REED: Robust, Efficient Filtering and Event Detection in Sensor Networks

Daniel J. Abadi and Samuel R. Madden

MIT CSAIL

{dna,smadden}@csail.mit.edu

Abstract

This paper presents an algorithm for handling many types of filters in sensor networks that cannot be expressed using a simple predicate. Specifically, the action of the filter may be predicated on sensor produced data where an entire table of sensor-data/result-value pairs are needed to resolve the filter. We describe and evaluate three algorithms that can perform these filters that take advantage of database distributed join techniques. Our join-based algorithms are capable of running in very limited amounts of RAM, can distribute the storage burden over groups of nodes, and are tolerant to dropped packets and node failures. REED is thus suitable for a wide range of event-detection applications that traditional sensor network database and data collection systems cannot be used to implement.

1. Introduction

A widely cited application of sensor networks is *event-detection* where a large network of nodes is used to identify regions or resources that are experiencing some phenomenon of particular concern to the user. Examples include *condition-based maintenance* in industrial plants [11], where engineers are concerned with identifying machines or processes that are in need of repair or adjustment. Another example is *process compliance* in food and drug manufacturing [22], where strict regulatory requirements require companies to certify that their products did not exceed certain environmental parameters during processing. A third class of applications is centered around homeland security, where shippers are concerned with verifying that their packages and crates were not tampered with in some unsavory manner.

A natural approach implementing such systems is to use an existing query-based data collection system for sensor networks. Through queries, a user can ask for the data he or she is interested in without concern for the technical details of how that data will be retrieved or processed. A number of research projects, including Cougar [28], Directed Diffusion [9], and TinyDB [16,17] have advocated a query-based interface to sensornets, and several implementations of query systems have been built and deployed.

Table 1: Example of a table of predicates used in condition-based monitoring

Condition_type	Time	Temp_thresh	Humid_thresh
1	10 pm	> 100° C	> 95 %
2	10 pm	> 110° C	> 90 %
3	10 pm	> 115° C	> 87 %
…	…	…	…

Unfortunately, these existing query systems do not provide an efficient way to evaluate the complex predicates these event-detection applications require – for example, in TinyDB [16], queries are limited up to three conjunctive filters (e.g., "`temp > 25`° C AND `nodeid = 7` AND `hour_of_day` BETWEEN 10 am and 6 pm"). TinyDiffusion [9] is similarly limited to just a few predicates. Unfortunately, many condition-based monitoring and compliance applications may have tens to thousands of conditions that need to be detected and reported; for example, we have been talking with Intel engineers deploying wireless sensornets for condition based maintenance in Intel's chip fabrication plants who report that they have thousands of sensors spread across hundreds of pieces of equipment that are each involved in a number of different manufacturing processes that are characterized by different modes of behavior [10,11].

In this paper, we present REED, a system for Robust and Efficient Event Detection in sensor networks that addresses this limitation, enabling the deployment of sensor networks for the types of applications described above. REED is based on TinyDB, but allows users to express queries that include complex, time and location varying predicates over any number of conditions. The key idea of REED is to store filter conditions as predicates in tables, and then use a fault tolerant protocol to distribute those tables throughout the network. Once these tables have been disseminated, each node applies the predicates to its readings by checking each tuple of readings it produces against all of the predicates, outputting a list of predicates that the tuple satisfies. This list of satisfying predicates is then transmitted out of the network to inform the user of conditions of interest. By performing this filtration in-network, REED can dramatically reduce the communications burden on the network topology, especially when there are relatively few satisfying tuples, as is typically the case when identifying failures in condition-based monitoring and process compliance applications. Reducing communication in this way is particularly important in many industrial monitoring scenarios when relatively high data rate sampling (e.g., 100's of Hertz) is required to perform the requisite monitoring [7]. Table 1 shows an example of the kinds of tables which we expect to transmit – in this case, the filtration predicates vary with time, and include conditions on both the temperature and humidity. Our discussions with various commercial companies (e.g., Honeywell and ABB) involved in process control suggest that these kinds of predicates are representative of many sensor-based monitoring deployments in the real world.

The database-savvy reader will have observed that the description of table-based predicate matching given above bears a great deal of similarity to joins in a database system. Indeed, some of the optimizations that we describe for applying these predicates are inspired by join techniques developed in the database community, though the small memory footprint and lossy and low bandwidth communication in sensornets requires significant alternations to traditional join techniques. Interestingly, both TinyDB [16] and Cougar [28] initially eschewed joins in their query languages as their authors believed joins were of limited utility – REED provides an excellent counter-example to this point of view. In fact, we have added support joins between external tables and sensor readings to TinyDB; users can now write queries of the form:

```
SELECT s.nodeid, a.condition_type
FROM sensors AS s, alert_table AS a
WHERE s.temp > a.temp_thresh
AND s.humidity > a.humid_thresh
AND s.time = a. time
SAMPLE PERIOD 1s
```

Here, we use TinyDB syntax, where `sensors` refers to the live sensors readings (produced once per second, in this case.) In REED the external `alert_table` (similar, for example, to Table 1) will be pushed into the network along with the query. The filter conditions will be evaluated by having each node match the `sensors` tuples that it produces with the conditions in the table, with matches producing tuples of the form `<nodeid, condition_type>` which are then transmitted out to the user.

Because storage on sensor network devices is typically at a premium (e.g., Berkeley motes have just a few kilobytes of RAM and half a megabyte of Flash), REED allows these predicate tables to be partitioned and stored across several sensors. It also can transmit just a fragment of the predicate table into the network, forcing readings which do not have entries in the table to be transmitted out of the network and filtered externally. REED attempts to determine which predicates are most important to send into the network based on historical observations of predicates which commonly are not satisfied.

We end with a caveat: the purpose of this paper is not to describe sophisticated signal processing or statistics-based algorithms for data filtration. Instead, we focus on the systems issues related to efficiently propagating large tables of predicates and evaluating join-like queries over them. We consider predicates that are individually simple, consisting of Boolean operations over real numbers. We omit any discussion of more sophisticated predicates that are sometimes needed in the types of monitoring applications described above. We note, however, that our approach is generalizable to more complex predicates, in that TinyDB is fully amenable to known techniques for extending databases via user-defined functions and predicates [26] that have been shown to be viable for various time-series and signal-based analyses.

1.1. Contributions

In summary, the major contributions of this work are:

- We show how complex filters can be expressed as tables of conditions, and show that those conditions can be evaluated using relational join-like operations.
- We describe the REED system and our sensor network filtration algorithms, which are tailored to provide robustness in the face of network loss and to handle very limited memory resources.
- We provide experimental results showing the substantial performance advantages that can be obtained by executing complex filters inside the sensor network, through evaluation in both simulation on a real, mote-based sensor network.
- We discuss a number of variants and optimizations of our approach, some of which are motivated by join optimizations in traditional databases and others which we have developed to address the particular properties of sensor networks.

Before describing the details of our approach, we briefly review the syntax and semantics of sensor network queries and the capabilities of current generation sensornet hardware.

1.2. Background: Sensor Networks and Motes

Sensor networks typically consist of tens to hundreds of small, battery-powered, radio-equipped nodes. These nodes usually have a small, embedded microprocessor, running at a few Mhz, with a small quantity of RAM and a larger Flash memory. Table 2 summarizes hardware characteristics of the Berkeley mica2 Mote, a popular sensor network hardware platform designed by UC Berkeley and sold commercially by Crossbow Corporation.

Storage: The limited quantities of memory are of particular concern for query processing, as they severely limit the sizes of join and other intermediate result tables. Although future generations of devices will certainly have somewhat more RAM, large quantities of RAM are problematic because of their high power consumption. Non-volatile flash can make up for RAM shortages to some extent, but flash writes are quite slow (several milliseconds per page, with typical pages less than 1 KB) and consume large amounts of energy – almost as much as transmitting data off of the mote [25].

Table 2: Hardware Attributes of the Mica2 Mote

Attribute	Value
Processor	7Mhz Atmega 128
Radio	38.6 Kbps CC1020 Range: ~100 ft
RAM	4 KB
Flash	512 KB
Battery	2xAA (2400 mAH)
Active Power Consumption	~15 mA
Sleep Power Consumption	~10 μA

Sensors: Mica2 motes include a 51-pin expansion slot that accommodates a number of sensor boards. Commonly available sensors can measure light, temperature, humidity, air-pressure, vibration, acceleration, and position (via GPS).

Communication: Radio communication tends to be quite lossy – without retransmission, motes drop significant numbers of packets. At very short ranges, loss rates may be as low as 5%; at longer ranges, these rates can climb to 50% or more [27]. Though retransmission can mitigate these losses somewhat, nodes can still fail, move away, or be subject to radio interference that makes them temporarily unable to communicate with some or all of their neighbors. Thus, any algorithm that runs inside of a sensor network must tolerate and adapt to some degree of communication failure.

TinyOS: Motes run a basic operating system called TinyOS [9], which provides a suite of software libraries for sending and receiving messages, organizing motes into ad-hoc, multihop routing trees, storing data to and from flash and acquiring data from sensors.

Power: Because sensors are battery powered, power consumption is of utmost concern to application designers. Power is consumed by a number of factors; typically, sensing and communicating dominate this cost [16,21]. In this paper, we focus on algorithms that minimize communication, as any join algorithm that includes all nodes in a network will pay the same cost for running sensors. We note that, if careful power management is not used, the cost of listening to the radio will actually dominate the cost of transmitting, as sending a message takes only a few milliseconds, but the receiver may need to be on continuously, waiting for a message to arrive. TinyDB and TinyOS address this issue by using a technique called *low-power listening* [20], where receivers sample 1 out of every k bits on the radio to see if someone is sending a message; if they detect a message, they wake up and begin receiving at full speed; otherwise, they sleep for the remaining k-1 bit times. Senders ensure that every message is preceded by a k-bit preamble. In this way, receivers never miss a message, but (by setting k to a large value, e.g., 100) it is possible reduce the cost of listening by approximately $1/k$, while increasing transmission cost by only k bits[1]. With appropriately aggressive low-power listening, the total number of messages transmitted on the radio channel dominates power consumption.

1.3. Background: TinyDB Data Model and Semantics

REED adopts the same data model and query semantics as TinyDB. Queries in TinyDB, as in SQL, consist of a `SELECT-FROM-WHERE` clause supporting selection, projection, and aggregation. REED extends this list of operators with joins. TinyDB treats data as a single table (`sensors`) with one column per sensor type. Results, or *tuples*, are appended to this table periodically, at well-defined intervals that are a parameter of the query, specified in the `SAMPLE PERIOD` clause. The period of time from the start of each sample interval to the start of the next is known as an *epoch*. Consider the query:

```
SELECT nodeid, light, temp
FROM sensors
SAMPLE PERIOD 1s FOR 10s
```

This query specifies that each sensor should report its own id, light, and temperature readings once per second for ten seconds. Thus, each epoch is one second long. The virtual table `sensors` contains one column for every attribute available in the system and one row for every possible instant in time. The term *virtual* means that these rows and columns are not physically materialized -- only the attributes and rows referenced in active queries are actually generated.

1.4. Data Collection in TinyDB

Query processing in the original TinyDB implementation works as follows. The query is input on the user's PC, or *basestation*. This query is optimized to improve execution – currently, TinyDB only considers the order of selection predicates during optimization (as the existing version does not support joins). Once optimized, the query is translated into a sensor-network specific format and injected into the network via a gateway node. The query is sent to all nodes in the network using a simple broadcast flood (TinyDB also implements a form of epidemic *query sharing* which we do not discuss.)

As the query is propagated, nodes learn about their neighbors and assemble into a *routing tree*; in TinyDB, this is implemented using a standard TinyOS service similar to what is described in the work by Woo *et al*. [27]. Each node in the network picks one node as its *parent* that is one network hop closer to the root than it is. A node's *level* is simply the number of radio hops required for a message it sends to reach the basestation.

As a node produces query answers, it sends them to its parent; in turn, parents forward data to their parents, until answers eventually reach the root. For some queries (and in our join implementation), parents will combine readings from children with local data to partially process queries within the network. The basestation assembles partial results from nodes in the network, completes query processing, and displays results to the user.

2. Applications and Query Classification

Given this basic introduction to sensors and TinyDB, we now describe some applications of REED. We use these applications to derive a classification of filter types that we will use to motivate the different algorithms we present in Section 3.

2.1. Query Types

REED extends the query language of TinyDB by allowing tables of filter predicates to appear in the FROM clause. In this section, we show the syntax of several example queries and describe their basic behavior.

[1] In practice, receivers may not be able to switch on and off in a single bit time. In such cases, if the switching time of the radio is b bits, we can make the preamble $b*k$ bits and still obtain a factor of k reduction in listening costs.

Industrial Process Control. Chemical and industrial manufacturing processes often require temperature, humidity, and other environmental parameters to remain in a small, fixed range that varies over time [8]. Should the temperature fall outside this range, manufacturers risk costly failures that must be avoided. Thus, they currently employ a range of wired sensing to avoid such problems [22,10]. Interestingly, companies in this area (e.g., GE, Honeywell, Rockwell, ABB, and others) are aggressively pursuing the use of mote-like devices to provide wireless connectivity, which is cheaper and safer than powered solutions as motes don't require expensive wires to be installed and avoid the risks associated with running high-voltage wires through volatile areas. Of course, for wireless solutions to be cost-effective, they must provide many months of battery life as well as equivalent levels of information as existing solutions. Thus, the power and communications efficiency of a system like REED is potentially of great interest.

It is easy to write a REED query that filters readings from sensors located at various positions with a time-indexed table of predicates that encodes, for example, allowable temperature ranges in a process control setting. Should the temperature ever fall outside the required range, users can be alerted and appropriate action can be taken. Such a query might look like:

```
(1)     SELECT a.atemp
        FROM schedule_table AS s,
             sensors AS a
        WHERE s.ts > t.tsmin AND
              s.ts < t.tsmax AND
              a.atemp > t.tempmin AND
              a.atemp < t.tempmax AND
              a.nodeid = t.nodeid
```

Here, results are produced only when an exceptional condition is reached (the temperature is outside the desired range), and thus relatively few tuples will match. We note that this is a *low selectivity* query, indicating that it outputs (*selects*) a small percentage of the original sensor tuples.

Failure and Outlier Detection. One of the difficulties of maintaining a large network of battery-powered, wireless nodes is that failures are frequent. Sometimes these failures are *fail-fast*: for example, a node's battery dies and it stops reporting readings. At other times, however, these failures are more insidious: a node's readings slowly drift away from those of sensors around it, until they are meaningless or useless. Of course, there are times when such de-correlated readings actually represent an interesting, highly localized event (i.e., an outlier). In either case, however, the user will typically want to be informed about the readings. We have implemented a basic application that performs both these tasks in REED. It works as follows: we build a list of the values that each node commonly produces during particular times of day from historical data and periodically update this list over time. We then use this list to derive a set of low-probability value ranges that never occur or that occur with some threshold probability ε or less frequently. Then, we run a query which detects these unusual values. For example, the following query detects outlier temperatures:

```
SELECT s.nodeid, s.temp
FROM sensors AS s, outlier_temp AS o
WHERE s.temp
      BETWEEN o.low_temp AND o.hi_temp
AND s.roomno = a.roomno
```

This query reports all of the readings that are within an outlier range in a given room number. Note that the `outlier_temp` table may be quite large in this case, but that the selectivity of this query is also low.

Power Scheduling. As a third example, consider a set of sensors in a remote environment where power conservation is of critical importance. To minimize power consumption in such scenarios, it is desirable to balance work across a group of sensors where each sensor only transmits its light reading some small fraction of the time. We can do this with an external table as well; for example:

```
SELECT sensors.nodeid, sensors.light
FROM sensors, roundrobin
WHERE sensors.nodeid = roundrobin.nodeid
  AND sensors.ts % |nodes| = roundrobin.ts
```

For this query, the `roundrobin` table is small (≤ `|nodes|` entries), and can likely fit on one node. This filter also has a low selectivity, as only one or two nodes satisfy the predicate per time step.

2.2. Query Classification and Optimization Tradeoffs

These queries allow us to make several observations about how and where we should execute filters. In general, it is advantageous to perform filters with low selectivity in the sensor network. This is because there will be many fewer results than original data and thus a smaller number of transmissions needed to get data to the basestation.

There are situations, however, when we might prefer not to push a filter into the network; for example, if the filter has a relatively high selectivity, and the size of the join table is very large, the cost of sending the filter into the network may exceed the benefit of applying the filter inside the network. We may also be unable to push a filter into the network if the size of the predicate table exceeds the storage of a single node or a group of nodes across which the table may be partitioned.

Thus, in REED, we differentiate between the following types of filters:

- *Small* filter tables that fit in the memory of a single node.
- *Intermediate* filter tables that exceed the memory of a single node, but can fit in the aggregate memory of a small group of nodes.
- *Large* filter tables that exceed the aggregate memory of a group of nodes.

We have developed filtration algorithms (all based on joins) that are suitable for all three classes of tables; we describe these algorithms in Sections 3 and 4 below.

For small filter tables, REED always chooses to push them into the network if their selectivity is smaller than one. For intermediate tables, the REED query optimizer makes a decision as to whether to push the filter into the network based on the estimated selectivity of the predicate (which may be learned from past performance or gathered statistics, or estimated using basic query optimization techniques [25]) and the average depth of sensor nodes in the network. It uses a novel algorithm to store several copies of the filter table at different groups of neighboring nodes in the network, sending each sensor tuple to one of the groups for in-network filtration.

For large filters, as well as intermediate filters that REED chooses not to place in-network, REED can employ a third set of algorithms that send a subset of the filter table into the network. REED tags this subset with a logical predicate that defines which sensor readings it can filter. For example, for Query (1) above, a filter subset might be tagged with a predicate indicating it is valid for nodes 1-5 at times between 5 am and 5 pm. For readings from these nodes in this time period, filters can be applied in-network; other readings will have to be transmitted out of the network and filtered externally. We describe algorithms for this kind of *partial filtering* in Section 4. If REED chooses not to apply partial filtering, all nodes transmit their readings out of the network where they are filtered externally.

In the following section, we present two algorithms: the first is a single-node algorithm for small filter tables. The second shows how to generalize this single-node technique to a group of nodes that work together to collectively store the filter table. We show that these algorithms are robust to failures and changes in topology as well as efficient in terms of communication and processing costs.

3. Join-Based Filter Algorithms

Once the query optimizer has decided to push a REED query into the network, we need an algorithm for applying our filters efficiently; in this section, we describe our approach for performing this computation. We focus on distributing and executing our filters throughout the network in a power-efficient manner that is robust in the face of dropped packets and failed nodes. Logically, our algorithms for filtration can be thought of as a *nested-loops join* between current sensor readings and a table of static predicates. Thus, for the remainder of this paper, we describe our filter algorithms in terms of joins, as what we have implemented in REED is actually a general purpose join processor.

Nested-loops joins are straightforward to implement in a sensornet, as shown by the following algorithm:

```
Join(Predicate q)
  for each tuple tr in sensors do
    for each tuple ts in predicates do
      if q(tr, ts) is satisfied
          add tr ∪ ts to result set r
    end
  end
  return r
```

There are two things to note about this algorithm. First, low selectivity filters might cause there to be fewer than one result (on average) per element of the outer loop, though it is in general possible for each tuple to match with more than one predicate. In such a scenario, it is advantageous to apply our filters as close as possible to the data source in a sensor network since this would reduce the total number of data transmissions in the network. Second, elements of *predicates* are independent of each other. Thus, *predicates* can be *horizontally partitioned* into a number of non-overlapping sub-tables, each of which can be placed on separate nodes. As long as the table partitions are disjoint, the union of the results of the filter on the independent nodes storing partitions of the table is equal to the results of the filter if the entire static table was stored at one location.

These two observations motivate our algorithms. The join-based filter is applied as close as possible to the data source. For the case where the static table fits on one sensor node, the static table is sent to every sensor node (using the TinyDB query flood mechanism) and the filter is performed on a sensor node as soon as the data is produced. For the case where the static table does not fit on one node, the predicates table (s) is horizontally partitioned into n disjoint segments s_1, s_2, …, s_n ($s=s_1 \cup s_2 \cup \ldots \cup s_n$). Each s_i is sent to a member of a group of sensor nodes in close proximity to each other formed specially to apply the filter. Each group is sent a copy of the predicates table. When a sensor data tuple is generated, it is sent to each node in exactly one of these groups to join with every partition (s_i) of the predicate table.

In Section 3.1 we describe in more detail the case where the predicates table fits on one node. In Section 3.2 extend this basic algorithm with a distributed algorithm for the case where the table is too big to fit on one node.

3.1. Single Node Join

Our join algorithm leverages the existing routing tree to send control messages and tuples between the nodes and the root. When a query involving a join is received at the basestation, a message announcing the query is flooded down to all the nodes. This announcement (actually implemented as a set of messages) is an extended version of the TinyDB “new query” messages, and includes the schema of the sensor data tuples, the name, size, and schema of the join table, the schema of the result tuples, and a set of expressions that form the join predicate. Upon receiving the complete set of these messages, every node in the sensor network knows whether it is participating in the query (by verifying that it contains the sensors that produce the fields in the schema) and how many tuples of the join table can be locally stored (by comparing the size of each join table tuple with the storage capacity the node is willing to allocate to the query).

If the node's storage capacity is sufficient to store the filter predicates table, the node simply sends a message to the root, requesting the table and indicating that it intends to store the entire table locally. The root assumes that there will likely be other nodes who can also store the entire table, so it floods each tuple of the table throughout the sensornet. Once the entire table is received, the node can begin to perform the join locally, transmitting the join results rather than the original data. Before then, nodes run a naïve join algorithm, where also sensor tuples are sent to the root of the network to be joined externally.

A simple optimization that can be performed is that if the result of the join consists of more than one tuple, the node can simply send the original sensor tuple. The join for this tuple can then be performed at the basestation; this technique is equivalent to *semi-joins*, a well known technique for join evaluation in database systems [1].

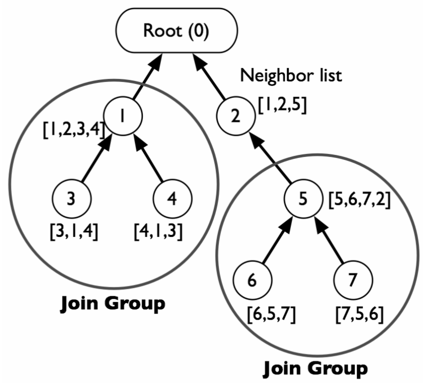


Figure 1: REED routing and join tree with group overlays

3.2. Distributed Join

In this section, we describe our in-network join algorithm in detail. Our algorithm consists of three distinct phases: group formation, table distribution, and query processing. We begin with a brief overview, and then describe each phase in turn.

3.2.1. Algorithm Overview

When the predicates table does not fit on one node, joins can no longer be performed strictly locally. Instead, the table must be horizontally partitioned. A tuple can only immediately join with the local partition at the node and must be shipped to other nodes to complete the join. Once the original tuple has reached every node that contains a partition of the table, it can be dropped and results can be forwarded to the root. Within each group, the members are within broadcast range of each other. Our group formation algorithm is described in detail in the next section.

Figure 1 shows the setup of such a distributed join query. The figure shows a multi-hop routing tree where tuples are passed to their parents on their path to the root basestation. For example, a tuple produced by node 7 is sent to node 5 which then sends the tuple to node 2 which sends the tuple to the basestation. Our join algorithm works by overlaying groups (shown as large circles in Figure 1) on top of this routing tree. The numbers in parentheses in the figure represent the set of nodes in broadcast range for that particular node. A tuple that needs to be joined is broadcast from a node to the other members of its group. Each member sends any joined results up the original routing tree. For example, if node 7 produces a tuple to be joined, it broadcasts it to nodes 5 and 6. If node 5 contains a tuple in the table that successfully joins with 7's tuple, it sends the result up to node 2 which forwards it to the root.

Note that when node 7 produces a tuple, three transmissions results – this is the same as if the original data was sent up the routing tree in the naïve or single-node case. In the worst case, there would have been two extra tuples: if node 5 produced a tuple which joined with a tuple on node 7 a total of 4 transmissions would have been performed. In general, no more than 2 + *depth* transmissions will be required, as any pair of nodes in the same group differ by no more than one level (by definition). For joins with predicates of low selectivity there will be many cases where no element of the table joins with the original data. When this occurs, performing the join in the group rather than sending the tuple back to the root can provide savings proportional to the depth of that group (instead of the *n* hops to get the original data to the root, only one transmission of the original data is made).

We now describe the algorithm that each node performs when it receives a join query with a predicates table whose size is too large to fit on that node. We discuss how groups

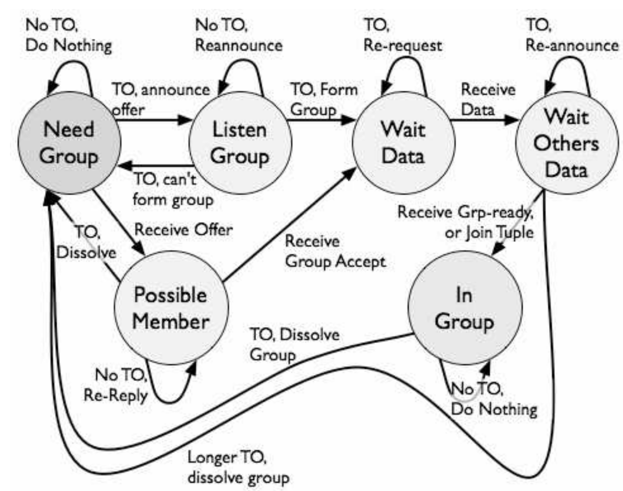


Figure 2: Join Algorithm Finite State Machine. The "TO" transitions represent timeouts, which are used to prevent deadlocks in the event of lost data or failed nodes.

are formed, how the table is distributed, and then the group join operation.

3.2.2. Group Formation

If a node calculates that it does not have enough storage capacity for the table, it initiates the group formation algorithm. To minimize the number of times an original tuple must be transmitted to make it to every member of a group, we require that all nodes in the group are within broadcast range of each other. A second required property of a group is that it must have enough cumulative storage capacity to accommodate the table of predicates. Group formation is a background task that happens continuously throughout the lifetime of the join query as nodes come and go and network connectivity changes. Every group can be uniquely identified by its groupid and the queryid to which the group belongs. To simplify group formation and table distribution, we only allow a node to belong to one group for a given query. A node may belong to different groups for different simultaneously running queries.

Every node maintains a global, periodically refreshed list of neighbors that are within broadcast range. For each neighbor, an estimate of incoming link quality is computed by snooping on messages sent by surrounding nodes. Each REED message contains a counter, and upon overhearing a message, a node can calculate how many messages it missed since the previous message by subtracting the previous message's counter from the current counter. The receive rate can then be calculated by dividing the number of messages received by the total number of messages sent. Note that links may be asymmetric. For this reason, we assume that the neighbor list only contains information about incoming link quality from each node. Outgoing link quality is accounted for elsewhere. A neighbor node is placed on the neighbor list if the receive percentage is above some threshold (defaulting to 75%). This algorithm is similar to the algorithm used for measuring link quality in the TinyOS multihop radio stack [27], and we have been able to use the standard TinyOS implementation of this table in our REED implementation.

Once a node receives notification of a new query, it enters into a group formation finite state machine (FSM – see Figure 2). Each join query has its own FSM which, barring resource conflicts, operates independently. Every node is initialized to the *Need Group* state with a randomly set timeout value. Nodes transition between states in this diagram once per epoch. Groups are formed when a node (which will become known as the master node) in the *Need Group* state reaches a timeout and broadcasts a request to form a group to neighboring nodes (while transitioning to the *Listen Group* state). All nodes which hear this request and are also in the *Need Group* state (implying that they are not currently in groups and do not have offers pending to other potential masters) respond with offers to be members if the master node is in their neighbor list. These offer messages include the neighbor list of the sender and the number of join table tuples which the sender can locally store. Responding nodes move into the *Possible Member* state which prohibits them from offering themselves to other masters until a response is received or a timeout occurs. In order to minimize the probability that the master does not receive the reply and the node has to wait until it times out to be available again, a node in the *Possible Member* state retransmits its reply every fixed time period (one epoch) until it hears a response from the desired master.

The master node collects responses from available nodes for a fixed duration (occasionally retransmitting the group formation request in case previous requests were lost). At the end of this duration, the master chooses the group members. The algorithm to choose the group members attempts to make a group that involves the maximum number of nodes such that every node is in radio range of every other. This is done using the neighbor lists that each neighbor sent to the master and the master's own neighbor list. The master also must ensure that the group has enough combined storage capacity to store the entire predicates table. Ideally, a node would use the neighbor list of every node to create the optimal group. However, since memory is severely limited (and we want to save as much memory as possible for the storage of the table) the master node does not store the neighbor list of every reply it receives, but instead keeps a running intersection of current candidate group members' neighbor lists. Upon receiving a reply, the master takes the intersection of its current list with the neighbor list received in the reply. The master also keeps a running aggregate of the total storage available on all of the nodes. If the resulting intersection is sufficiently large (and the nodes previously accepted are still in the intersection), the master includes the node into the group and keeps the intersection. Otherwise, it rejects the node and rolls back to the previous intersection. Pseudo code for this greedy algorithm is shown in Figure 3.

Note that this algorithm takes into account the possibility of asymmetric links. To be in the same group, every group member must appear on the neighbor list of every other member. Thus, (for the example of two nodes, *a* and *b*) node *b* must appear on node *a*'s neighbor list and node *a* must appear on node *b*'s neighbor list.

```
t = some threshold parameter
N = current neighborlist
G = my.id //G will hold current group
   // member list which is initialized
   // to just contain the master
space = my.space
for every reply from node i do
   id = i.id
   M = i.neighborlist
   P = intersect(N, M)
   if (|P| > t) and (P ∩ G = G) do
      N = P;
      G = union(G, id)
      space = space + i.space
   end
end
```

Figure 3: Group Formation Algorithm

At the end of the time out period, the master examines the current group's aggregate storage space. If it is larger than the size of the predicates table, it can go ahead and inform the group of its successful creation. Otherwise, it must announce that it failed to create a group. If it succeeded in creating a group a message is broadcast containing the list of accepted members and the master transitions to the *Wait Data* state. Nodes that receive this broadcast check to see if they are on the accepted members list. If so, they also transition to the *Wait Data* state. Otherwise, they revert back to the *Need Group* state. As stated above, nodes in the *Possible Member* state periodically retransmit a group reply. Thus, if the group announcement broadcast is lost, these nodes will continue to send replies that will cause the master to retransmit the group announcement. Eventually, word will get to all possible members whether or not they are accepted. The group id of the resultant group is the node id of the master node.

Although now officially a query group, nodes in the group cannot start processing tuples as a group until the join table has been distributed. It is the responsibility of the basestation to keep track of which parts of the join table have been issued to which members of a group. This information is kept in a table indexed by group id and query id. Upon entering into a group and transitioning into the *Wait Data* state (either as a master node which just successfully formed a group or as a potential group member node which received the group accept broadcast), a node sends a *join table request* message to the root (which is periodically resent if no response from the root occurs before a timeout). This message includes the groupid, queryid, and the number of filter predicates that can be accepted. This number is computed from the space available at the node and the schema of the join table.

The basestation receives the join table requests and, based on the queryid, groupid and tuple capacity, decides which portion of the join table should be sent back to the node.

Upon receiving a set of join tuples from the root, a node transitions to the *Wait Others Data* state and broadcasts to its group the number of tuples it received (this message is also periodically rebroadcast while a node remains in the *Wait Others Data* state). Each node keeps track of which nodes have received how many tuples, and independently determines when the entire join table has been distributed to the group (since the join table size was distributed with the original query request). When any member of the group decides that the entire group has received the join table, it transitions to the *In Group* stage and broadcasts a *group ready* message. Nodes that hear the group ready message also transition to the *In Group* state. Nodes that do not hear the message will transition to the *In Group* state as soon as it overhears a message from one of the nodes who know that the group is ready (and is in the *In Group* state) that contains a broadcast a tuple for the group to join.

3.2.3. Operation

Sensor data tuples that need to be processed by a node are generated either by the sensors on the node itself or received from children in the REED routing tree. Nodes are responsible for forwarding child sensor data tuples at all times during the query, whether or not they are in an active join group. Until transitioning to the *In Group* state, all data tuples are forwarded up to the parent node in the REED tree. If all nodes along the way to the root basestation are not members of active groups, then the network behaves like the naive join with all the original sensor data tuples being forwarded to the root where the join is performed.

However, if a node along the way *is* in the *In Group* state, then instead of forwarding the data message to its parent, it broadcasts the tuple to its group. Each group member then joins that data tuple with the locally stored portion of the join table and forwards the resulting joined tuples up the original REED tree; these result tuples need no more joining and can be output once they reach the root.

Nodes in the In Group state eventually time out and attempt to dissolve the group. They also might choose to dissolve the group if it senses that a node has ceased to respond or if the message loss percentage from a node in the group rises above the desired threshold. A node dissolves a group by broadcasting a group dissolve message and transitioning to the *Need Group* state. Nodes that receive this message also transition to the *Need Group* state. Nodes that do not will continue to try to send data tuples to the group which will cause the node that dissolved the group to retransmit the dissolve message and forward these data tuples back up the original REED routing tree.

3.3. Robustness Under Message Loss and Node Failure

It is critical that the distributed join algorithm just described works properly in the face of message loss and node failure. If the table fails to distribute properly, or if nodes in a group incorrectly make assumptions regarding the data storage of other members of the group, data loss is compounded due to repeated errors in data processing within the group. For example, if node *y* times out (or fails or moves out of broadcast range) at any point in the group formation process and other nodes continue to process tuples as if that node were a group member, then all tuples processed by that group that joined node *y's* partition will be lost. Since these errors are so important to avoid, we briefly discuss the robustness properties of this algorithm.

The key attribute of this algorithm that allows it to avoid problematic scenarios is that each state is "soft" in the sense that if enough time elapses with a node stuck an any particular state, it gives up and returns to the *Need Group* state. Message loss and node failure are the primary reasons that a node might get stuck in a particular state. For example, a node that has sent out a response to a group offer might get stuck in the *Possible Member* state if the node it responded to either fails or further communication between the nodes is lost. For every state except the *Wait Others Data* and *In Group* states, reverting back to the *Need Group* state can happen for free, without causing any loss. This is because every node is transmitting data back to the basestation using the naïve algorithm until a group is formed, so until a node has announced that it owns a partition of the table, it has not made any promises to other nodes that it cannot easily renege. By timing out during the group formation process, a node might delay induction into a group for both itself and other nodes that it may have indicated interest in joining, but no data is lost.

For the case where a node times out (or fails) while in the *Wait Other Data* or *In Group* states, we allow a bounded amount of data to be lost. Our initial implementation of REED used a reliable round-robin ACKing protocol between nodes. In this protocol, a node in a group stored the last *k* tuples that it transmitted to its group. Every group member then sent an ACK at least once in every *k* epochs. If any group member failed to perform this ACK, the remaining group members assumed that none of their last *k* transmissions made it to this node and transmitted their stored block of k tuples via the naïve algorithm to the basestation. This aggressive reliability protocol turned out to be overkill since REED applications expected some amount of loss in the sensor network anyway, and the highly reliable communication within a REED group did not match the low reliability of transmissions once the data left the group. For this reason, REED reacts with a delay to node failure inside a group. We use the active neighbor list in the TinyOS multi-hop routing layer to detect failures. When a group member disappears from a node's neighbor table, that node immediately commences the group dissolving process described above.

4. Optimizations

In this section, we extend the basic join algorithm described in the previous section with several optimizations that decrease the overall communication requirements of our algorithms and that allow us to apply in-network filters for large tables that exceed the storage of a group of nodes.

4.1. Bloom Filters

To allow nodes to avoid transmitting sensor data tuples that will not join with any entries in the join table, we can disseminate to every node in the network a *k*-bit Bloom filter [2], *f,* over the set of values, *J*, appearing in the join column(s) of the filter table. We also program nodes with a hash function, *H,* which maps values of the join attribute *a* into the range 1…*k*. Bits in *f* are set as follows:

$$\forall \text{values } v \text{ in the domain of } a$$

$$f(H(v)) = \begin{matrix} 1 \text{ i.f.f. } v \in J \\ 0 \text{ otherwise} \end{matrix}$$

Thus, if bit *i* of *f* is unset, then no value which *H* maps to *i* is in *J*. However, just because bit *i* is set does not mean that every value which hashes to *i* is included in *J*. We apply Bloom filters as in R*[15]: when a node produces a tuple, *t,* with value *v* in the join column, it computes *H*(*v*) and checks to see if the corresponding entry is set in *f.* If it is not, it knows that this tuple will definitely not join. Otherwise, it must forward this tuple, as it *may* join. Assuming simple, uniform hashing, choosing a larger value of *k* will reduce the probability of a false positive where a sensor tuple is forwarded that ultimately does not join, but will also increase the cost of disseminating the Bloom filter. We can apply bloom filters with the group protocol, to avoid even one transmission of data to group members, or in isolation as a locally-filtered version of the naïve (external) filtration algorithm.

4.2. Partial Filtering

For situations in which there are a very large number of tuples in the join table, we can just disseminate information that allows sensors to identify tuples that definitely *do not* join with any filters. Suppose we know that there are no filters on attribute *a* in the range $a_1 \ldots a_2$. If we transmit this range in the network, then a sensor tuple, *t*, with value *t.a* outside $a_1 \ldots a_2$ is guaranteed to not join with any filters and need not be transmitted; if *t* does intersect with the range, we must transmit it to the root to check and see if this tuple joins with any filters. Of course, for a multidimensional join query, there will be many such ranges with empty values, and we will want to send as many of them into the network as the nodes can store.

Thus, the challenge in applying this scheme is to pick the appropriate values of a_1 and a_2 in each range we send into the network so as to maximize the benefit of this approach. If few tuples that are produced by the sensors are outside of this range, we can substantially decrease the number of tuples that nodes must transmit. Of course, the range of values which commonly join may change over time, suggesting that we may want to change the subset of the table stored in the network adaptively, based on the values of sensor tuples we observe being sent out of the network. We discuss one such adaptive algorithm in the next section.

4.2.1. Cache Diffusion

The key idea of our approach is to observe the data that sensor nodes are currently producing. We assume that each node contains two cache tables. The first, the *local value cache*, contains the last *k* tuples that a node *n* produced. The second table (which is organized as a priority queue) holds empty range descriptions (ERDs) of the join. An ERD is a range of values over all join attributes such that no combination of tuple values within the range joins with the filter table; e.g., an ERD for a query filtering by `nodeid` and `temperature` might consist of the range [20-25] on `temperature` and the range [5-7] on `nodeid`; a different ERD might consist of the range [23-30] on `temperature` and [1-3] on `nodeid`. Tuples that are within the range described by an ERD will not join. We define the *size* of an ERD to be the product of the width of the ranges in the ERD. We define a *maximal ERD* for a non-joining tuple to be the ERD of the largest size that the tuple overlaps. We currently compute the maximal ERD via exhaustive search at the basestation.

The cache diffusion algorithm then works as follows. Every time the root basestation receives a tuple that does not join, it sends the maximal ERD which that tuple intersects one hop in the direction that the tuple came from. This node then checks its local value cache for tuples that are contained within this ERD. If one is found, this value and any other values that overlap with the ERD are removed from the local value cache, and the ERD is added to the ERD cache table with priority 1. If no match is found, then the ERD is also placed in the ERD cache table, but we mark it with priority 0. Priorities are used to determine which ERDs to evict first, as described below.

Upon receiving a tuple from a child for forwarding, a node first checks the ERD cache to see if the tuple falls within any of its stored ERDs. If so, the node filters the tuple and sends the matching ERD to the child. Further, if node *x* overhears node *y* sending a tuple to node *z* (where node *z* is not the basestation), it also checks its ERD table for matching ERDs and, if, it finds one, forwards it to node *y*. The ERD cache is managed using an LRU policy, except that low-priority ERDs are evicted first. Here "last-use" indicates the last time an ERD successfully filtered a tuple.

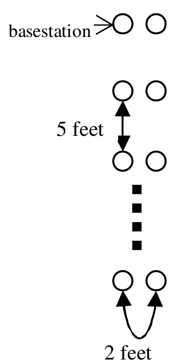


Figure 4: Mote Topology

Thus, for a node *x* of depth *d*, it takes *d* tuples that fall within an ERD to be produced before the ERD reaches node *x*. Note that these *d* tuple productions do not have to be consecutive as long as the matching ERD that diffuses to node *x* does not get removed from the ERD cache of its ancestor nodes on its way. Further, note that despite the fact that it takes *d* tuples before node *x* receives the ERD, these tuples get forwarded fewer and fewer times while the ERD gets closer and closer to *x*. In total, d + (d-1) + (d-2) … + 1 additional transmissions are needed before an ERD reaches node *x*. The advantage of this approach over directly transmitting the ERD to the node that produced the non-joining tuple is two fold: first, we do not have to remember the path each tuple took through the network; second, we do not have to transmit every ERD *d* hops – only those which filter several tuples in a row.

Once an ERD (or set of ERDs) arrive at node *x*, then as long as node *x* produces data within the ERD, no transmissions are needed. Thus, for joins with low selectivity on sensor attributes of high locality, we expect this cache diffusion algorithm to perform well, even for very large tables.

5. Experiments and Results

We have completed an initial REED implementation for TinyOS. Our code runs successfully on both real motes and in the TinyOS TOSSIM [13] simulator. We use the same code base for both TOSSIM and the motes, simply compiling the code for a different target. Most of the experimental results in this section are reported from the TinyOS TOSSIM simulator, which allows us to control the size and shape of the network topology and measure scaling of our algorithms beyond the small number of physical nodes we have available. We demonstrate that our simulation results closely match real world performance by comparing them to numbers from a simple five-mote topology.

We are running TOSSIM with the packet level radio model that is currently available in the `beta/TOSSIM-packet` directory of the TinyOS CVS repository. This simulator is much faster (approximately 1000x) than the standard TOSSIM radio model but still simulates collisions, acknowledgments, and link asymmetry. The primary difference between this simulator and the standard TOSSIM radio model is that the standard model simulates the modulation of every bit of every packet over the radio, modeling collisions by ANDing together the bits written by different senders. The packet simulator writes a whole packet at a time, delivering packets to receivers after an appropriate transmission delay (as long as other senders within radio range do not collide by writing packets during this delay period). For the measurements reported here, our algorithms perform similarly (albeit much more slowly) when using the standard bit-level simulator.

For the experiments below, we simulate a 20x2 grid of motes where there are 5 feet between each of the 20 rows and 2 feet between the 2 columns. The top-left node is the basestation. This is shown in Figure 4. With these measurements, a data transmission can reach a node of distance 1 away (horizontally, vertically, or diagonally in Figure 4) with more than 90% probability, of distance 2 away with more than 50% probability, and rarely at further distances. However the collision radius is much larger: nodes transmitting data with distance <=5 away from a particular node can collide with that node's transmission. For the distributed (group) join experiments, we set the group quality threshold described above to 75%, which yield groups almost always to consist of nodes all less than 10 feet away from each other. We chose this topology because it allows us to easily experiment with large depths so that nodes towards the leaves of the network can still reliably send data to the basestation while not requiring the TinyOS link layer to perform retransmissions during data forwarding. We have also experimented with grid topologies (such as 5x5) to confirm that the algorithm still performs correctly under different topologies (as long as the network is dense enough so that groups can form).

Our first set of experiments will examine the distributed (REED) join algorithm. We evaluate this algorithm along two metrics: power savings and result accuracy. We use number of transmissions as an approximation of power savings as justified in Section 1.2. We compare those results to a naïve algorithm that simply transmits all readings to the basestation and performs the join outside the network. We measure accuracy to determine whether our protocols have a significant effect on loss rates over an out-of-network join. We also show how combining this algorithm with a predication filter (such as Bloom) can further improve our metrics. In these experiments, we simulate a Bloom filter that accurately discards non-joining tuples with a fixed probability. We analyze the dimensions that contribute to this probability in later experiments.

For experiments of the distributed join, we use a join query like the industrial process control Query (1) described in Section 2 above, except that we use the same schedule at every node (so our query does not include a join on `nodeid`). Our schedule table has 62 entries, representing 62 different times and temperature constraints. On our mica2 motes with 4K of RAM, each mote has sufficient storage for about 16 tuples – the remainder of the RAM is consumed by TinyDB and forwarding buffers in the networking stack. We have also experimented with several other types of join queries and found similar results: irrespective of the query, join-predicate selectivity and average node depth have the largest effect on query execution cost for the distributed join algorithm.

For all graphs showing results for the distributed join algorithm,, we show power utilization and result accuracy at steady state, after groups have formed and nodes are performing the join in-network. We do not include table distribution costs in the total transmission numbers. We choose to do this for two reasons.

First, efficient data dissemination in sensor networks is an active, separate area of research [14,23]. Any of these algorithms can be used to disseminate the predicates table to the network. We use the most naïve of dissemination algorithms: flooding the table to the network. For every tuple sent into the network, each node will receive it once and rebroadcast it once. Thus, if there n nodes in the network, and the table contains k filter predicates, then there will be $n \cdot k$ transmissions per table dissemination. However, since multiple tables are disseminated (one per group), our naïve dissemination algorithm requires $n \cdot k \cdot g$ transmissions where g is the number of groups. A simple optimization would be to wait until all groups had been formed and transmit the table just once; doing this is non-trivial as groups may break-up and reform over the course of the algorithm. For the experiments we run, we found that on average 300 transmissions are made per predicate in the table for our 40 node network (since g is on average 7.7). For the 60 predicate table size we experimented with, this added 18K transmissions.

Second, applications of our join algorithm tend to be long running continuous queries. For this reason, we are more interested in how the algorithm performs in the long term, and we expect that these set up costs will be totally amortized over the duration of a query. For example, in 500 epochs (the duration of our experiments below), we already accrue up to 160K transmissions - well above the 18K transmissions needed to disseminate the table.

Our second set of experiments analyzes and compares the Bloom Filter and Cache Diffusion algorithms. Again we use the number of transmissions as the evaluation metric. We observe how the size of the join attribute domain and locality of data are good ways to decide between which algorithm to use.

5.1. Distributed Join Experiments

The following two experiments examine how two independent variables affect the metrics of power savings and accuracy for each join algorithm: *join predicate selectivity* and *average node depth*. For all experiments, data is collected once the system reaches steady state for 500 epochs. The table contains 62 predicates and each node has space for 16; resulting in groups of size 4 being created. Different numbers and combinations of groups form in different trial runs, so each data point is taken by averaging three trial runs. Error bars on graphs display 95% confidence intervals.

5.1.1. Selectivity

For this set of experiments, we varied the selectivity of the join predicate and observed how each join algorithm performed.

We model the benefit of the Bloom filter optimization described in Section 4.1 by inserting a filter that discards non-joining tuples with some probability p. We can directly vary p for the test query via an oracle which can determine whether or not a tuple will join, which is convenient for experimentation purposes. We will show later how in practice, the value of p can be obtained.

Figure 5 shows that for highly selective predicates (low predicate selectivity), both the REED algorithm and the Bloomjoin optimization provide large savings in the amount of data that must be transmitted in the network. The naïve algorithm is unaffected by selectivity because it must send back all of the original data to the basestation before the data is analyzed and joined. The REED algorithm does not have this same requirement: those nodes that are in groups can determine whether a produced tuple will join with the predicates table without having to forward it all the way to the basestation. Thus, the savings of the algorithm is linear in the predicate selectivity. The Bloomjoin algorithm improves these results even more since nodes no longer always have to broadcast a tuple to its group (or to its parent if not in a group) to find out if a tuple will join. In these experiments we filter 50% of the non-joining tuples in the Bloom filter.

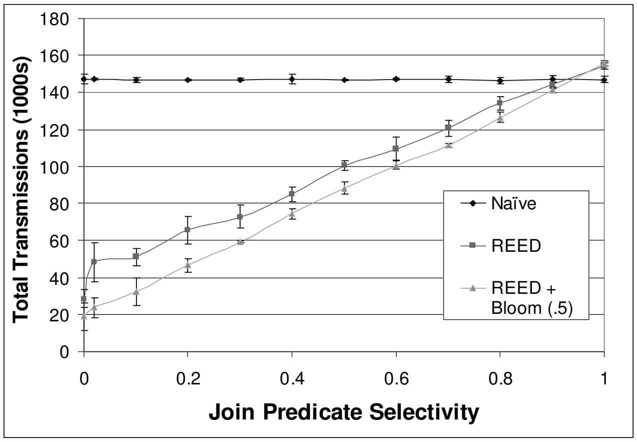


Figure 5: Total Transmissions vs. Selectivity

To better understand the performance of these algorithms, we broke down the type of transmissions into four categories: (1) the transmission of the originally produced tuple (to the node's parent if not in a group; otherwise to the group), (2) the first transmission of any joined tuples, (3) any further transmissions to forward either the original tuple or a joined result up to a parent in a group or to a basestation, and (4) transmissions needed as part of the overhead for the group formation and maintenance algorithms. Figure 6 displays this breakdown for the REED algorithm over varying selectivity. In this figure, the original tuple transmissions remain constant at approximately 20K. This is because every tuple needs to be transmitted at least once in the REED algorithm: if the node is not in a group, the tuple is sent to the node's parent; otherwise it is sent to the group. Once a tuple is sent to a group, no further transmissions are needed if the tuple does not join with any predicate. For the 20-hop node

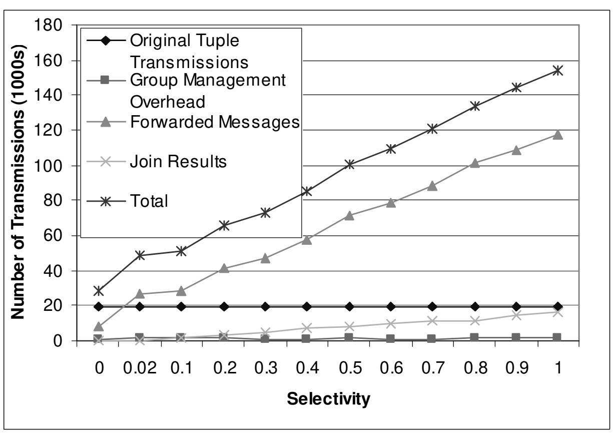


Figure 6: Breakdown of Transmission Types for Distributed Join with Varying Selectivity

topology used in this experiment, the forwarded messages dominate the cost. It is also worth noting that the figure shows that the group management overhead (at steady state) is negligible compared with any of the other types of transmissions.

Since Figure 6 showed that the reason why the REED reduces the number of transmissions is because it reduces the number of forwarded messages that need to be sent, once possible explanation for this could be that the algorithm causes more loss in the network and messages tend to get dropped before reaching the basestation (so they do not have to be forwarded). To affirm that this is not the case, we measured the number of tuples that reach the basestation at varying selectivities and compared each algorithm. These results are shown in Figure 7. As can be seen, all algorithms perform similarly; however the naïve algorithm has slightly less loss at high selectivities and the REED algorithms have slightly less loss at low selectivities. This can be explained as follows: group processing of the join occasionally requires 1-2 extra hops. This is the case when a node *x* that stores a partition of the predicates table that will join with a particular tuple produced by node *y* and *x* is located at the same depth as *y* or 1 node deeper. The former case requires 1 extra hop, the latter 2 extra hops. With each extra hop, there is extra probability that a tuple can be lost. This explains why there is more loss at high join predicate selectivities. However, at low selectivities, this negative impact of REED is outweighed by its reduction in the number of transmissions and thus network contention. Since fewer messages are being sent in the network, there is an increased probability that each message will be transmitted successfully.

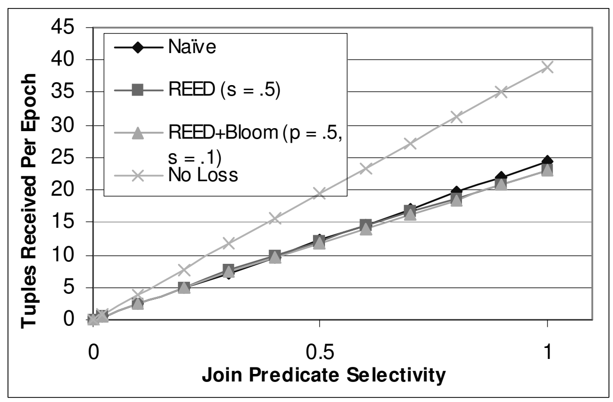


Figure 7: Received Tuples vs. Selectivity for Distributed Join Algorithm

5.1.2. Average Node Depth

For this set of experiments, we fixed the join predicate selectivity at 0.5 and 0.1 and varied the topology of the sensor network (in particular varying average node depth) and observed each how join algorithm performed. We varied node depth by subtracting leaf nodes from the 20x2 topology described earlier. The baseline 20x2 topology has a average depth of 10.26 (each node's parent is fixed to be the node above it in the network except for the top-right node which has the basestation as its parent). We eliminated the bottom 6 nodes to achieve an average depth of 8.76, another 6 nodes to achieve an average depth of 7.26, etc. to achieve depths of 5.76, 4.26, and 2.78; and then the bottom pairs for nodes to achieve average depths of 2.29, 1.80, and 1.33. The number of transmissions for each of the three join algorithms is given in **Figure** 8.

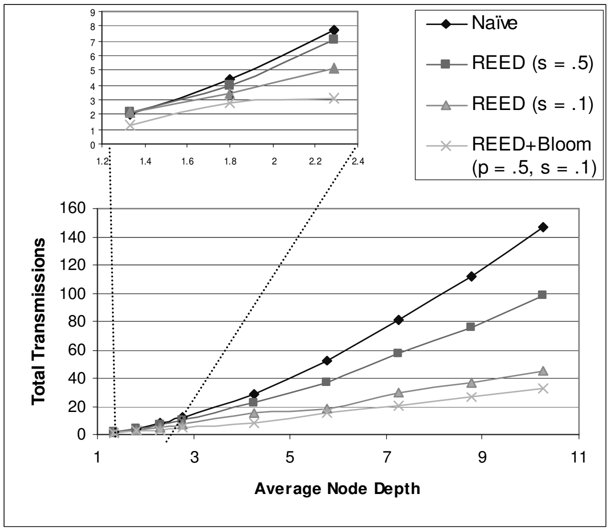


Figure 8: Total Data Transmissions for Varying Average Sensor Node Depths

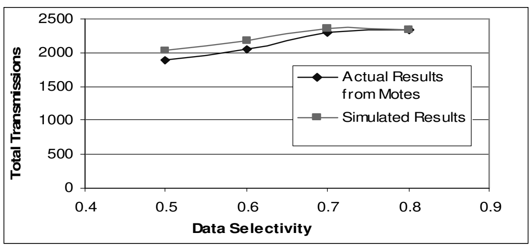


Figure 9: Simulated vs. Real World Results

These results show that the average depth necessary for REED (without using a Bloom filter) to perform better than the naïve algorithm is 1.8. The reason why REED performs worse than the naïve algorithm at low depths is twofold. The less significant reason is the small group formation and maintenance overhead incurred by REED. The more significant reason is that, as explained above, join processing occasionally requires 1-2 extra hops. At large depths, these extra hops get made up for in the saved forwarded transmissions, but for depths less than 2, this is not the case. However, if a Bloom filter is used, REED always outperforms the naïve algorithm.

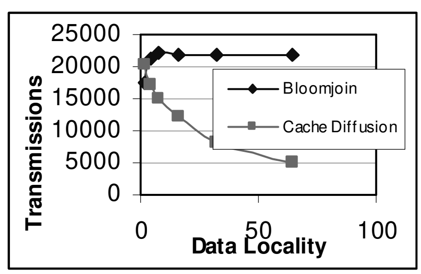
5.2. Real World Results

Although we expected that TOSSIM would be an accurate simulation for TinyOS code, we verified for ourselves that our join algorithm worked on a simple five-node one hop network. We tracked the number of transmissions by passing this number with the result of each join result (for simulations we could more easily track number of transmissions using debug statements). A side effect of this method, however, is that we can not test our algorithm at low selectivities as we can not accurately calculate the number of transmissions of each node at a particular point in time. As a result, we ran our REED algorithm with the Bloom filter optimization varying selectivities above .5. We ran the same experiment in simulation and compared the two results in Figure 9. Simulation and practice perform similarly; however the non-simulated results have slightly decreased number of transmissions due to a slightly higher amount of loss than was modeled in simulation.

5.3. Bloomjoin and Cache Diffusion

Although the Bloomjoin and Cache Diffusion (CD) algorithms described above can help optimize the REED distributed join algorithm, they also can be applied independently where the predicate table is too large to fit on even a group of nodes. Whether applied as an optimization or independently, it is important to decide which algorithm will perform best as usually there is not enough free space on sensor nodes to perform both (besides which each algorithm improves performance if allocated more space). For these experiments, we allocated 90 bytes total space for the data structures needed by each algorithm. For the Bloomjoin algorithm, this allowed a 720 bit Bloom filter to be distributed and for CD, this allowed 9 tuples or ERDs to be cached.

We found that the two most important dimensions that distinguish these algorithms from each other are domain size and data locality and thus we present our results using these dimensions as independent variables. The query used to run these experiments is the outlier detection query presented in section 2.1 except that we add light along with temperature as sensor produced data. In order to vary data locality as an independent variable, we generated data for each node using matlab where sensor readings are produced using a normal distribution with small variance and with a mean that is moved according to a cumulative sum of another normal distribution. Increasing and decreasing the variance of this second normal distribution, causes the mean to move around with decreased or increased jumps, affecting the locality of the data. We define locality in these experiments to be 1/(variance) of the second distribution. Figure 10 shows how total transmissions for a 5 node network of average depth=2 running for 2500 epochs varies with data locality of the Bloomjoin and CD algorithms.



In order to vary attribute domain size we simply mod these values by the desired domain size of each attribute. The size of the domain of the whole tuple is simply the multiplication of the domain sizes of each component attribute. Due to lack of space, we cannot show the graph for the Bloomjoin and CD algorithms with varying selectivity. In short, we found that domain size did not affect CD (however, this could be query dependent), but that Bloomjoin was greatly affected by it. If light was allowed to vary between only 64 values and temperature between 32 (resulting in a domain size of 2048), Bloomjoin approached the naïve algorithm in terms of number of transmissions. But for smaller domains, Bloomjoin performed extremely well. Thus Bloomjoin is preferred over CD when joining over only one attribute, but CD is preferred over Bloomjoin when the domain is larger than one attribute, as long sensor data is produced with reasonable locality.

6. Related Work

Work on distributed query processing for relational databases began as early as the late 1970s. For example, Epstein *et al.* [6] introduced an algorithm for the retrieval of data from a distributed relational database with communication traffic as a cost criteria for which nodes should perform joins. Bernstein *et al.* [1] introduced a semi-join algorithm which reduces the communication overhead of performing distributed joins by taking the intersection of the schemas of the tables to be joined, projecting the resulting schema on one of the tables, sending this smaller version of the table to the node containing the other table and joining at this node, and then sending this result back to the node containing the original table and joining again. This semi-join technique is an interesting possible optimization, though our Bloom-filter approach subsumes and likely outperforms it, for the same reasons as described in R* [15].

Determining how to horizontally partition a join table amongst a set of servers is classic problem in database systems. The Gamma [5] and R* [11] systems both studied this problem in detail, analyzing a range of alternative techniques for allocating sets of tuples to servers, though both sought to minimize total query execution time rather than communication or energy consumption.

TinyDB [16,17,18] and Cougar [28] both present a range of distributed query processing techniques for the sensor networks. However, these papers do not describe a distributed join algorithm for sensor networks.

There are a large number non-relational query systems that have been developed for sensor networks, many of which include some notion of correlating readings from different sensors. Such correlation operations resemble joins, though their semantics are typically less well defined, either because they do not impose a particular data model [9], or because they are probabilistic in nature [4] and thus fundamentally imprecise.

The work that comes closest to REED is the work from Bonfils and Bonnet [3], which proposes a scheme for join-operator placement within sensor networks. Their work, however, focuses on joins pairs of sensors, rather than joins between external tables and all sensors. They do not address the join-partitioning problem that we focus on.

7. Conclusion

REED extends the TinyDB query processor with facilities for efficiently executing complex, multi-predicate filtration queries inside of a sensor network. Our join-based algorithms are capable of running in very limited amounts of RAM, can distribute the storage burden over groups of nodes, and are tolerant to dropped packets and node failures. REED is thus suitable for a wide range of event-detection applications that traditional sensor network database and data collection systems cannot be used to implement. Moving forward, because REED incorporates a general purpose join processor, we see it as the core piece of an integrated query processing framework, in which sensor networks are tightly integrated into more traditional databases, and users are presented with a seamless query interface. Beyond filtrations, power-efficient joins that allow the combination of tables of values outside the sensornet to be joined with streaming data generated by sensors will be extremely valuable.

Bibliography

[1] Philip A. Bernstein, Dah-Ming W. Chiu, Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, 28(1):25-40, 1981.

[2] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422-426, 1970.

[3] Boris Jan Bonfils and Philippe Bonnet. Adaptive and Decentralized Operator Placement for In-Network Query Processing. In *IPSN*, 2003.

[4] M. Chu, H. Haussecker, and F. Zhao. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. In *International Journal of High Performance Computing*, 2002.

[5] D. J. Dewitt and S. Ghandeharizadeh and D. A. Schneider and A. Bricker and H. -I. Hsiao and R. Rasmussen. The Gamma Database Machine Project. In *IEEE TKDE*, 2(1):44-62, 1990. answers.

[6] R. Epstein, M.R. Stonebraker, and E. Wong, Distributed Query Processing in a Relational Database System. In *Proceedings of ACM SIGMOD*, 1978, pp 169-180.

[7] Mick Flanigan, Personal Communication. August, 2003.

[8] Hausman, Mary. Temperature Control Gets Smart.*Chemical Processing Magazine*, Aug., 2002.

[9] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of MobiCOM*, Boston, MA, August 2000.

[10] Wes Iverson. Heading off Breakdowns. *Automation World*, October, 2003. http://www.automationworld.com/articles/Features/280.htm

[11] Mark Lepedus. Intel Harnesses Wireless Sensors For Chip-Equipment Care. *TechWeb*, October 29th, 2003. http://www.techweb.com/wire/26802594

[12] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer and David Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of NSDI*, San Francisco, CA.

[13] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of ACM SenSys*, 2003.

[14] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *Proceedings of NSDI*, 2004.

[15] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proceedings of VLDB*, 1986.

[16] 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.

[17] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.

[18] Samuel Madden, Robert Szewczyk, Michael Franklin, and David Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Proceedings of WMCSA*, 2002.

[19] David Maier, Jeffrey D. Ullman and Moshe Y. Vardi. On the foundations of the universal relation model. In *ACM Transactions on Database Systems (TODS)*, 9(2):283-308, 1984.

[20] Joseph Polastre. Design and implementation ofwireless sensor networks for habitat monitoring. *Master's thesis, UC Berkeley*, 2003.

[21] G. Pottie and W. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51 – 58, May 2000.

[22]Rockwell Automation. Pharmaceutical Manufacturing Optimization. White Paper, 2002. http://domino.automation.rockwell.com/applications/gs/region/gtswebst.nsf/files/pmo.pdf/$file/pmo.pdf.

[23] Stanislav Rost, Hari Balakrishnan. Lobcast: Reliable Dissemination in Wireless Sensor Networks. In submission, 2004.

[24] P. Selinger et al. Access Path Selection in a Relational Database Management System. In *Proeedings of ACM SIGMOD, 1979.*

[25] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner-Allen, and Matt Welsh. Simulating the Power Consumption of Large-Scale Sensor Network Applications. To appear in *Proceedings of ACM SenSys*, 2004.

[26] M. Stonebraker and G. Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, 34(10), 1991.

[27] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of ACM SenSys*, 2003.

[28] Yong Yao and Johannes Gehrke. Query processing in sensor networks. In Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR), 2003.