The Design and Evaluation of a Query Processing Architecture for Sensor Networks

by

Samuel Ross Madden

B.S. (Massachusetts Institute of Technology) 1999
M.Eng. (Massachusetts Institute of Technology) 1999

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Michael J. Franklin, Chair
Professor Joseph M. Hellerstein
Professor John Chuang

Fall 2003

The dissertation of Samuel Ross Madden is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2003

The Design and Evaluation of a Query Processing Architecture for Sensor Networks

by

Samuel Ross Madden

ABSTRACT

The Design and Evaluation of a Query Processing Architecture for Sensor Networks

by

Samuel Ross Madden

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael J. Franklin, Chair

With the advent of small, battery-powered, wireless computing and sensing technology, it is now possible to monitor and observe the world at unprecedented levels of granularity. Networks of such devices typically consist of tens or hundreds of small, power constrained nodes deployed in remote locations which they are expected to monitor for months or years at a time. Such networks present significant new opportunities for the data management community.

In this dissertation, we summarize the issues and opportunities associated with collecting and processing information from these wireless sensor networks, focusing on the performance and ease-of-use advantages of a declarative, query-based approach. We present the architecture of a query processing system and design of a query language specifically tailored towards power-efficient acquisition and in-network processing of sensor data, with a focus on issues related to minimizing unnecessary communication and sensing. We present many of these ideas in the context of TinyDB, a prototype database system which runs on networks of Berkeley Motes, and evaluate them using both performance micro-benchmarks and measurements from real-world deployments.

ACKNOWLEDGMENTS

Mike Franklin and Joe Hellerstein, my advisors, contributed endless hours of advice and guidance regarding research, writing, speaking, and life in general. They were the best advisors I could have asked for, and I can only hope to one day be as generous, thoughtful, and engaged with my students.

Mike taught me to be a researcher – the subtleties of writing, the peculiarities of publishing papers in computer science, the importance of public speaking and getting to know other researchers in the community. He singlehandedly coerced me into being a database researcher, steering me towards the field during my early days at Berkeley. He suggested the connection between sensor networks and databases, and his insights and ideas are infused throughout this work. His tireless editorial effort vastly improved the quality of this dissertation, and of much of the rest of my research: wherever there is a graph with a font too small, or a paragraph with a sentence too meandering, Mike will be waiting to edit it.

Joe is a professorial machine – always available, always insightful, always looking out for his students. As with Mike, the many hours we spent working together were a joy, and this dissertation would have been a shadow of itself were it not for his ideas and input.

Wei Hong has been a co-conspirator in much of my research; his ideas and effort have shaped my work and helped make TinyDB into a successful, usable system. Our daily discussions and weeks spent together coding, debugging, and writing have been a pleasure.

The Berkeley database group, especially Sirish Chandrasekaran, Amol Deshpande, Vijayshankar Raman, and Mehul Shah, were a constant source of ideas, expertise, and amusement throughout my doctoral studies. Our many conversations made this a better dissertation.

The TinyOS gang – Jason Hill, Phil Levis, Joe Polastre, Rob Szewczyk, and Alec Woo – de-

serves credit for making TinyOS and the motes a real, usable platform, without which my work would have been much harder or would not have happened at all. Rob contributed significantly to early versions of the ideas presented in Chapter 4, and both he and Joe helped with the Botanical Garden deployment.

Thanks goes to David Culler for his TinyOS vision and drive to follow through on it. His successful effort to recruit me into Intel and persistence in pushing me to identify the real challenges regarding query processing in sensor networks were a tremendous help. Thanks also to John Chuang for his helpful comments and feedback on this dissertation and for taking time out of his schedule to serve on my committee.

Annie Abramson has been wonderfully caring and understanding throughout my graduate career, and especially during the past few months. I am unspeakably happy to have met her.

I have been blessed throughout my life with many great teachers and mentors. I am particularly indebted to Mike Casey, Robert Lundsford, Joel West, Tom Wiegand, Jonathan Pfautz, and Nat Durlach.

Finally, I am especially grateful to my family and friends for keeping me happy, sane, and normal, and for providing me with a life outside of computer science. My mom and dad, in particular, have always been wonderful and made me into the good kid I am today.

Portions of this work were originally published in shortened form in the proceedings of the ACM SIGMOD Conference [MFHH03], the USENIX OSDI Conference [MFHH02], the IEEE Workshop on Mobile Computing and Systems Applications (WMCSA) [MSFC02], and the Workshop on Information Processing in Sensor Networks (IPSN) [HHMS03]. My research was supported in part by the National Science Foundation under ITR/IIS grant 0086057, ITR/IIS grant 0208588, ITR/IIS grant 0205647, ITR/SI grant 0122599, and by ITR/IM grant 1187-26172, as well as research funds from IBM, Intel, Microsoft, and the UC MICRO program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advances in computing technology have led to the production of a new class of computing device: the wireless, battery powered, smart sensor. Traditional sensors deployed throughout buildings, labs, and equipment are passive devices that simply modulate a voltage based on some environmental parameter. In contrast, these new sensors are active, full fledged computers, that not only sample real world phenomena but can also filter, share, combine, and operate on the data they acquire.

1.1 Sensor Networks

At UC Berkeley, researchers have developed small sensor devices, called *motes*, and an operating system that is especially suited to running on them, called TinyOS [HSW+00]. Two motes, the Mica and Mica2Dot, which are similar in functionality but differ in form-factor, are shown in Figure 1.1. Motes are equipped with a radio, a processor, and a suite of sensors. TinyOS makes it possible to deploy *ad-hoc* networks of these devices. Such networks differ from traditional computer networks in that motes in an ad-hoc network can locate each other and route data without any prior knowledge about the network topology and with no dedicated configuration servers or routers (e.g. DHCP servers or BGP routers.) As of this writing, several significant deployments these networks have already been undertaken, including two new deployments during the summer of 2003 in the

Berkeley Botanical Garden and Great Duck Island near Acadia national park.

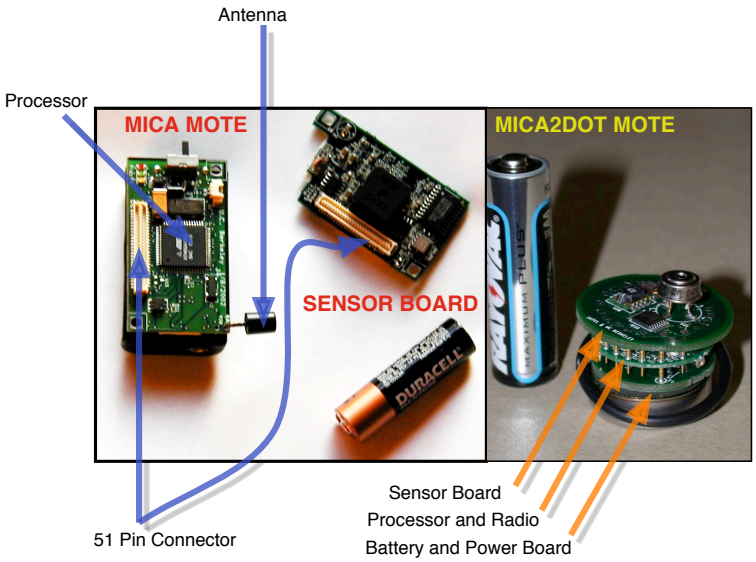


Figure 1.1: Annotated Motes

Mote technology has enabled a broad range of new applications: the low cost, small size, and untethered nature of these devices makes it possible to sense information at previously unobtainable resolutions. Animal biologists can monitor the movements of hundreds of different animals simultaneously, receiving updates of location as well as ambient environmental conditions every few seconds [MPSC02, CED+01, JOW+02]. Vineyard owners can place sensors on each of their plants, providing an exact picture of how various light and moisture levels vary in the microclimates around each vine [BB03]. Supervisors of manufacturing plants, temperature controlled storage warehouses, and computer server rooms can monitor each piece of equipment, and automatically dispatch repair teams or shutdown problematic equipment in localized areas where temperature spikes or other faults occur.

Using traditional techniques, deployments such as those described above require months of design and engineering time, even for a skilled computer scientist. Some of this cost is hardware-related and domain-specific: for example, the appropriate choice of sensing hardware and device

enclosure will vary dramatically if a network is designed for a forest canopy versus a sea floor. Aside from these domain-specific considerations, however, there is a substantial amount of software involved in each of these deployments, much of which is similar across different scenarios. The bulk of that software, however, is written from scratch in each deployment.

In some ways, this lack of reuse stems simply from bad software engineering – improperly designed interfaces or hastily assembled code that lacks proper abstractions. In other cases, however, the developers of an application make low-level, application specific decisions that make their code difficult to adapt to new settings. For example, in a recent version of the software for monitoring Great-Duck Island [MPSC02], the authors hard-coded the data record format such that the application could only collect a small, fixed set of sensor attributes in a particular order. Subtle timing dependencies between the various pieces of software that manage the low-level hardware also severely restrict the portability of the code.

1.2 The Declarative Approach to Data Collection in Sensor Networks

This *application specific* view of sensor network programming is a problem because it requires months of software engineering for each deployment, and does not allow successive deployments to take advantage of insights and innovations developed in prior systems. To address these concerns, we believe a different, *data-centric* perspective is needed, where users specify the data they are interested in collecting through simple, *declarative* queries. These queries are high-level statements of logical interests, such as "tell me the average temperature on the 4th floor of this building" or "tell me the location of the sensor with the least remaining battery capacity." Just as in a database system, this kind of data-centric interface allows a sensor network data collection system to efficiently collect and process requested data *within* the sensor network, freeing the user from concerns over the details of data collection and processing. Specific facilities provided by the query-processing

architecture we have developed include:

- Dissemination of queries into the sensor network;
- Identification of sensors corresponding to logical names used in queries (e.g. “sensors on the fourth floor”);
- Collection of results from the network, over multiple radio hops and in a power efficient manner;
- Acquisition of sensor readings from a variety of low-level hardware interfaces;
- Storage and retrieval of historical data records in the network; and,
- Adaptation of communication topology and data rates to minimize wasted communication and route around missing or off-line nodes.

We have built a prototype instantiation of this architecture, called TinyDB, which runs on collections of Berkeley motes. TinyDB has been successfully deployed in both our lab at Berkeley as well as the Berkeley Botanical Gardens. Using TinyDB, the “programming” of the queries for each of these deployments took only a few minutes, versus weeks or months for each custom application.

TinyDB is also being used by a number of commercial organizations around the world, including researchers at Accenture interested in equipment monitoring, Tayzone Systems Corp, a small company that is using TinyDB for agricultural monitoring, and Digital Sun, a startup company building smart sprinklers that adjust their output based on soil conditions.

Aside from greatly simplifying the amount of work that users of sensor networks must do to prepare for a deployment, this query-processing based approach to sensor management has the potential to offer dramatic improvements in energy efficiency – the typical measure of performance in sensor networks – for data collection applications. This echoes another well-known lesson from

the relational database community: because declarative queries include no specification of *how* the required data is to be collected and processed, the system is free to explore many possible physical instantiations (plans) that have the same logical behavior as the user's query, and to choose the one which is expected to offer the best performance. This process, called *query optimization*, is central to the performance of the architecture.

Not surprisingly, the particular language and set of optimizations used for query processing in sensor networks differs substantially from those found in traditional database systems. We have designed a language that is similar to SQL, but that has several extensions for sensor networks selected for their amenability to power-based optimization. We have also developed a suite of optimizations that enable *aggregate queries* [1] to be run efficiently within a sensor network. More broadly, we propose the concept of *acquisitional query processing* (ACQP) for environments like sensor networks where the data to be processed does not exist at the time a query is issued but rather, is actively generated by acquiring samples from special sensing hardware. ACQP comprises a suite of techniques for deciding when and where to sample and which samples to deliver from the network. The combination of in-network aggregate processing and the application of these acquisitional techniques results in a system that differs substantially from traditional parallel or distributed databases.

We note that this work is not the first to recognize the need for a data-centric interface to sensor networks. Other efforts, such as Directed Diffusion from USC, ISI, and UCLA [IGE00, IEGH01, HSI+01] and the Cougar project at Cornell [YG02, PJP01] have proposed techniques aimed at addressing this need. Diffusion, however is a considerably lower-level, non-declarative interface. As a result, it can be more expressive than our declarative approach (just as assembly language *can* be more expressive than Java), but pays a significant price in terms of usability and lacks potential for

[1] Aggregate queries compute statistics about logical groups of sensors (as opposed to collecting data from every sensor independently).

automated optimizations by the system. The Cougar project advocates a similar approach to ours but has yet to fully develop a complete implementation that demonstrates the viability of their approach. By developing the entire TinyDB system, we discovered a number of interesting algorithmic issues that were not initially apparent to us or these other research groups.

A careful comparison to these projects and other related work appears in the next chapter. First, however, we summarize the major research contributions of this dissertation.

1.3 Contributions

The major contributions of this dissertation are fourfold:

- First, **we show that the declarative approach we propose is extremely well suited to the problem of efficiently extracting data from sensor networks**. Our architecture offers an easy-to-use, reconfigurable, and power-efficient system that is readily adaptable to a number of monitoring and tracking applications. We believe that this work is the first to both fully articulate the advantages of a declarative approach and to demonstrate definitively, through a working software artifact, that a declarative approach is feasible and practical.

- Second, **we show how traditional query processing techniques such as query optimization and execution can be mapped onto a distributed sensor network**. The basic approach is presented in Chapter 3; Chapters 4 and 5 then focus on the details of query processing, discussing ways in which conventional optimization and query processing techniques fall short and proposing a novel, energy and communication-efficient data collection framework for sensor networks.

  Of particular importance is the *aggregate operator taxonomy* (described in Section 4.2.3), which classifies certain common query processing operators according to generic properties that can be exploited to transparently optimize query execution.

- Third, **we show how the declarative approach can be used to control the acquisition of data in a sensor network**. This approach is in contrast to the traditional role of a database system as a processor of stored data that was generated by some other process. We show how this novel *acquisitional* approach is able to manage subtle issues such as the timing of sampling and communication, identification and location of nodes with appropriate data, power management, and calibration of readings that arise when retrieving or *sampling* data from the sensors in a sensor network. We discuss the details of *acquisitional query processing* in Chapter 5.

- Fourth, **we describe the TinyDB system for collecting data from sensor networks via declarative queries**. TinyDB is an implementation of the declarative techniques mentioned above. We provide a high level overview of its architecture, motivate the particular design we have selected, describe query language features, and show that the system works well in several real-world scenarios, including a recent deployment in the Berkeley Botanical Garden.

1.4 Additional Contributions

In addition to the major contributions described above, as we describe our architecture, we address a number of subsidiary challenges; in particular:

- We show how to structure communication for query processing to promote power conservation. This is largely accomplished by *scheduling* the delivery of results, such that receivers know when to listen for data from senders, leaving their radios and processors off in the interim.

- We describe a number of novel query processing operators and applications of our architecture that allow it to perform sensor network specific tasks such as building contour maps of a

region, tracking moving vehicles in a space, and approximating data from sensor networks in a variety of ways.

- We discuss the design of PC-side software for interfacing with TinyDB-like systems, and present a high-level description of some related work we have done in this area.

- Finally, we propose a number of challenging query processing problems that remain to be solved in this area, in the hopes that this dissertation will serve as a starting point for many more efforts at the intersection of database systems and sensor networks.

1.5 Assumptions

To limit the scope of this dissertation, we make several assumptions about the type and configuration of the networks being queried.

First, it is assumed that the networks are relatively homogeneous – that is, each of the sensor nodes has roughly the same processing, energy, storage, and radio resources. Heterogeneity implies a specialization of different nodes for different tasks and a new set of optimization challenges and opportunities. While such problems are clearly important, they are broad enough that they merit research efforts in their own right; indeed, the research community has recently begun to adapt TinyDB to include support for heterogeneous environments [Ore03].

Second, the implementation and real world experiments are limited to the Mica and TinyOS platform. As summarized in the next chapter, there are a variety of alternative sensor-network platforms; we believe, however, that the Mote platform is representative of these other environments. The next chapter also outlines how sensor networks are expected to evolve in the next decade, and argues that the research in this dissertation will continue to be significant in future sensor networks.

We revisit and relax these assumptions when we discuss future plans in Chapter 8.

1.6 Summary

Wireless sensor networks are collections of radio-equipped, battery powered devices that can be used to monitor and observe remote environments at an unprecedented level of granularity due to their low cost and ease of deployment. In this chapter, we discussed how the current state of the art in software for these networks makes deployment unnecessarily difficult, and summarized our argument that a query processing interface is both more user-friendly and potentially more power-efficient due to its declarative nature and the potential for automatic optimization by the query processor.

The remainder of this dissertation justifies these arguments more completely. In the next chapter, we discuss the details of sensor networks and summarize relevant and related research projects. In Chapter 3 we introduce the query processing architecture and describe the implementation of TinyDB. Chapters 4 and 5 describe the details of query processing in these environments, focusing in particular on aggregation and data acquisition, respectively. Chapter 6 discusses a recent deployment of TinyDB in the Berkeley Botanical Garden. Then, chapter 7 discusses several other advanced applications of the techniques presented in previous chapters. In Chapter 8, we discuss remaining open issues and sketch directions for future work. Chapter 9 presents the conclusions of this dissertation.

Chapter 2

Background

In this chapter, we provide background on sensor networks, motes, TinyOS, and the basics of declarative query processing. We also discuss other research projects that are addressing some of the same data-collection issues that arise in TinyDB. We briefly discuss other hardware and software platforms for sensor networks, but predominately focus on the Berkeley mote platform, as it is the environment for which TinyDB was built.

2.1 Sensor Networks

We begin with a discussion of some of the applications of sensor networks, focusing on several real-world deployments, which we use to motivate many of the design decisions in this dissertation.

To date, many research groups [MPSC02, CED+01, JOW+02, BB03] have proposed deploying or have deployed sensor networks into real, non-laboratory environments. These deployments can be broadly classified as either *monitoring* or *tracking* applications. In monitoring scenarios [MPSC02, CED+01], which are the primary focus of TinyDB[1], sensors observe local phenomena and report those observations to some external user. Sensors may exchange information with each other to increase the fidelity of their observations (*sensor fusion*) or may suppress their own readings when a nearby node's information supersedes their own.

In tracking scenarios [NES, CHZ02], sensors arrayed over some space coordinate to track one

[1] We discuss uses of TinyDB for tracking in Chapter 7

or more moving objects in their midst. Though the current location and trajectory of the moving object may be reported externally to some monitoring process, the primary goal of the network is to track the object as accurately as possible.

We describe a real-world deployment relating to each of these scenarios, to familiarize the reader with the issues, constraints, and challenges involved in deploying sensor networks.

2.1.1 Habitat Monitoring on Great Duck Island

Leach's Storm Petrel is a common, nocturnal sea-bird that spends most of its life at sea, except during short periods when it returns to land to nest and breed in colonies like the one on Great Duck Island (GDI), off the coast of Maine (see Figure 2.1) [MPSC02, Soc]. Petrel nests consist of shallow burrows one to three feet in length, large enough that motes can comfortably be placed inside. Birds return to nesting sites used in prior years, reusing previously excavated burrows. During the Summer of 2002, researchers (three computer scientists from UC Berkeley and Intel-Berkeley and a small team of biologists from the College of the Atlantic in Maine) placed motes in a number of burrows just before the birds returned to the island for nesting season, and recorded sensor data as birds came and went. Captured information included light readings, ambient temperature, passive infrared (a measure of the surface temperature, or radiated heat, from a surface above the sensor), as well as humidity and air pressure.

The GDI deployment consisted of a multi-tiered architecture – sensors in nests were transmit-only, and entered a very low power sleep state between transmissions to a basestation. The basestation was connected to a high-gain antenna and a car-battery and relayed all messages several thousand feet across the island to a lighthouse where a laptop recorded readings on disk and periodically uploaded them onto the Internet via satellite.

Not surprisingly, this deployment taught us a great deal about the difficulties involved in actual deployments of experimental hardware and software. Among these were a number of interesting

software lessons (see [MPSC02] for details) relating to the difficulties that even skilled programmers experience when trying to deploy custom software under a tight time budget; of particular relevance to this dissertation were:

- **Power Management**: In order for GDI-like networks to last for months in the field, they need aggressive power management. The solution in the initial GDI deployment , where in-burrow motes are transmit-only, is problematic in that all sensors must be a single radio-hop from the basestation and because there is no way to reconfigure or re-task the network after it has been deployed. In TinyDB, we support scheduled, low-power, bi-directional communication and easy in-the-field reconfiguration, as described in Chapter 3.

- **Multihop Communication**: With a range of only a few hundred feet at most, it will often be the case that wireless sensors will need to use neighboring nodes to relay data to the basestation. In [MPSC02], researchers explicitly cite a clear need for multihop communication, though it was not implemented in their initial deployment.

- **Sensor Calibration and Data Collection**: The initial GDI deployment simply collected a fixed set of raw (uncalibrated) sensor readings from each devices. [MPSC02] cites a lack of calibration as a limiting factor in the consistency and quality of the data gathered during the deployment.

- **Re-tasking**: In [MPSC02], the authors spend a significant amount of time discussing the need for re-tasking – that is, reconfiguring the software running on the sensors to collect a different set of data items, process the data slightly differently, or to report data at different rates. In initial deployments on GDI, this was simply not possible. Conversely, one of the primary advantages of the declarative approach is that it allows users to easily issue new queries or to change details of running queries.

- **In-Network Processing**: The deployers of GDI note a need for results to be combined and partially processed, or *aggregated* within the network in some situations. The need for this stems from limited bandwidth and energy, which makes it difficult or impossible to bring complete (unaggregated) logs of data out of the network.

  As a simple example of in-network processing, imagine that scientists on GDI wanted to compute the average temperature on the island. One way to do this would be to compute the average externally, by bringing all of the temperature readings out of the network. A more communication-efficient alternative, however, would be to ask neighboring nodes to combine their values into a sum of temperatures and a count of nodes and just transmit those values to the outside of the network, reducing the amount of communication out of the network from one message per node to just this <*sum,count*> pair. As long as this combination of readings is done in a principled way so that no readings are lost or counted multiple times, this in-network approach will produce the same answer as the less efficient external collection process.

- **Health and Status Monitoring**: Finally, in [MPSC02], the authors discuss the need to monitor the remaining battery life, network connectivity, and general health of sensor nodes in addition to sensor data collection. In our declarative approach, these monitoring attributes are queryable, and status queries can run side-by-side with data-collection queries. This leads to a single, uniform interface sensor-querying and monitoring; tools developed for sensor applications can be reused for system monitoring.

2.1.2 Tracking Vehicle Signatures

In this section, we briefly describe a vehicle tracking scenario similar to those described in work by PARC [CHZ02] and by the NEST group at UC Berkeley [SSS+02]. The primary goal of most

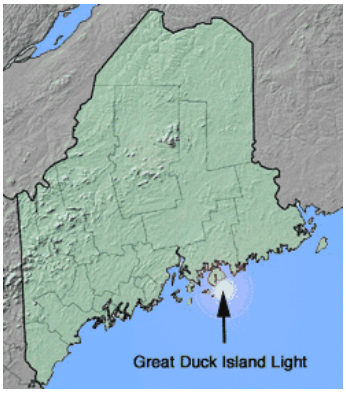


Figure 2.1: Great Duck Island, Near Acadia National Park off the Coast of Maine.

tracking applications is to keep the network informed about the location of some mobile vehicle in its midst. Of course, it may also be desirable to allow an outside observer to see where the network thinks the vehicle is at any point in time. In the case of the NEST project, the network tracks vehicles for the purpose of deploying a *pursuer* that relies on the network to locate an *evader* vehicle. The instances of such networks to date have focused largely on providing functional implementations of basic algorithms that run for a few hours (long enough for a demonstration), but both the PARC and Berkeley projects emphasize the need for network longevity, requiring energy conservation whenever possible.

Typically, detection in these environments is done with magnetometers, which detect the magnetic field induced by the movement of the motor and wheels in a vehicle, or sound and vibration sensors which measure a characteristic sound signal from the vehicle. Readings from the set of sensors nearest the vehicle are combined and averaged to minimize noise and localize the vehicle as accurately as possible. This location estimate is then collected at some *leader node* which is close to the target; the leader is selected via any one of a number of leader election protocols from the distributed algorithms literature [Lyn96]. Additionally, if needed, the leader may deliver the current position of the vehicle to the outside of the network or to some other node via tree-based [WC01]

or geographic [KK00] routing.

Vehicle tracking applications differ from monitoring deployments in several ways; in particular:

- **Locality of Activity**: Because the size of vehicles is typically small relative to the size of the monitoring network, an ideal deployment uses only a small subset of the nodes localized around each vehicle to determine positions. This allows the remainder of the network to sleep.

- **Tracking Handoff**: Rather than routing all data to some central, statically assigned location, most vehicle tracking solutions propose some kind of *handoff*, where the region of nodes responsible for vehicle tracking moves as the vehicle moves. This handoff is done from one round of detections to the next, typically from the old leader to the new leader.

- **Multi-Target Tracking and Disambiguation**: Interesting signal processing and statistical techniques have been proposed [CHZ02] for dealing with tracking multiple vehicles and disambiguating the path of several vehicles as they move through the network.

2.1.3 Lessons

Both of these application classes require sensor network engineers to address a number of challenges, such as: low-power routing and communication, sensor calibration, localization (e.g., [PCB00]) and in-network processing (e.g., aggregation on GDI or position estimation in tracking applications.)

One primary design goal for both monitoring and tracking networks is minimizing power utilization. Both scenarios address this concern: the monitoring environment by carefully scheduling communication, and the tracking network by attempting to localize the active region to the area around the vehicle. To motivate this aspect of sensor network design, we next look at why power is such a critical resource in sensor networks and discuss the primary contributors to power consumption in a typical sensing environment.

2.2 Sensor Network Hardware

This section begins by describing the specific capabilities of the mote hardware on which TinyDB is implemented. This is followed by a brief comparison with alternative hardware platforms and a discussion of the variety of sensing hardware that might be interfaced to a mote-like platform.

2.2.1 Motes

Mica motes have a 4Mhz, 8bit Atmel microprocessor. Their RFM TR1000 [RFM] radios run at 40 Kbits/second over a single shared CSMA/CA (carrier-sense multiple-access, collision avoidance) channel. Newer Mica2 nodes use a radio from ChipCon [Corb] corporation which runs at 38.4 Kbits/sec. Radio messages are variable size. Typically about 20 48-byte messages (the default size in TinyDB) can be delivered per second. Like all wireless radios (but unlike a shared EtherNet [DEC82], 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. We describe communication in TinyOS in more detail in Section 2.3.

Motes have an external 32kHz clock that the TinyOS operating system can synchronize with neighboring motes to approximately +/- 1 ms to ensure that neighbors are powered up and listening when there is information to be exchanged between them [EGE02]. 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.

Both generations of Mica motes are equipped with 512KB of off-processor non-volatile Flash memory that can be used for logging and data collection. Writes of 256-byte Flash pages complete in about 10ms, whereas reads of such pages require only about 10 μs to complete [Cora].

Mica and Mica2 hardware have a 51-pin connector that allows expansion boards to be added. Typically, a *sensor board* is placed in this connector, which adds a suite of sensors to the device, though actuators, co-processors, LCD displays, and other hardware may also be attached. We describe the variety of sensors and sensor-boards which are currently available for Mica motes in Section 2.2.3

Table 2.1 summarizes hardware characteristics of Mica and Mica2 nodes. Figure 1.1 shows a picture of a Mica mote and a standard sensor board which offers sensors for light, temperature, magnetic field, acceleration, and sound.

Characteristic	Mica	Mica2
CPU Type	Atmel 128	Atmel 128
CPU Frequency	4 Mhz	8 Mhz
CPU Data Memory	4 KB	4 KB
CPU Program Memory	128 KB	128 KB
Radio Type	RFM TR1000	ChipCon
Radio Frequency	917 Mhz	917 Mhz, 433 Mhz
Radio Throughput	40 Khz	39.2 Khz
Flash Type	Atmel AT45DB041	Atmel AT45DB041
Flash Capacity	512 KB	512 KB
Flash Write Time	10 ms / 256 bytes	10 ms / 256 bytes
Flash Read Time	10 μs / 256 bytes	10 μs / 256 bytes

Table 2.1: Hardware Characteristics of Mica and Mica2 Motes

Finally, we note that current generation motes were developed as a testbed for "smart dust" – millimeter scale integrated processors, radios, and sensors that, if they are ever produced, should have roughly the same capabilities as the Mica hardware. A 3^2mm single chip mote was recently built at UC Berkeley [Hil03]. We expect that the research in this dissertation will apply equally well to such hardware. Indeed, the design of this single chip mote was partly motivated by the resource requirements of TinyDB.

2.2.2 Mote Alternatives

There are a number of other platforms that have been developed to serve similar purposes as the motes. Though, as we show, some of them have substantially more power than a mote, we believe that the basic capabilities and issues will remain the same.

The Smart-Its platform [BG03], developed by the Smart-Its Consortium, a collection of European universities led by the Telecooperation Office (TecO) at the University of Karlsruhe, Germany, is a ubiquitous computing platform with characteristics similar to the Mote platform – a small embedded processor (in this case a 20 Mhz PIC), a wireless radio (an RFM TR1000, which was used in the original Mica), and a suite of sensors, including light, temperature, and acceleration. The development efforts of the Smart-Its group have largely been focused on developing artifacts with novel modes of interaction – devices which, for example, become "associated" with each other when shaken at the same frequency and then begin beeping when they are out of radio range of each other.

Another popular wireless sensing platform is Sensoria corporation's Linux-based "sGate" [Core]. This platform is substantially larger (it weighs about 3 pounds without batteries), more powerful (300 Mhz instead of 4Mhz processor, 64 MB of RAM instead of 4K), and is much less energy efficient, with an operational power draw of about 10 watts, versus 30 mW for a mote or 40 watts for a typical laptop. It can be battery powered for "up to 24 hours" using massive lead-acid batteries. It includes a 10 or 100 mW 2.4 Ghz radio. The mGate is designed more as a basestation device for collecting readings from a number of wired sensor or mote-like devices than as an off the shelf smart-dust prototype. The use of Linux and inclusion of a high power processor and range of radio interfaces means this device will continue to be less substantially less energy-efficient than mote-like hardware over time.

Finally, there are a number of commercial data loggers that are available and widely used in the

scientific and industrial community. Campbell Scientific [Sci] is a well known vendor; their data loggers are relatively large instruments that can interface to a wide range of sensors and are designed to operate in a number of harsh environments. Their newest generation of data loggers includes interfaces to single-hop radios, and the smallest versions feature power consumption characteristics that are roughly comparable to motes. Additionally, larger, more power hungry versions offer sampling rates and voltage sensitivities that can significantly outperform mote hardware. These devices collect readings periodically or according to a scheduled "program". In addition to basic logging facilities, these data loggers support time averaging and the tracking of windowed minima/maxima. These devices are designed with a fairly different usage model than motes: users are expected to buy one or two at a high premium (thousands of dollars apiece) to monitor a few locations, as opposed to hundreds or thousands of motes to finely monitor an area.

2.2.3 Sensing Hardware

A variety of sensors have been interfaced with the motes; a partial list includes sensors for light, surface and ambient temperature, acceleration, magnetic field, voltage, current (DC and AC), sound volume, ultrasound, barometric pressure, humidity, and solar radiation. Some of these sensors, such as basic light and temperature sensors, are completely passive devices whose resistance varies with environmental conditions. Others, such as those on the weather-sensing board developed for the 2nd Great Duck Island deployment during the summer of 2003 [Pol03b], are self-calibrating digital sensors with integrated microprocessors.

In the context of TinyDB, we are concerned mostly with the energy costs required to fetch samples from these sensors. We return to their other characteristics, such as accuracy and drift, in Chapter 8. The variations among sensors are dramatic, both in terms of power usage and time to obtain a sample. Some devices, such as pressure and humidity sensors, require as long as a second to capture a reading, which means that the per-sample energy costs are very high. Other

devices, such as the passive thermistor, whose resistance varies with ambient temperature, require only a few microseconds to sample, and thus contribute only a negligible amount to the total energy consumption of the mote.

Table 2.2 summarizes the power, time-to-sample, and per energy costs for a variety of calibrated digital sensors. This information is derived from [Pol03b]; citations for the various sensor manufacturers are given in-line. Notice that the variation in the energy costs per sample are quite dramatic – two orders of magnitude in some cases. In Chapter 5 cases are given when these variations affect the behavior and performance of our query processor.

Sensor	**Time per Sample (ms)**	**Startup Time (ms)**	**Current (mA)**	**Energy Per Sample (@3V), mJ**
Weather Board Sensors				
Solar Radiation [Sol02]	500	800	0.350	.525
Barometric Pressure [Int02b]	35	35	0.025	0.003
Humidity [Sen02]	333	11	.500	0.5
Surface Temp [Sys02a]	0.333	2	5.6	0.0056
Ambient Temp [Sys02a]	0.333	2	5.6	0.0056
Standard Mica Sensors				
Accelerometer [Ana]	0.9	17	0.6	0.0048
(Passive) Thermistor [Atm]	0.9	0	0.033	0.00009
Magnetometer	.9	17	5 [Hon]	.2595
Other sensors				
Organic Byproducts[2]	.9	> 1000	5	> 15

Table 2.2: Summary of Power Requirements of Various Sensors Available for Motes

2.3 Communication In Sensor Networks

We now turn our attention to radio communication, beginning with a brief and high level overview of the radio technology used in motes and then turning to software issues related to communication and data collection in sensor networks.

[2] Scientists are particularly interested in monitoring the micro-climates created by plants and their biological processes. See [DJ00, CED+01]. An example of such a sensor is Figaro Inc's H_2S sensor [Fig]

2.3.1 Radio Hardware Characteristics

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 sensor network deployments must make use of multi-hop communication, where intermediate nodes relay information for their peers. On Mica motes, all communication is broadcast.

The radios in both Mica and Mica2 are both capable of encoding approximately 40kbits/second of data onto the air. In the case of Mica2, the radio is running at a data rate of 76.8 Kbps, but uses Manchester encoding (where a logical 0 is sent as a 0 followed by a 1 and a logical 1 is sent as a 1 followed by a 0) – providing a delivered bandwidth of 38.4 Kbps. Manchester encoding is used to avoid synchronization errors which commonly occur in demodulation phase-locked-loop circuits when long sequences of 1's or 0's (e.g. signals with no transitions) are received.

Packets in TinyOS are (by default) 37 bytes, 30 of which are application data and the other 7 of which are packet header and CRC data from TinyOS. In addition, on Mica2, each packet is preceded by 19 bytes of preamble and synchronization data that is used to detect the start of an incoming packet [3]. The throughput is further reduced by random backoff before packet transmission that is at least 16 byte-times long (and is unbounded in the event of a large amount of radio contention). Thus, for each 30 byte packet, at least $7 + 19 + 16 = 42$ byte-times of overhead are incurred, for a effective application data rate of $(30bits/(30bits + 42bits)) * 38.4kbits/sec = 16kbits/sec$. Thus, the theoretical maximum number of 30 byte packets per second is about 70. In practice, however, with 1 or 2 senders, it is unusual to be able to successfully deliver more than about 20 packets per second.

[3] We do not describe in detail the packet overheads of the Mica TR1000 radio. They are roughly comparable to those of the Mica2 CC1000 Radio.

2.3.2 Communication Issues In Sensor Network Software

Radio is a broadcast medium. The operating system provides a software filter so that messages can be addressed to a particular node. If neighbors are awake, they can, however, still *snoop* on such messages at no additional energy cost since they have already transferred the decoded the message from the air. Nodes receive per-message, link-level acknowledgments indicating whether or not 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 by the devices (i.e., in an ad-hoc manner) rather than fixed at the time of network deployment. Typically, devices keep a short list of neighbors that 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 a simple ad-hoc routing protocol called *tree-based routing* in the next section.

2.3.3 Tree-Based Routing

A basic primitive in many data dissemination and collection protocols is a *routing tree*, which is a spanning tree rooted at a particular node over the radio-connectivity graph of the network [Lyn96]. In TinyDB, a routing tree 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 query results, as well as those of its children, to the basestation. We note that this type of *tree-based* communication topology is common within the sensor network community [WC01]. Finally, note that it is possible to have several routing trees if nodes keep track of multiple parents. This technique can be used to support several simultaneous queries with different roots.

Figure 2.2 shows an example sensor network topology and routing tree. Solid arrows indicate parent nodes, while dotted lines indicate nodes that can hear each other but do not use each other for routing. Notice that, in general, a node may have several possible choices of parent; typically, the chosen parent is the ancestor node that is closest to the root. In the event that multiple ancestors are at the same level, the node with the highest link quality is chosen. The details of this protocol are somewhat complicated, as nodes must learn to deal with routing loops, asymmetric link qualities, and so on. For a complete discussion of these issues, and a proposal for a link-quality metric that works well in practice, see [WTC03].

Once a routing tree has been constructed, every node has a depth d (where d is the node's depth in the tree) hop route to the root of the network. Thus, the average cost of collecting a single piece of information from a network with n nodes will be $log(n)$ messages, assuming a balanced routing tree [4]. Note, however, that the cost of constructing a routing tree or flooding a message to all nodes in the network is n messages. Thus, flooding is an expensive operation relative to basic data collection.

Note that the radio in current generation sensors is quite lossy: the ChipCon radio used on Mica2 motes, for example, will drop 20% to 30% of packets received at a range of about 10 meters. Of course, it is possible to retransmit these messages to avoid the losses at the application level, but this consumes additional energy.

Because of these high loss rates, as packets are forwarded, the network management layer is

[4] Though it may not be possible to construct a balanced tree, the worst-possible average data collection cost per mote will be $n/2$ messages in the case where all sensors are in a line – an extremely unlikely situation

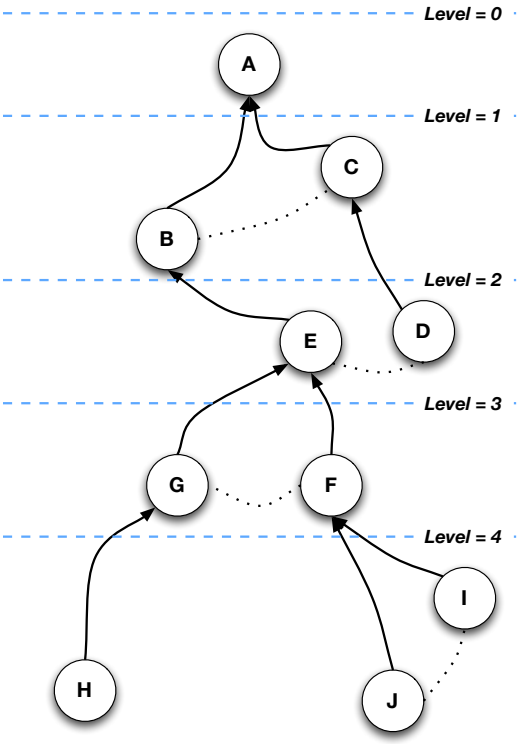


Figure 2.2: A Sensor Network Topology, with Routing Tree Overlay. Solid arrows indicate a child-parent relationship, while dotted lines indicate radio connectivity that is not (generally) used in routing.

constantly engaged in *topology maintenance*. This maintenance consists of trying to determine if there is a better (e.g., a less-lossy, or closer-to-the-root) choice of parent in the tree-based routing algorithm and tracking neighbors and children for use in other aspects of routing and query processing. The reader is referred to [WTC03] for more information on the current state of the art in network topology construction and maintenance.

2.3.4 Power Consumption In Sensor Networks

Given the hardware overview above, we now examine the relative costs of various elements of sensing, communication, and processing to develop a metric for performance in sensor networks. Our primary metric is *rate of energy consumption*, or power, which we typically measure in Joules

per unit time, or Watts (Joules per second).[5] We focus on energy for several reasons:

- Communication, processing, and sensing all consume energy, so minimizing energy consumption will require minimizing all three of these resources.

- 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.

- Processing time, in current generation sensor networks, consumes a small percentage of the total energy. The TinyDB system provides facilities to allow users to express complex functions over sensor readings, but this dissertation does not address issues related to making those functions computationally efficient. Instead, we seek to show that our framework is energy-efficient and promotes ease of use for both users and developers of such functions.

To give a sense of how energy consumption is divided between processing, communication, and sensing, we calculated the expected energy utilization of a Mica2 mote involved in a typical data collection scenario. The complete results of this study are shown in Appendix A. In this study, we measure the energy used in various phases of processing during a simple data-collection scenario where a mote transmits one sample of (air-pressure, acceleration) readings every ten seconds and listens to its radio for one second per ten-second period to receive and forward results for a small group of neighboring sensors. We found that 97% of energy consumption in this scenario is related

[5] 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 TinyOS motes run at 3V.

to communication, either from directly using the radio or as a result of the processor waiting for the radio to send data to or receive data from neighboring devices.

Because communication and time spent in the processor waiting for the radio tend to dominate power consumption, we typically use communication as a proxy for energy consumption in the performance metrics in this dissertation. There are some cases where the cost of sensing become significant (for example, certain sensors require as long as a second to power-up before they can provide a sample) – we consider these issues when we address sample acquisition in Chapter 5.

2.3.5 Summary

This completes our discussion of the basic technology available in today's sensor network nodes. We introduced the basic mote hardware, discussed the properties of its CPU and radio and saw how it differs from alternative sensor network platforms. We introduced the basic *tree-based* algorithm for communication in multi-hop sensor networks. Finally, we looked at the importance of power as a metric in this environment, studying the breakdown of power consumption in a typical data collection application, where we noted the significant proportion of power devoted to radio communication.

2.4 Technology Trends

This section examines trends in storage, communication, and CPU technology to understand the resources and constraints that will shape sensor network research over the next few years. As we show, it is likely that the CPUs on motes will continue to get more powerful and energy-efficient, but unlikely that the energy costs of communication or the capacity of batteries will change much over the next several years.

It is important to consider these trends because they provide insight into the research problems that will continue to be important in the future versus those which will disappear as hardware be-

comes more powerful – for example, throughout this dissertation, we focus on power-management, because we do not expect the energy density of batteries to improve dramatically in the near future, whereas we downplay the limitations imposed by a small memory footprint, even though running TinyDB in 4KB of RAM has been a significant implementation challenge.

2.4.1 Energy Storage Technologies and Capacity

Figure 2.3 shows the energy density (in kW/kg) of various batteries and fuel cells.Numbers in this figure were drawn from the companies indicated at the bottoms of the columns, or from *The Art of Electronics* [HH89]. Notice that the capacity of commercial batteries have increased from 150 to 230 kW/kg since 1989, and that the best case technology (micro-fuel cells from MTI micro) currently demonstrated in a lab is about 5 times as efficient as that.

Especially given the losses in efficiency that will certainly result as fuel-cells are deployed in the real world (notice the relatively modest density goals for automotive fuel cells), it seems unlikely that energy storage technologies will obviate the need for power management in wireless sensor networks.

It is, however, the case that the efficiency of transistors and radios may increase so dramatically that current energy storage technologies will be more than adequate for years of operation. Indeed, we expect that silicon technology will continue to scale according to Moore's law, but radio technology does not appear to be on the same curve, as discussed in the next two sections.

2.4.2 Wireless Communication

According to the Berkeley Wireless Radio Center's (BWRC) PicoRadio project, it should eventually be possible to build a 100μW radio transceiver (at 10% duty cycle), with a range of about 10 meters [RAK+02a]. The current Mica2 radio uses about 15mW when running in the 10m range, [Corb] when operating at 100% duty cycle, or about 1.5 mW at a 10% duty cycle. Thus, the difference

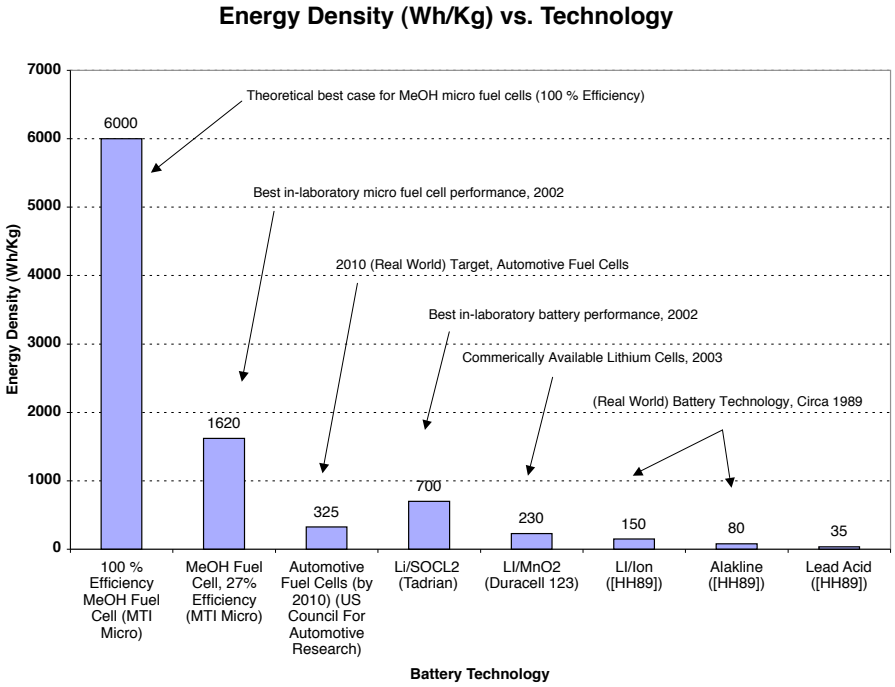


Figure 2.3: Energy Density, Fuel Cells and Batteries

in power consumption between a cheap, commercially available radio and the best the research community can conceive is about an order of magnitude. Though this reduction is significant, it is not enough to eliminate concerns associated with the energy costs of communication, particularly given that the PicoRadio predications hold only in low-channel-utilization, low-range environments which may not always be available.

One of the particularly interesting observations from the BWRC is that multihop routing will continue to be used in these networks, even if long range, relatively low power networks come into existence. The reason for this is that a multihop topology is fundamentally lower power than a single hop topology. This is because the energy to transmit goes up by a factor between the square and the cube of the distance between the sender and receiver, depending on the environment in which communication is being done – fading and multi-path effects in obstructed urban and indoor

environments substantially limit the maximum transmission range. To illustrate this, imagine that we have a radio in an unobstructed outdoor environment that can transmit 100 feet for a unit cost. To transmit 1000 feet via multihop will require 10 unit-cost transmissions, for a total cost of 10. By simply increasing the transmission power of the radio, its power would have to increase by a factor of 100 to transmit the same 1000 feet.

Thus, not only do we expect that communication will continue to be the dominant component of the energy budget in sensor networks, but also that multi-hop communication will continue to be used in most power sensitive applications.

2.4.3 Processing Costs

According to [Sak01], we can expect the density of transistors on high-end microprocessors to increase by a factor of 30 from their 1999 levels by 2014; in this same period, researchers expect that the power requirements of these same chips will double. Thus, it will be possible for mote-like systems in 2014 to use approximately 1/15th the energy of today's processors while retaining the same number of transistors in 1/30th the area. According to this same research, the capacity of DRAM chips should increase by a factor of 1000 in the next 15 years.

Because the energy cost of processing will continue to fall, and at a greater rate than radios will increase in efficiency, we expect that communication will continue to be the dominant factor in power consumption in the foreseeable future.

Furthermore, because RAM density is expected to increase so dramatically, we expect that concerns over the memory available on motes today will be practically irrelevant in the near future. Or, phrased differently, the storage capacity increase will be so dramatic (e.g. motes in 2015 will have 1/2 GB of Flash) that it is reasonable to consider fairly sophisticated signal processing algorithms, such as computer vision and tracking, running on motes.[6]

[6]One possible concern with large flash memories is their slow write performance. According to Intel at a recent

2.4.4 Alternative Technologies

There are a number of alternative technologies that promise to help reduce the power consumption or mitigate issues with battery technology in coming years. Examples of such technology include very low-power asynchronous logic, wind-power, energy harvesting, and solar cells. In this section, we focus on solar radiation, as it is the most promising (and readily available) resource that will allow motes to be divorced from batteries, or to at least reduce the frequency with which batteries must be replaced.

Solar Cells

Solar radiation provides about 1000 W/m^2 of energy in full sunlight on a bright day, or an average of about 200 W/m^2 over the course of a day over the entire year (in La Jolla, California) [Sys02b]. Solar cells, or *photovoltaics* convert this energy into electrical current, wasting substantial amounts of energy in the process. According to [Sto93], the best theoretical efficiency for this conversion is about 29 percent (without using concentrators like lenses or mirrors); the best cells known today provide about 15 percent efficiency, with commercially available cells being closer to 10 percent. At this efficiency, to provide the peak current of 50 mW needed by today's motes we can compute the size of a solar cell needed to power a mote to be:

$$.05W/(200W/m^2 * .1) = .0025m^2 = 25cm^2$$

Thus, a 5cm x 5cm panel would provide enough energy to power a mote continuously; however, this solution will only work outdoors, and will not obviate the need for power conservation as the energy available at night or on a cloudy day is substantially less than the average across a whole year. This size cell is currently too large to be economically viable for motes; however, a ten-fold reduction in the power consumption of the hardware and a doubling in the efficiency of solar cells,

Developer's Forum [Int02a], non-volatile memory technologies will perform much better in the near future. For example, non-volatile Ovonic unified memories (OUM) at 4 Mbit sizes with read/write performance comparable to traditional RAMs have already been prototyped.

both of which are plausible over the next ten years, would require a solar cell of just 1.25 cm^2, which is likely feasible.

2.4.5 Discussion

In this section, we have discussed three primary technology trends:

1. *Battery energy capacity* is not expected to increase by more than a small constant factor in the next ten years. Solar cells will likely become a viable alternative for outdoor deployments.

2. *Energy costs of communication* will decrease by as much as an order of magnitude per bit in the future given the best possible conditions, but will not become negligible.

3. *Energy costs of computation* will decrease by more than an order of magnitude in the next ten years; thus, the relative ratio between the energy costs of communication and computation will continue to widen.

This argues that the resource constraints we discussed in Chapter 1 – limited bandwidth and power – will continue to be the primary limitations in the near future.

This completes our discussion of the hardware technology behind sensor networks. We now shift focus and discuss TinyOS, the platform upon which TinyDB is based.

2.5 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. [Inc], and MIT's Cricket [PCB00] platform.

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, which we describe in Appendix D, designed to promote extensibility and composition of software while maintaining a high degree of concurrency and energy efficiency. Interested readers should refer to this appendix, though knowledge of nesC is not needed to follow this dissertation.

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:

- The radio stack, which sends and receives packets over the radio and manages the MAC layer [HSW+00, WC01].

- Software to read sensor values – each sensor device (e.g. the light sensor) is managed by a software component that provides commands to fetch a new reading from the sensor, and possibly access calibration and configuration features for more sophisticated digital sensors.

- Components to synchronize the clocks between a group of motes and schedule timers to fire at specified times [EGE02].

- Power management features that allow an application to put the device into a low power sleep mode without compromising the state of any other software components.

- Software to manage the off-chip Flash using a simple, file-system like interface.

- Components to localize a sensor relative to neighboring node by emitting sound or ultrasound pulses and measuring their time-of-flight to those neighbors [Whi02].

2.5.1 Summary

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 this dissertation. 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.

We assert that sensor networks will never be widely adopted if every application requires this level of engineering effort. As we show, the declarative model we propose reduces these applications to a few short statements in a simple language. In the next section, we introduce the basics of declarative query processing, which is the foundation for the language we propose in the next chapter.

2.6 Query Processing and Declarative Database Systems

So far, we have discussed background and related research from the sensor network literature. The notion of declarative queries was introduced in Chapter 1; the basic idea is that users specify what data they are interested in, not how the system should go about collecting that data. This frees the user from many of the difficult programming tasks associated with efficiently collecting and processing data. Rather than describing declarative techniques in detail here, we provide a tutorial on declarative queries and the SQL query language in Appendix E; Readers unfamiliar with these concepts may wish to review this appendix.

The next chapters show that traditional applications database techniques for query optimization and plan construction apply naturally to sensor networks. We propose a variant of SQL specially tailored to such networks, discuss ways in which this language isolates users from the underlying sensor network issues, and show examples of special optimizations (e.g., plan alternatives) that arise the context of smart sensors.

Finally, we note that, during the 1980's and 90's, the query processing community devoted a great deal of energy towards designing distributed query processing systems that ran over networks of machines, either in a local (e.g., parallel) environment, or spread across the wide area. Some of the issues that arose in these contexts have bearing on the distributed sensor network environment studied in this dissertation. Rather than summarizing the relevant pieces of this literature here, however, we return to the methods used in distributed databases throughout the remainder of the dissertation, describing ways in which such techniques are (or are not) appropriate for sensor networks. For an excellent overview of traditional distributed query processing techniques, see [Kos00].

2.7 Related Work

The previous sections introduced the technology and trends we draw upon in describing our approach in the next chapters. In this section, we survey relevant research from the sensor networks, database, and traditional networking communities. As TinyDB builds on research from a very broad, cross-cutting set of ideas, we cover a large number of projects. In successive chapters, more detailed comparisons are made to particularly relevant research.

2.7.1 Sensor Network Projects

We begin with a discussion of large research projects from the sensor network community that are motivated by goals similar to our own. We show why these alternative solutions do not address all of the issues covered in this dissertation.

Cougar

The Cougar project from Cornell is also exploring a declarative interface for sensor networks; their early work [YG03, PJP01] proposed an append-only, single table schema that inspired the data layout in our own work, and they were among the first researchers to note that declarative interfaces for sensor networks are a good idea. In general, however, the Cougar project has focused on simulations and implementations on Linux based hardware which has prevented them from addressing what we view as many of the central issues in sensor network query processing; in particular:

- Little concern is given to power-efficient communication scheduling, or to reducing power utilization.
- No consideration is given to the *acquisitional* nature of query processing, which we focus on in Chapter 5. In Cougar, there is no discussion of the computational or energy costs of sampling sensors, one of the interesting and important new aspects of query processing in sensor networks.
- Many of the unique opportunities in the world of sensor networks, such as novel kinds of optimizations created by their highly distributed nature and use of radio communication are not discussed.
- XML is used to encode messages and tuples; we believe the verbosity of XML makes it inappropriate as a wire format for the bandwidth constrained networks we are considering.

Thus, though our work and the Cougar project are conceptually similar, the Cougar project offers little in the way of examples of optimizations or deployments to make a convincing case that their approach is the right approach for low power sensor networks.

The Cougar project has, however, proposed several interesting communication protocols and data types that should be easily adaptable to the power efficient optimizations and techniques which

we propose. For example, the notion of a "Gaussian ADT" [FGB02] provides a mechanism for combining and processing results with inherent uncertainty, such as readings collected from sensor networks.

As another example, their recently proposed wave scheduling technique [TYD+03] provides an efficient approach for increasing the utilization of a shared communication channel that appears to work well in simulation. Our experience suggests that implementing such a complex schedule on real hardware may be difficult, but if successful, a real implementation could help mitigate bandwidth limitations in large sensor networks.

Directed Diffusion

Within the sensor network community, work on networks that perform data analysis is largely due to the USC/ISI and UCLA communities. Their work on directed diffusion [IGE00] discusses techniques for moving specific pieces of information from one place in a network to another, and proposes aggregation and other query-processing like operations (or *filters*) that nodes may perform as data flows through them.

Diffusion recognizes that in-network processing and aggregation dramatically reduce the amount of data routed through the network, but focuses on application-specific solutions that, unlike the declarative query approach of TinyDB, do not offer a particularly simple interface, flexible naming system, or any generic aggregation and join operators. In diffusion, such operations are viewed as application-specific operators, and must always be coded in a low-level language. Such user-defined operators are very difficult for a query optimizer to deal with, because it cannot understand their semantics, and thus cannot be sure when it may compose or re-order them with respect to other operators. As a corollary of this lack of system-managed semantics, the user bears the burden of getting the placement and ordering of operators correct, significantly complicating the development of complex and correct programs.

Sensor Network Companies

Several companies have recently been started with the goal of commercializing sensor network technology. Of particular interest are software companies like Ember [Corc], Millennial Net [Cord], and Xsilogy [Corf], as they are developing software solutions designed at achieving some of the same goals as our research. These commercial solutions, however, view the mote-like sensor devices as *push* nodes that periodically broadcast their data, rather than as active participants in the data collection and computation process. This means that many of the in-network aggregation and filtering techniques described in this dissertation are not possible, limiting the data collection bandwidth and scalability of diffusion-based applications in these environments.

The virtue of our declarative interface is that it allows many nodes to be configured, tasked, and monitored as logical groups. In contrast, these commercial solutions view sensors as disjoint entities, each of which is monitored and configured independently. This approach may work for a few tens of devices, but as sensor networks grow to thousands of nodes, this approach will become an insurmountable administrative burden; we believe our declarative approach will not suffer these same limitations.

2.7.2 Related Database Technologies

Aside from the work from the Cougar group, there has been little work in the core database community on query optimization or data collection in very low power distributed environments like sensor networks.

Mobile and Power Sensitive Databases

There is a small body of work related to query processing in mobile environments [IB92, AK93]. 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 [WSX+99]) 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 [AG93], 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 fewer power optimizations that can be applied.

Temporal and Continuous Queries

Related work has also been done in the temporal and streaming database communities. The query language we propose is related to languages and models from the Temporal Database literature; see [Sno95] for a survey of relevant work.

More recently, there has been a substantial amount of activity on the topic of continuous [MSHR02, CDTW00, LPT99] and stream [MWA+03, CCC+02, MSHR02, CCD+03] query processors. These systems seek to address language and performance issues that arise in the general context of endless streams of data flowing into a system; the languages proposed are likely adaptable to our domain, and some of the techniques (eddies [AH00], for example) are also applicable. In general, the techniques in these papers are aimed at substantially more powerful devices than TinyDB, making them less immediately useful in the context of sensor networks. However, because these systems are semantically compatible with the query language developed in this dissertation,

they will certainly prove useful for processing data that is output from a sensor network database like TinyDB.

2.7.3 Related Networking Projects

Protocols for routing in wireless networks have been a very popular research topic in the networking community [KRB99, AWSBL99, GAGPK01]. None of them, however, address higher level issues regarding how to effectively process data; rather, they are largely techniques for data routing.

Recently, there have been some higher level networking efforts designed to facilitate data collection in sensor networks. Though none of these approaches specifies a query language or mechanism for query evaluation, many of them speak of *querying the network*, suggesting that our techniques and languages would likely be applicable. We describe three recent network projects that propose interesting new communication mechanisms for sensor networks.

DTN and Zebra Net

Several publications have recently appeared related to routing in a disconnected environment. The notion of a *delay tolerant network* (or DTN) is introduced in [Fal03]. DTNs are networks subject to periodic disconnections between the source and sink, such that packets may have to be queued, or dropped or not admitted to the system if queues fill. A networking substrate that provided a DTN like interface would be extremely useful in a system like TinyDB, which currently manages the variable bandwidth provided by a sensor network in a fairly ad-hoc way, which we discuss in more detail in Section 5.5.

The ZebraNet [JOW+02] proposes DTN like semantics for sensor networks. The basic idea is that sensors are attached to zebras (or other wildlife) roaming in an African game preserve. Scientists would like to collect information from these animals, but cannot know when they will come into radio range of another node – be it one with a reliable connection to the Internet or a sensor on

another zebra. So the system must collect and store data for as long as possible, handing it off when connections to other nodes occur. Again, one could imagine layering TinyDB-like functionality on top of this networking substrate, or using query semantics to shape decisions about which data to store and which to forward.

Greedy Perimeter Stateless Routing

A second related networking idea is greedy-perimeter-stateless-routing (GPSR) [KK00], a technique for any-to-any routing when nodes know their physical location and data is destined for a particular geographic region. The basic idea is for nodes to keep track of their geographic positions as well as the positions of their immediate neighbors. Then, to route a message to a particular location, they simply forward the message to the neighbor nearest that location. Because it is possible that the only route to a particular location actually requires a network hop that is further from the destination than some local minima, a "perimeter mode" is used to make forward progress in this case. GPSR is an attractive technique when any-to-any routing is needed, and it should be readily usable in query processing applications like TinyDB.

Geographic Hash Tables and Other Storage Models

Finally, there has been some recent publication on techniques for distributing storage throughout sensor networks. Geographic hash tables [RKY+02] store data at a geographic location (using GPSR) corresponding to the two dimensional hash of a particular field (the *event type*) from that data item. In this way, fields with the same event type hash to the same location, and queries interested in a particular kind of event can quickly find data items relevant to it using GPSR.

An alternative approach to GHTs is DIM [LKGH03], which proposes multi-dimensional range index for querying a sensor network. Each node is allocated a contiguous region of the index-attribute space, and neighboring nodes in the network store adjacent ranges of the attribute space.

The advantage of this approach over GHTs is that it supports range queries (e.g., "find all the sensor readings where the temperature was greater than 80 ° F").

Both of these techniques will prove useful when implementing distributed storage in TinyDB becomes a priority – for the time being, we have focused on local storage (where each node stores the data it produces), and thus, have not needed this kind of distributed functionality.

2.7.4 Retasking and Reprogramming Techniques

In this section, we discussed the need for *re-tasking* in the Great Duck Island application. A variety of alternative re-tasking mechanisms, including virtual machines [LC02] and downloading new code images into the sensor via the radio have been proposed. The declarative approach proposed in this dissertation is an alternative to these programming methods. Declarative queries are an extremely easy way for end-users to write programs (unlike either VM programs or code images), are compact (only a few radio messages, in most cases), and are safe, such that the network is guaranteed not to crash or cease to function when an ill formed query is issued.

2.8 Discussion

In this section, we discussed the current state of the art in hardware and software for sensor networks, focusing on the design of TinyOS and the Berkeley mote hardware. We showed that power, particularly as it relates to sensing and communication, is the primary resource limitation of these networks, and argued that it will continue to remain so in the foreseeable future. We also summarized the basics of query processing, describing the general advantages of the declarative approach.

Finally, we discussed sensor network research closely related to our own. We believe that none of that research completely addresses the challenges and needs of deployments like GDI or the NEST tracking application. In particular, other declarative approaches like Cougar [YG02, PJP01] do not adequately address power management and communication sensitive query processing tech-

niques, while efforts like Directed Diffusion [IGE00] do not provide a reconfigurable interface which is easy to deploy in a variety of environments.

With these arguments in mind, we now present our architecture for query processing in sensor networks, illustrating how it addresses these challenges and describing the details of the TinyDB system that we have built for sensor network data collection.

Chapter 3

Query Processing Architecture for Sensor Networks

In this chapter, we begin by listing the key requirements of query processing in sensor networks; we then discuss the query language, features, and implementation of TinyDB and show how its design is well suited towards addressing these requirements. In particular, we show that the declarative approach enables TinyDB to carefully address resource management challenges in ways that users and most programmers could not, much as a traditional database system manages issues like operator ordering or physical data layout using techniques that would escape many developers.

This chapter presents the language, operators, and communication primitives that are central to TinyDB. We rely heavily upon these techniques in the following two chapters, where we focus on advanced processing techniques for improving power efficiency of sensor network query processing.

3.1 Design Requirements

We begin by describing the requirements that were used to drive the design of TinyDB. As described in the previous chapters, the first major challenge that must be addressed is resource management, particularly power. Recall that, communication and (to a lesser extent) sensing tend to dominate power consumption given the quantities of data and complexity of operations that are feasible on sensor networks. Furthermore, as discussed in Section 2.4 technology trends suggest that the energy

cost per CPU cycle will continue to fall [Sak01] as transistors become smaller and require less voltage, whereas fundamental physical limits and trends in battery technology suggest that the energy to transmit data via radio will continue to be expensive relative to the energy density of batteries.

The second major challenge has to do with managing the dynamic, transient nature of sensor networks: nodes come and go and signal strengths between devices vary dramatically as batteries run low and interference patterns change, but data collection must be interrupted as little as possible.

Third, a sensor network query processor must provide data reduction, logging, and auditing facilities. Transmitting all of the raw data out of the network may be prohibitively expensive (in terms of communication) or impossible given data collection rates. Since many applications, such as those looking for vibrations or measuring the speed of moving objects, will need sample rates of 100Hz or more, it is not always possible (or desirable for energy reasons) to deliver all data out of a large sensor network. Instead, some reduction of the data is necessary – reporting, for example, observed frequency of vibration or actual speeds of objects. However, users of sensor networks often want access to the raw data used to answer a query in order to verify accuracy and analyze the data in new ways. Thus, signal processing and aggregation techniques are needed to reduce the data that is transmitted online, but logging and offline delivery techniques are also needed for auditing and analytical processing of data.

Fourth, the system must provide an interface that is substantially simpler than the component-based embedded-C programming model of TinyOS. While being simple, this interface must also allow users to collect desired information and process it in useful ways.

Finally, users need tools to manage and understand the status of a deployed network of sensors. Such tools must facilitate the addition of new nodes with new sensors and the reconfiguration of existing deployments with new data collection tasks or rates.

In the following sections, we present a high-level overview of the system architecture we de-

signed to satisfy these requirements. We show how this architecture promotes energy and communication efficiency, and, as we mentioned in Chapter 1, how its declarative interface is central in addressing these challenges.

3.2 Query Language and Semantics

To introduce many of the features of TinyDB, we begin with an overview of the syntax and semantics of the query language used in TinyDB. Appendix B provides a complete specification of the query language.

As we discussed in the previous two chapters, queries provide a simple, declarative (e.g. non-procedural) mechanism for describing what data users are interested in. Asking users to simply declare their interests rather than specify a procedure for how the relevant data should actually be retrieved is the key to both *ease-of-use* and *performance* in sensor networks.

The ease-of-use argument arises from the fact that users need not know where in the network a particular data item resides and do not need to concern themselves with the subtleties of power management, communication scheduling, multihop networking, or embedded C programming. They simply write a short declarative statement, inject it into the network, and the system begins streaming out answers.

The idea that declarative queries can also perform well is perhaps less obvious. The basic argument is the same as in traditional database systems: since the user does not fully specify how data must be acquired and processed, the system is free to choose from any of a number of plans when deciding how to evaluate a query. In some cases, these options are the same in sensor networks as in traditional database systems – for example, highly selective operators should typically be executed early in a sensor network just as in a traditional database system. However, there are other cases where sensor networks present interesting new optimization possibilities: for example,

in Chapter 5, we show that certain sensors require much more energy to sample than others, and discuss how those energy differences can be exploited during query execution to provide significant reductions in the energy costs of queries.

3.2.1 Basic Language Features

Queries in TinyDB, as in SQL, consist of a `SELECT-FROM-WHERE` clause supporting selection, join, projection, and aggregation. We also extend this with explicit support for sampling, windowing, and sub-queries via materialization points. As is the case in the Cougar system [PJP01], we view sensor data as a single table with one column per sensor type. Results, or *tuples*, are appended to this table periodically, at well-defined intervals that are a parameter of the query. The period of time from the beginning of each sample interval to the start of the next 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 sensor should report its own id, light, and temperature readings once per second for ten seconds. Thus, each epoch is one second long. The virtual table `sensors` contains one column for every attribute available in the system and one row for every possible instant in time. The term *virtual* means that these rows and columns are not physically materialized – only the attributes and rows referenced in active queries are actually generated.

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 an epoch number corresponding to the time it was produced.

When a query is issued in TinyDB, it is assigned an identifier (id) that is returned to the is-

suer. 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, as shown above

Note that in TinyDB, the `SAMPLE PERIOD` must be at least as long as the time it takes for a mote to process and transmit a single radio message and do some local processing – about 30 ms (including average MAC backoff in a low-contention environment) for current generation motes. This yields a maximum sample rate of about 33 samples per second per mote.

3.2.2 Storage Points and Joins

Note that the `sensors` table is (conceptually) an unbounded, continuous *data stream* of values. As is the case in other streaming and online systems [MSHR02, MWA+03, CCD+03, CCC+02], certain blocking operations (such as sort or nested-loops 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 seconds
 AS (SELECT nodeid, light FROM sensors
 SAMPLE PERIOD 1s)
```

This statement provides a shared, local (i.e., single-node) location called `recentLight` to store a streaming view of recent data similar to materialization points in other streaming systems like Aurora [CCC+02] or STREAM [MWA+03], or materialized views in conventional databases. We also allow storage points to be created without a defining query, for example:

```
CREATE
 STORAGE POINT recentLight SIZE 8 seconds
 (nodeid uint16, light uint16)
```

Users may then issue queries which insert into storage points, for example:

```
(1)    SELECT nodeid, light
         INTO recentLight
         SAMPLE PERIOD 1s
```

Only one query may write to a storage point at a time.

Of course, a storage point of fixed size can become full. We allow storage points to be created with different management policies, such that referencing queries can be halted when they try to insert into a full storage point or can overwrite the oldest existing row (treating the storage point as a circular buffer). The former condition (halting) is the default; the latter can be specified by writing `CREATE CIRCULAR` in the storage point definition.

A storage point can be deleted using the `DROP STORAGE POINT` command. When such a command is received, any queries inserting into or reading from the storage point are immediately aborted.

Joins are allowed between two storage points on the same node, or between a storage point and the `sensors` relation. When a `sensors` tuple arrives, it is joined with tuples in the storage point at its time of arrival. This is a *landmark query* [GKS01] common in streaming systems. Consider, as an example:

(2)
```
SELECT COUNT(*)
 FROM sensors AS s, recentLight AS rl
 WHERE rl.nodeid = s.nodeid
 AND s.light < rl.light
 SAMPLE PERIOD 10s
```

In this query, `COUNT` is an example of an aggregation function. 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. We require that join-queries include an equality-join on `nodeid`, since we do not currently allow readings to be combined across sensors (as tree-based routing does not easily support routing between an arbitrary pair of nodes.)

In the event that the `SAMPLE PERIOD` in a storage point definition and the `SAMPLE PERIOD` in a query reading from that storage point differ, two simple rate matching constructs are pro-

vided that allow interpolation between successive samples (if the SAMPLE PERIOD in the query is shorter), or specification of an aggregation function to combine multiple rows (if the sample period in the storage point is shorter.) For example, because Query 1 enqueues into the storage point ten-times faster than Query 2 reads from it, we could have written:

```
SELECT COUNT(*)...
 COMBINE avg
```

in Query 2 above. This would cause each row read in the join query to be the average of the last ten rows enqueued into the buffer. Conversely, if the join query were to read from recentLight at a faster rate than query 1 enqueues into it, we could write:

```
SELECT COUNT(*)...
 LINEAR INTERPOLATE
```

This would cause every row read by the join query to be a linear interpolation between the next-to-last row in the storage point and the last row in the storage point. Note that use of the LINEAR INTERPOLATE clause causes the output of the join query to lag behind the data in the storage point by one row, since interpolation requires access to the next row to determine the value of the current (interpolated) row.

3.2.3 Aggregates and Temporal Aggregates

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 ([IGE00, YG02]). TinyDB includes a mechanism for user-defined aggregates and a metadata management system that supports optimizations over them. We discuss the semantics and execution of aggregate queries in more detail in Chapter 4.

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

the maximum sound volume over time and reporting that volume periodically; this could be done with the following query:

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

This query will report the average volume over the last 30 seconds once every 5 seconds, sampling once per second. The `WINAVG` aggregate is an example of a *sliding-window* operator common in streaming systems [CCD+03, MWA+03, GKS01]. The final two parameters represent the window size, in seconds, and the sliding distance, in seconds, respectively.

Note that there are some constraints on the values of these parameters: the window size and sliding distance must be greater than or equal to the sample period, and the window size must be greater than or equal to the sliding distance.

3.2.4 Output Actions

Finally, queries may specify an `OUTPUT ACTION` that will be executed when a tuple satisfying the query is produced. This action can take the form of a low-level operating system command (such as "Turn on the red LED"), or the instantiation of another query. For example, the query:

(3)
```
SELECT nodeid, temp
 WHERE temp > 100° F
 OUTPUT ACTION alarm()
 SAMPLE PERIOD 1 minute
```

will execute the command `alarm()` whenever a tuple satisfying this query is produced. This command is stored in system catalog (described in Section 3.7 below) and is essentially an invocation of an arbitrary system function.

Given this introduction to the basic TinyDB query language, we now proceed to describe how these queries are executed within a network of motes.

3.3 Mote-Based Query Processing in TinyDB

Figure 3.1 shows a simple block diagram of the TinyDB architecture for query processing in sensor networks. The architecture consists of two main pieces: sensor-side software that runs on the motes and server software that runs on a PC (the *basestation*).

The sensor software that runs on the motes is the heart of TinyDB and is the primary focus of this chapter. As shown in the "Distributed In-Network Query Processor" detail box on the left side of Figure 3.1, this software consists of a number of components built on top of TinyOS. We give an overview of these in this section and then focus in detail on the query processing technology in Chapters 4 and 5 of the dissertation.

Because the motes are lacking with respect to input, display devices, and large storage devices, TinyDB relies on the PC-based basestation for accepting queries and outputting and storing results. We describe the structure of the basestation software in Section 3.9.

The TinyDB mote-side software consists of approximately 20,000 lines (6,500 semicolons) of C code. This includes several components that would commonly be associated with the operating system, such as a dynamic memory manager and a metadata manager that were created to address specific needs of TinyDB that are not otherwise met by TinyOS. Each mote runs a nearly identical copy of this code, although motes may have different physical hardware available to them and so may vary somewhat in the low-level components included for acquiring sensor readings. Appendix C shows a breakdown of the various software components by size and describes each of them briefly.

3.3.1 Overview

Once a query has arrived on the sensor, it is passed through the components of the software stack shown in Figure 3.1. This stack represents the major phases of query execution, as follows:

- In the **network management layer**, queries and results are received and sent over the net-

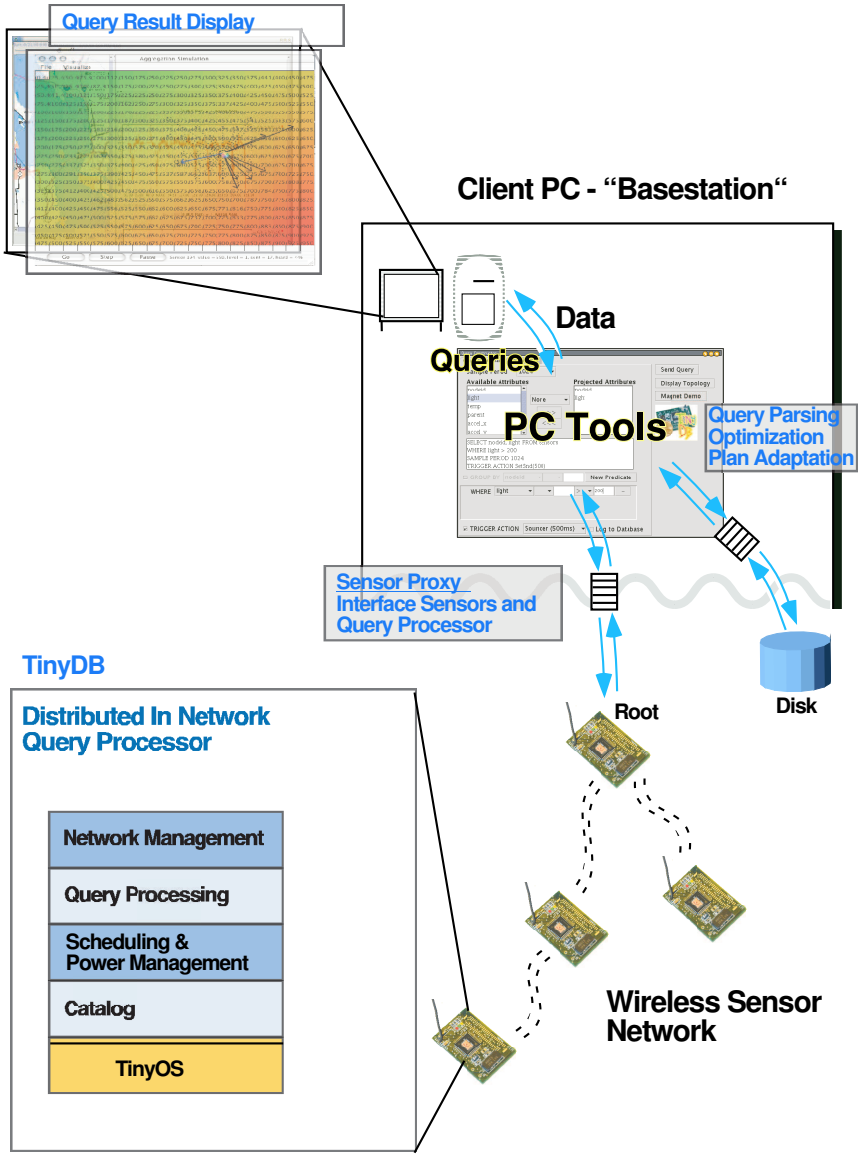


Figure 3.1: Sensor Network Query Processor Architecture

work.

- The bulk of query processing occurs in the **query processing layer**. Data is sampled from the sensors, query processing operators are applied, results from neighboring sensors are combined with local values, and these results are forwarded to the network management layer.

- In the **scheduling and power management layer**, the points in time when the application will be awake and turned on are scheduled with neighboring sensors, so that devices can minimize power consumption while maintaining network connectivity.

- In the **catalog layer**, nodes keep metadata about the access functions and types for the various data attributes they can sense.

We discuss each of these layers in the following sections.

3.4 Query Dissemination and the Network Management Layer

Queries are input at the *root* node, connected to the basestation (typically via a serial UART) and disseminated through the network according to the tree-based flooding protocols described above in Section 2.3.3. As the queries are flooded, nodes assemble into the routing tree to be used for delivery of results back to the basestation. This layer is included as a part of the query processor because the multihop networking API provided by TinyOS is too narrow to support some techniques used in TinyDB to improve query performance and to shape network topologies according to query semantics. [1]

An example of a feature that TinyDB requires of the routing layer is a *promiscuous mode* where any radio traffic heard by the sensor is made available to the application. This mode allows a mote

[1] Because no standard multihop communication API was available when TinyDB was originally built, early versions of TinyDB included their own implementation of the basic multihop protocol as well.

to intercept *non-local* messages from neighboring nodes.[2] Throughout the dissertation, we show numerous ways in which the ability to see the traffic of other nodes can be used to reduce the amount of communication required during processing.

Before being disseminated, queries are translated from SQL into a compact binary representation on the basestation in order to eliminate the need for a SQL parser on each mote and reduce the size of the query messages. Table 3.1 summarizes the contents of this data structure. This binary format is spread across a number of physical messages, one per operator in the query. The query data structure is stored in such a way that motes may receive these query fragments in any order and are insensitive to duplicate fragments; this makes it possible to flood query messages very aggressively when injecting queries, using, for example, gossip-based techniques [HHL02] (see Section 3.4.1 below) and multiple retransmissions to ensure that all nodes receive a query.

3.4.1 Query Sharing

In the spirit of ad-hoc networks, one of the features of TinyDB is the ability to share queries between neighboring nodes (a form of epidemic propagation [VB00].) This provides a way to disseminate queries to nodes that join a network late or miss the initial broadcast. Sharing works as follows: when a sensor hears a query result from some sensor s for a query id that it is not running, it sends a `queryRequest` message to s. s constructs a `queryMessage` (see Figure 3.2) and sends it to the requesting mote.

This kind of epidemic propagation of queries has proven to be extremely useful for code distribution in sensor networks [LS03], as it allows new nodes to be easily added to a network and provides a way to include nodes which (inevitably) miss initial query broadcasts in large sensor networks without expensive re-broadcasts or reliable protocols.

[2] Note that, due to the architecture of the TinyOS radio stack and hardware on current generation motes, there is only a very small penalty for using promiscuous communication: even when not in promiscuous mode, the radio stack completely decodes all messages from the radio and rejects non-local messages in software after they have been completely read off of the air.

3.5 Query Processing

Once a query has been disseminated, it is handed off to the query processor, which is responsible for the bulk of the computation in TinyDB. Query processing consists of two primary phases: preprocessing that happens once upon query arrival, and execution, which happens repeatedly, once per epoch.

3.5.1 Preprocessing

The first phase of query processing is preprocessing, which involves checking the validity of queries and also preparing them for execution. The goal of validation is to check properties of the query that cannot be checked on the server side; in general, due to the limited capabilities of the sensors, we expect that as much validation as possible will happen on the server. However, because the server's connection to a sensor network is not persistent, it is possible that the server may not have knowledge of all of the fields (and their types) or tables (and their schemas) in the sensor network. For this reason, the network must verify that queries are valid.

Preprocessing also sets up the internal data structures and storage needed to execute a query. In TinyDB, this storage includes a tuple data structure to store results that are being constructed and processed, as well as some intermediate storage for operators that need it (like aggregates and joins.) Per-operator storage (e.g., buffers for aggregation and joins) is contained in the *field data* record, and is stored and managed with the query, simplifying query deallocation, migration, and serialization to Flash memory. The query processor passes the appropriate storage into each operator on every invocation.

3.5.2 Query Execution

Once queries have been preprocessed, they are *executed*. Execution consists of four phases: *startup*, *operator application*, *result delivery*, and *power-down*. Execution is initiated at the beginning of

every epoch and begins with the *startup* phase, during which the application activates the sensing hardware needed in the query. On each mote, each sensor needed for a query is turned on, in parallel. Note that the duration of this phase varies depending on the sensors involved in the query (see the "Startup Time" column in Table 2.2) – some sensors, such as the Solar Radiation Sensor [Sol02] take almost a second to start up.

Once all of the sensors have been started, the operators are applied according to the order specified in the query plan (the structure of which we describe in more detail below). The operator application phase is somewhat unusual in TinyDB, as data acquisition is considered an operator, which means that each field of the tuple is acquired through the invocation of a special acquisition operator. If the tuple fails to pass a join or selection operator, it is rejected and the query processor skips straight to the power-down phase. We focus on the behavior of each of the operators in more detail below, in Section 3.5.5.

During the result delivery phase, tuples are enqueued for delivery to the parent node and sent out over the radio. Results are delivered to the mote's parent using a simple queued network interface. Note that each query may produce multiple results in an epoch, if, for example, there are joins or grouped aggregates in the query. Due to limitations in the throughput and queue size of the network interface, the maximum number of results per sensor per epoch is limited to eight tuples on current hardware – the primary limiting factor is available RAM for the message queue. We describe the details of the protocol used for scheduling communication between parent and child nodes in Section 3.6 below.

Finally, once all of the results have been enqueued and delivered, TinyDB enters the power down phase, where sensors are turned off and the power management component is signaled to indicate that processing is complete. The mote then sleeps until the start of the next epoch.

This is the general process followed for all of query processing. We describe the format of tuples

and query plans, as well as the types of operators available in TinyDB in the next three sections.

3.5.3 Tuple Format

Tuples in TinyDB are laid out in a packed byte array. Variable length fields are not supported; strings are fixed at 9 bytes (a one byte length with 8 character bytes.) The type of each field is determined by consulting the Field Data of the query to which it belongs.

Tuple management is complicated by the fact that tuples may contain NULLs (or placeholders for missing or empty records.) In particular, if a sensor receives a query that refers to a field not in its local catalog, it will mark that field as NULL in all results. This allows deployments where the set of available attributes differs between nodes, which we believe will be necessary to support heterogeneous deployments and rolling upgrades of sensor networks. To indicate that a field is or is not NULL, tuples include a `notnull` bitmap (32 bits in the current implementation), in which the i^{th} bit is set if the i^{th} field is non-null.

Figure 3.2 illustrates the format of a tuple and shows a sample layout for a simple two field query (without any NULLs). In this case, both fields are two bytes long.

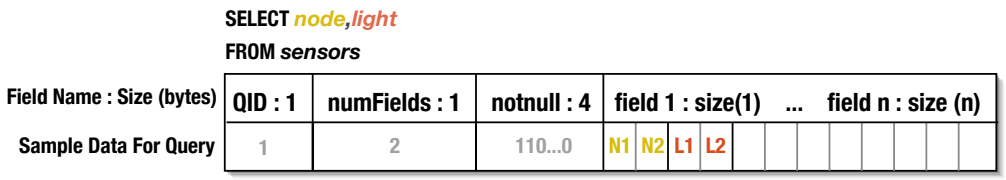


Figure 3.2: Tuple Format and Example Layout

3.5.4 Query Plan Structure

Queries in TinyDB are similar to query plans in a standard database system, except that they include *acquisition* operators which are responsible for retrieving the value of a particular field. These operators impose a constraint on the possible valid plans: a field must be *acquired* before an operator over it can be applied.

Field	Description
Query ID	A unique identifier for the query. The basestation chooses this id by consulting the root of the network about currently running queries. Sensors ignore broadcasts of queries that they are already running.
Root Node	The id of the root node where results will be collected; the basestation typically chooses this id by consulting the root node for its id, although a query could be disseminated from a node that is different from the location where its results will be collected.
Number of Fields	The number of fields in the SELECT list of the query
Number of Expressions	The number of expressions in the WHERE, SELECT, and HAVING lists of the query
Sample Period	The number of milliseconds between samples
Duration	The number of samples for which the query will run, or 0 if the query should run indefinitely
Destination Buffer	Identifier indicating the table into which results should be written (see `SELECT INTO` below)
Triggering Event	The name and condition of the event that triggers this query; NULL if the query should begin running immediately.
Field Data	Variable length data containing names and transformations of fields in the SELECT list as well as the name of the storage point from which the field should be selected, if it is not `sensors`. Names are translated into integer-ids in each device's local catalog after query delivery.
Expression Data	Variable length data containing filters from the WHERE clause, as well as HAVING expressions and aggregates from the SELECT list.

Table 3.1: Fields in a TinyDB Query Message

In the current implementation, plans contain four types of operators: selection operators, aggregate operators, join operators, and sampling (acquisition) operators. These plans are constructed and passed in via the query data structure shown above in Table 3.1. The current version of TinyDB does no in-network optimization of the query plan – instead, we assume the server has sufficient knowledge to optimize the query before injecting it; this optimization process is discussed in more detail in Chapter 5. We revisit in-network query optimization in Chapter 8.

Just as in a traditional DBMS, plans consist of physical operators that implement the logical operations specified in the query. These physical operators may not have a one-to-one mapping to logical operations – for example, in Section 5.3.2, we describe how we can inject extra physical operators into plans to avoid unnecessary work via a technique called *exemplary aggregate push-down.* Furthermore, these physical operators are less declarative than their logical counterparts – for example, the logical aggregate AVERAGE is mapped to a specific implementation of average in

TinyDB: each node reports a record containing a 16 bit sum s and a 16 bit count c such that s/c is the average of the aggregated attribute over the routing subtree rooted at the node. This implementation of AVERAGE must be specified in the plan so that neighboring nodes (and the basestation) know how to coordinate to compute an average over the whole network.

Most of the actual operator implementations in TinyDB are very simple – in general, they do not support complex expressions, user defined functions, or the complete generality of their counterparts in enterprise database systems. For example, selection operators apply to a single attribute and can include (at most) a single mathematical operator. Valid selection predicates include the example expressions `(s.light > 10)` or `(s.temp / 10 = 100)`. Expressions such as `(s.light > s.temp)` or `((s.light/10) + 5) > 100` are not currently allowed. Such limitations are largely due to our desire to minimize the size and complexity of radio messages used to represent expressions; they are not fundamental limitations of our approach.

The general problem of optimization of acquisitional query plans is discussed in Section 5.3. Given this basic structure of query plans in TinyDB, we now describe the individual operators in more detail.

3.5.5 Operators

As described above, TinyDB includes four basic kinds of operators: data acquisition operators, selection operators, aggregation operators, and join operators. We describe each in turn.

Acquisition Operators

Data acquisition operators are responsible for retrieving the value of a specific attribute. Acquisition operators are interesting in two ways: first, they are relatively expensive, because (as was shown in Table 2.2), samples can take a long time to acquire. Second, a field must be acquired before any other operations over that field are applied.

The energy cost of these operators is determined by the time to acquire the sample times the per-unit-time energy usage of the device (usually measured in Watts.) In the current version of TinyDB, acquisition operators are parameterless (i.e., they do not support selection predicate push-down).

Note that we currently rely on the implementation of the low-level code which samples attributes to produce calibrated readings – TinyDB does not apply calibration functions in the processing layer. The reason for this is that calibration is extremely attribute-dependent; some sensors are linear, others very non-linear. Some digital sensors provide calibration coefficients that can be used to compute calibrated values, but the computation can be surprisingly convoluted; for example, the relative humidity voltage output from Sensirion's humidity and temperature sensors [Sen02] is converted to a humidity reading via the following calculation:

$$RH = C_1 + C_2 \times V + C_3 \times V^2$$

Where C_1, C_2, and C_3 can be read from the device via a special protocol. Rather than providing a generic facility for expressing such conversion functions in TinyDB, we embed this knowledge in the code for sampling the attributes themselves.

Selection Operators

Selection operators in TinyDB are straightforward: they apply a simple predicate to every tuple (such as "`light` $>$ `100`"); if the predicate evaluates to true, they allow the tuple to be processed further, otherwise, processing for that tuple stops.

As discussed above, the predicates allowed in selection expressions are currently limited; future versions of TinyDB may include a more sophisticated expression evaluator and allow selection predicates over multiple attributes; such features have not been needed to date.

Aggregation Operators

In TinyDB, aggregation operators are the primary way in which readings from several sensors are combined together. The reason aggregates are so central is that, because of the limited communications bandwidth and potentially large number of readings, users may want to (or be required to) look at summaries of data from logical groups of sensors, a task for which aggregate queries are well suited.

Aggregate queries in TinyDB are fully implemented, with a few limitations. As with selections, predicates are limited to single `<attr> <op> <const>` triplets. Only a single `GROUP BY` attribute may be specified. Queries may specify aggregates over an arbitrary number of attributes, however, and TinyDB includes an extensible framework that allows developers to extend the system with new aggregate functions. We describe the aggregation features of TinyDB in more detail in Chapter 4.

Join Operators

Equality joins are allowed between a single attribute from the `sensors` table and a local `STORAGE POINT`. In the current implementation, this is executed as a nested loops join, where a tuple from the (outer) `sensors` relation is used to scan the (inner) `STORAGE POINT`. Note that an equality-join must also be specified on the `nodeid` attribute, so that joins are always executed locally, without shipping data to any other node.

3.5.6 Table and Storage Management

As presented in Section 3.2 above, results in TinyDB can be stored in `STORAGE POINTS` in the Flash memory on the local device. In the current implementation, such storage points are partitioned according to `nodeid`, such that each device stores and queries only the readings it produced.

In-network storage in TinyDB serves an important role as it enables data collection rates that ex-

ceed available network bandwidth. The EEPROM component in TinyOS can write about 8 KBytes / second, whereas the radio, as discussed in Chapter 2, is limited to about 20 48-byte packets per second in aggregate, across all of the devices in a radio cell. Logged results can then be delivered offline, at a slower data rate.

Note that with such a high data rate, logging queries will eventually exhaust available storage. At 100 samples per second, a 4Mbit Flash will fill after

$4 * 10^6$ bits / (10 bits per sample * 100 samples per second) = 4,000 seconds

or just over an hour worth of samples of a single attribute. Thus, such high-data rate queries are typically run for short periods of time to monitor some phenomena of interest and then are stopped so that the logged results can be delivered.

Note that two queries, one that logs results to a table in Flash and another that aggregates results enqueued into this table, may run simultaneously in TinyDB. This allows users to see real-time summaries (e.g. sliding window averages with minima and maxima) of results being logged at high data rates.

This completes our discussion of the basic query processing algorithms, the associated tuple and plan data structures, and the specifics of TinyDB operators. We now turn to query processing issues that are unique to sensor networks, beginning with a discussion of communication scheduling.

3.6 Power Management via Communication Scheduling

A key element of power conservation in TinyDB is synchronized, scheduled communication. Communication scheduling refers to placing sensors in a low-power state between rounds of communication. Without a schedule, nodes would have to be listening to their radios at all times, since they would never know when to expect to hear a reading from a nearby sensor. Such an approach would be very wasteful of energy.

3.6.1 Scheduling Protocols

Two simple scheduling protocols used for result collection in TinyDB are described here. Both work by aligning the times when parents and children are receiving and sending (respectively) in the tree-based communication protocol sketched in Section 2.3.3. For the purposes of this discussion, each node is assumed to produce exactly one result per epoch, which must be forwarded all the way to the basestation. When we discuss in-network processing in the next chapter, we study cases where less data than this is actually transmitted out of the network.

In the *basic approach*, each node in the network wakes and sleeps at exactly the same time for a fixed portion of every epoch. It is during this waking interval that nodes perform all of their computation and communication, and during which nodes collect and forward results from their children in the routing tree.

In the *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. Nodes wake up at the start of their interval, perform their computation and communication, and go back to sleep. 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.) In this way, information travels up the tree in a staggered fashion, eventually reaching the root of the network during interval 1. Note that, in this approach, parents are not required to explicitly maintain a list of children, and children can learn when a parent will be listening simply by learning the parent node's level in the routing tree.

Notice that in both of these approaches, the nodes must exchange clock information that tells them when each epoch begins and ends. Because epochs are of a fixed length, it is sufficient if nodes agree on the time at which the first epoch began and the current time (i.e., their clocks are

synchronized). The start time of the first epoch is carried with the query as it is transmitted; we describe our approach for clock synchronization in the next section.

Figure 3.3 illustrates these two approaches for a particular network topology. Arrows represent messages being transmitted; in this case, every message is forwarded to the root of the network. Notice that in both approaches, a significant percentage of nodes' time is spent sleeping, though in the slotted approach different nodes are asleep at different points in the epoch.

Looking at Figure 3.3, it is not immediately obvious whether one approach is better than the other. The slotted approach, however, turns out to be preferable for two reasons. First, because there is less communication in each slot, there is less potential for radio collisions. Notice that it is still possible for collisions to occur with this approach: for example, nodes G and F can hear each other but also transmit during the same interval. Therefore, we rely on the CSMA media access protocol (see [WC01]) to avoid collisions in *both* approaches. However, the reduced potential for collisions in the slotted approach means that we can make the length of each node's waking interval much shorter.

The second reason that the slotted approach is preferable is that parents see results from all of their children before they communicate any data to their own parent. This allows parent nodes to possibly filter readings from children or combine them with their own readings, which will prove to be extremely important in the next chapter when we examine techniques for pushing query processing into the network.

Both of these approaches depend on number of configuration parameters to run properly. In the basic approach, the duration of the waking interval must be long enough that nodes can collect and forward all of their data. In the slotted approach, this same problem occurs, but there must also be enough intervals for all of the nodes to participate. To solve this latter problem, we simply disallow nodes from joining the network if their depth is greater than the number of intervals.

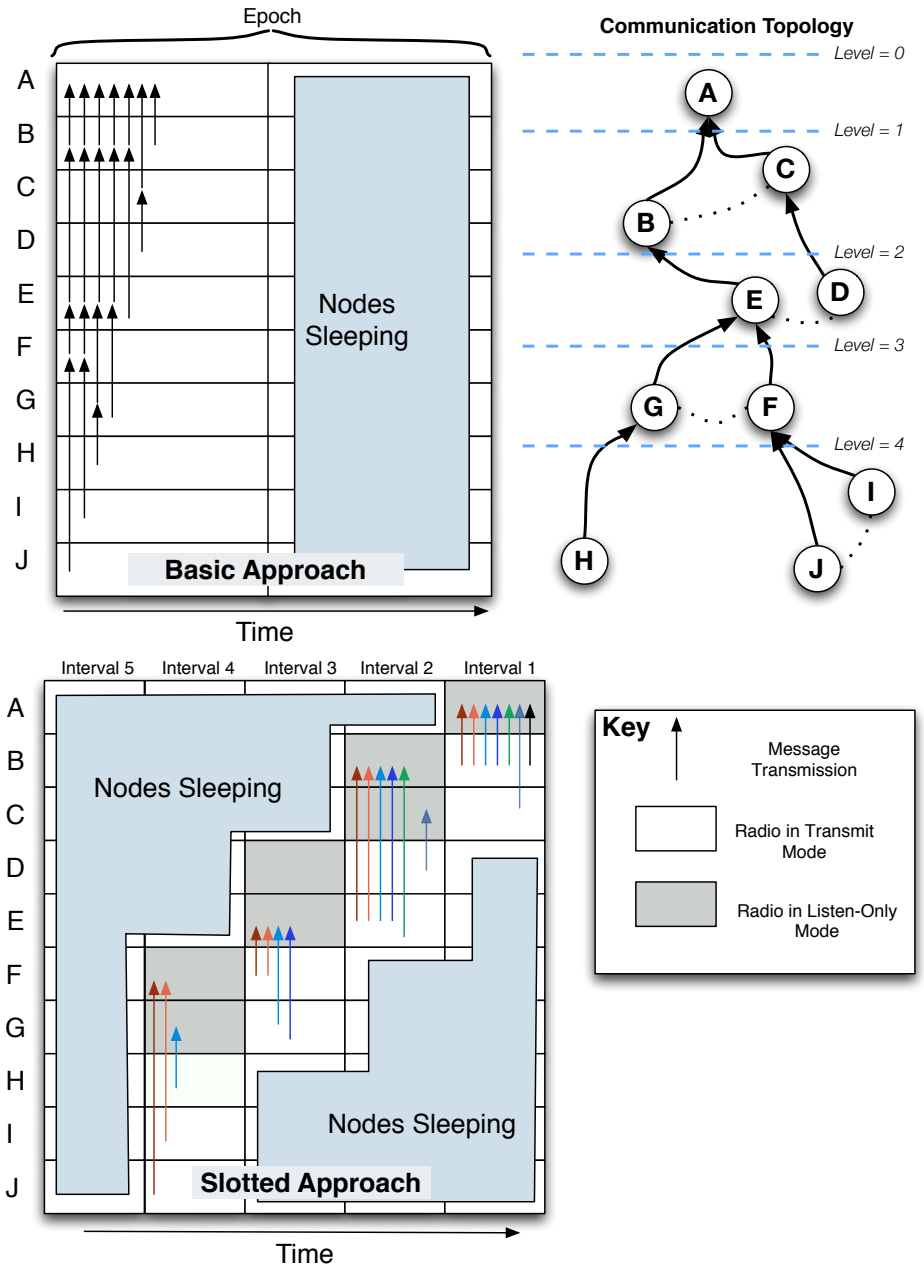


Figure 3.3: Two variants of scheduled communication for TinyDB

The correct setting for the waking-interval duration (in either approach) depends on the kinds of sensors involved in the query (since it takes differing amounts of time to acquire samples from different sensors) as well as the network density. The correct number of intervals in the slotted approach depends on the *size* of (or number of hops across) the network. In the current TinyDB implementation, these constants are configuration parameters that are set as the network is deployed. Setting them typically involves trying a few different values; bad choices are immediately obvious as no results will be produced if the waking interval is too short, or certain sensors will never report values if there are too few intervals. Because choosing correct settings for these parameters increases the complexity of deploying a network, TinyDB eventually will need to be able to automatically learn proper settings of these values.

Note that this scheduling is not done for queries where the sample period is too short to allow the node to sleep or when there are multiple queries with different sample periods. In these cases, TinyDB switches to an always-on radio mode.

Finally, note that both approaches support bidirectional communication by having parents transmit during the waking interval of their children. In the slotted approach, this means that periods when parents are normally only listening will sometimes be used for transmission as well.

3.6.2 Time Synchronization

To implement either of the approaches described above, we require that the nodes synchronize their clocks to agree on the timing of sleep/wake intervals. Motes synchronize their clocks by listening for a clock reference that is periodically broadcast from the root of the network. This broadcast contains the current clock reading (in milliseconds since power on) at the basestation. Nodes at level 1 synchronize their clocks and retransmit this message, including their own time stamp (to account for propagation delay from the reception of the root's message until its re-transmission several milliseconds later.) To ensure that timestamps are as accurate as possible, timestamps are

written into these packets at the lowest level of the radio stack, after they reach the head of the radio queue and the MAC protocol indicates that the channel is free – just before they are sent out over the air.

To demonstrate the effectiveness of time synchronization and scheduled communication, we logged a trace of 2,500 messages collected by 10 sensors running TinyDB in a multi-hop routing topology, using the basic scheduling approach described above. Motes ran a simple selection query that produced one result every 15.3 seconds, with the waking period set to 3.1 seconds. Each packet was judged as synchronized if it was logged at the server within the waking period. We disabled power management on the root node so that it could hear packets transmitted during the sleep period. The start of the query and the beginning of every epoch was determined by the root node. We found that only 5 of the 2,500 packets were not synchronized; in all of these cases, synchronization was off by less than a second and was corrected by the next epoch. Though we were able to log these non-synchronized packets, had the root node not been listening all of the time, they would have been lost.

Figure 3.4 shows a short trace from this data set. Different motes are along the Y axis (numbered 1-13, with 3,7, and 12 missing), and time of day is along the X axis. Dark vertical lines indicate the ends of epochs (every 15.3 seconds) and lighter lines indicate the intervals (3.1 seconds) between epochs; in this case all of the sensors were asked to transmit within the first interval. Plotted points indicate the time when a message arrived at the basestation from a particular mote; data was not received from all motes on every epoch. Note that in the case shown, all of the results arrive within their allotted time slot. Small differences in the arrival times of packets are due to the fact that only one mote may deliver a packet to the root at a time.

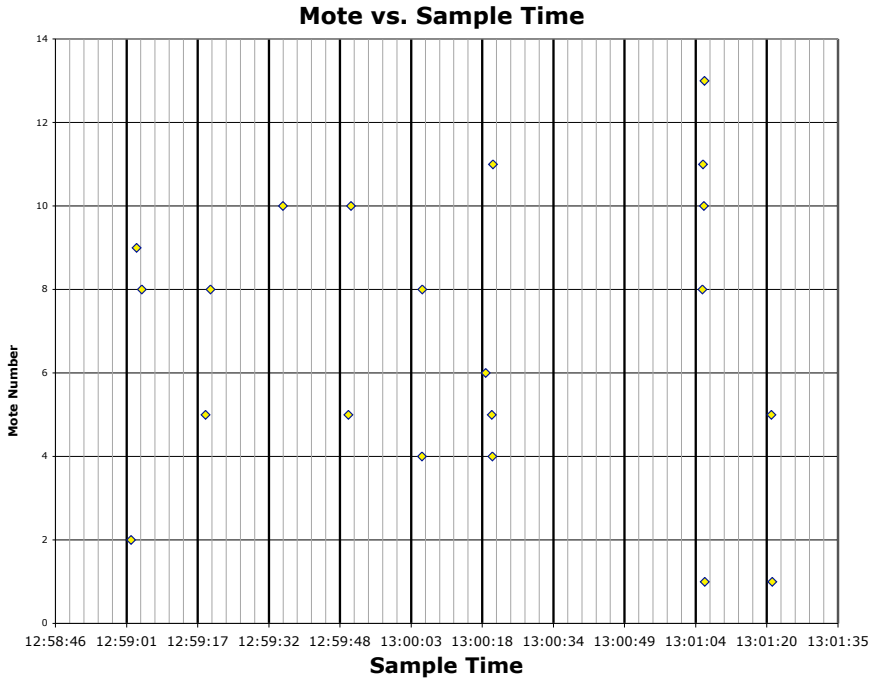


Figure 3.4: Plot showing the time synchronization between sensors during a 3 minute period.

3.6.3 Benefit of Scheduling

The primary benefit of scheduled communication is a reduction in power utilization and commensurate increase in network lifetime. In this section, we measure the power consumption of sensors running TinyDB and use those measurements to derive expected lifetimes for a network of sensors.

Figure 3.5 shows the power utilization (in mA) for a Mica2Dot mote running a selection query over light, temperature, humidity, and air-pressure sensors with a 15s sample period, with a 3s awake-interval during a 50s time window. In the active state, the power utilization is about 12mW, whereas in the sleep state, it is approximately 200 μA. Figure 3.6 shows the same sensor, but during a shorter, 1s interval when the device is running at full power. In both traces, the devices have their radios on during the entire waking period, and transmission can be seen as very short (30ms) spikes in the trace in Figure 3.6 as the radio stops sampling the channel and begins receiving bits. Given the relative brevity of these receive spikes, we simply approximate the current consumption in the

active state to be 12 mW in the following calculations.

For a pair of AA batteries with 3000 mAh (at 3V) of capacity, our network would run for 3000/12 = 250 hours, or about 10 days with no communication scheduling (i.e., with radios at full duty cycle.) Using communication scheduling and transmitting a sample per minute (a 5% duty cycle), we can extend this lifetime to 3000/(.05 * 12 + .95 * .200) = 3000/.79 = 3797 hours, or about 160 days. With a sample every 5 minutes (a 1% duty cycle), this lifetime will increase to about 400 days.

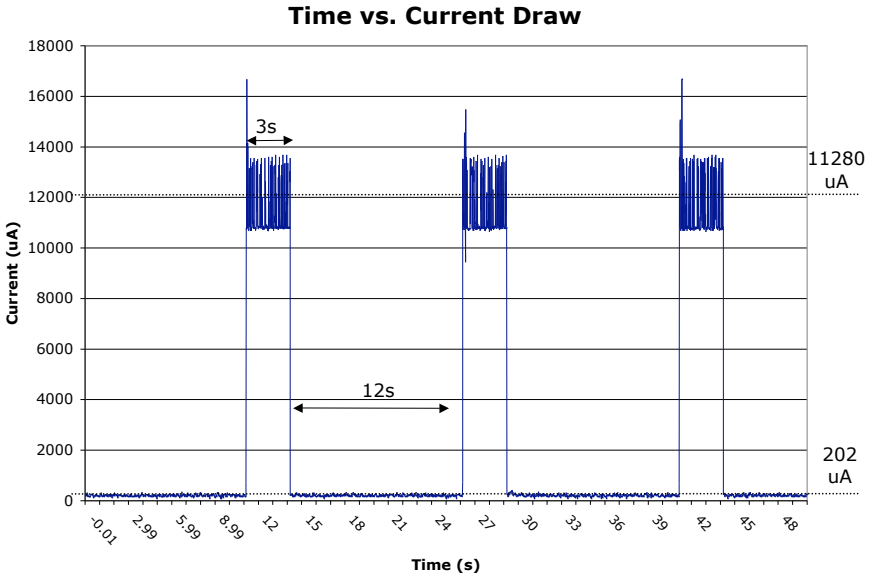


Figure 3.5: Plot showing current consumption on a sensor with a 3s waking period and 15s epoch over a 50s window.

3.6.4 Alternative Scheduling Approaches

There are alternatives to our schedule-based power management that could be used – for example, using a low-power listening mode as proposed in [Pol03b] or using a second, very low power radio to detect radio traffic as proposed in [RAK+02b]. These techniques, however, require all nodes to wake

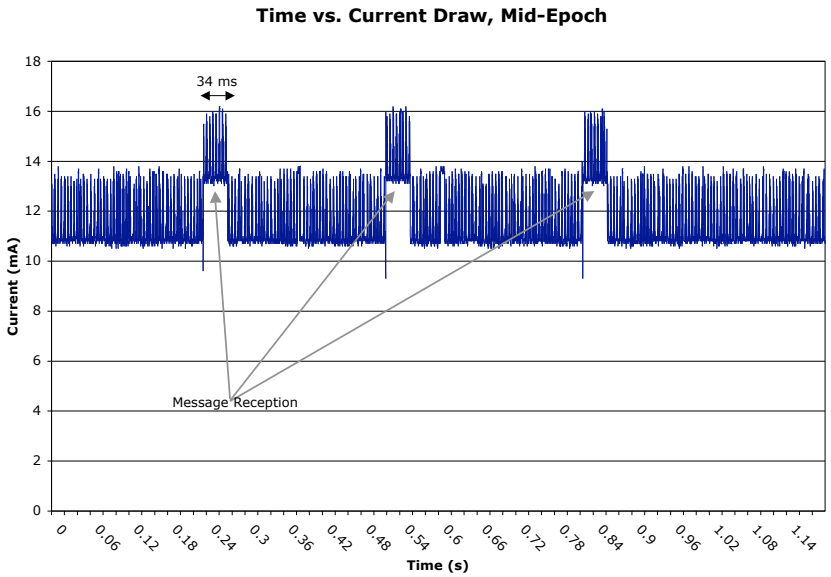


Figure 3.6: Current consumption on same sensor as in Figure 3.5, focused in on a 1s interval during the active state.

up whenever any communication occurs (whether or not it is addressed to the local node), unlike our scheduled communication scheme which exploits the structured nature of the query processing algorithms and topology to limit the nodes that are listening at any point in time.

Low-power listening works by having the sending radio transmit a very long preamble at the beginning of every packet. Receivers come out of sleep periodically and sample the radio; if they detect a preamble being sent, they fully receive and process the packet. Receivers must wake up at least once every half-preamble time, so that they are guaranteed to hear the preamble no matter the phase alignment between the time when transmission starts and the receiver next samples the channel. Thus, longer preambles increase the transmission cost for senders, but allow receivers to sleep longer between radio samples. A major disadvantage of low-power listening is that any receiver in range of a packet must wake up and receive the entire packet to determine whether it is destined for the local node, which means that it cannot be used effectively in high-data rate

environments.

Even in a low-data rate environment, however, scheduled communication is usually more effective than low-power listening. For example, the low-power-listening-based approach recently deployed on Great Duck Island was estimated to able to last about two months on an 850mAh battery with a 20 minute sample period [Pol03a]. On the same battery and at the same data rate either of our communications scheduling approach should last more than sixteen months.

Other techniques for building communication schedules are also possible; for example, the S-MAC protocol [YHE02] from UCLA is a TDMA-like approach where nodes explicitly exchange schedules with all of their neighbors. Because multiple senders may attempt to contact the same receiver, the channel is negotiated via an RTS/CTS protocol where the request to send is implicitly granted to the first requester.[3]

The S-MAC approach provides power conservation benefits (once schedules have been exchanged) comparable to the benefit of the basic scheduled approach described above. The problem with the S-MAC approach is that it requires nodes to exchange and store schedules of all of their neighbors, a process that can be prohibitively expensive for large networks with lots of nodes coming and going. Our approach avoids this exchange by implicitly deriving a schedule based on the depth of the sensors in the tree.

3.7 Catalog Layer

The catalog layer is responsible for storing named query *attributes* as well as named *commands* and *events*. The catalog provides a unified interface to a number of TinyOS components that sit below it; for example, in TinyDB, queries can fetch the `light` attribute, which is provided by the `Photo.nc` component, as well as a count of the nodes in the mote's one-hop network neighborhood

[3] This approach is roughly comparable to the TinyOS CSMA MAC protocol where the channel is implicitly granted to the first mote that begins sending.

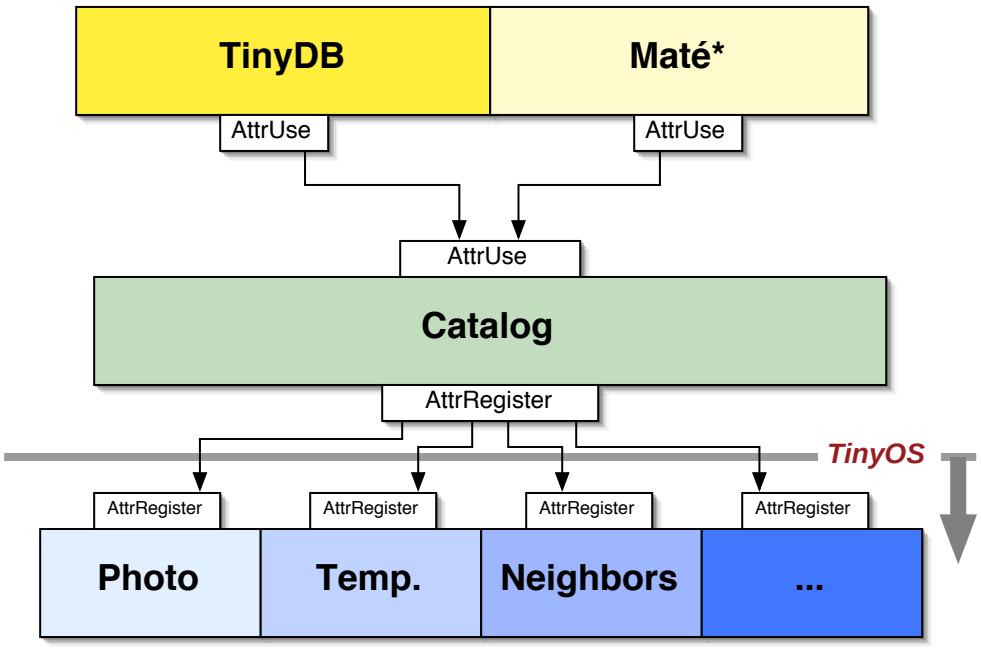


Figure 3.7: The narrow catalog API provides a uniform application interface for accessing sensor readings and internal state of low-level TinyOS components.

from the `neighbors` attribute, which comes from the `NetworkC.nc` component. Rather than requiring TinyDB to interface directly with every low-level component it wishes to query, these components implement the appropriate interface and register themselves with the catalog; TinyDB then fetches the values of these attributes through the catalog command **getAttr**`(...)`.

Figure 3.7 illustrates the software architecture of the catalog – the narrow catalog interface sits between applications like TinyDB and Maté [LC02] and allows them access to a number of attributes registered by lower level components.

The catalog provides a similar mechanism for registering commands which can be used to take some action in response to a query (for example, sounding the buzzer on the sensors when the temperature goes above some threshold, as in Query 3). Commands referenced in queries are invoked through the catalog, and low-level components make their externally accessible commands

available through a well-defined catalog interface.

Similarly, the catalog contains information about named events that are used to initiate a query in response to some asynchronous event – for example, the user pressing a button on the mote. We will discuss the semantics of events in queries in more detail and describe additional pieces of information stored in the catalog in Chapter 5.

3.8 The Role of TinyOS

In Figure 3.1, TinyOS sits at the bottom of the mote-side architecture, and it provides a number of features essential to TinyDB. This section summarizes some of the ways in which the features of TinyOS are used in TinyDB and discusses areas where the TinyOS interfaces proved inadequate or lacking in TinyDB.

The foremost role of TinyOS is as an abstraction for the sensors, radio, and timers with a higher-level interface than direct interaction with the I/O ports on the processor (for more details about these abstractions, see Section 2.5 above.) Unfortunately, as mentioned in the discussion of the TinyDB routing layer and the catalog, these abstractions are often not at the correct level for TinyDB, and thus are partially replicated or replaced at the application level. For example, the software components provided by TinyOS for sensor readings do not export a uniform interface and do not each have a unique name that can be used to reference the attribute in an external application. The TinyDB catalog, therefore, must provide these features itself.

In other instances, TinyDB must provide services that are lacking from the OS. Two instances of this are scheduled communication (and the associated time synchronization protocol) and power management. TinyDB explicitly schedules its communication, using the interval based approaches described above for deciding when to enqueue radio messages. Though the OS could provide a scheduling mechanism, TinyDB is able to exploit the semantics of queries – that is, knowledge

about the sample period – to set up a very regular schedule in a simple way. Similarly, one might expect the OS to provide the time synchronization protocol needed for this scheduling; however, many months of engineering time by the TinyOS team were spent trying to develop a low-level instance of such a protocol that never worked properly. TinyDB's synchronization works because it uses application semantics, specifically knowledge about when the application can safely (without causing incorrect behavior) adjust the firing time and rate of the low-level hardware timers used to schedule periodic operations.

Power management is another feature which is largely implemented in the application, though the OS does provide a mechanism for putting the device into a low-power state. Again, the issue here is that the low-level OS lacks knowledge of *when* such power cycling can safely occur – the application may intend to perform some operation at just the instant the OS decides to enter a low-power state for several seconds. The periodic nature of TinyDB queries, however, makes this very easy to predict in the database system, and thus requires that this functionality be pushed up into the application.

3.9 Basestation Processing

Thus far, we have discussed the design and features of the TinyDB query processor, including the structure of tuples and query plans, the basics of query processing and the behavior of operators. This section briefly discusses the basestation's (e.g. PC's) processing of queries. The PC is responsible for accepting, parsing, validating and optimizing queries before disseminating them into the network. It is also, in general, responsible for the collection and display of results. We rely on the basestation to input and parse queries largely because input peripherals are not readily available for motes. Parsing is a relatively memory-intensive step that is just as well done outside of the network.

Validation is done so that the amount of error handling code within the network can be greatly

reduced. By ensuring that a query is valid before it is input into the system, we reduce the size of the code on the motes and eliminate the need for a large set of error messages to be stored in and sent from the sensors. In general, validation consists of checking properties such as:

- **SQL Conformity:** This involves verification that the query is valid SQL – e.g. that it contains a `SELECT` statement, there are no aggregates over fields in the `GROUP BY` clause, and so on.
- **Validity of Sample Rates:** The PC tries to check that the system could reasonably expect to provide data at the requested rate (given the selected attributes and query complexity.)
- **Compatibility of Field Types and Names:** The PC verifies that the types of fields and operators over them are compatible. Various numerical and boolean operations over fields and constants or pairs of fields have constraints on the types they can accept. Fields referenced in the query must also be in the server's catalog.

After a query has been validated, it is optimized. We defer a detailed discussion of query optimization until Chapter 5; the basic idea is to select a query plan that minimizes the total energy cost of query execution.

3.9.1 Result Display and Visualization

In addition to parsing and verifying queries, the TinyDB basestation provides a number of tools for collecting and visualizing results outside of the sensor network. As a first step, the system automatically stores results in a standard relational database for later visualization and processing. Results are stored in a standard, append-only relational table with one column for every attribute in the TinyDB query and one row for every result.

TinyDB also comes with several interfaces to visualize and interact with sensor data, a few of which we briefly summarize here because they give an example of the kinds of applications we

envision being built on top of TinyDB and because the availability of tools for data display and visualization is an important part of making systems like TinyDB easy to deploy and use.

Basic Graph Visualization

TinyDB provides a simple, line-plot-based visualization of a single attribute versus time. This visualization is provided in real-time, as results arrive in the system. Figure 3.8 shows an example of this display.

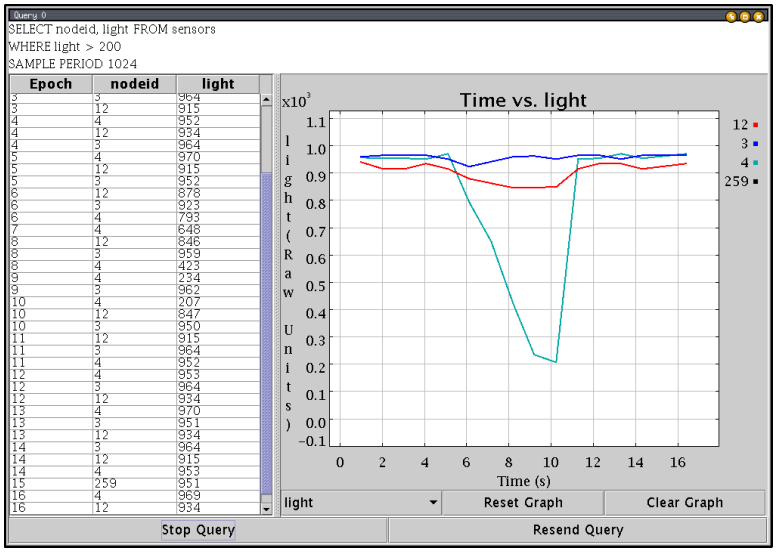


Figure 3.8: Graph-based visualization of results from a simple TinyDB query, showing light values vs. time for three sensors.

This simple visualization works reasonably for many different kinds of queries. However, it is also possible to use TinyDB to drive a number of application-specific displays, several of which are described in the following two sections.

Topology Display

Figure 3.9 shows a simple, application-specific display for visualizing the a sensor-network tree-based communication topology. This visualization was authored as a TinyOS demonstration application; we adapted it for TinyDB by removing the application-specific mote software and replacing it with TinyDB running a simple selection query. This illustrates the advantages of the *retasking*

features of our approach: using TinyDB, a number of different applications can be built very easily on top of a sensor network deployment.

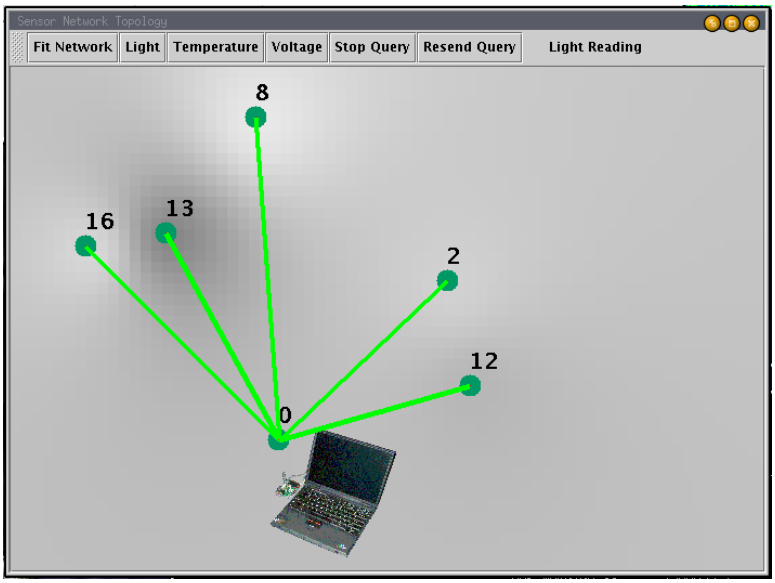


Figure 3.9: Visualization of the sensor network topology, with light values displayed in the background gradient. This visualization is generated by running a simple selection query in TinyDB and feeding the results into a rendering module.

Traffic Sensor Visualization

Next, as an illustration of an application that TinyDB will be used for in the future (but is not yet), we describe a prototype visualization over traffic sensor data. The sensor data is currently collected by the California Highway Patrol (CHP) and posted on a web site. The current system depends on expensive, PC class hardware with a cellular connection to manage each data-collection point on the freeway, of which there are thousands in the Bay Area. These collection points record samples at about 60Hz from the sensors, then report aggregate average speeds and inter-arrival times over the cellular gateway. An ad-hoc, smart-sensor network should be able to provide this same data collection facility without expensive PCs attached to every sensing device.

Figure 3.10 shows a screenshot of the prototype visualization. It includes a number of features: the map on the left displays the locations of sensors (as small, colored squares.) It also includes

a display overlay for roads, a set of camera icons representing known locations of web-cams displaying road conditions, and a set of recently reported traffic incidents from the CHP web site. All of this data is collected and updated in real-time. The image in the upper right is an example of a web-cam shot from downtown San Francisco. The graph in the lower right shows a few hours worth of vehicle speed data at a single sensor on I-80 near Berkeley.

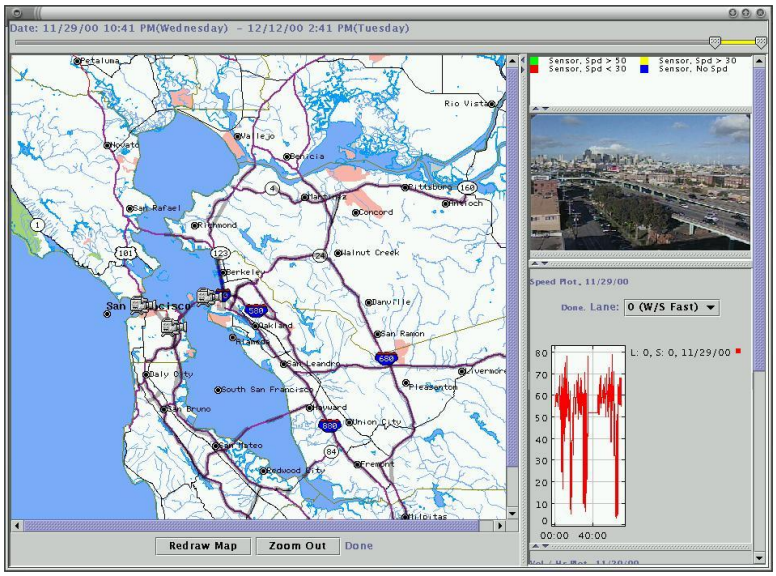


Figure 3.10: Prototype Visualization of Bay Area Traffic Sensor Data

In this section, we demonstrated three visualizations that have been or could be built on top of TinyDB. These examples illustrate the range of applications of a declarative query processor and show that the declarative interface we provide is flexible enough to provide the data substrate for some significant and useful applications. This completes our discussion of the features of TinyDB.

3.10 Comparison to Other TinyOS Applications

We look now at ways in which this architecture is better suited to the challenges of data collection in sensor networks than existing TinyOS solutions. We begin with a discussion of the existing TinyOS applications for monitoring and tracking – in particular, the custom software from Great Duck Island and the NEST vehicle tracking demo discussed in Section 2.1.

Functionality similar to both of these applications was implemented in TinyDB using a very simple set of queries. The "programming" of the queries for each of these deployments took only a few minutes. [4] In the case of GDI, these were simple selection queries over several attributes (for example, light, humidity, air pressure, etc.); in the case of vehicle tracking, the query was a simple aggregate:

```
SELECT ONEMAX(mag, nodeid)
 FROM sensor WHERE MAG > thresh
 SAMPLE PERIOD 64
```

The `ONEMAX` aggregate, in this case, returns the `nodeid` of a sensor that has the maximum magnetometer reading since it is possible, in principle, for many sensors to all have the maximum reading – in traditional SQL, this query would need to be computed through a nested query (assuming at least one sensor has a reading above the activation threshold `thresh`.) This is essentially the same as the behavior of the vehicle tracking application initially implemented for the NEST project – the sensor with the largest magnetometer reading repeatedly forwarded its value out of the network. [5]

One obvious way in which the declarative approach wins over either the vehicle tracking or GDI applications is that it naturally supports both classes of application with a very simple set of queries. The same basestation Java tools that allow queries to be injected and results to be retrieved and displayed can be used in both situations. The TinyDB API supports a number of features – changing data rates and queries on the fly, adding new nodes to the network without having to reconfigure the system, and so on, that are not available in either of the standalone implementations.

We compare TinyDB to the GDI software in terms of energy efficiency, reliability, and other metrics in Chapter 7, so omit we a detailed comparison of those features here. In short, however, even for a simple, selection-only workload, our architecture provides significant advantages over the current software in GDI. In particular, it supports power-efficient, multi-hop communica-

[4] Substantial development time went into developing TinyDB, but that effort produced a fully reusable platform whose development costs will be amortized over the lifetime of the software.

[5] Note that the NEST demo used geographic routing [KK00] rather than tree-based routing as in TinyDB.

tion, the ability to re-task the network by requesting different subsets of the data fields, and time-synchronized readings that are time-stamped within the network.

3.11 Revisiting the TinyDB Requirements

Having described the basic architecture of TinyDB, we now revisit the requirements from Section 3.1, considering how our architecture addresses each of them and mentioning more sophisticated techniques proposed in the next two chapters:

- *Power Management*: Scheduled communication features are at the heart of power management in TinyDB. By turning off the processor and radio for all but a few seconds in every epoch, long network lifetimes can be achieved, which is extremely important for remote monitoring deployments in hard-to-reach locations.

  Throughout the following chapters, a number of additional techniques aimed at power management are discussed. In particular, in Chapter 4, several semantically driven techniques are proposed for avoiding unnecessary processing in aggregate queries. Then, Chapter 5 discusses how TinyDB is sensitive to the costs of acquiring sensor data, and how that sensitivity can be used to improve its performance.

- *Communication Reduction*: The design of TinyDB promotes network efficiency in a number of other ways: first, the continuous nature of queries allows users to request a large number of results in a stream instead of repeatedly broadcasting expensive requests for current sensor values throughout the network. Second, as discussed in Chapter 4, TinyDB provides cardinality-reducing query processing operators that can be employed deep within the network to avoiding unnecessary communication. Finally, the query optimizer is capable of generating plans and using optimizations which minimize communication, something that is lacking in non-declarative approaches.

- *Network Dynamism*: Adaptivity is the key to dealing with the dynamic nature of sensor networks. The most basic form of adaptivity – at the network layer – allows sensors to adapt to the appearance or disappearance of neighboring sensors and to account for transient interference effects that alter the connectivity and link quality.

  The applicability of other forms of adaptivity – as it relates to operator placement, for example – are also directly related to our choice of a declarative architecture and help to deal with the dynamicity of sensor networks. Because the user is concerned only with obtaining the data he or she needs, the system is free to alter the query plan or physical placement of operators in that plan as it sees fit. Adaptivity can be used in a number of other attractive ways: to avoid and provide redundancy against failed nodes, to take advantage of nodes with especially good connectivity, or to focus computation on nodes with excess power or bandwidth. We return to these issues in Chapter 8.

- *Reduction, Logging, and Auditing*: The declarative architecture of TinyDB is flexible enough to provide all three of these features. Multiple simultaneous queries are fully supported, some of which can summarize and rapidly report results, while others can log to storage points and satisfy audit demands by reliably delivering those logs offline (i.e., not in real time.)

- *Management and Ease-of-Use*: Finally, the declarative interface, combined with the PC-side front end, provides for a vastly improved user experience than the interface a system like TinyOS provides for sensor networks. For programmers, this means they can focus on extending the system through the catalog interface and avoid the difficulties associated with managing the details of data collection, power management, and multihop routing. For end-users it means they have the flexibility and reconfigurability offered by a SQL-like interface. This will allow them to explore and collect data much more easily than with other data-collection ap-

plications for sensor networks, which provide only the ability to fetch fixed-schema samples at pre-defined intervals.

3.12 Summary

In this chapter, the main requirements of query processing in sensor networks were summarized and the basic architecture of the TinyDB query processing system was presented as a way to address these challenges.

The primary benefits of the query processing approach are easy deployment and simple, expressive reconfiguration via declarative queries, combined with a power-efficient execution and result collection framework. In the next chapter, we see how to extend this framework with a notion of *in-network processing*, which yields substantial reductions in the communication burden placed upon the sensor network.

Over the course of the next two chapters, this framework is also enhanced to include a number of sensor network specific query optimization techniques that can be applied automatically by the system. These techniques further improve the efficiency of our basic approach while maintaining the benefits described above.

Chapter 4

In-Network Processing of Aggregate Queries

In this chapter, we focus on the procesing of aggregation queries. TinyDB processes such queries using an in-network service called Tiny AGgregation (TAG). The primary contribution of this chapter is the description of a framework for efficient execution of aggregate queries in networks of low-power, wireless sensors. We also discuss various generic properties of aggregates, and show how those properties affect the performance of our in-network approach. We include a performance study demonstrating the advantages of our approach over alternative, centralized, out-of-network methods, and discuss a variety of optimizations for improving the performance and fault-tolerance of the basic solution.

4.1 Introduction

As we discussed in the Chapter 2, many sensor applications depend on the ability to extract data from the network. Often, this data consists of summaries (or aggregations) rather than raw sensor readings. Although other researchers have noted the importance of data aggregation in sensor networks [IGE00, HSI+01, IEGH01], this previous work has tended to view aggregation as an application-specific mechanism to be programmed into the devices on an as-needed basis, typically in error-prone, low-level languages like C. In contrast, our position is that because aggregation is

so central to emerging sensor network applications, it must be provided as a *core service* by the system software. Instead of a set of extensible C APIs, we believe this service should consist of a generic, easily invoked high-level programming abstraction integrated with a data-collection system like TinyDB. This approach enables users of sensor networks, who often are not networking experts or even computer scientists, to focus on their applications free from the idiosyncrasies of the underlying embedded OS and hardware.

4.1.1 The TAG Approach

Following this philosophy, we have developed Tiny AGgregation (TAG), a generic aggregation service for sensor networks, which is used in TinyDB. There are two essential attributes of this service. First, it extends TinyDB's simple, declarative interface for data collection with aggregation capabilities inspired by aggregation facilities in database query languages. Second, it intelligently distributes and executes aggregation queries in the sensor network in a time and power-efficient manner, and is sensitive to the resource constraints and lossy communication properties of wireless sensor networks. TAG processes aggregates *in the network* by computing over the data as it flows through the sensors, discarding irrelevant data and combining relevant readings into more compact records when possible.

The basic approach of query processing in TAG is as follows: as data from an aggregation query flows up the routing tree, it is aggregated in-network according to the aggregation function and value-based partitioning specified in the query. As an example, consider a query that counts the number of nodes in a network of indeterminate size. First, the request to count is injected into the network. Then, each leaf node in the tree reports a count of 1 to their parent; interior nodes sum the count of their children, add 1 to it, and report that value to their parent. Counts propagate up the tree in this manner, and flow out at the root.

4.1.2 Overview of the Chapter

The contributions of this chapter are four-fold: first, we describe in the details of the aggregation features of the TinyDB Query language sketched in Chapter 3. As we do this, we will identify key properties of aggregation functions that affect the extent to which they can be efficiently processed inside the network. Second, we demonstrate how such in-network execution can yield an order of magnitude reduction in communication compared to centralized approaches. Third, we show a number of aggregate-specific optimizations that, by virtue of the declarative interface provided by TinyDB, can be transparently applied to further reduce the data demands on the system. Finally, we show that our focus on a high-level language leads to useful end-to-end techniques for reducing the effects of network loss on aggregate results.

4.2 Aggregate Query Model and Environment

We introduced the basic semantics of queries (including aggregate queries) in Section 3.2 above. In this section, we specify the semantics of aggregate queries more completely, extending the previous query definition with a notion of logical *groups* and the ability to filter out certain groups from the query result. We begin with an example query.

4.2.1 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 it is deregistered.

In general then, aggregate queries have the form:

```
SELECT {agg(expr), attrs} FROM sensors
 WHERE {selPreds}
 GROUP BY {attrs}
 HAVING {havingPreds}
 SAMPLE PERIOD i
```

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. The `SELECT` clause specifies an arbitrary arithmetic expression over one or more aggregation attributes. We expect that the common case here is that $expr$ will simply be the name of a single attribute. `Attrs` (optionally) selects the attributes by which the sensor readings are partitioned; these can be any subset of `attrs` that appear in the `GROUP BY` clause. The syntax of the agg clause is discussed below; note that multiple aggreggates may be computed in a single query. The `WHERE` clause filters out individual sensor readings before they are aggregated. Such predicates are executed locally, within the network, before readings are communicated, as in [PJP01, MF02]. The `GROUP BY` clause specifies an attribute-based partitioning of sensor readings. Logically, each reading belongs to exactly one group, and the evaluation of the query results in a table of group identifiers and aggregate values. The `HAVING` clause filters that table by suppressing groups that do not satisfy the `havingPreds` predicates.

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[1] and the readings used to compute an aggregate record all belong to the same the same epoch.

4.2.2 Structure of Aggregates

The problem of computing aggregate queries in large clusters of nodes has been addressed in the context of shared-nothing parallel query processing environments [SN95]. Like sensor networks, those environments require the coordination of a large number of nodes to process aggregations. Thus, while the severe bandwidth limitations, lossy communications, and variable topology of sensor networks mean that the specific implementation techniques used in the two environments must differ, it is still useful to leverage the techniques for aggregate decomposition used in database systems [BBKV87, YC95].

The approach used in such systems (and followed in TAG) 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 $< S_2, C_2 >$:

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

[1]Queries may optionally specify that the system time, in milliseconds since the start of the query, be included as a field in each result tuple.

	MAX, MIN	COUNT SUM	AVERAGE	MEDIAN	COUNT [3] DISTINCT	HIST [4]	Section
Duplicate Sensitive	No	Yes	Yes	Yes	No	Yes	S. 4.6.5
Exemplary (E), Summary (S)	E	S	S	E	S	S	S. 4.5.2
Monotonic	Yes	Yes	No	No	Yes	No	S. 4.3.2
Partial State	Distributive	Distributive	Algebraic	Holistic	Unique	Content-Sensitive	S. 4.4.1

Table 4.1: Classes of Aggregates

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, any operation that can be expressed as commutative applications of a binary function is expressible.

4.2.3 Taxonomy of Aggregates

Given our basic syntax and structure of aggregates, an obvious question remains: what aggregate functions can be expressed in TAG? The original SQL specification offers just five options: COUNT, MIN, MAX, SUM, and AVERAGE. Although these basic functions are suitable for a wide range of database applications, we did not wish to constrain TinyDB to only these choices. For this reason, we present a general classification of aggregate functions and show how the dimensions of that classification affect the performance of TAG throughout the paper.

When aggregation functions are registered with TAG, they are classified along the dimensions described below. In the current implementation of TinyDB,aggregates are pre-compiled into motes and the aggregate properties are manually added to the catalog. [2]

[2] We have begun discussions on using proposed virtual-machine languages (e.g, Maté [LC02]) for this purpose.

[3] The HIST aggregate sorts sensor readings into fixed-width buckets and returns the size of each bucket; it is content-sensitive because the number of buckets varies depending on how widely spaced sensor readings are.

[4] COUNT DISTINCT returns the number of distinct values reported across all motes.

We classify aggregates according to four properties that are particularly important to sensor networks. Table 4.1 shows how specific aggregation functions can be classified according to these properties, and indicates the sections of the chapter where the various dimensions of the classification are emphasized.

The first dimension is duplicate sensitivity. *Duplicate insensitive* aggregates are unaffected by duplicate readings from a single device while *duplicate sensitive* aggregates will change when a duplicate reading is reported. Duplicate sensitivity implies restrictions on network properties and on certain optimizations, as described in Section 4.6.5.

Second, *exemplary* aggregates return one or more representative values from the set of all values; *summary* aggregates compute some property over all values. This distinction is important because exemplary aggregates behave unpredictably in the face of loss, and, for the same reason, are not amenable to sampling. Conversely, for summary aggregates, the aggregate applied to a subset can be treated as a robust approximation of the true aggregate value, assuming that either the subset is chosen randomly, or that the correlations in the subset can be accounted for in the approximation logic.

Third, *monotonic* aggregates have the property that when two partial state records, s_1 and s_2, are combined via f, the resulting state record s' will have the property that either $\forall s_1, s_2, e(s') \geq MAX(e(s_1), e(s_2))$ or $\forall s_1, s_2, e(s') \leq MIN(e(s_1), e(s_2))$. This is important when determining whether some predicates (such as `HAVING`) can be applied *in network*, before the final value of the aggregate is known. Early predicate evaluation saves messages by reducing the distance that partial state records must flow up the aggregation tree.

The fourth dimension relates to the amount of state required for each partial state record. For example, a partial `AVERAGE` record consists of a pair of values, while a partial `COUNT` record constitutes only a single value. Though TAG correctly computes any aggregate that conforms to the

specification of f in Section 4.2 above, its performance is inversely related to the amount of intermediate state required per aggregate. The first three categories of this dimension (e.g. distributive, algebraic, holistic) were initially presented in work on data-cubes [GBLP96].

- In *Distributive* aggregates, the partial state is simply the aggregate for the partition of data over which they are computed. Hence the size of the partial state records is the same as the size of the final aggregate.
- In *Algebraic* aggregates, the partial state records are not themselves aggregates for the partitions, but are of constant size.
- In *Holistic* aggregates, the partial state records are proportional in size to the set of data in the partition. In essence, for holistic aggregates no useful partial aggregation can be done, and all the data must be brought together to be aggregated by the evaluator.
- *Unique* aggregates are similar to holistic aggregates, except that the amount of state that must be propagated is proportional to the number of distinct values in the partition.
- In *Content-Sensitive* aggregates, the partial state records are proportional in size to some (perhaps statistical) property of the data values in the partition. Many approximate aggregates proposed recently in the database literature are content-sensitive. Examples of such aggregates include fixed-width histograms, wavelets, and so on; see [BDF+97] for an overview of such functions.

In summary, we have classified aggregates according to their state requirements, tolerance of loss, duplicate sensitivity, and monotonicity. We will refer back to this classification throughout the chapter, as these properties will determine the applicability of communication optimizations we present later. Understanding how aggregates fit into these categories is an important issue that is critical (and useful) in many aspects of sensor data collection.

Note that our formulation of aggregate functions, combined with this taxonomy, is flexible

enough to encompass a wide range of sophisticated operations. For example, we have implemented an *isobar finding* aggregate. This is a duplicate-insensitive, summary, monotonic, content-sensitive aggregate that builds a topological map representing discrete bands of one attribute (light, for example) plotted against two other attributes (x and y position in some local coordinate space, for example.) We will discuss isobar aggregates and other sophisticated, in-network techniques in Chapter 7.

4.3 In-Network Aggregates

In this section, we discuss the implementation of the core TAG algorithm for in-network aggregation.

A naive implementation of sensor network aggregation would be to use a centralized, *server-based* approach where all sensor readings are sent to the base station, which then computes the aggregates. In TAG, however, we compute aggregates in-network whenever possible, because, if properly implemented, this approach requires fewer message transmissions, is lower latency, and uses less power than the server-based approach. We measure the advantage of in-network aggregation in Section 4.4 below; first, we present the basic algorithm in detail. We consider the operation of the basic approach in the absence of grouping; we show how to extend it with grouping in Section 4.3.2.

4.3.1 Tiny Aggregation

TAG consists of two phases: a *distribution* phase, in which aggregate queries are pushed down into the network, and a *collection* phase, where the aggregate values are continually routed up from children to parents. Recall that our query semantics partition time into epochs of duration i, and that we must produce a single aggregate value (or a single value per group) that combines the readings of all devices in the network during that epoch.

Basic Scheduling

Given our goal of communicating as little as possible, the collection phase must ensure that parents in the routing tree wait until they have heard from their children before propagating an aggregate value for the current epoch. To accomplish this, we use the slotted communication scheduling approach described in Section 3.6 above. Recall that this approach works by subdividing the epoch such that children transmit their data (partial state records, in this case) during the interval when their parents will be listening, which is defined by the parent's level in the routing tree.

When a mote p receives an aggregate query, r, it awakens, synchronizes its clock according to timing information in the message, and prepares to participate in aggregation by allocating temporary state for aggregate records and initializing the query plan. p then forwards the query request r down the network, to include any nodes that did not hear the previous transmission, and include them as children. These nodes continue to forward the request in this manner, until the query has been propagated throughout the network.

During the epoch after query propagation, each mote listens for messages from its children during the interval corresponding to its level in the routing tree. It then computes a partial state record consisting of the combination of any child values it heard with its own local sensor readings. Finally, during its own transmission interval the mote transmits this partial state record up the network. In this way, aggregates flow back up the tree interval-by-interval. Eventually, a complete aggregate arrives at the root. During each subsequent epoch, a new aggregate is produced.

Figure 4.1 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 sensors 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, just as in our scheduled communication approach from Section 3.6, motes are idle for a significant portion of each epoch so they can enter a low power sleeping state.

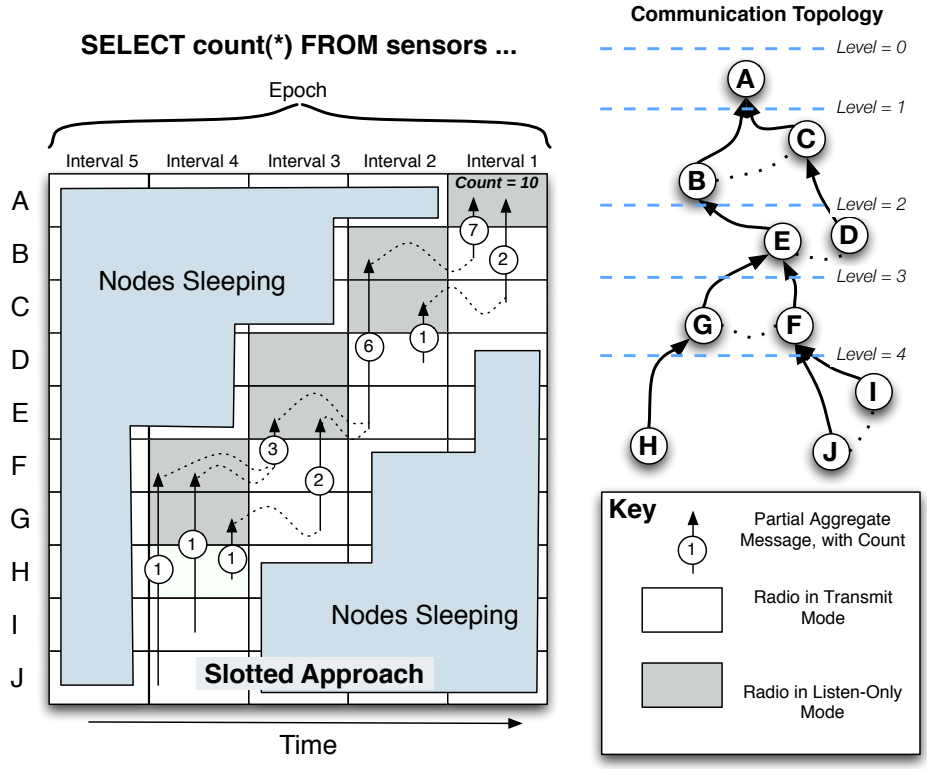


Figure 4.1: Partial state records flowing up the tree during an epoch using the basic, interval-based communication approach.

Increasing Throughput Via Pipelining

Note that communication scheduling imposes a lower bound SAMPLE PERIOD, constraining the maximum sample rate of the network, since the epoch must be long enough for partial state records from the bottom of the tree to propagate all the way to the root. To increase the sample rate, we

can pipeline. With pipelining, the output of the network will be delayed by one or more epochs, as some nodes must wait until the next epoch to report the aggregates they collected during the current epoch. The benefit of this approach is that the effective sample rate of the system is increased (for the same reason that pipelining a long processor stage increases the clock rate of a CPU.)

Figure 4.2 illustrates a pipelined version of the communications schedule shown in Figure 4.1 where the `SAMPLE PERIOD` is two-fifths the sample period in the original schedule. Notice, however, that results from epoch N do not arrive at the root until epoch $N + 1$, and that a far greater portion of the network is active during every cycle, since each node transmits two-and-a-half times more often but the duration of each interval remains the same.

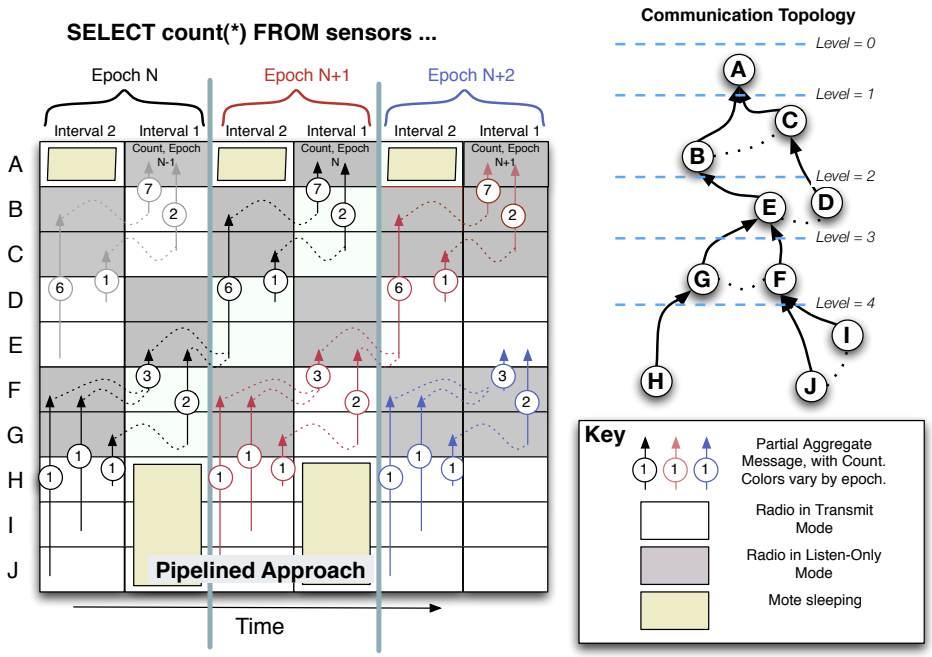


Figure 4.2: Pipelined approach to time scheduled aggregation.

In general, for a network that is d levels deep, with intervals that are i seconds long and a sample period of t seconds, pipelining will be needed if $t < d \times i$. The number of intervals per epoch can then be set to $I = \lfloor t/i \rfloor$ – notice that this will result in a faster sample period than requested if t is not an integer multiple of i. For example if $t = 8s$ and $i = 3s$, then $I = 2$ will result in a reading

being produced every six seconds. To correct for this, we introduce an extra sleep interval at the end of every epoch equal to t mod i seconds. For the example above $8 \bmod 3 = 2$, so all nodes will sleep for an additional two seconds at the end of every $I = 2$ intervals.

In this section, we described two approaches for scheduling communication when running TAG, observing that the pipelined approach can be used in situations where a short sample period is needed. Deciding which approach to use can be done automatically by the system on a per-query basis: when the sample period is long enough that pipelining is not needed, it is not used, since it unnecessarily delays the output of readings. Given these algorithms for in-network aggregation, we extend them with support for grouping in the next section.

4.3.2 Grouping

As described in Section 4.2, grouping in TAG is functionally equivalent to the `GROUP BY` clause in SQL: each sensor reading is placed into exactly one group, and groups are partitioned according to an expression over one or more attributes. The basic grouping technique is to push the grouping expression down with the query, ask nodes to choose the group they belong to, and then, as answers flow back, update the aggregate values in the appropriate groups.

Partial state records are aggregated just as in the approach described in the previous section, except that those records are now tagged with a group id. When a node is a leaf, it applies the grouping expression to compute a group id. It then tags its partial state record with the group and forwards it on to its parent. When a node receives an aggregate from a child, it checks the group id. If the child is in the same group as the node, the parent combines the two values using the combining function f. If it is in a different group, the parent stores the value of the child's group along with its own value for forwarding in the next epoch. If another child message arrives with a value in either group, the node updates the appropriate aggregate. During the next epoch, the node sends the value of all the groups about which it collected information during the previous epoch, concatenating

information about multiple groups into a single message as long as message size permits.

Figure 4.3 shows an example of computing a query that selects average light readings grouped by temperature. The basic network topology and light and temperature readings of each of the sensors are shown on the left, and the contents of the query messages are shown on the right. Each query message contains one or more groups, as well as the running average for that group. Note that, as described in Section 4.2, these running averages can be decomposed into <SUM,COUNT> pairs. Parenthesized numbers next to group readings indicate the sensors that contributed to a particular average; these are not actually transmitted.

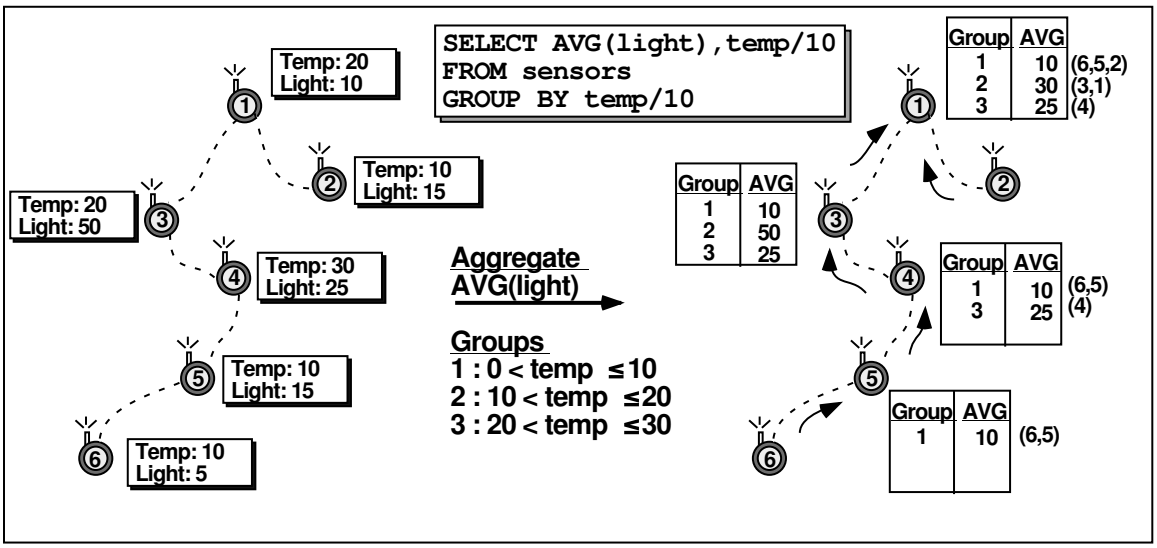


Figure 4.3: *A sensor network (left) with an in-network, grouped aggregate applied to it (right).* Parenthesized numbers represent nodes that contribute to the average they are included for the Reader's benefit – the sensors do not actually track this information.

Recall that queries may contain a HAVING clause, which constrains the set of groups in the final query result. This predicate can sometimes be passed into the network along with the grouping expression. The predicate is sent only if it can potentially be used to reduce the number of messages that must be sent: for example, if the predicate is of the form MAX(attr) $<$ x, then information about groups with MAX(attr) $\geq$ x need not be transmitted up the tree, and so the predicate is sent down into the network. When a node detects that a group does not satisfy a HAVING clause,

it can notify other nodes of this to suppress transmission and storage of values from that group. Note that `HAVING` clauses can be pushed down only for monotonic aggregates; non-monotonic aggregates are not amenable to this technique. However, not all `HAVING` predicates on monotonic aggregates can be pushed down; for example, `MAX(attr)` $>$ `x` cannot be applied in the network because a node cannot know that, just because its local value of $attr$ is less than x, the `MAX` over the entire group is less than x.

Grouping introduces an additional problem: the number of groups can exceed available storage on any one (non-leaf) device. Our solution is to evict one or more groups from local storage. Once an eviction victim is selected, it is forwarded to the node's parent, which may choose to hold on to the group or continue to forward it up the tree. Notice that a single node may evict several groups in a single epoch (or the same group multiple times, if a bad victim is selected). This is because, once group storage is full, if only one group is evicted at a time, a new eviction decision must be made every time a value representing an unknown or previously evicted group arrives. Because groups can be evicted, the base station at the top of the network may be called upon to combine partial groups to form an accurate aggregate value. Evicting partially computed groups is known as *partial preaggregation*, as described in [Lar02].

There are a number of possible policies for choosing which group to evict. We hypothesized that policies that incur a significant storage overhead (more than a few bits per group) are undesirable because they will reduce the number of groups that can be stored and increase the number of messages that must be sent. Intuitively it seems as though evicting groups with low membership is likely a good policy, as those are the groups that are least likely to be combined with other sensor readings and so are the groups that benefit the least from in-network aggregation.

We have shown how to partition sensor readings into a number of groups and properly compute aggregates over those groups, even when the amount of group information exceeds available storage

in any one device. We briefly mention experiments with grouping and group eviction policies in Section 4.4.2. First, we summarize some of the additional benefits of TAG.

4.3.3 Additional Advantages of TAG

The principal advantage of TAG is its ability to dramatically decrease the communication required to compute an aggregate versus a centralized aggregation approach. However, TAG has a number of additional benefits.

One of these is its ability to tolerate disconnections and loss. In sensor environments, it is very likely that some aggregation requests or partial state records will be garbled, or that devices will move or run out of power. These issues will invariably result in some nodes becoming *lost*, either without a parent or not incorporated into the aggregation network during the initial flooding phase. If we include information about queries in partial state records, lost nodes can reconnect by listening to other node's state records not necessarily intended for them as they flow up the tree. We revisit the issue of loss in Section 4.6.

A third advantage of the TAG approach is that, in most cases, each mote is required to transmit only a single message per epoch, regardless of its depth in the routing tree. In the centralized (non-TAG) case, as data converges towards the root, nodes at the top of the tree are required to transmit significantly more data than nodes at the leaves; their batteries are drained faster and the lifetime of the network is limited. Furthermore, because, in the centralized scheme, the top of the routing tree must forward messages for every node in the network, the maximum sample rate of the system is inversely proportional to the total number of nodes. To see this, consider a radio channel with a capacity of n messages per second. If m motes are participating in a centralized aggregate, to obtain a sample rate of k samples per second, $m \times k$ messages must flow through the root during each epoch. $m \times k$ must be no larger than n, so the sample rate k can be at most n/m messages per mote per epoch, regardless of the network density. When using TAG, the maximum transmission

rate is limited instead by the occupancy of the largest radio-cell; in general, we expect that each cell will contain far fewer than m motes.

Yet another advantage of TAG is that, by explicitly dividing time into epochs, a convenient mechanism for idling the processor is obtained. The long idle times in Figure 4.1 show how this effect: during idle intervals, the radio and processor can be put into deep sleep modes that use very little power. Of course, some bootstrapping phase is needed where motes can learn about queries currently in the system, acquire a parent, and synchronize clocks. The simple strategy used in TinyDB is to have every node wake up infrequently (but periodically) and advertise this information. Devices that have never synchronized or have not heard from their neighbors for some time leave their radios on for several advertising periods at a time, hoping to hear a neighbor and join the query. Research on energy aware MAC protocols [YHE02] presents a similar scheme in detail.

Taken as a whole, these TAG advantages provide users with a stream of aggregate values that changes as sensor readings and the underlying network change. These readings are provided in an energy and bandwidth efficient manner.

4.4 Simulation-Based Evaluation

In this section, we present a simulation environment for TAG and evaluate its behavior using this simulator. We also discuss the performance of the basic features of TAG in TinyDB in an experiment at the end of the chapter, in Section 4.7.

To study the algorithms presented in this chapter, we simulated TAG in Java. The simulator models mote behavior at a coarse level: time is divided into units of epochs, messages are encapsulated into Java objects that are passed directly into nodes without any model of the time to send or decode. Nodes are allowed to compute or transmit arbitrarily within a single epoch, and each

node executes serially. Messages sent by all nodes during one epoch are delivered in random order during the next epoch to model a parallel execution. Note that this simulator cannot account for certain low-level properties of the network: for example, because there is no fine-grained model of time, it is not possible to model radio contention at a byte level.

Our simulation includes an interchangeable communication model that defines connectivity based on geographic distance. Figure 4.4 shows screenshots of a visualization component of our simulation; each square represents a single device, and shading (in these images) represents the number of radio hops the device is from the root (center); darker is closer. We measure the size of networks in terms of *diameter*, or width of the sensor grid (in nodes). Thus, a diameter 50 network contains 2500 devices.

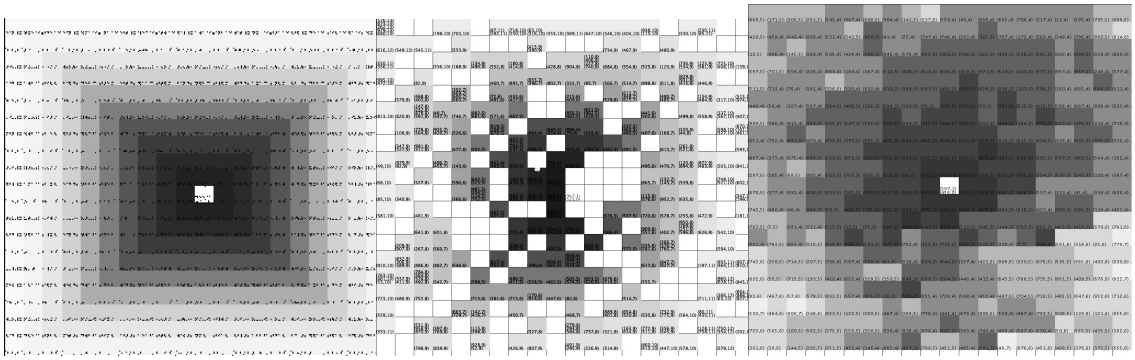
We have run experiments with three communications models; 1) a *simple* model, where nodes have perfect (lossless) communication with their immediate neighbors, which are regularly placed (Figure 4.4(a)), 2) a *random* placement model (Figure 4.4(b)), and 3) a *realistic* model that attempts to capture the actual behavior of the radio and link layer on TinyOS motes (Figure 4.4(c).) In this last model, notice that the number of hops from a particular node to the root is no longer directly proportional to the geographic distance between the node and the root, although the two values are still related. This model uses results from real world experiments [Gan02] to approximate the actual loss characteristics of the TinyOS radio. Loss rates are high in in the realistic model: a pair of adjacent nodes loses more than 20% of the traffic between them. Devices separated by larger distances lose still more traffic.

The simulator also models the costs of topology maintenance: if a node does not transmit a reading for several epochs (which will be the case in some of our optimizations below), that node must periodically send a *heartbeat* to advertise that it is still alive, so that its parents and children know to keep routing data through it. The interval between heartbeats can be chosen arbitrarily;

choosing a longer interval means fewer messages must be sent, but requires nodes to wait longer before deciding that a parent or child has disconnected, making the network less adaptable to rapid change.

This simulation allows us to measure the the number of bytes, messages, and partial state records sent over the radio by each mote. Since we do not simulate the mote CPU, it does not give us an accurate measurement of the number of instructions executed in each mote. It does, however, allow us to obtain an approximate measure of the state required for various algorithms, based on the size of the data structures allocated by each mote.

Unless otherwise specified, our experiments are over the simple radio topology in which there is no loss. We also assume sensor values do not change over the course of a single simulation run.



(a) Simple (b) Random (c) Realistic

Figure 4.4: The TAG Simulator, with Three Different Communications Models, Diameter = 20.

4.4.1 Performance of TAG

In the first set of experiments, we compare the performance of the TAG in-network approach to centralized approaches on queries for the different classes of aggregates discussed in Section 4.2.3. Centralized aggregates have the same communications cost irrespective of the aggregate function, since all data must be sent to the root. We compared this cost to the number of bytes required for distributive aggregates (`MAX` and `COUNT`), an algebraic aggregate (`AVERAGE`), a holistic aggre-

gate (MEDIAN), a content-sensitive histogram aggregate (HIST), and a unique aggregate (COUNT DISTINCT); the results are shown in Figure 4.5. Values in this experiment represent the steady-state cost to extract an additional aggregate from the network once the query has been propagated; the cost to flood a request down the tree is not considered.

The results presented below are highly dependent on the size of the partial state records used to compute aggregates. The benefit seen from in-network aggregation ranges from about 1800% for algebraic aggregates to 0% for holistic aggregates.

In the 2500 node ($d = 50$) network, MAX and COUNT have the same cost when processed in the network, about 5000 bytes per epoch (total over all nodes), since they both send just a single integer per partial state record; similarly AVERAGE requires just two integers, and thus always has double the cost of the distributive aggregates. MEDIAN costs the same as a centralized aggregate, about 90000 bytes per epoch, which is significantly more expensive than other aggregates, especially for larger networks, as parents have to forward all of their children's values to the root. COUNT DISTINCT is only slightly less expensive (73000 bytes), as there are few duplicate sensor values; a less uniform sensor-value distribution would reduce the cost of this aggregate. For the HIST aggregate, we set the size of the fixed-width buckets to be 10; sensor values ranged over the interval [0..1000]. At about 9000 messages per epoch, HIST provides an efficient means for extracting a density summary of readings from the network.

Note that the benefit of TAG will be more or less pronounced depending on the topology. In a flat, single-hop environment, where all motes are directly connected to the root, TAG is no better than the centralized approach. For a topology where n motes are arranged in a line, centralized aggregates will require $n^2/2$ partial state records to be transmitted, whereas TAG will require only n records.

Thus, this experiment has demonstrated that, for our simulation topology, in-network aggre-

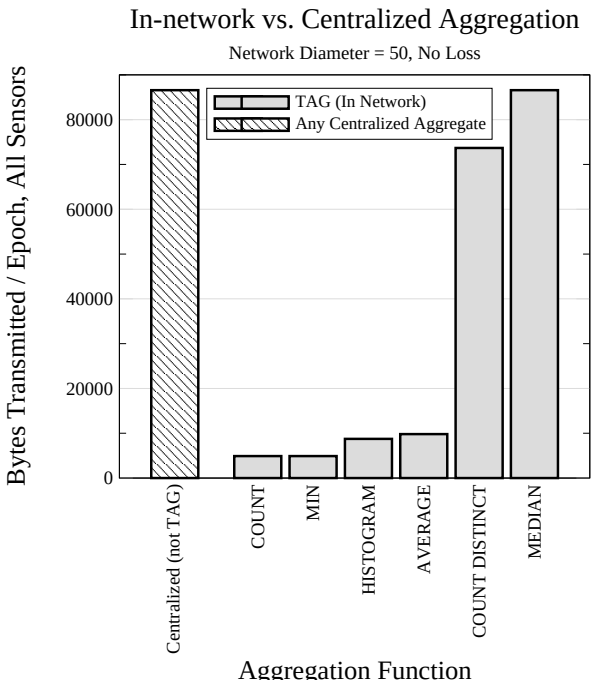


Figure 4.5: In-network Vs. Centralized Aggregates (2500 Nodes)

gation can reduce communication costs by an order of magnitude over centralized approaches, and that, even in the worst case (such as with `MEDIAN`), it provides performance equal to the centralized approach.

4.4.2 Grouping Experiments

We also ran several experiments to measure the performance of grouping in TAG, focusing on the behavior of various eviction techniques in the case that the number of groups exceeds the storage available on a single mote. Sensors had two attributes, *light* and *temperature*; we selected the light attribute grouped by temperature, where temperature can take on 100 distinct values (or, alternatively, can fall within 1 of 100 temperature bins). We measured the number of partial state records sent collectively by all the sensors. We varied three parameters: the eviction policy, the distribution

of sensor values, and the amount of storage available to sensors for group data (in terms of state records).

We experimented with four simple eviction policies; a *random* policy, where the group to evict is selected at random from the uniform distribution, the *evict smallest* policy, where the group with the fewest members is evicted, the *evict largest* policy, where the group with the most members is evicted, and the *evict largest id* policy, where the group representing the most popular temperature attribute is evicted. This latter policy was included to cause parents and children to tend to evict the same groups, allowing parents to combine more groups with their children. For each of these policies, we also experimented with evicting multiple groups simultaneously.

We found that the choice of policy made little difference for any of the sensor-value distributions we tested. This is largely due to the use of a tree topology: near the leaves of the tree, most devices will see only a few groups, and the eviction policy will matter very little. At the top levels of the tree, the eviction policy becomes important, but the cost of forwarding messages from the leaves of the tree tends to dominate (in terms of energy consumed by the network as a whole) the savings obtained at the top. In the most extreme case, the difference between the best and worst case eviction policy accounted for less than 10% of the total messages. We also observed that, when evicting, the best policy was to evict multiple groups at a time, up to the number of group records that will fit into a single radio message.

4.4.3 Effect of Sensor-Value Distribution

In the process of testing our group-eviction policies, we also experimented with several sensor-value distributions. Although the choice of distribution had little impact on the effectiveness of the eviction policies, it did, as expected, effect the total number of messages sent. Widely spaced, uniform distributions, with lots of groups in many parts of the network, tended to incur many more evictions, and hence send more messages, than more clustered, non-uniform distributions with fewer

or more spatially co-located groups.

Specifically, we used: a *random distribution*, in which the temperature attribute for every sensor was selected independently from the uniform distribution; a *geographic distribution*, in which a temperature gradient was applied to the the topology such that sensors near each other tended to fall within the same group; a *90/10 distribution*, such that 90% of the sensors were in a single group, with the remaining 10% being distributed uniformly among the remaining groups; and a *normal distribution*, where sensor temperature was independently selected from a normal distribution centered around the 50th group, with a standard deviation of 15 groups (such that the probability of any sensors falling in the outermost 10 groups is less than 1%). We ran experiments comparing these topologies. In Figure 4.6, we show the results for an experiment with storage for five groups per node with just one record per eviction. Note that the random topology requires the most messages, about 5000 per epoch for a network diameter of 50 (2500 sensors), because sensor values are distributed over a broad range of groups, resulting in more evictions per sensor. Similarly, the 90/10 distribution has the least number of transmissions, at about 3000 per epoch, because groupings are tightly packed. Thus, as would be expected, the cost of grouping depends greatly on the underlying sensor distribution.

4.5 Optimizations

The previous sections described the overall approach to aggregate processing in TAG and examined the performance of several aspects of the approach. In this section, we present several techniques to further improve the performance and accuracy of the basic approach described above. Some of these techniques apply to all aggregates, but some are function dependent; that is, they can only be used for certain classes of aggregates. All of these techniques,however, can be applied in a user-transparent fashion, since they are not explicitly a part of the query syntax and do not affect the

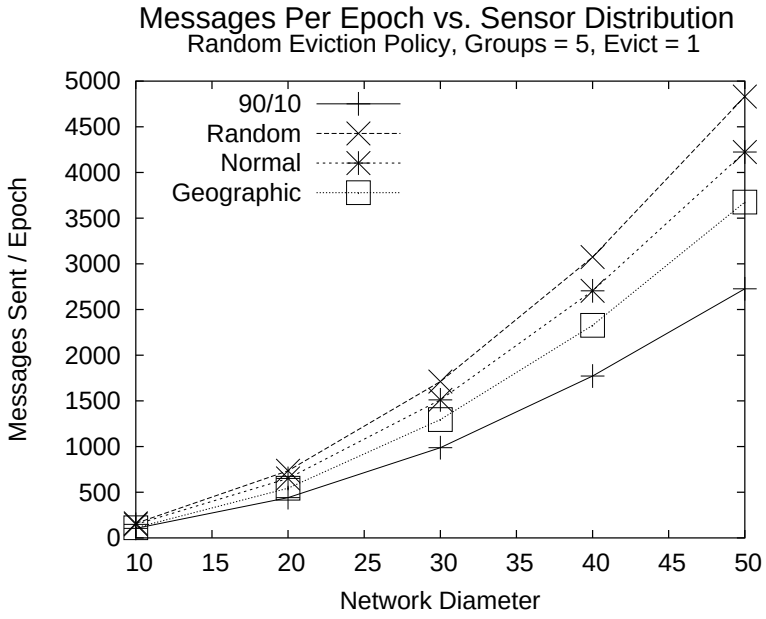


Figure 4.6: Performance of Random Eviction Policy with 5 Groups Storage, Evicting 1 Group Per Eviction, in Different Sensor-Value Distributions.

semantics of the results.

4.5.1 Taking Advantage of A Shared Channel

In our discussion of aggregation algorithms up to this point, we have largely ignored the fact that motes communicate over a shared radio channel. The fact that every message is effectively broadcast to all other nodes within range enables a number of optimizations that can significantly reduce the number of messages transmitted and increase the accuracy of aggregates in the face of transmission failures.

In Section 4.3.3, we saw an example of how a shared channel can be used to increase message efficiency when a node misses an initial request to begin aggregation: it can initiate aggregation even after missing the start request by *snooping* on the network traffic of nearby nodes. When it hears another device reporting an aggregate, it can assume that it too should be aggregating. By allowing

nodes to examine messages not directly addressed to them, motes are automatically integrated into the aggregation. Note that snooping does not require nodes to listen all the time; by listening at predefined intervals (which can be short once a mote has time-synchronized with its neighbors), duty cycles can be kept quite low.

Snooping can also be used to reduce the number of messages sent for some classes of aggregates. Consider computing a `MAX` over a group of motes: if a node hears a peer reporting a maximum value greater than its local maximum, it can elect to not send its own value and be sure that it will not affect the correctness of the final aggregate.

4.5.2 Hypothesis Testing

The snooping example above showed that we only need to hear from a particular node if that node's value will affect the end value of the aggregate. For some aggregates, this fact can be exploited to significantly reduce the number of nodes that need to report. This technique can be generalized to an approach we call *hypothesis testing*. For certain classes of aggregates, if a node is presented with a guess as to the proper value of an aggregate, it can decide locally whether contributing its reading and the readings of its children will affect the ultimate value of the aggregate.

For `MAX`, `MIN` and other monotonic, exemplary aggregates, this technique is directly applicable. There are a number of ways it can be applied – the snooping approach, where nodes suppress their local aggregates if they hear other aggregates that invalidate their own, is one. Alternatively, the root of the network (or any subtree of the network) seeking an exemplary sensor value, such as a `MIN`, might compute the minimum sensor value m over the highest levels of the subtree, and then abort the aggregate and issue a new request asking for values less than m over the whole tree. In this approach, leaf nodes need not send a message if their value is greater than the minimum observed over the top k levels; intermediate nodes, however, must still forward partial state records, so even if their value is suppressed, they may still have to transmit.

Assuming for a moment that sensor values are independent and uniformly distributed, then a particular leaf node must transmit with probability $1/b^k$ (where b is the branching factor, so $1/b^k$ is the number of nodes in the top k levels), which is quite low for even small values of k. For bushy routing trees, this technique offers a significant reduction in message transmissions – a completely balanced routing tree would cut the number of messages required to $1/k$. Of course, the performance benefit may not be as substantial for other, non-uniform, sensor value distributions; for instance, a distribution in which all sensor readings are clustered around the minimum will not allow many messages to be saved by hypothesis testing. Similarly, less balanced topologies (e.g. a line of nodes) will not benefit from this approach.

For summary aggregates, such as AVERAGE or VARIANCE, hypothesis testing via a guess from the root can be applied, although the message savings are not as dramatic as with monotonic aggregates. Note that the snooping approach cannot be used: it only applies to monotonic, exemplary aggregates where values can be suppressed locally without any information from a central coordinator. To obtain any benefit with summary aggregates and hypothesis testing, the user must define a fixed-size error bound that he or she is willing to tolerate over the value of the aggregate; this bound is sent into the network along with the hypothesis value.

Consider the case of an AVERAGE: any device whose sensor value is within the error bound of the hypothesis value need not answer – its parent will then assume that its value is the same as the approximate answer and count it accordingly (to apply this technique with AVERAGE, parents must know how many children they have.) It can be shown that the total computed average will not be off from the actual average by more than the error bound, and leaf nodes with values close to the average will not be required to report. Obviously, the value of this scheme depends on the distribution of sensor values. In a uniform distribution, the fraction of leaves that need not report approximates the size of the error bound divided by the size of the sensor value distribution interval.

If values are normally distributed, a much larger fraction of leaves do not report.

We conducted a simple experiment to measure the benefit of hypothesis testing and snooping for a MAX aggregate. The results are shown in Figure 4.7. In this experiment, sensor values were uniformly distributed over the range [0..100], and a hypothesis was made at the root. Lines in the graph represent the aggregate number of messages sent by all the sensors in each epoch using different hypotheses, or the snooping approach described in the previous section, at different network diameters.

Notice that the performance savings are nearly two-fold for a hypothesis of 90 over the "No Hypothesis" case at a diameter of 50. When we compared the hypothesis testing approach with the snooping approach (which will be effective even in a non-uniform distribution), however, we found that snooping beat the other approaches by offering a nearly three-fold performance increase over the no-hypothesis case. This is because in the densely packed simple node distribution, most devices have three or more neighbors to snoop on, suggesting that only about one in four readings will be forwarded at *every level of the tree*. Conversely, suppression due to hypothesis testing only reduces communication at the leaves. For example, in the "Hypothesis=90" case, even though only ten-percent of the values are transmitted, a substantial portion of the higher levels of the network will be involved in communicating those readings to the root, so the actual savings is only about fifty-percent.

4.6 Improving Tolerance to Loss

Up to this point in our experiments we used a reliable environment where no messages were dropped and no nodes disconnected or went offline. In this section, we address the problem of loss and its effect on the algorithms presented thus far. Unfortunately, unlike in traditional database systems, communication loss is a a fact of life in the sensor domain; the techniques described in the section

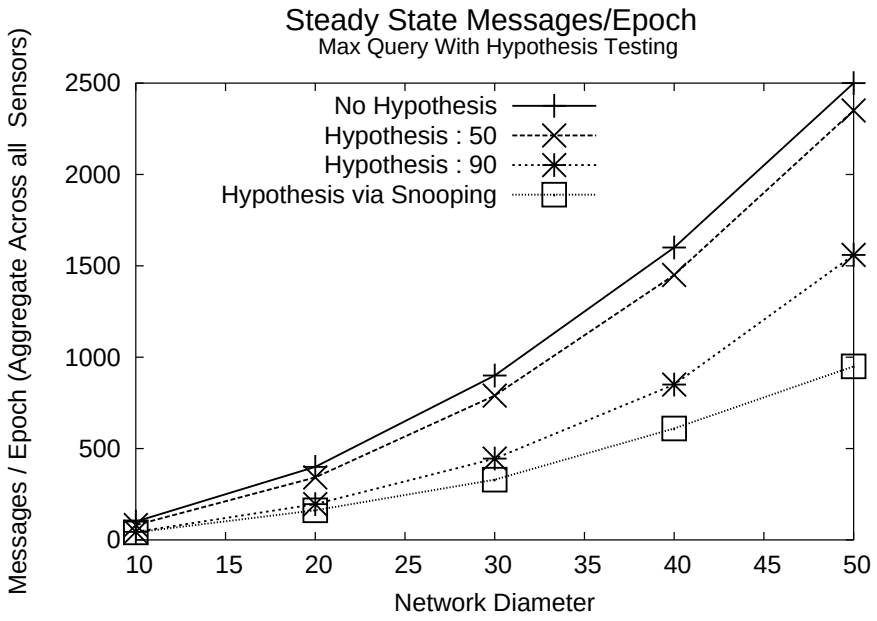


Figure 4.7: Benefit of Hypothesis Testing for MAX

seek to mitigate that loss.

4.6.1 Topology Maintenance and Recovery

TAG is designed to sit on top of a shifting network topology that adapts to the appearance and disappearance of nodes. Although a study of mechanisms for adapting topology is not central to this thesis, for completeness we describe a basic topology maintenance and recovery algorithm which we use in both our simulation and our implementation. This approach is similar to techniques used in existing TinyOS sensor networks, and is derived from the general techniques proposed in the ad-hoc networking literature [SH00, PC97].

Networking faults are monitored and adapted to at two levels: First, each node maintains a small, fixed-sized list of neighbors, and monitors the quality of the link to each of those neighbors by tracking the proportion of packets received from each neighbor. This is done via a locally-unique

sequence number associated with each message by its sender. When a node n observes that the link quality to its parent p is significantly worse than that of some other node p', it chooses p' as its new parent *if* p' is as close or closer to the root as p and p' does not believe n is its parent (the latter two conditions prevent routing cycles.)

Second, when a node observes that it has not heard from its parent for some fixed period of time (relative to the epoch duration of the query it is currently running), it assumes that its parent has failed or moved away. It then resets its local level (to ∞) and picks a new parent from the neighbor table according to the metric used for link-quality. This technique can cause a parent to select a node in the routing subtree underneath it as its parent, so child nodes must reselect their parent (as though a failure had occurred) when they observe that their own parent's level has increased.

Note that switching parents does not introduce the possibility of multiple records arriving from a single node, as each node transmits only once per epoch (even if it switches parents during that epoch.) Parent switching can cause temporary disconnections (and thus additional lost records) in the topology, however, due to children selecting a new parent when their parent's level increases.

4.6.2 Effects of A Single Loss

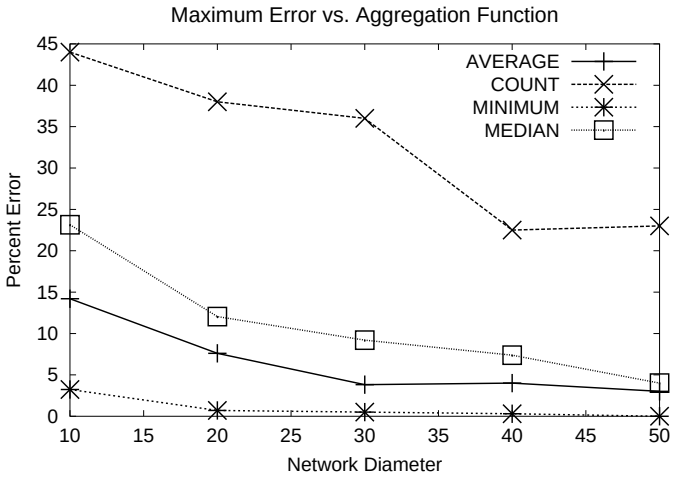
Using the topology maintenance approach described in the previous section, we now study the effect that a single device going offline has on the value of the aggregate; this is an important measurement because it gives some intuition about the magnitude of error that a single loss can generate. Note that, because we are doing hierarchical aggregation, a single mote going offline causes the entire subtree rooted at the node to be (at least temporarily) disconnected. In this first experiment we used the *simple* topology, with sensor readings chosen from the uniform distribution over [1..1000]. After running the simulation for several epochs, we selected, uniformly and at a random, a node to disable. In this environment, children of the disabled node were temporarily disconnected but eventually their values were reintegrated into the aggregate once they discovered their new parents. Note that

the amount of time taken for lost nodes to reintegrate is directly proportional to the depth of the lost node, so we did not measure it experimentally. Instead, we measured the maximum temporary deviation from the true value of the aggregate that the loss caused in the perceived aggregate value at the root during any epoch. This maximum was computed by performing 100 runs at each data point and selecting the largest error reported in any run. We also report the average of the (maximum) error across all 100 runs.

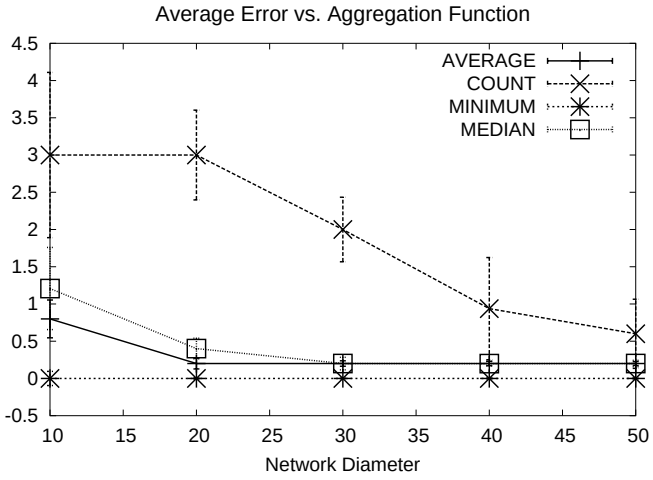
Figure 4.8 shows the results of this experiment. Note that the maximum loss (Figure 4.8(a)) is highly variable and that some aggregates are considerably more sensitive to loss than others. `COUNT`, for instance, has a very large error in the worst case: if a node that connects the root to a large portion of the network is lost, the temporary error will be very high. The variability in maximum error results from the fact that a well connected subtree is not always selected as the victim. Indeed, assuming some uniformity of placement (e.g. the devices are not arranged in a line), as the network size increases, the chances of selecting such a highly connected node go down, since a larger proportion of the nodes reside towards the leaves of the tree. In the average case (Figure 4.8(b)), the error associated with a `COUNT` is not as high: most losses do not result in a large number of disconnections. Note that `MIN` is insensitive to loss in this uniform distribution, since several nodes are at or near the true minimum. The error for `MEDIAN` and `AVERAGE` is less than `COUNT` and more than `MIN`: both are sensitive to the variations in the number of nodes, but not as dramatically as `COUNT`.

4.6.3 Effect The of Realistic Communication Model

In the next experiment, we examine how well TAG performs in the *realistic* simulation environment and topology (discussed in Section 4.4 above). In such an environment, without some technique to counteract loss, a large number of partial state records will invariably be dropped and not reach the root of the tree. We ran an experiment to measure the effect of this loss in the *realistic* environment.



(a) Maximum Error



(b) Average Error

Figure 4.8: *Effect of a Single Loss on Various Aggregate Functions.* Computed over a total of 100 runs at each point. Sensor values were uniformly distributed over the range [0,1000] and the simple network topology was used. Error bars indicate standard error of the mean, 95% confidence intervals.

As before, we varied the network diameter from 10 to 50. In this case, the "No Cache" line of Figure 4.9 shows the performance of this approach (the other lines are explained in the next section); the sensor value distribution and type of aggregate used does not matter. The simulation ran until the first aggregate arrived at the root, and then the average number of motes involved in the aggregate over the next several epochs was measured. At diameter 10, about 40% of the partial state records are reflected in the aggregate at the root; by diameter 50, this percentage has fallen to less than 10%. Performance falls off as the number of hops between the average node and the root increases, since the probability of loss is compounded by each additional hop. Thus, the basic TAG approach presented so far, running on current prototype hardware (with its very high loss rates), is not highly tolerant to loss, especially for large networks. Note that any centralized approach would suffer from the same loss problems.

4.6.4 Child Cache

To improve the quality of aggregates, we propose a simple caching scheme: parents remember the partial state records that their children reported for some number of rounds, and use those previous values when new values are unavailable due to lost child messages. As long as the lifetime of this memory is shorter than the interval at which children select new parents, this technique will increase the number of nodes included in the aggregate without over-counting any nodes. Of course, caching tends to temporally smear the aggregate values that are computed, reducing the temporal resolution of individual readings and possibly making caching undesirable for some workloads. Note that caching is a simple form of interpolation where the interpolated value is the same as the previous value. More sophisticated interpolation schemes, such as curve fitting or statistical observations based on past behavior, could be also be used.

We conducted some experiments to examine the improvement caching offers over the basic approach; we allocate a fixed size buffer at each node and measure the average number of devices

involved in the aggregation as in Section 4.6.3 above. The results are shown in the top three lines of Figure 4.9 – notice that even five epochs of cached state offer a significant increase in the number of nodes counted in any aggregate, and that 15 epochs increases the number of nodes involved in the diameter 50 network to 70% (versus less than 10% without a cache).

Aside from the temporal smearing described above, there are two additional drawbacks to caching; First, it uses memory that could be used for group storage. Second, it sets a minimum bound on the time that devices must wait before determining that their parent has gone offline. Given the benefit it provides in terms of accuracy, however, we believe it to be useful despite these disadvantages. In fact, the substantial benefit of this technique suggests that allocating RAM to application level caching may be more beneficial than allocating it to lower-level schemes for reliable message delivery, as such schemes cannot take advantage of the semantics of the data being transmitted.

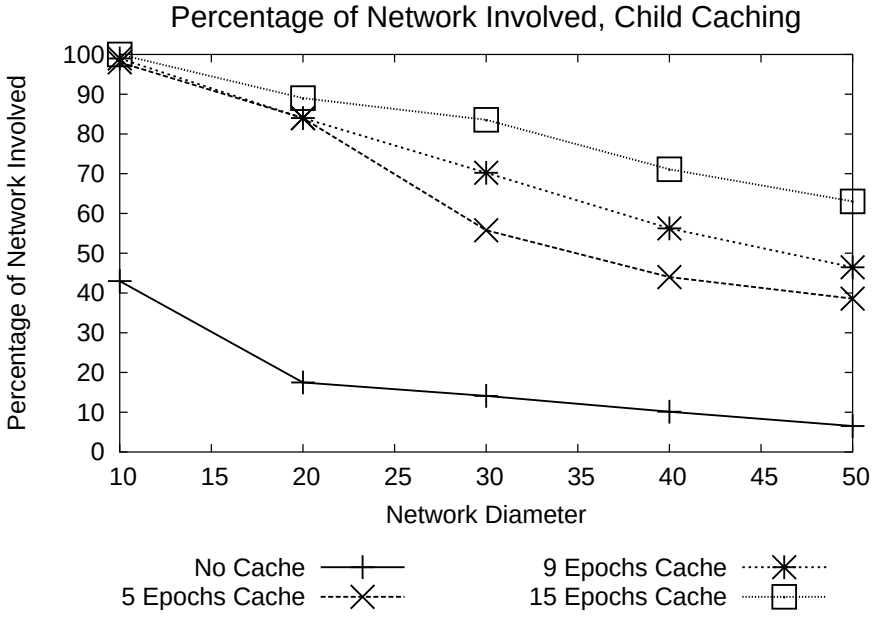


Figure 4.9: Percentage of Network Participating in Aggregate For Varying Amounts of Child Cache

4.6.5 Using Available Redundancy

Because there may be situations where the RAM or latency costs of the child cache are not desirable, it is worthwhile to look at alternative approaches for improving loss tolerance. In this section, we show how the network topology itself can be leveraged to increase the quality of aggregates. Consider a mote with two possible choices of parent: instead of sending its aggregate value to just one parent, it can send it to both parents. Of course, for duplicate-sensitive aggregates (see Section 4.2.3), sending results to multiple parents has the undesirable effect of causing the node to be counted multiple times. The solution to this problem for such aggregates is to send part of the aggregate to one parent and the rest to the other. Consider a `COUNT`; a mote with $c-1$ children and two parents can send a `COUNT` of $c/2$ to both parents instead of a count of c to a single parent. Generally, if the aggregate can be linearly decomposed in this fashion, it is possible to broadcast just a single message that is received and processed by both parents, so this scheme incurs no message overheads, as long as both parents are at the same level and request data delivery during the same sub-interval of the epoch.

A simple statistical analysis reveals the advantage of doing this: assume that a message is transmitted with probability p, and that losses are independent, so that if a message m from node s is lost in transition to parent P_1, it is no more likely to be lost in transit to P_2. [5] First, consider the case where s sends c to a single parent; the expected value of the transmitted count is $p \times c$ (0 with probability $(1-p)$ and c with probability p), and the variance is $c^2 \times p \times (1-p)$, since these are standard Bernoulli trials with a probability of success p multiplied by a constant c. For the case where s sends $c/2$ to both parents, linearity of expectation tells us the expected value is the sum of the expected value through each parent, or $2 \times p \times c/2 = p \times c$. Similarly, we can sum the variances

[5] Although independent failures are not always a valid assumption, they will occur when local interference is the cause of loss. For example, a hidden node may garble communication to P_1 but not P_2, or one parent may be in the process of using the radio when the message arrives.

through each parent:

$$\text{var} = 2 \times (c/2)^2 \times p \times (1-p) = c^2/2 \times p \times (1-p)$$

Thus, the variance of the multiple parent COUNT is much less than with just a single parent, although its expected value is the same. This is because it is much less likely (assuming independence) for the message to both parents to be lost, and a single loss will less dramatically affect the computed value.

We ran an experiment to measure the benefit of this approach in the *realistic* topology for COUNT with a network diameter of 50. We measured the number of devices involved in the aggregation over a 50 epoch period. When sending to multiple parents, the mean COUNT was 974 ($\sigma = 330$), while when sending to only one parent, the mean COUNT was 94 ($\sigma = 41$). Surprisingly, sending to multiple parents substantially increases the mean aggregate value, though our initial analytical analysis indicated it would only affect the variance. The reason for unexpected result is that losses are not, in fact, independent. Without parent splitting, when certain *critical links* at the top levels of the tree go down, all the values from nodes under those link will also be lost – such nodes are *dependent* on this link. Conversely, with parent splitting, values tend to be much more evenly distributed, such that no link is as critical. Figure 4.10 illustrates this scenario; in the graph on the left where no splitting is used, a failure on the critical link causes data from most of the network to be lost.

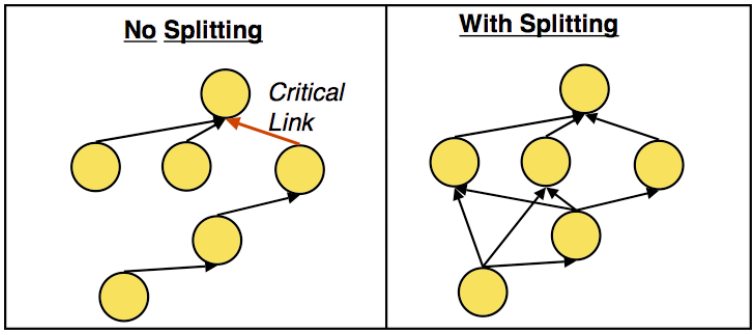


Figure 4.10: Parent splitting helps eliminate critical links.

This technique applies equally well to any distributive or algebraic aggregate. For holistic aggregates, like MEDIAN, this technique cannot be applied, since partial state records cannot be easily decomposed.

4.7 TinyDB Implementation

Based on the encouraging simulation results presented above, we implemented TAG-like techniques in TinyDB. In this section, we briefly summarize results from experiments with this implementation, to demonstrate that the simulation numbers given above are consistent with actual behavior and to show that substantial message reductions over a centralized approach are possible in a real implementation.

These experiments involved sixteen motes arranged in a depth four tree, computing a COUNT aggregate over 150 4-second epochs (a 10 minute run.) No child caching or snooping techniques were used. Figure 4.11 shows the COUNT observed at the root for a centralized approach, where all messages are forwarded to the root, versus the count produced by the in-network TAG approach. Notice that the quality of the aggregate is substantially better with TAG; this is due to reduced radio contention. To measure the extent of contention and compare the message costs of the two schemes, we instrumented motes to report the number of messages sent and received. The centralized approach required 4685 messages, whereas TAG required just 2330, representing a 50% communications reduction. This is less than the order-of-magnitude shown in Figure 4.5 for COUNT because our prototype network topology had a higher average fanout than the simulated environment, so messages in the centralized case had to be retransmitted fewer times to reach the root. Per hop loss rates were about 5% in the in-network approach. In the centralized approach, increased network contention drove these loss rates to 15%. The poor performance of the centralized case is due to the multiplicative accumulation of loss, such that only 45% of the messages from nodes at the bottom

of the routing tree arrived at to the root.

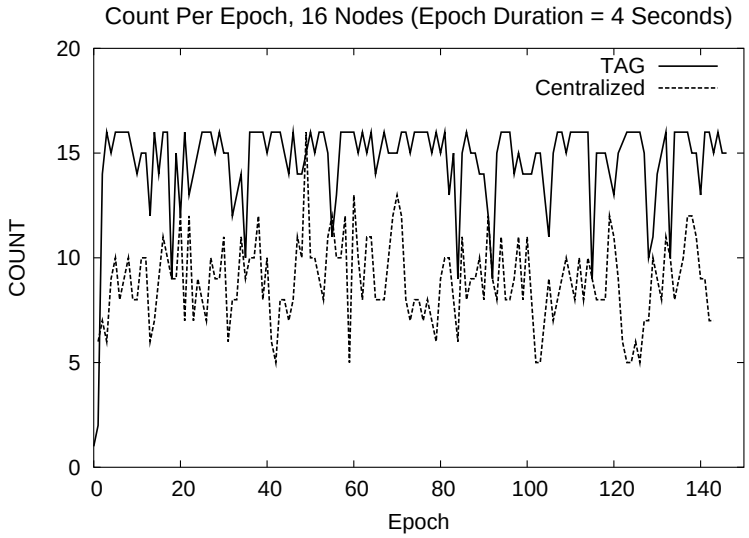


Figure 4.11: Comparison of Centralized and TAG based Aggregation Approaches in Lossy, Prototype Environment Computing a COUNT over a 16 node network.

This completes our discussion of algorithms for TAG. We now summarize research from the networking and database communities that is specifically related to in-network and aggregate query processing.

4.8 Related Aggregation and In-Network Processing Literature

The database community has proposed a number of distributed and push-down based approaches for aggregates in database systems [SN95, YL95], but these universally assume a well-connected, low-loss topology that is unavailable in sensor networks. None of these systems present techniques for loss tolerance or power sensitivity. Furthermore, their notion of aggregates is not tied to a taxonomy, and so techniques for transparently applying various aggregation and routing optimizations are lacking. The partial preaggregation techniques [Lar02] used to enable group eviction were proposed as a technique to deal with very large numbers of groups to improve the efficiency of hash

joins and other bucket-based database operators.

The first three components of the partial-state dimension of the taxonomy presented in Section 4.2.3 (e.g. algebraic, distributive, and holistic) were originally developed as a part of the research on data-cubes [GBLP96]; the duplicate sensitivity, exemplary vs. summary, and monotonicity dimensions, as well as the unique and content-sensitive state components of partial-state are our own addition. [TGO99] discusses online aggregation [HHW97] in the context of nested-queries; it proposes optimizations to reduce tuple-flow between outer and inner queries that bear similarities to our technique of pushing `HAVING` clauses into the network.

With respect to query language, our epoch based approach is related to languages and models from the Temporal Database literature; see [Sno95] for a survey of relevant work. The Cougar project at Cornell [PJP01] discusses queries over sensor networks, as does our own work on Fjords [MF02], although the former only considers moving selections (not aggregates) into the network and neither presents specific algorithms for use in sensor networks.

Literature on active networks [TSS+97] identified the idea that the network could simultaneously route and transform data, rather than simply serving as an end-to-end data conduit. The recent SIGCOMM paper on ESP [CGW02] provides a constrained framework for in-network aggregation-like operations in a traditional network.

The work on directed diffusion [IGE00] from UCLA proposes several techniques related specifically to TAG. Low-level-naming [HSI+01] is a scheme for imposing names onto related groups of devices in a network, in much the way that our scheme partitions sensor networks into groups. Work on greedy aggregation [IEGH01] discusses networking protocols for routing data to improve the extent to which data can be combined as it flows up a sensor network – it provides low level techniques for building routing trees that could be useful in computing TAG style aggregates.

We mentioned earlier (in Section 2.7.1) that in Diffusion, aggregation is viewed as an application-

specific operation that must always be coded in a low-level language. Although some TAG aggregates may also be application-specific, we ask that users provide certain functional guarantees, such as composability with other aggregates, and a classification of semantics (quantity of partial state, monotonicity, etc.). These properties enable transparent application of various optimizations and create the possibility of a library of common aggregates that TAG users can freely apply within their queries. Furthermore, directed diffusion puts aggregation APIs in the routing layer, so that expressing aggregates requires thinking about how data will be collected, rather than just what data will be collected. By contrast, our goal is to separate the expression of aggregation logic from the details of routing. This separation allows users to focus on application issues and enables the system to dynamically adjust routing decisions using general (taxonomic) information about each aggregation function.

Work on (N)ACKs (and suppression thereof) in scalable, reliable multicast trees [FJL+97, LP96] bears some similarity to the problem of propagating an aggregate up a routing tree in TAG. These systems, however, consider only fixed, limited types of aggregates (e.g. ACKs or NAKs for regions or recovery groups.) Finally, we presented an early version the aggregation protocols discussed in this chapter in a workshop publication [MSFC02].

4.9 Summary

In this chapter, we presented the TAG framework for in-network processing of queries. The basic framework offers substantial reductions in the radio bandwidth and energy used to process aggregate queries using a sensor network by computing aggregation functions at intermediate points in the network topology. We further refined the approach by classifying aggregation operators according to a taxonomy and discussing a number of optimizations that can be applied to operators with certain taxonomic properties.

In the next chapter, we look at issues related to the costs of data acquisition in sensor networks. The *acquisitional* techniques we propose are complimentary to the execution framework described in this chapter – TAG benefits by being sensitive to the costs of sampling, and, as we will see, TAG-like techniques are often useful when choosing which samples to acquire from a sensor network.

Chapter 5

Acquisitional Query Processing In Sensor Networks

In this chapter, we introduce the concept of acquisitional query processing. 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 compared to 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 and show how acquisitional techniques can provide significant reductions in power consumption on our sensor devices.

5.1 Introduction

At first blush, it may seem as though query processing in sensor networks is simply a power-constrained version of traditional query processing: given some set of data, the goal of sensor network query processing is to process that data as energy-efficiently as possible. Typical strategies (some of which we saw in the previous chapter) 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.

There is, however, another fundamental difference between systems like sensor networks and traditional database systems, that has to do with the role of data acquisition in query processing. In this chapter we describe the process of *acquisitional query processing* (ACQP), focusing 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 for 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 ACQP features in TinyDB. While TinyDB has many of the features of a traditional query processor (e.g., the ability to select, join, project, and aggregate data), it 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 systems that do not actively control when and where data is collected.

In this chapter, 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 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.

The remainder of the chapter discusses each of these phases of ACQP: Section 5.2 covers our query language, Section 5.3 highlights optimization issues in power-sensitive environments, Section 5.4 discusses query dissemination, and finally, Sections 5.5 discusses our adaptive, power-sensitive model for query execution and result collection.

5.2 An Acquisitional Query Language

In this section, we introduce extensions to our query language for ACQP, focusing on issues related to when and how often samples are acquired.

5.2.1 Event-Based Queries

As a variation on the continuous, polling based mechanisms for data acquisition described earlier, TinyDB supports *events* as a mechanism for initiating data collection. Events in TinyDB are generated explicitly, either by another query or by the operating system. In the latter 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 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.

Such events are central in TinyDB, as they allow the system to be dormant until some external conditions occur, instead of continually polling or blocking, waiting for 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.

To demonstrate the potential benefits, Figure 5.1(a) 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 Figure 5.1(b), 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.

Queries may also generate events in the `OUTPUT ACTION` clause. For example, the query:

```
SELECT nodeid,temp
 WHERE temp > 80°F
 SAMPLE PERIOD 10s
 OUTPUT ACTION SIGNAL(hot(temp))
```

will cause the “hot” event to be signaled whenever the temperature is above eighty degrees.

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.

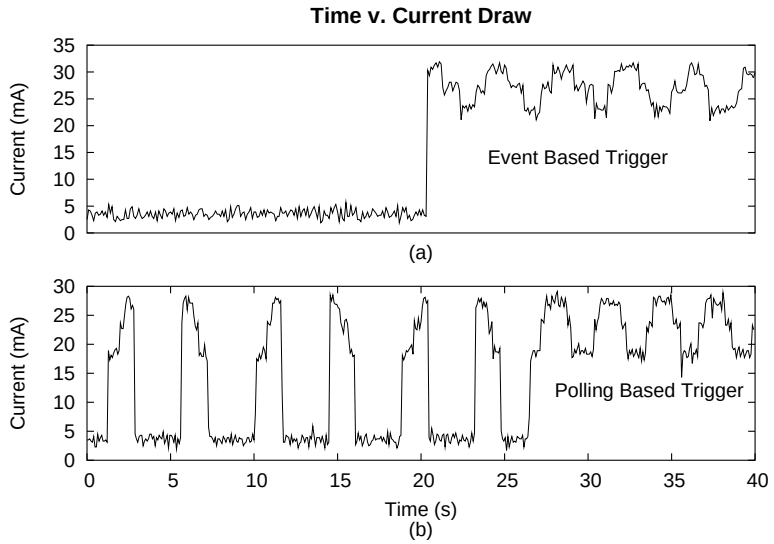


Figure 5.1: (a) External interrupt driven event-based query vs. (b) Polling driven event-based query .

Note that in some cases it is desirable to view event-based queries simply as a join between a stream of sensor tuples and a stream of event tuples. We discuss the optimization tradeoffs of these alternatives in detail in Section 5.3.3.

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).

5.2.2 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 con-

sumption. 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. Below we show 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 acquired. The ordering of multiple predicates and the interleaving of sampling and selection are discussed in detail in Section 5.3. Table 5.1 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.)

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

Parameter	Description	Units
l	Time remaining until lifetime goal is reached	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 5.1: Parameters Used in Lifetime Estimation

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 constant sample rate of 15.2 seconds per sample. We measured the remaining voltage on the device 9 times over 12 days. Because we started with a fresh pair of AA batteries which provide close to 3.2V, the first two voltage readings were outside the 0-3.034V [1] dynamic range of the analog-to-digital converter 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 5.2 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

[1] The Mica's voltage circuit takes a single-ended ADC reading directly from raw battery voltage, with the reference voltage set to the output of the Mica's voltage regulator, which is configured to deliver 3.034V.

of samples, it is important that senders and receivers synchronize their wake cycles. As described in Section 3.6, 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.[2] 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 faster than this rate, sampling proceeds at the computed rate. Otherwise, sampling is fixed at a rate of r and the prior computation for transmission rate is redone. Note that this yields a a different rate for sampling and transmission. To provide the requested lifetime while satisfying both rates 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 5.5.2.

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 and parameters that vary over time (such as the number of children, C), we need to periodically re-estimate power consumption. Section 5.5.3 discusses this runtime re-estimation in more detail.

5.3 Power-Based Query Optimization

We now turn to query processing issues. We begin with a discussion of optimization, and then cover query dissemination and execution.

[2] One possible optimization would involve selecting or reassigning the root to maximize transmission rate.

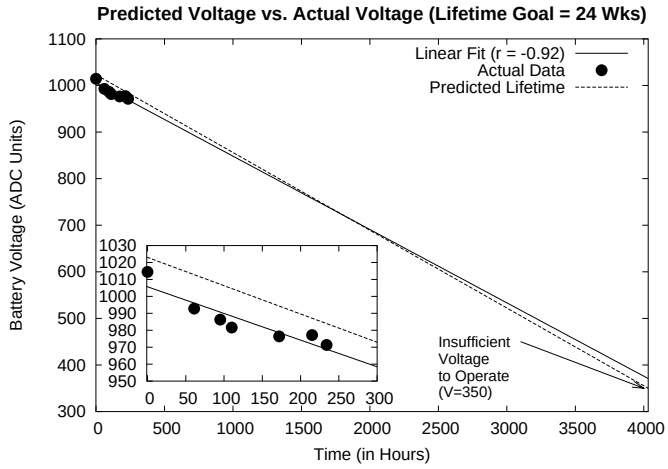


Figure 5.2: Predicted versus actual lifetime for a requested lifetime of 24 weeks (168 days)

Chapter 3 described how 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. Recall that, before queries are disseminated, the basestation performs a simple query optimization phase to choose the correct ordering of sampling, selections, and joins; that optimization process is the focus of the remainder of this section.

As with all database systems, optimization is done to improve the execution performance of queries; in the case of sensor networks, the primary focus is on reducing the energy requirements of query processing. In brief, power-based optimization must take into account two primary factors: the energy costs of acquiring samples, and the communication costs of shipping results throughout a network. This leads to two guiding principles:

- *Avoid sampling, especially of expensive sensors, whenever possible.* As we saw in Section 2.2.3, there is a large spread between the energy costs of sampling expensive sensors and those of sampling cheap sensors. Because some query processing operators, like conjunctive predicates in a `WHERE` clause, will cause the evaluation of a query for a particular tuple to be

aborted before all operators have been applied, we should evaluate such operators early in the query plan, especially when they only require access to low-cost sensor attributes.

- *Avoid applying cardinality increasing operators deep in the network.* Such operators, like joins, should be pushed as close to the root of the network (if not outside of the network), as possible.

We use a simple cost-based optimizer to choose a query plan that will yield the lowest overall power consumption. Optimizing for power subsumes issues of processing cost and radio communication, which both contribute to power consumption. 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.) To understand the costs of sample acquisition, we proceed by looking at the types of metadata stored by the optimizer. Our optimizer focuses on ordering aggregates, selections, and sampling operations that run on individual nodes.

5.3.1 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 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.

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 5.2: Metadata Fields Kept with Each Attribute

Event metadata consists of a name, a signature, and a frequency estimate that is used in query optimization (see Section 5.3.3 below.) User-defined predicates also have a name and a signature, along with a selectivity estimate that is provided by the author of the function.

Table 5.2 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 the value 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 2.2 gave examples of power and sample time values for some actual sensors – for the purposes of this discussion, we are concerned with the fact 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 [SK91] 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 described in the previous chapter, aggregate authors must provide information about functional properties, e.g. whether the aggregate is *monotonic* and whether it is *exemplary* or *summary*.

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 3.5 – 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 Mica motes

(using the same batteries), we found that the average current draw with processor and radio active varied from 13.9 to 17.6 mA (with the average being 15.66 mA).

5.3.2 Technique 1: Ordering of Sampling And Predicates

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

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, predicates that refer to high energy-per-sample attributes are "expensive", and need to be ordered carefully. The predicate ordering problem here is somewhat different than in the earlier literature (e.g., [Hel98]) because here (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 [IK84, KBZ86, Hel98]; Monma and Sidney present the *Series-Parallel Algorithm Using Parallel Chains* [MS79], which gives an optimal ordering of the jobs in $O(|S| \log |S|)$ time.

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 appropriate 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 PERIOD .1s
```

The order of magnitude difference in per-sample costs shown in Table 2.2 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.

Figure 5.3 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.

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 PERIOD 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`), and is general (e.g., it applies outside the context of sensor networks.) 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.

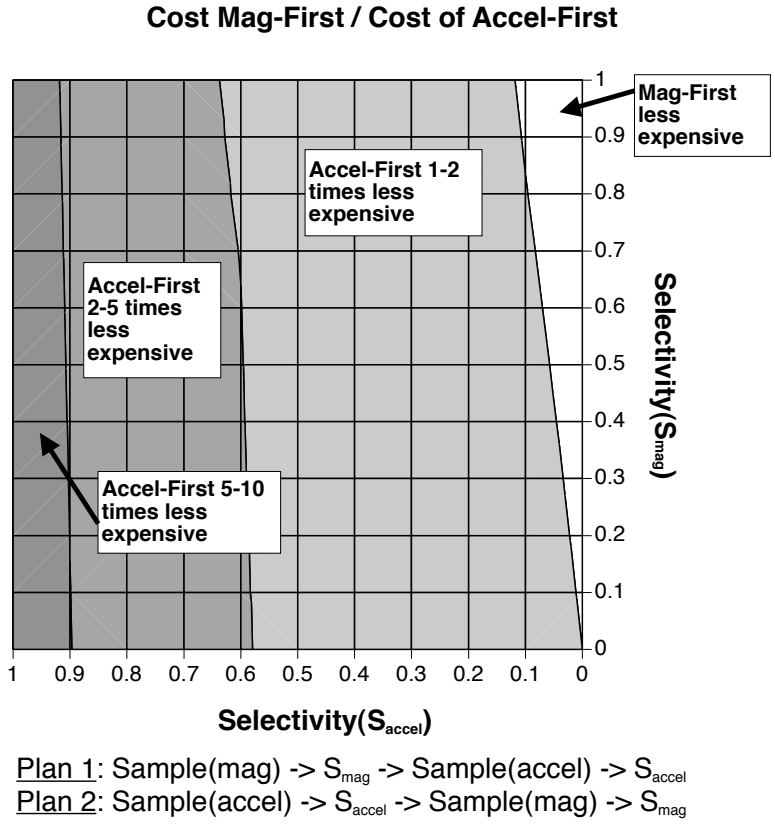


Figure 5.3: Ratio of costs of two acquisitional plans over differing-cost sensors.

5.3.3 Technique 2: Event Query Batching to Conserve Power

In this section, we look at a second example of power-aware optimization that deals specifically with queries with `ON EVENT` clauses. We begin by considering 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
```

A straightforward implementation of this query would 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 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 5.4 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 derived the cost in milliwatts of the two approaches using an analytical 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. Table 5.3 summarizes the parameters used in this experiment; "derived" values are computed by the model below. Power consumption numbers and sensor timings are drawn from Figure 2.2 and the Atmel 128 data sheet [Atm].

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

$$t_{idle} = t_{sample} - n_{events} \times dur_{event} \times ms_{sample}/1000$$

$$mJ_{idle} = mW_{idle} \times t_{idle}$$

$$mJ_{sample} = mW_{sample} \times ms_{sample}/1000$$

$$mW_{events} = (n_{events} \times dur_{event} \times mJ_{sample} + mJ_{idle})/t_{sample}$$

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

$$t_{idle} = t_{sample} - (ms_{eventCheck} + ms_{sample})/1000$$

$$mJ_{idle} = mW_{idle} \times t_{idle}$$

$$mJ_{check} = mW_{proc} \times ms_{eventCheck}/1000$$

$$mJ_{sample} = mW_{sample} \times ms_{samples}/1000$$

$$mW_{streamJoin} = (mJ_{check} + mJ_{sample} + mJ_{idle})/t_{sample}$$

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

Table 5.3: Parameters used in Async. Events vs. Stream Join Study

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 firing 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

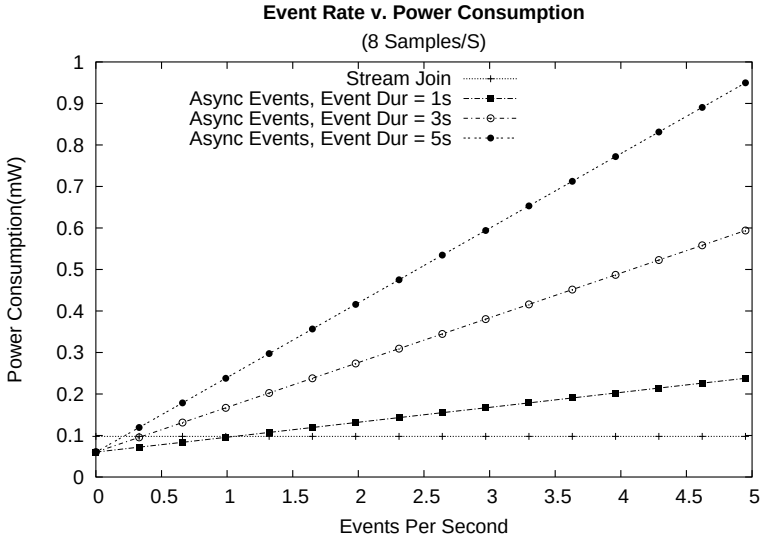


Figure 5.4: The Cost of Processing Event-based Queries as Asynchronous Events Versus Joins.

the power cost of the CPU or radio.

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 event-based queries triggered by frequent events, 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.

5.4 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 *semantic routing trees* (SRTs). We describe the properties of SRTs in the next section, and briefly outline how they are created and maintained.

5.4.1 Semantic Routing Trees

An SRT is a routing tree (similar to the tree discussed in Section 2.3.3 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 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 [WC01], 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. [3] 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

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

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 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 5.5 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.)

5.4.2 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

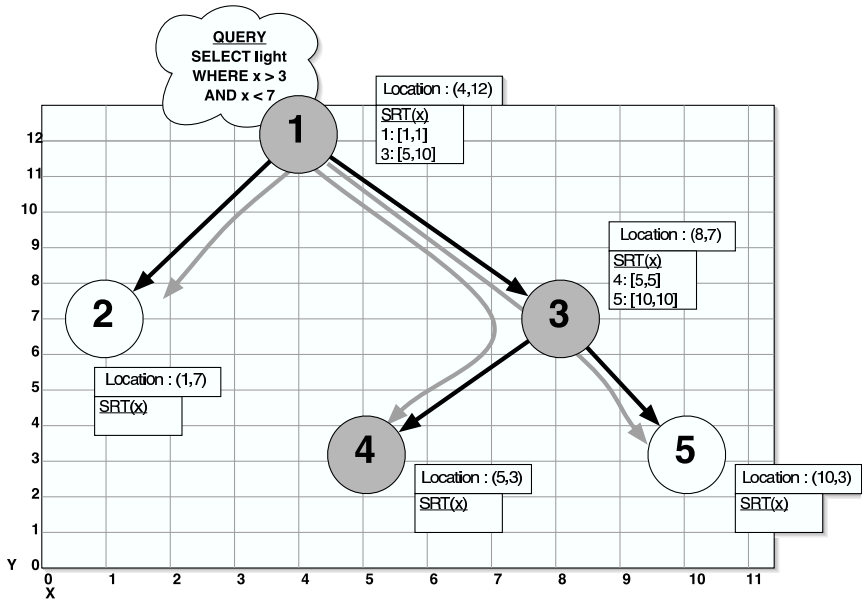


Figure 5.5: A semantic routing tree in use for a query. Gray arrows indicate flow of the query down the tree, gray nodes must produce or forward results in the query.

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 proposed maintenance rules 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.

5.4.3 Evaluation 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 approach, *random*, each node picks a random parent from the nodes with which it can communication reliably. In the second approach, *closest-parent*, 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 third approach, *clustered*, 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 the simple connectivity model (as described in Chapter 4) where each node can 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* and the *no SRT* approaches. The *best-case* results when exactly those nodes that overlap the range predicate are 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.

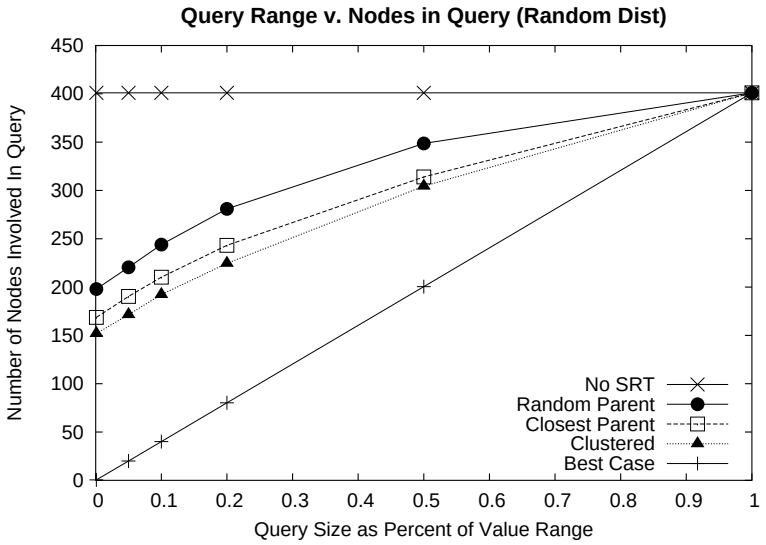
Figure 5.6 shows the number of nodes which participate in queries over variably-sized query

intervals of the attribute space (where the interval size is shown on the X axis) when sensor values are distributed according to the (a) random or (b) geographic distribution. In both experiments, 400 sensors were arranged on a 20x20 grid. The starting endpoint of the interval for each query 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. The results reported are the results of these measurements.

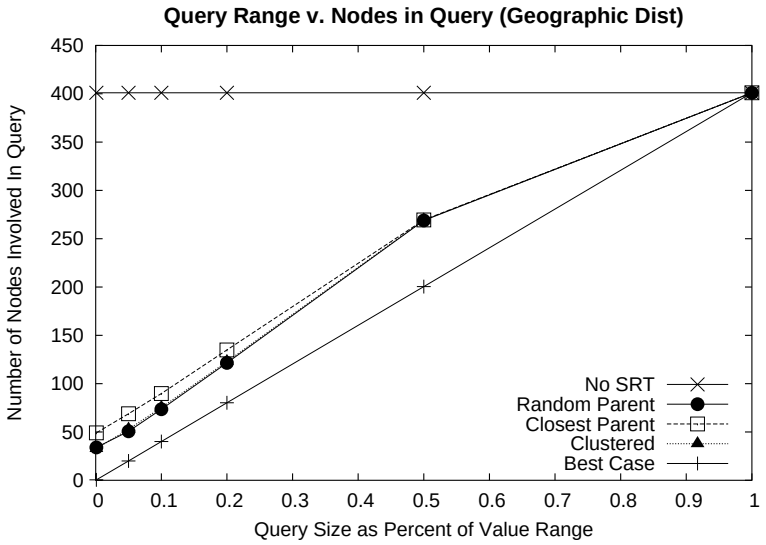
For both distributions, the clustered approach was superior to other SRT algorithms, beating the random approach by about 25% and the closest parent approach by about 10% on average. With the geographic distribution, the performance of the clustered approach is close to best-case: for most ranges, all 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 in our experiment. Real sensor network deployments show significant but imperfect correlation [GKW+02].

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 result 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



(a)Random Distribution



(b)Geographic Distribution (Clustered and Random Perform Identically)

Figure 5.6: 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.)

always require confirmation from their children in other sensor network environments.

5.4.4 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. In general, of course, the benefit of this approach will depend on the depth and bushiness of the network topology.

5.5 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.

5.5.1 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 to 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 5.5.3 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 rely upon the Tiny Aggregation approach proposed in the previous chapter.

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 to a local materialization point.

5.5.2 Prioritizing Data Delivery

Given this basic approach to data collection, we now discuss how results that have been produced are prioritized for delivery over the radio. Results from running queries 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 enqueue the overflow tuple in favor of some other pending result, combine it with some other tuple for the same query, or simply discard the tuple.

The ability to make runtime decisions about the worth 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 [RRH02]. 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. [GG01, CGRS01]), 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 5.7 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.

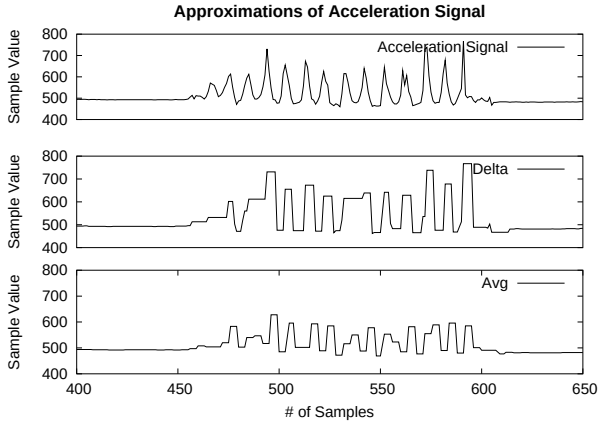


Figure 5.7: An acceleration signal (top) approximated by a delta (middle) and an average (bottom), K=4.

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 5.4. 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 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 omit a discussion of prioritization policies for aggregation queries. In previous chapter, we

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

Table 5.4: RMS Error (in Raw ADC Units) for Different Prioritization Schemes and Signals (1000 Samples, Sample Interval = 64ms)

saw see several snooping-based techniques unique to sensor networks that can be used to prioritize 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 [GG01, CGRS01]. 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.

5.5.3 Adapting Rates and Power Consumption

We saw in the previous section 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 the fill rates of queues. This adaptation is an important aspect 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 Query Scrambling [UFA98], Tukwila [IFF+99] and Eddy [AH00] dynamically reorder operators as the execution environment changes, TinyDB must react to changing conditions. Failure to adapt in TinyDB can bring the system to its knees, reducing data flow to a trickle or causing the system to severely exceed the

available power budget.

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. To measure the benefit of this adaptive backoff, we ran a simple selection query on a set of four motes as well as a single mote in isolation, varying the requested `SAMPLE PERIOD` of the query. We compared the mote in isolation with the performance of the four motes with and without adaptive backoff. In this case, the adaptive scheme reduces the rate at which transmission attempts are made when it observes high network contention. Network contention is measured by counting the number of packets snooped from neighboring nodes as well as the number of radio retransmissions required per packet. The results of this experiment are shown in Figure 5.8. The `SAMPLE PERIOD` is shown along the X axis, and the number of packets successfully delivered by all nodes in the network is shown on the Y axis. Each point is the average of 5 trials. Note that the "1 mote" line saturates the radio channel at about 8 samples per second.

Adaptive backoff turns out to be very important: as the *4 motes* line of Figure 5.8 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.

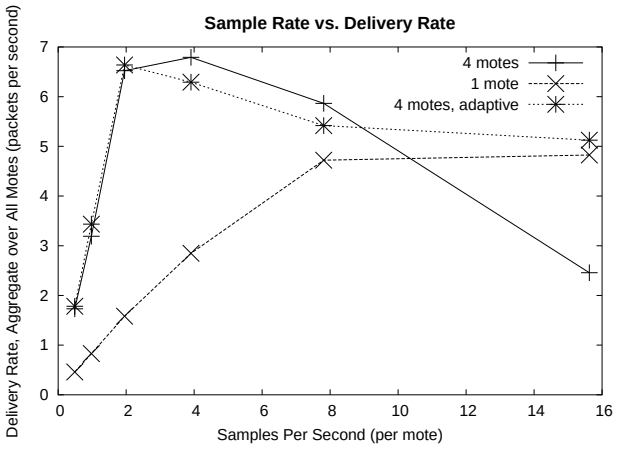


Figure 5.8: Per-mote Sample Rate Versus Aggregate Delivery Rate.

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 5.5.2 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 5.9. 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.

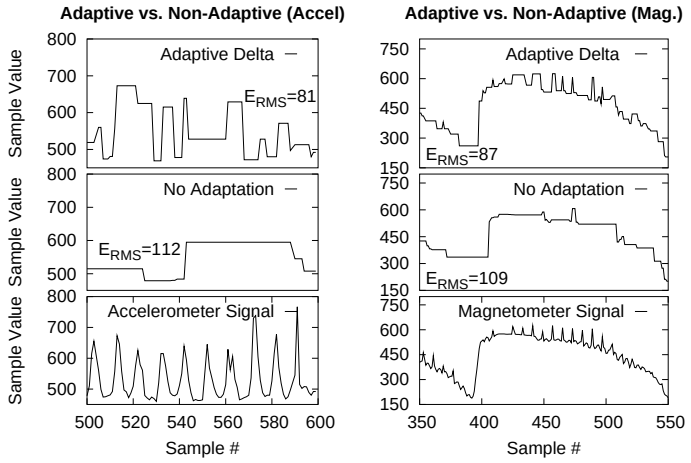


Figure 5.9: Comparison of delivered values (bottom) versus actual readings for from two motes (left and right) sampling at 16 packets per second and sending simulataneously. Four motes were communicating simultaneously when this data was collected.

Measuring Power Consumption

We now turn to the problem of adapting the tuple delivery rate to meet specific lifetime requirements in response to incorrect sample rates computed at query optimization time (see Section 5.2.2). We first note that, using computations similar to those shown Section 5.2.2, 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 (see Figure 5.2 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 to results for other sensors without compromising the lifetime goals of long running monitoring queries.

Finally, we note that incorrect estimation of power consumption may also be due to incorrect estimates of the cost of various phases of query processing, or may be a result of incorrect selectivity estimation. We cover both by tuning the sample rate. As future work, we intend to explore adaptation of optimizer estimates and ordering decisions (in the spirit of other adaptive work like Kabra and Dewitt [KD98] and Eddies [AH00]) 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.)

5.6 Related Work

There has been some recent publication in the database and systems communities on query processing-like operations in sensor networks [IGE00, MFHH02, PJP01, MF02, YG02]. As mentioned in Chapter 2, 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.

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 [Kos00] 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 [CG02]. 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 [WSX+99, IB92]), but this work relies on centralized location servers for tracking recent positions of objects.

The notion of a *geographic hash table* [RKY+02] has been developed to support a goal similar to that of SRTs, though for the purposes of data storage rather than on-line data collection. The basic idea is that every data item is hashed, on an *event-type attribute*, to some location within the network. The system then uses geographic routing [KK00] to route data to the node nearest that location. Then, when a query arrives for data with that event-type, the query can be quickly directed to the node with the appropriate results (again, using geographic routing.) SRTs perform a similar function by directing queries to the narrow set of nodes with data relevant to a particular query; the difference is that in GHTs every result tuple must be stored across the network rather than locally as in our approach.

The observation that interleaving the fetching of attributes and the application of operators also arises in the context of compressed databases [CGK01], 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 [CKAK94], as well as systems for efficiently determining when an event has fired, such as [Han96] could (possibly) be useful in TinyDB.

Approximate and best effort caches [OJ02], as well as systems for online-aggregation [RRH02] and approximate [GG01] and stream query processing [MWA+03, CCC+02] 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 under-

lying continuous signal. The Aurora project [CCC+02] mentions a need for this kind of metric, but proposes no specific approaches.

5.7 Conclusions

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. This is a fundamental issue in sensor networks, and one of the primary challenges of any sensor network query processing system.

We described a number of acquisitional techniques: treating acquisition as an expensive query plan operator for optimization purposes, using semantic routing trees to locate nodes with relevant data, allowing users to use lifetimes and events to specify when samples should be acquired, and using prioritization schemes to select particularly useful samples to bring out of the network. Together, these techniques dramatically improve the energy efficiency and usefulness of TinyDB.

In the past two chapters, we have shown how declarative queries can be distributed and efficiently executed over sensor networks. The in-network approach described in Chapter 4 can provide an order of magnitude reduction in bandwidth consumption over approaches where data is processed centrally. The declarative query interface allows end-users to take advantage of this benefit for a wide range of a operations without having to modify low-level code or confront the difficulties of topology construction, data routing, loss tolerance, or distributed computing. Furthermore, this interface is tightly integrated with the network, enabling transparent optimizations that further decrease message costs and improve tolerance to failure and loss.

We believe that the acquisitional aspects of query processing discussed in this chapter, combined with the efficient aggregation techniques we just discussed in the previous chapter, 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.

Finally, it is important to note that the users of these new deployments are often end-users who lack fluency in embedded software development but who are interested in using sensor networks to enrich their lives or further their research. For such users, high-level programming interfaces are a necessity; these interfaces must balance simplicity, expressiveness, and efficiency in order to meet data collection and battery lifetime requirements. Given this balance, we see TinyDB as a very promising service for data collection: the simplicity of declarative queries, combined with the ability of efficiently optimize and execute them makes it a good choice for a wide range of sensor network data processing situations.

Chapter 6

Initial Deployment of TinyDB

In this chapter we discuss a "real world" deployment of TinyDB in the UC Berkeley botanical garden. We study the query workload and show that TinyDB provides good power management and reliable communication delivery. We also identify some of its shortcomings and areas where future development effort is needed.

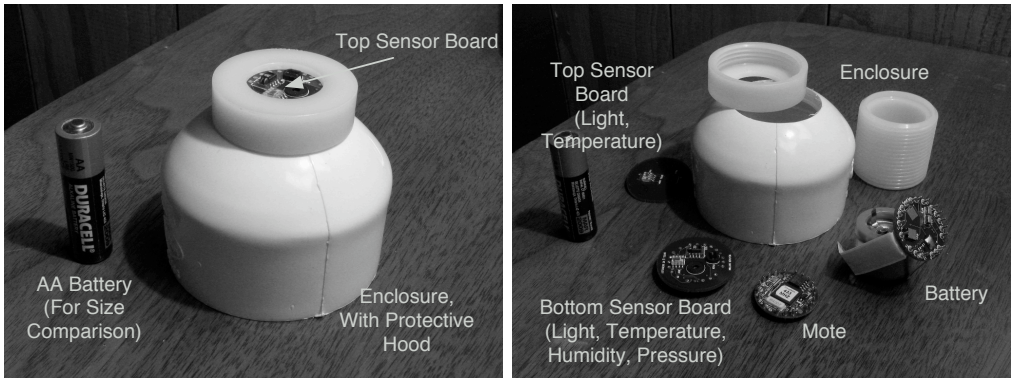
6.1 Berkeley Botanical Garden Deployment

During June and July of 2003, we began a deployment of the TinyDB software in the Berkeley Botanical Garden, located just East of the main UC Berkeley Campus. The purpose of this deployment was to monitor the environmental conditions in and around Coastal Redwood trees (the *microclimate*) in the Garden's redwood grove.

This grove consists of several hundred new-growth redwoods. Botanists at UC Berkeley [Daw98] are actively studying these microclimates, with a particular interest in the role that the trees have in regulating and controlling their environment.

The initial sensor deployment in the garden consists of 16 Mica2 sensors on a single 36m redwood tree. Each mote is equipped with a weather board that provides light, temperature, humidity, solar radiation, photosynthetically active radiation, and air pressure readings. The processor and battery are placed in a water-tight PVC enclosure, with the sensors exposed on the outside of the

enclosure. A loose fitting hood covers the bottom of the sensor to protect humidity and light sensors from rain. The light and radiation sensors on the top of the assembly are sealed against moisture and thus remain exposed. Figure 6.1 shows two photographs of a mote; in Figure 6.1(a), the mote is packaged within its enclosure. Figure 6.1(b) shows the mote disassembled, with the various pieces labeled.



(a) Assembled Mote (a) Disassembled Mote

Figure 6.1: Two photographs of the Mica2Dot mote assembly deployed in the Berkeley Botanical Garden, next to a AA battery for size-comparison.

Figure 6.2 shows the placement of the sensors within the tree, which is located in the center of the grove. Three clusters of four sensors were placed at three different altitudes within the tree; within each cluster, sensors were placed in the cardinal compass directions about 2 feet from the center of the tree. The remaining four sensors were placed along the tree trunk, from the top down, with a few meters of space between each of them. Table 6.1 summarizes the locations of each of the sensors; they are also shown (roughly) in Figure 6.2. The top two sensors (15 and 16) were placed above the canopy, such that they are exposed to the sky.

Sensors on the tree run a simple selection query which retrieves a full set of sensor readings every 30 seconds and sends them towards the basestation, which is attached to an antenna on the roof of a nearby field station, about 150 feet from the tree. The field station is connected to the Internet, so, from there, results are easily logged into a PostgreSQL database for analysis and observation.

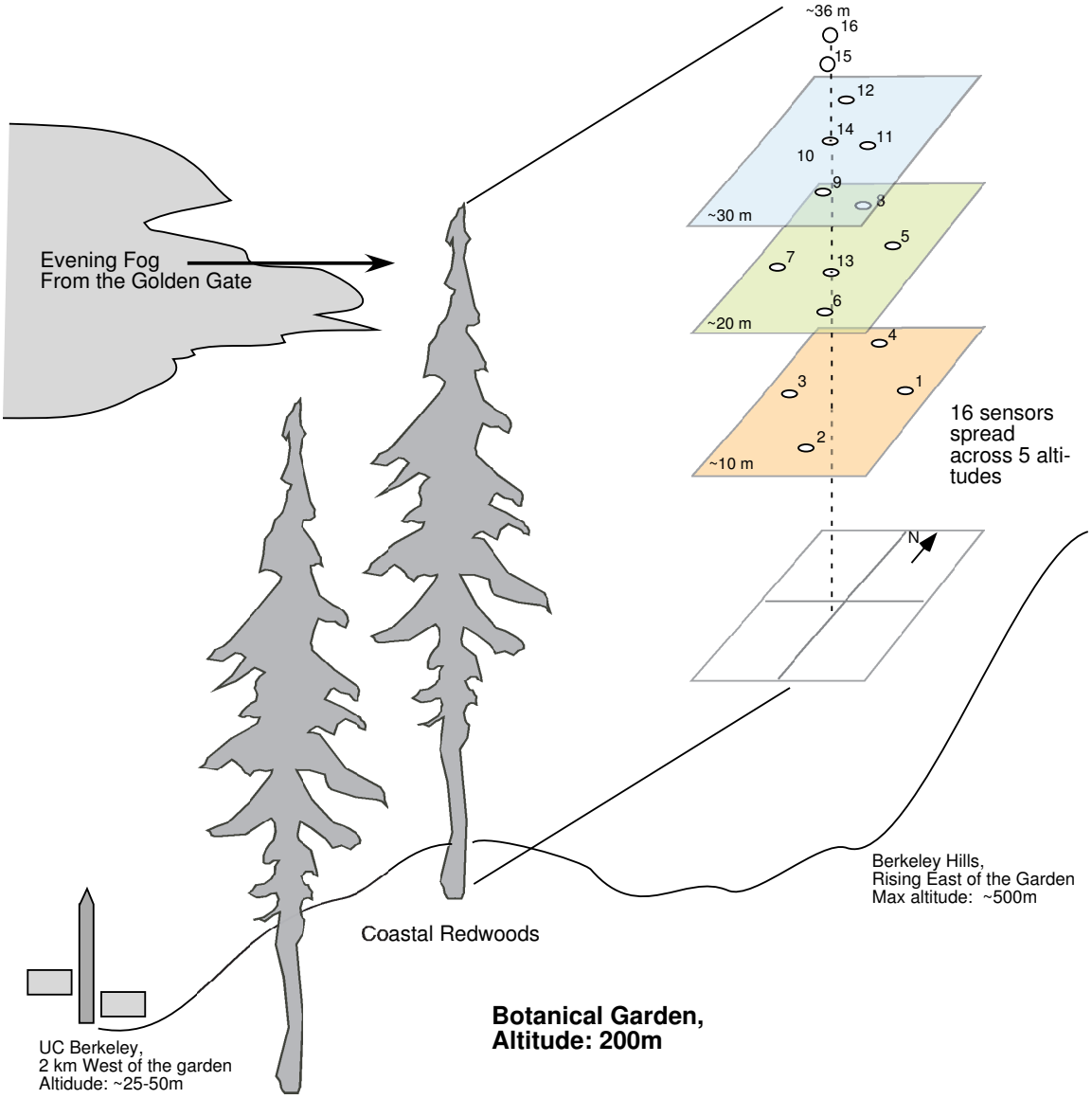


Figure 6.2: Diagram showing the placement of sensors on a Coastal Redwood in the Berkeley Botanical Garden

Sensor Number	Height	Distance and °CW from North
1	10m	.3m @ 90°
2	10m	.3m @ 180°
3	10m	.3m @ 270°
4	10m	.3m @ 0°
5	20m	.3m @ 90°
6	20m	.3m @ 180°
7	20m	.3m @ 270°
8	20m	.3m @ 0°
9	30m	.3m @ 90°
10	30m	.3m @ 180°
11	30m	.3m @ 270°
12	30m	.3m @ 0°
13	20m	.1m @ 0°
14	30m	.1m @ 0°
15	34m	.1m @ 0°
16	35m	.1m @ 0°

Table 6.1: Placement of Sensors In Instrumented Redwood

As of this writing, the sensors have been running continuously for about three weeks.

6.2 Measurements and High-Level Results

Figure 6.3 shows data from five of the sensors collected during the first few days of August, 2003. The periodic bumps in the graph correspond to daytime readings; at night, the temperature drops significantly and humidity becomes very high as fog rolls in. August 2nd and 3rd were relatively cool (below 21°C) and likely overcast as are many summer days in Berkeley; the 4th was sunny and somewhat warmer, reaching as high as 24° at the tops of the tree. Note that during the daytime, it can be as much as 8° degrees cooler and 25% more humid at the bottom of the canopy (sensor 1) than at the top (sensors 15 and 16). On hot days, this effect is particularly pronounced; compare, for example, sensors 1 and 15 just after 10am on the 4th. Anyone who has ever been walking in a redwood forest has felt this cool dampness under the cover of these trees.

This increase in humidity and decrease in temperature is one of the effects the biologists involved in the project are interested in studying. In this case, the trees act as a buffer, releasing water

into the air on hot days and absorbing it on foggy days. Looking closely at the line for sensor ID 1, it is apparent that not only is the mote cooler and wetter during the day, but at night it is somewhat warmer and less humid than the exposed sensors near the treetop. Furthermore, the rate of change of humidity and temperature at the bottom of the tree is substantially less than at the top – Sensor 1 is the last sensor to begin to warm up *and* the last sensor to cool off.

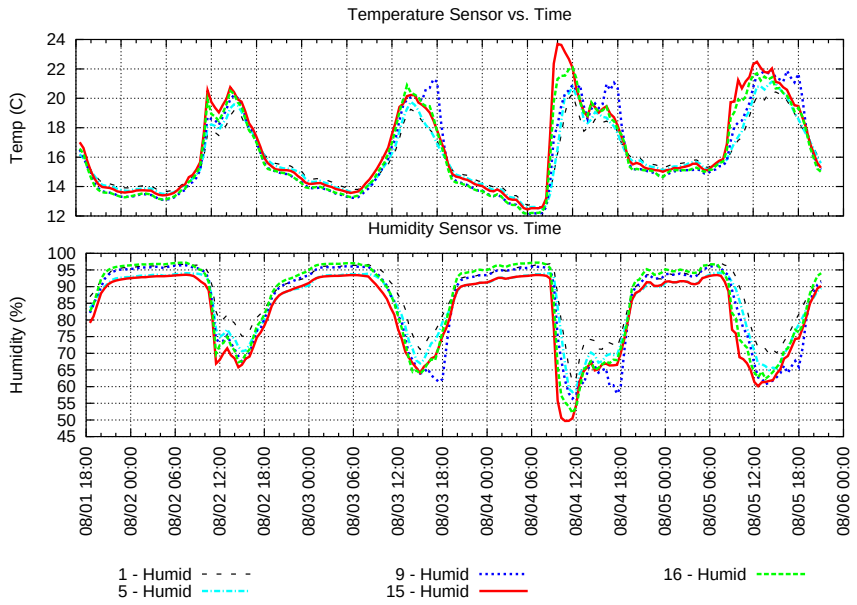


Figure 6.3: Temperature (top) and Humidity (bottom) Readings from 5 Sensors in the Berkeley Botanical Garden

There are some other interesting effects to be seen in Figure 6.3. For example, around 6PM on August 3rd, 4th, and 5th sensor 9 became much warmer than the other sensors in the tree. This effect turns out to be due to increased sunlight on this portion of the tree: apparently, the sun was able to shine into the canopy at just the right angle, illuminating the sensor for about an hour.

To illustrate this, Figure 6.4 shows the light readings from the photosynthetically active radiation (PAR) sensor on this same set of motes. The PAR sensor measures light in the 400 to 600nm range of the spectrum that plants are most sensitive to. Notice that the amount of light hitting sensor 9 is

considerably higher than that hitting the other sensors during this time.

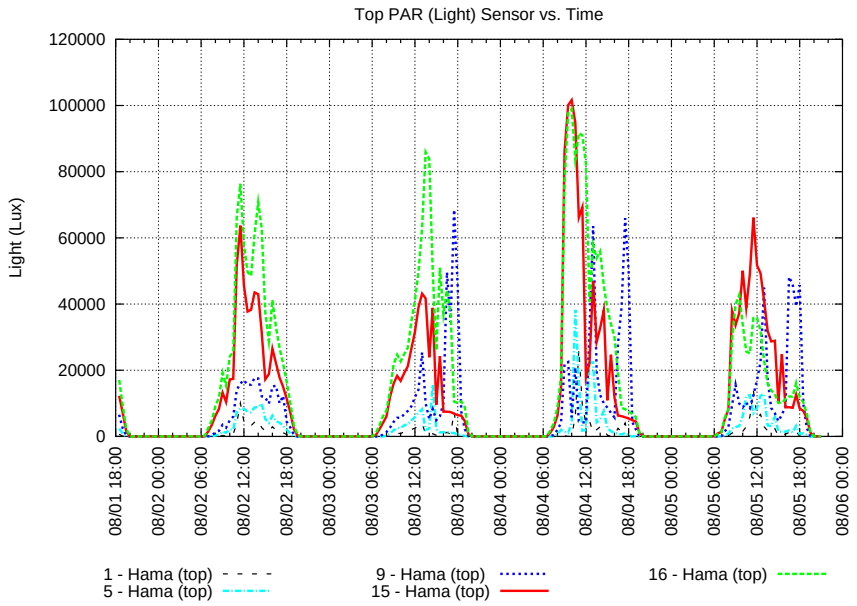


Figure 6.4: Photosynthetically Active Radiation Readings from Sensors in the Instrumented Tree

6.3 Loss Rates and Network Topology

In this section, we study the loss rates and network topology of the deployment. We note that, in this deployment, no effort is made to retransmit packets – if a transmission fails, that data is lost. The primary reason for this is a limitation in the current TinyOS network layer that disables retransmission. Fortunately, in this case, the biologists we are working with only need data every 5-10 minutes, so that we can afford to lose a relatively high percentage of the results.

Figure 6.5 shows the number and percentage of results received from each node, with a break-down by parent in the routing tree. Sensors are along the X axis, and the packet count is along the Y axis. Shading indicates the portion of each sensor's packets that were forwarded by a given parent. Parent ID 0 is the basestation – note that the majority of readings for most sensors are transmitted

to this node in a single hop. Only sensors 6,7,9, 11, and 14 transmit a significant portion of their packets via multiple radio hops. It is interesting to note that these are sensors in the middle of the tree – nodes at the bottom and very top seem to be better connected.

In this case, the best sensor (ID 2) successfully transmitted about 75% of its data. Except for sensor 3 (which failed), the worst performing sensor, 14, successfully transmitted 22% of its results. Losses are uniformly distributed across time and do not appear to be correlated across nodes.

We believe that the primary cause of loss is network collisions. Despite the fact the each node transmits infrequently, time synchronization causes sensors wake up and transmit during exactly the same 1s communication interval. Even without multihop communication, this means that at least 16 packets are sent during the same second, which is difficult for the TinyOS MAC layer to handle. The problem of automatically selecting the appropriate waking interval is an interesting area for future work.

6.4 Deployment Plans and Expected Long-Term Behavior

Using the simple measurements and analytical model described in Section 3.6.3 and given that we are sampling once every 30 seconds using 850 mAh batteries, we expect each mote in the garden deployment to last about forty days before running out of energy. Our goal is to eventually instrument five trees over the next several months; to meet this goal, we will need to deploy additional nodes or replace batteries as existing batteries run out of energy. The query-sharing features of TinyDB make such incremental re-deployment extremely easy.

Looking at the network connectivity shown in Figure 6.5, it is clear that some motes, such as 8 and 12, are forwarding more messages than others. This suggests that they will run out of energy before other nodes, decreasing the ability of weakly connected motes, like 7,11, and 14, to get their data out of the network. However, given that each of these weakly-connected devices has forwarded

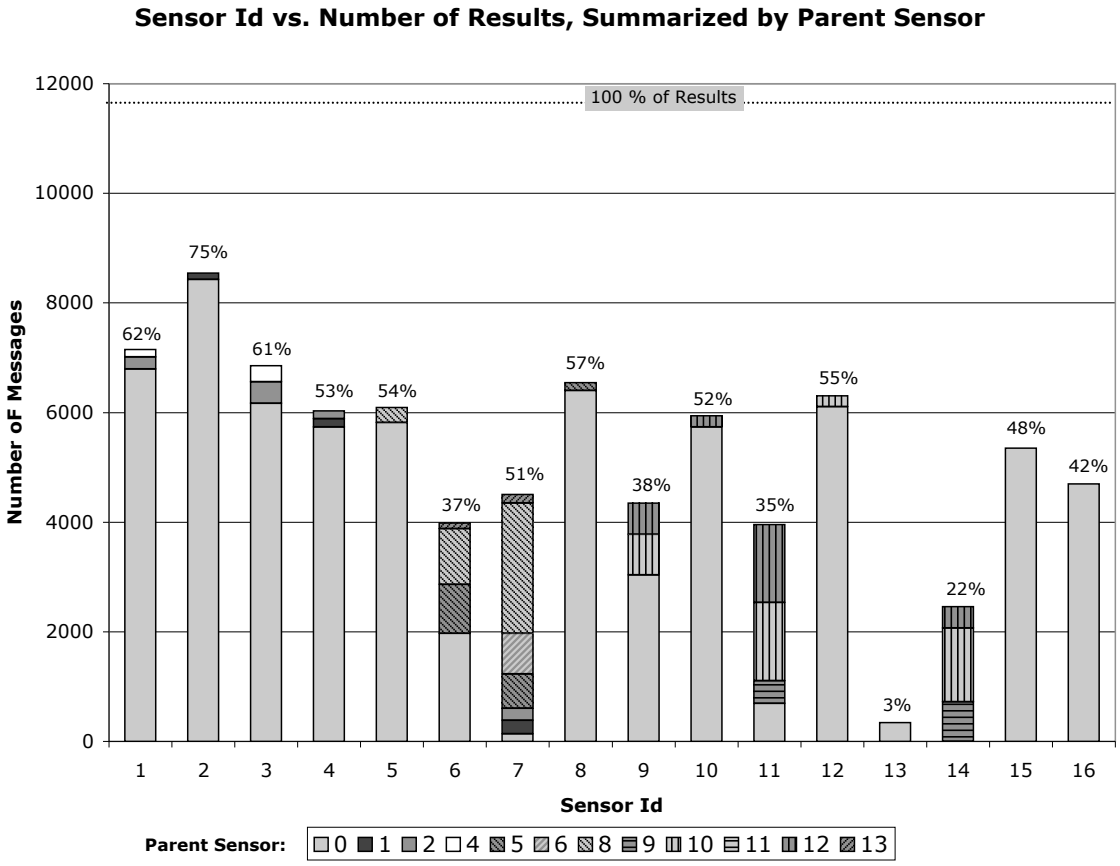


Figure 6.5: Graph showing the number of results received from every sensor, broken down by parent in the routing tree.

through several different parents, we expect that the topology will be able to adapt to failures of commonly used parents. Interestingly, contrary to the linear decay predicted by the single-node lifetime modeled by Figure 5.2, such adaptation could require remaining nodes to increase the amount of forwarding they do, causing them increase their rate of energy-consumption at the end of their lifetime.

This implies that techniques for parent selection and topology construction that take into account remaining energy levels of candidate parents may prove useful in extending the longevity of deployments like this one. They should allow the network to balance energy consumption, avoiding non-linearities by ensuring that parents run out of energy at approximately the same time. Aslam

et. al [JAR03] propose such a model in their work on power-aware routing, as does work on heterogeneous sensor networks [Ore03] from Intel Research, Portland.

6.5 Lessons and Shortcomings

As the deployment has not yet concluded, it is too early to draw real conclusions. We have, however, already made a number of observations and learned a few lessons. Thus far, the deployment has been largely successful. We were able to program the sensors to begin data collection in just a few minutes – far and away the most time consuming aspects of the deployment involved the packaging of the devices, obtaining access to the garden and permissions to instrument the trees, and climbing the tree and placing sensors within it.

Compared to other TinyOS data collection applications, this is a significant improvement. For example, the software development for the Great Duck Island application took several weeks, excluding the several days time to write the code interface to the low-level sensor hardware, since that time would have been necessary with our approach as well. Furthermore, TinyDB has a number of advantages over the GDI applications: the GDI application does not support any reconfigurability and, as discussed in Chapter 3, it uses a low-power listening based power management protocol that is significantly less power-efficient than our scheduled communication approach.

There were some difficulties involved in the set-up and execution of TinyDB. Many of these issues are simply evidence of that fact the entire TinyOS software stack, including TinyDB, is still research quality. For example: the mote-to-Java interface, called SerialForwarder, repeatedly crashed or dropped bytes, falling out-of-sync with the sending application. The Windows XP based laptop where results are being logged crashed several times under a heavy Postgres query load. On one occasion, the TinyDB software began forwarding a query endlessly, with neighboring motes bouncing query messages between each other in a loop that did not end until they were powered off.

This initial deployment does not exercise any of the in-network aggregation or storage features of TinyDB. The primary reason for this is that the scientific users we are working with are interested in collecting all of the data from the network, so that they can generate plots and verify that the data is in fact correct.

Sensor 13 stopped returning results about two days into the deployment. Though we have not retrieved the sensor to physically examine it, we believe that it was physically damaged in some way – either by moisture seeping into the sealed main compartment, which can happen when the O-rings around the sensor boards are not properly seated, or by one of the battery or antenna connections coming lose.

Sensor 3's humidity and one of its temperature sensors are reading obviously incorrect values. The voltage reading from the sensor is unusually low (about 1.8V) – so low that the device should no longer be functioning, which suggests a faulty voltage detection circuit. Since part of the calibration of these sensors involves normalizing for the battery voltage, this is the likely explanation of this effect.

In future versions of this deployment, we hope to move to an approach where we log all results to the EEPROM of the devices and then just transmit summaries of that data out of the network, only dumping the EEPROM on demand or during periods where there is no other interesting activity (e.g., at night.) Once scientists believe that our hardware and software functions correctly, we believe they will be more likely to accept this style of approach.

We did find that the ability to reconfigure running queries and control sensors remotely was very useful. For example, as we were initially injecting queries, we adjusted query rates and started and stopped the running query several times. During the time the sensors have been on the trees, we have successfully reset several misbehaving sensors. After reset, these sensors downloaded the running query from their neighbors and began correctly reporting results.

We believe that future deployments will make use of automatic reconfigurability via events. Looking at the data we have collected thus far, changes are generally quite slow, such that a `SAMPLE PERIOD` of 5 or 10 minutes would be sufficient to characterize the dynamics of the forest. However, there are brief periods where sudden, interesting changes occur; the sunlight hitting Sensor 9 in Figure 6.3 is an example of such a situation. In such periods of rapid change, event-based queries that run for a short period of time and collect data at a high rate will prove useful and allow the network to automatically balance the need to conserve energy by using a low sample rate most of the time with the ability to monitor rapid changes in the ecosystem.

6.6 Conclusions

This concludes our discussion of the initial deployment of TinyDB in the Berkeley Botanical Garden. We believe the fact that we have successfully deployed a working, long running instance of our software is testament to the viability of our approach.

In the next chapter, we turn to other possible applications of the TinyDB framework, discussing in particular several applications seen as important to the general field of sensor network research.

Chapter 7

Advanced Aggregate Queries

Initial feedback from the sensor network community indicates that TinyDB's SQL-based interface is very attractive to a number of users interested in distributed sensing. However, we have also heard concerns about apparent limits to the functionality of simple SQL queries. In this chapter, We show that it is possible to deploy more complex sensing tasks in TinyDB. Our intention is both to illustrate TinyDB's potential as a vehicle for complex sensing algorithms, and to highlight some of the unique features and constraints of embedding these sensing algorithms in an extensible, declarative query framework.

In this chapter, we revisit the TAG framework from Chapter 4, and show how it can be used to implement two sensing applications that are relatively distant from vanilla database queries: distributed mapping via isobars and vehicle tracking.

7.1 Isobar Mapping

In this section, we explore the problem of building a topographic (contour) map of a space populated by sensors. Such maps provide an important way to visualize sensor fields, and have applications in a variety of biological and environmental monitoring scenarios [Est]. We show how TinyDB's aggregation framework can be leveraged to build such maps. Conceptually, the problem is similar to that of computing a `GROUP BY` over both space and quantized sensor readings – that is, our

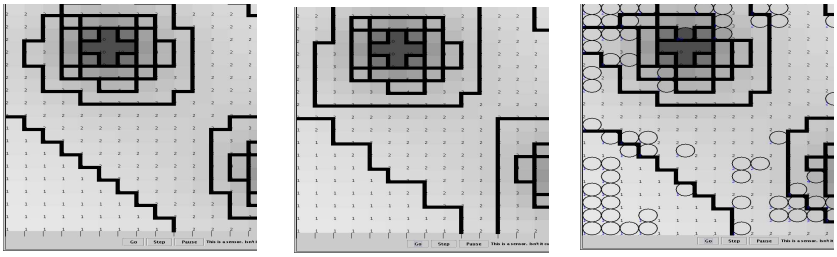
algorithms partition sensors into *isobars* that are contiguous in space and approximately equal in sensor value. Using in-network aggregation, the storage and communication costs for producing a topographic map are substantially less than the cost of collecting individual sensor readings and building the map centrally. We discuss three algorithms for map-building: a centralized, *naive* approach, an exact, *in-network* approach, and an approximate, *lossy* approach. The general process to build a topological map is as follows: each sensor builds a small representation of its local area, and sends that map to its parent, where it is combined with the maps from neighbors and ancestors and eventually becomes part of a complete map of the space at the root of the tree.

To support topographic operations on sensors, we require a few (very) simple geometric operators and primitives. To determine adjacency in our maps, we impose a rectangular grid onto the sensors, and assign every sensor into a cell in that grid. Our goal is to construct isobars, which are orthogonal polygons with holes; we need basic operations to determine if two such polygons overlap and to find their union. Such operations can be performed on any polygon in $nlog(n)$ time (where n is the number of edges in the polygon) using the Leonov-Nitkin algorithm [LN97]. There are a number of free libraries which implement such functionality.

We begin with a discussion of the three algorithms, assuming that every cell in the grid is occupied. We return to mapping sparse grids in Section 7.1.4.

7.1.1 Naive Algorithm

In the naive algorithm, we run an aggregate-free query which returns the location and attribute value of all of the sensors in the network; these results are combined via code outside the network to produce a map. We implemented this approach in a simulation and visualization, as shown in figure Figure 7.1(a). In this first simulation, sensors were arranged in a grid, and could communicate losslessly with their immediate neighbors. The isobars were aggregated by the node at the center of the network. The network consisted of 400 nodes in a depth 10 routing tree. In the screenshot, the



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

Figure 7.1: Screenshots of a visualization of isobars imposed on a grid of sensors. Each cell represents a sensor, the intensity of the background color indicates sensor value, and black lines frame isobars. See text for a description of the individual figures.

saturation of each grid cell indicates the sensor value, and the thick black lines show isobars.

7.1.2 In-Network Algorithm

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

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

where k defines the width (in attribute-space) of each of the isobars. Following the notation of Section 4.2, we can then define the three aggregation functions, i, f, and e, as follows:

- i: The initialization function takes an `xloc, yloc`, and `attr`, and generates as a partial state record the singleton set containing an isobar with the specified `attr` value and a container polygon corresponding to the grid cell of the sensor.

- f: The merging function combines two sets of isobars, I_1 and I_2 into a new isobar set, I_3, where each element of I_3 is a disjoint polygon that is the union of one or more polygons from I_1 and I_2. This new set may have several non-contiguous isobars with the same attribute value. Conversely, merging can cause such disjoint isobars in I_1 to be joined when an isobar from I_2 connects them (and vice-versa.) Figure 7.2 shows an example of this happening as two isobar sets are merged.

- e: The evaluation function generates a topographic map of contiguous isobars, each labeled with their attribute value. The final answer produced by the evaluation function is the same as the answer produced by the naive algorithm, but at a substantially reduced communication cost.

7.1.3 Lossy Algorithm

The lossy algorithm works similarly to the in-network algorithm, except that the number of vertices v used to define the bounding polygon of each isobar is limited by a parameter of the aggregate. This reduces the communication cost of the approach, but makes it possible for isobars to overlap, as they will no longer perfectly trace out the edges of the contours.

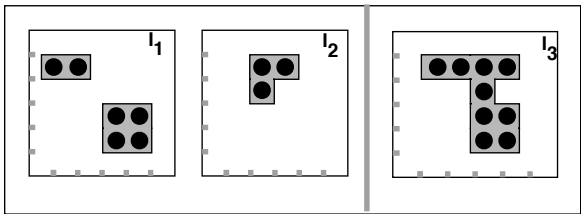


Figure 7.2: Two isobar sets, I_1 (with two elements) and I_2 (with one element) being merged into a new isobar set, I_3 (also with one element).

In the lossy algorithm, i is the same as in the in-network case. For f, we compute I_3 as above, but we do not use it as the partial state record. Instead, for the containing polygon p in each set of I_3, we compute a bounding box, b, and then take from b a number of maximally sized rectangular "cuts" that do not overlap p. We continue taking cuts until either b contains v vertices, or the next cut produces a polygon with more than v vertices. We omit the details of how we compute maximal cuts; because our polygons are orthogonal, this can be done via a scan of the vertices of p. We use these cut-bounding-boxes as approximations of the containing polygons in the isobars of the PSRs resulting from our merge function. Figure 7.3 shows a containing polygon approximated by a bounding rectangle with a single cut.

In the lossy evaluation function e, one or more isobars in the final aggregate state record may

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

We simulated this lossy algorithm for the same sensor value distribution as was shown in Figure 7.1(a), using a maximum of 4 "cuts" per isobar. The results are shown in Figure 7.1(b); notice that the basic shape of the isobars is preserved.

We compared the total amount of data transmitted by our simulation of the lossy, in-network, and naive algorithms for the isobars shown in Figure 7.1(a) and 7.1(b), and found that the naive algorithm used a factor of four more communication than the lossy algorithm and about 40% more communication than the in-network algorithm.

7.1.4 Sparse Grids

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

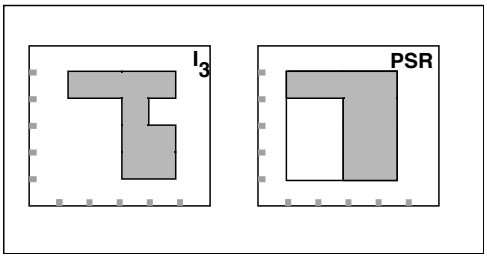


Figure 7.3: A lossy approximation of a containing polygon (I_3) as a bounding box with a single cut (PSR).

A similar situation arises in dense topologies with network loss, when some sensor values are not be reported during a particular epoch. We implemented this sparse grid approach and used it to visualize isobars with a high-loss radio model, where the probability that two sensors can communicate with each other falls off with the distance between the sensors. For adjacent sensors, loss rates are about 5%; for sensors that are three cells away (the maximum communication range), loss rates are about 20%. The result is shown in Figure 7.1(c), with black circles on the nodes whose values were lost during the epoch being visualized. Notice that, despite the large number of losses, the shape of the isobars is largely preserved.

7.2 Vehicle Tracking

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

Target tracking via a wireless sensor network is a well-researched area [LWHS02]. There are different versions of the tracking problem with varying degrees of complexities. For ease of illustration, in our discussion we only deal with a very simple version of the tracking problem, based on the following assumptions and constraints:

- There is only a single target to track.
- The target is detected when the running average of the magnetometer sensor readings goes over a pre-defined threshold.

- The target location at any point in time is reported as the node location with the largest running average of the sensor reading at that time.
- The application expects to receive a time series of target locations from the sensor network once a target is detected.

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

There are some clear advantages to implementing tracking applications on top of TinyDB. First, TinyDB's generic query language is available as a resource, allowing applications to mix and match existing spatial-temporal aggregates and filters in a query. Applications can also run multiple queries in the sensor network at the same time, for example one tracking query and one network health monitoring query. Second, TinyDB takes care of many of sensor-network systems programming issues such as multi-hop routing, coordination of node sleeping, query and event dissemination, etc. Third, by registering tracking subroutines as user-defined aggregates in TinyDB, they become reusable in other TinyDB queries in a natural way. Fourth, we are optimistic that TinyDB's query optimization techniques [MFHH02] can benefit tracking queries. For example, each node can "snoop" the messages from its neighboring nodes and suppress its output if any neighbor has detected a stronger sensor signal.

We describe how two versions of the tracking application could be implemented in TinyDB with increasing levels of query complexity for better energy efficiency. We describe these implementations in TinyDB's SQL-like query language, though some off the language features used in this section are not available in the current TinyDB release. In all the TinyDB SQL statements, mag is a TinyDB attribute for the magnetometer reading, $time$ is an attribute that returns the current timestamp as an integer. We assume the sensor nodes are time synchronized within 1 millisecond

using protocols like [EGE02]. $nodeid$ is a TinyDB attribute for the unique identifier of each node. We assume that the target is detected when the magnetometer reading goes over a constant value, *threshold*. $winavg(10, 1, mag)$ is for the 10-sample running average for the magnetometer readings. $max2(arg1, arg2)$ is another TinyDB aggregate that returns the value of $arg2$ corresponding to the maximum value of $arg1$. $max2(avgmag, nodeid)$ is used in our implementations to find the $nodeid$ with the largest average magnetometer reading. As mentioned above, we use this to represent the location of our target and assume that the basestation is capable of mapping $nodeid$ to some spatial coordinate. $max2$ is really a place holder that can be replaced with much more sophisticated target localization aggregates. In both implementations, we need to apply $max2$ to group of values with the same timestamp. Values are grouped by $time/10$ to accommodate minor time variations between nodes.

7.2.1 The Naive Implementation

Figure 7.4 shows the TinyDB queries that implement our initial tracking application. In this implementation, each sensor node samples the magnetometer every 100 milliseconds and computes the 10-sample running average of the magnetometer readings. If the running average of magnetometer readings is over the detection threshold, the current time, nodeid and average value of the magnetometer are inserted into the storage point $running\_avg\_sp$.

Recall that storage points in TinyDB provide temporary in-network storage for query results and facilitate applications to issue nested queries. The second query in Figure 7.4 is a query that runs over the storage point $running\_avg\_sp$ every second and computes the target locations using the $max2$ aggregate.

7.2.2 The Query-Handoff Implementation

The problem with the naive implementation is that all sensor nodes must wake up and sample the magnetometer every 100 milliseconds. This is extremely power-inefficient because at any point in time, the target can only be detected by a small number of nodes assuming the sensor nodes are spread over a wide area. Thus, for a large percentage of nodes, the energy spent on waking up and sampling the magnetometer is wasted.

Ideally, we would like to start the target tracking query on a node only when the target is near it and stop the query when the target moves away. TinyDB will put a mote to sleep when there are no queries running. This means that we need a TinyDB event to trigger the tracking query.

The query-handoff implementation requires some special standalone hardware such as a motion detector that detects the possible presence of the target, interrupts the mote processor, and pulls it out of sleep mode. $target\_detected$ is the TinyDB event corresponding to this external interrupt. It is unrealistic to require this special hardware be installed with every node. However it might be feasible to only install it on a small number nodes near the possible entry points for the target (e.g. endpoints of a line of sensors along a road). These nodes will be awakened by the $target\_detected$ event and start sampling the magnetometer to determine the current target locations. At the same time, they also try to predict the possible locations the target may move to next via a custom aggregate $next\_location$, and signal a remote event $target\_approaching$ on nodes at these locations to alert them to start sampling their magnetometers and tracking the incoming target. Nodes that receive the $target\_approaching$ event will wake up and basically do the same. The $target\_approaching$ event relies on the functionality of remotely waking up a neighboring node via a special radio signal. Such functionality is available on the Berkeley Rene motes (TinyOS 0.6) or the PicoRadios described in [RAK+02b]. The TinyDB queries for this implementation is shown in Figure 7.2.2. We call this the *query-handoff* implementation because the node hands the tracking

queries off from one set of nodes to another set of nodes following the target movement.

Query handoff is probably the most unique query processing feature required by tracking applications, and one that at first we expected to provide via low-level network routing infrastructure. However, we were pleased to realize that event-based queries and storage points allow handoff to be expressed reasonably simply at the query language level. This bodes well for prototyping other application-specific communication patterns as simple queries. For example, in a workshop paper [HHMS03], we also illustrated how wavelets can be structured as an aggregate as a tool for constructing histograms of sensor values, in the spirit of [MVW98]. An ongoing question in such work will be to decide when these patterns are deserving of a more efficient, low-level implementation inside of TinyDB.

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

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

Figure 7.4: Naive Implementation

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

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

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

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

Figure 7.5: Handoff Implementation

We have now seen several applications of the TAG and TinyDB framework to more sophisticated aggregates that are arguably more applicable to the sensor network world than the standard `MIN`, `MAX`, `SUM`, `COUNT`, and `AVERAGE` provided by traditional database systems. But we have yet to show any evidence that TinyDB actually performs reasonably well in the real world, which we will do in the next section.

7.3 Related Work

Building contour maps is a frequently mentioned target application for sensor networks; see, for example, [Est], though, to our knowledge, no one has previously described a viable algorithm for constructing such maps using sensor networks. There is a large body of work on building contour maps in the image processing and segmentation literature – see [MBLS01] for an excellent overview of the state of the art in image processing. These computer vision algorithms are substantially more sophisticated than those presented here, but assume a global view where the entire image is at hand.

The query handoff implementation for the tracking application in Section 7.2 is based on the single-target tracking problem discussed in [LWHS02]. The tracking algorithms described in [LWHS02] is implemented on top of UW-API [RSWC01] which is a location-centric API for developing collaborative signal processing applications in sensor networks. UW-API is implemented on top of Directed Diffusion [IGE00] and is focused on routing data and operations based on dynamically created geographical regions.

7.4 Conclusions

We believe the work described in the previous two chapters justifies our optimistic belief that the declarative interface provided by TinyDB is the correct one for a wide-range of sensor network applications. We demonstrated several non-trivial applications outside the realm of traditional SQL queries. In the future, we hope to continue this thrust by forming additional collaborations with do-

main experts in the development of new applications; this includes both application experts outside computing (such as the biologists working with us in the Botanical Garden), and experts in other aspects of computing including collaborative signal processing and robotics. As with the Botanical Garden Deployment, our goal is for TinyDB to serve as an infrastructure that allows these experts to focus on issues within their expertise, leaving problems of data collection and movement in the hands of TinyDB's query engine.

Chapter 8

Future Work

In this chapter, we revisit some of the assumptions and limitations of the approach described in previous chapters, discussing opportunities for future work that we hope to pursue in the future.

8.1 Adaptivity

In the context of query processing, adaptivity usually refers to the modification or re-building of query plans based on runtime observations about the performance of the system. Although the current TinyDB contains some adaptive query processing features, further adaptivity will be required to maximize the longevity and utility of sensor networks; examples of adaptation opportunities include:

- *Query Reoptimization*: Traditional query optimization, where queries are optimized before they are executed, is not likely to be a good strategy for long running continuous queries, as noted in [MSHR02]. This is because statistics used to order operators may change significantly over the life of a query, as we discuss below.

- *Operator Migration*: Operators in a query plan may be placed at every node in the network, or at only a few select nodes. Choosing which nodes should run particular operators is tricky: for example, a computationally expensive user-defined selection function might significantly reduce the quantity of data output from a node. Conventional wisdom suggests that this filter

should be propagated to every node in the network. However, if such a filter is expensive enough, running it may consume more energy than would be saved from the reduction in communication, so it may be preferable to run the filter at only a few select nodes (or at the powered basestation). Making the proper choice between these alternatives depends on factors such as the local selectivity of the filter, the number of hops which non-filtered data must travel before being filtered, processor speed and remaining energy capacity.

- *Topology Adjustment*: As discussed briefly in Section 2.3.3, the system needs to adapt the network topology. Existing sensor-network systems generally perform this adaptation based on low-level observations of network characteristics, such as the quality of links to various neighboring nodes [WC01], but a declarative query processor can use query semantics to shape the network. We saw a simple example of this when we discussed *semantic routing trees* in Section 5.4 where the proper choice of parent in a routing tree was shown to have a dramatic effect on communication efficiency. There are, however, other situations where this topology adaptation would be beneficial – for example, when computing grouped aggregates, the maximum benefit from in-network aggregation will be obtained when nodes in the same group are also in the same routing sub-tree.

8.1.1 Query Reoptimization

In this section, we focus more on the first kind of adaptivity discussed above: reoptimizing a running query plan over time. As a simple example of a situation where the network's lack of up-to-date knowledge about the costs or selectivities of various operators consider the long running query:

```
SELECT nodeid, light
 FROM sensors
 WHERE temp > 80 ° F
 AND expensive_predicate
 SAMPLE PERIOD 2s
```

On a hot summer day, the `WHERE` clause will not be particularly selective; at night, however, the

temperature may always fall below 80°, such that the predicate will be very selective. During the day, we might prefer not to acquire the temperature reading until after testing the expensive predicate; at night, however, we would certainly want to measure the temperature first.This is an example where the system would benefit from adaptive query processing by adjusting the order of operators in the query plan over time to account for these changes in selectivity.

There have been several proposals in the database literature for implementing adaptivity [UF00, AH00, Ive02, UFA98] – we briefly discuss eddies [AH00]. Our research on Continuous Adaptive Continuous Queries [MSHR02] discusses adapting the eddy framework for continuous queries that are similar to those run in TinyDB. The basic idea behind the eddy-based approach is as follows: for every tuple (or every n tuples), reorder operators based on recent observations of selectivities. This approach can be more efficient than re-running a full-blown optimizer on every tuple because it is possible to track selectivities over time in a lightweight fashion, effectively amortizing the cost of optimization over many tuples.

There are a number of cases where an eddy-like adaptive query processor would be useful in TinyDB. We gave the basic example of operators whose selectivities change slowly over time above. Other arguments in favor of adaptivity in sensor networks include:

- **Lack of Statistics**: Because a sensor network can be quite large and the costs of bringing optimizer statistics out of every node can be prohibitive, one advantage of an adaptive approach is that it allows each node to locally choose its own plan, without the benefit of a global optimizer.

- **Cost of Optimization**: As we discussed, the eddy-based adaptive approach amortizes expensive query optimization over the processing of many tuples. In a conventional high-throughput query processor, the per-tuple performance costs of eddies can significantly impair system throughput [MSHR02], but in a sensor network, where throughput is not the primary con-

cern, this small overhead is likely to be acceptable.

- **Changes in Rates and Burstiness**: One of the primary advantages of fine-grained adaptivity provided by an approach like eddies is that it makes it possible to adapt to highly variable, bursty rates and sensor-value distributions. Some attributes in a sensor network will have such a highly variable character: acceleration, magnetometer, and light signals, for instance, can change very rapidly when a mote is moved or covered.

- **Exemplary Aggregate Pushdown**: Finally, there are some optimizations made possible only when the optimizer can learn and adapt as a query runs: for example, the exemplary aggregate pushdown technique we presented in Chapter 5 is only effective when the cost of the expensive predicate is known or can be determined and the selectivity of the exemplary aggregate can be accurately estimated – both of these quantities are hard to estimate statically but easy to observe in an eddy-like environment.

As of this writing, we are in the process of evaluating the advantages that an eddy-like query processor can provide in a sensor network, focusing in particular on new optimizations like exemplary aggregate pushdown. We expect to begin by implementing a simple form of adaptivity: sending a set of candidate plans into the network and selecting one of these plans on every successive tuple.

Beyond adaptive query processing, there are a number of other features we are planning to implement in TinyDB. One such feature is adding a notion of *answer quality* to queries and query answers.

8.2 Accuracy and Precision In Query Processing

In the current version of TinyDB, there is no way for the user to express preferences about answer *quality*. By *quality*, we mean any of a variety of factors, including the number of devices involved

in an aggregate, the precision (in bits, for example) or accuracy (e.g., in PPM or percent) of the various sensing hardware, or the maximum acceptable loss rate of the network.

We intend to extend future versions of TinyDB with simple features for controlling answer quality via annotations to queries. For example, if the user wanted a 95% confidence in an average reading, she might write:

```
SELECT AVERAGE(temp)
 FROM sensors
 CONFIDENCE 95% WITHIN 1.0
```

This would cause the system to collect temperature readings until its confidence is greater than 95% that the answer is within 1.0 of the actual temperature. The details of how this will be implemented remain unsolved – essentially, some mechanism is needed whereby the system can sample the sensors uniformly and at random until the answer confidence is above 95%. A conceptual reference for the issues related to using statistical sampling to estimate the value of attributes in databases can be found in [HOT88].

A similar model, called "Information Driven Sensor Querying (IDSQ)" [CHZ02] has been proposed by researchers at PARC. IDSQ was proposed in the context of vehicle tracking, and the basic idea is to choose sensors to sample from to minimize uncertainty about the position of the vehicle. This is done by electing a leader, and having the leader task neighboring sensors in turn until the size of the uncertainty region (which similar to a confidence interval) is as small as desired.

Our tree based query processing model is somewhat at odds with this leader based tasking approach, but the basic idea – sampling sensors to minimize the size of a confidence interval – gets at the spirit of what we are hoping to do with respect to incorporating a notion of answer quality into TinyDB.

A quality-based approach is important to users for two reasons: first, it provides definite guarantees about query answers, which the current implementation of TinyDB does not. Second, it can significantly reduce the amount of data collection which the network must do, since only a sufficient

number of samples to satisfy the user's quality goal must be collected.

8.3 Other Challenges

Adaptivity and answer quality are the biggest open research problems for systems like TinyDB. There are, however, a number of implementation tasks and smaller challenges that need to be addressed as the system matures. In this section, we briefly describe some features that we believe will prove to fruitful areas of research in the context of declarative queries for sensor networks.

- **Sharing for Multiple Queries.** Though the TinyDB system supports multiple queries running simultaneously on a single device, very little processing is shared between the execution of those queries. There has been substantial research on this topic in the context of traditional database systems [CDTW00, MSHR02, RSSB00, MRSR01], but it is not clear that current generation sensor hardware has sufficient memory or a fast enough processor to directly apply these techniques.

- **Multiple Base Stations.** Another limitation of the current TinyDB implementation is that all of the results for a particular query are delivered to a single end-point (i.e., routing-tree root.) In the future, it will be important to be able to partition results amongst multiple base-stations.

- **Advanced Storage Management.** The `STORAGE POINT` features we described in Section 3.2 are an initial implementation of in-network storage using a declarative syntax. In the current implementation of TinyDB, however, these storage points are allocated locally at the node where the query was issued. Instead, the system should be able dynamically and adaptively place storage based on the workload and available resources in the network.

  We have begun to sketch out ways in which these advanced storage management features can be built into TinyDB. One alternative would be to use a GHT-like [RKY+02] model

for data storage, though the random-hashing approach of GHTs does not offer good locality properties, something that will be a severe problem in sensor networks. Ghose et al. [GGC03] propose a replication approach for GHTs that can help to mitigate access overhead associated with random hashing.

- **Exception Handling.** The current implementation of TinyDB suffers from the fact that relatively little status or error information is exchanged between the basestation and the sensor network. This can make it hard to understand what happened when something goes wrong, and makes it hard to tell if a query has been successfully disseminated or if all query results have been collected. Error handling presents a tradeoff: investing significant energy in bringing status information out of the network is wasteful of energy and bandwidth, but can improve the usability of the network. Exploring this tradeoff and intelligently propagating informative exceptions is an important issue for sensor networks.

- **Calibration and Conversion.** In the current implementation of TinyDB, calibration – the process of converting raw readings from the analog-to-digital converter (ADC) into real-world units like degrees Celsius – is done by the low-level attributes or simply applied once data has been retrieved from the network. Calibrating in-network, at the attributes, is preferable, since it allows users to specify predicates over calibrated attribute values rather than over raw readings from the analog-to-digital converted (ADC). However, it can be relatively costly to calibrate readings, especially on embedded processors like the Atmel devices used in motes, since they lack a floating point unit. Thus, choosing when to apply calibration in-network is an important area of future work.

- **Nested Queries, Many to Many Communication, and Other Distributed Programming Primitives.** TinyDB supports a limited form of nested queries through `STORAGE POINTS`

– queries can be executed over the logged data in these buffers, thus providing a simple form of nesting. However, more complex, *correlated queries* that express fairly natural operations like "find all the sensors whose temperature is more than 2 standard deviations from the average" cannot be executed efficiently within the network. The reason for this is that this is really a nested query consisting of an inner query that computes the average temperature and an outer query that compares every node's temperature to this average. To execute this query in-network, some mechanism for propagating the average to the sensors is needed; understanding whether many-to-many communication primitives (like GPSR [KK00]) are adequate for this task and provide any benefit over simply computing the answer externally is an important topic of future research

- **Integration with Other Query Processing Engines.** Our original vision for TinyDB, as sketched in our early work on Fjords [MF02] was as a small part of a larger federated database system that would combine readings from the sensor network with data from data sources such as the web and conventional, relational database engines. Following through on this vision requires addressing a number of problems regarding how the federating system learns the capabilities of TinyDB (and other services) and how it decides what operations should best be pushed down into the various federated systems.

- **Actuation and Event Detection.** TinyDB currently includes basic support for actuating hardware when a query result is produced; the user may specify an "output action" that invokes a named command on the device instead of transmitting results out of the network. For example, the query::

```
SELECT nodeid,light
 WHERE light > 10000 lux
 OUTPUT ACTION soundAlarm(10s)
```

causes the `soundAlarm` command to be invoked whenever the light levels on a node go

above the specified threshold. This is an example of local, per-device actuation; one could instead imagine triggering such actions on other nodes when a condition is satisfied, but this requires sophisticated in-network communication primitives as well as new language constructs to allow users to express these conditions.

- **Support for Other Hardware Platforms.** We have designed TinyDB to run on the Berkeley motes, though we have no reason to believe that it is not adaptable to other hardware platforms, particularly if it is possible to port TinyOS to those platforms. TinyOS makes extremely modest assumptions about the underlying hardware – e.g., that the processor provides basic features such as a hardware clock and various I/O lines to which the radio and sensors may be connected. For this reason, it should be easily portable to almost any platform; indeed, developers around the world have already ported it to the TI and ARM embedded microprocessors.

- **Heterogeneity.** Heterogeneity can arise at a number of levels: nodes can have different sensors available, may be processing different queries, can have differing amounts of remaining energy, or can have different fundamental capabilities – e.g. a faster radio, more sophisticated processor, or more storage. Some work has been done in the networking community to exploit networks of nodes with heterogeneous radio interfaces or variable amounts of available bandwidth [Ore03], but we believe that using a query driven approach can make the system much more adaptable to variations in capabilities between nodes and will be an important area for future work.

8.4 Summary

In this section, we summarized some of the most interesting and challenging areas of future work related to query processing in sensor networks. Of particular importance are notions of adaptivity

and answer quality, but we also suggested a number of other interesting problems related to resource sharing, error handling, calibration, actuation, and heterogeneity. These diverse problems illustrate the plethora of open issues related to data collection in sensor networks – a sign that this will continue to be a viable research topic for years to come.

Chapter 9

Concluding Remarks

In this dissertation we presented the TinyDB architecture for executing declarative queries over sensor networks. We argued that this declarative approach provides a considerably easier programming environment than the embedded-C like languages traditionally used to program and interact with sensor networks. We also showed that this declarative approach can be quite efficient, due to a number of sensor-network specific optimizations we have designed and implemented for TinyDB. Finally, we demonstrated that our approach is expressive enough to enable a variety of sophisticated sensor network target applications, such as distributed mapping, tracking, and environmental monitoring.

These results are further confirmed by the fact that TinyDB has begun to be deployed in the real-world – we discussed an early deployment in the Berkeley botanical garden, noting its ability to provide long-running, power-efficient, flexible data collection in a remote environment.

Moving forward, we believe the high-level, declarative approaches we have developed will become the primary way in which users interact with sensor networks. Sensor network environments are so complex and difficult to program that even sophisticated programmers find it difficult to get applications to run reliably. By allowing users to program the entire network rather than one node at a time and by relying on the system to manage and adapt to failures, communication losses, changing network topologies and data rates, the lives of programmers (and end users) is greatly simplified.

This will enable sensor networks to grow to thousands of autonomous nodes, enabling monitoring and tracking at unprecedented granularity in both time and space.

Appendix A

Energy Consumption Analysis

In this appendix, we describe in detail the results of the analysis used to measure the breakdown of energy consumption during various phases of processing in a typical data collection application, as summarized in Section 2.3.4.

In this study, we assume that each mote transmits one sample of (air-pressure, acceleration) readings every ten seconds and listens to its radio for one second per ten-second period to receive results from from neighboring sensors and obtain access to the radio channel. We assume the following hardware characteristics: a supply voltage of 3V, an Atmega128 processor [Atm] that can be set into Power Down Mode and runs off the internal oscillator at 4Mhz, the use of the Intersema Pressure Sensor [Int02b] and Analog Devices accelerometer [Ana], and a ChipCon CC1000 Radio [Corb] 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 mode to reduce reception power when no other radios are communicating, and that, on average, each node has ten neighbors that it hears per 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 and to allow the radio to employ channel sampling and reduce power consumption when not actively receiving a packet. The results are shown in Table A.1.

These results show that the processor and radio together consume the majority of energy for

Hardware	Current (mA)	Active Time, Per 10s	% Total Energy
Sensing, Accelerometer	.6	.0027s	.02
Sensing, Pressure	.35	.025s	.09
Communication, Sending (70 bytes @ 38.4bps x 2 packets)	10.4	2 x .0145 s	4.48
Communication, Receive Packets (70 bytes @ 38.4bps x 10 packets)	9.3	.145s	20.03
Communication, Sampling Channel	.074	.855s	.94
Processor, Active	5	1s	74.3
Processor, Idle	.001	9s	.13
Average current draw per second	.6729 mA		

Table A.1: *Expected Power Consumption for Major Hardware Components,* a query reporting light and accelerometer readings once every ten seconds.

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. For example, in the above table, 0.035 seconds are spent waiting for the light sensor to start, and another .0145 seconds are spent transmitting. Furthermore, the MAC layer on the radio introduces a delay proportional to the number of devices transmitting. To measure this effect, we measured 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), this suggests that, with 10 nodes, the average MAC delay will be at least $(.06 - .007) \times 5) = 0.265s$. Thus, of the 1 second each mote is awake, at least 0.3145 seconds of that time is spent waiting for the sensors or radio.

In fact, however, this 1 second delay is selected almost exclusively to allow MAC and sensing

delay to occur. Application computation is almost negligible for basic data collection: 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 74% of energy spent on the processor, fewer than 3% of its cycles are spent in application processing. For the example given here about 4% of this 74% is spent waiting for sensors, and another 28% waiting for the radio to send or receive. The remaining 65% 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 or sending with the energy used by the radio itself, we get:

$$(0.65 + 0.28) \times\ 0.74 + 0.05 + 0.20 = .94$$

Thus, about 94% of power consumption in this simple data collection task is due to communication.

Appendix B

TinyDB 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 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 <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 B.1.

Language Construct	Section
ON EVENT	Section 5.2.1
SELECT-FROM-WHERE	Section 3.2
GROUP BY, HAVING	Section 4.2
OUTPUT ACTION	Section 3.2.4
SIGNAL <event>	Section 5.2.1
INTO STORAGE POINT	Section 3.2.2
SAMPLE PERIOD	Section 3.2
FOR	Section 3.2
STOP ON	Section 5.2.1
COMBINE	Section 3.2
ONCE	Section 3.2
LIFETIME	Section 5.2.2

Table B.1: References to sections in the main text where query language constructs are introduced.

B.2 Storage Point Creation and Deletion Syntax

The syntax for storage point creation is:

```
CREATE [CIRCULAR] STORAGE POINT name
SIZE [ ntuples | nseconds]
[( fieldname type [, ... , fieldname type])] |
[AS SELECT ... ]
[SAMPLE PERIOD nseconds]
```

and for deletion:

```
DROP STORAGE POINT name
```

Both of these constructs are described in Section 3.2.2.

Appendix C

TinyDB Components and Code Size

The latest version of TinyDB (as of June, 2003), consists of about 20,000 lines (including white space and comments), or 6,500 semicolons. The breakdown of this code by module is shown in Figure C.1. Aside from the aggregate and attribute definitions, which consume a large amount of code because there are many of them (14 of each, as of this writing), the largest components are `TupleRouterM.nc` and `DBBufferC.c`. The TupleRouter[1] is responsible for the bulk of query processing – it implements the query preprocessing and sharing functionality as well as facilities to construct tuples and route them through queries. The DBBufferC is the storage management code that constructs and manages `STORAGE POINTS`.

Briefly, the modules in this figure are as follows (from the top, going clockwise):

- `ParsedQuery.nc`: The component responsible for managing the query data structure. Includes methods to provide the types and sizes of all of the fields in the query, as well as the order and types of the operators.

- `Network`: Several components for passing query and data messages to and from the TupleRouter, and for managing the TinyDB-specific interfaces to the low-level TinyOS networking components.

[1]The "M" in the filename denotes that this is the TupleRouter *Module* and thus contains code rather than a interface or configuration (see Appendix D).

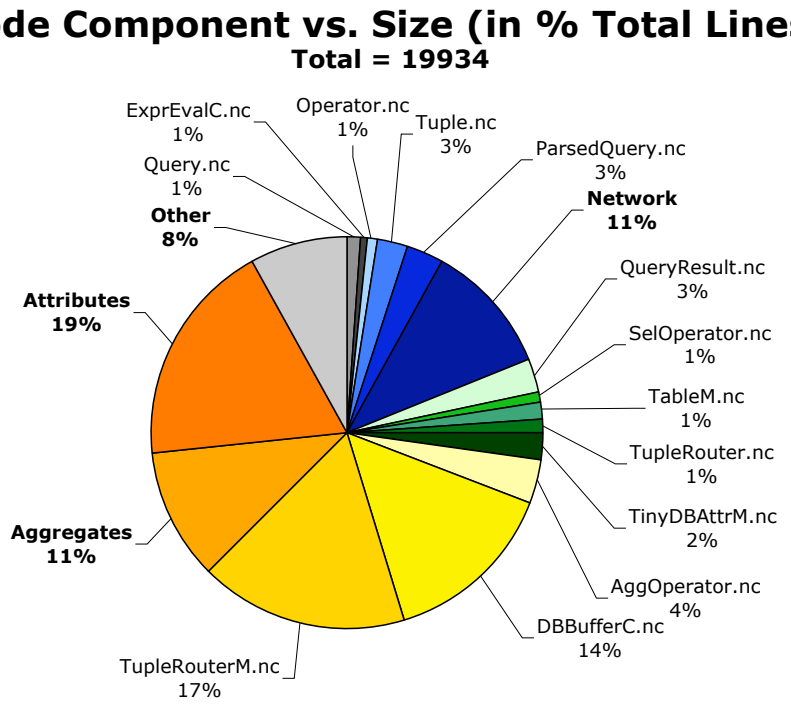


Figure C.1: Breakdown of code size by module in TinyDB. Names in **bold** indicate several fields with related functionality that have been combined into a single pie-slice.

- `QueryResult.nc`: QueryResults are sets of tuples that have been wrapped in network headers for transmission outside of the query processor. Often, each query result contains just a single tuple representing the data collected in the current epoch, but may contain many tuples for grouped aggregates and join queries.

- `SelOperator.nc`: Applies a selection predicate over a particular tuple.

- `TableM.nc`: Manages a named table with a specific schema. In the current implementation, schemas are implemented as parsed queries with no operators, so this file is basically just a wrapper on ParsedQuery.nc.

- `TupleRouter.nc`: (Not to be confused with `TupleRouterM.nc`). The main configuration file for TinyDB that specifies how all of the query processing and operating system

components are connected together. See Appendix D for a discussion of configuration files in TinyOS.

- `TinyDBAttrM.nc`: Specifies and connects together all of the attributes that are accessible in TinyDB queries.

- `AggOperator.nc`: Implements the functionality common to all aggregate functions: group management, expression evaluation, and so on.

- `DBBufferC.nc`: Implements `STORAGE POINTS`. Manages access to and from the Flash memory, controls data layout and handles multiple readers/writers for a single `STORAGE POINT`.

- `TupleRouterM.nc`: The main query processing engine. Accepts and pre-processes queries from the network. Builds tuples using data acquisition operators, and routes tuples through aggregation and selection operators.

- `Aggregates`: All of the aggregate functions (14 files in all); each specifies taxonomy-related metadata as well as implementations of the initialization, merging, and finalization functions

- `Attributes`: All of the attributes available for querying (14 files in all). Each attribute specifies relevant metadata, such as per-sample energy cost, as well as a pointer to a TinyOS function that acquires the value of the attribute.

- `Other`: Miscellaneous support and interface files.

- `Query.nc`: Code to manage the query data structures as they arrive over the network and before they are converted to parsed queries. The main difference between `Query.nc` and

`ParsedQuery.nc` is that queries contained named fields that are replaced by integer offsets into the catalog in parsed queries.

- `ExprEvalC.nc`: A simple expression evaluator used in selection and aggregation predicates.
- `Operator.nc`: The interface specifying the structure of an operator, implemented by aggregates and selections.
- `Tuple.nc`: Code to manage a single result row with a list of typed fields.

Appendix D

Component Based, Event-Driven Programming

The TinyOS programming language, nesC [GLvB+03], is a C-like language where programs are structured as *components*. Table D.1 summarizes the terminology used in nesC. Components *provide*, or implement, certain *interfaces*, that define publicly available functions contained in the component. Components can also *require* interfaces, in which case they must be connected, or *wired*, to another component that provides the interface. Wiring is done via special *configurations*, which connect together a set of components and may themselves provide and require certain interfaces (so can be used just like components in other configurations.) Figure D.1 illustrates how a configuration can be used to wire together two components that require / provide a shared interface.

Functions in nesC can be one of three types: *commands*, *events*, or *tasks*. Commands and events are usually paired (although this pairing is not explicitly done at the language level); commands request that some action be taken, and events signal that an action has completed or an event has

Term	Definition
Component	A code module that implements some set of interfaces.
Interface	A set of function signatures which components can implement
Configuration	A collection of components that are connected together according to a requires / provides relationship.
Provides	A component *provides* an interface if it implements it.
Requires	If a component *requires* an interface, it must be connected (via a configuration) to some other component which provides that interface.

Table D.1: Terminology used in nesC/TinyOS

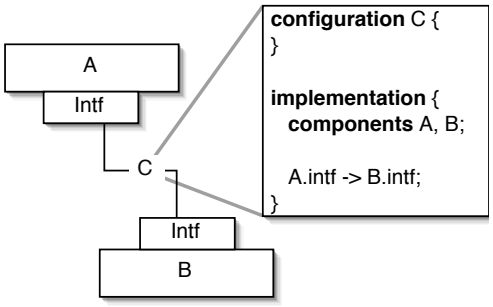


Figure D.1: The Requires - Provides Relationship

occurred. Example commands include “get the current light level”, or “run query Y”; example events include “the current light level is X”, or “query Y has finished”. Commands and events may preempt each other – an event handler may fire at any time (often in the middle of execution of another event or command), and that event handler may in turn call commands. Preemption happens in practice because events are frequently associated with low-level processor interrupts; for example, when the Atmel CPU finishes sending a bit over the UART (serial port), it will fire an interrupt which will cause an event to be signaled. NesC provides atomic sections which are guaranteed not to be preempted; this is (typically) implemented by disabling processor interrupts.

Unlike preemptive commands and events, tasks are executed sequentially by TinyOS; an application can *post* a task, which will cause it to be enqueued and executed at some later point in time. Tasks are guaranteed to run in isolation of each other, but commands and events may occur while a task is executing.

Thus, nesC provides a simple form of concurrency via interrupt driven events and commands, as well as a more traditional, sequential execution model via tasks on top of the basic features of C.

Appendix E

Declarative Queries and SQL

This appendix provides a brief tutorial on declarative queries and the SQL programming language.

To illustrate the power of a declarative interface, we consider an enterprise database system containing an employee table, or *relation* with age, salary, and department information about each employee. In most database systems, we would query the fields of this relation using the *structured query language*, or SQL. For example, we can compute the average salary of all employees in the accounting department who are over 25 with a SQL query like:

(4)
```
SELECT AVG(emp.salary)
 FROM emp
 WHERE emp.age > 25
 AND emp.deptNo = ACCOUNTING_ID
```

Here, the `FROM` clause specifies that this is a query over the table `emp` (which contains employee records.) This table has one row per employee, and (as far as we can tell from this query), contains each employee's age, salary, and department id. The `WHERE` clause filters out those employees who do not satisfy the age and department criteria, while the `AVG` operator indicates this is an *aggregate* query that combines multiple rows into a single result. In this case, that result will contain the average salary of of all employees who satisfy the conditions in the `WHERE` clause.

E.1 Complex Queries

SQL is a considerably more complex language than this query indicates. Queries may contain *joins*, which concatenate together rows from two tables which satisfy some *join condition*. Consider the query "Tell me every employee's name and the name of his or her departmental manager". Suppose that the departmental manager can be found by consulting the `dept` (departments) table for the manager's `empId`, which uniquely identifies the manager's employee record, including his or her name. To answer this query, a join between the employees table and departments table must be used. Each result from this query will be the combination two records from the `emp` table – the employee's record and the manager's record. Thus, two employee records will be joined only when the employee's department is the same as the department that the manager supervises[1]. In SQL, this would be:

```
SELECT emp.name, mgr.name
 FROM emp, emp as mgr, dept
 WHERE mgr.empId = dept.mgrId
 AND emp.deptNo = dept.deptNo
```

Note that we actually have two join conditions here: one to find the correct department record given the employees department, `emp.deptNo` and one to find the manager's name given his or her employee id, `dept.mgrId`.

Besides joins, much of the complexity in SQL query processing comes from *nested queries*. The idea of a nested query is very simple: because the result of any query is a table, queries may be applied to the results of previously executed queries. These queries may be combined together into a single statement. For example, to answer the query "find all the employees who make more than the average salary", we could execute the SQL statement:

[1]Note that this query may produce |`emp`| x |`dept`| results in the worst case, where |*t*| indicates the number of rows in table *t*. This would only happen, however, if multiple records for the same department appear in the departments table, which is unlikely – we expect that `deptNo` is a *primary key* for the departments table, meaning that each value of `deptNo` appears at most once in the table.

```
SELECT emp.salary
 FROM emp
 WHERE emp.salary > (
 SELECT AVG(emp.salary) from emp
 )
```

There are many other features of SQL that we cannot cover here. The interested reader should consult one of many introductory texts; see, for example [RG00] for an excellent overview.

E.2 Query Optimization

It is interesting to note, even for a query as simple as Query 4, there are a number of alternative ways, or *plans* which the query processor could consider during execution. The following is a (by no means exhaustive) list of alternatives:

1. It could first build a list of all the employees over 25, then build a list of all the employees in the accounting department. It could merge these lists and then compute the average salary of the merged list.

2. It could scan the table once, checking for each record whether the employee age is over 25 and that the employee is in the accounting department. For each satisfying record, it could update a running average.

3. It could use a sorted *index* of employee ages to quickly find all the employees older than 25, filter each matching employee record for department, and then update a running average with each satisfying record.

4. Along with the table, it could maintain an average of the salaries of all employees. It could then scan the table, removing from the average employees whose age and department do not satisfy the query.

Note that the user does not consider any of these issues or alternatives – the process of plan formulation and selection is done completely *transparently*, inside the database system.

The process by which the *best*, or most efficient, plan is selected is called *query optimization*. Such optimization is typically done by associating an estimated cost with each step in the plan, computing the total cost of each plan, and then selecting the plan of minimal cost. Cost estimates are typically computed by maintaining statistics about the data distribution. There is a vast literature associated with this process, which we have greatly simplified; the interested reader is referred to [RG00] for a simple introduction to the topic.

So which of the above plans is the best choice? Clearly plan 1 is inefficient, because it requires the system to store portions of the table and merge them, rather than computing the answer in a single pass. If most employees are under 25, plan 3 likely dominates plan 2; however, if most employees are over 25, plan 2 may be preferable because it allows the table to be scanned sequentially from disk, rather than using an index to look up employees in order of age (which may require random disk accesses, depending on the layout of the index on disk.) Plan 4 is hard to predict (and in fact, most database systems would not include this kind of optimization): because average is easy to update incrementally, it will likely perform similarly to plan 2, but could be slightly more efficient if there are very few records that do not satisfy both predicates.

Bibliography

[AG93] 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.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD*, pages 261–272, Dallas, TX, May 2000.

[AK93] Rafael Alonso and Henry F. Korth. Database system issues in nomadic computing. In *ACM SIGMOD*, Washington DC, June 1993.

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

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

[AWSBL99] William Adjue-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *ACM SOSP*, December 1999.

[BB03] Tim Brooke and Jenna Burrell. From ethnography to design in a vineyard. In *Proceeedings of the Design User Experiences (DUX) Conference*, June 2003. Case Study.

[BBKV87] François Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. FAD, a powerful and simple database language. In *VLDB*, 1987.

[BDF+97] Daniel Barbará, William DuMouchel, Christos Faloutsos, Peter J. Haas, Joseph M. Hellerstein, Yannis E. Ioannidis, H.V. Jagadish, Theodore Johnson, Raymond T. Ng, Viswanath Poosala, Kenneth A. Ross, and Kenneth C. Sevcik. The New Jersey data reduction report. *Data Engineering Bulletin*, 20(4):3–45, 1997.

[BG03] M. Beigl and H. Gellersen. Smart-its: An embedded platform for smart objects. In *Proceedings of the Smart Objects Conference (sOc)*, Grenoble, France, May 2003.

[CCC+02] 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.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *First Annual Conference on Innovative Database Research (CIDR)*, 2003.

[CDTW00] Jianjun Chen, David DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD*, 2000.

[CED+01] 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.

[CG02] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS*, July 2002.

[CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In *ACM SIGMOD*, 2001.

[CGRS01] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *VLDB Journal*, 10, 2001.

[CGW02] Kenneth Calvert, James Griffioen, and Su Wen. Lightweight network support for scalable end-to-end services. In *ACM SIGCOMM*, 2002.

[CHZ02] M. Chu, H. Haussecker, and F. Zhao. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. In *International Journal of High Performance Computing Applications, 2002.*, 2002.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, 1994.

[Cora] Atmel Corporation. *Atmel 4-Megabit 2.7-Volt Only Serial DataFlash (AT45DB041) Datasheet*. `http://www.atmel.com/atmel/acrobat/doc1938.pdf`.

[Corb] ChipCon Corporation. *CC1000 Single Chip Very Low Power RF Transceiver Datasheet*. `http://www.chipcon.com`.

[Corc] Ember Corporation. Ember. Web Site. `http://www.ember.com/`.

[Cord] Millenial Corporation. Millennial networks. Web Site. `http://www.millennial.net/`.

[Core] Sensoria Corporation. sGate wireless sensor gateway. Data Sheet. http://www.sensoria.com/downloads/sGate%20Brochure%2003217-01%20Rev%20A.pdf.

[Corf] Xsilogy Corporation. Xsilogy. Web Site. http://www.xsilogy.com/.

[Daw98] T.E. Dawson. Fog in the california redwood forest: ecosystem inputs and use by plants. *Oecologia*, 117(4):476–485, 1998.

[DEC82] Xerox Digital Equipment Corporation, Intel. *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (Version 2.0)*, 1982.

[DJ00] Kevin A. Delin and Shannon P. Jackson. Sensor web for *in situ* exploration of gaseous biosignatures. In *IEEE Aerospace Conference*, 2000.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI*, 2002.

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

[Fal03] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the ACM SIGCOMM 2003*, August 2003.

[FGB02] Anton Faradjian, Johannes Gehrke, and Philippe Bonnet. GADT: A Probability Space ADT For Representing and Querying the Physical World. In *ICDE*, 2002.

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

[FJL+97] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE Transactions on Networking*, 5(6):784–803, 1997.

[GAGPK01] Tom Goff, Nael Abu-Ghazaleh, Dhananjay Phatak, and Ridvan Kahvecioglu. Preemptive routing in ad hoc networks. In *ACM MobiCom*, July 2001.

[Gan02] Deepak Ganesan. Network dynamics in Rene Motes. PowerPoint Presentation, January 2002.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, February 1996.

[GG01] M. Garofalakis and P. Gibbons. Approximate query processing: Taming the terabytes! (tutorial). In *VLDB*, 2001.

[GGC03] A. Ghose, J. Grossklags, and J. Chuang. Resilient data-centric storage in wireless sensor networks. In *Proceedings of the 4th International Conference on Mobile Data Management (MDM)*, Melbourne, Australia, January 2003.

[GKS01] Johannes Gehrke, Flip Korn, and Divesh 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.

[GKW+02] Deepak Ganesan, Bhaskar Krishnamachari, Alec Woo, David Culler, Deborah Estrin, and Stephen 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.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Network Embedded Systems. In *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[Han96] Eric 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.

[Hel98] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. *TODS*, 23(2):113–157, 1998.

[HH89] Paul Horowitz and Winfield Hill. *The Art of Electronics (2nd Edition)*. Cambridge University Press, 1989.

[HHL02] Zygmunt J. Haas, Joseph Y. Halpern, and Li Li. Gossip-based ad hoc routing. In *IEEE Infocom*, 2002.

[HHMS03] Joseph Hellerstein, Wei Hong, Samuel Madden, and Kyle Stanek. Beyond average: Towards sophisticated sensing with queries. In *Proceedings of the First Workshop on Information Processing in Sensor Networks (IPSN)*, March 2003.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen Wang. Online aggregation. In *Proceedings of the ACM SIGMOD*, pages 171–182, Tucson, AZ, May 1997.

[Hil03] Jason Hill. *System Architecture for Wireless Sensor Networks*. PhD thesis, UC Berkeley, 2003.

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

[HOT88] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeo K. Taneja. Statistical estimators for relational algebra expressions. In *Proceedings of the Seventh ACM Conference on Principles of Database Systems*, pages 288–293, March 1988.

[HSI+01] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *SOSP*, October 2001.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, and David Cullerand Kristofer Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

[IB92] T. Imielinski and B.R. Badrinath. Querying in highly mobile distributed environments. In *VLDB*, Vancouver, Canada, 1992.

[IEGH01] Chalermek Intanagonwiwat, Deborah Estrin, Ramesh Govindan, and John Heidemann. Impact of network density on data aggregation in wireless sensor networks. ICDCS-22, November 2001.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *Proceedings of the ACM SIGMOD*, 1999.

[IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCOM*, Boston, MA, August 2000.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *TODS*, 9(3):482–502, 1984.

[Inc] Dust Inc. Company Web Site. http://www.dust-inc.com.

[Int02a] Intel Developer's Forum. *Futue Trends in Non-Volative Memory Technology*, February 2002. Slide Presentation, available at http://www.ee.ualberta.ca/~salamon/OUM/StefanLaiIDF0202.pdf.

[Int02b] Intersema. MS5534A barometer module. Technical report, October 2002. http://www.intersema.com/pro/module/file/da5534.pdf.

[Ive02] Zachary G. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, 2002.

[JAR03] Qun Li Javed Aslam and Daniela Rus. Three power-aware routing algorithms for sensor networks. *Journal of Wireless Communications and Mobile Computing*, 3(2):187–208, March 2003.

[JOW+02] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, LiShiuan Pehand, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *Proceedings of the Tenth ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pages 128–137, 1986.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the ACM SIGMOD Conference*, pages 106–117, 1998.

[KK00] Brad Karp and H.T. Kung. Greedy perimeter stateless routing for wireless networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 243–254, Boston, MA, 2000.

[Kos00] Donald Kossman. The state of the art in distributed query processing. *ACM Computing Surveys*, 2000.

[KRB99] Joanna Kulik, Wendi Rabiner, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *MobiCOM*, 1999.

[Lar02] Per-Åke Larson. Data reduction by partial preaggregation. In *ICDE*, 2002.

[LC02] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of The International Conference on Architectural Support For Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, 2002.

[LKGH03] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceeding of the First ACM Conference on Sensor Systems (SenSys)*, 2003. To Appear.

[LN97] Michael V. Leonov and Alexey G. Nitikin. An efficient algorithm for a closed set of boolean operations on polygonal regions in the plane. Technical report, A.P. Ershov Institute of Informatics Systems, 1997. Preprint 46 (In Russian.) English translation available at `http://home.attbi.com/~msleonov/pbpaper.html`.

[LP96] J.C. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. In *INFOCOM*, pages 1414–1424, 1996.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven informa-

tion delivery. *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.

[LS03] Philip Levis and Scott Shenker. Epidemic algorithms for code distribution in sensor networks. In Submission, July 2003.

[LWHS02] Dan Li, Kerry Wong, Yu Hen Hu, and Akbar Sayeed. Detection, classification and tracking of targets in distributed sensor networks. *IEEE Signal Processing Magazine*, 19(2), Mar 2002.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, Inc., 1996.

[MBLS01] Jitendra Malik, Srege Belognie, Thomas Leung, and Jianbo Shi. Contour and texture analysis for image segmentation. *International Journal of Computer Vision*, 43(1):7–27, 2001.

[MF02] Samuel Madden and Michael J. Franklin. Fjording the stream: An architechture for queries over streaming sensor data. In *ICDE*, 2002.

[MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.

[MFHH03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *ACM SIGMOD*, 2003. To Appear.

[MPSC02] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, and David Culler. Wireless sensor networks for habitat monitoring. In *ACM Workshop on Sensor Networks and Applications*, 2002.

[MRSR01] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD*, 2001.

[MS79] C. L. Monma and J.B. Sidney. Sequencing with seriesparallel precedence constraints. *Mathematics of Operations Research*, 1979.

[MSFC02] Samuel Madden, Robert Szewczyk, Michael Franklin, and David Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *WMCSA*, 2002.

[MSHR02] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continously adaptive continuous queries over data streams. In *ACM SIGMOD*, Madison, WI, June 2002.

[MVW98] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD*, pages 448–459, Seattle, Washington, 1998.

[MWA+03] 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 *First Annual Conference on Innovative Database Research (CIDR)*, 2003.

[NES] NEST Group. NEST Challenge Architecture. Web Site. `http://webs.cs.berkeley.edu/tos/api/`.

[OJ02] C. Olston and J.Widom. In *Best Effort Cache Sychronization with Source Cooperation*, 2002.

[Ore03] Intel Research Oregon. Heterogeneous sensor networks. Technical report, Intel Corporation, 2003. Web Page. `http://www.intel.com/research/exploratory/heterogeneous.htm`.

[PC97] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM*, 1997.

[PCB00] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *MOBICOM*, August 2000.

[PJP01] P.Bonnet, J.Gehrke, and P.Seshadri. Towards sensor database systems. In *Conference on Mobile Data Management*, January 2001.

[Pol03a] Joe Polastre. Personal communication. July 2003.

[Pol03b] Joseph Polastre. Design and implementation ofwireless sensor networks for habitat monitoring. Master's thesis, UC Berkeley, 2003.

[RAK+02a] J. Rabaey, J. Ammer, T. Karalar, S. Li, B. Otis, M. Sheets, and T. Tuan. PicoRadios for Wireless Sensor Networks: The Next Challenge in Ultra-Low-Power Design. In *Proceedings of the International Solid-State Circuits Conference*, San Francisco, CA, Feburary 2002. Also see: `http://bwrc.eecs.berkeley.edu/Publications/2002/presentations/isscc2002/ISSCCslides.pdf`.

[RAK+02b] Jan M. Rabaey, Josie Ammer, Tufan Karalar, Suetfei Li, Brian Otis, Mike Sheets, and Tim Tuan. PicoRadios for wireless sensor networks: The next challenge in ultra-low-power design. In *Proceedings of the International Solid-State Circuits Conference*, San Francisco, CA, 2002.

[RFM] RFM Corporation. RFM TR1000 Datasheet. `http://www.rfm.com/products/data/tr1000.pdf`.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2nd edition, 2000.

[RKY+02] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. GHT: A geographic hash table for data-centric storage. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks (WSNA)*, 2002.

[RRH02] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering. *The VLDB Journal*, 9(3), 2002.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD*, pages 249–260, 2000.

[RSWC01] P. Ramanathan, K. Saluja, K-C. Wang, and T. Clouqueur. UW-API: A Network Routing Application Programmer's Interface. Draft version 1.0, January 2001.

[Sak01] Takayasu Sakurai. Interconnection from design perspective. In *Proceedings of the Advanced Metallization Conference 2000(AMC 2000)*, pages 53–59, 2001.

[Sci] Campbell Scientific. Campbell Scientific Inc. Measurement and Control Systems. Web Page. `http://www.capbellsci.com`.

[Sen02] Sensirion. SHT11/15 relative humidity sensor. Technical report, June 2002. `http://www.sensirion.com/en/pdf/Datasheet_SHT1x_SHT7x_0206.pdf`.

[SH00] Lakshminarayanan Subramanian and Randy H.Katz. An architecture for building self-configurable systems. In *MobiHOC*, Boston, August 2000.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, 1991.

[SN95] Ambuj Shatdal and Jeffrey Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*, 1995.

[Sno95] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publisher, 1995.

[Soc] Audobon Society. Leach's storm-petrel (oceanodrama lucorhoa). Web site. `http://www.audubon.org/bird/puffin/virtual/stormpetrel.html`.

[Sol02] Texas Advanced Optoelectronic Solutions. TSL2550 ambient light sensor. Technical report, September 2002. `http://www.taosinc.com/images/product/document/tsl2550.pdf`.

[SSS+02] S. Sastry, S. Schaffert, L. Schenato, S. Simic, B.Sinopoli, E. Brewer, D. Culler, and D. Wagner. Development of the NEST Challenge Application: Distributed Pursuit Evasion Games, 2002. Presentation. `webs.cs.berkeley.edu/weekly/Berkeley_challenge_app.pdf`.

[Sto93] Jack L. Stone. Photovoltaics: Unlimited electrical energy from the sun. *Physics Today*, September 1993. Available online. `http://www.nrel.gov/ncpv/documents/pvpaper.html`.

[Sys02a] Melexis Microelectronic Integrated Systems. MLX90601 infrared thermopile module. Technical report, August 2002. `http://www.melexis.com/prodfiles/mlx90601.pdf`.

[Sys02b] Suntrek Alternative Energy Systems. Solar insolation tables. Web Site., July 2002. http://www.suntrekenergy.com/sunhours.htm.

[TGO99] Kian-Lee Tan, Cheng Hian Goh, and Beng Chin Ooi. Online feedback for nested aggregate queries with multi-threading. In *VLDB*, 1999.

[TSS+97] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survery of active network research. *IEEE Communications*, 1997.

[TYD+03] Niki Trigoni, Yong Yao, Alan Demers, Johannes Gehrke, and Rajmohan Rajaramany. Wavescheduling: Energy-efficient data dissemination for sensor networks. In Submission, June 2003.

[UF00] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, pages 27–33, 2000.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD*, 1998.

[VB00] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical Report CS-200006, Duke University, April 2000.

[WC01] Alec Woo and David Culler. A transmission control scheme for media access in sensor networks. In *ACM Mobicom*, July 2001.

[Whi02] Kamin Whitehouse. The Design of Calamari: an Ad-hoc Localization System for Sensor Networks. Master's thesis, University of California at Berkeley, 2002.

[WSX+99] Ouri Wolfson, A. Prasad Sistla, Bo Xu, Jutai Zhou, and Sam Chamberlain. DOMINO:

Databases fOr MovINg Objects tracking. In *ACM SIGMOD*, Philadelphia, PA, June 1999.

[WTC03] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *ACM SenSys*, 2003. To Appear.

[YC95] Andrew Yu and Jolly Chen. *The POSTGRES95 User Manual*. UC Berkeley, 1995.

[YG02] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. In *SIGMOD Record*, September 2002.

[YG03] Yong Yao and Johannes Gehrke. Query processing in sensor networks. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[YHE02] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *IEEE Infocom*, 2002.

[YL95] Weipeng P. Yan and Per Åke Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.