Energy-Efficient Data Organization and Query Processing in Sensor Networks

Ramakrishna Gummadi[*] Xin Li[*] Ramesh Govindan[*] Cyrus Shahabi[*] Wei Hong[†]

ABSTRACT

Recent sensor networks research has produced a class of data storage and query processing techniques called *Data-Centric Storage* that leverages locality-preserving distributed indexes to efficiently answer multi-dimensional range and range-aggregate queries. These distributed indexes offer a rich design space of a) logical decompositions of sensor relation schema into indexes, as well as b) physical mappings of these indexes onto sensors. In this paper, we discuss this space for energy-efficient *data organizations* (logical and physical mappings of tuples and attributes to sensor nodes) and examine the performance of *purely local query optimization techniques* for processing queries that span such decomposed relations.

1. INTRODUCTION

Wireless sensor networks are an emerging class of highly distributed systems with widespread applicability. In such networks, nodes generate, process and store sensor readings within the network. This architecture is necessitated by the relatively high energy cost of wireless communication—this cost makes it infeasible to consider centrally collecting and processing voluminous sensor data. An important component of these networks, then, is an energy-efficient system that enables users to query the stored data.

Existing approaches to organizing data and processing queries fall under one of the two broad categories namely, *Data-Centric Routing* (DCR) and *Data-Centric Storage* (DCS). In DCR, the data generated by the sensors is stored at the nodes that generate them, and queries are flooded throughout the network. Data from the sensors in the sensornet is then aggregated along the query tree that is built during the query flooding phase on a per-query basis. This approach, pioneered by early systems such as TinyDB [10] and Cougar [1], is efficient for *continuous* (long-running) queries, where the high energy cost incurred during the query flooding and per-query data aggregation phases is amortized over time.

Compared to DCR, DCS is a relatively new class of data storage and query methodologies proposed in [13]. In DCS, data generated by a sensor is stored at remote nodes as soon as it is generated such that related sensor data gets stored together regardless of where in the sensornet the data originates. Consequently, queries can be directed to the precise locations of the network where relevant data is stored, and data can be aggregated locally and more efficiently than in DCR-based approaches. Thus, the *overall* (insertion+query) cost for DCS is lower for many *ad-hoc* (short-lived) workloads.

[*]Computer Science Department, University of Southern California, Los Angeles, CA 90089, USA. `Email: {gummadi, xinli, ramesh, cshahabi}@usc.edu`
[†]Intel Research at Berkeley, 2150 Shattuck Ave., Suite 1300, Berkeley, CA 94704, USA. `Email: wei.hong@intel.com`

DCS can use any locality-preserving geographically distributed index structure such as DIM [8], GHT[12], DIFS [5], and DIMENSIONS [3]. Our focus in this paper is to improve the overall energy performance of vanilla DCS by a) exploiting the flexibility offered by these underlying data structures during the data insertion phase and b) optimizing query plans during query execution phase. Thus, while each DCS system currently defaults to using a fixed *data organization* (by this we mean mappings of tuples and attributes to network nodes), we seek to understand the design space of possible data organizations, and identify more energy efficient (in terms of total insertion+query costs) candidates.

In this paper, we use a distributed index called DIM [8], which serves as our basic storage, indexing, and querying layer, and is interesting because of its locality-preserving property. However, we emphasize that the choice of this distributed index is orthogonal to the data organization and query processing ideas described in this paper, and it is possible to use other indexes like GHT[12], DIFS [5], and DIMENSIONS [3]. DIM is overviewed in Section 2.1, and can be thought of as a search tree that is spatially overlaid on a sensor network. In this sense, it resembles classical database indexes. However, DIMs are also intended to store the *primary copy* of the data.

Consider a sensor network with an m-relation schema $\langle uuid, a_1, a_2, \ldots, a_m \rangle$. Tuples in this schema can be stored in one DIM. Alternatively, we can fully decompose them into m DIMs each of which stores a single relation of the form $\langle uuid, a_i \rangle$, and we can then join on *uuid* on demand to evaluate queries. A spectrum of partial decompositions of the base relation into sub-relations of the form $\langle uuid, a_i, \ldots, a_j \rangle$ is, of course, also conceivable. Clearly, we can expect these different data organizations to yield different performance under different workloads. Our measure of performance is the total energy cost incurred for a given workload, including data inserts and query retrievals; sensor networks are energy-constrained, and communication expends significant energy. We approximate the energy cost of a single message as a product of the size of the message (in bits) and the number of hops the message traverses.

We find that, in many cases, fully decomposing the base relation performs better than partial decomposition, even when the query workload is known in advance. We then study three related mechanisms that can improve the efficiency of query processing when a base relation is fully decomposed into multiple DIMs:

- Spatially Partitioning Sub-Relations. Each fully decomposed sub-relation is stored in a DIM, and all DIMs are assigned spatially disjoint sections of the sensor field.

- Efficient Query Planning via Decentralized Join-Ordering. Queries are satisfied by applying an equi-join on the decomposed sub-relations. We show that can good join-order can be obtained by summarized global information in the form of a low overhead coarse-grained multi-dimensional histogram that approximates the distribution of data stored within the network.

- Efficient Query Execution via Optimistic Join-Caching. We describe a simple and robust mechanism to cache the results of partial joins across sub-relations locally at each sensor node (Section 3.3). This caching strategy enhances query performance by eliminating redundant tuple movement during query execution.

We show using extensive simulations that for a variety of data distributions (both synthetic and real-world) and synthetic query workloads, these schemes *together* provide more than a four-fold reduction in energy expenditure over storing the base relation in one DIM even for a small number (4) of sensor attributes; we argue analytically in Section 3.1 that we can expect this factor of improvement to improve with increasing number of sensor attributes.

2. BACKGROUND AND MOTIVATION

In this Section, we describe the mechanics of insertion and querying in DIM, and motivate the performance advantages of decomposing base relations using qualitative arguments.

2.1 DIM Overview

Sensor networks are typically tasked to individually or collaboratively sense an environment and produce higher-level *events* or *features* after local signal processing and filtering. Examples of such events might be local micro-climate temperature gradients or bird sightings. It is these events that we are primarily interested in querying in an energy-efficient manner. An event can, thus, be thought of as a tuple consisting of a small (typically 4-6) number of attributes. Each attribute corresponds to a sensor type and can be treated as a column in a single relation table consisting of all possible sensor types. For example, a typical habitat-monitoring sensornet may generate tuples consisting of 4 attributes: $< l, t, x, y >$ corresponding to light and temperature readings, and the (x, y) coordinates of the sensor that sensed this reading. The events are then cast into a tuple and timestamped. Each tuple is assigned a universally-unique identifier (uuid). The uuid can be constructed as a simple concatenation of node number and a locally unique sequence number (which could be the timestamp itself).

The relational schema for a general sensor network can thus be viewed as a single table of the form $(uuid, a_1, a_2, ..., a_k)$ where k is the number of sensor attributes. Throughout the rest of this paper, we call this single logical table *sensors*. Whenever a sensor generates a tuple, it *inserts* it (or some decomposed version of it) into one or more DIM indexes, as described later.

A DIM index is best described by visualizing a collection of sensor nodes distributed on a two-dimensional surface. In DIM, this geographic region occupied by the sensor nodes is spatially partitioned such that each node “owns” the part of the region around it (we call these spatial sub-divisions “zones”). This spatial partitioning can be logically thought of as recursive equal-sized subdivisions of the 2-D space alternately along the *x* and *y* axes. Each spatial region resulting from a series of subdivisions can be assigned a unique bit code; for example in Figure 1, the zone assigned a code 1001 indicates that the zone is on the right side of the first subdivision (along the x-axis) as indicated by the 1 in the first bit, along the bottom half of the second subdivision (along the y-axis) as indicated by the 0 in the second bit, and so on. This spatial partitioning can be accomplished by a distributed algorithm that is described in [8].

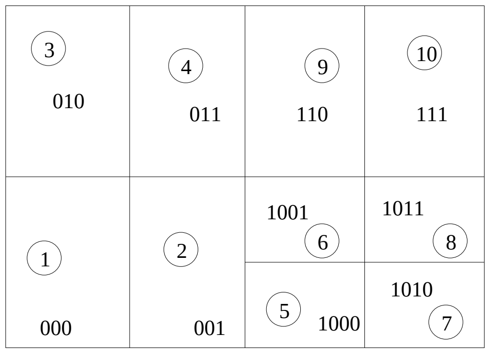

Figure 1: Example DIM Organization

Then, hyper-rectangles in the attribute space are mapped to zones. Given a tuple, nodes can compute which zone the tuple belongs to entirely locally—the only global information they need is an approximate boundary of the sensor field. They do this by essentially subdividing the attribute space in the same way that the sensor field is geographically partitioned (this algorithm generalizes to more than 2 dimensions on the attribute space). Thus each tuple can be assigned a code and will be stored at the node whose code matches that of the tuple. Tuples are then routed to the appropriate zone using a geographic routing algorithm [7]. Multi-dimensional range queries on a DIM can be described by hyper-rectangles in the attribute space. Given these hyper-rectangles, nodes can map them to DIM zones using the same mapping algorithm as was used for tuple insertions.

2.2 Motivating Alternative Data Organizations

We are now ready to consider the subject of the paper; the trade-offs involved in storing a relation in a single DIM versus decomposing it across multiple DIMs. A crucial aspect of this tradeoff is our *cost metric*: the energy cost of transporting a message from one node to another is proportional to the product of the message size and the number of hops traversed (in sensor nodes, each transmission costs significant energy).

In DIM, the cost of a query response is influenced by the node distribution (which defines the zone structure) and by how many zones need to be consulted as part of the query (*i.e.*, which nodes contain data for parts of the query hyper-rectangle). For example, in Figure 1, if the DIM is organized as a 2-D index structure over 2 attributes (l, t), to answer a query $Q1$ of the form: `select avg(t), from sensors, where 0.25<=t<0.5` the nodes with zone prefixes $\{0001, 0011, 1001, 1011\}$ (in this example, these would be nodes 1, 2, 6, and 8) would have to be consulted because the first attribute, *l*, can take on either 0 or 1.

Consider now an alternative organization in which we construct two separate DIMs, one each for *l* and *t* (we call these 1-DIMs). The 1-DIMs are fully-decomposed tables of the form $(uuid, a_i)$ where a_i is the value of the *i*’th sensor type in the tuple. The 1-DIM itself is constructed on the sensor attribute a_i, and the *uuid*’s are used for joining a_i with other sensor attributes during query execution. In this organization, we need to only consult the two nodes 3 and 4 corresponding to zone prefixes $\{010, 011\}$ to answer query $Q1$. Also, more importantly, these two zones are closer together in terms of geometric distance by a factor of 2 compared to the four zones that would need to be searched with 2-DIM.

On the other hand, for a query $Q2$ of the form: `select avg(l), from sensors, where 0.25<=l<0.75 and 0.5<=t<0.75` we would have to consult nodes owning zones with prefixes between 0110 and 1100 in case of a 2-DIM (assuming it is constructed on (l,t)). These nodes would be 4, 5, 6, 7, 8, and 9. With 1-DIMs, the prefixes would be 0100 to 1011 for l, and 1000 to 1011 for t (thus, the nodes would be 3, 4, 5, 6, 7, 8), comparable in number and distribution to the 2-DIM case. Thus, we see that the size of the query hyper-rectangle can critically affect query efficiency.

Thus, to answer queries of type $Q1$ using a 2-DIM, we needed to scan all zones for which the second attribute of the DIM, t, can be treated as a wildcard (this is true as soon as at least one attribute in a multi-dimensional query is absent from the range part). On the other hand, if we were to use two separate 1-DIMs, one each for l and t, we would not have this inefficiency. In general, we typically end up having to visit fewer and more closely separated nodes with fully-decomposed DIMs. However, in order to answer queries of the form $Q2$, using 1-DIMs, we need to (a) select both attributes $(uuid, l)$ that make up the table data for the 1-DIM on t, (b) transport them to the nodes containing the zone $0.25 < l < 0.75$, (c) locally match the l tuples with t tuples on uuid's and filter out those l's that don't have a matching t within the query range, and (d) finally aggregate the remaining values of l and return the result to the query issuer. Thus, we see that this form of query execution on partially- or fully-decomposed relations is generalizable to a larger number of sensor attributes and naturally leads to the familiar database notion of *joins* in sensor networks.

2.3 The Focus of the Paper

This paper focuses on efficiently supporting multi-dimensional range and range-aggregate queries using DIMs. Before we describe the questions that the previous section motivates, we describe some important assumptions. We assume that queries can be issued from any node in the network and data can be inserted from any node in the network at any time. We wish to accommodate a wide variety of aggregation operators because we allow for aggregation to be performed both locally at a node after collecting relevant tuples, and as a form of in-network processing. Thus, we aim to support a flexible, wide-ranging set of range, aggregation and order-statistics operators (like medians [9], *etc.*, that are not fully amenable to any form of hierarchical aggregation) without sacrificing efficiency when computing simpler aggregates like sums and averages. In all cases, our system supports SQL-like relational queries involving standard clauses like "select", "where", "group by", *etc.*

The previous subsection pointed out that decompositions of the base relation into multiple DIMs can have different query performance characteristics. The important tradeoff here is that when sub-relations are stored in DIMs, scans are more efficiently supported than when the base relation is stored in one DIM. On the flip side, however, to answer queries in general, the sub-relations need to be joined on uuid's. Joins entail additional costs in the form of data movement between the various DIMs. Clearly, then, the performance of decomposition will depend on the query workloads.

This discussion motivates several additional questions: given a decomposed base relation, how might a node decide on an efficient join-ordering? Given that query hyper-rectangles might overlap, is it beneficial to cache the results of joins to reduce data movement costs? If so, what mechanism might we use to do this? We address these questions in the next section.

3. DATA ORGANIZATION FOR EFFICIENT QUERYING

3.1 Full Decomposition and Spatial Partitioning

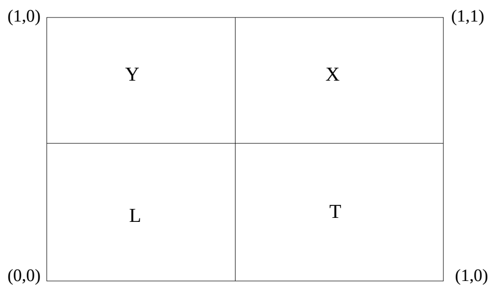

Figure 2: Example spatially-partitioned fully-decomposed DIMs

We offered qualitative arguments in Section 2.2 that a full decomposition of a relation of k attributes into multiple 1-DIMs can achieve significant energy efficiency over storing the base relation in a single DIM. A related obvious optimization we can leverage is *spatial partitioning*: the DIMs that store a sub-relation can be assigned to spatially disjoint partitions of the original sensor field. For example, in Figure 2, the l (for light) attribute values of a tuple are all stored in a 1-DIM in the lower left (quadrant 0) corner of the sensor-field, all the t (for temperature) values are stored in the lower right (quadrant 1), and the x and y values in quadrants 2 and 3 respectively. This organization constrains the distance data must move within each DIM by clustering related attribute values more densely, thereby further reducing the overall cost compared to a 4-DIM.

To motivate the utility of this optimization, consider the execution of the following simple query on this data organization: `select avg(l), from sensors`. The average message cost for answering such a query in this data organization can be expected to be $n \times 1/4$ where n is the total number of tuples in the system because these tuples would have to move an average of $1/4$ (assuming the whole square to be a unit square) to get aggregated somewhere near the centroid of the lower left quadrant. Analytically, the average distance an attribute would move would be $\frac{\int_0^{1/2}\int_0^{1/2}|x-1/4|+|y-1/4|dxdy}{\frac{1}{4}}$ (because the area of this quadrant=1/4; assuming an $\mathscr{L}_1$ norm). With $k > 4$ attributes, we can analytically show that we can get correspondingly higher multiple of message reduction, although this multiple grows as $O(\sqrt{k})$.

For a non-partitioned data organization with separate overlapping DIMs for individual attributes, intuitively, the average execution cost would be $n \times 1/2$ because the tuples now have to move an average of $1/2$ to get aggregated near the center of the unit square. Thus, the non-partitioned case moves tuples twice farther than the partitioned one, and, hence, incurs double the energy cost in this scenario. It also *quadruples* its hotspot concentration compared to partitioned.

3.2 Optimizing Join Orders

Having established that fully decomposing a base relation is at least a reasonable data organization, it remains to show how to efficiently determine join orders. Clearly, different join orders can have vastly different costs, but we make the following key observation: knowing which DIMs exist in the system, and given an approximate joint data distribution, each node can *independently* compute an efficient join order for the query. This is possible for three reasons. First, DIMs are spatially distributed indexes, and the mapping between a data item and the location it is stored can be computed locally. Second, we use a histogram to give us an *approximate* indication of the number of tuples generated by selecting the appropriate range from each of the sub-relations. Third, knowing the selectivity and the locations of the query hyper-rectangles, each node can compute the approximate cost of a query plan by estimating the messaging cost by either the Euclidean or the $\mathscr{L}_1$ distance.

In our design, each node in the network contains a query optimizer. When the query optimizer is presented with a range or range-aggregate query, it outputs a query plan consisting of a sequence of select and join operations that needs to be executed before the final aggregation. In our current instantiation of the design, the optimizer considers all possible join orders ($O(k!)$ (this is feasible if the number of attributes k is small; also, we don't consider join trees that are not simple chains because chains are simple and robust to execute distributively and tend to be good enough) and determines the least cost one.

To analytically understand the importance of good join ordering, consider Figure 2 once again. If our query is of the form: `select avg(l),avg(t), from sensors, where 0<=l<0.5`. The cost incurred for executing this query by moving t to l is approximately $n/2 \times 1/2$ for join because roughly $n/2$ tuples would be selected by the range query $0 <= l < 0.5$, and moving these $n/2$ tuples incurs a cost of $n/2 \times 1/2$ as can be intuitively seen and analytically shown. The query also has to pay an aggregation cost of $2 \times n/2 \times 1/4$ because the 2 attributes (l,t) numbering $n/2$ need to be moved $1/4$ distance after reaching the quadrant 1. Thus, the total energy cost in this case is $n/2$. On the other hand, if we were to execute this query by first joining l with t, we would have incurred a join cost of $n \times 1/2$ and an aggregation cost of $2 \times n/2 \times 1/4$ for a total cost of $3n/4$.

It is instructive to consider this query cost with that of a 4-DIM: there is only aggregation cost, and it is $n/2 \times 2 \times 1/2 = n/2$ because each node in the 4-DIM has to locally retrieve the l and t attributes of each tuple and route them to the centroid of the unit square. While this is as expensive as with spatially-partitioned 1-DIMs that use optimal join ordering, we can readily see that the join component of the query cost in the 1-DIM case can be eliminated if we can cache the join results. This would double the efficiency of the data organization scheme employing normalized DIMs, and one such caching scheme is described in Section 3.3.

Once this consistent and high join cost component of a query is eliminated, we can intuitively see why it is possible to get a small multiple (indeed, more than 2 even with 4 attributes) benefit in total energy savings by using normalized DIMs compared to 4-DIMs: the data values returned by a query come from nodes all over the sensor field (due to full-interleaving of attributes in 4-DIM) and end up getting aggregated near the center of the unit square after traveling long distances, while a normalized 1-DIM that can minimize join overhead through caching only has to pay a small variable localized aggregation cost (because of zero interleaving, the values desired by the attribute range of a query tend to come from close by nodes whose dispersion is defined only by the selectivity of a query). Additionally, spatial partitioning adds a small, on an average constant, but useful percentage to the proceedings. A secondary benefit of spatial partitioning is that it distributes the aggregation hotspots over the entire network in contrast to 4-DIM which tends to reinforce the single hotspot region around the network centroid with each query.

3.3 Reducing Join Costs through Caching

As we have seen in Section 3.2, the energy cost of a query has two components: the join cost, and the aggregation cost. The aggregation bit energy cost differs for the various data organizations, and is primarily a function of the size of the sensor field. It needs to be paid for every query by all schemes. However, the join cost need not be paid for by every query, and can be very effectively reduced through use of simple caching techniques.

The caching technique we study here is a very simple localized scheme. Consider a join order in which node **A** has to send tuples from its sub-relation to node **B** for joining. **A** remembers which tuples it has earlier transferred to **B** that fall within the current queried range. It then refrains from sending these tuples during this join step. **B** receives both the query and a partial list of tuples and knows that it has to add missing ranges of the joining tuples from its local cache.

Thus, the caching protocol is straightforward and robust because the sender nodes do not need to know which receiver has which data ranges, but only whether they have sent the relevant tuples previously or not. Furthermore, there is no distributed cache maintenance overhead to guarantee correctness in the presence of caching node failures. When node failures cause some cached data in the receiver to become unavailable, we use a simple on-demand resend-based failure recovery scheme to replenish the unavailable data. DIM provides a way for nodes in adjacent zones to takeover the failed nodes. These newly responsible nodes can invalidate the caches for data ranges in the adopted zones. Then, during the join step, they inform the sending nodes that they need to resend the entire set of tuples in the query range.

4. PERFORMANCE EVALUATION

In this Section, we evaluate the performance of our approach using simulations over both real-world and synthetic datasets on a wide variety of query workloads. Our goal is to quantify the total performance benefits of our data organization and query processing over more straightforward approaches, and we use a full-dimensional DIM as the base case against which we compare energy efficiency.

4.1 Methodology

We have implemented our mechanisms on a custom simulator that allows us to evaluate the effects of decomposition, spatial partitioning, caching and join-ordering. Our primary performance metric is the total *bit* energy cost incurred by the network as part of query execution. We evaluate this cost by keeping track of the size and number of messages transmitted by each query, and the number of hops undertaken by each message. Message sizes include header and payload sizes, and these are set to be the same as in the implementation described in Section 5 (7 and 36 bytes respectively).

We vary the number of sensor nodes in the network from 50-200 to explore scaling effects. The node locations are generated by using radio and node-connectivity models assuming a radio range of 250m and node connectivity of 9. This generates the node locations within a square grid whose size is determined by the node number, radio range, and node connectivity. Without loss of generality, we then normalize the locations to fall within a unit square.

We also vary the query workload from 100-400 queries in which each query computes aggregates over up to four attributes, with each attribute being included or excluded with equal probability; we do likewise for the range attributes. We use a variety of data workloads including uniform distributions, correlated Gaussian distributions (to better model natural sensor readings and to skew and stress the network), and data from a real-world set (measurements from the Great Duck Island dataset) [11] to better understand the behavior of our approach on varying workloads.

We normalize all the data in the real-world dataset to fall within

$[0,1)$. The synthetic dataset generator also produces values in this range. For the Gaussian distribution, we generate a four-attribute tuple in which the first two attributes are positively correlated with the mean vector $(0.5, 0.5)$ and co-variance matrix $\begin{pmatrix} 0.04 & 0.039 \\ 0.039 & 0.04 \end{pmatrix}$ and the next two attributes are negatively correlated with the same mean vector and co-variance matrix $\begin{pmatrix} 0.09 & -0.089 \\ -0.089 & 0.09 \end{pmatrix}$.

The Great Duck Island set called gdinet consists of 23548 tuples while the synthetic datasets had 4000 tuples on average each. Thus, the query-data ratio varied from $\approx 10\%$ to $\approx 0.5\%$. We normalize our energy cost by dividing the total cost by the number of nodes in the system as well as the number of tuples in the dataset. This gives us the average cost per tuple per query. Note that this cost includes the tuple's original insertion cost as well.

We also tested our approach on a wide variety of synthetic query workloads. For each workload, we re-ran our simulations with multiple instances of the workload until the standard deviation of the set of averaged results fell below 20% of that of any one run. The error-bars for all points within a series (like *Optimized* in Figure 3) are then calculated as the worst case error-bars and plotted.

We compare the performance of our proposed organization technique with four other cases, giving us a total of five scenarios for each workload and dataset:

- 4-DIM (abbreviated as *4-DIM* in the graphs). This is the straightforward case with all four attributes being stored in a single DIM.
- $1-\text{DIM}_o$ (abbreviated as *Optimized*). This is the performance with optimized join-ordering and with caching enabled on 4 fully-decomposed 1-DIMs that are spatially partitioned and each assigned to one quadrant of a unit square. However, note that the optimal join ordering does *not* use knowledge of the current state of the global cache, and only uses information that an oblivious query issuer would have access to.
- $1-\text{DIM}_u$ (abbreviated as *Uncached*). This is same as $1-\text{DIM}_o$ above, but with caching turned off.
- $1-\text{DIM}_r$ (abbreviated as *Random*). This is the performance with *random* join ordering but otherwise same as $1-\text{DIM}_o$.
- $1-\text{DIM}_m$ (abbreviated as *Worst*). This is the performance of a *worst*-case join ordering, but otherwise same as $1-\text{DIM}_o$.

4.2 Main Results

Given space limitations, we only present the main results from our extensive simulations (Figure 3 depicts some of these results).

- Caching eliminates a large fraction of joins, and gets us to the point of efficiency of 4-DIMs after which we can use optimizations like optimal join orders, full decompositions, and spatial partitioning to deliver us significant benefits.
- The relative spreads of the various series (*Optimized*, *Random*, *4-DIM*, *Worst*, *Uncached*) is roughly the same across all simulation runs, workloads, and number of queries. The absolute values depend on the simulation parameters.
- Random joining is nearly as bad as worst, and we also found (but not plotted) that average (of all possible join combinations) is also nearly as bad; this means optimal join ordering is crucial (as will become clear later on, most "good enough" join orders are OK too).
- Gaussians give much better performance over both uniform and gdinet data because of high attribute densities within the bell; but this also creates hotspots, and performance degenerates if we employ additional load-balancing. However, they occur abundantly in nature, and gdinet approximates Gaussians more than uniform.
- While we don't report the hotspot values here, we observed that $4-\textit{DIM}$ has the highest hotspot value, and *Optimized* has lowest (less intense than $4-\textit{DIM}$ by a factor of 4, as to be expected).

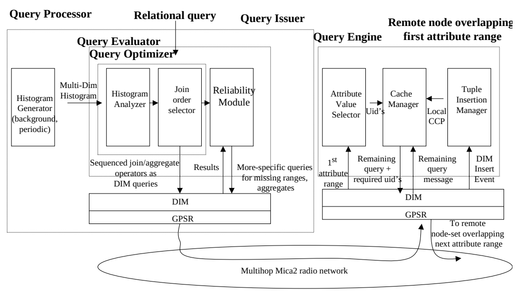

Figure 4: Software Components of the Query Engine

5. IMPLEMENTATION

In this section, we describe our design and implementation of a prototype of the main data organization and query processing ideas described in the paper and report its performance. We have implemented this prototype on the popular Mica2 mote platform[2]. The software is written in NesC [4] and runs on top of an existing DIM implementation done using the TinyOS [6] software platform so as to provide a realistic proof-of-concept demonstration of a practical and workable system. It is designed to reuse as much of the existing software infrastructure on the Mica2's as possible, and achieve the desired functionality through cleanly introduced abstractions and API's. In particular, we use a full-fledged implementation of DIM that doubles as the geographic hash-based primary storage and range-index layer, and a complete implementation of GPSR [7] as the underlying routing and packet-delivery mechanism over a multihop Mica2 radio network. This software stack organization and design is shown in Figure 4.

In Figure 4, the query evaluator runs on the query issuer and consists of a query optimizer and an end-to-end reliability module (the reliability module is responsible for retrying missing parts of the received answers). There is also an instance of the histogram generator that runs in the background on each node that lazily collects and maintains a coarse-grained multi-dimensional histogram of the tuple values present in the DIM. The query evaluator and the histogram generator taken together form the query processor.

The query optimizer, in turn, has two sub-components: a histogram analyzer that takes as input the current query and the k-dimensional histogram cube to compute the number of tuples that would be generated along each attribute dimension, and the join-order selector that then selects the optimal join order of the attributes so as to minimize the total energy cost.

The output of the query optimizer is a sequenced join and aggregate operation chain that specifies the order in which the attributes must be moved for joining and aggregation. The resulting instruction chain is packaged as a DIM query that is then handed down to the DIM layer (using the standard DIM query API) as a range query on the first attribute in the join order, with the rest of the instruction sequence forming the query payload. Thus, the query processor consumes the standard DIM API in this phase, and exposes the SQL-like query API described in Section 2.3 to the user.

The query engine running inside each node is responsible for

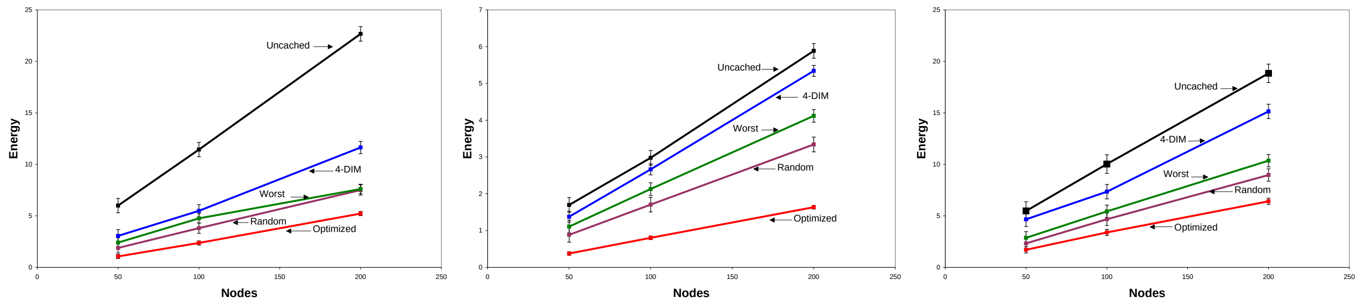

Figure 3: Comparison of 4-DIM, Optimized, Random, Worst, Uncached for Uniform with 100 queries (left), Gaussian with 200 queries (center), and GDInet with 400 queries (right)

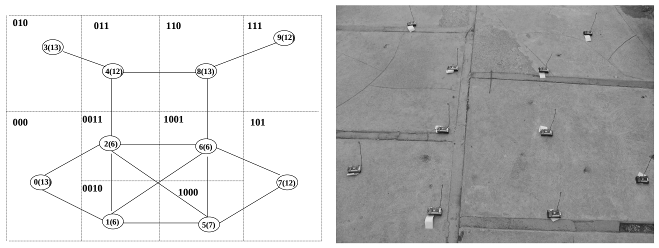

Figure 5: Topology used for Implementation

controlling the cached data (timeouts, new data inserts), and is also responsible for propagating the remainder of the query and answers.

Figure 5 shows an example 10 node topology of two 1-DIMs, one each on attributes l and t. The node id's are indicated in the circles. The number of tuples inserted into each of the 2 DIMs at each node is indicated in the parentheses adjacent to the node id. The DIM zone code of a node is indicated in the upper left corner. The right half of Figure 5 shows the physical arrangement of the nodes. They are configured to be within a few feet of each other, and the diameter of the network is four hops, as indicated by the logical link structure. We take advantage of the broadcast property during query execution (in the join caching phase).

We ran several simple SQL queries on these two attributes, and observed a factor of 2.7 performance improvement with $1-\mathrm{DIM}_o$ over standard 2-DIM.

6. FUTURE WORK AND CONCLUSION

In this paper, we tried to understand the design space of data organization and query processing strategies built on top of DCS. We examined several general techniques that can be used to provide efficient and robust infrastructure support in sensornets. In particular, we identified a few key concepts like joins, decomposition, caching, and partitioning. We are currently working on how to efficiently implement various join algorithms under our energy cost models. We are also interested in exploring better caching designs that can leverage the broadcast feature of wireless sensor networks.

7. REFERENCES

[1] P. Bonnet, J. E. Gerhke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of the Second International Conference on Mobile Data Management*, Hong Kong, January 2001.

[2] Crossbow Technology, Inc. MTS Data Sheet. http://www.xbow.com/Products/productsdetails.aspx?sid=72.

[3] D. Ganesan, D. Estrin, and J. Heidemann. DIMENSIONS: Why do we need a new Data Handling architecture for Sensor Networks? In *Proc. HotNets-I*, Princeton, NJ, October 2002.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proc. PLDI*, San Diego, CA, June 2003.

[5] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. DIFS: A Distributed Index for Features in Sensor Networks. In *Proc. IEEE WSNPA*, Anchorage, AK, May 2003.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of ASPLOS 2000*, Cambridge, MA, Novmeber 2000.

[7] B. Karp and H. T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Mobicom 2000*, Boston, MA, August 2000.

[8] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional Range Queries in Sensor Networks. In *Proc. Sensys*, Los Angeles, CA, November 2003.

[9] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGregation Service for Ad-Hoc Sensor Networks. In *OSDI*, Boston, MA, December 2002.

[10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. ACM SIGMOD*, pages 491–502. ACM Press, 2003.

[11] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97. ACM Press, 2002.

[12] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A Geographic Hash Table for Data-Centric Storage. In *Proc. WSNA*, Atlanta, GA, September 2002.

[13] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. *SIGCOMM Comput. Commun. Rev.*, 33(1):137–142, 2003.