The Design of an Acquisitional Query Processor For Sensor Networks

Samuel Madden, Michael J. Franklin, and Joseph M. Hellerstein
{madden,franklin,jmh}@cs.berkeley.edu
UC Berkeley

Wei Hong
wei.hong@intel-research.net
Intel Research, Berkeley

Abstract

We discuss the design of an acquisitional query processor for data collection in sensor networks. Acquisitional issues are those that pertain to where, when, and how often data is physically acquired (*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 over traditional passive systems that assume the *a priori* existence of data. We discuss simple extensions to SQL for controlling data acquisition, and show how acquisitional issues influence query optimization, dissemination, and execution. We evaluate these issues in the context of TinyDB, a distributed query processor for smart sensor devices, and show how acquisitional techniques can provide significant reductions in power consumption on our sensor devices.

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 sensors [37, 21, 32, 8]. 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 lead to a significantly altered view of software systems than that of more traditional mobile or distributed environments.

In this paper, 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 [25, 30, 36, 43, 31]. 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 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 present *acquisitional query processing* (ACQP), where we focus 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. Of course, 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 an ACQP engine, called TinyDB, that is a distributed query processor which runs on each of the nodes in a sensor network. TinyDB runs on the Berkeley Mica *mote* platform, on top of the TinyOS [21] operating system. We chose this platform because the hardware is readily available from commercial sources [12] 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 improvement 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 ACQP-related questions, including:

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 unique to ACQP. 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 in ACQP 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 *base station*) , 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, Sections 6 discusses our adaptive, power-sensitive model for query execution and result collection.

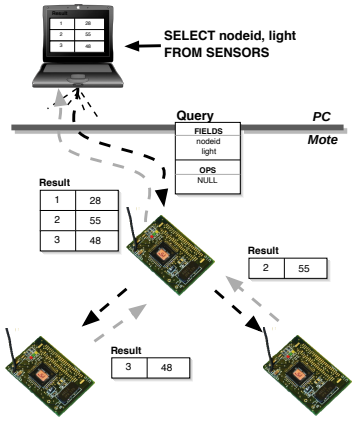


Figure 1: A query and results propagating through the network.

2 Sensor Networks and Data Collection

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

In the past several years, the sensor network research community has developed and engaged in real deployments of these devices, making it possible to understand the data collection needs specific to the sensor environment. As an example, consider recent environmental monitoring deployments on Great Duck Island and James Reserve[32, 8]. In these scenarios, motes collect light, temperature, humidity, and other environmental properties. On Great Duck Island, off the coast of Maine, sensors have been placed in the burrows of Storm Petrels, a kind of endangered sea bird. Scientists plan to use 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 [40] and sensors for building infrastructure monitoring and control [29].[1]

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 sensors, 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.

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)[2]. Power is of utmost importance. If used naively, individual sensor nodes will deplete their energy supplies in only a few days. 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.

Mica motes have a 4Mhz, 8bit Atmel microprocessor. Their RFM TR1000 radios run at 40 kbits/second over a single shared CSMA channel. Radio messages are variable size. Typically about 10 48-byte messages (the default size in TinyDB) can be delivered per second. Power consumption tends to be dominated by radio communication. When powered on, radios consume about as much power as the processor. However, because communication is so slow, every *bit* of data transmitted by the radio costs as much energy as executing 1000 CPU instructions. As an additional feature, motes have an external 32kHz clock that the TinyOS operating system can synchronize with neighboring motes +/- 1 ms to ensure that neighbors are be powered up and listening when they wish to send a message[14].

Power consumption in sensors occurs in four phases, which we illustrate in Figure 2 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.

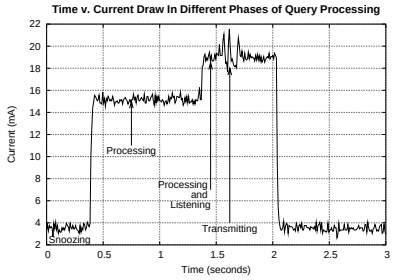


Figure 2: Phases of Power Consumption In TinyDB

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

[2] 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 sensor, because current is easy to measure directly; note that power (in Watts) = current (in Amps) * voltage (in Volts), and that Mica motes run at 3V.

2.2 Communication in Sensor Networks

Typical communication distances for low power wireless radios such as those used in Mica 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 operating 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've already transferred the decoded the 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 [42].

3 An 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.[3]

3.1 Basic Language Features

Queries in TinyDB, as in SQL, consist of a `SELECT-FROM-WHERE` clause supporting selection, join, projection, and aggregation. We also include explicit support for sampling, windowing, and sub-queries via materialization points. As is the case in the Cougar and TAG systems [36, 31], we view sensor data as a single table with one column per sensor type. Tuples are appended to this table periodically, at well-defined *sample intervals* that are a parameter of the query. The period of time between each sample interval is known as an *epoch*. As we discuss in Section 6, epochs provide a convenient mechanism for structuring computation to minimize power consumption. Consider the query:

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

This query specifies that each sensor 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 an ever-growing sequence of tuples, clustered into 1s time intervals. Each tuple includes a time stamp corresponding to the time it was produced.

Note that the `sensors` table is (conceptually) an unbounded, continuous *data stream* of values; as is the case in other streaming and online systems, 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 as fixed-size 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 INTERVAL 10s)
```

This statement provides a shared, local (i.e. single-node) location to store a streaming view of recent data similar to materialization points in other streaming systems like Aurora or STREAM [7, 34], or materialized views in conventional databases. Joins are allowed between two storage points on the same node, or between a storage point and the `sensors` relation, in which case `sensors` 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* [18] 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 INTERVAL 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), or specification of aggregation function to combine multiple rows (if the inner query is faster.) Space prevents a detailed description of this mechanism here.

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 ([31, 25, 43]) - 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.

[3] Our query language includes a number of other unusual features tailored to the sensor network domain, such as the ability to log data for later offline delivery and the ability to actuate physical hardware in response to a query, which we will not discuss here.

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 INTERVAL 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 [34, 18].

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.)

3.2 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 the operating system (in which case the code that generates the event must have been compiled into the sensor node.) 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 INTERVAL 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.

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 3.

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 3, 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.

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 query when the specified event arrives and the condition holds.

In the current implementation of TinyDB, events are only signalled on the local node – we do not 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).

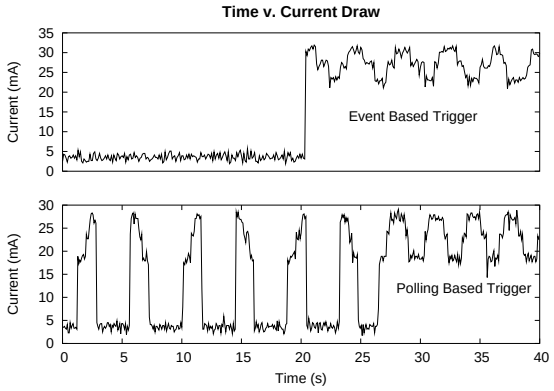


Figure 3: External interrupt driven event-based query (top) vs. Polling driven event-based query (bottom).

3.3 Lifetime-Based Queries

In lieu of a explicit `SAMPLE INTERVAL` 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. We show below a lifetime computation for simple queries of the form:

```
SELECT a1, ... , anumSensors
  FROM sensors
  WHERE p
  LIFETIME l hours
```

To simplify the equations in this example, we present a query with a single selection predicate which is applied after attributes have acquired. The ordering of multiple predicates and interleaving of sampling and selection are discussed in detail in Section 4. Table 1 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.)

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	Number of children nodes routing through this node	

Table 1: Parameters used in lifetime estimation

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 our children:

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

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 remaining 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 4 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 tranmission 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.[4] 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 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 INTERVAL` clause.) Otherwise, sampling is fixed 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 6.2.

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

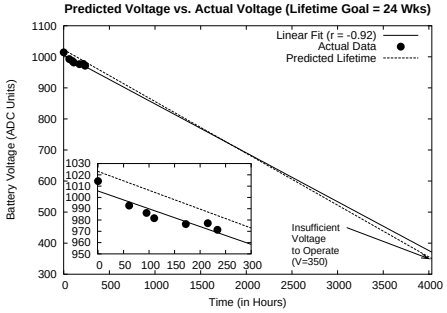


Figure 4: Predicted versus actual lifetime for a requested lifetime of 24 weeks (168 days)

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 6.3.1 discusses this runtime re-estimation in more detail.

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.

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 (which are, most frequently, very simple.) 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	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	What range of values can this attribute take on (pair of units)

Table 2: Metadata fields kept with each attribute

Metadata are 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 2 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 3 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.

Sensor	Power mW	Sample time ms	Sample Energy (VI * t), uJ
Light, Temp	.9	.1 [5]	90
Magnetometer	15 [22]	.1 [5]	1500
Accelerometer	1.8 [3]	.1 [5]	180
Organic Byproducts[5]	15	> 1000	$> 1.5 \times 10^7$

Table 3: Energy costs of accessing various common sensors

.

The catalog also contains metadata about TinyDB's extensible aggregate system. As with other extensible database systems [39] 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[31] system, 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.

[5]Scientists are particularly interested in monitoring the micro-climates created by plants and their biological processes. See [13, 8]. An example of such a sensor is Figaro Inc's H_2S sensor [15]

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 2 – 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 Ordering of Sampling And Predicates

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

As Table 3 shows, 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. [20]) 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 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 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 [23, 28, 20]; Monma and Sidney present the *Series-Parallel Algorithm Using Parallel Chains* [33], which gives an optimal ordering of the jobs in $O(|S| \log |S|)$ time.

Due to space constraints, we have glossed over the details of handling the expensive nature of sampling in the `SELECT`, `GROUP BY`, and `HAVING` clauses. The basic idea is to add them to S with appropri-

ate selectivities, costs, and ordering constraints.

As an example of this process, consider the query:

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

The order of magnitude difference in per-sample costs for the accelerometer and magnetometer suggests that the power costs of plans with different orders of sampling and selection 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. We compared the cost of these three plans, and, as expected, found that the first was always more expensive than the other two. More interestingly, the second can be an order of magnitude more expensive than third, when S_{accel} is much more selective than S_{mag}. Conversely, when S_{mag} is highly selective, it can be cheaper to sample the magnetometer first, although only by a small factor (.8). The order of magnitude difference in relative costs represents an absolute difference of 1320 uJ per sample, or 3.96 mW at a (slow) sample rate of one sample 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.

Similarly, we note that 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 INTERVAL 1s
```

In this query, the maximum of eight seconds worth of light readings will be computed, but only light readings from sensors whose magentometers read greater than x will be considered. Interestingly, it turns out that, unless the 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 pushdown* can be applied to any exemplary aggregate (e.g. `MIN`, `MAX`). Unfortunately, the selectivities of exemplary aggregates are very hard to capture, especially for window aggregates. We reserve the problem of ordering exemplary aggregates in query optimization for future work.

4.3 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 INTERVAL 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 at every d seconds for a duration of k seconds, at which point it stops running.

Note that, by the semantics formulated above, 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 INTERVAL 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 5 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 *asynch 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. We expect that the asynchronous approach will 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.

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 that 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.

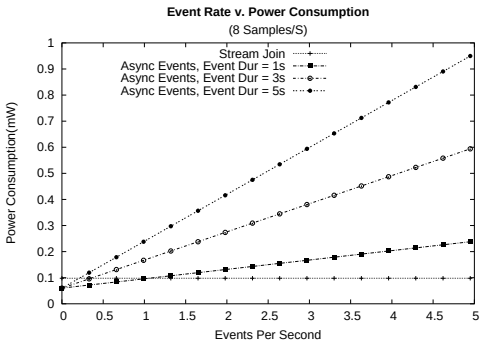


Figure 5: The cost of processing event-based queries as asynchronous events versus joins.

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 sensor hears the query, it must if decide 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 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 common 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. Similarly, if a node knows that none of its children will ever satisfy the value of some selection predicate, say because they have constant attribute values outside the predicate's range, it need not forward the query down the routing tree. To maintain information about child attribute values , we propose the use of 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 2.2 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 in situations in which there are several parents of comparable link quality. A link-quality-based parent selection algorithm, such as the one described in [42], should be used in conjunction with the SRT to prefilter the set of 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. [6] 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 (re-transmitted 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 value of A along with the child's id. When it has heard from all of its children, it chooses a

[6] A natural extension to SRTs would be to store multiple intervals at each node.

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.

Figure 6 shows an SRT over the latitude. 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.)

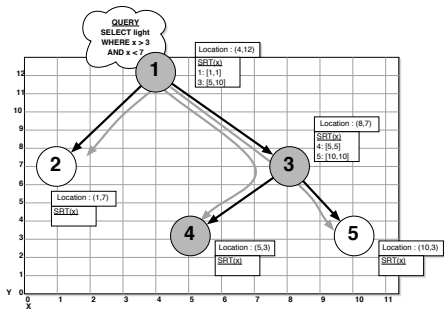


Figure 6: A semantic routing tree in use for a query. Gray arrows indicate flow of the query down the tree, gray nodes must produce or forward results in the query.

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.

Node appearance and link quality change can both require a node to switch parents. To do this, it 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 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.

Finally, we note that, by using these maintenance rules proposed, it is possible to support SRTs over non-constant attributes, although if those attributes change quickly, the cost of propagating changes in child intervals 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 geographic attributes, for example, since network topology is correlated with geography.

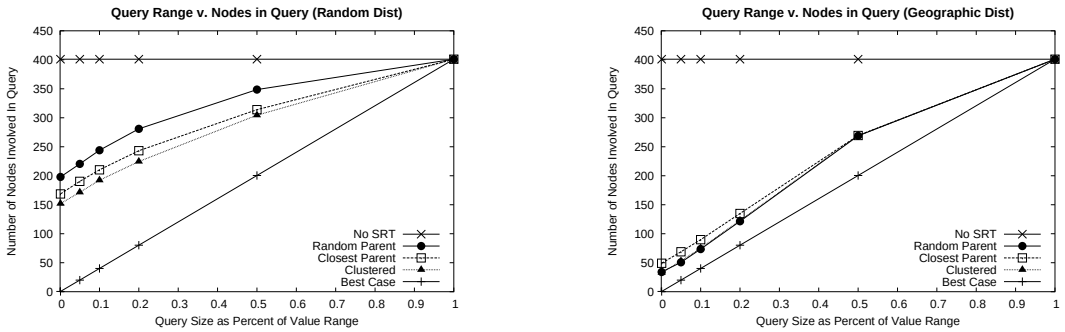
We study three policies for SRT parent selection. In the first, *random* approach, each node picks a random parent from the nodes with which it can communication reliably. In the second, *closest-parent* approach, each parent reports the value of its index attribute with the SRT-build request, and children 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 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 a number of sensor value distributions; we report on two here. 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 sensor's x and y position in the grid, such that a sensor's value tended to be highly correlated to the values of its neighbors.

Figure 7 shows the number of nodes which 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 distributions (for a total of 30 experiments). In each experiment, an SRT was constructed according to the appropriate policy and sensor value distribution. Then, for each interval size, the average number of nodes participating in 100 randomly constructed queries of the appropriate size was measured.

For both 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 distribution, 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. This simulation is admittedly optimistic, since geography and topology are perfectly correlated



(a)Random Distribution (b)Geographic Distribution

Figure 7: Number of nodes participating in range queries of different sizes for different parent selection policies in a semantic routing tree (20x20 grid, 400 sensors, each point average of 500 queries of the appropriate size.)

in our experiment. Real sensor network deployments show significant but not perfect correlation [16].

It is a bit surprising that, even for a random distribution of sensor values, the closest-parent and clustered approaches are substantially better than the random-parent approach. The reason for this is that these techniques reduce the spread of sensor values beneath parents, thereby reducing the probability that a randomly selected range query will require a particular parent to participate.

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.

5.4 SRT Summary

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.

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 contented 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, they all sleep and wake up at the same time, ensuring that results will not be lost as a result of a parent sleeping when a child tries to propagate a message. 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.

TinyDB uses a simple algorithm to scale t_{awake} based on the neighborhood size, the details of which we omit. 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 the next subsection.

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 6.3 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.

For aggregation queries across nodes, we adopt the approach of TAG [31], although TAG does not support temporal aggregates but only aggregates of values from different nodes produced in the same epoch.

The basic approach used in both TAG and TinyDB is to compute a *partial state record* at each intermediate node in the routing topology. This record represents the partially evaluated aggregation of local sensor values with sensor values received from child nodes as they flow up the routing tree. The benefit of doing this is that a great deal less data is transmitted than when all sensors' values are sent to the root of the network to be aggregated together.

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.

6.2 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 try to retransmit this tuple, re-enqueue this tuple and try to send a different tuple, combine this tuple with some other tuple for the same query, or simply discard the tuple.

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 [38]. 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. [17, 9]), 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 it non-obvious how to apply them.

We begin with a comparison of three simple prioritization schemes, *naive*, *winavg*, and *delta* for simple selection queries. 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.

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 8 shows how delta and winavg approximate a high-periodicity trace of sensor readings generated by a shaking accelerometer (we omit naive due to space constraints.) 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.

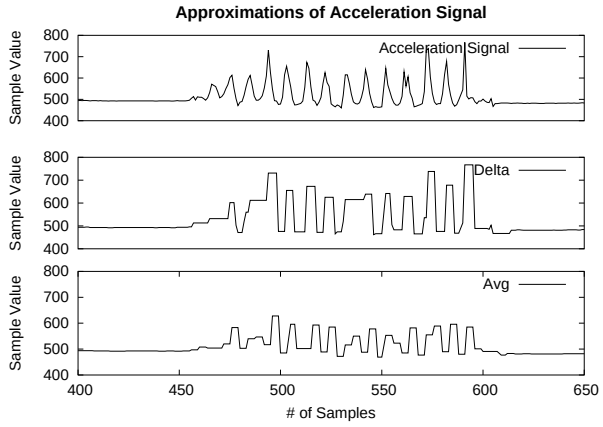


Figure 8: An acceleration signal (top) approximated by a delta (middle) and an average (bottom), K=4.

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 4. 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 out-

performs either other approach for the fast changing step-functions in the light signal because it does not smooth edges as much as average.

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

Table 4: RMS Error for Different Prioritization Schemes and Signals (1000 Samples, Sample Interval = 64ms)

We omit a discussion of prioritization policies for aggregation queries. TAG [31] discusses several snooping-based techniques unique to sensor networks that can be used to priortize aggregation queries. There is also significant related work on using wavelets and histograms to approximate distributions of aggregate queries when there are many groups, for example [17, 9]. These techniques are applicable in sensor networks as well, although we expect that the number of groups will be small (e.g. at most tens or hundreds), so they may be less valuable.

Thus, we have illustrated some examples where 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.

6.3 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 systems like Tukwila and Eddy [26, 6] 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 bring the system to its knees, 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 9 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.

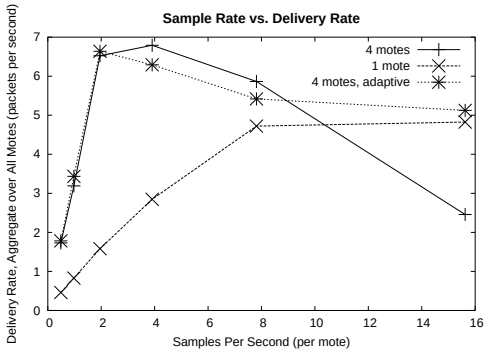


Figure 9: Per-mote sample rate versus aggregate delivery rate.

The problem with reducing 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 6.2 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 10. 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.

6.3.1 Measuring Power Consumption

We now turn to the problem of adapting tuple delivery rate to meet specific lifetime requirements in response to incorrect sample rates computed at query optimization time (see Section 3.3). We first note that, using similar computations to those shown Section 3.3, it is possible to compute a *predicted battery voltage* for a time t seconds into processing a query. We omit the calculation due to space constraints.

The system can then compare its current voltage to this predicted volt-

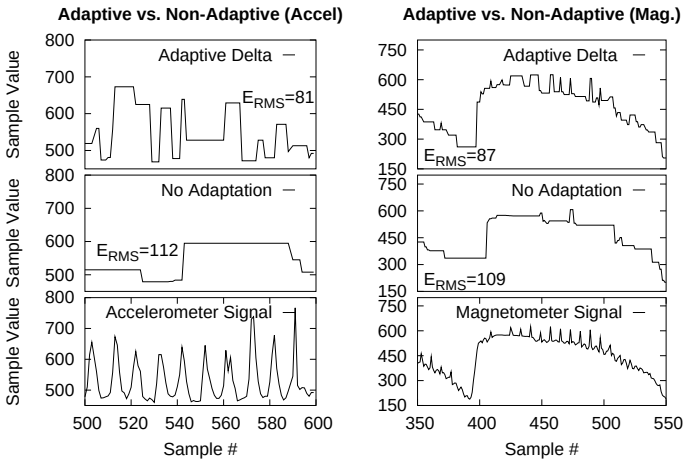


Figure 10: Comparison of delivered values (bottom) versus actual readings for from two motes (left and right) sampling at 16 packets per second and sending simulataneously. Four motes were communicating simultaneously when this data was collected.

age. By assuming that voltage decays linearly (see Figure 4 for empirical evidence of this property), 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.

Although this calculation and re-optimization are straightforward, they serve an important role by allowing sensors in TinyDB to satisfy occasional ad-hoc queries and relay results for other sensors without compromising the 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 like Eddies [6]) and the effect of frequency of re-estimation on lifetime (currently, in TinyDB, re-estimation can only be triggered by an explict request from the user.)

7 Summary

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 aspects of an acqusitional 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 priortizing data according to quality and discussed the need for techniques to adapt the transmission and sampling rates of an ACQP system.

8 Related Work

There has been some recent publication in the database and systems communities on query processing-like operations in sensor networks [25, 31, 36, 30, 43]. As mentioned above, 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, acqusitional 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 [24, 2]. 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 [41]) 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 [1], although, as with the previous work, the focus is on optimizing costs in tradtional 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 [27] 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 [4]. 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 [41, 24]), 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 [11], 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.

There is a large body of work on event-based query processing in the active database literature. Languages for event composition and systems for evaluating composite events, such as [10], as well as systems for efficiently determining when an event has fired, such as [19] could (possibly) be useful in TinyDB.

Approximate and best effort caches [35], as well as systems for online-aggregation [38] and approximate [17] and stream query processing [34, 7] 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 [7] mentions a need for this kind of metric, but proposes no specific approaches.

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, emdedded 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 acqusitional 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 priortization 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 priortization 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 cognisant of semantics and the costs and capabilities of the underlying harware the success of such deployments will be limited.

References

[1] R. Alonso and S. Ganguly. Query optimization in mobile environments. In *Workshop on Foundations of Models and Languages for Data and Objects*, pages 1–17, September 1993.

[2] R. Alonso and H. F. Korth. Database system issues in nomadic computing. In *ACM SIGMOD*, Washington DC, June 1993.

[3] Analog Devices, Inc. *ADXL202E: Low-Cost 2 g Dual-Axis Accelerometer*. http://products.analog.com/products/info.asp?product=ADXL202.

[4] H. G. Arturo Crespo. Routing indices for peer-to-peer systems. In *ICDCS*, July 2002.

[5] Atmel Corporation. Atmel ATMega 128 Microcontroller Datasheet. http://www.atmel.com/atmel/acrobat/doc2467.pdf.

[6] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD*, pages 261–272, Dallas, TX, May 2000.

[7] D. Carney, U. Centiemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, 2002.

[8] A. Cerpa, J. Elson, D.Estrin, L. Girod, M. Hamilton, , and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, 2001.

[9] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *VLDB Journal*, 10, 2001.

[10] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, 1994.

[11] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *ACM SIGMOD*, 2001.

[12] I. Crossbow. Wireless sensor networks (mica motes). http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.

[13] K. A. Delin and S. P. Jackson. Sensor web for *in situ* exploration of gaseous biosignatures. In *IEEE Aerospace Conference*, 2000.

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

[15] Figaro, Inc. *TGS-825 - Special Sensor For Hydrogen Sulfide*. http://www.figarosensor.com.

[16] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wickera. Complex behavior at scale: An experimental study of low-power wireless sensor networks. Under submission. Available at: http://lecs.cs.ucla.edu/ deepak/PAPERS/empirical.pdf, July 2002.

[17] M. Garofalakis and P. Gibbons. Approximate query processing: Taming the terabytes! (tutorial). In *VLDB*, 2001.

[18] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Santa Barbara, CA, May 2001.

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

[20] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *TODS*, 23(2):113–157, 1998.

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

[22] Honeywell, Inc. *Magnetic Sensor Specs HMC1002*. http://www.ssec.honeywell.com/magnetic/spec_sheets/specs_1002.html.

[23] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *TODS*, 9(3):482–502, 1984.

[24] T. Imielinski and B. Badrinath. Querying in highly mobile distributed environments. In *VLDB*, Vancouver, Canada, 1992.

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

[26] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proceedings of the ACM SIGMOD*, 1999.

[27] D. Kossman. The state of the art in distributed query processing. *ACM Computing Surveys*, 2000.

[28] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pages 128–137, 1986.

[29] C. Lin, C. Federspiel, and D. Auslander. Multi-Sensor Single Actuator Control of HVAC Systems. 2002.

[30] S. Madden and M. J. Franklin. Fjording the stream: An architechture for queries over streaming sensor data. In *ICDE*, 2002.

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

[32] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler. Wireless sensor networks for habitat monitoring. In *ACM Workshop on Sensor Networks and Applications*, 2002.

[33] C. L. Monma and J. Sidney. Sequencing with seriesparallel precedence constraints. *Mathematics of Operations Research*, 1979.

[34] R. Motwani, J. Window, A. Arasu, B. Babcock, S.Babu, M. Data, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation and resource management in a data stream management system. In *CIDR (to appear)*, 2003.

[35] C. Olston and J.Widom. Best effort cache sychronization with source cooperation. *SIGMOD*, 2002.

[36] P.Bonnet, J.Gehrke, and P.Seshadri. Towards sensor database systems. In *Conference on Mobile Data Management*, January 2001.

[37] G. Pottie and W. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51 – 58, May 2000.

[38] V. Raman, B. Raman, and J. Hellerstein. Online dynamic reordering. *The VLDB Journal*, 9(3), 2002.

[39] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, 1991.

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

[41] O. Wolfson, A. P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. DOMINO: Databases fOr MovINg Objects tracking. In *ACM SIGMOD*, Philadelphia, PA, June 1999.

[42] A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. In *ACM Mobicom*, July 2001.

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