Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks

Samuel Madden, Robert Szewczyk, Michael J. Franklin and David Culler
University of California, Berkeley
{madden, szewczyk, franklin, culler}@cs.berkeley.edu

Abstract

We show how the database community's notion of a generic query interface for data aggregation can be applied to ad-hoc networks of sensor devices. As has been noted in the sensor network literature, aggregation is important as a data-reduction tool; networking approaches, however, have focused on application specific solutions, whereas our in-network aggregation approach is driven by a general purpose, SQL-style interface that can execute queries over any type of sensor data while providing opportunities for significant optimization. We present a variety of techniques to improve the reliability and performance of our solution. We also show how grouped aggregates can be efficiently computed and offer a comparison to related systems and database projects.

1 Introduction

Recent advances in computing technology have led to the production of a new class of computing device: the wireless, battery powered, smart sensor. Unlike traditional sensors deployed throughout buildings, labs, and equipment everywhere, these new sensors are not merely passive devices that modulate a voltage based on some environmental parameter: they are full fledged computers, capable of filtering, sharing, and combining sensor readings.

At UC Berkeley, researchers have developed small sensor devices called *motes*, and an operating system, called TinyOS, that is especially suited to running on them. Motes are equipped with a radio, a processor, and a suite of sensors. TinyOS makes it possible to deploy *ad-hoc* networks of sensors that can locate each other and route data without any a priori knowledge of network topology.

As various groups around the country have begun to to deploy large networks of sensors, a need has arisen for tools to collect and query data from these networks. Of particular interest are aggregates – operations which summarize current sensor values in some or all of a sensor network. For example, given a dense network of a thousand sensors querying temperature, users want to know temperature patterns in relatively large regions encompassing tens of sensors – individual sensor readings are of little value.

Sensor networks are limited in external bandwidth, i.e. how much data they can deliver to an outside system. In many cases the externally available bandwidth is a small fraction of the aggregate internal bandwidth. Thus computing aggregates in-network is also attractive from a network performance and longevity standpoint: extracting all data over all time from all sensors will consume large amounts of time and power as each individual sensor's data is independently routed through the network. Previous studies have shown [6] that aggregation dramatically reduces the amount of data routed through the network, increasing throughput and extending the life of battery powered sensor networks as less load is placed on power-hungry radios.

Previous networking research [10, 9, 6] approached aggregation as an application specific technique that can be used to reduce the amount of data that must be sent over a network. In the database community, however, aggregates are viewed as a generic technique that can be applied to any data, irrespective of the application. In this work, we adopt this database intuition: our system provides a generic aggregation interface that allows aggregate queries to be posed over networks of sensors. There are two benefits of this approach over the traditional network solution: first, by defining the language that users use to express aggregates, we can significantly optimize their computation. Second, because the same aggregation language can be applied to all data types, the burden on programmers is substantially less: they can issue declarative, SQL style queries rather than implementing custom networking protocols to extract the data they need from the network.

In this paper, we discuss the challenges associated with implementing the five basic database aggregates (`COUNT`, `MIN`, `MAX`, `SUM`, and `AVERAGE`) with grouping in ad-hoc networks of sensors. We show how our this generic approach leads to a significant power savings. Further, we show that sensor network queries can be structured as time series of aggregates, and how such queries adapt to the changing network struc-

ture. We have implemented early versions of these techniques and are in the process of experimentally validating them.

We begin with the relevant background in the TinyOS platform on which our aggregation algorithms are deployed, along with a brief summary of aggregation in database systems. Following that, we present our algorithms for aggregation, related and future work, and conclusions.

2 Background

In this section, we first discuss the relevant design aspects of the TinyOS operating system and mote architecture. For more complete treatment of these topics, refer to [8, 16, 7]. We then summarize aggregation in database systems and discuss how those techniques provide a useful and well defined framework for computing aggregates in sensor networks.

2.1 Motes

A photograph of the current generation of motes is shown in Figure 1. These devices are equipped with a 4Mhz Atmel microprocessor with 512 bytes of RAM and 8 kB of code space, a 917 MHz RFM radio running at 10 kb/s, and 32kB of EEPROM. An expansion slot accommodates a variety of sensor boards by exposing a number of analog input lines as well as popular chip-to-chip serial busses. Current sensor options include: light, temperature, magnetic field, acceleration (and vibration), sound, and power.

The radio hardware uses a single channel, and uses on-off keying. It provides an unbuffered bit-level interface; the rest of the communication stack (up to message layer) is implemented by TinyOS software. Like all single-channel radios, it offers only a half duplex channel. Currently, the default TinyOS implementation uses a CSMA-like media access protocol with random backoff scheme. Message delivery is unreliable by default, though applications can build up an acknowledgement layer. Often, a message acknowledgement can be obtained for free (see below in Section 2.3).

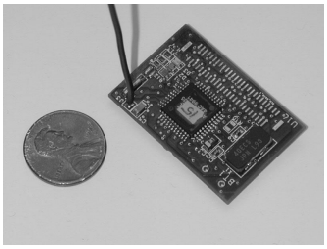

Figure 1: *A TinyOS Sensor Mote*

Power is supplied via a free-hanging AA battery pack or a coin-cell attached through the expansion slot.

The effective lifetime of the sensor is determined by its power supply. In turn, the power consumption of each sensor node is dominated by the cost of transmitting and receiving messages; including processor cost, sending a single bit of data requires about 4000 nJ of energy, whereas a single instruction on a 5mW processor running at 4Mhz consumes only 5 nJ (see [8]). Thus, in terms of power consumption, transmitting a single bit of data is equivalent to 800 instructions. This energy tradeoff between communication and computation implies that many applications will benefit by processing the data inside the network rather than simply transmitting the sensor readings.

2.2 TinyOS

TinyOS provides a number of services to greatly simplify writing programs that capture and process sensor data and transmit messages over the radio. The reader is referred to [7] for details of the operating system. For the purposes of this paper, TinyOS should be thought of as an API which can send and receive messages and read from sensors. The next section goes into some detail on the messaging and networking aspects of TinyOS and wireless sensors, as those are most relevant to the topic of aggregation.

2.3 Ad-hoc Sensor Networks

In this section, we discuss how data is routed in our ad-hoc aggregation network. To understand the solution, two properties of radio communication need to be emphasized. First, radio is a broadcast medium, such that any sensor within hearing distance can hear any message, irrespective of whether or not it is the intended recipient. Second, radio links are typically symmetric: if sensor a can hear sensor b, we assume sensor b can also hear sensor a. Note that this may not be a valid assumption in some cases: if a's signal strength is higher, because its batteries are fresher or its signal is more amplified, b will be able to hear a but not reply to it.

Messages in the current generation of TinyOS are a fixed size preprogrammed into sensors – by default, 30 byte messages are used. Each message type has a *message id* that distinguishes it from other types of messages. Sensor programmers write message id specific handlers that are invoked by TinyOS when a message of the appropriate id is heard on the radio. Each sensor has a unique *sensor id* that distinguishes it from other sensors. All messages specify their recipient (or broadcast, meaning all available recipients), allowing sensors to ignore messages not intended for them,

although non-broadcast messages must still be received by all sensors within range – unintended recipients simply drop messages not addressed to them.

Given this brief primer on wireless sensor communication, we now show how sensors route data. The technique we adopt is to build a routing tree.[1] We appoint one sensor to be the *root*. The root is the point from which the routing tree will be built, and upon which aggregated data will converge. Thus, the root is typically the sensor that interfaces the querying user to the rest of the network. The root broadcasts a message asking sensors to organize into a routing tree; in that message it specifies its own id and its *level*, or distance from the root, which is zero. Any sensor that hears this message assigns its own level to be the level in the message plus one, if its current level is not already less than or equal to the level in the message. It also chooses the sender of the message as its *parent*, through which it will route messages to the root. Each of these sensors then rebroadcasts the routing message, inserting their own ids and levels. The routing message floods down the tree in this fashion, with each node rebroadcasting the message until all nodes have been assigned a level and a parent. Nodes that hear multiple parents choose one arbitrarily, although we will discuss approaches in below (Section 3.3) where multiple parents can be used to improve the quality of aggregates. These routing messages are periodically broadcasted from the root, so that the process of topology discovery goes on continuously. This constant topology maintenance makes it relatively easy to adapt to network changes caused by mobility of certain nodes, or to the addition or deletion of sensors: each sensor simply looks at the history of received routing messages, and chooses the "best" parent, while ensuring that no routing cycles are created with that decision.

This application is makes it possible to efficiently route data towards the root. When a sensor wishes to send a message to the root, it sends the message to its parent, which in turn forwards the message on to its parent, and so on, eventually reaching the root. This application doesn't address point-to-point routing; however, for our purpose, flooding aggregation request and routing replies up the tree to the root is sufficient. We'll see in the Section 3 how, as data is routed towards the root, it can be combined with data from other sensors to efficiently combine routing and aggregation. First, however, we describe how aggregates are expressed in database systems.

[1]Note that this is one of many possible techniques that could be used; the reader is referred to [16, 10, 9, 11, 1] for more information. Our observations about aggregation of sensor data do not depend on a particular routing tree algorithm; rather, they exploit the fact that such a structure can be built and maintained efficiently in the presence of a changing network topology.

2.4 Aggregation in Database Systems

Aggregation in SQL-based database systems is defined by an *aggregate function* and a *grouping predicate*. The aggregate function specifies how a set of values should be combined to compute an aggregate; the standard set of SQL aggregate functions is COUNT, MIN, MAX, AVERAGE, and SUM. These compute the obvious functions; for example, the SQL statement:

```
SELECT AVERAGE(temp) FROM sensors
```

computes the average temperature from some table `sensors`, which represents a set of sensor readings that have been read into the system. Similarly, the `COUNT` function counts the number of items in a set, the `MIN` and `MAX` functions compute minimal and maximal values, and `SUM` calculates the total of all values. Additionally, most database systems allow *user-defined functions* (UDFs) that specify more complex aggregates than the five listed above.

Grouping is also a standard feature of database systems. Rather than merely computing a single aggregate value over the entire set of data values, a grouping predicate partitions the values into groups based on some attribute. For example, the query:

```
SELECT TRUNC(temp/10), AVERAGE(light)
    FROM sensors
    GROUP BY TRUNC(temp/10)
    HAVING AVERAGE(light) > 50
```

partitions sensor readings into groups according to their temperature reading and computes the average light reading within each group. The `HAVING` clause excludes groups whose average light readings are less than or equal to 50.

In the rest of this paper, we discuss the challenges associated with implementing the five basic aggregates with grouping in ad-hoc networks of TinyOS sensors. We start by considering a single aggregate being computed at a time, and then argue that often users are interested in viewing aggregates as sequences of changing values over time. We discuss the implication of this assertion in Section 6. Throughout this work, we will assume the user is stationed at a desktop-class PC with ample memory. Despite the simple appearances of this architecture, there are a number of difficulties presented by the limited capabilities of the sensors, as we will see in the next section.

Throughout the following analyses, the focus is on reducing total number of messages required to compute an aggregate; this is because, as discussed above, message transmission costs typically dominate energy consumption of sensors, especially when performing only simple computation such as the five standard database aggregates.

3 Generic Aggregation Techniques

A naive implementation of sensor network aggregation would be to use a centralized, *server-based* approach where all sensor readings are sent to the host PC, which then computes the aggregates. However, as was shown in [6], a distributed, *in-network* approach where aggregates are partially or fully computed by the sensors themselves as readings are routed through the network towards the host-PC can be considerably more efficient. In this section, we focus on the in-network approach, because, if properly implemented, it has the potential to be both lower latency and lower power than the server based approach.

To illustrate the potential advantages of the in-network approach, consider the simple example of computing an aggregate over a group of sensors arranged as shown in Figure 2. Dotted lines represent connections between sensors, solid lines represent the routing tree imposed on top of this graph (as discussed above) to allow sensors to propagate data to the root along a single path. In the centralized approach, each sensor value must be routed to the root of the network; for a node at depth n, this requires n-1 messages to be transmitted per sensor. The sensors in Figure 2(a) have been labeled with their distance from the root; summing these numbers gives a total of sixteen messages required to route all aggregation information to the root. Contrast this with the sensors in Figure 2(b): sensors with no children simply transmit their readings to their parents. Intermediate nodes (with children) combine their own readings with the readings of their children via the aggregation function f and propagate the partial aggregate, along with any extra data required to update the aggregate, up the tree.

Notice that the amount of data transmitted in this solution depends on the aggregate. Consider the AVERAGE function: at each intermediate node n, the sum and count of all children's sensor readings are needed to compute the average of sensor readings of the subtree rooted at n. We assume that, in the case of AVERAGE, both pieces of information will easily fit into a single 30 byte message. Thus, a total of five messages need to be sent for the average function. In the case of the other standard SQL aggregates, no additional state is required: COUNT, MIN, MAX, and SUM can be computed by a parent node given sensor or partial aggregate values at all of the child nodes.

In this work we focus on a class of aggregation predicates that is particularly well suited to the in-network regime. Such aggregates can be expressed as an aggregate function f over the sets a and b such that:

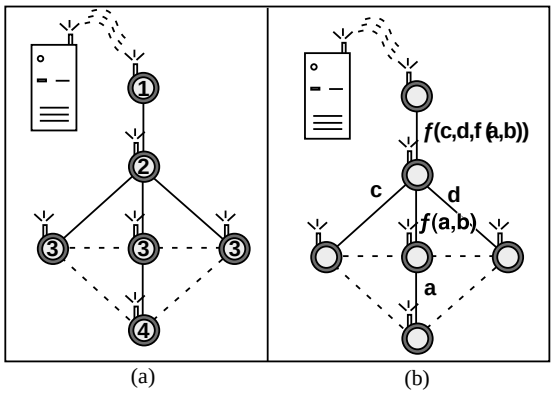

Figure 2: *Server-based (a) versus In-network (b) aggregation* In (a), each node is labelled with the number of messages required to get data to the host PC: a total of 16 messages are required. In (b), only one message is sent along each edge as aggregation is performed by the sensors themselves.

$$f(a \cup b) = g(f(a), f(b)) \tag{1}$$

We focused on this class of aggregates for two reasons: first the basic SQL aggregates all exhibit the above property, and second because the problems with this substructure map easily onto the underlying network. We expect to tackle more generalized aggregation predicates, such as median, in a future work.

For the reasons described above, in network aggregation is always a superior choice. Given the in-network regime, we next give a brief description of how aggregation queries are pushed down into a sensor network and how results are returned to the user. For the purposes of this discussion, we assume aggregate queries do not specify groups; queries with groups are discussed in Section 4. Then, in the remainder of this section, we examine other problems that can arise in ad-hoc sensor environments and sketch possible solutions.

3.1 Injecting a Query

Computing an aggregate consists of two phases: a *propagation* phase, in which aggregate queries are pushed down into sensor networks, and an *aggregation* phase, where the aggregate values are propagated up from children to parents. The most basic approach to propagation works just like the network discovery algorithm described above, except that leaf nodes (nodes with no children) must discover that they are leaves and propagate singular aggregates up to their parents. Thus, when a sensor p receives an aggregate a, either from

another sensor or from the user, it transmits a and begins listening. If p has any children, it will hear those children retransmit a to their children, and will know it is not a leaf. If, after some time interval t, p has heard no children, it concludes it is a leaf and transmits its current sensor value up the routing tree. If p has children, it assumes they will all report within time t, and so after time t it computes the value of a applied to its own value and the values of its children and forwards this partial aggregate to its parent.

Notice that choosing too short a duration for t can lead to missed reports from children, and also that the proper value of t varies depending on the depth of the routing tree. We will discuss a possible solution to this problem in the next section; for now, assume that t is set to be long enough that the message has time to propagate down to all leaves below p and back, or, numerically:

$$t = 2 \times (d_p - d_{tree}) \times (t_{xmit} + t_{process}) \quad (2)$$

where t_{xmit} is the time to send a message and ($t_{process}$ is the time to process an aggregation request. Empirical studies suggest that $(t_{xmit} + t_{process})$ needs to be 200 or more milliseconds. The time to transmit a 30-byte message on a 10kbit radio is about 50 ms: each nibble must be DC balanced (have the same number of ones and zeros), costing extra bits, and simple forward error correction is used, meaning that for every byte, 18 bits must be transmitted; 18 * 30 bytes / 10000 bits / sec = 50ms. Computation time is small, but significantly more than 50 ms must be allocated per hop to account for differences in clock synchronization between sensors and random collision detection back-off that sensors engage in. Thus, for a deep sensor network, computing a single aggregate can take several seconds. In the next section, we will see that the unreliable communication inherent to sensor networks, coupled with such long computation times make this simple simple in-network approach undesirable.

3.2 Streaming Aggregates

Sensor networks are inherently unreliable: individual radio transmission can fail, nodes can move, and so on. Thus, it is very hard to guarantee that a significant portion of a sensor network was not detached during a particular aggregate computation. Consider, for example, what happens when a p broadcasts a and its only child, c, somehow misses the message (perhaps because it was garbled during transmission.) p will never hear c rebroadcast, and will assume that it has no children and that it should forward only its own sensor value. The entire network below p is thus excluded from the aggregation computation, and the end result is probably incorrect. Indeed, when any subtree of the graph can fail in this way, it is impossible to give any guarantees about the accuracy of the result.

One solution to this problem is to double-check aggregates by computing them multiple times. The simplest way to do this would be to request the aggregate be computed multiple times at the root of the network; by observing the common-case value of the aggregate, the client could make a reasonable guess as to its true value. The problem with this technique is that it requires retransmitting the aggregate request down the network multiple times, at a significant message overhead, and the user must wait for the entire aggregation interval for each additional result.

Instead, we propose using a *pipelined aggregate*, which works as follows. In this scheme, aggregates are propagated into the network as described above. However, in the pipelined approach, time is divided into intervals of duration i. During each interval, every sensor that has heard the request to aggregate transmits a partial aggregate by applying a to its local reading and the values its children reported during the previous interval. Thus, after the first interval, the root hears from sensors one radio-hop away. After the second, it hears aggregates of sensors one and two hops away, and so on. In order to include sensors which missed the request to begin aggregation, a sensor that hears another sensor reporting its aggregate value can assume it too should begin reporting its aggregate value.

In addition to tending to include nodes that would have been excluded from a single pass aggregation, the pipelined solution has a number of interesting properties: first, after aggregates have propagated up from leaves, a new aggregate arrives every i seconds. Note that the value of i can be quite small, about the time it takes for a single sensor to produce and transmit a sensor reading, versus the value of t in the simple multi-round solution proposed above, which is roughly $depth_{tree}$ times larger. Second, the total time for an aggregation request to propagate down to the leaves and back to the root is roughly t, but the user begins to see approximations of the aggregate after the first interval has elapsed; in very deep networks, this additional feedback may be a useful approximation while waiting for the true value to propagate out and back. These two properties provide users with a stream of aggregate values that changes as sensor readings and the underlying network change. As discussed above, such continuous results are often more useful than a single, isolated aggregate, as they allow users to understand how the network is behaving over time. Figure 3 illustrates a simple aggregate running in a pipelined fashion over a small sensor network.

The most significant drawback of this approach is that a

number of additional messages are transmitted to extract the first aggregate over all sensors. For the example shown in Figure 3, 22 messages are sent, since each aggregating node is transmits once per time interval. The comparable non-pipelined aggregate requires only 10 messages – one down and one back along each edge. Note, however, that, in this example, after this initial 12 message overhead, each additional aggregate arrives at a cost of only 5 messages and at a rate of one update per time interval. Still, it is useful to consider optimizations to reduce this overhead. One option is that sensors could transmit only when the value of the aggregate computed over their subtree changes, and parents could assume their children's aggregate values are unchanged unless they hear differently. In such a scheme, far fewer messages will be sent, but some of the ability to incorporate nodes that failed to hear the initial request to aggregate will also be lost, as there will be fewer aggregate reports for those nodes to snoop on. We reserve the analysis of the tradeoffs of these approaches for future work.

We believe a hybrid pipeline scheme will significantly improve the robustness of aggregates by tending to incorporate nodes that lose initial aggregation requests. Pipelining also improves throughput, which can be important when a a single aggregate requires seconds to compute. With this pipelined model in mind, we now consider a number of other optimizations that can improve the efficiency of aggregates in sensor networks.

3.3 Taking Advantage of A Shared Channel

In our discussion of aggregation algorithms up to this point, we have largely ignored the fact that sensors communicate over a shared radio channel. The fact that every message is effectively broadcast to all other sensors within range enables a number of optimizations that can significantly reduce the number of messages transmitted and increase the accuracy of aggregates in the face of transmission failures.

We saw an example of how a shared channel can be used to increase message efficiency when a sensor that misses an initial request to begin aggregation: it can initiate aggregation even after missing the start request by *snooping* on the network traffic of nearby sensors. When it sees another sensor reporting an aggregate, it can assume it too should be aggregating.

This technique is not only beneficial for improving the number of sensors participating in any aggregate; it also substantially reduces the number of messages that must be sent when using the pipelined aggregation scheme. Because nodes assume they should begin aggregation any time they hear an aggregate reported, a sensor does not need to explicitly tell its children to begin aggregation. It can simply report its value to its parents, which its children will also hear. The children will assume they missed the start request and initiate aggregation locally. For the simple example in Figure 3, none of the messages associated with black arrows actually need to be sent. This reduces the total messages required to compute the first full aggregate of the network from 22 to 17, for a total savings of 23%.

Of course, for later rounds in the aggregation, when no messages are sent from parents to children, this savings is no longer available. Snooping can, however, be used to reduce the number of messages sent for certain classes of aggregates. Consider computing a maximum over a group of sensors; if a sensor hears a peer reporting a maximum value greater than its local maximum, it can elect to not send its own value and be assured of not affecting the value of the final aggregate. We will discuss variants of this technique in more detail in Section 3.4 below.

In addition to reducing the number of messages that must be sent, the inherently broadcast nature of radio also offers communications redundancy which improve reliability. Consider a sensor with two parents: instead of sending its aggregate value to just one parent, it can send it to both parents. It is easy for a node to discover that it has multiple parents, since it can simply build a list of nodes it has heard that are one step closer to the root. Of course, for aggregates other than MIN and MAX, sending to multiple parents results has the undesirable effect of causing the node to be counted multiple times. The solution to this is to send part of the aggregate to one parent and the rest to the other. Consider a `COUNT`; a sensor with $c-1$ children and two parents can send a `COUNT` of $c/2$ to both parents instead of a count of c to a single parent. A simple statistical analysis reveals the advantage of doing this: assume that a message is transmitted with probability p, and that losses are independent, so that if a message m from sensor s is lost in transition to parent P_1, it is no more likely to lost in transit to P_2. [2] First, consider the case where s sends c to a single parent; the expected value of the transmitted count is $p \times c$ (0 with probability $(p-1)$ and c with probability p), and the variance is $c^2 \times p \times (1-p)$, since these are standard Bernoulli trials with a probability of success p multiplied by a constant c. For the case where s sends $c/2$ to both parents, linearity of expectation tells us the expected value is the sum of the expected value through each parent, or $2 \times p \times c/2$. Similarly, we can sum the variances through each parent to

[2] Although failure independence is not always a valid assumption, it will occur when a hidden-node garbles communication to P_1 but not P_2, or when one parent is forwarding a message and another is not.

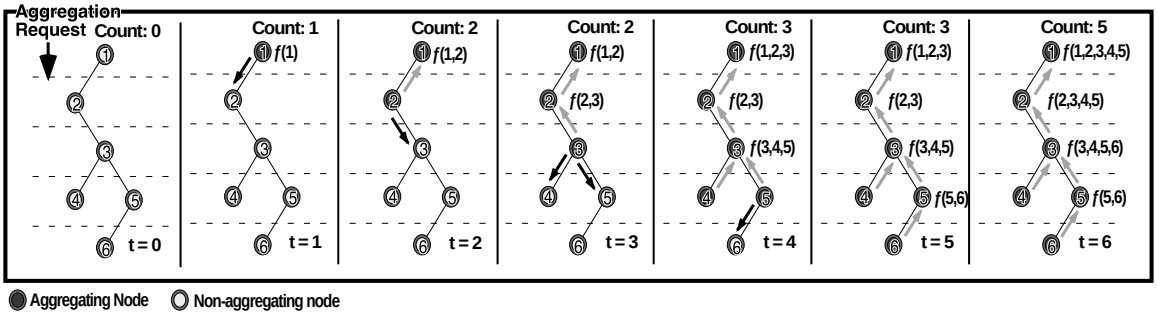

Figure 3: *Pipelined computation of aggregates*

get:

var = $2 \times (c/2)^2 \times p \times (1-p) = c^2/2 \times p \times (1-p)$

Thus, the variance of the multiple parent `COUNT` is much less, although its expected value is the same. This is because it is much less likely (assuming independence) for the message to both parents to be lost, and a single loss will less dramatically effect the computed value. Note that the probability that no data is lost is actually lower with multiple parents (p^2 versus p), suggesting that there this may not always be a useful technique. However, since losses are almost assured of happening occasionally when aggregating, we believe users will prefer that their aggregates be closer to the correct answer than exactly right more often.

This technique applies equally well for `SUM` and `AVERAGE` aggregates or for any aggregate which is a linear combination of a number of values. For rank-based aggregates, like mode and median, this technique cannot be applied.

We now present our final technique for increasing the efficiency of aggregates: rephrasing aggregates as hypotheses to dramatically reduce the number of sensors required to respond to any aggregate.

3.4 Hypothesis Testing

Although the above techniques offer significant gains in terms of number of messages transmitted and robustness with respect to naive approaches, these techniques still require input from every node in a network to compute an aggregate. In this section, we observe that we only need to hear from a particular sensor if that sensor's sensor value will affect the end value of the aggregate. For some aggregates, this can significantly reduce the number of nodes that need to report.

We presented a simple example of hypothesis testing above: when computing a `MAX` or `MIN`, a sensor can snoop on the values its peers report and omit its own value if it knows it cannot affect the final value of the aggregate. This technique can be generalized to an approach we call *hypothesis testing*. If a node is presented with a guess as to the proper value of an aggregate, either by snooping on another sensor's aggregate value or by explicitly being presented with a hypothesis by the user or root of the network, it can decide locally whether contributing its reading and the readings of its children will affect the value of the aggregate.

For `MAX`, `MIN` and other *top-n*[3] aggregates, this technique is directly applicable. There are a number of ways it can be applied – the snooping approach is one. As another example, the root of the network seeking a `MIN` sensor value might compute the value of the aggregate over the top k levels of the network (using the pipelined approach described above), and then abort the aggregate and issue a new request asking for only those sensor values less than the minimum observed in the top k levels. In this approach, leaf nodes will be required to send no message if their value is greater than the minimum observed over the top k levels (intermediate nodes must forward the request to aggregate, so they must still send messages.) If we assume sensor values are independent and randomly distributed (a big assumption!), then a particular leaf note must transmit with probability $1/2^k$, which is quite low for even small values of k. Since, in a balanced tree, half the nodes are in the bottommost level, this can reduce the total number of messages that must be sent by almost a factor of two.

For other aggregates that accumulate a total, such as `SUM` and `COUNT` this technique will never be applicable. For the a third class of statistical aggregates, such as `AVERAGE` or variance, this technique can reduce the number of messages, although not as drastically. To obtain any benefit with such aggregates, the user must define an error bound that he is willing to tolerate over the value of the aggregate. Given this error bound, the same approach as for top-n aggregates can be applied. Consider the case of an average: any sensor that is within the error bound of the approximate answer need not

answer – its parent can assume its value is the same as the approximate answer and count it accordingly (this scheme requires parents to know how many children they have.) The total computed average will not be off from the actual average by more than the error bound, and leaf sensors with values close to the average will not be required to report. Obviously, the value of this scheme varies greatly on the distribution of sensor values. If values are uniformly distributed, the fraction of leaves that need not report will approximate the size of the error bound. If values are normally distributed, a much larger percentage of leaves will not report. Thus, the value of this scheme depends on the expected distribution of values and the tolerance of the user to inaccurate error bounds.

In summary, we proposed using in-network aggregation to compute aggregates. By pipelining aggregates, we were able to increase throughput and smooth over intermittent losses inherent in radio communication. We improved on this basic approach with several other techniques: snooping over the radio to reduce message load and improve accuracy of aggregates, and hypothesis testing to invert problems and further reduce the number of messages sent. In the next section, we augment the algorithms presented in this section to support grouping.

4 Grouping

Recall that grouping computes aggregates over partitions of sensor readings. The basic technique for grouping is to push down a set of predicates that specify group membership, ask sensors to choose the group they belong to, and then, as answers flow back, update the aggregate values in the appropriate groups.

Group predicates are appended to requests to begin aggregation. If sending all predicates requires more storage than will fit into a single message, multiple messages are sent. Each group predicate specifies a group id, a sensor attribute (e.g. light, temperature), and a range of sensor values that define membership in the group. Groups are assumed to be disjoint and defined over the same attribute, which is typically not the attribute being aggregated. Because the number of groups can be large enough such that information about all groups does not fit into the RAM of any one sensor, sensors pick the group they belong to as messages defining group predicates flow past and discard information about other groups.

Messages containing sensed values are propagated just as in the pipelined approach described above. When a sensor is a leaf, it simply tags the sensor value with its group number. When a sensor receives a message from a child, it checks the group number. If the child is in the same group as the sensor, it combines the two values just as above. If it is in a different group, it stores the value of the child's group along with its own value for forwarding in the next interval. If another child message arrives with a value in either group, the sensor updates the appropriate aggregate. During the next interval, the sensor will send out the value of all groups it collected information about during the previous interval, combining information about multiple groups into a single message as long as the message size permits. Figure 4 shows an example of computing a query grouped by temperature that selects average light readings. In this snapshot, data is assumed to have filled the pipeline, such that results from the bottom of the tree have reached the root.

Recall that SQL queries also contain a `HAVING` clause that constrains the set of groups in the final query result by applying a filtration predicate to each group's aggregate value. We sometimes pass this predicate into the network along with partitions. The predicate is only sent into the network if it can potentially be used to reduce the number of messages that must be sent: for, example, if the predicate is of the form `MAX(attr)` $>$ `x`, then information about groups with `MAX(attr)` $\leq$ `x` need not be transmitted up the tree, and so the predicate is sent down into the network. However, other `HAVING` predicates, such as those filtering `AVERAGE` aggregates, or of the form `MAX(attr)` $<$ `x`, cannot be applied in the network because they can only be evaluated when the final group-aggregate value is known.

Because the number of groups can exceed available storage on any one sensor, a way to evict groups is needed. Once an eviction victim is selected, it is forwarded to the sensor's parent, which may choose to hold on to the group or continue to forward it up the tree. Because groups can be evicted, the user workstation at the top of the network may be called upon to combine partial groups to form an accurate aggregate value. Evicting partially computed groups is known as *partial preaggregation*, as described in the database literature [12].

There are a number of possible policies for choosing which group to evict. We believe that policies which incur a significant storage overhead (more than a few bits per group) are undesirable because they will reduce the number of groups that can be stored and increase the number of messages that must be sent. Evicting groups with low membership is likely a good policy, as those are the groups that are least likely to be combined with other sensor readings and so are the groups that benefit the least from in-network aggregation.

Evicting groups forces information about the current time interval into higher level nodes in the tree. Since in the

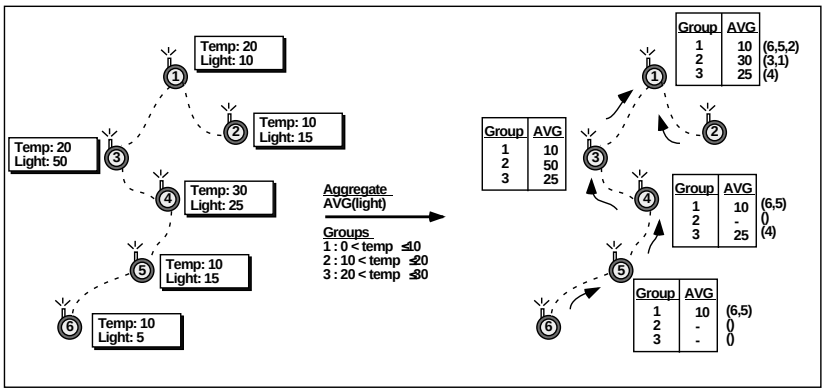

Figure 4: *A sensor network (left) with an in-network, grouped aggregate applied to it (right). Parenthesized numbers represent the sensors that contributed to the average; they are included for the reader's benefit – the sensors do not actually track this information.*

standard pipelined scheme presented above, aggregates are computed over values from the previous time interval, this presents an inconsistency. We believe, however, that this will not dramatically effect aggregates; verifying this remains an area of future work.

Thus, we have shown how to partition sensor readings into a number of groups and properly compute aggregates over those groups, even when the amount of group information exceeds available storage in any one sensor.

5 Related Work

In this section, we discuss related work from both the database and sensor networking communities. Although the networking community has begun to explore issues of data collection within sensor networks, there is no other work that we are aware of that proposes a generic, query-based scheme for extracting data from sensor networks.

With respect to aggregation, the semantics used here are largely a part of the SQL standard [2]. The partial preaggregation techniques [12] used to enable group eviction were proposed as a technique to deal with very large numbers of groups to improve the efficiency of hash joins and other bucket-based database operators.

The Cougar project at Cornell [14] discusses queries over sensor networks, as does our own work on Fjords [13], although the former only considers moving selection operators onto sensors and neither presents a specific, power-sensitive algorithms for use in sensor networks.

Literature on active-networks [15] first identified the idea that the network could simultaneous route and transform data, rather than simply serving as an end-to-end data conduit. Within the sensor network community, work on networks that perform data analysis has been largely confined to the USC/ISI and UCLA communities. Their work on directed diffusion [10] discusses techniques for moving specific pieces of information from one place in a network to another, and proposes aggregation-like operations that nodes may perform as data flows through them. [6] proposes a scheme for imposing names onto related groups of sensors in a network, in much the way that our scheme partitions sensor networks into groups. [9] discusses networking protocols for routing data to improve the extent to which data can be combined as it flows up a sensor network – it provides low level techniques for building routing trees that could be useful in computing database style aggregates.

Networking protocols for routing data in wireless networks are very popular within the literature [11, 1, 4, 5], however, none of them address higher level issues of data processing, merely techniques for data routing. Our tree based routing approach is clearly inferior to these approaches for peer to peer routing, but works well for the aggregation scenarios we are focusing on.

The TinyOS group at UC Berkeley has published a number of papers describing the design of motes [8], the design of TinyOS [7], and the implementation of the networking protocols used to construct ad-hoc sensor networks [16]. None of this work directly addresses issues of data collection or aggregation, but is important as the platform on which our solution operates.

6 Future Work

There are a number of areas of future work. Clearly, experimental and mathematical validation of many of the techniques presented in this paper is needed. As researchers at UC Berkeley, we are currently working with the sensor testbed built by the TinyOS group to empirically verify the algorithms we have presented. Beyond verification, however, there are several significant challenges that have been glossed over in this work.

We have not explored the tradeoffs between fully pipelined communication and techniques such as sending values only when sensor readings change. There are a number of options in this space, each of which has different message costs and robustness properties.

We do not yet fully understand how our approach behaves when sensors move. Although the routing tree construction algorithm allows moving nodes to reattach, and the pipelined aggregation scheme can eventually adjust to moved nodes or subtrees, it is important to formally characterize how movements and disconnections affect the value of aggregates.

Finally, we have not explored the problem of computing multiple simultaneous aggregates over a single sensor network. It should be possible for sensors to accommodate multiple queries (just as they handle multiple groups) up to some small number of queries. There may be an eviction option, as with grouping, but there may also be a point at which the in-network approach is so slow that the server-based approach again becomes viable. The implementation issues associated with simultaneous aggregates must be explored before these in-network approaches can be implemented in a database system that supports concurrent queries.

7 Conclusion

We have demonstrated techniques for applying database style aggregates with groups to sensor readings flowing through ad-hoc sensor networks. By applying generic aggregation operations in the tradition of database systems, our approach offers the ability to query arbitrary data in a sensor network without custom-building applications. By pipelining the flow of data through the sensor network, we are able to robustly compute aggregates while providing rapid and continuous updates of their value to the user. Finally, by snooping on messages in the shared channel and applying techniques for hypothesis testing, we are able to substantially improve the performance of our basic approach.

This work marks a first step towards a generic, in-network approach for collecting and computing over sensor data. SQL, as it has developed over many years, has proven to work work well in the context of database systems. We believe a similar language, when properly applied to sensor networks, will offer similar benefits as SQL: ease of use, expressiveness, and a standard on which research and industry can build.

References

[1] W. Adjue-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM SOSP*, December 1999.

[2] ANSI. *SQL Standard*, 1992. X3.135-1992.

[3] M. J. Carey and D. Kossman. Processing top n and bottom n queries. *Data Engineering Bulletin*, 20(3):12–19, 1197.

[4] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad-hoc wireless networks. In *ACM MobiCom*, July 2001.

[5] T. Goff, N. Abu-Ghazaleh, D. Phatak, and R. Kahvecioglu. Preemptive routing in ad hoc networks. In *ACM MobiCom*, July 2001.

[6] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *SOSP*, October 2001.

[7] J. Hill. A software architecture to support network sensors. Master's thesis, UC Berkeley, 2000.

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

[9] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. Submitted for Publication, ICDCS-22, November 2001.

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

[11] J. Kulik, W. Rabiner, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th Annual IEEE/Mobicom Conference*, Seattle, WA, 1999.

[12] P.-A. Larson. Data reduction by partial preaggregation. In *ICDE*, 2002. (to appear).

[13] S. Madden and M. J. Franklin. Fjording the stream: An architechture for queries over streaming sensor data. In *ICDE*, 2002. (to appear).

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

[15] D. Tennenhouse. Active networks. In *OSDI*, October 1996.

[16] A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. In *ACM Mobicom*, July 2001.