The Sensor Network as a Database

Ramesh Govindan, ICSI and USC
Joseph M. Hellerstein, UC Berkeley
Wei Hong, Intel Berkeley Labs
Samuel Madden, UC Berkeley
Michael Franklin, UC Berkeley
Scott Shenker, ICSI

Abstract

Wireless sensor networks are an emerging area of research interest with a number of compelling potential applications. By architecting sensor networks as virtual databases, we can provide a well-understood non-procedural programming interface suitable to data management, allowing the community to realize sensornet applications rapidly. We argue here that in order to achieve an energy-efficient and useful implementation, query processing operators should be implemented within the sensor network, and that approximate query results will play a key role. We observe that in-network implementations of database operators require novel data-centric routing mechanisms, as well as a reconsideration of traditional network and database interface layering.

1 Introduction

Wireless sensor networks have received significant recent attention in both the networking and operating systems communities [23, 25]. These networks are predicated on advances in miniaturization that will make it possible to design small form-factor devices with significant on-board computation, wireless communication, and sensing capabilities.

Anticipating the development of such devices, recent work has also begun exploring potential applications of sensor networks for instrumenting and monitoring various environments. Examples of such applications include: monitoring in-building energy usage for planning energy conservation [1]; military and civilian surveillance [12]; fine-grain monitoring of natural habitats with a view to understanding ecosystem dynamics [6]; data gathering in instrumented learning environments for children [35]; and measuring variations in local salinity levels in riparian environments [36]. The variety of these applications clearly conveys the enormous potential impact of wireless sensor networks.

Several characteristics of these sensor networks make them different from today's wired and wireless networks. In most of the applications described above, the sensor networks will operate unattended and untethered. The devices will likely be battery powered, and *energy-efficiency* will be a primary design consideration. In particular, the energy cost of communication is expected to be significantly higher than the cost of local computation [30, 23]. This implies that data collection techniques that transport large volumes of data across significant distances can seriously degrade network lifetime. Furthermore, in these networks, the devices interact with the physical world, and generate data about locally observed events. As a consequence, *robustness* to node and communications failures, as well as to noisy sensor readings, will be an important design consideration. Finally, the expected mode of usage of sensor networks will be that users will *query the network* and thereby obtain one or more responses. For example, a user might ask of an in-building network: "What is the average late afternoon temperature in the west wing?".

These characteristics of sensor networks impact their structure in interesting ways. Sensor networks are best designed in a *data-centric* manner: the low-level communication primitives in these networks are designed in terms of *named* data rather than the node identifiers used in traditional networked communication [25]. This architecture follows from the fact that in these networks, individual nodes do not necessarily have an identity of interest; rather, the data that they generate is of interest independent of source node identity.

The characteristics of sensor networks described above also give rise to another intriguing architectural view of sensor networks; the sensor network as a *database*. This view is complementary to the view of the network as having a data-centric routing system, in that routing is a bottom-up *mechanism*, whereas a database view is a top-down data modeling and application development *interface*. Recall that nodes in a sensor network generate named data against which one or more users issue queries. This is quite similar to the traditional view of relational databases, in which disk blocks (whose individual identities are irrelevant to the application) contain data records against which queries are issued.

We have said that the database view is a modeling and interface issue, but of course the database community has developed a great deal of algorithmics and system architecture to span this level of indirection in an efficient and dynamic fashion. These include query optimization and indexing techniques, and a canonical set of primitive query operators that can be composed to form complex queries. A core challenge in relational database systems has been to ensure that query-based applications remain robust in the face of changing data distributions, physical storage characteristics, and query workloads. The infrastructure for data access in sensor networks will also need to provide a similar form of robustness, in addition to being robust to node failures and noisy sensor readings. Indeed, data generation and routing in the sensor networks seem quite analogous to data storage and query processing in databases. As such, it seems quite natural to view the sensor network as a database, and attempt to leverage similar benefits in this new context. Of course the properties of the infrastructure and the data in a sensor network are quite different than in a database system.

In this paper, we explore challenges in realizing this architectural view of the sensor network as a database. Specifically, this view allows users to issue database queries to one or more (perhaps designated) nodes within the sensor network. These queries can be "one-shot" relational queries with a fixed answer set, or ongoing *continuous* queries that produce an unbounded stream of results. The compelling advantage of a query-specification interface is that it defines an *application-independent way of programming data collection from the sensor network*.[1]

We suggest that a sensor network database (or a sensornet database, for short) should be architected on two important ideas.

The first is *in-network implementations* of primitive database query operators such as grouping, aggregation, and joins. By "in-network" we mean group communication and routing protocols which, together with possible processing at intermediate nodes, im-

[1]Clearly, not *all* applications will use the query-specification interface. For example, our mechanisms will not be applicable for actuation. They may be applicable for event notification, although we do not discuss such uses in this paper.

plement each operator in an application-independent way. We argue that such implementations will require *novel routing mechanisms* which take into account network resource constraints as well as the order in which database operators are processed.

Second, unlike the strict semantics associated with traditional data models and query languages, we argue for relaxing the semantics of database queries to allow *approximate results*. This relaxation enables energy-efficient implementations even given the expected high level of network dynamics (such as packet loss, node failures *etc.*). A sensor network is a proxy for a continuous real-world phenomenon, and by nature *samples* that phenomenon discretely at some rate, with some degree of error. Hence it is not only convenient but indeed more accurate to present approximate semantics, and expose a spectrum of tradeoffs between concise and precise communication. As we discuss below, several pieces of prior work on online sampling and approximation in the database community are applicable in this context.

2 Background

In this section, we briefly review the state of sensor network subsystems, and provide the necessary background in database systems. In subsequent sections, we discuss challenges in designing a sensornet database.

2.1 Sensor Network Subsystems

Prototypes of sensor devices are starting to appear on the horizon. One class of devices is exemplified by the *mote* [23]. Motes contain an 8-bit processor, a low baud-rate radio, several megabytes of memory, and MEMS sensors for detecting temperature, ambient light, and vibration. A class of larger devices [30] contains PC-class processors, spread-spectrum radios, infrared dipoles, acoustic geophones, and electret microphones. In both these classes of devices, the radios represent a key design constraint: communication using these radios requires significantly more energy than computation.

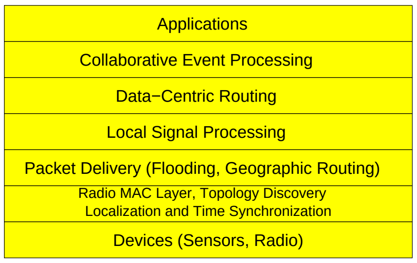
Applications
Collaborative Event Processing
Data−Centric Routing
Local Signal Processing
Packet Delivery (Flooding, Geographic Routing)
Radio MAC Layer, Topology Discovery
Localization and Time Synchronization
Devices (Sensors, Radio)

Figure 1: Sensor Network Software Subsystems

While the eventual form of sensor network hardware can be reasonably extrapolated from the above classes the form of sensor network software subsystems is less clear. Figure 1 depicts an emerging modularization of sensor network software. Although drawn as a stack, we do not mean to suggest that this is the most appropriate modularization of sensor network software, or indeed that sensor network software can even be "layered". As we illustrate later, it might be necessary to collapse layers or selectively break abstraction boundaries for efficiency or robustness reasons.

Term	Description
Data Model	Framework for data representation and data semantics
Tuple	A data record, usually consisting of several attributes
Source	Sensor node that generates a tuple
Table	A logical collection of similarly typed tuples. It can represent an infinite stream of tuples.
Operator	A function that takes one or more tables as input and outputs a table
Query	A composition (specifically, a tree) of operators

Figure 2: Glossary of Terms

Figure 1 allows us to place ongoing work in developing sensor network subsystems in context. Working our way up the layers in Figure 1, examples of such related research include: an efficient operating system for sensor nodes [23]; low-level network self-configuration systems [7], including systems for localizing nodes [31, 32, 33], and performing time synchronization [11]; a data-centric routing system [25], and possibly collaborative signal processing systems [39] that can, for example, track moving targets.

2.2 Data Models

A prerequisite for discussing the database view of sensor networks is a *data model*, which is a framework for describing data representation and semantics. The most popular data model in use today is the relational model. In the context of a sensor network, this model is best described as follows. In our descriptions, we assume, for ease of exposition, a sensor network where nodes do not move. Each sensor produces one or more **tuples**. The node that generates the tuple is termed the **source**. For example, a temperature sensor might produce a tuple of the form <`nodeLocation`, `timestamp`, `temperature`>. Similarly, at a node that uses acoustic and vibration signal patterns to detect vehicles, signal processing software might generate a tuple of the form <`nodeLocation`, `timestamp`, `vehicle-type`, `detectionConfidence`>. A collection of similarly-typed tuples from a group of sensors forms a "snapshot". In database terminology, this snapshot constitutes a relational **table** which is horizontally partitioned across the sensors in the group. For example, the tuples generated by a collection of temperature sensors form a temperature table.

Relational tables are typically stored on disks in conventional relational database systems. It is important to note that the tables we discuss in the sensor network context are all *virtual* tables. They are relational views of the data generated by a sensor network; the database concepts we discuss apply to virtual tables as well as they do to conventional databases. Accesses to these virtual tables are automatically translated into corresponding data-collecting operations on each relevant sensor nodes, *e.g.*,, `GetTemperature`, `GetLightIntensity`, *etc.* Virtual tables can be *unbounded*, representing, for example, streams of data.

The goal of the sensornet database design should be to preserve *location transparency*. A sensor network application writer should be able to get live temperature information by issuing database queries against a temperature table without any knowledge of actual topology of the network. Managing the location and routing of these tuples is left to the infrastructure. This greatly eases the task of the application developers, and – more importantly – ensures that application code continues to function when data locations and/or routing schemes change.

Note that each tuple can have a key that identifies where the tuple was generated. In our example above, this is the `nodeLocation`, but more generally it can be any unique node identifier. This identifier is used to correlate multiple readings from a single node, for example, using the *join* operation (described later). It might also be useful for network monitoring. However, such an identifier does not compromise our requirement for location transparency as long as applications do apply physical interpretations to the identifiers (*e.g.*, assuming that a node is up by checking if there exist tuples with the specified identifier).

One final note on data modeling. Especially outside the database community, the term "relational database" often evokes notions of strong guarantees on storage consistency and availability. This paper does not discuss challenges in the *storage* of tuples or in designing mechanisms to guarantee availability of tuples (*e.g.*, availability of a tuple generated within a time window for sequence-centric operations on that window).

2.3 Database Operators

The basic thesis of our paper is that core relational database operators like aggregation, grouping, selection and join form appropriate *building blocks* for application development on sensor networks. The following paragraphs describe some of these traditional database operators in a sensor network context. We stick to an SQL-style *multiset* (or *bag*) semantics, in which duplicate tuples are not eliminated by default; this is typically the desired semantics for aggregation-centric applications. Note that the algebra of these relational operators is *closed*, meaning that the result of any of these operators is a relational table, which can serve as input to further operators.

Aggregation is an operation that is fundamental to a data-rich, large-scale, yet energy-constrained, sensor network. By aggregation we mean the summarization of a column (or arithmetic expression over multiple columns) into a single numerical value. "The average temperature on the third floor" is an example of an aggregate defined on a temperature table consisting of tuples from sensors in an in-building sensor network. Most commercial databases provide common aggregation operators such as SUM, COUNT, AVERAGE, MIN, MAX, and STDDEV (standard deviation). We anticipate aggregation queries will be very prevalent in sensor networks.

In traditional databases, the *join* operator is used to correlate data from multiple tables. A join can be defined as a selection over the cross-product of a pair of tables; a join of tables R and S is denoted by $R \bowtie S$. One simple implementation of a join is to generate all pairs of tuples, and then extract those which satisfy the selection predicate. However it is quite common to implement joins in a more efficient fashion that does not form all pairs. A common join predicate is an equality match across columns of the two tables (an *equi-join*). For example, consider a temperature table with tuples of the form <`nodeLocation`, `timestamp`, `temperature`>. Also, assume that some sensor nodes with temperature sensors also have light sensors, each of which produces tuples of the form <`nodeLocation`, `timestamp`, `lightlevel`>. An equi-join of these two tables on the `nodeLocation` column would produce a table with tuples of the form: <`nodeLocation`, `timestamp`, `lightlevel`, `temperature`>, where tuples are only defined for nodes that have both temperature and light sensors.

There are several other relational operators like grouping (partitioning a table according to a predicate), selection (extracting tuples based on a predicate), projection (extracting one or more columns from a table), union, difference, duplicate elimination, and distinct aggregates that we do not discuss for brevity.

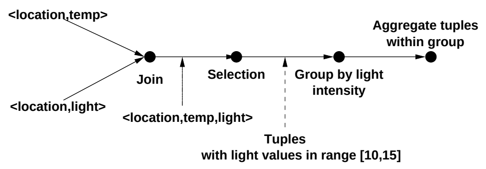

Figure 3: Complex Query Example

Finally, it is natural to write complex queries that compose multiple operators. Consider an in-building network that contains sensor nodes with light and temperature sensors. An example of a complex query defined on this network is: find the average temperature in different iso-light-intensity regions within a range of light intensities $[10, 15]$. Figure 3 describes how such a query could be accomplished using the operators described above.

3 Sensornet Database: Overview

We have said that a sensornet database allows any user to issue a query to the sensor network as if it is a database system (perhaps from any node attached to the network) and obtain a response to that query.

There are at least two obvious realizations of a sensornet database. The first is a centralized (data warehouse) realization, where all data from each node in the network is sent to a designated node within the network attached to which is a large database. Users can then simply query that database. This can be impractical in the sensor network context since it requires significant communication and that requires energy. The other alternative, a distributed database, can be energy efficient when the query rate is less than the rate at which data is generated. However, traditional distributed databases are unsuitable for large-scale sensor networks because distributed database design has traditionally assumed well-maintained global metadata about data distribution and network topology.

We believe that a fundamentally different architecture is necessary to realize a sensornet database. This architecture rests on two features. The first feature is *in-network* implementation of database operators. When a user (or an application) poses a query to the network, that query is disseminated across the network (either to all the nodes using simple flooding, or to a geographically constrained set of nodes using variants of well-known geographic routing algorithms [27]). In response to the query, each node generates tuples that match the query, and transmits matching tuples towards the origin of the query. As the tuples are routed through the network, intermediate nodes might apply one or more database operators. Other work has shown that *in-network processing* of sensor data is fundamental to achieving energy-efficient communication in sensor networks [18].

A second feature is that, unlike traditional databases, the sensornet database will provide *approximate results*. In sensor networks, the availability of data might be reduced as a result of message loss caused by vagaries in wireless communication, or by node failure.

We argue—given the energy constraints on sensor network design, and given the time-varying nature of sensor data—that classical approaches to data recovery (*e.g.*, replication, reliable transmission protocols) may be too heavy-weight. Rather, by relaxing the semantics of database operators to allow approximate results, we argue that it might be possible to use data recovery methods better tuned to operator semantics.

Related to the notion of approximate answers is another feature, called *streamed results*, that we think will be important for sensor networks, particularly those used for continuously monitoring the environment. This feature will enable partial query results to be displayed in real-time to a human user, and will allow users of the sensor network *dynamically refine* their queries. This capability, called *online aggregation*, has been proposed in the database literature for large on-line decision support systems [22, 19, 20]. In the sensor network context, such a capability could allow users to drill down to more specific queries (*e.g.*, an outlier in the query of Figure 3 might indicate an unusual source of heat).

In the next section, we illustrate the research challenges involved in realizing these features by considering the implementation of some database operators. Before we do so, however, we contrast our overall approach with three other closely related pieces of work.

The notion of representing data generated by sensors as tuples is superficially similar to the notion of data naming discussed in the context of data-centric sensor network routing [18] and wide-area information discovery [3]. However, modeling the data generated by a sensor network as a relational database allows us to present a well understood application interface, and to leverage standardized data manipulation techniques defined for databases.

The COUGAR project at Cornell University [5] is one of the first attempts to model a sensor network as a database system. It focuses on the interaction between the sequence data produced in sensor networks and stored data in backend relational databases. It extends both the SEQ [34] sequence data model and the relational data model by introducing new operators between sequence data and relational data. COUGAR is implemented as an extension to Cornell's PREDATOR Object-Relational database system. It models sensors as columns with Abstract Data Types (ADTs). Users invoke sensors functions by calling ADT functions on sensor columns in queries. COUGAR does not currently focus on exploiting the special characteristics of sensor networks, nor does it explore the interaction between query processing and networking. Rather, from an architectural point of view it simply layers (novel) database functionality on top of a traditional network model. We intend to leverage the data modeling work that has been done by the COUGAR group, and instead focus on the architectural and algorithmic issues of efficiently integrating query processing logic into sensor networking subsystems.

Finally, Srivastava *et al.* [35] point out the need for a data management middleware for sensor network data analysis and mining, in the context of a particular application (the "smart" kindergarten). Our paper takes this a step further and identifies specific challenges in realizing one aspect of this middleware, a relational database.

4 Operators

What we have discussed so far lays the groundwork for discussing the research issues in designing sensornet databases. We now begin to highlight some of these research issues by considering the implementation of two database operators: *joins* and *aggregation*.

4.1 Join

In the sensornet database, the complexity of the join can vary with the particular query. The simplest example of a join, one which joins the temperature and light tables by node location (see example in Section 2.3) can be accomplished locally. That is, each individual node can perform the join on the temperature and light tuples that it generates before transmitting the joined tuple to the query originator.

More generally, however, the tuples generated at different nodes might be joined at a single node. Consider, for example, a multimodal vehicle identification sensor network in which some nodes have vibration sensors, others acoustic sensors, and yet others imagers (nodes may also have more than one sensor). A vibration sensor generates a tuple of the form <`eventType`, `vibrationAmplitude`, `confidenceLevel`, `targetLocation`>. Other sensors produce similar tuples, perhaps differently typed. To correlate events from different sensors, one might wish to perform an equijoin on the `eventType` column.

The database literature has studied several generic join implementation methods, such as *nested-loop*, *merge-sort*, and *hashjoin* [15]. Some of these methods only apply to equi-joins (Section 2.3). However, these conventional methods have one drawback that makes them unsuitable for sensor network environments. These methods are *blocking*. For example, the hashjoin algorithms commonly used in database systems [9] cannot produce any tuples until one of the tables is fully scanned. Blocking is infeasible in sensor networks because the tables can contain unbounded streams of data, and the amount of memory available on each sensor node is limited *relative to the potential sizes of sensornet database tables*.

Database join algorithms can be modified with two basic techniques to become applicable to the sensor net context: pipelining and partitioning. We discuss these next.

Pipelining

A suite of non-blocking pipelined join methods have been developed in recent years. One example is symmetric hash-join [38]. It builds and maintains two hash tables (keyed by the column(s) used for the join), one for each input table. When an input tuple arrives, it looks up matching tuples from the other input's hash table and outputs any matching results, then inserts itself into its own hash table. It is "symmetric" because the action for each tuple from either table is the same.

A generalization of symmetric hash-joins is the family of join methods called *ripple joins* [17]. These join methods statistically *sample* the two tables to be joined, in order to produce a stream of joined tuples. The relative rates at which the two tables are sampled adapt to match the variance produced by the data in each. When used together with an aggregation operator, they provide online aggregation.

These pipelined join methods, because they are non-blocking, will be the methods most directly applicable to sensor nets. In addition to the memory constraint imposed by the sensor nodes, there are two reasons for preferring pipelined joins. We have argued that on-line query refinement will be important in sensor nets used for monitoring. Pipelined joins, because they provide streamed partial answers can enable query refinement. Furthermore, pipelining schemes like ripple joins form a low energy approach to obtain *approximate* answers and can be used together with sampling (as we discuss in Section 4.2).

Partitioning and Interactions with Routing

How will a join query be realized on our sensornet database? Will there be a single node in the network that will perform the pipelined join? Especially for a geographically constrained join (*e.g.,* select records from a certain region of the network and then perform a join), it might be possible to elect a node (*e.g.,* the one closest to the centroid of the region) to perform the join. More generally, this approach points to a technique used in parallel database systems called *partitioning*. Here, tuples are partitioned based on their join-column values (either by range or by hashing), and redistributed on the fly across multiple nodes; the work of joining the individual partitions is done in parallel by each of the nodes [10]. This idea is applicable in the sensornet database as well; partitions can be defined by value, geographically, or by sensor type, and a node (or nodes) can be designated to perform the join for the partition. The goal here is both to leverage parallelism, and to exploit aggregate RAM space across multiple nodes, since joins can be memory-intensive, and the sensor nodes may be memory-constrained.

An obvious research challenge is to develop techniques for partitioning joins in an energy-efficient way. Geographic partitioning can be energy efficient (in that tuples are locally joined). However, if the join is not along a node location column, geographic partitioning may not be applicable. A possible approach is to partition a column by hash values, using data-centric storage schemes like CAN, Chord, Pastry and Tapestry. Like traditional parallel hash joins, these schemes partition a key space across a collection of nodes. However, traditional databases did this on a fully-connected cluster interconnect, whereas data-centric storage schemes are scalable over arbitrary topologies in the wide area. Data-centric storage can require transporting tuples over significant distances; however if the key space is partitioned across nodes within a loosely bounded geographical region, the overhead of this technique might be acceptable.

While these approaches are somewhat simplified, they point out an important issue: the realization of relational operators in a sensornet database can be posed as a *routing* problem. In these examples, we have discussed relatively simple instances of this problem. In Section 5, we discuss situations where the routing subsystem is invoked at a finer granularity (*e.g.,* to route individual tuples differently).

4.2 Aggregation

The next class of operators we study are the aggregation operators. The mechanics of computing aggregates is *conceptually* simple; a query is flooded throughout the network or to a specified geographic region, and the responses are routed on the reverse path trees, possibly being aggregated across several nodes. However, achieving this in the context of sensor networks proves to have surprising richness.

A Taxonomy of Aggregates

Aggregation on multiple nodes is not new – is has been extensively explored in the parallel database literature, particularly in the context of parallel systems with user-extensible interfaces for defining ad hoc aggregation functions [26]. A taxonomy of aggregates was developed [16] to categorize the different classes of aggregates in terms of their partitioning across multiple nodes in a cluster. We use and extend that taxonomy here to organize the various types of aggregation functions in a sensornet database.

In any multi-node aggregation scheme, the basic idea is for each node to aggregate some subset of the data, and then pass some partially-aggregated state to other nodes; this partial state is itself aggregated more from multiple sources. Eventually, some node receives a set of partially-aggregated state that covers the entire table, and computes a final answer. In sensor networks, one key performance goal is to extend the lifetime of the network by minimizing communications. Hence, aggregation functions can be usefully categorized by the sizes of the partial state records that get passed around.

As an example, the AVERAGE aggregate is computed by each node sending the SUM and COUNT of its readings to its parent, with parents sending the SUM of SUMs and COUNT of COUNTs upwards recursively. The root finalizes the aggregate by dividing the total SUM by the total COUNT. Hence the partial state for AVERAGE is two numbers (partial COUNT and partial SUM), and twice the size of the base readings.

In Table 1 we present a simple taxonomy of aggregation functions, and the amount of partial state they must communicate. The first three classifications were initially presented in the context of traditional databases [16]; we devised the other entries to capture the sensor network case. As in [16], we are as general as possible in our taxonomy, covering not only the traditional SQL aggregates, but also any user-defined aggregation functions that might arise. This taxonomy helps us discuss aggregation techniques for related aggregates in a unified way.

One key challenge in computing aggregates in a sensornet database is energy-efficiency. In particular, the network-wide aggregate (where each node responds with its value) can incur significant communication. Energy-efficiency might not be an issue if these aggregations were infrequent; however, we believe that aggregation will be a frequently-used query operation. This is especially true in interactive settings: user studies of information analysts have shown that the first request is often for a "big picture" of the data, which is used to decide what other questions to ask [29].

Energy-efficient Aggregation

Energy-efficiency can be achieved using *approximate aggregates*. We have argued in Section 3 that a sensornet database can provide approximate results to queries. Approximate aggregates are useful for on-line monitoring, can reduce the communication costs, and simplify or obviate networking mechanisms for in-network error recovery. This approach brings up an important consideration in the design of approximate aggregates. The measure of goodness of such mechanisms is not the number of packets successfully delivered by the sensornet database, but the *information quality* of the delivered result (*i.e.,* how close the approximate result is to the true result). This information quality may be significantly affected by packet loss only under certain circumstances (*e.g.,* when the distribution of values is highly variable) and only for certain kinds of aggregates (*e.g.,* `MIN`).

There are several possible techniques for computing approximate aggregates. We now discuss each technique briefly. We intend to investigate these techniques in our research.

Class	Partial State Size	Examples and Description
Distributive	sizeof(agg)	COUNT, MIN, MAX, SUM. Partial state is a partial aggregate
Algebraic	$c \cdot$ sizeof(agg)	AVERAGE and STDDEV. State is constant size, on the order of the size of an aggregate.
Holistic	$\vert$records$\vert$	MEDIAN and RANK. All values are needed to compute the aggregate.
Content-Sensitive	$\propto f$(records)	Any holistic aggregate enhanced with compression of state records. Also many approximate "signatures" of the data set, e.g. wavelet approximations of the source distribution. [14] State is proportional to the content (e.g. entropy) of the data in the partition.
Unique	$\propto$ $\vert$distinct records$\vert$	DISTINCT variants of holistic aggregates.

Table 1: Classes of aggregates

The first of these is uniform sampling. This approach is applicable to algebraic aggregates like AVERAGE, and has been proposed in for online aggregation in traditional databases [22]. In this approach, tuples in a table are uniformly sampled and the resulting average is assumed to represent the actual average (the approach also applies to distributive aggregates like COUNT, with minor modifications). It is possible, using the weak law of large numbers, to obtain confidence intervals for the approximation. In the networking context this fails, because packet loss might invalidate the statistical assumptions that these intervals depend on. The technique itself might still be applicable in the networking context; simulations can perhaps give us an idea of how the error in the count depends on loss levels.

A variant of this approach that is applicable to counting is a class of probabilistic counting methods that use *logarithmic* sampling [28, 13]. In these approaches, the number of respondents (or the size of memory needed for the count, depending on the scheme) scales logarithmically with the size of the network. These approaches generally provide looser error bounds but use significantly less memory or communication.

Another class of approximate counting methods leverages the particular structure of result propagation and is applicable to some distributive and algebraic aggregates. Recall that the results of a query are sent up the reverse path tree towards the originator. Instead of sending, for example, a partial SUM to its parent, a node can evenly distribute the sum among all nodes within its radio range that are siblings of its parent. We call this approach *flow-based* because it splits up a count or value into many "flows" and thereby reduces the sensitivity of the aggregate to loss.

Some of the distributive aggregates like MIN or MAX are, of course, not amenable to sampling since they are highly sensitive to packet loss. For these, we can use a class of approaches that we call *hypothesis testing*. To answer certain aggregates like MAX, the query originator can pose a hypothesis answer, and see if anybody refutes it – this limits communication costs to aggregation of "refutations". This is potentially a multi-round protocol and there is a tradeoff between good guesses (which require few responses) and sensitivity to drops (less comm = more sensitivity). More generally, we think counting based schemes are amenable to hypothesis testing. Thus, an n-tile is a hypothesis of the form "there are exactly $|nodes|/n$ readings whose value is greater than a value x." It may be possible to generalize this idea, by noting that determining the shape of a distribution (which is the basis of estimating aggregates) can be done by trying to count discrete segments of the distribution – *i.e.*, build a histogram.

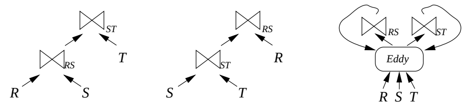

Figure 4: Three equivalent query plans: two traditional static plans, and one with an eddy. Each plan has three input tables R, S and T, and two join operators combining R with S, and S with T. The eddy version is able to adaptively reorder the join operators, effectively choosing among the two static plans dynamically.

Finally, for aggregates where the size of the partial state is a function of the number of records, data *compression* techniques are applicable. In the database literature, there has been analogous work on communicating lossily-compressed data "synopses" (e.g., [2]). Also applicable in this context is multi-resolution communication of aggregates using wavelets. With these techniques, the performance improvements clearly depend on the underlying data distribution.

In the database literature, the statistical quality of approximate results can be robustly described via confidence intervals for aggregate estimators run over i.i.d. samples of the database (e.g., [24, 22, 17]). Such a robust statistical characterization of approximate result quality for sensor networks is a much more complex challenge, since it may require modeling network losses in tandem with sensor sampling rates, noise models, and so on. One way to address this issue would use simulation and implementation to observe the phenomena causing approximation in these networks (*e.g.*, losses), and also to see empirically the relative perturbation of answer quality when reliable protocols are forfeited. We expect that these simulations will validate the usefulness of lightweight mechanisms and approximation. However, beyond these experiments, we believe that it will be necessary to do statistical research on mathematically characterizing the approximation quality of results.

5 Complex Query Optimization

Thus far, we have described how database operators might be realized in a sensornet database. In practice, as we have argued before, it will be likely that queries will comprise several operators. Generally speaking, such complex queries can be described as a tree of operators. For a given query, the order of operator evaluation can determine resource utilization. For energy-efficiency, therefore, *optimizing* complex queries will be an important goal. As we shall see, in a sensornet database, complex query optimization is intimately related to routing.

To motivate complex query optimization, consider a complex join query of the form $R \bowtie (S \bowtie T)$ (recall that $R \bowtie S$ denotes the join of tables R and S). Joins are commutative and associative, and hence the above expression is equivalent to the expression $(R \bowtie S) \bowtie T$. These expressions represent different *query execution plans* (or simply, query plans). In the first plan, the join $S \bowtie T$ is evaluated first and the resulting table is joined with R. In the second, the join $R \bowtie S$ is evaluated first and the resulting table is joined with T. These two query plans may have different costs. For example, if $R \bowtie S$ has a small number of tuples, the latter query plan may be more energy-efficient than the former. (See the left two query plans in Figure 4.)

In database systems, a *query optimizer* determines a query execution plan for complex queries. Query optimization in the database literature has treated this as a classical search problem. The search problem has three parameters: the set of feasible plans (the "plan space"), a cost model for estimating the efficiency of a plan, and an efficient search algorithm for finding the min-cost plan in the space. Given these three pieces, traditional optimizers examine a query, choose (or "compile") the best plan, and pass the plan off for execution.

Unfortunately, such static plan execution may not be appropriate for a sensornet database. Query costs are extremely dynamic in a sensor network. In the sensornet database, we expect the main query cost to be energy consumption. This is affected by the input data distributions and the operator ordering, which jointly determine the sizes of intermediate results in the query pipeline. It is also affected by network parameters including topology, loss rates and so on. Both the data and the communication in a sensor network are highly volatile, and hence a more *adaptive* query optimization approach is required.

5.1 Adaptive Optimization Schemes

Adaptive query optimization is an area of emerging interest in the database community for server-side query processing over remote data sources [21]. Among the most flexible approaches is the notion of an *eddy*, which addresses the operator ordering problem at runtime in an adaptive fashion. We now briefly describe eddies; the reader is referred to [4] for more detail. An eddy is a dataflow operator that is interposed between commutative query processing operators, as shown in the rightmost plan of Figure 4. The eddy marks tuples as it sends them to each operator, so that it knows to send a tuple to each operator at most once. Each operator may modify the tuple's contents and return it (or even multiple copies), or the operator may delete the tuple from the flow. Based on observations of consumption and production rates of the operators, an eddy *routing policy* can route incoming tuples to "better" operators first, in order to optimize the flow of data through all the operators (a simple but effective routing policy based on lottery scheduling [37] is described in [4].) Hence eddies dynamically do query optimization at runtime: they continuously recalibrate operator costs (by observing rates) and make moves in the plan space (by trying different orderings) in an adaptive fashion.

As originally envisioned for centralized processing, eddies route data among commutative operators on a single node. In a sensornet database, however, where operator execution may span multiple nodes, it might be necessary for eddies to function in a distributed, parallel fashion. This is an open area of research. One possible approach is to have an independent eddy on any node that contains more than one commutative operator, with the eddy making local decisions. Another approach is to have multiple eddies coordinate (either by observing each other's data rates, or by communicating on a control channel), and make better global decisions – possibly including decisions about operator partitioning and placement as described in the previous section.

This latter approach is essentially dynamic routing of tuples, but with some differences. The routing protocol is application-specific, as are the metrics. This is a fascinating example of an integration of functionality that would, in more traditional systems, have been considered as belonging to separable layers [8]. Also important to this kind of an adaptive query optimization is some knowledge of topology; this would help the adaptive placement of operators.

6 Conclusions

A standardized query interface for programming data collection from a wireless sensor network will greatly enhance the development of distributed sensing applications. Modeling the sensor network as a relational database can provide this functionality. Such a sensornet database can be realized, but only by carefully implementing database operators inside the network, and by relaxing the semantics of database queries to allow for approximate results. An important outcome of research in this area will be an understanding of the appropriate modularization (we hesitate to call it "layering") of sensor network subsystems, and an appreciation of the level of integration needed between different modules (*e.g.,* the routing subsystem and the database subsystem) to achieve a robust and efficient system.

In closing, we note that modeling a sensornet database as a relational table is a reasonable starting point. However, because each sensor produces a temporally ordered *stream* of tuples, it is perhaps more realistic to expect that extensions to the relational model will be necessary. The database literature has explored temporal and other *sequence-centric* data models (those in which data sequences are the conceptual units). An example of such a model is SEQ [34], which introduces sequence-based operators but does not fundamentally change the execution and optimization techniques developed for the relational model [15]. There exist conceptually straightforward extensions to the implementation of database operators that will enable sequence semantics.

References

[1] Brainy Buildings Conserve Energy. Center for Information Technology Research in the Interest of Society. http://www.citris.berkeley.edu/SmartEnergy/brainy.html.

[2] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD Conference*, pages 487–498, 2000.

[3] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 186–201, Charleston, SC, 1999.

[4] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, Dallas, May 2000.

[5] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14, 2001.

[6] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat Monitoring: An Application-Driver for Wireless Communication Technology. In *Proceedings of the First ACM SIGCOMM Latin America Workshop*, 20001.

[7] A. Cerpa and D. Estrin. Adaptive Self-Configuring Sensor Network Topologies. In *To appear, Proceedings of IEEE Infocom*, 2002.

[8] D. D. Clark and D. L. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proceedings of ACM SIGCOMM*, 1993.

[9] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Michael R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *ACM SIGMOD International Conference on Management of Data*, pages 1–8, 1984.

[10] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6):85–98, 1992.

[11] J. Elson and D. Estrin. Time Synchronization in Wireless Sensor Networks. In *Proceedings of the IPDPS Workshop on Parallel and Distributed Computing Issues for Wireless and Mobile Systems*, 2001.

[12] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Scalable Coordination in Sensor Networks. In *In Proc. of ACM/IEEE Mobicom*, 1999.

[13] T. Friedman and D. Towsley. Multicast Session Membership Size Estimation. In *Proc. of IEEE Infocom*, 1999.

[14] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *The VLDB Journal*, pages 79–88, 2001.

[15] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[16] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[17] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 287–298, Philadelphia, 1999.

[18] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the Symposium on Operating Systems Principles*, pages 146–159, Chateau Lake Louise, Banff, Alberta, Canada, October 2001. ACM.

[19] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive Data Analysis: The Control Project. *IEEE Computer*, 32(8):51–59, August 1999.

[20] Joseph M. Hellerstein, Ron Avnur, and Vijayshankar Raman. Informix under CONTROL: Online Query Processing. *Data Mining and Knowledge Discovery*, 4(4), October 2000.

[21] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[22] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1997.

[23] J. Hill, R. Szewcyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[24] W. Hou, G. Ozsoyoglu, and B. Taneja. Statistical estimators for relational algebra expressions. In *Proc. Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 276–287, 1988.

[25] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000)*, 2000.

[26] Michael Jaedicke and Bernhard Mitschang. On parallel processing of aggregate and scalar functions in object-relational dbms. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 379–389, 1998.

[27] B. Karp and H. Kung. Greedy Perimeter Stateless Routing. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000)*, 2000.

[28] R. Morris. Counting Large Numbers of Events in Small Registers. *Communications of the ACM*, 21(10), October 1978.

[29] V. O'day and R. Jeffries. Orienteering in an information landscape: How information seekers get from here to there. In *INTERCHI*, 1993.

[30] G. Pottie and W. Kaiser. Wireless Sensor Networks. *Communications of the ACM*, 2000.

[31] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location Support System. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000)*, 2000.

[32] N. Priyantha, A. M. K. Liu, H. Balakrishnan, and S. Teller. The Cricket Compass for Context-Aware Mobile Applications. In *Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2001)*, 2001.

[33] A. Savvides, C.-C. Han, and M. B. Srivastava. Dynamic Fine-Grain Localization in Ad-Hoc Networks of Sensors. In *Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2001)*, 2001.

[34] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A model for sequence databases. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, pages 232–239, Taipei, Taiwan, 1995.

[35] M. Srivastava, R. Muntz, and M. Potkonjak. Smart Kindergarten: Sensor-Based Wireless Networks for Smart Developmental Problem-Solving Environments. In *Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2001)*, 2001.

[36] D. Steere, A. Baptista, D. McNamee, C. Pu, and J. Walpole. Research Challenges in Environmental Observation and Forecasting Systems. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000)*, 2000.

[37] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

[38] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[39] F. Zhao, J. Shin, and J. Reich. Information-Driven Dynamic Sensor Collaboration for Tracking Applications. In *IEEE Signal Processing Magazine*, March 2002.