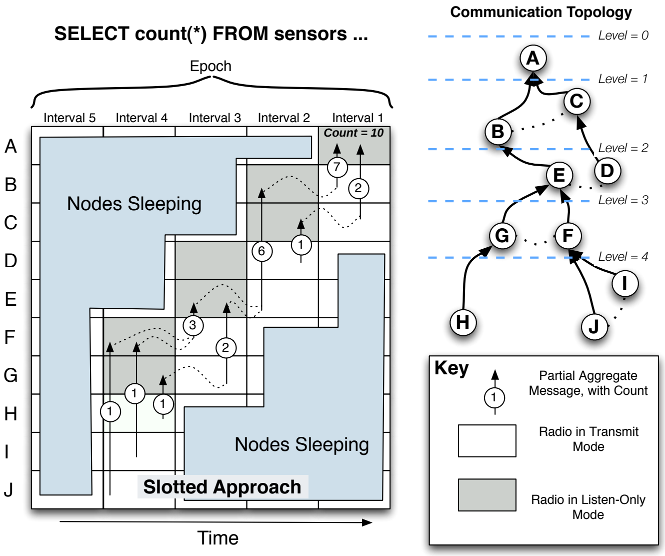


Fig. 12. Partial state records flowing up the tree during an epoch using interval-based communication.

6.2 Multiple Queries

We note that, although TinyDB supports multiple queries running simultaneously, we have not focused on multi-query optimization. This means that, for example, SRTs are shared between queries, but sample acquisition is not: if two queries need a reading within a few milliseconds of each other, this will cause both to acquire that reading. Similarly, there is no effort to optimize communication scheduling between queries: transmissions of one query are scheduled independently from any other query. We hope to explore these issues as a part of our long-term sensor network research agenda.

6.3 Prioritizing Data Delivery

Once results have been sampled and all local operators have been applied, they are enqueued onto a radio queue for delivery to the node's parent. This queue contains both tuples from the local node as well as tuples that are being forwarded on behalf of other nodes in the network. When network contention and data rates are low, this queue can be drained faster than results arrive. However, because the number of messages produced during a single epoch can vary dramatically, depending on the number of queries running, the cardinality of joins, and the number of groups and aggregates, there are situations when the queue will overflow. In these situations, the system must decide if it should discard the overflow tuple, discard some other tuple already in the queue, or combine two tuples via some aggregation policy.

The ability to make runtime decisions about the value of an individual data item is central to ACQP systems, because the cost of acquiring and delivering data is high, and because of these situations where the rate of data items arriving at a node will exceed the maximum delivery rate. A simple conceptual approach for making such runtime decisions is as follows: whenever the system is ready to deliver a tuple, send the result that will most improve the "quality" of the answer that the user sees. Clearly, the proper metric for quality will depend on the application: for a raw signal, root-mean-square (RMS) error is a typical metric. For aggregation queries, minimizing the confidence intervals of the values of group records could be the goal [Raman et al. 2002]. In other applications, users may be concerned with preserving frequencies, receiving statistical summaries (average, variance, or histograms), or maintaining more tenuous qualities such as signal "shape".

Our goal is not to fully explore the spectrum of techniques available in this space. Instead, we have implemented several policies in TinyDB to illustrate that substantial quality improvements are possible given a particular workload and quality metric. Generalizing concepts of quality and implementing and exploring more sophisticated prioritization schemes remains an area of future work.

There is a large body of related work on approximation and compression schemes for streams in the database literature (*e.g.,*[Garofalakis and Gibbons 2001; Chakrabarti et al. 2001]), although these approaches typically focus on the problem of building histograms or summary structures over the streams rather than trying to preserve the (in order) signal as best as possible, which is the goal we tackle first. Algorithms from signal processing, such as Fourier analysis and wavelets are likely applicable, although the extreme memory and processor limitations of our devices and the online nature of our problem (*e.g.,*choosing which tuple in an overflowing queue to evict) make them tricky to apply. We have begun to explore the use of wavelets in this context; see [Hellerstein et al. 2003] for more information on our initial efforts.

6.3.1 *Policies for Selection Queries.* We begin with a comparison of three simple prioritization schemes, *naive*, *winavg*, and *delta* for simple selection queries, turning our attention to aggregate queries in the next section. In the *naive* scheme no tuple is considered more valuable than any other, so the queue is drained in a FIFO manner and tuples are dropped if they do not fit in the queue.

The *winavg* scheme works similarly, except that instead of dropping results when the queue fills, the two results at the head of the queue are averaged to make room for new results. Since the head of the queue is now an average of multiple records, we associate a count with it.

In the *delta* scheme, a tuple is assigned an initial score relative to its difference from the most recent (in time) value successfully transmitted from this node, and at each point in time, the tuple with the highest score is delivered. The tuple with the lowest score is evicted when the queue overflows. Out of order delivery (in time) is allowed. This scheme relies on the intuition that the largest changes are probably interesting. It works as follows: when a tuple t with timestamp T is initially enqueued and scored, we mark it with the timestamp R of this most recently delivered tuple r. Since tuples can be delivered out of order, it is possible that a tuple with a timestamp between R and T could be delivered next (indicating that r was delivered out of order), in which case the score we computed for t as well as its R timestamp are now incorrect. Thus, in general, we must rescore some enqueued tuples after every delivery. The delta scheme is similar to the value-deviation metric used in [Garofalakis and Gibbons 2001] for minimizing deviation between a source and a cache although value-deviation does not include the possibility of out of order delivery.

We compared these three approaches on a single mote running TinyDB. To measure their effect in a controlled setting, we set the sample rate to be a fixed number K faster than the maximum delivery rate (such that 1 of every K tuples was delivered, on average) and compared their performance against several predefined sets of sensor readings (stored in the EEPROM of the device.) In this case, delta had a buffer of 5 tuples; we performed reordering of out of order tuples at the basestation. To illustrate the effect of winavg and delta, Figure 13 shows how delta and winavg approximate a high-periodicity trace of sensor readings generated by a shaking accelerometer. Notice that delta is considerably closer in shape to the original signal in this case, as it is tends to emphasize extremes, whereas average tends to dampen them.

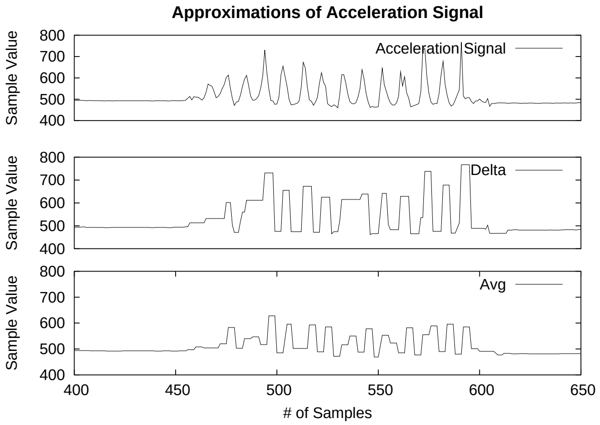


Fig. 13. An acceleration signal (top) approximated by a delta (middle) and an average (bottom), K=4.

We also measured RMS error for this signal as well as two others: a square wave-like signal from a light sensor being covered and uncovered, and a slow sinusoidal signal generated by moving a magnet around a magnetometer. The error for each of these signals and techniques is shown in Table V. Although delta appears to match the shape of the acceleration signal better, its RMS value is about the same as average's (due to the few peaks that delta incorrectly merges together.) Delta outperforms either other approach for the fast changing step-functions in the light signal because it does not smooth edges as much as average.

We now turn our attention to result prioritization for aggregate queries.

Table V. RMS Error for Different Prioritization Schemes and Signals (1000 Samples, Sample Interval = 64ms)

	Accel	Light (Step)	Magnetometer (Sinusoid)
Winavg	64	129	54
Delta	63	81	48
Naive	77	143	63

6.3.2 *Policies for Aggregate Queries.* The previous section focused on prioritizing result collection in simple selection queries. In this section, we look instead at aggregate queries, illustrating a class of *snooping based* techniques first described in the TAG system [Madden et al. 2002] that we have implemented for TinyDB. We consider aggregate queries of the form:

```
SELECT $f_{agg}$($a_1$)
  FROM sensors
  GROUP BY $a_2$
  SAMPLE PERIOD $x$
```

Recall that this query computes the value of f_{agg} applied to the value of a_1 produced by each device every x seconds.

Interestingly, for queries with few or no groups, there is a simple technique that can be used to prioritize results for several types of aggregates. This technique, called *snooping*, allows nodes to locally suppress local aggregate values by listening to the answers that neighboring nodes report and exploiting the semantics of aggregate functions, and is also used in [Madden et al. 2002]. Note that this snooping can be done for free due to the broadcast nature of the radio channel. Consider, for example, a MAX query over some attribute a – if a node n hears a value of a greater than its own locally computed partial MAX, it knows that its local record is low priority, and assigns it a low score or suppresses it altogether. Conversely, if n hears many neighboring partial MAXs over a that are less than its own partial aggregate value, it knows that its local record is more likely to be a maximum, and assigns it a higher score.

Figure 14 shows a simple example of snooping for a MAX query – node 2 is can score its own MAX value very low when it hears a MAX from node 3 that is larger than its own.

This basic technique applies to all monotonic, exemplary aggregates: MIN, MAX, TOP-N, etc., since it is possible to deterministically decide whether a particular local result could appear in the final answer output at the top of the network. For dense network topologies where there is ample opportunity for snooping, this technique produces a dra-

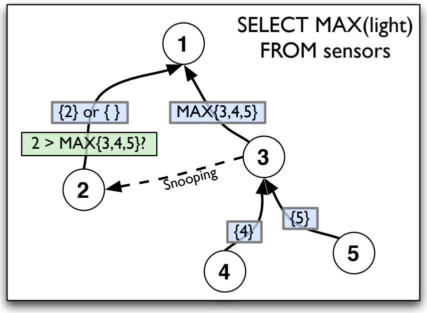


Fig. 14. Snooping reduces the data nodes must send in aggregate queries. Here node 2's value can be suppressed if it is less than the maximum value snooped from nodes 3,4, and 5.

matic reduction in communication, since at every intermediate point in the routing tree, only a small number of node's values will actually need to be transmitted.

It is also possible to glean some information from snooping in other aggregates as well – for example, in an AVERAGE query, nodes may rank their own results lower if they hear many siblings with similar sensor readings. For this approach to work, parents must cache a count of recently heard children and assume children who do not send a value for an average have the same value as the average of their siblings' values, since otherwise outliers will be weighted disproportionately. This technique of assuming that missing values are the same as the average of other reported values can be used for many summary statistics: variance, sum, and so on. Exploring more sophisticated prioritization schemes for aggregate queries is an important area of future work.

In the previous sections, we demonstrated how prioritization of results can be used improve the overall quality of that data that are transmitted to the root when some results must be dropped or aggregated. Choosing the proper policies to apply *in general*, and understanding how various existing approximation and prioritization schemes map into ACQP is an important future direction.

6.4 Adapting Rates and Power Consumption

We saw in the previous sections how TinyDB can exploit query semantics to transmit the most relevant results when limited bandwidth or power is available. In this section, we discuss selecting and adjusting sampling and transmission rates to limit the frequency of network-related losses and fill rates of queues. This adaptation is the other half of the runtime techniques in ACQP: because the system *can* adjust rates, significant reductions can be made in the frequency with which data prioritization decisions must be made. These techniques are simply not available in non-acquisitional query processing systems.

When initially optimizing a query, TinyDB's optimizer chooses a transmission and sample rate based on current network load conditions, and requested sample rates and lifetimes. However, static decisions made at the start of query processing may not be valid after many days running the same continuous query. Just as adaptive query processing techniques like eddies [Avnur and Hellerstein 2000], Tukwila [Ives et al. 1999], and Query Scrambling [Urhan et al. 1998] dynamically reorder operators as the execution environment changes, TinyDB must react to changing conditions – however, unlike in previous adaptive query processing systems, failure to adapt in TinyDB can cripple the system, reducing data flow to a trickle or causing the system to severely miss power budget goals.

We study the need for adaptivity in two contexts: network contention and power consumption. We first examine network contention. Rather than simply assuming that a specific transmission rate will result in a relatively uncontested network channel, TinyDB monitors channel contention and adaptively reduces the number of packets transmitted as contention rises. This backoff is very important: as the *4 motes* line of Figure 15 shows, if several nodes try to transmit at high rates, the total number of packets delivered is substantially less than if each of those nodes tries to transmit at a lower rate. Compare this line with the performance of a single node (where there is no contention) – a single node does not exhibit the same falling off because there is no contention (although the percentage of successfully delivered packets does fall off.) Finally, the *4 motes adaptive* line does not have the same precipitous performance because it is able to monitor the network channel and adapt to contention.

Note that the performance of the adaptive approach is slightly less than the non-adaptive approach at 4 and 8 samples per second as backoff begins to throttle communication in this regime. However, when we compared the percentage of successful transmission attempts at 8 packets per second, the adaptive scheme achieves twice the success rate of the non-adaptive scheme, suggesting the adaptation is still effective in reducing wasted communication effort, despite the lower utilization.

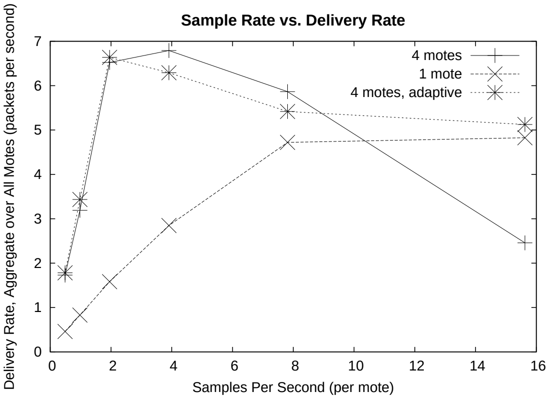


Fig. 15. Per-mote sample rate versus aggregate delivery rate.

The problem with reducing the transmission rate is that it will rapidly cause the network queue to fill, forcing TinyDB to discard tuples using the semantic techniques for victim selection presented in Section 6.3 above. We note, however, that had TinyDB not chosen to slow its transmission rate, fewer total packets would have been delivered. Furthermore, by choosing which packets to drop using semantic information derived from the queries (rather than losing some random sample of them), TinyDB is able to substantially improve the quality of results delivered to the end user. To illustrate this in practice, we ran a selection query over four motes running TinyDB, asking them each to sample data at 16 samples per second, and compared the quality of the delivered results using an adaptive-backoff version of our delta approach to results over the same dataset without adaptation or result prioritization. We show here traces from two of the nodes on the left and right of Figure 16. The top plots show the performance of the adaptive delta, the middle plots show the non-adaptive case, and the bottom plots show the the original signals (which were stored in EEPROM to allow repeatable trials.) Notice that the delta scheme does substantially better in both cases.

6.4.1 *Measuring Power Consumption.* We now turn to the problem of adapting tuple delivery rates to meet specific lifetime requirements in response to incorrect sample rates computed at query optimization time (see Section 3.6). We first note that, using the computations shown in Section 3.6, it is possible to compute a *predicted battery voltage* for a time t seconds into processing a query.

The system can then compare its current voltage to this predicted voltage. By assuming that voltage decays linearly we can *re-estimate* the power consumption characteristics of the device (*e.g.,*the costs of sampling, transmitting, and receiving) and then re-run our lifetime calculation. By re-estimating these parameters, the system can ensure that this new lifetime calculation tracks the actual lifetime more closely.

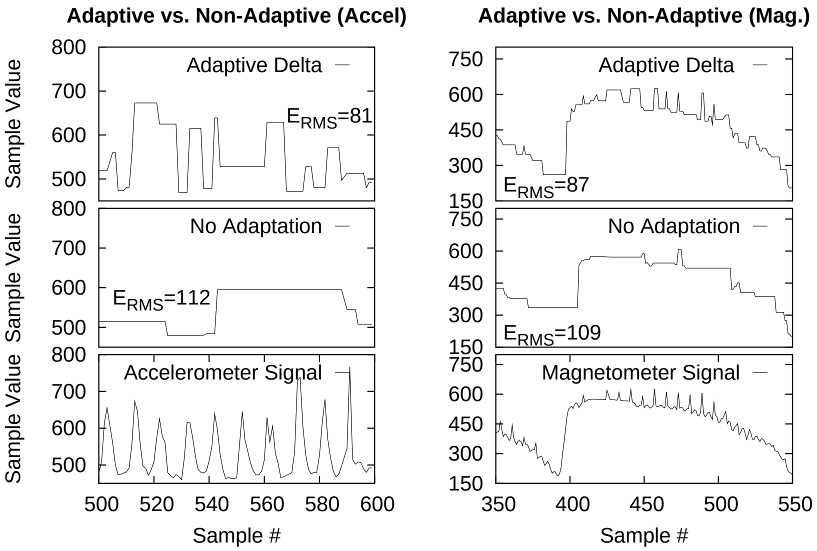


Fig. 16. Comparison of delivered values (bottom) versus actual readings for from two motes (left and right) sampling at 16 packets per second and sending simultaneously. Four motes were communicating simultaneously when this data was collected.

Although this calculation and re-optimization are straightforward, they serve an important role by allowing TinyDB motes to satisfy occasional ad-hoc queries and relay results for other nodes without compromising lifetime goals of long-running monitoring queries.

Finally, we note that incorrect measurements of power consumption may also be due to incorrect estimates of the cost of various phases of query processing, or may be as a result of incorrect selectivity estimation. We cover both by tuning sample rate. As future work, we intend to explore adaptation of optimizer estimates and ordering decisions (in the spirit of other adaptive work [Hellerstein et al. 2000]) and the effect of frequency of re-estimation on lifetime.

7. SUMMARY OF ACQP TECHNIQUES

This completes our discussion of the novel issues and techniques that arise when taking an acquisitional perspective on query processing. In summary, we first discussed important

Table VI. Summary of acquisitional query processing techniques in TinyDB.

Technique (Section)	Summary
Event-based Queries (3.5)	Avoid polling overhead
Lifetime Queries (3.6)	Satisfy user-specified longevity constraints
Interleaving Acquisition/Predicates (4.2)	Avoid unnecessary sampling costs in selection queries
Exemplary Aggregate Pushdown (4.2.1)	Avoid unnecessary sampling costs in aggregate queries
Event Batching (4.3)	Avoid execution costs when a number of event queries fire
SRT (5.1)	Avoid query dissemination costs or the inclusion of unneeded nodes in queries with predicates over constant attributes
Communication Scheduling (6.1.1)	Disable node's processors and radios during times of inactivity
Data Prioritization (6.3)	Choose most important samples to deliver according to a user-specified prioritization function
Snooping (6.3.2)	Avoid unnecessary transmissions during aggregate queries
Rate Adaptation (6.4)	Intentionally drop tuples to avoid saturating the radio channel, allowing most important tuples to be delivered

aspects of an acquisitional query language, introducing event and lifetime clauses for controlling when and how often sampling occurs. We then discussed query optimization with the associated issues of modeling sampling costs and ordering of sampling operators. We showed how event-based queries can be rewritten as joins between streams of events and sensor samples. Once queries have been optimized, we demonstrated the use of semantic routing trees as a mechanism for efficiently disseminating queries and collecting results. Finally, we showed the importance of prioritizing data according to quality and discussed the need for techniques to adapt the transmission and sampling rates of an ACQP system. Table VI lists the key new techniques we introduced, summarizing what queries they apply to and when they are most useful.

8. RELATED WORK

There has been some recent publication in the database and systems communities on query processing in sensor networks [Intanagonwiwat et al. 2000; Madden et al. 2002; P.Bonnet et al. 2001; Madden and Franklin 2002; Yao and Gehrke 2002]. These papers noted the importance of power sensitivity. Their predominant focus to date has been on *in-network* processing – that is, the pushing of operations, particularly selections and aggregations, into the network to reduce communication. We too endorse in-network processing, but believe that, for a sensor network system to be truly power sensitive, acquisitional issues of when, where, and in what order to sample and which samples to process must be considered. To our knowledge, no prior work addresses these issues.

There is a small body of work related to query processing in mobile environments [Imielinski and Badrinath 1992; Alonso and Korth 1993]. This work is concerned with laptop-like devices that are carried with the user, can be readily recharged every few hours, and, with the exception of a wireless network interface basically have the capabilities of a wired, powered PC. Lifetime-based queries, notions of sampling the associated costs, and runtime issues regarding rates and contention are not considered. Many of the proposed techniques, as well as more recent work on moving object databases (such as [Wolfson et al. 1999]) focus on the highly mobile nature of devices, a situation we are not (yet) dealing with, but which could certainly arise in sensor networks.

Power sensitive query optimization was proposed in [Alonso and Ganguly 1993], although, as with the previous work, the focus is on optimizing costs in traditional mobile devices (*e.g.,*laptops and palmtops), so concerns about the cost and ordering of sampling do not appear. Furthermore, laptop-style devices typically do not offer the same degree of rapid power-cycling that is available on embedded platforms like motes. Even if they did, their interactive, user oriented nature makes it undesirable to turn off displays, network interfaces, etc. because they are doing more than simply collecting and processing data, so there are many fewer power optimizations that can be applied.

Building an SRT is analogous to building an index in a conventional database system. Due to the resource limitations of sensor networks, the actual indexing implementations are quite different. See [Kossman 2000] for a survey of relevant research on distributed indexing in conventional database systems. There is also some similarity to indexing in peer-to-peer systems [Crespo and Garcia-Molina 2002]. However, peer-to-peer systems differ in that they are inexact and not subject to the same paucity of communications or storage infrastructure as sensor networks, so algorithms tend to be storage and communication heavy. Similar indexing issues also appear in highly mobile environments (like

[Wolfson et al. 1999; Imielinski and Badrinath 1992]), but this work relies on a centralized location servers for tracking recent positions of objects.

The observation that interleaving the fetching of attributes and application of operators also arises in the context of compressed databases [Chen et al. 2001], as decompression effectively imposes a penalty for fetching an individual attribute, so it is beneficial to apply selections and joins on already decompressed or easy to decompress attributes.

The `ON EVENT` and `OUTPUT ACTION` clauses in our query language are similar to constructs present in ECA/active databases [Chakravarthy et al. 1994]. There is a long tradition of such work in the database community, and our techniques are much simpler in comparison, as we we have not focused on any of the difficult issues associated with the semantics of event composition or with building a complete language for expressing and efficiently evaluating the triggering of composite events. Work on systems for efficiently determining when an event has fired, such as [Hanson 1996] could be useful in TinyDB. More recent work on continuous query systems [Liu et al. 1999; Chen et al. 2000] describes languages that provide for query processing in response to events or at regular intervals over time. This earlier work, as well as our own work on continuous query processing [Madden et al. 2002], inspired the periodic and event-driven features of TinyDB.

Approximate and best effort caches [Olston and J.Widom 2002], as well as systems for online-aggregation [Raman et al. 2002] and stream query processing [Motwani et al. 2003; Carney et al. 2002] include some notion of data quality. Most of this other work is focused on quality with respect to summaries, aggregates, or staleness of individual objects, whereas we focus on quality as a measure of fidelity to the underlying continuous signal. Aurora [Carney et al. 2002] mentions a need for this kind of metric, but proposes no specific approaches. Work on approximate query processing [Garofalakis and Gibbons 2001] includes a scheme similar to our delta approach, as well as a substantially more thorough evaluation of its merits, but does not consider out of order delivery.

9. CONCLUSIONS AND FUTURE WORK

Acquisitional query processing provides a framework for addressing issues of when, where, and how often data is sampled and which data is delivered in distributed, embedded sensing environments. Although other research has identified the opportunities for query processing in sensor networks, this work is the first to discuss these fundamental issues in an acquisitional framework.

We identified several opportunities for future research. We are currently actively pursuing two of these: first, we are exploring how query optimizer statistics change in acquisitional environments and studying the role of online re-optimization in sample rate and operator orderings in response to bursts of data or unexpected power consumption. Second, we are pursuing more sophisticated prioritization schemes, like wavelet analysis, that can capture salient properties of signals other than large changes (as our delta mechanism does) as well as mechanisms to allow users to express their prioritization preferences.

We believe that ACQP notions are of critical importance for preserving the longevity and usefulness of any deployment of battery powered sensing devices, such as those that are now appearing in biological preserves, roads, businesses, and homes. Without appropriate query languages, optimization models, and query dissemination and data delivery schemes that are cognizant of semantics and the costs and capabilities of the underlying hardware the success of such deployments will be limited.

APPENDIX

A. POWER CONSUMPTION STUDY

This appendix details an analytical study of power consumption on a mote running a typical data collection query.

In this study, we assume that each mote runs a very simple query that transmits one sample of (light, humidity) readings every minute. We assume each mote also listens to its radio for two seconds per one-minute period to receive results from from neighboring devices and obtain access to the radio channel. We assume the following hardware characteristics: a supply voltage of 3V, an Atmega128 processor [Atmel Corporation] that can be set into Power Down Mode and runs off the internal oscillator at 4Mhz, the use of the Taos Photosynthetically Active Light Sensor [TAOS, Inc. 2002] and Sensirion Humidity Sensor [Sensirion 2002], and a ChipCon CC1000 Radio [ChipCon Corporation] transmitting at 433Mhz with 0 dBm output power and -110 dBm receive sensitivity. We further assume the radio can make use of its low-power sampling [13] mode to reduce reception power when no other radios are communicating, and that, on average, each node has ten *neighbors*, or other motes within radio range, period, with one of those neighbors being a child in the routing tree. Radio packets are 50 bytes each, with a 20 byte preamble for synchronization. This hardware configuration represents real-world settings of motes similar to values used in deployments of TinyDB in various environmental monitoring applications.

The percentage of total energy used by various components is shown in Table VII. These results show that the processor and radio together consume the majority of energy for this particular data collection task. Obviously, these numbers change as the number of messages transmitted per period increases; doubling the number of messages sent increases the total power utilization by about 19 percent as a result of the radio spending less time sampling the channel and more time actively receiving. Similarly, if a node must send 5 packets per sample period instead of 1, its total power utilization rises by about 10 percent.

This table does not tell the entire story, however, because the processor must be active during sensing and communication, even though it has very little computation to perform[14].

[13] This mode works by sampling the radio at a low-frequency – say, once every k bit-times, where k is on the order of 100 – and extending the synchronization header, or preamble, on radio packets to be at least $k + \epsilon$ bits, such that a radio using this low-power listening approach will still detect every packet. Once a packet is detected, the receiver begins packet reception at the normal rate. The cost of this technique is that it increases transmission costs significantly.

[14] The requirement that the processor be active during these times is an artifact of the mote hardware. Blue-

Table VII. *Expected Power Consumption for Major Hardware Components,* a query reporting light and accelerometer readings once every minute.

Hardware	Current (mA)	Active Time (s)	% Total Energy
Sensing, Humidity	0.50	0.34	1.43
Sensing, Light	0.35	1.30	3.67
Communication, Sending (70 bytes @ 38.4bps x 2 packets)	10.40	0.03	2.43
Communication, Receive Packets (70 bytes @ 38.4bps x 10 packets)	9.30	0.15	11.00
Communication, Sampling Channel	0.07	0.86	0.31
Processor, Active	5.00	2.00	80.68
Processor, Idle	0.001	58.00	0.47
Average current draw per second: .21 mA			

For example, in the above table, 1.3 seconds are spent waiting for the light sensor to start and produce a sample[15], and another .029 seconds are spent transmitting. Furthermore, the MAC layer on the radio introduces a delay proportional to the number of devices transmitting. To measure this delay, we examined the average delay between 1700 packet arrivals on a network of ten time-synchronized motes attempting to send at the same time. The minimum inter-packet arrival time was about 0.06 seconds; subtracting the expected transmit time of a packet (.007s) suggests that, with 10 nodes, the average MAC delay will be at least $(.06 - .007) \times 5) = 0.265s$. Thus, of the 2 seconds each mote is awake, about 1.6 seconds of that time is spent waiting for the sensors or radio. The total 2 second waking period is selected to allow for variation in MAC delays on individual sensors.

Application computation is almost negligible for basic data collection scenarios: we measured application processing time by running a simple TinyDB query that collects three data fields from the RAM of the processor (incurring no sensing delay) and transmits them over an uncontested radio channel (incurring little MAC delay). We inserted into the query result a measure of the elapsed time from the start of processing until the moment the result begins to be transmitted. The average delay was less than 1/32 (.03125) seconds, which is the minimum resolution we could measure.

Thus, of the 81% of energy spent on the processor, no more than 1% of its cycles are spent in application processing. For the example given here at least 65% of this 81% is spent waiting for sensors, and another 8% waiting for the radio to send or receive. The remaining 26% of processing time is time to allow for multihop forwarding of messages and as slop in the event that MAC delays exceed the measured minimums given above. Summing the processor time spent waiting to send or sending with the percent energy used by the radio itself, we get:

$(0.26 + 0.08) \times \ 0.80 + 0.02 + 0.11 + .003 = .41$

This indicates that about 41% of power consumption in this simple data collection task is due to communication. Similarly, in this example, the percentage of energy devoted to sensing can be computed by summing the energy spent waiting for samples with the energy costs of sampling.

$.65 * .81 + .01 + .04 = .58$

Thus, about 58% of the energy in this case is spent sensing. Obviously, the total percentage of time spent in sensing could be less if sensors that powered up more rapidly were used. When we discuss query optimization in TinyDB in Section 4 we will see a range of sensors with varying costs that would alter the percentages in this section.

B. QUERY LANGUAGE

This appendix provides a complete specification of the syntax of the TinyDB query language as well as pointers to the parts of the text where these constructs are defined. We will use $\{\}$ to denote a set, [] to denote optional clauses, and $<>$ to denote an expression, and italicized text to denote user-specified tokens such as aggregate names, commands, and arithmetic operators. The separator "|" indicates that one or the other of the surrounding

tooth radios, for example, can negotiate channel access independently of the processor. These radios, however, have significantly higher power consumption than the mote radio; see [Leopold et al. 2003] for a discussion of Bluetooth as a radio for sensor networks

[15] On motes, it is possible to start and sample several sensors simultaneously, so the delay for the light and humidity sensors are not additive.

tokens may appear, but not both. Ellipses ("...") indicate a repeating set of tokens, such as fields in the `SELECT` clause or tables in the `FROM` clause.

B.1 Query Syntax

The syntax of queries in the TinyDB query language is as follows:

```
[ON [ALIGNED] EVENT  event-type[{paramlist}]
                    [ boolop  event-type{paramlist} ... ]]
  SELECT [NO INTERLEAVE] <expr>| agg(<expr>) |
                              temporal_agg(<expr>), ...
   FROM [sensors |  storage-point], ...
   [WHERE {<pred>}]
   [GROUP BY {<expr>}]
   [HAVING {<pred>}]
  [OUTPUT ACTION [ command |
                  SIGNAL  event({paramlist}) |
                  (SELECT ... ) ] |
  [INTO STORAGE POINT  bufname]]
  [SAMPLE PERIOD  seconds
         [[FOR  nrounds] |
          [STOP ON  event-type [WHERE <pred>]]]
         [COMBINE { agg(<expr>)}]
         [INTERPOLATE LINEAR]] |
   [ONCE] |
   [LIFETIME  seconds [MIN SAMPLE RATE  seconds]]
```

Each of these constructs are described in more detail in the sections shown in the table VIII.

B.2 Storage Point Creation and Deletion Syntax

The syntax for storage point creation is:

```
CREATE [CIRCULAR] STORAGE POINT  name
SIZE [ ntuples |  nseconds]
```

Table VIII. References to sections in the main text where query language constructs are introduced.

Language Construct	Section
ON EVENT	Section 3.5
SELECT-FROM-WHERE	Section 3
GROUP BY, HAVING	Section 3.3.1
OUTPUT ACTION	Section 3.7
SIGNAL <event>	Section 3.5
INTO STORAGE POINT	Section 3.2
SAMPLE PERIOD	Section 3
FOR	Section 3.2
STOP ON	Section 3.5
COMBINE	Section 3.2
ONCE	Section 3.7
LIFETIME	Section 3.6

```
[( fieldname type [, ... , fieldname type])] |
[AS SELECT ... ]
[SAMPLE PERIOD nseconds]
```

and for deletion:

```
  DROP STORAGE POINT name
```

Both of these constructs are described in Section 3.2.

REFERENCES

ALONSO, R. AND GANGULY, S. 1993. Query optimization in mobile environments. In *Workshop on Foundations of Models and Languages for Data and Objects*. 1–17.

ALONSO, R. AND KORTH, H. F. 1993. Database system issues in nomadic computing. In *ACM SIGMOD*. Washington DC.

ANALOG DEVICES, INC. Adxl202e: Low-cost 2 g dual-axis accelerometer. Tech. rep. `http://products.analog.com/products/info.asp?product=ADXL202`.

ATMEL CORPORATION. Atmel ATMega 128 Microcontroller Datasheet. `http://www.atmel.com/atmel/acrobat/doc2467.pdf`.

AVNUR, R. AND HELLERSTEIN, J. M. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD*. Dallas, TX, 261–272.

BANCILHON, F., BRIGGS, T., KHOSHAFIAN, S., AND VALDURIEZ, P. 1987. FAD, a powerful and simple database language. In *VLDB*.

BROOKE, T. AND BURRELL, J. 2003. From ethnography to design in a vineyard. In *Proceeedings of the Design User Experiences (DUX) Conference*. Case Study.

CARNEY, D., CENTIEMEL, U., CHERNIAK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2002. Monitoring streams - a new class of data management applications. In *VLDB*.

CERPA, A., ELSON, J., D.ESTRIN, GIROD, L., HAMILTON, M., AND ZHAO, J. 2001. Habitat monitoring: Application driver for wireless communications technology. In *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*.

CHAKRABARTI, K., GAROFALAKIS, M., RASTOGI, R., AND SHIM, K. 2001. Approximate query processing using wavelets. *VLDB Journal 10*.

CHAKRAVARTHY, S., KRISHNAPRASAD, V., ANWAR, E., AND KIM, S. K. 1994. Composite events for active databases: Semantics, contexts and detection. In *VLDB*.

CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *First Annual Conference on Innovative Database Research (CIDR)*.

CHEN, J., DEWITT, D., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD*.

CHEN, Z., GEHRKE, J., AND KORN, F. 2001. Query optimization in compressed database systems. In *ACM SIGMOD*.

CHIPCON CORPORATION. *CC1000 Single Chip Very Low Power RF Transceiver Datasheet*. `http://www.chipcon.com`.

CRESPO, A. AND GARCIA-MOLINA, H. 2002. Routing indices for peer-to-peer systems. In *ICDCS*.

CROSSBOW, INC. Wireless Sensor Networks (Mica Motes). `http://www.xbow.com/Products/Wireless_Sensor_Networks.htm`.

DELIN, K. A. AND JACKSON, S. P. 2000. Sensor web for *in situ* exploration of gaseous biosignatures. In *IEEE Aerospace Conference*.

DEWITT, D. J., GHANDEHARIZADEH, S., SCHNEIDER, D. A., BRICKER, A., HSIAO, H. I., AND RASMUSSEN, R. 1990. The gamma database machine project. *IEEE TKDE 2*, 1, 44–62.

DUST INC. Company Web Site. `http://www.dust-inc.com`.

FIGARO, INC. Tgs-825 - special sensor for hydrogen sulfide. Tech. rep. `http://www.figarosensor.com`.

GANERIWAL, S., KUMAR, R., ADLAKHA, S., AND SRIVASTAVA, M. 2003. Timing-sync protocol for sensor networks. In *Proceedings of ACM SenSys*.

GAROFALAKIS, M. AND GIBBONS, P. 2001. Approximate query processing: Taming the terabytes! (tutorial). In *VLDB*.

GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. 2003. The nesC Language: A Holistic Approach to Network Embedded Systems. In *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*.

GEHRKE, J., KORN, F., AND SRIVASTAVA, D. 2001. On computing correlated aggregates over continual data streams. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Santa Barbara, CA.

HANSON, E. N. 1996. The design and implementation of the ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering 8,* 1 (February), 157–172.

HELLERSTEIN, J., HONG, W., MADDEN, S., AND STANEK, K. 2003. Beyond average: Towards sophisticated sensing with queries. In *Proceedings of the First Workshop on Information Processing in Sensor Networks (IPSN)*.

HELLERSTEIN, J. M. 1998. Optimization techniques for queries with expensive methods. *TODS 23,* 2, 113–157.

HELLERSTEIN, J. M., FRANKLIN, M. J., CHANDRASEKARAN, S., DESHPANDE, A., HILDRUM, K., MADDEN, S., RAMAN, V., AND SHAH, M. 2000. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin 23,* 2, 7–18.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., AND PISTER, D. C. K. 2000. System architecture directions for networked sensors. In *ASPLOS*.

HONEYWELL, INC. Magnetic Sensor Specs HMC1002. Tech. rep. `http://www.ssec.honeywell.com/magnetic/spec_sheets/specs_1002.html`.

IBARAKI, T. AND KAMEDA, T. 1984. On the optimal nesting order for computing n-relational joins. *TODS 9,* 3, 482–502.

IMIELINSKI, T. AND BADRINATH, B. 1992. Querying in highly mobile distributed environments. In *VLDB*. Vancouver, Canada.

INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCOM*. Boston, MA.

INTERSEMA. 2002. MS5534A barometer module. Tech. rep. October. `http://www.intersema.com/pro/module/file/da5534.pdf`.

IVES, Z. G., FLORESCU, D., FRIEDMAN, M., LEVY, A., AND WELD, D. S. 1999. An adaptive query execution system for data integration. In *Proceedings of the ACM SIGMOD*.

KOSSMAN, D. 2000. The state of the art in distributed query processing. *ACM Computing Surveys*.

KRISHNAMURTHY, R., BORAL, H., AND ZANIOLO, C. 1986. Optimization of nonrecursive queries. In *VLDB*. 128–137.

LEOPOLD, M., DYDENSBORG, M., AND BONNET, P. 2003. Bluetooth and sensor networks: A reality check. In *ACM Conference on Sensor Networks (SenSys)*.

LIN, C., FEDERSPIEL, C., AND AUSLANDER, D. 2002. Multi-Sensor Single Actuator Control of HVAC Systems.

LIU, L., PU, C., AND TANG, W. 1999. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering*. Special Issue on Web Technology.

MADDEN, S. 2003. The design and evaluation of a query processing architecture for sensor networks. Ph.D. thesis, UC Berkeley.

MADDEN, S. AND FRANKLIN, M. J. 2002. Fjording the stream: An architechture for queries over streaming sensor data. In *ICDE*.

MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. 2002. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*.

MADDEN, S., HONG, W., FRANKLIN, M., AND HELLERSTEIN, J. M. 2003. TinyDB web page. `http://telegraph.cs.berkeley.edu/tinydb`.

MADDEN, S., SHAH, M. A., HELLERSTEIN, J. M., AND RAMAN, V. 2002. Continously adaptive continuous queries over data streams. In *ACM SIGMOD*. Madison, WI.

MAINWARING, A., POLASTRE, J., SZEWCZYK, R., AND CULLER, D. 2002. Wireless sensor networks for habitat monitoring. In *ACM Workshop on Sensor Networks and Applications*.

MELEXIS, INC. 2002. MLX90601 infrared thermopile module. Tech. rep. August. `http://www.melexis.com/prodfiles/mlx90601.pdf`.

MONMA, C. L. AND SIDNEY, J. 1979. Sequencing with seriesparallel precedence constraints. *Mathematics of Operations Research*.

MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., S.BABU, DATA, M., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. 2003. Query processing, approximation and resource management in a data stream management system. In *First Annual Conference on Innovative Database Research (CIDR)*.

OLSTON, C. AND J.WIDOM. 2002. In Best Effort Cache Sychronization with Source Cooperation. *SIGMOD*.

P.BONNET, J.GEHRKE, AND P.SESHADRI. 2001. Towards sensor database systems. In *Conference on Mobile Data Management*.

PIRAHESH, H., HELLERSTEIN, J. M., AND HASAN, W. 1992. Extensible/rule based query rewrite optimization in starburst. In *Proceedings of ACM SIGMOD*. 39–48.

POTTIE, G. AND KAISER, W. 2000. Wireless integrated network sensors. *Communications of the ACM 43,* 5 (May), 51 – 58.

PRIYANTHA, N. B., CHAKRABORTY, A., AND BALAKRISHNAN, H. 2000. The cricket location-support system. In *MOBICOM*.

RAMAN, V., RAMAN, B., AND HELLERSTEIN, J. M. 2002. Online dynamic reordering. *The VLDB Journal 9,* 3.

RFM CORPORATION. RFM TR1000 Datasheet. `http://www.rfm.com/products/data/tr1000.pdf`.

SENSIRION. 2002. SHT11/15 relative humidity sensor. Tech. rep. June. `http://www.sensirion.com/en/pdf/Datasheet_SHT1x_SHT7x_0206.pdf`.

SHATDAL, A. AND NAUGHTON, J. 1995. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*.

STONEBRAKER, M. AND KEMNITZ, G. 1991. The POSTGRES Next-Generation Database Management System. *Communications of the ACM 34,* 10, 78–92.

SUDARSHAN, S. AND RAMAKRISHNAN, R. 1991. Aggregation and relevance in deductive databases. In *Proceedings of VLDB*. 501–511.

TAOS, INC. 2002. TSL2550 ambient light sensor. Tech. rep. September. `http://www.taosinc.com/images/product/document/tsl2550.pdf`.

UC BERKELEY. 2001. Smart buildings admit their faults. Web Page. Lab Notes: Research from the College of Engineering, UC Berkeley. `http://coe.berkeley.edu/labnotes/1101.smartbuildings.html`.

URHAN, T., FRANKLIN, M. J., AND AMSALEG, L. 1998. Cost-based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD*.

WOLFSON, O., SISTLA, A. P., XU, B., ZHOU, J., AND CHAMBERLAIN, S. 1999. DOMINO: Databases fOr MovINg Objects tracking. In *ACM SIGMOD*. Philadelphia, PA.

WOO, A. AND CULLER, D. 2001. A transmission control scheme for media access in sensor networks. In *ACM Mobicom*.

YAO, Y. AND GEHRKE, J. 2002. The cougar approach to in-network query processing in sensor networks. In *SIGMOD Record*.