Implementing a Sensor Database System using a Generic Data Dissemination Mechanism

Omprakash Gnawali[‡], Ramesh Govindan[‡], and John Heidemann[§]

[‡]USC/Department of Computer Science, [§]USC/Information Sciences Institute
gnawali@usc.edu, ramesh@usc.edu, johnh@isi.edu

Abstract

The vision of a sensor network as a database has been reasonably well explored in the recent literature. Several sensor database systems have been prototyped [1, 11], and some have even been deployed [2]. However, these systems have largely integrated the query processing and the routing mechanisms. In this paper, we explore the design and implementation of a sensor database system (specifically, TinyDB [11]) on top of a generic sensor network data dissemination mechanism (Directed Diffusion [8]). Such a decoupled design is desirable, since it allows us to significantly re-use functionality and promotes overall system robustness. In conducting this exercise, we found that TinyDB influenced the re-design of Diffusion in several important ways.

1 Introduction

Prior work has proposed the use of database-like interfaces to program sensor networks. Enabling technology that uses SQL [3] to program sensor networks makes large-scale and fine-grained sensing accessible to scientists and researchers in other disciplines. Several such sensor database systems have been prototyped [1, 11], and some have even been deployed [2]. TinyDB [11] is one such system. TinyDB allows users to query a sensor network using SQL queries. In response to the SQL queries, query processors that run in each node in the network process and aggregate streams of sensor values much like how streams are processed in a database.

TinyDB developers have crafted a networking mechanism tailored specially for routing and topology maintenance in a TinyDB network. When queries are propagated to the network, a tree is formed which is used to route data back to the base station. While this might be adequate for specific platforms in which TinyDB has been developed, to make systems like TinyDB more flexible to changes in networking technologies and mechanisms in the future, we argue for building TinyDB on top of a standard networking substrate for sensor networks. The benefits of doing this are two-fold: it isolates networking from the application logic so that TinyDB can be ported to different networks with minimal change, and TinyDB or TinyDB-like system developers can reuse the networking layer services without having to roll out their own and spend a lot of effort debugging the networking layer. At a higher-level, our work can be said to be a concrete step towards examining how tuples in sensor databases can be routed using generic (as opposed to hand-crafted) routing mechanisms.

To demonstrate these benefits and to make a case for using a previously implemented and well-tested framework, we modified TinyDB to make it run in the filter framework [7] and over the Directed Diffusion [8] routing

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

protocol. Many database-like aggregation operations can be naturally expressed using these filters. Directed Diffusion provides a mechanism for naming the data in a generic fashion and also the routing protocols to efficiently search the network for relevant nodes and route the data back to the *sink* node.

Our system, called TinyDBD (TinyDB on Diffusion), has the same front-end as TinyDB. However, queries are injected into the network running Diffusion using *interest* messages. An interest message consists of a set of attributes that defines the query. Replies to the queries are routed to the querying node using *data* messages. A data message is also represented as a collection of attributes describing the data. The query processor is implemented as a filter on the nodes. Directed Diffusion allows for multiple nodes to inject interest messages at the same time. So, a TinyDBD system can have multiple active queries from different nodes at the same time.

Our implementation of TinyDBD runs on a testbed of PC104 nodes. It does not include any networking code and relies on Directed Diffusion to properly route queries and replies. In addition to supporting the TinyDB features, it is able to support multiple nodes issuing simultaneous queries.

In summary, the contribution of this paper is the design of, and experiences with, a framework for in-network processing that can be used to implement a SQL-like query system in sensor networks. Furthermore, we note the opportunity to reap the benefits of using a well-designed and well-tested framework for aggregation in a sensor network. Such benefits include isolation of aggregation and routing logic, assurance of functionality of the routing system, and potential to evolve the system to new platforms and new routing algorithms with minimal software changes.

2 Related Work

Directed Diffusion [8] is a data-centric networking substrate that provides mechanisms for naming, routing, and in-network processing in sensor networks. The filter framework [7] in Directed Diffusion allows custom code to run on the sensor nodes. Thus, filters can be used to process the information (aggregation) in the network before data reaches the querying node. TinyDBD is an attempt to use this architecture to implement an SQL-like query processing system. Prior work [5, 1] has suggested the feasibility of abstracting a sensor network as a database to increase the accessibility of the sensor data. TinyDB [11, 10] is an attempt to port database technology to sensor networks. TinyDB looks similar to desktop or server database systems at the interface layer. It comes with its own network stack. We integrated the TinyDB front-end with our Diffusion-based backend to demonstrate that applications like TinyDB can be built using the filter architecture in Diffusion and also to demonstrate the benefits of using an existing networking substrate to build such applications. Cougar [13] is another sensor database system that uses a query plan to determine the role of the nodes in in-network query processing. The Cougar project is also studying the interaction of characteristics of queries and the underlying routing layer.

3 Design

TinyDB on Diffusion (TinyDBD) exports an SQL interface to the end users of a sensor network. TinyDBD uses the filter architecture of Directed Diffusion to implement query processing mechanisms in the sensor nodes. It consists of three major components: (a) The base station, (b) Filter based query processing, and (c) Sensors.

The base station. The base station is a PC node that allows a user to formulate a query using a GUI. The station injects the query into the network using the transceiver node. TinyDBD supports multiple base stations injecting a query to the network at the same time. Each base station runs its own instance of the front end and the glue code. In a TinyDBD base station there are two main components: the (1) TinyDB front-end, which communicates with (2) Diffusion-based base station code. Figure 1 shows the architecture of the base station software.

We use the TinyDB front-end in TinyDBD. It is written in Java and presents an interface in which users can

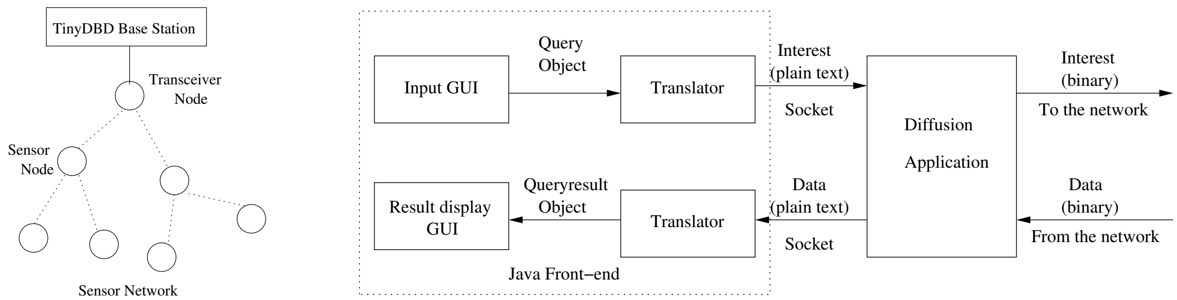

Figure 1: TinyDBD and Sensor Network (left) and architecture of the base station software (right)

```
SELECT AVG(light), temp
    FROM sensors
    GROUP BY temp
    SAMPLE PERIOD 2048
```

Figure 2: SQL Query

```
CLASS IS INTEREST_CLASS
PROTOCOL IS ONE_PHASE_PULL
TARGET IS DB
QID EQ 0
OP IS AVG
EPOCH IS 2048
LIGHT IS -1
FIELD IS TEMP
```

Figure 3: TinyDBD Query

```
CLASS IS DATA_CLASS
TARGET IS DB
QID IS 0
EPOCH IS 3
LIGHT IS 25
TEMP IS 15
```

Figure 4: TinyDBD Response

manipulate GUI controls to compose an SQL query. As in TinyDB, one sampling interval is called an *epoch* and is specified in the SQL query as a *sample period*. A sample query is shown in Figure 2. This query describes the result set consisting of average light values corresponding to each distinct temperature value in the network. This query specifies a sampling interval of two seconds.

In TinyDB, the glue code between the front-end and the transceiver node used serial communication and was designed for sending messages to a sensor network in TinyDB message format. We removed this code and inserted our code directly underneath the front end to generate messages in a format suitable for a network running Diffusion.

We parse the query object created in the TinyDB Java environment and compose the corresponding Diffusion messages. The idea is to translate the queries into a set of attributes that describe the orginal query. The glue code translates the query (Figure 2) to a Diffusion message shown in Figure 3. We serve the translated query in plain text using sockets to the Diffusion-based base station software. The base station, upon receiving this "interest" message, translates the text message into Diffusion-style attribute-value pairs. Table 1 presents the attributes used in TinyDBD Diffusion messages. The transceiver node injects this message into the network.

When the query results stream back to the transceiver node, it translates diffusion messages into text messages (Figure 4) and serves them to the Java based TinyDB front-end. The Java-based front-end translates the plain-text message into data structures expected by the TinyDB GUI.

Query processing in the nodes. Queries are distributed to the nodes in the network as Diffusion *interest* messages. Each node in the network has a filter, DBFilter, that listens to new queries in the network and sets up appropriate state to generate, aggregate, and forward results back to the querying node as data messages.

Diffusion provides a framework for in-network query processing using filters. TinyDBD uses the filter framework to do in-network processing (aggregation) of query results. DBFilter specifies the attribute `TARGET EQ DB` during the startup. This instructs Diffusion to forward to DBFilter all the messages that include the attribute `TARGET IS DB`.

When a node receives a query (an interest message), DBFilter sets up appropriate buffer and state for the query and floods the query to the neighboring nodes. This hop-by-hop flooding eventually distributes the query to the entire network. DBFilter also sets up a timer to wake up query processing module every epoch to process

Attribute	Description
CLASS	INTEREST_CLASS for queries, DATA_CLASS for results.
PROTOCOL	For TinyDBD, either ONE_PHASE_PULL or TWO_PHASE_PULL
TARGET	All the messages in TinyDBD is tagged with TARGET IS DB attribute
QID	Query ID. This is formed by concatenating HostID and a locally unique counter.
OP	Aggregation Operator such as AVG, MIN, MAX. If non aggregate query, OP IS SEL.
EPOCH	Duration of epoch in milliseconds.
LIGHT	For each field in the query, an attribute with that name is created (its value is ignored) and included in the interest message. In DATA message, the value of this attribute is the value of this field in the query.
FIELD	Group by field.

Table 1: Attributes used in TinyDBD messages.

the results. When the query results (data message) are forwarded by the neighbors, the receiving node buffers it until the begining of the next epoch. The query processing module then aggregates the buffered results and forwards it to the next hop towards the sink. If accumulated results are for a non-aggregate query, the module forwards all the messages and flushes the buffer.

DBFilter is concerned only with appropriately processing the query results, and is the application-specific component running on each node. DBFilter itself is not involved in forming or discovering the paths to be used for forwarding the query replies. This is done by the Gradient filter, which is the generic routing component usable by other applications as well. The Gradient filter runs in the nodes at the highest priority, and it intercepts any incoming and outgoing messages to maintain and provide the routing information. While queries are flooded to the network, the Gradient filter keeps track of the incident edge for an interest. When the results stream back for that query, the Gradient filter looks up the interest cache for the next hop for the matching attributes in the data message (query result), and forwards the response to the neighbor. This is one possible configuration for Gradient filter. For discussion on other configurations, please see the discussion in Section 5.

Generating data (Sensor Application). Sensor nodes run an application to drive the sensors within the framework provided by the Diffusion application API. The Diffusion application API consists of two calls: *Subscribe* and *Publish*. *Subscribe* describes to the Diffusion software the type of messages that an application is interested in. Diffusion will forward messages to the application only if the attributes in a given message match the attributes specified by the application during its startup. *Publish* is used to push data to the network.

On startup, the sensor application subscribes for queries in the network. Then, it stays idle until it receives a query. Upon receiving a query, it stores the parameters for the query and schedules the sensing module to wake up once every epoch. The sensing module samples the sensors and makes *Publish* calls to disseminate fresh data. Alternatively, the sensor application could start publishing data to the network immediately after the startup rather than idling till the first query is received. The effect of initial idling is conservation of energy while there is no query running in the network. One can put the application in idle state after the query has expired to avoid spending energy to run and sample the sensors when there is no query. Either way, the Gradient filter running in each node ensures that the sensor data propagates to the network only when there are active and unexpired queries.

4 Implementation

TinyDBD was implemented using Diffusion 3.1.2 libraries in C++. The sensors and the front end make *Publish* and *Subscribe* calls. The database logic is written in a filter called DBFilter. We have tested TinyDBD on a 10-node PC104 network. Our limited experiments show that it is possible to implement a database-like system on Diffusion. During the implementation of TinyDBD, we decided to use a TCP socket interface between the TinyDB front end and Diffusion-based base station software rather than integrating these two pieces of software.

This enabled us to use previously available software as much as possible. It is possible to implement the database logic as an application on the nodes, but we used filters because they provide a more natural framework for in-network processing of packets.

We exposed a subtle bug in the design of Diffusion 3.1.2 during our implementation of TinyDBD. In TinyDBD, at the end of each epoch, DBFilter generates a new message aggregating all the messages collected during the epoch. However, the Gradient filter expects packet IDs to be preserved even when messages are aggregated; this is an unclean design since, ideally, the Gradient filter should have been matching on attributes as the semantics of Diffusion matching dictates. This bug has since been fixed in Diffusion 3.2.

5 Design Choices

Our goal was to demonstrate the feasibility of our approach for implementing in-network query processing in sensor networks. Yet, many interesting design issues came up during the project. In this section, we list some of the interesting design issues that arose in TinyDBD and explain their implications for sensor database design.

Schema. TinyDB has an online schema system called TinySchema. One can define new attributes and push them to the network. In TinyDBD, we did not implement this dynamic schema definition mechanism. Our sensor ID's are hard coded in the application code and base station software. We think that attributes will rarely need to be redefined in a running network. When attributes change and they need to be propagated to the nodes, we believe that generic code distribution mechanisms [9] can be employed for this purpose.

Disjunctive queries. TinyDBD relies on the Gradient filter to match the attributes and find the path back to the sink. Attribute matching works by matching *all* the attributes which effectively computes a conjunctive query. A query with predicate such as $light < 50$ *or* $sound > 10$ translates to the following set of attributes in Diffusion: `LIGHT LT 50, SOUND GT 10`. The default attribute matching mechanism will forward a data message to the next hop if both $light < 50$ and $sound > 10$ are true. This makes disjunctive queries nontrivial to implement in Diffusion. This is an example where our investigation revealed a shortcoming of Diffusion.

However, it is possible to support a disjunctive query by supporting a new attribute matching mechanism, where a match is found if at least one attribute corresponding to disjunctive fields matches. Alternatively, all the disjunctive clauses can be concatenated into a single attribute with an *is* operator to exclude it from the list of attributes to be matched by the Gradient filter: `PRED IS LIGHT LT 50 OR SOUND GT 10`. Note that this attribute has an *is* operator so it will match all the gradients at the routing layer. This shifts the responsibility of disjunctive query attribute matching to DBFilter.

Aggregating over the same epoch. The current implementation schedules in-network processing without any synchrony to the schedules of the children nodes in the aggregation tree. Even though the aggregation uses a single value from each child per epoch, data from different epochs can processed (compared, aggregated) together because the epochs are not synchronized. In the worst case, the aggregate data will have sensor values sampled at n different epochs where n is the depth of the forwarding tree.

We chose the above approach for simplicity in implementation and it is not a reflection of limitations of our platform. Synchronized schedules (where the parent wakes up after its children) ensure that the information from the bottom of the tree can propagate all the way to the root of the tree within one epoch. This is the approach taken by TinyDB. Tagging data with epoch ID's, maintaining buffer size equal to the depth of the tree, and aggregating data only with matching tags would be another possible solution.

Choice of routing algorithm. The only requirement that TinyDBD imposes on the underlying routing system is that forwarding paths get setup and forwarding uses attribute matching. Because of this minimal assumption, it is possible to run TinyDBD using a variety of routing protocols without any change in the software.

Gradient, which is a filter responsible for implementing the routing protocol, uses a One Phase Pull protocol by default, however it can be made to use other algorithms [6]. If Gradient is configured to use Two Phase Pull, that will change the way forwarding paths are set up. However, DBFilter will still work without any change. The requirement that a path is setup towards the sink is fulfilled in either case. DBFilter does not work with another variant of Diffusion called Push, because that would require computing and disseminating replies to all the possible queries before a sink injects a query to the network. One or Two phase pull implementations set up paths along the nodes that are able to reach the source nodes with the least latency. In a different study, we have modified One Phase Pull to use routing metrics other than latency [4]. This will enable forming query-reply forwarding paths along the edges that have different levels of resources or reliability depending on the routing metric. This mechanism could be transparently used by DBFilter.

Choice of Platform. Our implementation was done on PC-104 and PC based platforms running linux. After this project, a stable version of Diffusion called TinyDiffusion [12] has become available for the motes. It is now possible to implement TinyDBD on the mote platform but one needs to be careful to design less verbose communication protocols in this resource-constrained platform. Even though the basic functionality will be similar, performance in terms of message reliability is likely to be different in different platforms. Some platforms might also make available CPU cycles to perform sophisticated aggregation in the intermediate nodes.

6 Conclusion

We have demonstrated that Diffusion can be used to build a database-like, in-network query processing system for sensor networks. We also wanted to investigate how proper networking infrastructure can aid in the development of applications that require in-network aggregation. We note that a proper networking infrastructure with a framework for in-network processing (e.g., Diffusion) helps isolate application logic from routing logic. This makes the system easy to understand and maintain. Unlike TinyDB, we were able to leverage previous work in routing in sensor networks and build a query system that is more portable.

References

[1] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *MDM*, 2001.

[2] P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, and S. Madden. TASK: Sensor Network in a Box. In *EWSN*, 2005.

[3] C.J. Date and H. Darwen. *A Guide of the SQL Standard*. Addison Wesley, third edition, 1994.

[4] O. Gnawali, M. Yarvis, J. Heidemann, and R. Govindan. Interaction of Retransmission, Blacklisting, and Routing Metrics for Reliability in Sensor Network Routing. In *IEEE SECON 2004*.

[5] R. Govindan, J. M. Hellerstein, W. Hong, S. Madden, M. Franklin, and Scott Shenker. The sensor network as a database. Tech. Report 02-771, USC/CS, 2002.

[6] J. Heidemann, F. Silva, and D. Estrin. Matching Data Dissemination Algorithms to Application Requirements. In *SenSys*, 2003.

[7] J. Heidemann, F. Silva, Y. Yu, D. Estrin, and P. Haldar. Diffusion Filters as a Flexible Architecture for Event Notification in Wireless Sensor Networks. Tech. Report ISI-TR-556, USC/ISI, 2002.

[8] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Mobile Computing and Networking*, 2000.

[9] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *NSDI*, 2004.

[10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *ACM SIGMOD*, 2003.

[11] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[12] E. Osterweil, M. Mysore, M. Rahimi, and A. Wu. The Extensible Sensing System, 2003. Center for Embedded Networked Sensing (CENS) Poster.

[13] Y. Yao and J. Gehrke. The Cougar Approach to In-network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3):9–18, 2002.