Query Processing for Streaming Sensor Data

Ph.D. Qualifying Exam Proposal

Samuel Madden
Computer Science Division
University of California, Berkeley
madden@cs.berkeley.edu

1 Research Summary

Over the past few years, a great deal of attention in the networking and mobile-computing communities has been directed toward building networks of ad-hoc collections of sensors scattered throughout the world. MIT, UCLA, and UC Berkeley [23, 32, 12, 38] have all embarked on projects to produce small, wireless, battery-powered sensors and low-level networking protocols. These projects have brought us close to the the vision of ubiquitous computing [49], in which computers and sensors assist in every aspect of our lives. To fully realize this vision, however, it will be necessary to combine and query the readings produced by these collections of sensors, since manually retrieving and combining data from thousands of nodes is tedious and infeasible. An obvious approach for doing this would be to apply traditional data-processing techniques and operators from the database community. Unfortunately, standard DBMS assumptions about the reliability, availability, interface, and requirements of data sources do not apply to sensors, so a significantly different architecture is needed. Understanding, designing and implementing this architecture, called *TeleTiny*, is the focus of my research. TeleTiny consists of two primary components:

1. *Server Side:* Modifications to traditional query processor architectures to enable them to access and efficiently query streaming sensor data, without wasting limited energy reserves of the sensors.
2. *Sensor Side:* Modifications to the software which runs on the sensors themselves to enable database-style queries involving joins, aggregates, and other operators to be partially executed within sensor networks, in an effort to reduce communication costs and power consumption.

This document summarizes my progress to date in building TeleTiny, with a particular focus on these two components and the interfaces between them, and discusses how my Ph.D. will yield a complete architecture for energy-efficient query processing over streaming sensor data. The remainder of this section summarizes the challenges of sensor-query processing, sketches the TeleTiny architecture, and shows how it meets those challenges.

1.1 Challenges

There are four primary differences between sensor based data sources and traditional database sources which make standard query processors poorly suited for querying sensor data. First, sensors typically deliver data in *streams*: they produce data continuously, often at well defined time intervals, without having been explicitly asked for that data. Queries over those streams need to be processed in near real-time, as data arrives, partly

because it may be extremely expensive to save raw sensor streams to disk, and partly because sensor streams represent real-world events, like traffic accidents and attempted network break-ins, which need to be responded to. The second major challenge with processing sensor data is that sensors are fundamentally different from the well engineered data-sources typical in a business DBMS. They do not deliver data at reliable rates, the data is often garbled, and they have limited processor and battery resources that the query engine needs to conserve whenever possible. Third, sensors are general purpose computing devices, not just blind producers of data and can thus be used to partially compute queries as data flows through them. Finally, sensors are typically connected together in ad-hoc networks that cover a geographic area larger than the radio range of a single sensor, such that receiving data from arbitrary parts of the network requires the sensors to agree on a multi-hop transmission protocol and coordinate to route and process each other's data.

1.2 Sensor-Motivated Query Processor Architecture

My initial research focused on the architectural issues that arise in sensor query processing with respect to the central query processor. This research has identified a number of areas where a query processing engine can be modified to make it better suited to queries over sensor data:

- *Fjords and Sensor Proxies*: These two concepts form the core of the DBMS side sensor query processing system that is a part of the Telegraph [19] dataflow system under development at UC Berkeley. A *Fjord* is a modified query plan-like data structure which facilitates the combination of streaming data from sensors with traditional, disk-based data sources. Fjords allow query processors to be tolerant to intermittent and lossy data streams, since they compute only when sensor tuples are *pushed* into them, avoiding blocking that would arise in the traditional *iterator* based processing [16].

  A sensor proxy is a database operator that multiplexes multiple user queries over a collection of sensors. It caches sensor readings, chooses appropriate sample rates for sensors based on sample rates from all user queries, and attempts to shield sensors from having to serve many copies of the same data to different queries.

  A paper describing this work appeared in ICDE 2002 [28].

- *Continuously Adaptive Continuous Queries*: Because sensors networks will be used to monitor large, highly accessible spaces, there will be many simultaneous, similar queries over those networks. Traditional continuous query research, such as the NiagaraCQ system from Wisconsin [6] allows multiple related queries to share processing by identifying common portions of query plans and executing those common portions only once. My research on continuously adaptive continuous queries (CACQ) extends traditional CQ research in two directions: by enabling stream processing facilities for sensor data and adding adaptivity to changes in the query workload over time.

  A paper on continuously adaptive continuous queries will appear in SIGMOD 2002 [30].

- *Visualizations for Sensor Data*: A third area of research has to do with visualizations of sensor data, and the use of user interface to coordinate sensor readings with disk and Internet based data sources.

1.3 Database Motivated Sensor Software Architecture

Over the past year, I have also been working with the TinyOS [20] group at UC Berkeley on developing software for their "mote" sensor platform to support query processing. My research in this regard has focused on the following areas:

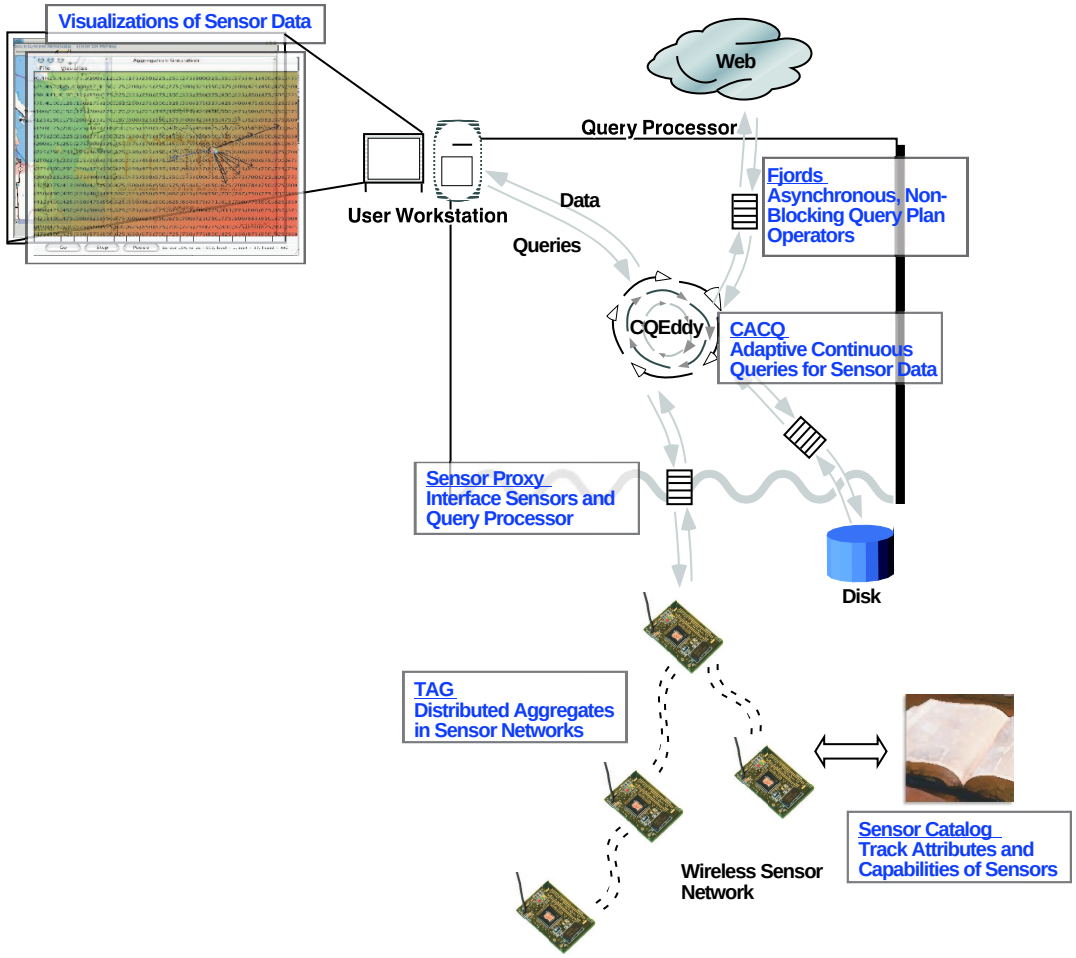

Figure 1: *The TinyDB Sensor Query Processor*

- *Catalog Management for Sensor Networks*: Sensors need a generic way to advertise themselves to other sensors and sensor database systems if they are to be used in a heterogeneous, randomly deployed fashion. I have designed a simple catalog layer which our query processor can use to locate and query TinyOS motes.

- *Aggregation in the Network*: The most important class of queries over sensor networks are *aggregates* – queries about the values of groups of sensors. Efficiently evaluating these queries without sending all data-tuples back to a central processing system is a major goal of the TinyOS project. My work on Tiny Aggregates (TAG), submitted to VLDB 2002 and published in a preliminary form in WMCSA 2002 [31], discusses the design of a system for processing such queries.

Thus, the on-sensor portions of the query processor enable location and power efficient querying of motes; when combined with the server-side query processor described above, most of the pieces of the sensor-query processor are in place. Figure 1 shows an overview of these components and how they fit together – we will discuss the additional components that are needed in Section 5 below.

The remainder of this proposal is divided as follows: the next section offers some details about Fjords, sensor-proxies, and the CACQ system. Section 3 then presents details of the catalog and on-sensor aggregation

systems discussed above. This is followed by a discussion of related work in Section 4. Finally, a discussion of additional research that will be incorporated into my Ph.D, with a particular focus on the work required to combine the components described above into a system for sensor query processing is given in Section 5.

2 Query Processing Architecture

This section, summarizes Fjords, Sensor Proxies, the CACQ system, and the visualization prototypes which serve as the core of the server side sensor-data query processor.

2.1 Fjords

Previous database architectures are not suited to combining streaming and static data. They are either primarily pull-based, as with the basic iterator model, or primarily push based, as in parallel processing environments. In the Telegraph project, we have developed a hybrid approach called Fjords, whereby streams can be combined with static sources in a way which varies from query-to-query. We believe that this is an essential part of any data processing system that is intended to compute over streams.

The key advantage of Fjords is that they allow distributed query plans to use a mixture of push and pull connections between operators. Push or pull is implemented by the queue that connects a pair of operators: a push queue relies on its input operator to put data into it which the output operator can later get. A pull queue actively requests that the input operator produce data in response to a get call on the part of the output operator. Conversely, in a push queue, the input operator's get request is non-blocking, does not schedule the output operator, and only yields data when the output operator has made data available.

The insight underlying Fjords is that in interactive, adaptive, and streaming systems, the traditional iterator model breaks down because the query processor cannot afford to block waiting for long running operators to complete, for slow web pages to return results, or for individual sensors, which may have run out of power or temporarily disconnected, to come back online. One way to deal with this is the solution proposed in Volcano [16], in which a special "exchange" operator between two query processing operators, one of which is producing some data and one of which is consuming that data. In this model, the producer can run in its own thread (or on another machine); it delivers results to the exchange operator, which queues them and synchronously delivers them to the consumer when it needed. The consumer, however, is still forced to block when no data is available, limiting the ability of the query processing system to adapt. Instead, Fjords allow the query planner to decide if two operators should be connected by a pull-queue, in which case the consumer will block waiting for data from the producer, or a push-queue, in which case the producers will asynchronously produce and and enqueue results, and control will be returned to the consumer when the queue is empty so it can pursue some other computational avenue.

Push queues make it possible for the query processor to handle sensor streams. When a sensor tuple arrives from a sensor, the query processor pushes it onto the input queues of the queries which use it as a source. The operators draining those queues never actively ask the sensor for data; they merely operate on sensor data as it is pushed into them, thus triggering computation when data is available and unblocking the query processor so it can make useful progress despite slow data delivery from a particular sensor.

Figure 2 illustrates several possibilities for connecting operators in a push or pull fashion. Three types of connections between a producer and a consumer operator are shown; in the iterator approach, operators function in lock-step: when the consumer blocks looking for data, the producer is scheduled and runs until it produces data. Exchange decouples the producer and the consumer, but the consumer still blocks until data is produced–

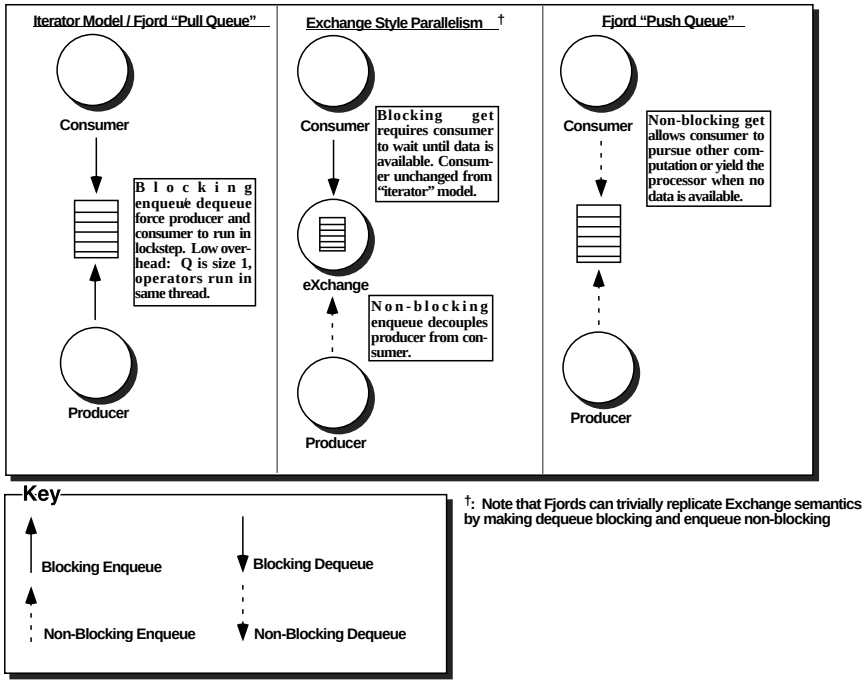

Figure 2: *Producer-Consumer Coupling in Fjords, Iterators, and Exchange.* Notice that Fjords can connect operators in any of the three configurations show here. The "pull-queue" model enables sensor query processing.

this approach limits the ability of the system to adapt to slow, offline, or disconnected producers. The pull-queue approach rectifies this situation by returning control back to the consumer when no data is available so it can perform other computation. Note that, by encapsulating the blocking behavior of the system in the queues between operators, any of these three modes of computation is easily implemented via Fjords.

The Fjords architecture has been used to execute queries over data streamed from traffic sensors along the freeway near UC Berkeley. Fjords allow data from the sensors to be pushed into the query processor, where it is combined with fixed data about traffic accidents to answer queries about current freeway conditions.

2.2 Sensor Proxies

The second major component of our sensor-query solution is the *sensor-proxy*, which acts as an interface between a single sensor and the Fjords querying that sensor. The sensor proxy runs in the query processor on a host-machine, and serves a number of purposes. The most important of these is to shield the sensor from having to individually deliver data to hundreds of interested end-users. It accepts queries and services them on behalf on the sensor, using the sensor's processor to simplify this task when possible.

Another role of the sensor proxy is to adjust the sample rate of the sensors, based on user demand. If users are only interested in a few samples per second, there is no reason for sensors to sample at hundreds of hertz, since lower sample rates are directly proportional to longer battery life. Similarly, if there are no user queries over a sensor, the sensor proxy can ask the sensor to power off for a long period, coming on-line every few seconds to see if queries have been issued.

A third role of the proxy is to direct the sensor to aggregate samples in predefined ways, or to download a completely new program into the sensor if needed. For instance, if the proxy observes that all of the current

user queries are interested only in samples with values above or below some threshold, the proxy can instruct the sensor to not transmit samples outside that threshold, thereby saving communication.

Via simulations (with a microprocessor simulator), I have been able to show that isolating sensors from queries and intelligently aggregating on those sensors can their reduce power consumption (and thus battery life) by more than an order of magnitude. These experiments are a primary motivation for my recent work on in-network aggregation, discussed in section 3 below.

2.3 Continuously Adaptive Continuous Queries (CACQ)

Fjords and Sensor Proxies are aimed at providing the mechanism for bringing sensor data into a database and for isolating sensors from individual queries over the network, but do not address the issue of efficiently evaluating queries over that data once it has been made available. It is expected that some sensor networks will need to support hundreds or thousands of simultaneous queries over them, particularly ones that monitor large shared spaces. Imagine, for instance, a sensor network scattered over freeways in the Bay Area which could be used to provide instantaneous reports of traffic conditions for commuters: thousands of commuters would use this type of system every day. Our CACQ work is focused on adapting existing research on continuous queries to the sensor-network environment.

Continuous queries (CQs) allow users to pose queries over information sources that are updated over time. Rather than forcing users to re-evaluate an entire query when a new value arrives or changes, continuous query systems maintain a set of user queries. When a new tuple arrives, apply the relevant queries to that tuple and ship the results, if any, to the users who posed the queries. The NiagaraCQ [6] continuous query system provides efficient performance by folding together similar query plans into a single, shared query plan which is significantly less expensive to compute than a number of separate queries.

CACQ extends existing shared continuous query systems like NiagaraCQ by introducing two important new ideas: first, they are adaptive, which means that as users pose new queries, queries stop running, or sensors change the delivery rates of tuples to existing queries, the order in which query operators are applied also changes. This adaptivity is in contrast to existing CQ systems, which use a static query optimizer to fix the order of operator execution at the time queries are introduced. Adaptivity in CACQ is obtained via a modified Eddy [4] operator which makes routing decisions on a per tuple basis, altering the flow of tuples though the plan as the number of queries sharing a particular operator changes or the selectivity of join and selection operators shifts.

The second major innovation introduced by the CACQ system is that it applies to streaming data. Streaming data complicates CQ systems by requiring all operators to be non-blocking and to operate for finite time over infinite inputs – sorting, for example, cannot easily be expressed as an operation over streams. Thus, the CACQ system has been equipped with a suite of stream-appropriate operators, such as those discussed in [41].

A paper on the CACQ system will appear in this year's SIGMOD conference[30]; experiment results presented in that paper indicate that the CACQ system is able match or exceed the performance of existing continuous query systems under a variety of workloads. Several experiments were run to measure the ability of CACQ to adapt to changes in query workload. One of them is summarized below: Table 1 shows five simple queries and their time of arrival in the system. All queries are over a data stream S, each tuple of which contains six fields: an *index*, and five randomly generated fields *a,b,c,d,e* and *f*, each of which is randomly and uniformly distributed over the range [0..100].

Figure 3 shows the percentage of tuples routed to each filter over time for the three workloads. The percentage of tuples received is a measure of the routing policy's perceived value of an operator. Highly selective operators are of higher value because they reduce the number of tuples in the system, as are operators which

Table 1: **Queries in Adaptivity Experiment, with Query Arrival Times.**

Workload	Arrival Time (Seconds)
1. `select index from S where a > 10`	0
2. `select index from S where b > 30`	5
3. `select index from S where c > 50`	10
4. `select index from S where d > 70`	15
5. `select index from S where e > 90`	20

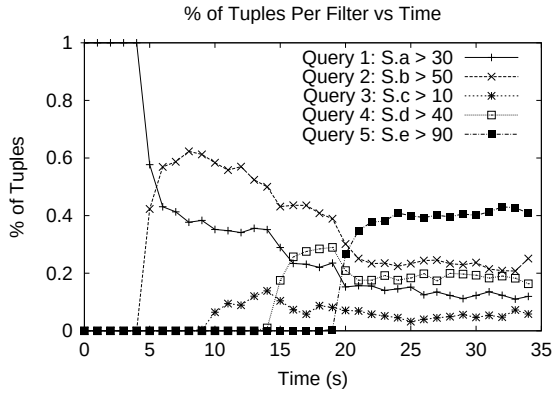

Figure 3: *Percentage of Tuples Routed to Filters Over Time*. Notice that the most selective queries rapidly adapt to receive the most tuples.

apply predicates for many queries, because they perform more net work. Notice how quickly the system adapts to newly introduced filters: in most cases, four seconds after a filter is added the percentage of tuples it receives has reached steady-state.

In this particular workload, the most selective query, Query 5, receives the most tuples in steady-state, as expected. In the paper, we show this adaptivity working in two other, more complex query workloads. Such adaptivity is very important in the context of long-running sensor-network monitoring queries, where selectivities and query workloads may change over the lifetime of queries that are running for hours or days, causing good initial queries optimization decisions to be very sub-optimal by the end of the query.

In the paper, the performance of CACQ is compared to NiagaraCQ, the previous state-of-the-art continuous query processor: CACQ is able to match or beat the performance of the static, non-adaptive, cost-based NiagaraCQ query optimizer and processor over several different query workloads.

2.4 Visualizing Sensor Data

Given the Fjord and CACQ environments for accessing and efficiently processing sensor data, applications demonstrating their usefulness are needed. As an example of such an application, I built a visualization of traffic in the Bay area. This visualization shows data from eight traffic sensors deployed on the freeway near UC Berkeley and visually correlates that data with reports of traffic accidents from the California Highway Patrol (CHP), web-cams available from news sites, and road locations and maps available from the US Census Bureau.

As another example of a visualization environment, Figure 4 shows a screenshot of the user interface for querying this small subset of traffic data. The leftmost panel contains a map of the Bay Area, with the East Bay, near Berkeley, on the right side of the bay. Squares up and down the freeway near Berkeley – which are not visible in a grayscale reproduction so they are labeled with arrows – represent the current state of traffic sensors

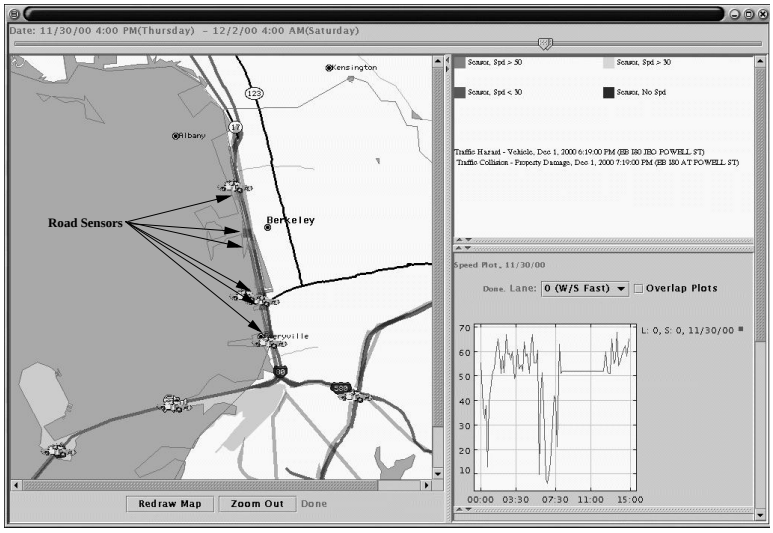

Figure 4: *Traffic Sensor Visualization*

for which we have data. Police-car icons represent traffic incidents the CHP has reported. On the right hand side are graphs of speed and flow which display historical trends in traffic conditions. The slider along the top shows the current time range which is being queried. A write-up on the traffic visualization system is available on-line [26].

Figure 5 shows a simulation of a sensor network built to model the behavior of a network of sensors arranged in a multi-hop topology for computing aggregate queries (see Section 3) below. This visualization facilitates rapid prototyping and debugging of networking algorithms by showing connectivity, data flow, and various network parameters, either as a color-coded "intensity" overlaid on the topology, or as a scatter plot (or both). Figure 5 color codes sensors with their distance from the root (middle) of the network, with lighter nodes being closer. Arrows underneath the cursor, which is currently at the center of the graph, show children, neighbors, and parents (all nodes adjacent to the root are children.) The textual status at the bottom gives numeric information about the sensor under the cursor. The line plot shows the historical average aggregates as computed at several nodes in the network.

3 Query Processing in Sensor Networks

Having an efficient architecture for obtaining and querying sensor data is just one component of the TeleTiny query-processor. In order to conserve precious network bandwidth and battery life on sensors, and reduce the load on the central query processor, some mechanism to evaluate all or part of a query in the network itself is needed.

Work on in-network query processing is being primarily conducted in the context of TinyOS[20], a UC Berkeley effort to develop an operating system for use on wireless sensors. The sensors which the TinyOS project is focused on are referred to as "motes". Figure 6 shows an example of such a sensor. Current generation *Mica* motes have 4kB of RAM, a 50kbit radio, a 4mHz processor, and a modular connector for attaching sensor devices. Sensor boards which measure light, temperature, vibration, sound, and acceleration are available.

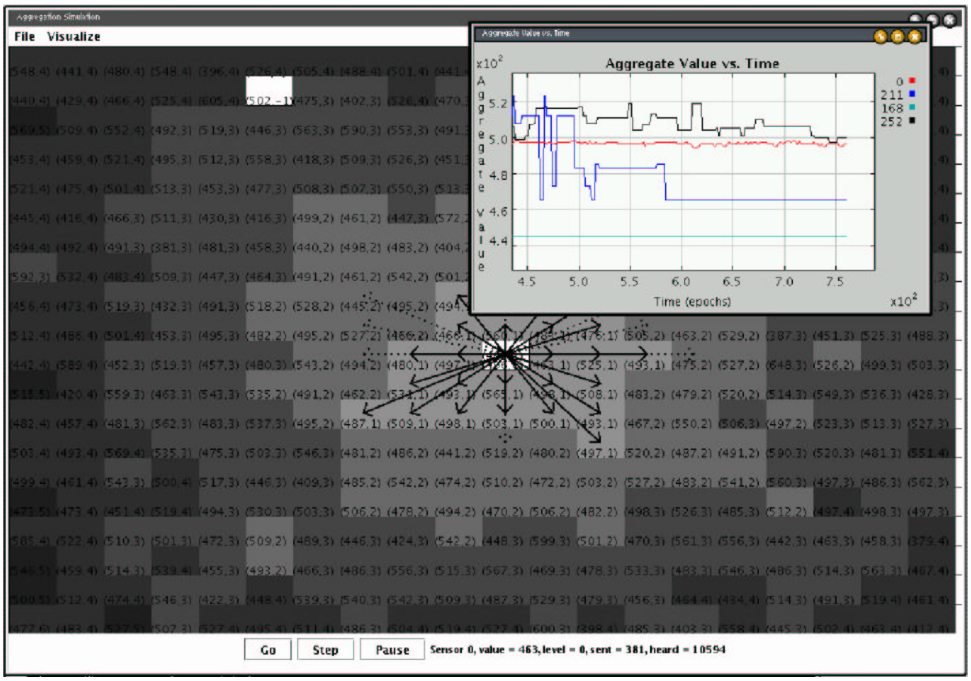

Figure 5: *Visualization of Sensor Network Aggregation Algorithm, with Aggregate Value Plot (inset)* Arrows underneath the cursor (which is over the root of the network) indicate child nodes. As the mouse moves, arrows to show neighbors and parents are also drawn. The graph shows the aggregate value at various sensors - the stable (red) line with value around 500 is the root of the network.

3.1 Sensor Catalog Management

Before any data can be extracted from a sensor network, it is necessary to have some facility to understand the capabilities of each sensor. The sensor catalog is a lightweight layer designed to run on every TinyOS mote to allow it to answer queries about the properties of the mote's on board sensors. A simple text-based schema file which indicates the name, units, and range of each sensor, along with the set of functions to call to extract a sensor value, is compiled into a few bytes of data which is stored in the EEPROM of the mote. When a catalog query is received by the mote, the schema information is used to describe the capabilities of the mote. This capability description can in turn be used by the query-processor to ask the mote for the reading on a specific sensor. As a part of this system, I have also built a simple interface to automatically detect and register sensors

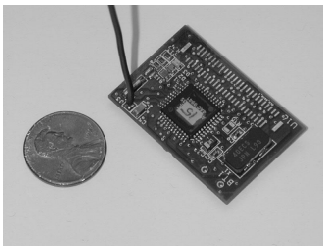

Figure 6: *A TinyOS Sensor Mote*

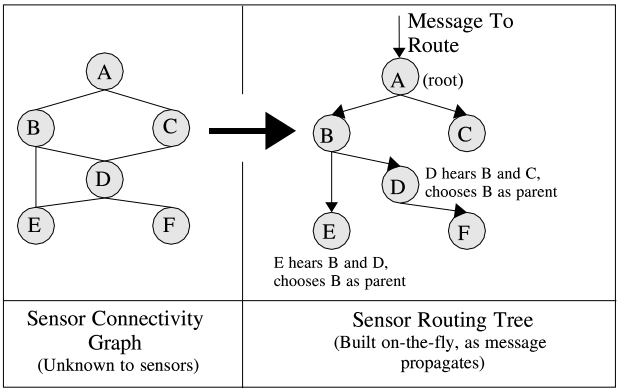

Figure 7: *Inferring a Routing Tree in a Sensor Network*

with the Telegraph query processor.

3.2 In Network Aggregation

Given the sensor-catalog as a way to identify motes, it is possible to explore techniques for doing in-network query processing. The first attempt at such a technique is a system for computing aggregates. Aggregation is the process in which standard statistics are computed over groups of data items. In a sensor network, this involves two tasks: identifying groups, and computing statistics over members of those groups. The goal of in-network aggregation is to reduce the cost of computing aggregates from the naive solution where each sensor reports its values to a central query processor that determines group membership and computes statistics locally.

As a part of my work with the TinyOS group, I have developed a technique called TAG (Tiny AGregation) for computing aggregates in a more efficient way than simply sending all sensor readings to a centralized workstation for processing. The approach works by taking advantage of the routing topology used by sensor networks to propagate messages from one sensor to another, so it is necessary to understand the basic routing algorithm used in large sensor networks. [1]

Routing works as follows: network topologies are represented as connectivity graphs, upon which a routing tree is overlaid. To route a message to some node n, the message is introduced at some *root* node, which broadcasts the message to its children, which in turn broadcast that message to their children, and so on, eventually reaching node n. During this routing process, each node chooses a single parent, thus forming a routing tree which can be used to propagate messages from any node back to the root. When n wants to reply, it sends a message to its parent, which sends a message to its parent, and so on, until the message reaches the root. Figure 7 shows a simple example of how a tree is built on the fly from an undiscovered routing topology as a message propagates from the root to the leaves of the network.

Now, to compute an aggregate, the aggregate is introduced at the root of the network and propagated down the graph of sensors to form a routing tree. Once an aggregate message has propagated down the tree, the leaves propagate up a *partial-aggregate*, the value of the aggregate function is computed over just the local sensor reading. The parents then compute a new partial aggregate which combines their children's values with their own local readings and propagate that value up. This continues to the root of the network, where the final value

[1] Note that the TAG algorithm is independent of a specific routing protocol; the protocol presented here is an example of a commonly used, simple approach.

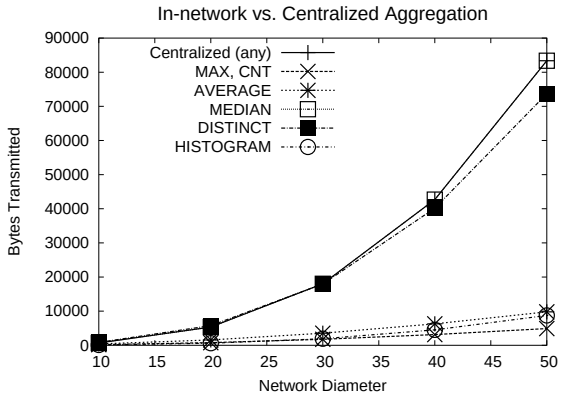

Figure 8: *Messages Sent in TAG vs. Centralized Aggregation*

of the aggregate is computed.

A paper on this work has been submitted to VLDB 2002 [29]. This paper classifies aggregate functions according to their state requirements and semantic properties. This classification partitions aggregates according to their memory requirements and functional properties (e.g. monotonicity) and uses that partitioning to generalize about their behavior in the face of network errors and the applicability of various optimizations. A number of experiments on the storage and communication of each type or aggregate are also presented. Additionally, the paper presents techniques for handling grouping (and the limited storage available on sensor nodes), and for mitigating the effects of loss due to low-quality radio links in the current generation of TinyOS hardware.

Figure 8 (from [29]) shows an example of the benefit of the TAG approach, in terms of the aggregate number of bytes that must be sent by all sensors in the network, for a variety of different aggregation functions. In the simulation used to generate this graph, sensors were placed on a fully-packed grid of the specified diameter; sensor values were randomly selected from the uniform distribution over the range [0,1000]. `MEDIAN`, `MAX`, and `COUNT`, and `AVERAGE` are standard statistical functions; `HISTOGRAM` partitions values into a fixed number of bins; `DISTINCT` counts the total number of distinct sensor values reported. `MEDIAN` requires all values to be sent to the root of the network, and thus its performance matches the centralized algorithm. Other aggregates, however, show a substantial reduction in the number of messages: better than an order of magnitude over the centralized approach in the case of simple aggregates like `MAX` and `COUNT`.

This completes the summary of the completed elements of the TeleTiny system. The next section summarizes related work.

4 Related Work

The TeleTiny system is related to two major branches of research: database research on continuous queries and querying data streams and networking projects that discuss routing, data-shipping, and aggregation in sensor-networks.

The most immediately relevant work is the Cougar [36] project from Cornell for querying sensor-networks. Cougar's primary focus is on the use of Object-Relational abstractions for querying sensor-based systems rather than on architectural aspects of sensor query processing. According to [14], the current implementation of the Cougar system is designed to run on WINS nodes designed by Sensoria Corporation [8], which are large, Linux based devices that are not subject to the same power or communications limitations as the sensor devices being

developed for the TinyOS project. Indeed, the decision by the Cougar group to use XML to encode messages between sensors reveals the lack of bandwidth concerns in their environment, suggesting that the tradeoffs and correct design decisions in Cougar will be substantially different than those in TeleTiny.

4.1 Related Database Projects

Work on CACQ is heavily indebted to a variety of adaptive query processing systems, most notably Eddies. Eddies were originally proposed in [4]. The work was not specifically focused on continuous query processing, although attention to adaptivity and responsiveness in eddy lead to some shared attributes with continuous query systems, such as the use of non-blocking, pipelined operators and the ability to perform mid-query reoptimization via tuple lineage.

Notions of adaptivity and pipelining used throughout my research are well established in the research community. Parallel-pipelined joins, used heavily in CACQ, were proposed in [50]. Adaptive systems such as XJoin, Query Scrambling, and Tukwila [47, 48, 22] demonstrated the importance of pipelined operators to adaptivity. Fjords is closely related to these systems as well, as it seeks to provide a mechanism to unblock query processing operators that might otherwise be forced to wait for blocking operators to return.

Continuous Queries

Existing work on continuous queries provides techniques for simultaneously processing many queries over a variety of data sources. These systems propose the basic continuous query framework adopted in CACQ and also offer some extensions for combining related operators within query plans to increase efficiency. Generally speaking, the techniques employed for sharing work between queries are considerably more complex and less effective at adapting to rapidly changing query environments than CACQ.

Efficient trigger systems, such as the TriggerMan system[17] are similar to continuous queries in that they perform incremental computation as tuples arrive. In general, the approaches used by these systems is to use a discrimination network, such as RETE [13] or TREAT [33], to efficiently determine the set of triggers to fire when a new tuple arrives. These approaches typically materialize intermediate results to reduce the amount of work required for each update.

Continuous queries were proposed and defined in [46] for filtering of documents via a limited, SQL-like language. In the OpenCQ system [27], continuous queries are likened to trigger systems where queries consists of four element tuples: a SQL-style query, a trigger-condition, a start-condition, and an end-condition.

The NiagaraCQ project [6] is the most recently described CQ system. Its goal is to efficiently evaluate continuous queries over changing data, typically web-sites that are periodically updated, such as news or stock quote servers. Users install queries that consist of an XML-QL [9] query as well as a duration and re-evaluation interval. The NiagaraCQ approach to optimizing continuous queries is substantially different from the approach presented in CACQ: grouping in NiagaraCQ is performed by a static query optimizer, and the order in which operators are applied is fixed by this query plan. Although plans can be “dynamically regrouped”, heuristics for when to do so are unclear, and, unlike our continuous adaptivity, regrouping is an expensive operation that cannot be performed frequently.

The XFilter and follow on YFilter system [3, 11] are also recent continuous-query systems. Both are based on XML-documents streaming into the system and being matcher matched to filter-profiles expressed in the XPath [7] language. They optimizes queries by indexing profiles based on the filter conditions that appear within those profiles. Thus, when a new XML document arrives in the system, each of its tags is matched against this filter-condition index to rapidly determine which profiles have conditions that need to be checked

against the document: essentially, it is an extremely efficient predicate index for matching XML-documents and XPath predicates. It is explicitly focused on handling the structure of XML and does not address joins or adaptivity.

The problem of sharing work between queries is not new. Multi-query optimization, as discussed in [40] seeks to exhaustively find an optimal query plan, including common subexpression, between a small number of queries. Recent work, such as [37, 34] provides heuristics for reducing the search space, but is still fundamentally based on the notion of building a *query-plan*, which we avoid in this work.

Streams

The query processing architecture used in this paper is essentially a stream-processor. Fundamental notions of time-series processing are developed in SEQ[41], including extensions to SQL for windows and discussions of non-blocking and timestamped operators. SVP [35] discusses the semantics of stream processing and presents a nice overview of parallel-processing in stream-based environments. DeWitt, et al. [10] propose windows as a means of managing joins over very large sets of data. Tribeca [44] discusses operators for processing streams in the context of network routing; it includes an interesting discussion of appropriate query languages for streaming data. There are many other relevant languages in models from the Temporal Database literature; see [43] for a survey of relevant work.

Aggregation

As a distributed database-style aggregation engine, the TAG component of TeleTiny bears some superficial similarity to existing distributed query processors. Indeed, the database community has proposed a number of distributed and push-down based approaches for aggregates in database systems [42, 51], but as discussed in the TAG paper[29], most of these assume a well-connected, low-loss topology that is unavailable in sensor networks; such assumptions make the techniques developed in these papers largely inapplicable.

4.2 Related Networking Projects

The TAG and TeleTiny architectures combine elements of databases and networking to efficiently process queries in the sensor network. Literature on active networks [45] discusses the idea that the network could simultaneously route and transform data, rather than simply serving as an end-to-end data conduit. Within the sensor network community, work on networks that perform data analysis has been largely confined to the USC/ISI and UCLA communities. Their work on directed diffusion [21] discusses techniques for moving specific pieces of information from one place in a network to another, and proposes aggregation-like operations that nodes may perform as data flows through them. Heidemann et al. [18] propose a scheme for imposing names onto related groups of sensors in a network, in much the way that our scheme partitions sensor networks into groups. These papers recognize that aggregation dramatically reduces the amount of data routed through the network but are focused on application specific solutions that, unlike the approach in TAG, are not obviously applicable or efficient in a wide range of environments.

Networking protocols for routing data in wireless networks are very popular within the literature [24, 1, 15], however, none of them address higher level issues of data processing, merely techniques for data routing. Our tree-based routing approach is clearly inferior to these approaches for peer to peer routing, but works well for the aggregation scenarios we are focusing on.

Given this overview of related work, the remainder of this paper is devoted to a discussion of the remaining elements of the TeleTiny system that remain to be built and studied and a brief timeline outlining the completion of these projects and my dissertation.

5 Research Plan and Timeline

The techniques discussed in this paper are the beginnings of a full-fledged system to extract data from and execute queries over streams of data flowing from sensor networks. To increase their value, however, there are a number of future directions I plan to explore for the remainder of my Ph.D.

The major push will be toward the construction of a real TeleTiny implementation that combines TAG and the schema interface running on Berkeley Mica sensor motes with an adaptive query processing system such as the Telegraph query processor. This construction effort will yield two significant pieces of research. The first will be an evaluation of the performance and common-uses of my software as deployed in a real sensor-network running on tens or hundreds of motes. Such a network is currently being deployed in the Intel research lab in downtown Berkeley, and my summer employment will involve integrating TeleTiny (with instrumentation for data collection) into this network and gather statistics about how it is used by lab denizens.

The second major research effort will be a characterization of the resource constraints that arise in sensor query processing, and a set of techniques for limiting use of the resources and expressing constraints to users. There are a number of resource-constraint problems that arise in TeleTiny: power, radio-bandwidth, and memory are all limited resources that could easily be exhausted under heavy query workloads. Understanding how to conserve these resources, and what control and feedback needs to be given to the user to maximize that conservation is an important research area that will dramatically affect the usefulness of any sensor query processing system. Some of the techniques needed in this area come directly from other sensor research: entering a very low power state between data samples or radio-communications, for instance, is a goal of all TinyOS software. Other techniques, such as suppressing values that change very little or that are superseded by the values of neighboring sensors, are approaches unique to TeleTiny.

Development Effort

Pursuing this research requires a major development effort, as neither of the above projects can be properly done without an actual system: simulations, though valuable to a certain extent, are simply not capable of modeling radio loss or interference, embedded CPU load, or power consumption accurately. As with previous efforts, TeleTiny development can be broadly partitioned into sensor-software tasks and query processor tasks. On the sensor side, this includes:

1. *On-mote software:* I am currently developing a suite of software components that allow database style queries, consisting of grouped aggregates and selection predicates, to be pushed down into a network of motes running TinyOS. This software includes the infrastructure for forwarding queries and results between motes, scheduling and sharing the delivery of data for multiple simultaneous queries, and interfacing to the on-board catalog interface that allows motes to determine what data fields they can provide to a query.

2. *User defined functions:* One of the key features of the in-mote aggregation system is the ability to support user-defined filters and aggregates. These take the form of virtual machine functions pushed into the sensors as Maté [25] (a TinyOS virtual machine) programs; an effort is currently underway to integrate Maté

into TeleTiny. This capability is particularly important in the context of sensors, as many of the potential applications have a signal-processing flavor, in which unconventional (from a database standpoint) filters and transformations are needed.

In addition to these pieces of sensor software, several mote-to-DBMS interfaces need to be designed and implemented before a fully functional sensor-query processor can be deployed:

1. *Catalog Server*: Although a facility for managing catalog information on-board sensors was described above, conventional (centralized) database catalog designs are not made to support thousands of devices that come and go frequently. Furthermore, most sensor queries will not be of the form *select value from sensor x* but rather *select value from sensors with property x*. Designing a catalog that can handle both of these cases is a requirement before a fully usable sensor-query system can be implemented. Wei Hong (at Intel Berkeley) and I have designed the interfaces for such a system. An initial implementation is under construction at the Intel lab; I hope to be able to leverage this development.

2. *Integration into Telegraph Wrapper Interface*: The Telegraph database system includes an interface for heterogeneous data sources that allows them to be "wrapped" into native relational tables. Integrating sensor data into Telegraph via this interface is an important step toward building an integrated query processing system (fully realizing this integration requires the sensor-proxy, described below.)

3. *Sensor Proxy Policies and Query Optimizer:* The sensor-proxy, as discussed above, serves as the interface between motes and a more conventional DBMS by distributing queries between sensors and the database system. The mechanism for performing this distribution is fairly straightforward (as described in [28] and [29]), but still needs to be built.

   Furthermore, there will be situations where the resource limitations of sensors will preclude them from executing certain types of queries, or where there will be so many queries that not all of them can be executed on the motes. In these situations, some policy for choosing which queries to push down and which to execute at the DBMS is needed. One possible formulation for this policy is as a cost-based query-optimization problem (as in [39]); by using such a formulation, it should be possible to determine a power-efficient partitioning of queries between motes and the main DBMS.

Finally, there are some query-processor issues that are closely related to problems in sensor-query processing that I have been exploring in conjunction with the Berkeley database group:

1. *Historical Data Interface*: The current Telegraph system does not provide a mechanism for logging and querying historical sensor data, or for combining historical data with data currently streaming into the system. Over this summer, I plan to work with the Telegraph research group to insure that the next implementation of the Telegraph system includes support for logging and querying historical sensor data. One possible approach for implementing such an interface is to use *broadcast-disk* [2] like data scheduling to stream historical records needed by the current query workload when queries are blocked waiting for additional current-time results to be delivered.

2. *Query Model and Streaming Data Semantics*: Sirish Chandrasekaran and I are currently working to enumerate the space of query semantics over data streams. We hope to integrate this work into the Telegraph query engine, which will provide a richly expressive tool for querying data flowing in from sensors.

Real-world Performance Study

As discussed previously, one significant contribution of this work will be a performance characterization of the unified system and its various sub-systems. This characterization will take the form of a number of empirical measurements of the parameters of the system in a real-world environment, namely the building monitoring application currently being deployed at the Intel-Berkeley lab. By deploying a real system in the lab, and allowing lab occupants to execute queries over the system for an extended period of time, it should be possible to measure how people would actually use an interactive sensor query processing system. In addition to validating the usefulness of my thesis, a "user-study" of this sort would be an extremely valuable asset, as there are many assumptions made by current research projects on streaming and continuous queries (both at Berkeley and other universities) about rate of data arrival, commonality of work between queries, loss rates, etc., that may or may not be valid. Measurements that would ideally come from a performance study include:

1. *Types of Queries*: What kinds of queries are people interested in running over such a network? Are the queries typically over the current state of the system, or over the historical state of the sensors? What data rates do users want? To what extent is sharing (in the style of CACQ) possible given this query workload?

2. *Loss Characteristics*: What percentage of data that sensors send actually makes it to the root of the network? If mechanisms to improve reliability (such as retransmission) are used, how effective are they and what is their power overhead?

3. *Power Characteristics*: What is the power drain on the sensors? How do different types of queries affect that power drain?

4. *Amount of Storage*: What are the storage demands placed on the sensors themselves? TAG requires that sensors keep a small cache of the aggregate values of their children. How big is that cache typically (in terms of bytes?) Are there situations where the memory demands of the queries exhaust the available RAM?

5. *Server Load*: At what rate is data actually being delivered to the server-based query processor? Is the performance of the query processing system significant? Of the queries running at a given time, how many can be executed in the network and how many must be run centrally?

6. *Variability in Data Rates and Selectivities*: How much variability is there in data rates from different sensors? How do the selectivities of various selection predicates vary over time, and are there correlations with external phenomena (e.g. day of week, time of day)? Determining if such variability exists will help to show whether adaptive query processing techniques of the sort proposed by the Telegraph project are actually necessary.

7. *Lifetime of Queries*: How long do typical queries run? Are users mostly just looking for snapshots of the current state of the network, or do they pose long running monitoring queries that look for particular phenomena?

Resource Mitigation

As discussed above, the second additional research component of my thesis will involve characterizing the resource limitations of sensors and sensor-query processors and a set of techniques to deal with these limitations. Possible limitations include:

1. *Power*: Any battery-based sensor network must strive to reduce power consumption whenever possible, since as soon as power is exhausted, query processing ends. Reducing communication, sample rate, and sleeping the processor are the primary techniques for power reduction. Such techniques must be built into TeleTiny from the ground up.

2. *Radio Communication*: Radio bandwidth is limited. It is may not be possible to deliver data at the user requested rate from all sensors in the network. Techniques to aggregate and reduce communication (such as TAG), as well as notify the user and allow him or her to adapt his query to changing bandwidth availability are needed. Furthermore, radio communication is inherently lossy; some techniques to mitigate such losses (such as retransmission) are possible, but the query processor must also notify the user that loss is happening and provide information about the magnitude and location of such loss.

3. *Sensor CPU*: In current research [20, 28, 29] the load on the sensor CPU is considered secondary to the communication costs of algorithms, since communication tends to dominate power consumption. However, understanding what proportion of power consumption is due to the CPU, and what tradeoffs in query processing can be made to reduce that consumption is important, particularly if the radio is used infrequently.

4. *Sensor RAM*: With only 4K of RAM, Motes are very limited in what they can store. Understanding the memory requirements of various algorithms, such as TAG, and the places where additional memory can be used to reduce communication or increase performance, is an important contribution.

5. *Host CPU:* Although it seems that the small quantity of data flowing in from TinyOS-style motes would not be enough to overwhelm a modern PC, the host PC must multiplex all user queries over the sensor data streaming into the system, and combine that data with other data sources users may have queried. If there are enough queries, the non-sensor data is large enough or the sensor communication channel is augmented to deliver more bandwidth, the centralized host PC could become a bottleneck.

6. *Host Storage:* Streams, even low bandwidth streams from sensor networks, are infinite and will eventually exhaust the storage capacity of a host PC.

Thus, there are many resources at play in TeleTiny. Part of the performance study discussed in the previous section will be an effort to determine which of these resources are actually limited in the building monitoring scenario.

Some of the techniques for dealing with resource limitations were proposed in earlier work: for instance, TAG deals directly with the bandwidth and power costs of query processing, and proposes several techniques to increase the reliability of communications in the TeleTiny environment. Similarly, CACQ is an effort to reduce query processor CPU utilization and RAM requirements in streaming continuous query environments.

It is less well understood how to handle other types of resource limitations. For example, because streams are effectively infinite, they can easily exhaust storage capacity of a host query processor, even one with a big disk. Some set of techniques for expressing to the user the subset of the data that is still in the system, as well as techniques for summarizing and expiring old data, are needed. The STREAM project at Stanford [5] discusses summarization techniques, but more work is needed before its clear how or when to apply such summaries.

Summary

Fjords [28], CACQ [30], and TAG [29], along with the sensor catalog and visualization systems I have built form a solid research foundation for sensor query processing. However, to complete the research projects described

above, and empirically verify the ideas presented in these papers, a complete system must be built, deployed, and studied.

Thus, my primary research agenda is to develop the TeleTiny query processing system for streaming sensor data. There are two key components to TeleTiny: an on-sensor query engine for computing aggregates and selections in a power-efficient manner; and a server-side query-processor specially designed to handle streaming data and continuous queries. Furthermore, the interface between these two components is very important, serving two key roles: first, it allows sensor data to be treated like other types of streaming data without squandering the limited power available to the motes and, second, it interfaces the database's catalog the internal schema's of the neighboring motes.

Along with this core system, there are several important research problems the TeleTiny system will help to answer. The first is a characterization of the real-world query and data workloads that are placed on the system with an understanding of the bottlenecks and demands of query processing in a sensor environment. The second is set of techniques (and expressions of those techniques to the end-user) that can be used to control and limit the utilization of the very scarce power, memory, and bandwidth available to sensors.

Timeline

I plan address each of the above research tasks in the next 6 - 9 months, with the goal of completing my thesis in the Spring or Summer of 2003. The following timeline maps out the schedule of research:

- *May - June, 2002*: Complete sensor-side software with User-Defined Functions, Schema API, and Catalog Server (with Wei Hong.)
- *June 2002*: Submit ICDE paper on streaming semantics (with Sirish Chandrasekaran).
- *June - August, 2002*: Integrate with Telegraph (or its successor), build sensor proxy with some initial set of policies, deploy lab monitoring application, and collect data on usage patterns in the network (with Wei Hong, Intel Berkeley, and Telegraph research group.)
- *August - November, 2002*: Integrate with historical data interface of Telegraph. Assuming Telegraph engine implementation goes as planned, this should be straightforward (with Telegraph research group.)
- *November 2002*: SIGMOD paper on real-world measurements from sensor-networks.
- *August - January, 2003*: Explore and implement mechanisms for handling resource constraints.
- *February, 2003*: VLDB paper on resource constraints.
- *February - May 2003*: Complete Dissertation.

References

[1] W. Adjue-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM SOSP*, December 1999.

[2] D. Aksoy, M. J. Franklin, and S. Zdonik. Data staging for on-demand broadcast. In *VLDB*, Rome, September 2001.

[3] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *International Conference on Very Large Data Bases*, September 2000.

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

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Modes and issues in data stream systems. In *ACM PODS*, 2002.

[6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*, 2000.

[7] J. Clark and S. DeRose. XML path language (XPath) version 1.0, November 1999. http://www.w3.org/TR/xpath.

[8] S. Corporation. WINS NG 2.0 platform status. Presentation, April 2001. http://www.darpa.mil/ito/research/proceedings/sensit2001apr/WINS_NG2 Sensoria-200104.pdf.

[9] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML, 1998. http://www.w3.org/TR/NOTE-xml-ql.

[10] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, Barcelona, Spain, 1991.

[11] Y. Diao, M. Franklin, H. Zhan, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. Submitted for Publication.

[12] D. Estrin, L. Girod, G. Pottie, and M. Srivastava. Instrumenting the world with wireless sensor networks. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001)*, Salt Lake City, Utah, May 2001.

[13] C. Forgy. RETE: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[14] J. Gerhke, M. Calimlim, W. F. Fung, and D. Sun. Cougar design and implementation. Design Documnent. http://www.cs.cornell.edu/database/cougar/CougarDesignDoc.htm.

[15] T. Goff, N. Abu-Ghazaleh, D. Phatak, and R. Kahvecioglu. Preemptive routing in ad hoc networks. In *ACM MobiCom*, July 2001.

[16] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[17] E. Hanson, N. A. Fayoumi, C. Carnes, M. Kandil, H. Liu, M. Lu, J. Park, and A. Vernon. TriggerMan: An Asynchronous Trigger Processor as an Extension to an Object-Relational DBMS. Technical Report 97-024, University of Florida, December 1997.

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

[19] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

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

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

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

[23] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile networking for smart dust. In *MOBICOM*, Seattle, WA, August 1999.

[24] J. Kulik, W. Rabiner, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *MobiCOM*, Seattle, WA, 1999.

[25] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. Submitted for Publication.

[26] F. Li, S. Madden, and M. Thomas. Traffic sensor data visualzation. Project Report. http://www.cs.berkeley.edu/ mct//infovis/project/traffic.html, December 2000.

[27] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.

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

[29] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. TAG: Tiny AGgregate Queries in Ad-Hoc Sensor Networks. Submitted for Publication, 2002.

[30] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continously adaptive continuous queries over data streams. In *ACM SIGMOD*, Madison, WI, June 2002. To Appear.

[31] S. Madden, R. Szewczyk, M. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. Submitted, Workshop on Mobile Computing and Systems Applications, 2002.

[32] R. Min, M. Bhardwaj, S.-H. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan. An architecture for a power-aware distributed microsensor node. In *IEEE Workshop on Signal Processing Systems (SiPS '00)*, October 2000.

[33] D. P. Miranker. TREAT: A better match algorithm for ai production system matching. In *Proceedings of AAAI*, pages 42–47, 1987.

[34] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD*, 2001.

[35] D. S. Parker, E. Simon, and P. Valduriez. SVP - a model capturing sets, streams, and parallelism. In *VLDB*, Vancouver, British Columbia, 1992.

[36] P.Bonnet, J.Gehrke, and P.Seshadri. Towards sensor database systems. In *2nd International Conference on Mobile Data Management, Hong Kong*, January 2001.

[37] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD*, pages 249–260, 2000.

[38] P. Saffo. Sensors: The next wave of innovation. *Communications of The ACM*, 40(2):92 – 97, February 1997.

[39] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. pages 23–34, Boston, MA, 1979.

[40] T. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 1986.

[41] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *VLDB*, 1996.

[42] A. Shatdal and J. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*, 1995.

[43] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publisher, 1995.

[44] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[45] D. Tennenhouse. Active networks. In *OSDI*, October 1996.

[46] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *ACM SIGMOD*, pages 321–330, 1992.

[47] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, pages 27–33, 2000 2000.

[48] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *ACM SIGMOD*, 1998.

[49] M. Weiser. Some computer science problems in ubiquitous computing. *Communications of the ACM*, July 1993.

[50] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77, December 1991.

[51] W. P. Yan and P. Å. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.