Thinking Big About Tiny Databases

Michael J. Franklin
UC Berkeley
franklin@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Abstract

Work on early tiny database systems, like TinyDB [17] and Cougar [23] has shown that a declarative approach can provide a powerful and easy to use interface for collecting data from static sensor networks. These early systems, however, have significant limitations; in particular, they are useful only for low-rate data collection applications on static sensor networks and they don't integrate well with existing database and IT tools. In this paper, we discuss recent research that has addressed these limitations, showing that similar "tiny database thinking" can provide solutions to much bigger problems, including network protocol specification and mobile and high data rate signal-oriented data processing.

1 Introduction

Sensornets – collections of inexpensive, battery powered and wirelessly networked computers with sensing hardware for measuring light, temperature, vibration, sound, and a variety of other parameters – promise to measure the world at remarkably fine granularity. Touted applications include precision agriculture (monitoring the growth of individual plants), preventative maintenance of individual pumps or pipes in industrial settings, ecological monitoring of habitats, soil and water quality, and so on.

As with traditional enterprise database management systems, sensor network databases, like TinyDB [17] and Cougar [23] have proven to be a powerful tool for deploying such applications. In particular, these systems provide high level languages that allow users to specify what data they would like to receive without worrying about low-level details regarding how that data is collected or processed. In particular, details such as network and power management, time synchronization, and where data processing should be performed is completely hidden from the end user, greatly simplifying deployment.

Early "tiny databases" were developed several years ago. Since then, the idea of high level, declarative languages for sensor network programming has been refined in several ways. In this paper, we review a number of such refinements we have been working on in our research groups, including:

Declarative network programming. Early sensornet database systems provided declarative query languages for traditional database-like tasks of specifying data of interest to the user. We have since shown that recursive query languages can be used much more broadly throughout sensor network software infrastructure, resulting in code that is radically simpler to write and maintain than equivalent code in a traditional programming language [5]. We have demonstrated that a wide variety of ad-hoc sensor network protocols can be specified declaratively in

Copyright 2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

a very compact way, compiling down to efficient code that runs in a small footprint. The declarative approach relieves the programmer from many complexities that arise at both the networking and data processing layers, while raising a number of opportunities for innovation in both areas.

Sophisticated signal-oriented programming languages. The programming interface provided by TinyDB and Cougar was very simple, and allowed only basic relational operations of the sort that are common in business processing applications. This is insufficient for many sensor network applications which require more sophisticated signal-oriented processing. For example, in many industrial monitoring applications where vibrational analysis is used to detect malfunctions or failures, frequency transforms are used to measure the energy in certain frequency bands that characteristically show increased energy during failures. A major thrust of our recent research involves extending the declarative programming model to allow such sophisticated processing to be expressed in the same language as basic data processing operations.

Support for large-scale sensing applications. Early systems ignored issues related to how sensor networks integrate into a larger information processing ecosystem. One focus of our recent work has been on the integration of sensor data with other data sources and the processing of "roll up"-style aggregation queries over data from diverse collections of sensors.

Support for mobility and intermittent connectivity. Mobile sensor networks offer the potential to monitor much larger geographic areas than fixed networks. For example, by mounting accelerometers that can detect potholes and other road surface anomalies on a small number of cars, it is possible to monitor hundreds or thousands of miles of roads. Such mobile networks require a different software infrastructure from that provided by systems like TinyDB and Cougar, as mobile nodes often lack always-on network connections, requiring special support for intermittent connectivity.

Before describing our research in these areas in detail, we first summarize the basic declarative programming model developed in TinyDB and other tiny database systems.

2 Early Tiny Databases

The typical usage model for early sensor network databases is as follows: a collection of static sensor nodes is placed in some remote location, each of which is pre-programmed with the database software. These nodes report data wirelessly (often over multiple radio hops) to a nearby "basestation" – typically a laptop-class device with an Internet connection, which then relays data to a server where data is stored, visualized, and browsed.

Users interact with the system by issuing queries at the basestation, which in turn broadcasts queries out into the network. Queries are typically disseminated via flooding, or perhaps using some more clever gossip based dissemination scheme (e.g., Trickle [14]). As nodes receive the query, they begin processing it. The basic programming model is data-parallel: each node runs the same query over data that it locally produces or receives from its neighbors For example, a simple query that asks nodes to report when their temperature readings go above some threshold looks as follows:

```
SELECT nodeid, temp
 FROM sensors
 WHERE temp > thresh
 SAMPLE PERIOD 1s
```

Here, `sensors` is a "virtual table" containing one field per type of sensor available. By "virtual", we mean that it is never actually materialized in the memory of the sensor – instead, new rows are created in by sampling the sensors at the rate specified in the `SAMPLE PERIOD` clause, and those rows are transmitted off of the device before the next row is produced. Each node produces rows corresponding to its own local `nodeid`. Note that the evaluation of the `temp` $> thresh$ predicate can clearly be done on each individual node in this case, but that

for more complicated predicates – e.g., that depend on the values of other nodes – in-network computation may not be as trivial. Models for such in-network processing are the focus of much work in the sensor network data processing community – see, for example, work on support for in-network aggregation of data from multiple nodes [16] and in-network join processing [2, 1].

When a node has some data to transmit, it relays it to the basestation using a so-called *tree-based routing protocol*. The idea with these protocols is that nodes arrange themselves into a tree rooted at the basestation. This tree is formed by having the basestation periodically broadcast a beacon message. Nodes that hear this beacon re-broadcast it, indicating that they are one hop from the basestation; nodes that hear those messages in turn re-broadcast them, indicating that they are two hops from the basestation, and so on. This process of (re)broadcasting beacons occurs continuously, such that (as long as the network is connected) all nodes will eventually hear a beacon message. When a node hears a beacon message, it chooses a node from which it heard the message to be its *parent*, sending messages through that parent when it needs to transmit data to the basestation[1].

This basic programming and data collection model has proven to be useful in a number of data collection applications, but as discussed in the introduction, is also somewhat limited. It only works with static networks, only allows applications that rely on a fairly limited subset of SQL to be expressed, and does not address how these sensor network applications integrate into a larger information processing ecosystem. We discuss how we have addressed these limitations in our recent work in the remainder of this paper.

3 Declarative Sensor Networks

Sensor networks are notoriously difficult to program. One reason for the success of sensor databases is the attraction of a familiar declarative language like TinySQL that hides the details of the network and its devices. In many cases this can radically simplify programs, replacing hundreds of lines of distributed, embedded C code with a few lines of SQL. But the ambition of this approach – to hide the sensornet entirely and focus on the data – also limits the utility of sensor databases as an aid to programmers. First, sensor databases address simple data acquisition and analysis, and this is only one piece of the sensornet programming puzzle. Second, like traditional database systems, the first generation of sensor databases were monolithic applications that could not be easily adapted for reuse in unanticipated settings. Unfortunately, experimental systems that cannot be reused in unintended ways tend not to be reused much at all.

In considering a second generation of database-style sensornet technology, it is worth revisiting the core programming challenges in this context. By definition, wireless sensor network programmers need to focus on three main issues: (a) power management (wireless), (b) data management (sensor), and (c) network design (networks). Our hypothesis in follow-on work is that a declarative approach can radically simplify (a) and (c), in addition to (b). Moreover, by handling all three in a uniform declarative language, opportunities emerge for cross-layer optimizations, and cross-disciplinary research impact [21].

To date, we have focused on networking issues, in the context of the *DSN* Declarative Sensor Network system [5]. This follows on the heels of our *P2* system for declarative overlay networks on the Internet [15]. Both systems take declarative network specifications in Datalog-like langauges, and compile them down to executable distributed dataflow programs. Our initial results in this direction have been quite positive. In typical examples in both the Internet and wireless domains, our declarative protocol implementations are orders of magnitude shorter to express than the traditional imperative programs, very close to the protocol pseudocode in syntax, and competitive in performance and robustness. On the sensornet front, we have implemented several very different classes of traditional sensor network protocols, services and applications entirely declaratively; these include radio link estimation, tree and geographic routing, data collection, and version coherency [5]. All

[1] In general, parent selection is quite complicated, as a node may hear beacons from several candidate parents. Early papers by Woo and Culler [22] and DeCouto et al [6] provide details.

```
% Initial facts for a tree rooted at "root"
dest(@AnyNode, root).
shortestCost(@AnyNode, root, infinity)

% 1 hop neighbors to root (base case)
R1 path(@Source,Dest,Dest,Cost) :? dest(@Source,Dest?,
        link(@Source,Dest,Cost).

% N hop neighbors to root (recursive case)
R2 path(@Source,Dest,Neighbor,Cost) :? dest(@Source,Dest?,
        link(@Source,Neighbor,Cost1),
        nextHop(@Neighbor,Dest,NeighborsParent,Cost2),
        Cost=Cost1+Cost2, Source!=NeighborsParent.

% Consider only path with minimum cost
R3 shortestCost(@Source,Dest,<MIN,Cost>) :?
        path(@Source,Dest,Neighbor,Cost),
        shortestCost(@Source,Dest,Cost2?, Cost<Cost2.

% Select next hop parent in tree
R4 nextHop(@Source,Dest,Parent,Cost) :?
        shortestCost(@Source,Dest,Cost),
        path(@Source,Dest,Parent,Cost?.
```

Figure 1: Tree routing in SNLog

.

have consisted of a small number of rules, which compile down to code that runs on the resource-constrained Berkeley Mote platform running TinyOS.

To illustrate declarative networking, we review an example from [5] that specifies shortest-path routing trees (as in Section 2) from a set of nodes to a set of one or more tree roots. In Figure 3 we present this protocol in DSN's SNLog [15]. Ignoring the "@" and "∼" symbols in Figure 3 for a moment, this looks much like a standard recursive Datalog program involving transitive closure. The two "fact" statements at the beginning of the program instantiate a tuple in each of the `dest` and `shortestCost` relations at each node in the network, stating that the destination is a node called "root", and (in the absence of other information) the distance between each node and the root is inifinity. The next two rules `R1` and `R2` find all possible paths from sources to tree roots. The last two rules prune this set: `R3` finds the min-cost path grouped by distinct source/destination pairs, and `R4` sets the `parent` of each source in each tree to be the `NextHop` in the shortest path. Finally, the query specifies that all such `parents` should be returned as output.

There are a few obvious distinctions from Datalog in Figure 3. First, each relation has one field prepended with the "@" symbol; this field is called the *location specifier* of the relation. The location specifier specifies data distribution: each tuple is to be stored at the address in its location specifier field. For example, the `path` relation is partitioned by the first (source) field; each partition corresponds to the networking notion of a local routing table. Second, the tilde ("∼") is a hint to the execution engine that the arrival of new tuples from the associated body predicate can be postponed, and need not trigger immediate reevaluation of the rule. More subtleties that distinguish SNLog from Datalog are discussed in [5].

Note that the `link` relation that captures radio link quality estimation has not been specified in the figure. As discussed in the initial DSN paper [5], it can be implemented as a "built-in" functional relation that calls a low-level radio driver in the OS, or it can be implemented in a fully declarative fashion as well using radio broadcast primitives. Loo et al. [15] show how location specifiers and link relations enable a compiler to generate a distributed protocol that is guaranteed to be executable over the underlying network topology captured by the link relation.

While this example may seem relatively simplistic, many of the other features we have written are almost as short, but can express networking issues from the lowest level of managing radio link quality, to very high-level

application semantics such as object replica dissemination (the Trickle protocol mentioned above [14]). In all the examples, our code is orders of magnitude shorter to express than the native TinyOS implementations we replace [5], leading us to believe that a declarative approach can result in marked gains in software productivity and maintainability.

Given this starting point, we are pursuing several research issues that build on the basic language:

Query Optimization: Given that we can implement nearly all layers of a sensor network program in a declarative language, there is an opportunity to optimize across them all simultaneously. In addition to the intrinsic interest of this as a database research problem, it can lead to auto-generation of new network protocols, which can inform the development of networking technology. As an aspect of this challenge, we need an extensible optimizer to support new and custom optimization rules as they arise. To that end, we are designing a declarative *meta-optimizer* for P2 and DSN: an optimizer for distributed variants of Datalog that is itself written in Datalog. This turns out to be fairly natural: cost-based statistics-gathering can be expressed in queries, dynamic programming is a simple recursive program in Network Datalog, and recursive rule rewrites like Magic Sets Optimization are easy to express as well.

Heterogeneous Sensornets: The first generation of sensornet database systems focused on a network of homogeneous devices like Berkeley motes. A more realistic platform might mix these low-function devices with a variety of more capable but more power-hungry nodes. The design of protocols and algorithms that account for this heterogeneity is an important challenge in sensornets. Given P2 and DSN as execution engines targeted at large- and small-footprint devices, we are considering how to inform a query optimizer to synthesize protocols that make use of the heterogeneous resources intelligently.

Power Management: Related to the previous two points, a key aspect of sensornet programming is power management. Query optimization for sensor network protocols and applications must center on power as a key metric. Beyond that, the programming metaphor should expose power metrics and controls to the language, so that the programmer can control the optimization tradeoffs in an application-specific way. This requires a much richer interface between language and optimizer than the "optimizer hints" used in database systems. We are optimistic that a meta-optimizer framework can accelerate the exploration of these kinds of designs.

Distributed Statistical Methods: Sensor data is characteristically noisy and incomplete, so even the "data management" aspect of sensornet databases needs to be upgraded in the next generation. There has been a fair bit of research in recent years on statistical techniques to deal with these challenges, including work by the authors. But to date there has been no software infrastructure to make the programming of distributed statistical methods tractable. We hypothesize that languages like SNLog are actually a good fit to expressing distributed versions of these algorithms as well, and we have prototyped core statistical algorithms like Bayesian Belief Propagation. Making such programs fit into embedded sensor devices is an interesting additional challenge, given the memory footprint needed for maintaining sufficient statistics for these algorithms.

4 Complex Data Processing

Early tiny databases only allowed a limited set of basic database operations over data, which made them insufficient for a number of scientific applications that require sophisticated signal processing and analysis. For example, in one deployment in which we have been heavily involved, we are looking at the use of a collection of microphone-equipped sensors to localize, track, and classify wildlife in the field. One group of scientists we have been working with are particularly interested in populations of yellow-bellied marmots — large rodents endemic to the western US (see Figure 2). Such acoustic applications require high data rate (10's of kHz per channel) audio signals to be converted into the frequency domain (using an FFT operator) to identify frequencies that are characteristic of the animals being tracked. When such frequencies are detected, beamforming algorithms that perform triangulation of the signal are applied for tracking purposes.

Figure 2: An angry marmot.

To support these kinds of applications, we have developed a new programming language called WaveScript and a sensor network runtime system called WaveScope [9]. WaveScript is a functional programming language that allows users to compose sensor network programs that perform complex signal and data processing operations on sensor data. Similar to most streaming database systems, WaveScript programs can be thought of as a chain of operators that sensor data is pushed through; each operator filters or transforms data tuples and (possibly) passes those tuples on to downstream operators.

Unlike TinyDB, WaveScope doesn't impose a particular (tree based) communication model on sensors, but instead allows programs to operate on named data streams, some of which may come from remote nodes. To simplify programming tasks that involve data from many nodes, programmers can group nodes together, creating a single, named stream that represents the union or aggregate of many nodes' data streams.

Like TinyDB, WaveScope derives a number of benefits from the use of a high level programming model. Programmers do not need to worry about time synchronization, power management, or the details of networking protocols – they simply express their data processing application using the WaveScriptlanguage and the WaveScoperuntime takes care of executing these programs in efficient manner.

In the remainder of this section, we briefly overview the WaveScope data and programming model, focusing on expressing a few simple example applications.

4.1 The WaveScript Data Model

The WaveScript data model is designed to *efficiently* support high volumes of isochronous sensor data. Data is represented as streams of tuples in which each tuple in a particular stream is drawn from the same *schema*. Each field in a tuple is either a primitive type (*e.g.*, integer, float, character, string), an array, a set, a tagged union, or a special object kind of object called a *signal segment* (SigSeg).

A SigSeg represents a window into a signal (time series) of fixed bit-width values that are regularly spaced in time (isochronous). Hence, a typical signal in WaveScope is a stream of tuples, where each tuple contains a SigSeg object representing a fixed sized window on that signal. A SigSeg object is conceptually similar to an array in that it provides methods to get values of elements in the portion of the signal it contains and determine its overall length.

Although values within a SigSeg are isochronous, the data stream itself may be asynchronous, in the sense that the arrival times of tuples are *not* constrained to arrive regularly in time.

The WaveScope data model treats SigSegs as first-class entities that are transmitted in streams. This is unlike other streaming database systems [19, 3, 4] that impose windows on individual tuples as a part of the execution of individual operators. By making SigSegs first-class entities, windowing can be done once for a chain of operators, and logical windows passed between operators, rather than being defined by the each operator in the data flow, greatly increasing efficiency for high data rate applications.

4.2 The WaveScript Language

With WaveScript, developers use a single language to write all aspects of stream processing applications, including queries, subquery-constructors, custom operators and functions. In contrast, tiny databases and stream processing systems [19, 3, 4] typically provide a high level SQL-like language for writing queries but require user-defined functions to be written in external programming languages such as C. The WaveScript approach avoids mediating between the main script and user-defined functions defined in an external language, while further allowing *type-safe* construction of queries. In contrast, while SQL is frequently embedded into other languages, there is no compile-time guarantee that such queries are well-formed.

An example WaveScript script: Figures 3 and 4 show an WaveScript script query, first as a workflow diagram and then as the equivalent WaveScript subquery. The `marmot` function uses `detect`, a reusable detection algorithm, to identify the portions of the stream most likely to contain marmot calls, and then extracts those segments and passes them to the rest of the workflow, which enhances and classifies the calls. Several streams are defined: `Ch0..3` are streams of `SigSeg<int16>`, while `control` is a stream of `<bool,time,time>` tuples. Type annotations (*e.g.*, line 2) may be included for clarity, but they are optional. Types are inferred from variable usage using standard techniques [18]: for example, the definition of the `beamform` function implies that the type of `beam` is `Stream<float[360],SigSeg<float>>`.

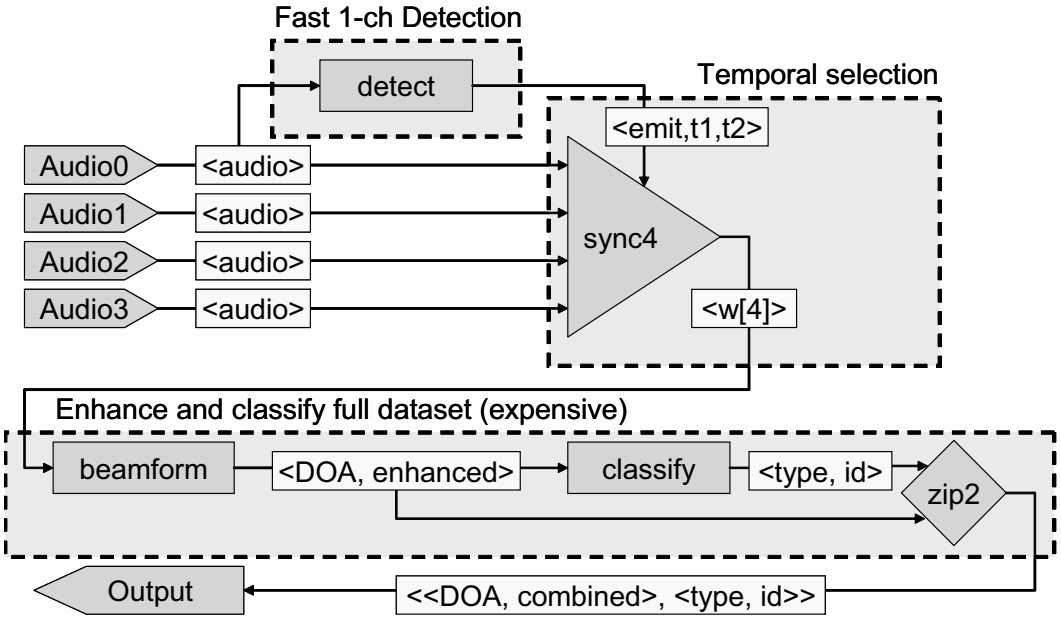

Figure 3: Marmot call detection workflow.

The `detect` subquery constructor has several parameters, including `marmotScore`, a custom function which computes the energy in frequency bands that are characteristic of marmot calls. This produces a `<bool,time,time>` stream of local marmot call detections that is merged with elements from the `net` input stream (representing a stream of tuples from a remote node). This merged and filtered stream is fed as a control stream to `sync4`, along with the four raw input streams from the audio sensors. `Sync4` aligns the four data streams in time, and "snapshots" synchronized segments of data according to the time-ranges specified in the control stream. In this way, `sync4` reduces the volume of input data by passing through only those segments of audio that `detect` suspects contain marmot calls.

Next, the synchronized windows of data from all four audio channels are processed by a beamforming algorithm. The algorithm computes a direction-of-arrival (DOA) probability distribution and enhances the input

```
fun marmot(Ch0, Ch1, Ch2, Ch3, net) {
  Ch0 : stream<SigSeg<int16>>
  // Detector on sensor inputs
  control : stream<bool,time,time>
  control = detect(Ch0, marmotScore, <64,192>,
                   <16.0, 0.999, 40, 2400, 48000>);
  // Control stream used to extract data windows.
  windows : stream<SigSeg<int16>[4]>
  windows =
    sync4(filter_lapped(merge(control, net)),
          Ch0, Ch1, Ch2, Ch4);
  // ... and process them: enhance
  beam<doa,enhanced> = beamform(windows, geometry);
  // ... and classify
  marmots = classify(beam.enhanced, marmotSig);
  // Return tuple of control and result streams
  return <control, zip2(beam, marmots)>;
}
```

Figure 4: Equivalent WaveScript subquery.

signal by combining phase-shifted versions of the four channels according to the most likely direction of arrival. The `beam` function returns a stream of two-tuples. We use a special binding syntax, "`beam<doa,enhanced>` = ...", to give temporary names to the fields of these tuples. That is, `beam.doa` projects a stream of direction-of-arrivals, and `beam.enhanced` contains the enhanced versions of the raw input data. Finally, this enhanced signal is fed into an algorithm that classifies the calls by type (male, female or juvenile), and, when possible, identifies individuals.

4.3 The WaveScope Runtime

In this section, we briefly review the runtime system which executes compiled WaveScript programs. Operators in compiled WaveScript query plans are run by a scheduler, which picks boxes one at a time and runs them to produce outputs.

A special *timebase manager* is responsible for managing timing information corresponding to signal data. This is a common problem in signal processing applications, since signal processing operators typically process vectors of samples with sequence numbers, leaving the application developer to determine how to interpret those samples temporally. A timebase is a dynamic data structure that represents and maintains a mapping between sample sequence numbers and time units. Examples of timebases include those based on abstract units (such as seconds, hertz, and meters), as well as timebases based on real-world clocks, such as a CPU counter or sample number.

Finally, a *memory manager* is responsible for creation, storage, and management of memory, particularly as is associated with SigSeg objects that represent the majority of signal data in WaveScope. Designing and efficient memory manager is of critical importance, since memory allocation can consume significant processing time during query processing.

These features – the timebase manager, memory manager, and scheduler – simplify the task of the programmer (who no longer has to worry about these details), and allow WaveScope to provide excellent performance. In our initial benchmarks of the marmot application, the current WaveScopesystem processes 7 million samples per second on a 3 GHz PC, and about 80K samples per second on a 400 MHz ARM platform (which has a 10x penalty for floating point emulation in addition to a reduced processor speed).

5 HiFi: A Data-Centric Architecture for Large-Scale Sensor Networks

Early work on wireless sensor networks, RFID, and other physical sensing infrastructure was necessarily focused on small-scale deployments designed in isolation from other computing and application systems. Starting from the assumption that these research efforts will ultimately lead to large-scale deployments, the HiFi project

looks towards the future, addressing questions that must be answered before such technology can be effectively integrated and utilized in realistic big-science and global enterprise scenarios.

The core issues that arise involve systems architecture, programming paradigms, optimization, and integration with existing IT infrastructure. Furthermore, "macroscope" sensor network deployments such as those being proposed by emerging scientific collaborations represent costly infrastructure investments whose economics dictate that they be shared by many user communities and applications. Such multi-purpose deployments require a rethinking of fundamental sensornet service abstractions.

To meet these challenges, the HiFi project is developing abstractions and corresponding software infrastructure to support "high fan-in" architectures: widely distributed systems whose edges contain large numbers of receptors such as sensor networks, RFID readers, and network probes, and whose interior nodes are traditional host computers organized using the principles of cascading stream query processing and hierarchical aggregation. Such systems have a characteristic fan-in topology; Readings collected at the network edges are continually filtered, summarized, refined and passed towards the network interior. Application scenarios include RFID-enabled supply chain management, large-scale environmental monitoring, and various types of network and computing infrastructure monitoring. The central premise of HiFi is that High Fan-in systems should be programmable with a uniform, declarative, data-centric language.

The initial architectural work on HiFi was described in a paper that appeared in CIDR 2005 [8] and a prototype of the system was demonstrated at VLDB 2004. HiFi builds upon the TelegraphCQ stream query processing engine [4] for in-network processing, and the TinyDB system for pushing queries out to the network edge. The key additional consideration of this initial effort was the development of software components to efficiently execute distributed, heterogeneous sensing applications and to allow summary data and alerts to be rapidly propagated through the system while less time- critical detail data is efficiently archived for later search and bulk transmission.

Experience with this initial implementation proved the feasibility of using SQL-based processing across the entire network and raised several new challenges. These new challenges have been the focus of subsequent work on HiFi:

Virtual Device Interface: The declarative approach lends itself naturally to the development of a "Virtual Device" abstraction layer that shields application developers from the complexity of the underlying devices [12]. We developed an API for this abstraction that allows Virtual Devices to be constructed from multiple, heterogeneous physical devices, enabling them to provide higher quality, application-specific readings. Furthermore, we extended the API to support data cleaning functionally, thereby avoiding the need to include such functionality in every application [11].

Edge Event Processing: A clear lesson from our early implementations was the need to enable users to easily express event patterns of interest and to enable event processing to be pushed closer to the sensors themselves. We developed an event pattern extension to SQL and demonstrated the use of this functionality in an RFID-based library check-out system [20].

Shared Hierarchical Query Processing: Another key challenge was the development of algorithms for efficiently processing multiple queries in the hierarchical HiFi environment. This work involved the development of sophisticated query re-writes on window clauses and predicates in order to produce efficient shared plans [13].

The work described above represents a start towards the development of an architecture for large- scale sensor deployments but many problems remain to be solved. The key insight of the HiFi effort was that database techniques such as declarative query processing and various levels of data abstraction can and should play a central role in the development of such solutions. From this perspective, HiFi can be seen as a direct extension of the original TinyDB philosophy.

6 Supporting Mobility

Finally, existing tiny database systems assume a relatively static and well-connected network topology. Though networks are formed dynamically, and do adapt to failed nodes or small changes in connectivity, the basic tree-based routing protocols used in these systems do not work well when nodes are highly mobile or when the network is often disconnected.

In the CarTel [10] project, we are building a mobile sensor networking system for data collection from vehicles. Applications include collection of speed and position data for monitoring traffic conditions, monitoring of road-surface quality using accelerometers (to, for example, provide reports of potholes in the Boston area), and tracking of cellular and WiFi connectivity is over the entire city. By mounting sensors are cars it is possible to cover a very large urban area with just a few sensors (we cover thousands of miles of road a day using a collection of 27-cab mounted sensors.) CarTel includes a so-called *delay tolerant networking layer*, called *CafNet*. The idea with CafNet (and other systems, like DTN [7]) is that it buffers data packets during periods of disconnectivity and delivers those packets during fleeting moments of connectivity, for example, when two mobile nodes pass each other or when a mobile node comes into range of fixed networking infrastructure that can transmit data to a centralized collection point.

Delay tolerant networking by itself, however, is not sufficient to support mobility in sensor networks. The issue has to do with fact that in mobile settings, bandwidth is highly variable: during times of good connectivity, it may be abundant, allowing large amounts of data to be collected. During periods of poor connectivity, however, it may be so scarce that if applications continue to produce data at the highest rate possible, buffers will become huge. Because most networking stacks deliver data in a first-in/first-out fashion, this means that old data will often clog buffers, preventing new (or more important) data from being able to propagate out of the network.

For this reason, we have developed a new data management component for CarTel called ICEDB (for Intermittently Connected Embedded Data Base) that allows programmers to specify *prioritization policies* for data collected by their queries [24]. The basic query model is very similar to TinyDB: users pose continuous queries that request data be collected at a particular rate from mobile nodes (on cars, in this case.) Nodes buffer data until connectivity is available and then deliver it with CafNet. However, because connectivity may be fleeting or limited, when new data is produced by a query, it is added to a priority queue of data waiting to be sent by CafNet with a priority specified by an additional clause that can be added to every query.

ICEDB allows both inter- and intra-query priorities to be specified. Inter-query priorities specify that all of the results of a given query should be transferred before any of the results of some other query. Intra-query prioritization is more sophisticated. It allows each individual data item to be assigned a priority according to one of two user-defined prioritization functions. If a *local* function is specified, the priority of all data items waiting to be sent by a given query are re-evaluated using the local function whenever a new data item is produced. For example, if a car is sending a trajectory of <location, speed> pairs, simply sending the most recent positions may not be desirable if the application needs to accurately approximate the path taken by a car. Instead, it may be preferable to send points that are in the middle of the largest sections of unsent data. We call this policy "bisect", as it recursively bisects the trajectory of points into smaller segments, providing an improved approximation of the entire path as more data is sent. To express this as a query, a user adds a "DELIVERY ORDER BY" clause to his query, as follows:

```
SELECT lat,long,speed
 FROM sensors
 SAMPLE PERIOD 1s
 DELIVERY ORDER BY bisect
```

Here, `bisect` is a built in function, but in general users can specify any function that takes as input a list of unsent points and totally orders them.

ICEDB also provides a *global* prioritization mechanism. Global prioritization is important in cases where there are multiple mobile sensors covering the same geographic area – for example, if there are several cars that

have each driven on the same section of the freeway, the data from later cars may have lower priority if another car has already delivered similar data. Global prioritization works in conjunction with a server. The idea is that each mobile node produces a compact *summary* of the data it has available and sends that summary to the server before sending the (much larger) raw data. The server then assigns a priority to each element in the summary, and sends the new priorities back to the mobile nodes. The mobile nodes then order the raw (unsummarized) data according to the priorities assigned by the server. To specify a summary, users provide a SQL query that computes a summary over the buffer of readings waiting to be sent, as well as a function that runs on the server that assigns priorities to each element in the summary.

Taken together, these local and global prioritization schemes allow ICEDB to adapt to the variable bandwidth that is inevitable in highly mobile sensing applications. Without such adaptation, programmers would have little control of when their data is transmitted, forcing them either to artificially constrain data collection rates to work with minimal bandwidth, or requiring them to develop complex home-grown networking and prioritization solutions.

7 Conclusion

Early tiny database systems provided a powerful set of declarative tools for collecting data from static sensor networks. Building on these previous results, our recent research has shown how to extend these ideas to other domains, including as network protocol specification, mobile systems, high data rate and signal-oriented systems, and integrated systems that including traditional information processing infrastructure. These next-generation tools maintain much of the ease of use, simplicity, and optimizability offered by using a database-like approach to system design while addressing key limitations of early systems.

Acknowledgments

This work was supported by the National Science Foundation under grants CNS-0205445, CNS-0520032, and CNS-0509261, and by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan.

References

[1] D. Abadi and S. Madden. Reed: Robust, efficient filtering and event detection in sensor networks. In *VLDB*, 2005.

[2] B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proceedings of the First Workshop on Information Processing in Sensor Networks (IPSN)*, April 2003.

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

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of First Annual Conference on Innovative Database Research (CIDR)*, 2003.

[5] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *ACM SenSys*, 2007.

[6] D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of ACM MOBICOM*, 2003.

[7] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the ACM SIGCOMM 2003*, August 2003.

[8] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The hifi approach. In *CIDR*, pages 290–304, 2005.

[9] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. The Case for a Signal-Oriented Data Stream Management System. In *Proc. CIDR*, Jan. 2007.

[10] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In *Proc. ACM SenSys*, Nov. 2006.

[11] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. In *Proceedings of the ACM Conference on Pervasive Computing*, pages 83–100, 2006.

[12] S. R. Jeffery, M. J. Franklin, and M. N. Garofalakis. An adaptive middleware for supporting metaphysical data independence. In *VLDB Journal (to appear)*, 2008.

[13] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006.

[14] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor network. In *NSDI*, 2004.

[15] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking with distributed recursive query processing. In *ACM SIGMOD International Conference on Management of Data*, June 2006.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of USENIX OSDI*, 2002.

[17] S. Madden, W. Hong, J. M. Hellerstein, and M. Franklin. TinyDB web page. `http://telegraph.cs.berkeley.edu/tinydb`.

[18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[19] R. Motwani, J. Widom, A. Arasu, B. Babcock, S.Babu, M. Data, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation and resource management in a data stream management system. In *Proceedings of First Annual Conference on Innovative Database Research (CIDR)*, 2003.

[20] S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, N. Burkhart, A. Edakkunni, and L. Liang. Events on the edge. In *Proceedings of SIGMOD*, pages 885–887, 2005.

[21] A. Tavakoli, D. Chu, J. M. Hellerstein, P. Levis, and S. Shenker. A declarative sensornet architecture. In *WWSNA*, 2007.

[22] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of SenSys*, 2003.

[23] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.

[24] Y. Zhang, B. Hull, H. Balakrishnan, and S. Madden. Icedb: Intermittently connected continuous query processing. In *Proceedings of*, 2007.