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

Samuel Madden
madden@cs.berkeley.edu

Michael J. Franklin
franklin@cs.berkeley.edu

March 2, 2001

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 world 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 end-user query throughputs. We evaluate our architecture using traces from a network of traffic sensors deployed on Interstate 80 near Berkeley, present performance results, and show how query throughput, communication costs, and power consumption are necessarily coupled in sensor environments.

1 Introduction

Recent developments in wireless networking and embedded processor design have made feasible the vision of ubiquitous computing [28] in which computers and sensors assist in every aspect of our lives. As this vision is realized, networked sensors will appear everywhere, and will produce very large amounts of data, which needs to be combined and aggregated with other sensor data to analyze and react to the world. The ability to apply traditional data-processing techniques to 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 solution to the problem of querying sensor data operates on two levels: First, we propose a modified query plan architecture called Fjords ("Framework in Java for Operators on Remote Data Streams"), which allows users to pose long running queries and facilitates the combination of multiple related sensor queries into a single query-plan. These queries take advantage of 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.

A key component of Fjords is that data flows into them from sensors and is pushed into query operators. Operators do not actively pull data to process, rather, they operate on the samples when sensors make them available and are otherwise idle. They never wait for a particular tuple to arrive. Because of this passive-behavior, the adaptive-query processing situation of an operator being "blocked", waiting for input, does not arise.

We now present an overview of the sensor-query processing environment and discuss the sensor testbed which 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, offer directions for future work and conclude.

2 Sensor Environment

In this paper, we focus on a specific type of sensor processing environment 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 directly involve them in the query processing: they simply relay their data 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 which 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 general query environment is shown in Figure 1. 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 [12] query processing engine which is under development at UC Berkeley. 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. New Fjords may be

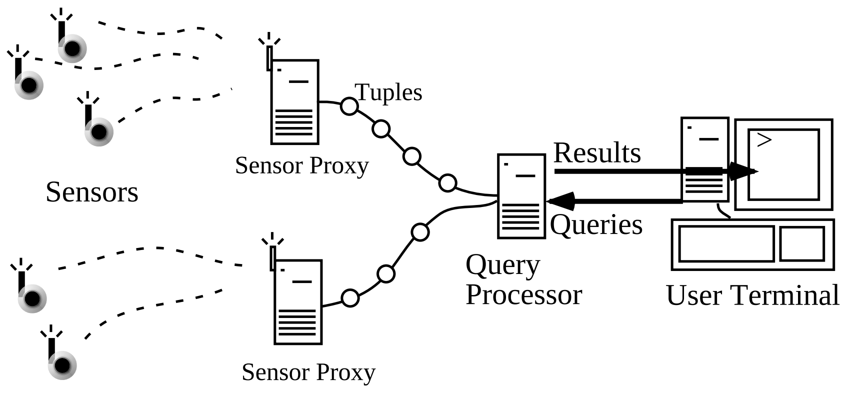


Figure 1: *Environment For Sensor Query Processing*

merged into already running Fjords with similar structures over the same sensors, or may run independently. 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 very similar to a standard relational catalog. The data which sensors provide is assumed to be divisible into a typed schema, which users can query much as they would a standard 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 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 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 interested in 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. Except for the current test project underway (see section 2.2.1), this data is not available for real-time processing; data is simply collected and stored for later, offline analysis of flow patterns.

In the near future, it is expected that real-time sensor data will be widely available, as CalTrans is working to deploy intelligent sensors which can relay information from all over the Bay Area to its own engineers and the BHL. With several thousand sensors streaming data, efficient techniques for executing queries over those streams will become crucial. The exact nature of this new generation of sensors has not yet been determined, but the Berkeley Smart-Dust project is working on a class of sensors which is very well-suited to this type of environment.

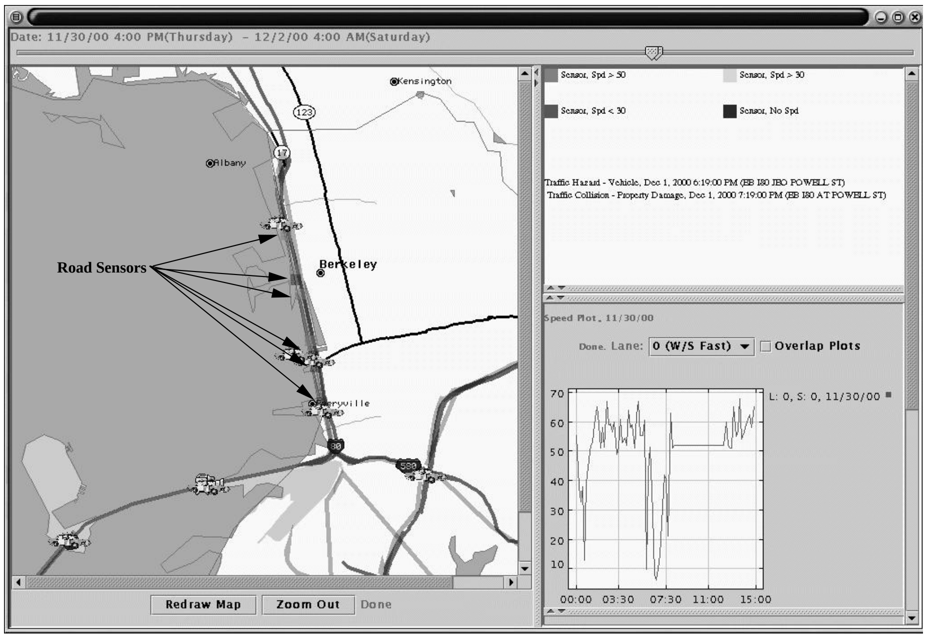
2.2 Smart-Dust Traffic Sensors

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.) The SmartDust project at UC Berkeley is developing very small, wireless radio-driven, battery powered sensors called "motes". Current prototypes are about 10^3cm, but the ultimate goal is to produce devices in the 1^3mm range – about the size of a gnat [16]. Current prototypes use an Atmel 8bit microprocessor with 8k of RAM running from 1 to 16 MHz, with a small radio capable of transmitting at 9.6Kbits with a range of a few hundred feet[13]. Other researchers, such as those at the USC Information Sciences Institute and UCLA have similar models for sensors, which they envision scattering around a battlefield to observe troop movement, or placing throughout offices in a building to locate hard-to-find colleagues[14].

In the smart-dust vision, motes could be scattered over the freeway, and could be small enough that they would 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, inferring location information from the latencies of the sensors relative to the fixed base stations[20]. Motes would use radios to relay samples to nearby wireless basestations, which would, in turn, forward them to query processing workstations.

One key component of smart-dust sensors is that they are capable of computation and remote reprogramming. This provides control over the source, quality, and density of samples. For example, two sensors in the same lane within a few feet of each other are virtually guaranteed to see the same set of cars, so asking both for the same information is unnecessary. However, there may be times when the two sensors can complement each other – for instance, the light sensor on one mote could be used to corroborate the reading of the pressure sensor from the other, or to verify that the other is functioning properly.

Similarly, the ability of motes to perform some computation and aggregation 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, reducing the amount of communication which is required and saving the central server from having to count the number of cars from the voltage readings. In an environment with tens of thousands of sensors, the benefits of such reductions can be substantial.



▨ Sensor, Spd > 50
▨ Sensor, Spd > 30
▨ Sensor, Spd < 30
■ Sensor, No Spd

Traffic Hazard - Vehicle, Dec 1, 2000 6:19:00 PM (EB I80 JBO POWELL ST)
Traffic Collision - Property Damage, Dec 1, 2000 7:19:00 PM (EB I80 AT POWELL ST)

Figure 2: *Current Traffic Sensor Visualization*

2.2.1 Current Highway Monitoring Infrastructure

The BHL is currently monitoring 32 of CalTrans' inductive loops using 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. We use this data for many of the simulations presented in this paper.

Figure 2 shows a screenshot of our 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 aren't visible in a grayscale reproduction so are labeled with arrows – represent the current state of traffic sensors for which we have data. Police-car icons represent traffic incidents the California Highway Patrol 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.

As we will show, the streaming nature of sensor data, the limited processing and energy resources of a very large number of smart-dust sensors, the tendency of wireless connections to drop individual updates or go completely offline, and the fact that updates occur at imprecise intervals and are pushed into our servers, yield an environment which is not suitable for a traditional query processing engine.

3 Requirements for Query Processing over Sensors

Given our desire to run queries over these traffic sensors, we now examine some of the issues posed by sensors that are not present in traditional data sources. We begin with a discussion of the physical properties of sensors, and then discuss the streaming nature of sensor data and the need for persistent, near-real time queries over it. After summarizing these requirements, we offer our proposed architecture, which has been developed to meet these demands.

3.1 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 current SmartDust prototypes, at full speed for only 3.5 hours. Similarly, the current SmartDust radio, which uses about 4 μJ per bit of data transmitted, can send a total of 14MB of data using such a battery.

The principal way that battery power is conserved is by powering down, or *sleeping*, parts of sensors when they are not needed. It is very important that sensors maximize the use of these low power modes whenever possible. An easy way to allow sensors to spend more time sleeping is to decrease the rate at which sensors collect and relay samples. The database system must enable this by dynamically adjusting the sample rate based on current query requirements.

In addition to power limitations, smart-dust sensors have the added feature that they include small processors which can be used to perform some processing on samples. Database systems need to able to take advantage of this processing ability, as it can dramatically reduce the power consumption of the sensors and reduce the load on the query processor itself.

3.2 Streaming Data

Another property of sensors is that they produce continuous, never ending streams of data. 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: they cannot be *blocking*. Many traditional operators, such as sorts, aggregates, and some join algorithms, like sort-merge join, 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. The conventional iterator model does not map well onto sensor streams: changing sensors to sample only when data is requested requires reprogramming them, which may be difficult or impossible. Even in cases where it is possible, implementing the iterator model 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. Also, it requires sensors to buffer samples locally until the query processor asks for them, which requires memory that many sensors do not have.

Since most smart-dust sensors have wireless connections, data streams are 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.

3.3 Processing Complex Queries

Sensors pose additional difficulties in a query processing system which is running complex or concurrent queries. In many sensor scenarios, multiple users pose similar queries over the same data streams. In the traffic scenario, 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 read-only, there is no reason that a particular sensor reading should not be shared across many queries. As our experiments in Section 5.3 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.

Finally, users querying sensors need a way to name the sensors that exist in the world. Traditional databases assume that users know the names of the sources they wish to run queries over. In a sensor environment with thousands of dynamic sensor-streams, this is not a realistic assumption. Instead, users must be able query the catalog, looking for sources which have desired properties. For instance in our traffic testbed, users will need to find sensors which are near a specific the part of the freeway.

4 Solution

Having described the difficulties involved in query processing over sensors, we now present our solution, which, as previously mentioned, consists of two core components: Fjords, an adaptive modular system for running complex queries over streaming sensor data, and the sensor-proxy, for mediating between sensors and the query-processor, while taking into account the specific limitations of sensors.

4.1 Fjords: Generalized Query Plans for Sensor 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

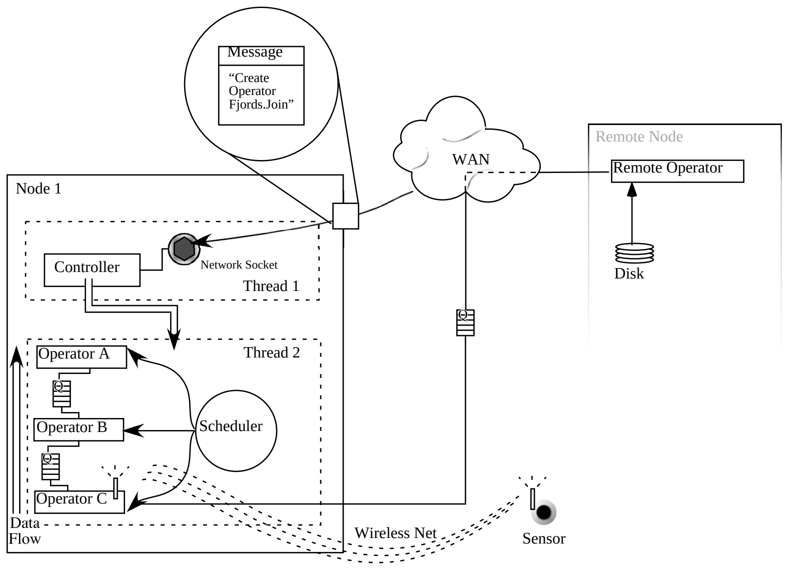


Figure 3: *The Fjord Architecture*

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.

Figure 3 shows a simple Fjord running across two machines. 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 produce outputs when they are not explicitly called by their parents in the plan.

Fjords are constructed by a *Fjord Master*, which is analogous to the query planner and optimizer in a traditional DBMS. It is responsible for choosing the operators to create, the machines to create them on, and the queues with which to connect those operators. A traditional query plan can be built by choosing operators from the set of standard operators. These operators pull from their children via function calls in the local area or transactional queues in the wide area. However, this architecture enables a number of other configurations which are attractive in the stream-processing domain: for instance, transactional queues can be replaced with more efficient, less reliable network queues, proxies can be interposed in front of sensors, and query plans can be dynamically adjusted as sensors come on and off line. Section 4.3 discusses the ways in which Fjords facilitate sensor-based queries. The following section discusses the specifics of Fjord operators and queues in more detail.

4.1.1 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. O reads tuples in any order it chooses from Q_i and outputs any number of tuples along 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.

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.

4.1.2 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 return to the same state) and possibly produce some set of output tuples o. Implicit with this model of state programming is that operators never 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.

Formulating query plan operators as state-machine states presents several interesting issues. First, 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 4 shows an example selection operator (Figure 4a) and pull queue (Figure 4b.) 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.

Second, state-machine programming requires a shift the in database-programmer's thinking about how operators work. Telling a module to $transition$ does not force it to produce a tuple, or constrain it to producing only a single tuple. The module computes for some period of time, then returns control to the scheduler which chooses another module to $transition$ to. A badly behaved operator can block for long periods of time, effectively preventing other operators from being able to run if all modules are in a single thread. The scheduler can account for this to some extent, by using a ticket scheme where operators are charged a number of tickets proportional to the amount of time they compute, as in [27]. This problem, however, is not restricted to Fjords: in general, faulty operators can impair an iterator-based query plan

```
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 predicate passes it
            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 4: *Code Snippet For Selection Operator and Pull Queue*

as well. In our experience, we have not found that this state-machine model makes operator development overly difficult.

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 operator-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 [29]. Since all state machine operators can run in a single thread, we never pay these penalties, regardless of the operating system.

4.1.3 Flexible Data Flow

In addition to the benefits of the state-machine model, an important 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 (see Figure 4b).

Push queues make it possible for sensor streams to work. 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.

4.1.4 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 . Aggregates like average and count cannot be applied to a stream of data; neither can sorts. Some joins, such as sort-merge join, which require the entire outer relation also fail. We use a variety of special operators in place of these blocking solutions.

First, non-blocking join operators can be used to allow joins over streaming data. Such operators have been discussed in detail in adaptive query processing systems such as Xjoin [26], Tukwila [15], and Eddy [3]. We have implemented an in memory symmetric hash-join [30], 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. This technique finds all joining tuples, is reasonably efficient, is tolerant to delays in either data source, and is non-blocking with respect to both relations.

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. Of course, this incremental aggregation requires operators upstream from the aggregates to understand that values are being updated, and for the user interface to redisplay updated aggregate values. Similar techniques were also developed in the context of adaptive databases, for instance, 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 regular relation which can be used in any database operator. This approach is similar to previous work done at Cornell on windows in sequence database systems [23].

Instead of using windowed operators, we can rely on a user interface to keep tuples in a sorted list and update aggregates as tuples arrive. Such an interface needs other functionality: as tuples stream endlessly in, it will eventually need to discard some of them, and thus some eviction policy is needed. Furthermore, since queries never end, the user needs to be given a way to stop a continuous query midstream. The Control project discusses a variety approaches to such interfaces[11].

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.

4.2 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.

In order to keep the sensor's radio-receiver powered down as much as possible, the proxy only sends control messages to the sensor during limited intervals. As a sensor relays samples to the proxy, it occasionally piggybacks a few bytes indicating that it will listen to the radio for some short period of time following the current sample. The exact duration of this window and interval between windows depends on the sensors and sample rates in question; we expect that for our traffic sensor environment the window size will be a few tens-of-milliseconds per second. Though this is a significant power cost, it is far less than the cost of keeping the receiver powered up at all times.

One function of control messages 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 might direct the sensor to use one of the three sampling algorithms presented in Section 6 below. Or, 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.

Sensor-proxies are long-running services that exist across many user queries and route tuples to different query operators based on sample rates and filtration 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.

We expect that in many case there will be a number of users interested in data from a single sensor. As we show in Section 5.3 below, the sensor proxy can dramatically increase the throughput of the 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.

4.3 Putting It All Together

Given Fjords and Sensor proxies as the main elements of our solution, it is straightforward to generate a Fjord from a user query over a sensor. As stated earlier there are some forms of queries we simply cannot support over streams due to their infinite nature. In particular, we cannot join two streams [1] 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

[1] Two streams can be joined by "zippering" them together as they flow past, but this works only when streams are synchronized and eqi-joined on sample time.

outer relation. Users can optionally specify a sample-rate for sensors, which is used to determine the rate at which tuples are delivered for the query.

Users may not know exactly which sensors they wish to query. To solve this, we are in the process of implementing a distributed, XML-based, queryable catalog which can be rapidly updated as sensors go on and offline. We associate semantic tags with each sensor that allow a user to determine relationships between sensors and find appropriate alternative sensors to query when a desired sensor is down. Existing service-catalog technologies, such as LDAP and the Service Discovery Service [9] provide similar technology, although capabilities to discover related services are not explicitly built-in. Using this catalog, we presume that users will formulate SQL-style queries over the sensors they wish to query. We give examples of such queries in Section 5.1 below.

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 and install our query into it, asking it to deliver tuples at the user-provided sample rate and also to apply any filters or aggregates which the user has specified for the sensor-stream. The sensor 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 all relations r that do not represent sensors, 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, such as the one presented in [22]. 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 outer relation of a hash or nested loops join, probing into the inner 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 of the relations is from a sensor, but the user has specified a window size, we treat this as a non-sensor relation by interposing a filtration operator above the sensor proxy which passes only those tuples in the specified window.

5 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 6, 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.

In this section, we show how the Fjord and sensor-proxy architectures can be applied to the traffic-sensor environment. We begin by giving examples of queries over those sensors, then show alternatives for translating one of those queries into a Fjord, and then finally present two Fjord performance experiments.

5.1 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. They are not meant to test the full functionality of our query evaluation engine. We are not, as of yet, committed to SQL as the language of choice for queries over sensor data but it is the language our tools currently support.

Query 1

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

This query selects the average speed over segments of the road, using an average window interval of w. 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 - $timeWindow$
  AND i.segment = s.segment
  AND AVG(s.speed,$w$) < $speedThreshold$
  AND s.segment $\in \{knownSegments\}$
GROUP BY i.description
```

This query 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. [5]. Evaluating this query requires a join between historical and streaming data, and is considerably more complicated to evaluate than Query 1.

5.2 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 in Section 5.3.2.

Figure 5 shows one Fjord which 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 for various stations, averages the speeds over some time interval, and then routes those aggregates to the multiplex operator, which forwards tuples to both a

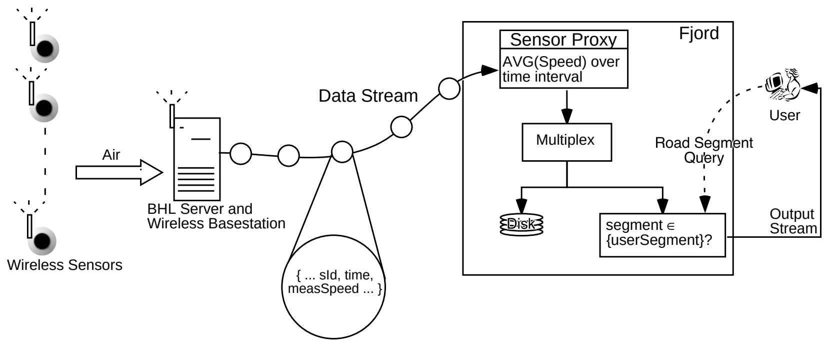


Figure 5: *Fjord Corresponding to Query 1*

save-to-disk operator and a filter operator for the ∈ clause in query 1. 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 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. User queries are used to parameterize Fjord operators, but the queries themselves do not cause any data to be fetched or delivered. 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 5 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, and is similar to work done as a part of Wisconsin's NiagaraCQ project[6]. A Fjord with such a combined sensor-proxy is illustrated in Figure 6.

5.3 Fjords for Performance

Having described the Fjords for our example queries, 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. This data set consists of records of the form $\{time, lane, station, avg(speed), flow\}$, with the following values:

$time$:	The start time of this sample window
$lane$:	The lane for this sample
$station$:	The station number for this sample
$avg(speed)$:	The average speed of vehicles observed in this sample
$flow$:	The number of vehicles observed in this sample

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.

5.3.1 Combining Fjords to Increase Throughput

For 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 5) 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 6.) 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. We implemented a very simple round-robin scheduler which schedules each operator in succession, one after the other, which is used in both cases.

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

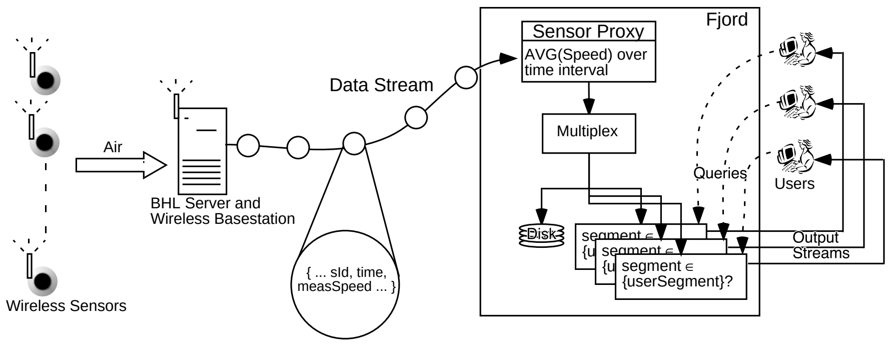
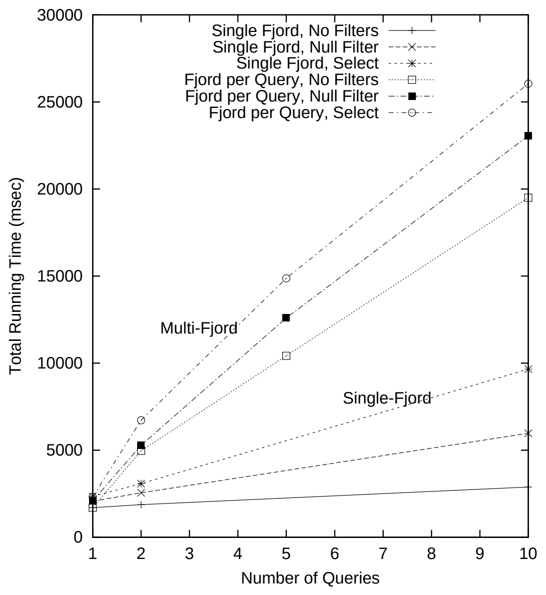
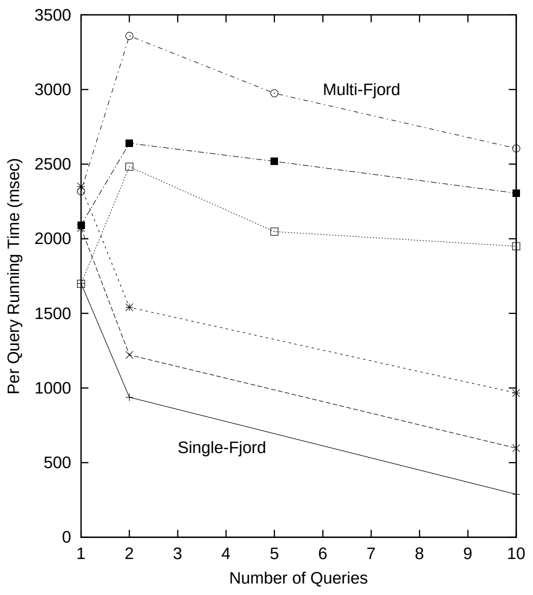


Figure 6: *Fjord Corresponding to Query 1 For Multiple User Queries*



(a)Total Running Time vs. No. of Queries

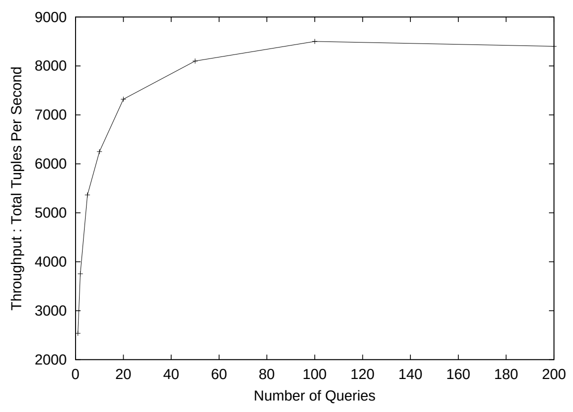


(b) Time Per Query vs. No. of Queries

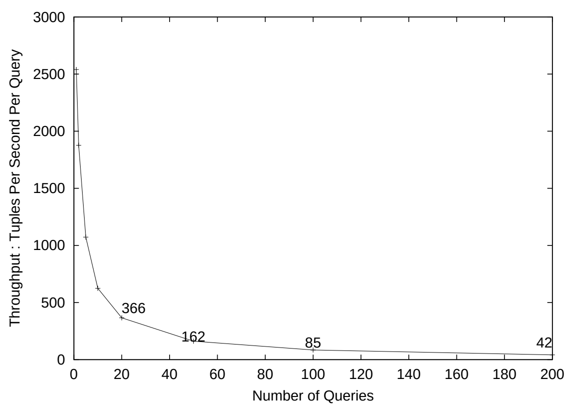
Figure 7: *Effect of Combining Multiple Queries Into A Single Fjord*

null-operator that simply forwards tuples from the sensor proxy to the end user.

Figure 7 shows the performance results of the two experiments, showing total query time (Figure 7(a)) and time per query (Figure 7(b)). 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 7(b) 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 7(b) is due to queries building up very long 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 7(b) drops off: all queries share a single output queue, which becomes very long for lots of queries.



(a) Total Tuples Per Second vs. No. of Queries



(b) Tuples Per Second Per Query vs. No. of Queries

Figure 8: *Tuple Throughput vs. Number of Queries*

5.3.2 Scaling to a Large Number of Queries

In the previous section, we showed the advantage of combining similar user-queries into a single Fjord. Now, in the second experiment, we show that this solution makes it possible to handle a very 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 5.1. 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 8(b) shows the number of tuples which the system processes per second per query versus n, while Figure 8(a) shows the the aggregate number of tuples processed versus n. 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. 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 5.1). 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.

6 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. In this section, we focus on sensors that are similar to the Smart-Dust motes described in Section 2.2 above. We choose to use this design rather than the fixed traffic sensors which are currently deployed on the freeway because we believe this is a more realistic model of the way sensors in the near future will behave, and because SmartDust sensors can be remotely reprogrammed, which is necessary for implementing the techniques we describe below.

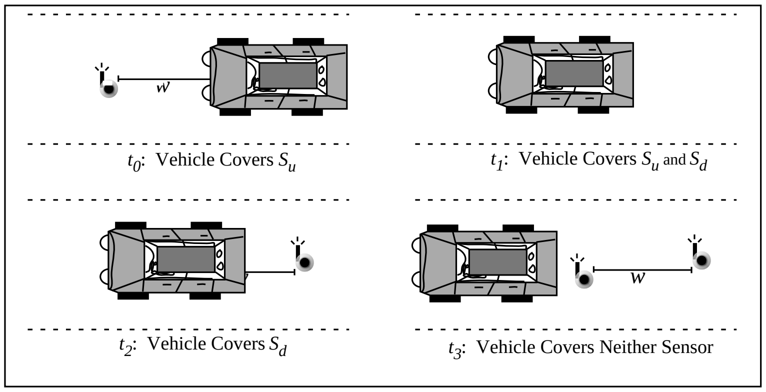
6.1 SmartDust for Traffic

In this section, we assume that smart-dust 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 9. 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[2].

These values are important because they indicate how accurate the timing measurements from the sensors need to be; imagine for instance that we wanted an estimate of the length of a car to an accuracy of one foot. To do this, readings t_0 and t_2 must be made to an accuracy of 6 inches, so that the maximum error of $t_0 - t_2$ is no greater than 12 inches. Thus, the car cannot travel more than 6 inches between sample periods. At 60mph, simple physics tells us a car will travel 6 inches in .0056 seconds, requiring a sample rate of about 180Hz. This is a relatively high sample rate, and, as we will show in the next section, in a naive implementation where sensors transmit every sample back to the host computer for speed and length detection, is high enough to severely limit the life of sensors and constrain the performance of the Fjord. Notice that this calculation assumes the speed of the vehicle is known; if we are estimating the length of the vehicle from our speed estimation, an even faster sample rate is needed.

[2]This analysis was derived based on computations for inductive loop sensors in [8].



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 9: *Vehicle moving across sensors S_u and S_l at times t_0,t_1,t_2, and t_3*

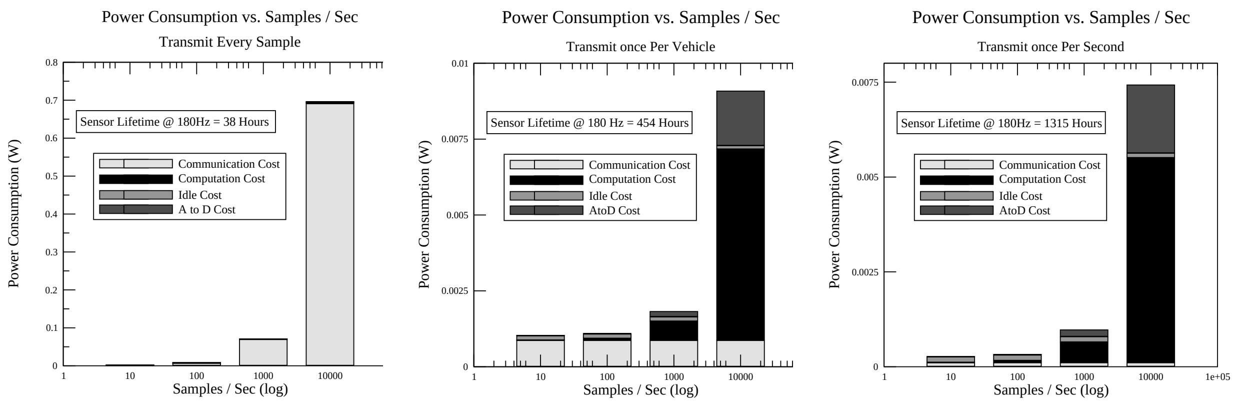
6.2 Sensor Power-Consumption Simulation

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 [13], 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. In the first, 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, uses the TinyOS communications protocol to transmit the sample, then sleeps until the next sample period arrives. In this naive approach, power consumption is dominated by communication costs. Figure 10(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 smart-dust 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 8(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,

Table 1: **Sensor Parameter Values.** *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	$100\ mAh$



(a) (b) (c)

Figure 10: *Power consumption for different sensor implementations*

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.

In the second scenario (shown in Figure 8(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. This corresponds to a vehicle every 18 feet at 60 MPH, which is a very crowded (and dangerous!) highway. Power consumption versus sample rate on each sensor is shown in Figure 10(b). In this example, 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 10(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. We present this scenario as an example of a technique that a properly programmed sensor-proxy 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. Thus, the choice of sensor programs reflect essential tradeoffs when dealing with low-power computing, and have a dramatic effect on battery life. 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. Therefore, a sensor-proxy which can actively monitor sensors, weighing user needs and current power conditions, and appropriately program and control sensors is necessary for getting acceptable sensor battery life and performance.

7 Related Work

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

7.1 Related Database Projects

The work most closely related to ours is the Cougar project at Cornell [18]. Cougar 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 their 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 recent work on databases to track moving objects, such as [31]. In such systems, object position updates are similar to sensor tuples in that they arrive unpredictably and their arrival rate is directly tied to number of queries which the database system can handle. Thus, such mobile databases could be an application for the techniques we present in this paper.

7.2 Distributed 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. As information propagates around the network, each sensor is better able to form a model of what is going on. An example of such a project is the Directed Diffusion model from the USC Information Sciences Institute; Users pose queries to a particular sensor about people or vehicles moving in some region, and sensors near that region propagate their local readings to the queried which it aggregates to answer the query [14]. It is exciting to think that such a decentralized model of query processing is possible; however, we believe that, at least for the current generation of sensors, power, storage, and processor limitations dictate that fixed, powered servers are needed to perform large scale sensor data-processing on the sensors themselves.

7.3 Data Mining on Sensors

Current scientific sensor research is similar to any data mining and warehousing application: data flows into local servers from each sensor, which create on-disk sensor logs. At some later point, this data is shipped to a central repository, which scientists then connect to and execute large aggregate and subset queries to facilitate their data exploration. The DataCutter [4] system exemplifies current research in this area: it focuses on providing an efficient platform in which scientists can collect data from multiple repositories and then efficiently aggregate and subset that data across multiple dimensions. This model is very different from the interactive sensor processing environment we envision.

7.4 Continuous Queries

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.

In the OpenCQ system [17], continuous queries are very similar to SQL triggers, except that they are more general with respect to the tables and columns they can be defined over and are geared toward re-evaluation of expressions as data sources change rather than dependency maintenance. In OpenCQ, continuous queries are four-element tuples consisting of a SQL-style query, a trigger-condition, a start-condition, and an end-condition. Some effort is taken to combine related predicates to eliminate redundant evaluations, but the NiagaraCQ system, discussed next, presents a more robust solution to this problem.

The NiagaraCQ project [6] is also focused on providing continuous queries over changing web sites. Users install queries, which consist of an XML-QL [10] query as well as a duration and re-evaluation interval. Queries are evaluated periodically based on whether the sites they query 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 similar to the NiagaraCQ system in that it queries a database of XML documents for the subset which match some filter-profile expressed in the XPath [7] language. It is complementary to the Niagara system in that it optimizes queries by indexing the set of profiles based on the filter conditions which appear within those profiles rather than by combining profiles which share common subexpressions. 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.

7.5 Stream Processing

The dQUOB system [19] is designed for very-high throughput filtration of data streams: it selects tuples which meet some set of query conditions and drops the remainder of the tuples. This is achieved primarily through two techniques: First, queries are compiled in *quoblets* – essentially, INGRES [25] style precompiled queries. Second, ordering of query predicates is done intelligently: either by running more selective predicates first, less costly predicates first, or some combination thereof. This system provides some interesting techniques for dealing with streams of data, but does not handle architecture or power issues in any way.

8 Future Work and Conclusions

As wireless communications technology and embedded processing technologies converge to allow the large scale deployment of tiny, low-power, radio-driven sensors, efficient techniques for querying the data they collect will be crucial. These techniques must not only be efficient, but also sensitive to the power limitations of the sensors.

Our sensor-stream processing solution satisfies these requirements 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. Second, sensor-proxies serve as mediators between sensors and query plans, using sensors to facilitate query processing while being sensitive to their power, processor, and communications limitations.

There are a number of interesting areas for future work. In particular, it would be useful to integrate tools for combining common sub-parts of user queries. We believe that many users will formulate similar sensor queries, and tools for rapidly evaluating similar classes of queries are needed. The XFilter[1] and NiagaraCQ[6] projects both offer techniques for extracting and evaluating common parts of user queries.

Secondly, Fjords are currently non-adaptive; that is, they do not modify the order of joins in the face of sensor delays or intermittent failures. We plan to explore the use of adaptive query operators such as Eddy[3] and XJoin[26] in Fjords. We believe that these operators can serve an important role in integrating non-streaming and streaming data by buffering streaming sources and masking latencies in traditional sources.

The solutions we have presented 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. Our sensor-stream processing techniques integrate sensors into Telegraph in a seamless and efficient manner.

References

[1] 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.

[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 *Proceedings of the ACM SIGMOD*, pages 261–272, Dallas, TX, May 2000.

[4] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the 2000 Mass Storage Systems Conference*. IEEE Computer Society Press, March 2000.

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

[6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 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] B. Coifman. Vehicle reidentification and travel time measurement using the existing loop detector infrastructure. In *Transportation Research Board*, 1998.

[9] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proceedings of Fifth ACM Conf. on Mobile Computing and Networking (MOBICOM)*, Seattle, WA, 1999. ACM.

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

[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. 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.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, and D. C. K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[14] C. Intanagonwiwat, R. Govindan, , and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *In Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCOM 2000)*, Boston, MA, August 2000.

[15] 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.

[16] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile networking for smart dust. In *Proceedings of Fifth ACM Conf. on Mobile Computing and Networking (MOBICOM))*, Seattle, WA, August 1999.

[17] 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.

[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] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *Proceedings of High Performance Distributed Computing*, 2000.

[20] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *Proc. of the Sixth Annual ACM International Conference on Mobile Computing and Networking (MOBICOM)*, August 2000.

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

[22] 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.

[23] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database systems. In *International Conference on Very Large Data Bases*, Mumbai, India, September 1996.

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

[25] M. Stonebraker, E. Wong, and P. Kreps. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

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

[27] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 1–12, Monterey CA (USA), 1994.

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

[29] 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.

[30] 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.

[31] O. Wolfson, A. P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. DOMINO: Databases fOr MovINg Objects tracking. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 547–549, Philadelphia, PA, June 1999. ACM Press.