Fjording the Stream: An Architecture for Queries over Streaming Sensor Data

Samuel Madden, Michael J. Franklin
University of California, Berkeley
{madden, franklin}@cs.berkeley.edu

Abstract

If industry visionaries are correct, our lives will soon be full of sensors, connected together in loose conglomerations via wireless networks, each monitoring and collecting data about the environment at large. These sensors behave very differently from traditional database sources: they have intermittent connectivity, are limited by severe power constraints, and typically sample periodically and push immediately, keeping no record of historical information. These limitations make traditional database systems inappropriate for queries over sensors. We present the Fjords architecture for managing multiple queries over many sensors, and show how it can be used to limit sensor resource demands while maintaining high query throughput. We evaluate our architecture using traces from a network of traffic sensors deployed on Interstate 80 near Berkeley and present performance results that show how query throughput, communication costs, and power consumption are necessarily coupled in sensor environments.

1. Introduction

Over the past few years, a great deal of attention in the networking and mobile-computing communities has been directed towards building networks of ad-hoc collections of sensors scattered throughout our environment. Researchers at UC Berkeley [15] and other universities have 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 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 sensor readings produced by these collections of sensors. Sensor networks will produce very large amounts of data, which needs to be combined and aggregated to analyze and react to the world. Clearly, the ability to apply traditional data processing languages and operators to this sensor data is highly desirable. Unfortunately, standard DBMS assumptions about the characteristics of data sources do not apply to sensors, so a significantly different architecture is needed.

There are two primary differences between sensor based data sources and standard database sources. 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, partly because it is often 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 over-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 which the query engine needs to conserve whenever possible.

Our contribution to the problem of querying sensor data operates on two levels: First, we propose an enhanced query plan data structure called Fjords ("Framework in Java for Operators on Remote Data Streams"), which allows users to pose queries that combine streaming, push-based sensor sources with traditional pull-based sources. To execute these queries, our system provides non-blocking and windowed operators which are suited to streaming data. Second, we propose power-sensitive Fjord operators called *sensor proxies* which serve as mediators between the query processing environment and the physical sensors.

Sensor data processing and the related area of query processing over data streams have been the subject of increasing attention recently. Systems groups around the country are providing key technology that will be necessary for data intensive sensor based applications. Our work differs, however, in that it is focused on providing the *underlying systems architecture* for sensor data management. Thus, our focus is on the efficient, adaptive, and power sensitive infrastructure upon which these new query processing approaches can be built. To our knowledge, this is the first

This work has been supported in part by the National Science Foundation under the ITR grant IIS00-86057, by DARPA under contract #N66001-99-2-8913, and by IBM, Microsoft, Siemens, and the UC MICRO program.

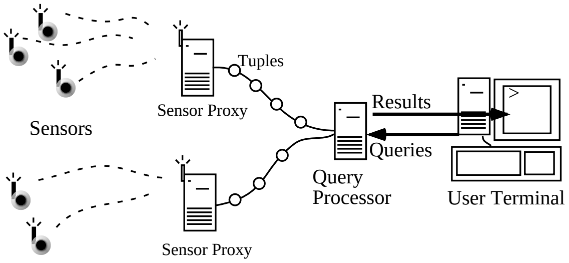


Figure 1. *Environment For Sensor Query Processing*

work addressing the low level database engine support required for sensor centric data-intensive systems. We address related work in more detail in Section 6.

We now present an overview of the sensor query processing environment and discuss the sensor testbed that we are building. In the remaining sections, we present the specific requirements of sensor query processing, propose our solutions for satisfying those requirements, present some initial performance results, discuss related work, and conclude.

2. Background

2.1. Sensor Environment

We focus on sensor processing environments in which there are a large number of fairly simple sensors over which users want to pose queries. For our purposes, a sensor consists of a remote measurement device that provides data at regular intervals. A sensor may have some limited processing ability or configurability, or may simply output a raw stream of measurements. Because sensors have at best limited capabilities, we do not ask them to parse queries or keep track of which clients need to receive samples from them: they simply sample data, aggregate that data into larger packets, and route those packets to a data processing node, which is a fixed, powered, and well connected server or workstation with abundant disk and memory resources, such as would be used in any conventional database system.

We call the node that receives sensor data the sensor's *proxy*, since it serves as that sensor's interface into the rest of the query processor. Typically, one machine is the proxy for many sensors. The proxy is responsible for packaging samples as tuples and routing those tuples to user queries as needed. The general query environment is shown in Figure 1. Users issue queries to the server; the server processes the query, instantiates operators and locates sensor proxies, and starts the flow of tuples. Although sensors do not directly participate in query processing, their proxy can adjust their sample rate or ask them to perform simple aggregation before relaying data, which, as we will show, is an important aspect of efficiently running queries over many sensors.

We are building this system as a part of the Telegraph data flow processing engine [23]. We have extended this system with our Fjords data flow architecture. In Telegraph, users pose queries at a workstation on which they expect results to appear. That workstation translates queries into Fjords through a process analogous to normal query optimization. Queries run continuously because streams never terminate; queries are removed from the system only when the user explicitly ends the query. Results are pushed from the sensors out toward the user, and are delivered as soon as they become available.

Information about available sensors in the world is stored in a catalog, which is similar to a traditional database catalog. The data that sensors provide is assumed to be divisible into a typed schema, which users can query much as they would any other relational data source. Sensors submit samples, which are keyed by sample time and logically separated into fields; the proxy converts those fields into native database tuples which local database operators understand. In this way, sensors appear to be standard object-relational tables; this is a technique proposed in the Cougar project at Cornell[18], and is consistent with Telegraph's view of other non-traditional data sources, such as web pages, as relational tables.

2.1.1. Traffic Sensor Testbed. We have recently begun working with the Berkeley Highway Lab (BHL), which, in conjunction with the California Department of Transportation (CalTrans), is deploying a sensor infrastructure on Bay Area freeways to monitor traffic conditions. The query processing system we present is being built to support this infrastructure. Thus, in this paper, we will use traffic scenarios to motivate many of our examples and design decisions.

CalTrans has embedded thousands primitive sensors on Bay Area highways over the past few decades. These sensors consist of inductive loops that register whenever a vehicle passes over them, and can be used to determine aggregate flow and volume information on a stretch of road as well as provide gross estimates of vehicle speed and length. Typically these loops are used to monitor specific portions of the highway by placing data collection hardware at sites of interest. With several thousand sensors streaming data, efficient techniques for executing queries over those streams are crucial.

2.1.2. Next-Generation Traffic Sensors. Current research on sensor devices is focused on producing very small sensors that can be deployed in harsh environments (e.g. the surface of a freeway.) Current sensor prototypes at UC Berkeley, MIT, and UCLA share similar properties: they are very small, wireless radio-driven, and battery powered, with a size of about 10 cm^3. The ultimate goal is to produce devices in the 1 mm^3 range – about the size of a gnat [15]. We refer to such miniscule sensors as "motes". Current prototypes use 8bit microprocessors with small amounts of RAM (less than 32kBytes) running from 1 to 16 MHz, with a small radio capable of transmitting at tens of kilobits per second with a range of a few hundred feet[12]. As sensors become miniaturized, they will be small enough that they could be scattered over the freeway such that they will not interfere with cars running over them, or could fit easily between the grooves in the roadway. Sensors could be connected to existing inductive loops, or be augmented with light or pressure sensors that could detect whenever a car passed over them.

One key component of this new generation of sensors is that they are capable of on board computation, which may take the form of simple filtering or aggregation. This ability allows the computational burden on the server and the communications burden on the motes to be reduced. For example, rather than directly transmitting the voltage reading from a light sensor many times a second, motes could transmit a count of the number of cars which have passed over them during some time interval. This aggregation reduces the amount of communication required and saves work at the central server. In an environment with tens of thousands of sensors, the benefits of such reductions can be substantial.

2.2. Requirements for Sensor Query Processing

In this section, we focus on the properties of sensors and streaming data that must be taken into account when designing the low level infrastructure needed to efficiently integrate streaming sensor data into a query processor. There are other important issues in sensor stream processing, such as query language and algebra design and the detailed semantics of stream operators, which are not the focus of this work. We discuss these in more detail in Section 6.

2.2.1. Limitations of Sensors. Limited resource availability is an inherent property of sensors. Scarce resources include battery capacity, communications bandwidth, and CPU cycles. Power is the defining limit: it is always possible to use a faster processor or a more powerful radio, but these require more power which often is not available. Current small battery technology provides about 100mAh of capacity. This is enough to drive a small Atmel processor, like the one used in several wireless sensor prototypes, at full speed for only 3.5 hours. Similarly, the current TRM-1000 radio, also used in many sensor prototypes, uses about 4 μJ per bit of data transmitted: enough to send just 14MB of data using such a battery. In practice, the power required for sending data over the wireless radio is the dominant cost [19], so it is often worth spending many CPU cycles to conserve just a few bytes of radio traffic.

2.2.2. Streaming Data. Another property of sensors is that they produce continuous, never ending streams of data (at least, until the sensors run out of battery power!). Any sensor query processing system needs to be able to operate directly on such data streams. Because streams are infinite, operators can never compute over an entire streaming relation: i.e. they cannot be *blocking*. Many implementations of traditional operators, such as sorts, aggregates, and some join algorithms therefore, cannot be used. Instead, the query processor must include special operators which deliver results incrementally, processing streaming tuples one at a time or in small blocks.

Streaming data also implies that sensors *push* data into a query plan. Thus, the conventional pull-based iterator model [8] does not map well onto sensor streams. Although possible, implementing a strict iterator model-like interface on sensors requires them to waste power and resources. To do so, sensors must keep the receivers on their radios powered up at all times, listening for requests for data samples from multiple user queries. Power, local storage, and communications limitations make it much more natural for sensors to deliver samples when those samples become available.

Since many sensors have wireless connections, data streams may be delivered intermittently with significant variability in available bandwidth. Even when connectivity is generally good, wireless sensor connections can be interrupted by local sources of interference, such as microwaves. Any sensor database system needs to expect variable latencies and dropped or garbled tuples, which traditional databases do not handle. Furthermore, because of these high latencies, an operator looking for a sensor tuple may be forced to block for some time if it attempts to pull a tuple from the sensor. Thus, operators must process data only when sensors make it available.

2.2.3. Processing Multiple Queries. Sensors pose additional difficulties when processing complex or concurrent queries. In many sensor scenarios, multiple users pose similar queries over the same data streams. In the traffic scenario, for example, commuters will want to know about road conditions on the same sections of road, and so will issue queries against the same sensors. Since streams are append-only, there is no reason that a particular sensor reading should not be shared across many queries. As our experiments in Section 4.3 show, this sharing greatly improves the ability of a sensor query system to handle many simultaneous queries.

Furthermore, the demands placed on individual sensors vary based on time of day, current traffic conditions, and user requirements. At any particular time users are very interested in some sensors, and not at all interested in others. A query processing system should be able to account for this by dynamically turning down the sample and data delivery rates for infrequently queried sensors.

3. Solution

Having described the difficulties of integrating sensor streams into a query processor, we now present our solution: Fjords and sensor proxies.

3.1. Fjords: Generalized Query Plans for Streams

A Fjord is a generalization of traditional approaches to query plans: operators export an iterator-like interface and are connected together via local pipes or wide area queues. Fjords, however, also provide support for integrating streaming data that is pushed into the system with disk-based data which is pulled by traditional operators. As we will show, Fjords also allow combining multiple queries into a single plan and explicitly handle operators with multiple inputs and outputs.

Previous database architectures are not capable of combining streaming and static data. They are either strictly pull-based, as with the standard iterator model, or strictly push based, as in parallel processing environments. We believe that the hybrid approach adopted by Fjords, whereby streams can be combined with static sources in a way which varies from query-to-query is an essential part of any data processing system which claims to be able to compute over streams.

Figure 2 shows a Fjord running across two machines, with the left side detailing the modules running on a local machine. Each machine involved in the query runs a single *controller* in its own thread. This controller accepts messages to instantiate *operators*, which include the set of standard database modules – join, select, project, and so on. The controller also connects local operators via *queues* to other operators which may be running locally or remotely. Queues export the same interface whether they connect two local operators or two operators running on different machines, thus allowing operators to be ignorant of the nature of their connection to remote machines. Each query running on a machine is allocated its own thread, and that thread is multiplexed between the local operators via procedure calls (in a pull-based architecture) or via a special *scheduler* module that directs operators to consume available inputs or to produce outputs if they are not explicitly invoked by their parents in the plan. The Fjord shown in Figure 2 was instantiated by a message arriving at the local controller; it applies two predicates to a stream of tuples generated by joining a sensor stream with a remote disk source.

In the rest of this section, we discuss the specific architecture of operators and queues in Fjords.

3.1.1. Operators and Queues. Operators form the core computational unit of Fjords. Each operator O has a set of input queues, Q_i and a set of output queues Q_o. These outputs route results of a single select or join, or an entire query, to multiple upstream operators or end users. O reads tuples in any order it chooses from Q_i and outputs any number of tuples to some or all of the queues in Q_o. This definition of operators is intentionally extremely general: Fjords are a dataflow architecture that is suitable for building more than just traditional query plans. For instance, in Section 4 we discuss folding multiple queries into a single Fjord; this is accomplished by creating operators with multiple outputs.

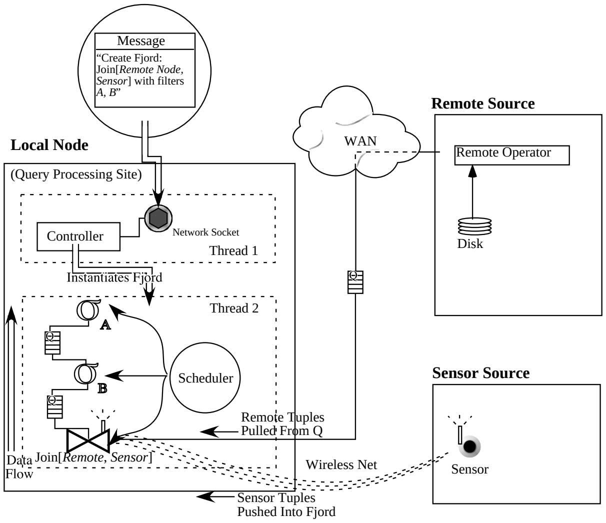


Figure 2. *A Fjord: Join[*`Remote Node`*,*`Sensor`*] with filters* A *and* B

Queues are responsible for routing data from one operator (the *input operator*) to another (the *output operator*.) Queues have only one input and one output and perform no transformation on the data they carry. Like operators, queues are a general structure. Specific instances of queues can connect local operators or remote operators, behave in a push or pull fashion, or offer transactional properties which allow them to guarantee exactly once delivery.

Naturally, it is possible for a queue to fill. When this happens, one has no choice but to discard some data: as is the case with network routers and multimedia streams, it is not possible to pause a stream of real-world data. The choice of which data to discard is handled via an application specific callback from the queue. For many sensors, the correct scheme is to discard the oldest samples first, as they are least relevant to the current state of the sensor.

3.1.2. Flexible Data Flow. 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: 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 (by calling its $transition$ method) in response to a get call on the part of the output operator.

Push queues make sensor streams feasible. When a sensor tuple arrives at a sensor proxy, that proxy pushes those tuples onto the input queues of the queries which use it as a source. The operators draining those queues never actively call the sensor proxy; they merely operate on sensor data as it is pushed into them. With this high level view of Fjords in mind, We now present a detailed view of the internals of a Fjord operator.

3.1.3. State Based Execution Model. The programming model for operators is based upon state machines: each operator in the query plan represents a state in a transition diagram, and as such, an operator is required to implement only a single method: $o \leftarrow transition[s, i]$ which, given a current state s and some set of inputs i causes the operator to transition to a new state (or remain in the same state) and possibly produce some set of output tuples o. Implicit in this model of state programming is that operators do not block: when an operator needs to poll for data, it checks its input queue once per $transition$ call, and simply transitions back to s until data becomes available. Alternatively, some operators, such as Eddy [3] and XJoin [26] are designed to pursue other computation (e.g. transition to states other than s) when no data is available on a particular input; the Fjords programming model is naturally suited to such adaptive query processing techniques.

Formulating query plan operators as a state machine presents several interesting issues. It leads to operators which are neither "push" nor "pull": they simply look for input and operate on that input when it is available. Traditional pull-based database semantics are implemented via the queue between two operators: when an operator looks for data on a pull-based input queue, that queue issues a procedure call to the child operator asking it to produce data and forces the caller to block until the child produces data. This allows operators to be combined in arbitrary arrangements of push and pull.

Figure 3 shows pseudocode for an example selection operator (Figure 3a) and pull queue (Figure 3b.) The selection operator simply checks its queue to see if there is data available; the queue may or may not actually return a tuple. If that queue is an instance of a pull queue, the $transition$ method of the operator below will be called until it produces a tuple or an error.

One important advantage of a state machine model is that it reduces the number of threads. Traditional push based schemes place each pushing operator in its own thread; that operator produces data as fast as possible and enqueues it. The problem with this approach is that operating system threads packages often allow only very coarse control over thread scheduling, and database systems may want to prioritize particular operators at a fine granularity rather than devoting nearly equal time slices to all operators. Our scheduler mechanism enables this kind of fine grain prioritization by allowing Fjord builders to specify their own scheduler which $transitions$ some modules more frequently than others. Furthermore, on some operating systems, threads are quite heavyweight: they have a high memory and context switch overhead [27]. Since all state machine operators can run in a single thread, we never pay these penalties, regardless of the operating system.

3.1.4. Sensor Sensitive Operators. Fjords can use standard database operators, but to be able to run queries over streaming data, special operators that are aware of the infinite nature of streams are required. Some relational operators, like selections and projections, work with streams without modification. Others cannot be applied to a stream of data: aggregates like average and count and sorts fall under this category. Some join implementations, such as sort-merge join, which require the entire outer relation also fail. We use a variety of special operators in place of these solutions.

First, non-blocking join operators can be used to allow incremental joins over streams of data. Such operators have been discussed in detail in adaptive query processing systems such as Xjoin [26], Tukwila [14], Eddy [3] and Ripple Joins [9]. We have implemented an in memory symmetric hash-join [28], which maintains a hashtable for each relation. When a tuple arrives, it is hashed into the appropriate hash table, and the other relation's table is probed for matches.

It is also possible to define aggregate operators, like count and average, which output results periodically; whenever a tuple arrives from the stream, the aggregate is updated, and its revised value is forwarded to the user. Similar techniques were also developed in the context of adaptive databases, for instance, the CONTROL Project's Online Aggregation algorithms [10] and the Niagara Internet Query System [24].

If traditional (i.e., blocking) aggregates, sorts, or joins must be used, a solution is to require that these operators specify a subset of the stream which they operate over. This subset is typically defined by upper and lower time bounds or by a sample count. Defining such a subset effectively converts an infinite stream into a relation which can be used in any database operator. This approach is similar to previous work done on windows in sequence database systems [22, 25].

By integrating these non-blocking operators into our system, we can take full advantage of Fjords' ability to mix push and pull semantics within a query plan. Sensor data can flow into Fjords, be filtered or joined by non-blocking operators, or be combined with local sources via windowed and traditional operators in a very flexible way.

3.2. Sensor Proxy

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 proxy serves a number of purposes. The most important of these is to shield the sensor from having to deliver data to hundreds of interested end-users. It accepts and services queries on behalf on the sensor, using the sensor's processor to simplify this task when possible.

```
public class Select extends Module {
    Predicate filter; //The selection predicate to apply
    QueueIF inputQueue;
    ...
    public TupleIF transition(StateIF state) {
        MsgIF mesg;
        TupleIF tuple = null;
        //Look for data on input queue
        mesg = inputQueue.get();
        if (mesg != null) {
            //If this is a tuple, check to see if it passes predicate
            if (mesg instanceof TupleMsg &&
                 filter.apply(((TupleMsg)mesg).getTuple()))
                     tuple=((TupleMsg)mesg).getTuple();
            else ... handle other kinds of messages ...
        }
         ... adjust state: Select is stateless, so nothing to do here ...
         return tuple; //returning null means nothing to output
    }
}
```

(a)Selection Operator

```
public class PullQueue implements QueueIF {
    //Modules this Queue connects
    Module below, above;
    StateIF bSt;
    ...
    public MsgIF get() {
        TupleIF tup=null;
        //Loop, pulling from below
        while (tup == null) {
            tup=below.transition(bSt);
            ... check for errors, idle ...
        }
        return new TupleMsg(tuple);
    }
}
```

(b)Pull Queue

Figure 3. *Code Snippet For Selection Operator and Pull Queue*

One function 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's 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 online every few seconds to see if queries have been issued.

An additional 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, in our traffic scenario, the proxy can direct the sensor to use one of several sampling algorithms depending on the amount of detail required by user queries. Or, if the proxy observes that all of the current user queries are interested only in samples with values within some range, the proxy can instruct the sensor to not transmit samples outside that range.

Also, we expect that in many cases there will be a number of users interested in data from a single sensor. As we show in Section 4.3 below, the sensor proxy can dramatically increase the throughput of a Fjord by limiting the number of copies of sensor tuples flowing through the query processor to just one per sample, and having the user queries share the same tuple data.

Sensor proxies are long running services that exist across many user queries and route tuples to different query operators based on sample rates and filter predicates specified by each query. When a new user query over a sensor stream is created, the proxy for that sensor is located and the query is installed. When the user stops the query, the proxy stops relaying tuples for that query, but continues to monitor and manage the sensor, even when no queries are being run.

3.3. Building A Fjord

Given Fjord operators and sensor proxies as the main elements of our solution, it is straightforward to generate a Fjord from a user query over a sensor. For this discussion, we will make the simple assumption that queries consist of a set of selections to be applied, a list of join predicates, and an optional aggregation and grouping expression. We are not focused on a particular query language, and believe Fjords are a useful architectural construct for any query language – other research projects, such as Predator and Tribeca, have proposed useful languages for querying streaming data, which we believe are readily adaptable to our architecture [22, 25]. We do not allow joins of two streams nor can we aggregate or sort a stream. Users are allowed to define windows on streams which can be sorted, aggregated, or joined. A single stream can be joined with a stream window or a fixed data source if it is treated as the outer relation of an index join or the probe relation of a hash-join. Users can optionally specify a sample rate for sensors, which is used to determine the rate at which tuples are delivered for the query.

Building the Fjord from such a query works as follows: for each base relation r, if r is a sensor, we locate the persistently running sensor proxy for r. We then install our query into r's proxy, asking it to deliver tuples at the user provided sample rate and to apply any filters or aggregates which the user has specified for the sensor stream. The proxy may choose to fold those filters or aggregates into existing predicates it has been asked to apply, or it may request that they be managed by separate operators. For a relation r' that does not represent a sensor, we create a new scan operator over r'. We then instantiate each selection operator, connecting it to a base relation scan or earlier selection operator as appropriate. If the base relation is a sensor, we connect the selection via a push-queue, meaning that the sensor will push results into the selection. For non-sensor relations, we use a pull queue, which will cause the selection to invoke the scan when it looks for a tuple on its input queue.

We then connect join operators to these chains of scans and selects, performing joins in the order indicated by a standard static query optimizer. If neither of the joined relations represents a sensor, we choose the join method recommended by the optimizer. If one relation is a sensor, we use it as the probe relation of a hash join, hashing into the static relation as each stream tuple is pushed into the join. The output of a join is a push queue if one relation is from a sensor, and a pull queue otherwise.

Sorts and aggregates are placed at the top of the query plan. In the case where one (or both) of the relations is from a sensor with a user specified window size, we treat it as a non-sensor relation by interposing a filter operator above the sensor proxy that passes only those tuples in the specified window.

3.4. Multiple Queries in a Single Fjord

One way in which streaming data differs from traditional data sources is that it is inseparably tied with the notion of *now*. Queries over streams begin looking at the tuples produced starting at the instant the query is posed – the history of the stream is not relevant. For this reason, it is possible to share significant amounts of computation and memory between several queries over the same set of data sources: when a tuple is allocated from a particular data source, that tuple can immediately be routed to all queries over that source – effectively, all queries are reading from the same location in the streaming data set. This means that streaming tuples need only be placed in the query processor's memory once, and that selection operators over the same source can apply multiple predicates at once. Fjords explicitly enable this sharing by instantiating streaming scan operators with multiple outputs that allocate only a single copy of every streaming tuple; new queries over the same streaming source are folded into an existing Fjord rather than being placed in a separate Fjord. A complete discussion of how this allocation and query folding works is beyond the scope of this paper, but related ideas are presented in literature on continuous queries [5]. We will, however, show how this folding can be used to improve query performance in the results section which follows.

4. Traffic Implementation and Results

We now present two performance studies to motivate the architecture given above. The first study, given in this section, covers the performance of Fjords. The second study, given in Section 5, examines the interaction of sensor power consumption and the sensor proxy and demonstrates several approaches to traffic sensor programs which can dramatically alter sensor lifetime.

4.1. Traffic Queries

We present here two sample queries which we will refer to through the rest of the section, as given by the following SQL excerpts. These queries are representative of the types of queries commuters might realistically ask of a traffic inquiry system. These queries are run over data from 32 of CalTrans' inductive loops collected by an older generation of sensors equipped with wireless radio links that relay data back to UC Berkeley. These sensors consist of sixteen sets of two sensors (referred to as "upstream" and "downstream"), with one pair on either side of the freeway on eight distinct segments of I-80 near UC Berkeley. The sensors are 386-class devices with Ricochet 19.2 kilobit modem links to the Internet. They collect data at 60Hz and relay it back to a Berkeley server, where it is aggregated into counts of cars and average speeds or distributed to various database sources (such as ours) via JDBC updates.

Query 1

```
SELECT AVG(s.speed, w)
FROM sensorReadings AS s
WHERE s.segment ∈ knownSegments
```

Query 1 selects the average speed over segments of the road, using an average window interval of w seconds. These queries can be evaluated using just the streaming data currently arriving into the system.They require no additional data sources or access to historical information.

Query 2

```
SELECT AVG(s.speed, w), i.description
FROM incidents as i,
   sensorReadings as s
WHERE i.time >= now - timeWindow
GROUP BY i.description
HAVING speedThreshold >
         (SELECT AVG(s.speed,w)
          FROM sensorReadings as s
          WHERE i.segment = s.segment
          AND s.segment ∈ {knownSegments})
```

Query 2 joins sensor readings from slow road segments the user is interested in to traffic incidents which are known to have recently occurred in the Bay Area. Slow road segments are those with an average speed less than $speedThreshold$. The set of segments the user is interested in is $knownSegments$. "Recently" means since $timeWindow$ seconds before the current time. The California Highway Patrol maintains a web site of reported incidents all over California, which we can use to build the incidents relation [4]. Evaluating this query requires a join between historical and streaming data, and is considerably more complicated to evaluate than Query 1.

4.2. Traffic Fjords

In this section, we show two alternative Fjords which correspond to Query 1 above. Space limitations preclude us from including similar diagrams for Query 2; we will discuss the performance of Query 2 briefly at the end of this section. Figure 4 shows a Fjord that corresponds to Query 1. Like the query, it is quite simple: tuples are routed first from the BHL server to a sensor proxy operator, which uses a JDBC input queue to accept incoming tuples. This proxy collects streaming data from various sensors, averages the speeds over some time interval, and then routes those aggregates to the multiplex operator, which forwards tuples to both a save-to-disk operator and a filter operator. The save-to-disk operator acts as a logging mechanism: users may later wish to recall historical information over which they previously posed queries. The filter operator selects tuples

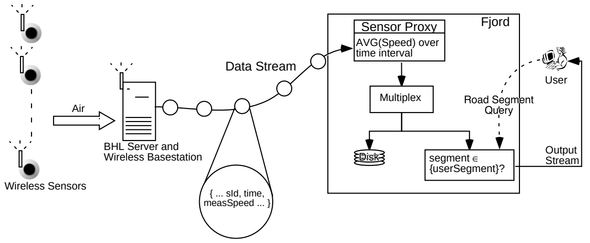


Figure 4. *Fjord Corresponding to Query 1*

based on the user query, and delivers to the user a stream of current speeds for the relevant road segment.

Notice that the data flow in this query is completely push driven: as data arrives from the sensors, it flows through the system. Also note that user queries are continuous: data is delivered periodically until the user aborts the query. The fact that data is pushed from sensors eliminates problems that such a system could experience as a result of delayed or missing sensor data: since the sensor is driving the flow of tuples, no data will be output for offline sensors, but data from other sensors flowing through the same Fjord will not be blocked while the query processor waits for those offline sources.

Figure 4 works well for a single query, but what about the case where multiple users pose queries of the same type as Query 1, but with different filter predicates for the segments of interest? The naive approach would be to generate multiple Fjords, one per query, each of which aggregates and filters the data independently. This is clearly a bad idea, as the allocation and aggregation of tuples performed in the query is identical in each case. The ability to dynamically combine such queries is a key aspect of the Fjords architecture. A Fjord which corresponds to the combination of three queries similar to Query 1 is illustrated in Figure 5.

4.3. Fjords for Performance

We now present two experiments related to Fjords: In the first, we demonstrate the performance advantage of combining related queries into a single Fjord. In the second, we demonstrate that the Fjords architecture allows us to scale to a large number of simultaneous queries.

We implemented the described Fjords architecture, using join and selection operators which had already been built as a part of the Telegraph dataflow project. All queries were run on a single, unloaded Pentium III 933Mhz with a single EIDE drive running Linux 2.2.18 using Sun's Java Hotspot 1.3 JDK. To drive the experiments we use traces obtained from the BHL traffic sensors. These traces are stored in a file, which is read once into a buffer at the beginning of each experiment so that tests with multiple queries over a single sensor are not penalized for multiple simultaneous disk IOs on a single machine.

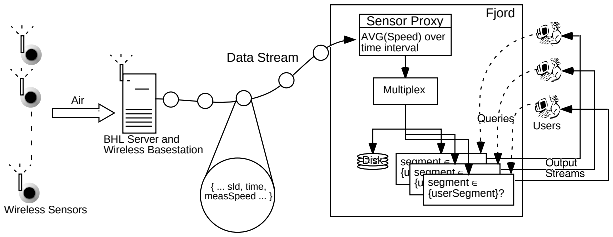


Figure 5. *Fjord for Multiple Instances of Query 1*

For the particular queries discussed here, sample window size is not important, so we generate traces with 30-second windows. The trace file contained 10767 30-byte records corresponding to traffic flow at a single sensor during June '00.

In the first experiment, we compare two approaches to running multiple queries over a single streaming data source. For both approaches, some set of n user queries, Q, is submitted. Each query consists of a predicate to be evaluated against the data streaming from a single sensor. The first approach, called the *multi-Fjord* approach allocates a separate Fjord (such as the one shown in Figure 4) for each query $q \in Q$. In the second approach, called the *single Fjord* approach, just one Fjord is created for all of the queries. This Fjord contains a filter operator for each of the n queries (as shown in Figure 5.) Thus, in the first case, n threads are created, each running a Fjord with a single filter operator, while in the second case, only a single thread is running, but the Fjord has n filter operators. In order to isolate the cost of evaluating filters, we also present results for both of these architectures when used with no filter operator (e.g. the sensor proxy outputs directly to the user queue), and with a null operator that simply forwards tuples from the sensor proxy to the end user.

Figure 6 shows the total time per query for these two approaches as the number of concurrent queries is increased from 1 to 10. All experiments were run with 150 MB of RAM allocated to the JVM and with a 4MB tuple pool allocated to each Fjord. Notice that the single Fjord version handily outperforms the multi-Fjord version in all cases, but that the cost of the selection and null filter is the same in both cases (300 and 600 milliseconds per query, respectively). This behavior is due to several reasons: First, there is substantial cost for laying out additional tuples in the buffer pools of each of the Fjords in the multi-Fjord case. In the single Fjord case, each tuple is read once from disk, placed in the buffer pool, and never again copied. Second, there is some overhead due to context switching between multiple Fjord threads.

Figure 6 reflects the direct benefit of sharing the sensor proxy: additional queries in the single Fjord version are less expensive than the first query, whereas they continue to be about the same amount of work as a single query in the multi-fjord version. The spike in the multi-fjords lines at two queries in 6 is due to queries building up very long

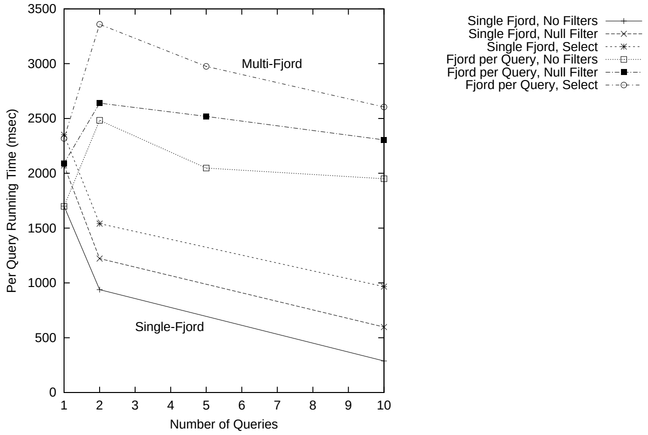


Figure 6. *Time Per Query vs. No. of Queries*

queues of output tuples, which are drained by a separate thread. Our queues become slower when there are more than a few thousand elements on them. This does not occur for more simultaneous queries because each Fjord thread runs for less time, and thus each output queue is shorter and performs better. This is the same reason the slope of the single fjord lines in Figure 6 drops off: all queries share a single output queue, which becomes very long for lots of queries.

Having shown the advantage of combining similar user queries into a single Fjord, we present a second experiment that shows this solution scaling to a large number of user queries. In these tests, we created n user queries, each of which applied a simple filter to the same sensor stream, in the style of Query 1 in Section 4.1. We instantiated a Fjord with a single sensor proxy, plus one selection operator per query. We allocated 150MB of RAM to the query engine and gave the Fjord a 4MB tuple pool. We used the same data file as in the previous section. Figure 7 shows the the aggregate number of tuples processed by the system as the number of queries is increased. The number of tuples per second per query is the limit of the rate at which sensors can deliver tuples to all users and still stay ahead of processing. Notice that total tuple throughput climbs up to about 20 queries, and then remains fairly constant, without decreasing. This leveling off happens as the processor load becomes maximized due to evaluation of the select clauses and enqueuing and dequeuing of tuples.

We also ran similar experiments from Query 2 (Section 4.1). Due to space limitations, we do not present these results in detail. The results of this experiments were similar to the Query 1 results: the sensor proxy amortizes the cost of stream buffering and tuple allocation across all the queries. With Query 2, the cost of the join is sufficiently high that the benefit of this amortization is less dramatic: 50 simultaneous queries have a per query cost which is only seven percent less than the cost of a single query. In [16], we discuss techniques for combining joins to overcome this limitation.

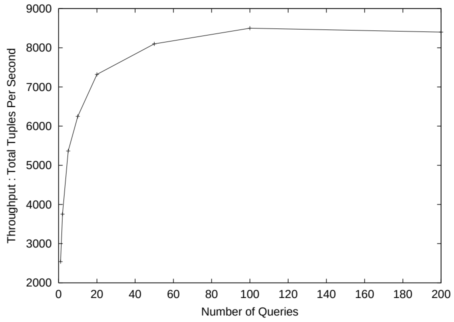


Figure 7. *Total Tuples Per Second vs. No. of Queries*

5. Controlling Power Consumption via Proxies

The experiments in the previous section demonstrated the ability of Fjords to efficiently handle many queries over streaming data. We now turn our attention to another key aspect of sensor query processing, the impact of sample rate on both sensor lifetime and the ability of Fjords to process sensor data. We focus on sensors that are similar to the wireless sensors motes described in Section 2.1 above.

5.1. Motes for Traffic

In this section, we assume that motes are placed or can self organize into pairs of sensors less than a car's length apart and in the same lane. We call these sensors S_u, the upstream sensor, and S_d, the downstream sensor. We assume that through radio telemetry with fixed basestations and each other, of the sort described in [20], it is possible for the sensors to determine that their separation along the axis of the road is d feet. These sensors are equipped with light or infrared detectors that tell them when a car is passing overhead.

Traffic engineers are interested in deducing the speed and length of vehicles traveling down the freeway; this is done via four time readings: t_0, the time the vehicle covers S_u; t_1, the time the vehicle completely covers both S_u and S_d ; t_2, the time the vehicle ceases to cover S_u, and t_3, the time the vehicle no longer covers either sensor. These states are shown in Figure 8. Notice that the collection of these times can be done independently by the sensors, if the query processor knows how they are placed: S_u collects t_0 and t_2, while S_d collects t_1 and t_3. Given that the sensors are d feet apart, the speed of a vehicle is then $d/(t_1 - t_0) ft/sec$, since $t_1 - t_0$ is the amount of time it takes for the front of the vehicle to travel from one sensor to the other to the other. The length of the vehicle is just $speed \cdot (t_0 - t_2)$, since $t_0 - t_2$ is the time it takes for both the front and back of the car to pass the S_u.

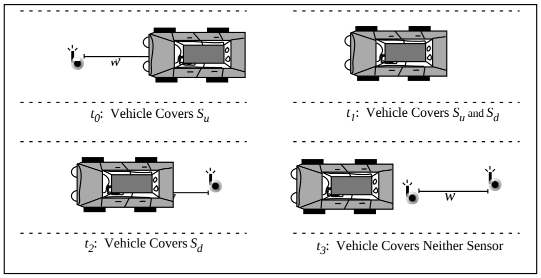

w
t_0: Vehicle Covers S_u
t_1: Vehicle Covers S_u and S_d
t_2: Vehicle Covers S_d
w
t_3: Vehicle Covers Neither Sensor

Figure 8. *Vehicle moving across sensors S_u and S_l at times t_0,t_1,t_2, and t_3*

These values are important because they indicate how accurate the timing measurements from the sensors need to be; we omit specific calculations due to space limitations, but a sample rate of about 180Hz is needed to compute vehicle lengths to an accuracy of one foot when vehicles are traveling 60mph.

Sensors relay readings to base stations placed regularly along the freeway. These base stations have wired Internet connections or high power wireless radios which can relay information back to central servers for processing. Such base stations are elevated, to overcome propagation problems that result between sensors that are on or embedded in a low reflectivity surface like the roadway. Sensors transmit samples along with time-stamps, which can be generated via techniques such as those proposed in [6].

The results described in this section were produced via simulation. Processor counts were obtained by implementing the described algorithms on an Atmel simulator, power consumption figures were drawn from the Atmel 8515 datasheet [2], and communication costs were drawn from the TinyOS results in [12], which uses the RFM TR100 916 Mhz [21] radio transceiver. Table 1 summarizes the communication and processor costs used to model power consumption in this section.

We present three sensor scenarios, as shown in Figure 9. In each scenario, the vehicle speed and length computation presented above is performed. By choosing to perform that computation in the network, rather than on the host PC, we will show a performance benefit of two orders of magnitude.

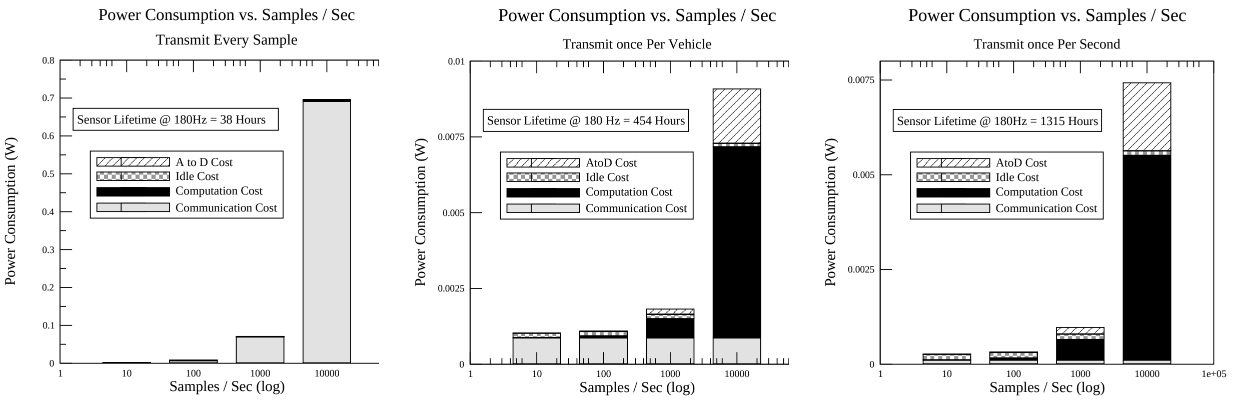
In the first scenario, sensors relay data back to the host PC at their sample rate, performing no aggregation or processing, and transmitting raw voltages. The code is extremely simple: the sensor reads from its A-to-D input, transmits the sample, then sleeps until the next sample period arrives. In this naive approach, power consumption is dominated by communication costs. Figure 9(a) illustrates this; the idle cost, computation cost, and A-to-D costs are all so small as to be nearly invisible. For the baseline sample rate of 180Hz, the power draw comes to 13mW or 2.6mA/h, enough for our sensor pairs to power themselves for about a day and a half: clearly this approach does not produce low maintenance road sensors. Furthermore, this approach places a burden on the database system: as Figure 9(a) shows, at 180 samples/per second a Fjord is limited to about 50 simultaneous simple queries, if the entire sample stream is routed through each query. In practice, of course, not all of the queries are interested in the entire data stream, so the sensor proxy can aggregate the samples into identified vehicles or vehicle counts.

Table 1. **Sensor Parameters.** *Power Parameters for Atmel 8515 Processor and RFM TR100 Radio.*

Parameter	Value
Radio Xmit Energy Cost	$4.31^{-6}\ J/bit$
Processor Voltage	$5V$
Processor Current (Active)	$6\ mW$
Processor Current (Idle)	$30\ \mu W$
Processor Speed	$1\ Mhz$
A-to-D Current	$.6\ mW$
A-to-D Latch Time	$60\ \mu S$
Battery Capacity	$100\ mAh$

In the second scenario (shown in Figure 9(b)), we use a small amount of processor time to dramatically reduce communication costs. Instead of relaying raw voltages, the sensors observe when a car passes over them, and transmit the $\{t_0, t_2\}$ or $\{t_1, t_3\}$ tuples needed for the host computer to reconstruct the speed and length of the vehicle. The sensors still sample internally at a fast sample rate, but relay only a few samples per second – in this case, we assume no more than five vehicle pass in any particular second. In this scenario, for higher sample rates, power consumption is dominated by the processor and A-to-D converter; communication is nearly negligible. At 180Hz, the total power draw has fallen to 1.1mW, or .22mA/h, still not ideal for a long lived sensor, but enough to power our traffic sensors for a more reasonable two and a half weeks. Also, by aggregating and decreasing the rate at which samples are fed into the query processor, the sensors contribute to the processing of the query and require fewer tuples to be routed through Fjords. Although this may seem like a trivial savings in computation for a single sensor, in an environment with hundreds or thousands of traffic sensors, it is non-negligible.

In the final scenario, we further reduce the power demands by no longer transmitting a sample per car. Instead, we only relay a count of the number of vehicles that passed in the previous second, bringing communications costs down further for only a small additional number of processor instructions per sample. This is shown in Figure 9(c); the power draw at 180Hz is now only .38mW, a threefold reduction over the second scenario and nearly two orders of magnitude better than the naive approach. Notice that the length and speed of vehicles can no longer be reconstructed; only the number of vehicles passing over each sensor per second is given. Thus, this scenario is an example of a technique that a properly programmed sensor proxy



(a) (b) (c)

Figure 9. *Power consumption for different sensor implementations*

could initiate when it determines that all current user queries are interested only in vehicle counts.

To summarize, for this particular sensor application, there are several possible approaches to sampling sensors. For traffic sensors, we gave three simple sampling alternatives which varied in power consumption by nearly two orders of magnitude. Lowering the sample rate increases sensor lifetime but reduces the accuracy of the sensor's model. Aggregating multiple samples in memory increases processor and CPU burden but reduces communication cost. Thus, a sensor proxy which can actively monitor sensors, weighing user needs and current power conditions, and can appropriately program and control sensors is necessary for achieving acceptable sensor battery life and performance.

6. Related Work

Having presented our solutions for queries over stream sensor data, we now discuss related projects in the sensor and database domains.

The work most closely related to ours is the Cougar project at Cornell [18], which is also intended for query processing over sensors. Their research, however, is more focused on modeling streams as persistent, virtual relations and enabling user queries over sensors via abstract data types. Their published work to date does not focus on the power or resource limitations of sensors, because it has been geared toward larger, powered sensors. They do not discuss the push based nature of sensor streams. Our work is complementary to theirs in the sense that they are more focused on modeling sensor streams, whereas we are interested in the practical issues involved in efficiently running queries over streams.

There has been significant work in the database community focused on the language and operator design issues for querying data streams. Early work sought to design operators for streams in the context of functional programming languages like Lisp and ML [17], or for specialized regimes like network router log analysis [25]. Seshadri, et. al. [22] brought this work fully into the domain of relational databases by describing extensions to SQL for stream query processing via windowed and incremental operators.

More recent research on streams continues to extend relational databases with complex operators for combining and mining data streams. For instance, [7] showed single pass algorithms to compute complex, correlated aggregates over sets of streams.

The CONTROL project [11] discusses the possibility of user interfaces for the incrementally sorting and aggregating very large data sets which is also applicable to streams. Shanmugasundaram et. al, [24] discuss techniques for percolating partial aggregates to end users which also apply.

Existing work on continuous queries provides some interesting techniques for simultaneously processing many queries over a variety of data sources. These systems provide an important starting point for our work but are not directly applicable as they are focused on continuous queries over traditional database sources or web sites and thus don't deal with issues specific to streaming sensor data.

The NiagaraCQ project [5] is the most recent work focused on providing continuous queries over changing web sites. Users install queries, which consist of an XML-QL query as well as a duration and re-evaluation interval. Queries are evaluated periodically based on whether the sites have changed since the query was last run. The system is geared toward running very large numbers of queries over diverse data sources. The system is able to perform well by grouping similar queries, extracting the common portion of those queries, and then evaluating the common portion only once. We expect that this technique will apply to streaming sensor queries as well: there will be many queries over a particular stream which share common subexpressions.

The XFilter system [1] is another example of a continuous query system. It indexes XML queries to enable efficient routing of streaming XML documents to users; this index could form the basis of an efficient implementation of query processing over XML data streaming from sensor networks.

In the remote sensing community, there are a number of systems and architecture projects focused on building sensor networks where data-processing is performed in a distributed fashion by the sensors themselves. In these scenarios, sensors are programmed to be application aware, and operate by forwarding their readings to nearby sensors and collecting incoming readings to produce a locally consistent view of what is happening around them. An example of such a design is Directed Diffusion from researchers at USC [13].

7. Conclusions

The large scale deployment of tiny, low power networks of radio-driven sensors, raises the need for power sensitive techniques for querying the data they collect.

Our solution addresses the low level infrastructure issues in a sensor stream query processor via two techniques: First, the Fjords architecture combines proxies, non-blocking operators and conventional query plans. This combination allows streaming data to be pushed through operators that pull from traditional data sources, efficiently merging streams and local data as samples flow past and enabling sharing of work between queries. Second, sensor proxies serve as intermediaries between sensors and query plans, using sensors to facilitate query processing while being sensitive to their power, processor, and communications limitations. These solutions are an important part of the Telegraph Query Processing System, which seeks to extend traditional query processing capabilities to a variety of non-traditional data sources. Telegraph, when enhanced with our sensor stream processing techniques, enables query processing over networks of wireless, battery powered devices that can not be queried via traditional means.

References

[1] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, September 2000.

[2] Atmel Corporation. Atmel 8bit AVR microcontroller datasheet. http://www.atmel.com/atmel/acrobat/doc1041.pdf.

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

[4] California Highway Patrol. Traffic incident information page. http://cad.chp.ca.gov/.

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

[6] J. Elson and D. Estrin. Time synchronization for wireless sensor networks. In *Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Parallel and Distributed Computing Issues in Wireless and Mobile Computing*, April 2001.

[7] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *ACM SIGMOD*, Santa Barbara, CA, May 2001.

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

[9] P. Hass and J. Hellerstein. Ripple joins for online aggregation. In *ACM SIGMOD*, pages 287–298, Philadelphia, PA, June 1999.

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

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

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

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

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

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

[16] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. Submitted for Publication, 2001.

[17] D. S. Parker. Stream databases. Technical report, UCLA, 1989. Final Report for NSF Grant IRI 89-17907.

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

[19] G. Pottie and W. Kaiser. Wireless sensor networks. *Communications of the ACM*, 43(5):51 – 58, May 2000.

[20] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *MobiCOM*, August 2000.

[21] RFM Corporation. RFM TR1000 Datasheet. http://www.rfm.com/products/data/tr1000.pdf.

[22] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database systems. In *VLDB*, Mumbai, India, September 1996.

[23] M. Shah, S. Madden, M. Franklin, and J. M. Hellerstein. Java support for data intensive systems. *SIGMOD Record*, December 2001. To Appear.

[24] J. Shanmugasundaram, K. Tufte, D. DeWitt, J. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *WebDB*, May 2000.

[25] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX*, New Orleans, LA, June 1998.

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

[27] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, UC Berkeley, April 2000. http://www.cs.berkeley.edu/ mdw/papers/events.pdf.

[28] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the International Conference on Parallel and Distributed Information Systems (PDIS*, pages 68–77, December 1991.