Rebalancing Distributed Data Storage in Sensor Networks

Xin Li* Ramesh Govindan* Wei Hong† Fang Bian*

Abstract

Sensor networks are an emerging class of systems with significant potential. Recent work [14] has proposed a distributed data structure called DIM for efficient support of multi-dimensional range queries in sensor networks. The original DIM design works well with uniform data distributions. However, real world data distributions are often skewed. Skewed data distributions can result in storage and traffic hotspots in the original DIM design. In this paper, we present a novel distributed algorithm that alleviates hotspots in DIM caused by skewed data distributions. Our technique adjusts DIM's locality-preserving hash functions as the overall data distribution changes significantly, a feature that is crucial to a distributed data structure like DIM. We describe a distributed algorithm for adjusting DIM's locality-preserving hash functions that trade off some locality for a more even data distribution, and so a more even energy consumption, among nodes. We show, using extensive simulations, that hotspots can be reduced by a factor of 4 or more with our scheme, with little overhead incurred for data migration and no penalty placed on overall energy consumption and average query costs. Finally, we show preliminary results based on a real implementation of our mechanism on the Berkeley motes.

1. Introduction

Sensor networks have attracted a lot of attention because of the unique challenges in this new, low-power, highly distributed computation regime. A typical sensor network consists of tens or hundreds of autonomous, battery-powered nodes that directly interact with the physical world and operate without human intervention for months at a time. The view of the sensor network as a distributed database has been well-accepted in the sensor network community with the development and successful deployment of the pioneering sensor network database systems such as TinyDB [15] and Cougar [3]. Most of the previous work, however, has focused on power-efficient in-network query processing. Very little attention has been given to efficient and robust in-network storage for sensor networks from the database community. A sensor network as a whole can pool together a significant amount of storage, even though each individual sensor node has only a limited amount of storage (*e.g.*, each Berkeley mote has 512KB data flash). In-network data storage can be critical for certain applications where the sensor network is disconnected from a host computer or where most of the sensor data are consumed inside the network.

Our previous work has proposed a solution for distributed data storage in sensor networks, called DIM [14]. With DIM, the sensor network field is recursively divided into *zones* and each zone is assigned a single node as its *owner*. In a similar way, DIM also recursively divides the multi-dimensional data space (readings from multiple sensors) into non-overlapped hyper-rectangles and then maps each hyper-rectangle to a unique zone. The way that DIM partitions the multi-dimensional data space is data *locality-preserving*, *i.e.*, neighboring hyper-rectangles in the data space are most likely mapped to neighboring zones geographically. The owner of a zone is the node responsible for storing data in the hyper-rectangle mapped to the zone. This way, DIM builds a distributed data storage with sensor network nodes and because the mapping is data locality-preserving, DIM can efficiently answer multi-dimensional range queries issued to the sensor network.

The original DIM design as described in [14] employs a fixed data-space partitioning scheme regardless of data distributions. Therefore, when the sensor data is highly skewed (as it is in some of today's sensor network deployments (Section 2.2), *hotspots* can result such that a small number of nodes have to bear most of the storage and/or query load, and run out of storage and/or deplete their energy much faster than the rest of the network.

In this paper, we propose a novel distributed algorithm for adjusting DIM's data-space partitioning scheme based on data distributions in order to adaptively rebalance the data storage and avoid network hotspots. This algorithm collects and disseminates approximate histograms of sensor data distributions throughout the network. The histogram enables each node to unilaterally and consistently compute the resized data-space partitions without the need for a global commitment protocol and without affecting the DIM zone layout. Then based on the

* Computer Science Department, University of Southern California, Los Angeles, CA 90089, USA. Email: {xinli, ramesh, bian}@usc.edu

† Intel Research at Berkeley, 2150 Shattuck Ave., Suite 1300, Berkeley, CA 94704, USA. Email: wei.hong@intel.com

updated data-space partition, data migrate from their old storage site to the new ones if needed. We show, using extensive simulations and a preliminary implementation on the Berkeley motes, that with the rebalanced DIM, network hotspots can be reduced by a factor of 4 or more, without sacrificing the overall energy consumption and average query costs (Section 4.2). On the other hand, the overhead of data migration is relatively small when data distributions change gradually, as observed by our sensor network deployments.

Although there are some analogies between traditional database indices and DIM, it is important to note the fundamental differences as discussed below:

- Most database indices are centralized while DIM must be distributed (because of the energy constraints faced by sensor networks) and each node must be able to make decisions autonomously.
- Traditional indices optimize for the number of disk accesses while DIM optimizes for power consumption and network longevity.
- Most traditional indices optimize for an OLTP workload while DIM's workload consists of streams of new sensor data insertions and snapshot queries.
- Traditional indices perform rebalancing per insertion. This is infeasible for DIM because the act of rebalancing involves data migration which can be very expensive. We argue that in sensor networks it only makes sense to perform the rebalancing when there is a global change in the data distribution.

The rest of the paper is organized as follows. Section 2 discusses the details of DIM and motivates the need to rebalance this structure. Section 3 details the rebalancing mechanisms and explains how to preserve query semantics during the process of rebalancing. Section 4 discusses the performance of DIM rebalancing using simulations on synthetic and real data sets. Section 5 describes our implementation on the Mica-2 mote platform. Section 6 discusses related work. We conclude the paper in Section 7.

2. Background and Motivation

A typical node in sensor networks is equipped with multiple sensors. For instance, on a single Crossbow MTS310 [5] sensor board there are sensors for measuring light, temperature, acceleration, magnetic field, and sound. Thus, data generated at a sensor node is expressed as *multi-attribute* tuples. Each tuple represents a snapshot that a sensor node takes of its local view of the physical environment, *e.g.*, light, temperature, and humidity. Therefore, in a sensor database, the data space of interest is usually multi-dimensional. DIM is a distributed data structure for efficiently answering range queries to this multi-dimensional sensor data space, without flooding the entire sensor network.

In this section, we briefly describe the DIM data structure, the mechanisms of inserting and querying in this structure and the semantics it provides. We then motivate the need for rebalancing DIM with real-world examples.

2.1. DIM Overview

In DIM, data generated at sensor nodes are stored *within* the sensor network. In typical scenarios, a sensor network is deployed with a pre-defined task and then is left in the field to periodically collect multi-attribute data according to the task specification. Queries will then be injected, perhaps periodically, into the sensor network to retrieve data of interest. The query results can be used as input to other applications such as habitat monitoring, event-driven action triggering, and so on. DIM is a distributed data structure designed for this type of applications and works as a primary index that guides the data insertion and query resolution.

Given a network of sensor nodes deployed on a 2-D surface[1], DIM recursively divides the network field into spatially disjoint *zones* such that each zone contains only one node. The divisions (or "cuts") are always parallel to either X-axis or Y-axis and after each cut the resulting area is a half of the one before the division. Zones are named by binary *zone codes*. The code follows from the cuts; on every cut, if the corresponding coordinates of a zone[2] is less than the dividing line, a 0-bit is appended; otherwise, a 1-bit is appended. Given the bounding box of the sensor field, nodes can easily compute their zones and zone codes with a distributed algorithm [14]. An example of DIM network with zones and zone codes is shown in Figure 1.

In a similar way, DIM divides the data space into disjoint hyper-rectangles and uniquely maps each hyper-rectangle to a unique zone. Given the ranges of all dimensions of the data space, a node associates each network cut with a cut in the data space. As with the partitioning of the sensor field, the data space partitioning is cyclically applied on each dimension, as shown in Figure 1. The hyper-rectangle and the zone it is mapped to share the same code, *i.e.*, all data in the same hyper-rectangle have the same code and will be mapped to the same zone. In most cases, neighboring zones are assigned with neighboring hyper-rectangles in the data space and vice versa. Therefore, DIM's hashing from data space to network coordinates is *data locality-preserving*.

Figure 1 shows a DIM example where the data space is the set of (H:humidity, T:temperature, L:light) tuples, assuming that $0 \leq H < 100, 0 \leq T < 50$, and $0 \leq L < 10$. Node 5, for instance, is in zone `[1100]` and stores all (H, T, L) where $50 \leq H < 75, 25 \leq T < 50, 0 \leq L < 5$. The data locality-preserving hashing is reflected by the fact that geographically close nodes are assigned close hyper-rectangles in

1 DIM can be easily extended to 3-D space. For simplicity, we consider only a 2-D surface in this paper.

2 The coordinates of a zone are the coordinates of its geographic centroid, also called the *address* of the zone.

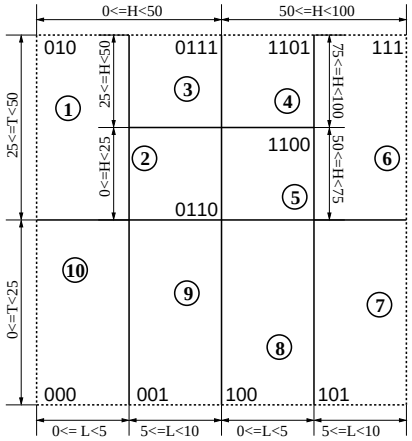

Figure 1: A DIM network: circles represent sensor nodes. Dashed lines show the network boundaries. This DIM is built on three attributes: humidity (H), temperature (T), and light (L).

the data space, *e.g.*, nodes 4 and node 5 disjointly share the hyper-rectangle $50 \leq H < 100, 25 \leq T < 50, 0 \leq L < 5$. The longer common prefix two codes have, the closer their corresponding hyper-rectangles are in the data space.

Data is stored in the node that *owns* the zone the data is mapped to. In DIM, the node storing data for a zone is called the *owner* of the zone. For example, the owner of zone `[1101]` in Figure 1 is node 4. Non-uniform deployments can cause data to be mapped to a zone where no node exists *i.e.*, an *empty zone*. An empty zone is assigned to the node whose zone code is the lexicographically closest to the code of the empty zone. For example, in Figure 1, if node 4 did not exist, the owner of zone `[1101]` would be node 5, since code `[1100]` is the lexicographically closest to code `[1101]`.

When a node sees a tuple, either generated locally or received from its neighboring node, it computes the code of the tuple using the hash function and derive the geographic coordinates of the corresponding zone. It then invokes some geographic routing protocol to route the tuple to the coordinates where the owner of the corresponding zone will be identified. All tuples to be stored are timestamped.

DIM supports tuple insertion and deletion. Updates to stored tuples are not allowed in DIM since tuples represent the snapshots of the physical environment; in other words, DIM preserves inserted data. Note that updates in themselves can be implemented easily, since they are mechanistically similar to insertion. When the storage at a node is full, data aging or summarizing schemes can be invoked to reduce the storage requirements (*e.g.*, as in [8]). The specific aging or summarization policies used are application-dependent and beyond the scope of the paper.

A multi-dimensional range query in DIM defines a hyper-rectangle in the data space. The code for a range query is the code of the smallest zone that encloses the corresponding hyper-rectangle. For example, in Figure 1, consider a query $Q = \langle 50 \leq H < 100, 25 \leq T < 50, * \rangle$ whose code is `[11]`[3] and covers nodes 4, 5, and 6. If a range query covers more than one node, it will be split into smaller sub-queries, each covering fewer nodes. The splitting of a query is executed on the query routing path whenever a node finds its zone covered by the query. Query splitting is always aligned with the data space division. For example, if query Q arrives at node 4 from node 3, it will be divided into three sub-queries $Q_1 = \langle 50 \leq H < 100, 25 \leq T < 50, 5 \leq L < 10 \rangle$ with code `[111]`, $Q_2 = \langle 50 \leq H < 75, 25 \leq T < 50, 0 \leq L < 5 \rangle$ with code `[1100]`, and $Q_3 = \langle 75 \leq H < 100, 25 \leq T < 50, 0 \leq L < 5 \rangle$ with code `[1101]`. Node 4 will then resolve sub-query Q_3 and send out the other two sub-queries.

DIM supports two query semantics depending on the availability of a mechanism to synchronize clocks network-wide [7]. In the absence of global time synchronization, the response to a query at a node is defined by the snapshot of the data that are stored at the node *when the query is processed at that node*. When all nodes' clocks are synchronized, queries can be time-stamped and a query will only retrieve tuples that are inserted before the time when the query was issued (modulo synchronization errors, of course).

2.2. Data Skews and Rebalancing

As shown in Figure 1, The data space partitions in the original DIM are, at each division, equal in size. This works well if data are uniformly distributed across the data space. However, data distributions in real world deployments are usually skewed as shown in Figures 2 and 3 (Detailed data analysis will be given in Section 4).

Skewed data distributions create *traffic hotspots* in DIM: if all data are stored at a small number of nodes, then the insertion and query traffic at those and the nearby nodes can deplete the energy of those nodes much faster than that of the other idle nodes.Thus, skewed data distributions can significantly reduce the lifetime of a sensor network that is running DIM. For this reason, DIM needs to have a mechanism that will rebalance the storage workload, and it is this mechanism that is the subject of the paper.

Classical database indices, either balanced such as B-trees [1] and R-trees [11] or unbalanced such as Point Quad Trees [6] and k-d trees [2], achieve rebalancing on a per-insertion basis by invoking *split* or *rotation* operations. Split and rotation are relatively cheap for centralized indices since they just involve pointer manipulations and/or buffer page copies. However, this per-insertion rebalancing is too expensive for DIM in sensor networks given the high cost of wireless communications. In particular, there are situations where a single insertion can cause nearly every node to transfer data (this is analogous to an insertion in centralized data structures triggering a restructuring of the entire search tree). This feature is espe-

3 According to the defi nition of a zone, code `[11]` represents a valid zone even though it is in fact divided into smaller zones as shown in Figure 1.

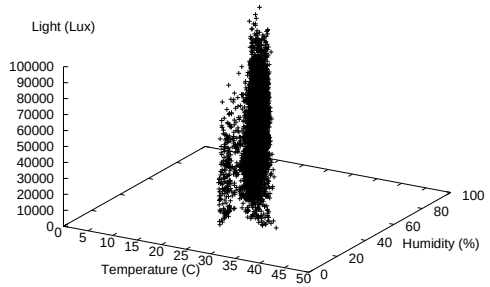

Figure 2: Forest data set collected in 15 consecutive days from a forest deployment of 10 nodes.

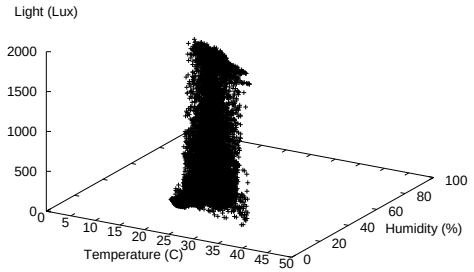

Figure 3: Office data set collected in 9 consecutive days from an office deployment of 52 nodes.

cially undesirable for volatile data, *e.g.*, short-term weather changes caused by a brief rain shower, since all data movements may need to be rolled back. For this reason, rather than considering this approach, we chose to leverage two important domain-specific properties. First, unlike the centralized indices, the goal of rebalancing in sensor networks is to reduce *hotspots* that cause node energy depletion. This implies that continuously balanced structures are not practical for performance. Second, we expect that the global data distributions in sensor networks are mostly skewed and change slowly over time in most cases, as have been observed in our real world deployments. Since it is the global data distributions that determine hotspots, this observation suggests a low overhead way of re-balancing DIM, which we discuss in the next section. Of course, if the global data distribution is significantly volatile, our approach does not work. However, as we show in this paper, local volatility in data does not de-stabilize our algorithm so long as it does not change the global data distribution characteristics.

3. Rebalancing DIM

DIM rebalances itself based on the global data distributions. In our implementation, these distributions are approximated by histograms. Global histograms are constructed by collecting and assembling the local histograms recorded at individual nodes. If a newly computed global histogram is significantly different[4] from the previous one, it is used to compute a new hashing function, *i.e.*, a new mapping from the data space hyper-rectangles to zones. As we will see in section 3.2, this *re-mapping* does not change the zones and zone codes; it just re-partitions the data space and re-assigns the hyper-rectangles to zones. Using the new hash function, a node can decide which tuples in its local storage no longer belong to its zone, and can route these tuples to their new storage sites (or owners).

4 The thresholds for when two histograms can be said to be signifi cantly different are application dependent. We do not address this issue in detail in this paper.

Figures 4 and 5 illustrate the data space partitions for the DIM network shown in Figure 1, before and after rebalancing based on the data distribution shown in Figure2. Compared with Figure 4, notice that the data space partition in Figure 5, *i.e.*, the balanced DIM, reflects the data distribution.

Unlike the original DIM design that tightly coupled the data space partitioning with the network geographic partitioning, our rebalancing scheme essentially *decouples* the two. The data space partitioning, or the DIM hash function, is entirely defined by the data distribution histograms and adapts to the global data distributions.

The cost of moving tuples from the old storage sites to the new ones due to data-space re-mapping might be expensive. However, our rebalancing design relies on a key observation of sensor network data: the changes in the *overall* data distributions can be expected to be fairly small in many cases. For example, in a habitat monitoring context, diurnal /nocturnal patterns can change the distribution, but those are on the timescale of several hours. So, in such networks, we expect that the data migration cost can be amortized over queries and inserts since rebalancing can be expected to happen infrequently (once a day or every few hours). In addition, it is possible to remove the cost of data migration completely by maintaining multiple DIM overlays on a single sensor network. We briefly discuss this in Section 3.4; the details of this technique is not the focus of this paper.

In this section, we discuss histogram collection and dissemination, the algorithm for remapping the data space, and a transition mechanism to ensure that queries are correctly executed during the process.

3.1. Collecting and disseminating histograms

Most prior work on multi-dimensional histograms has focused on centralized computation techniques. Distributed construction and/or computation of multi-dimensional histograms is still an open area and beyond the scope of this paper. Here we adopt a simplified multi-dimensional histogram scheme, assuming that the bucket size is fixed based on *a priori* knowledge about the data ranges. In Section 4 we show that fixed

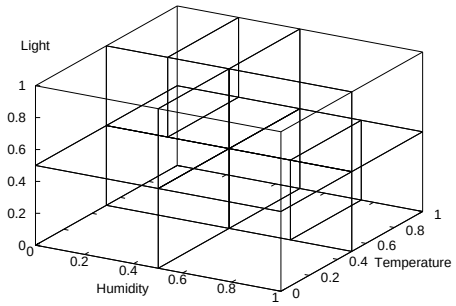

Figure 4: The 3-D data space (normalized) partition for the DIM network shown in Figure 1.

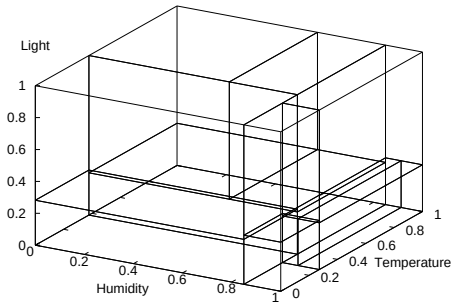

Figure 5: The 3-D data space partition of after rebalancing based on the distribution shown in Figure 2.

bucket sizes work sufficiently well for highly skewed data distributions.

The fundamental histogram collection-dissemination procedure is as follows. Each node records its local histogram which counts the number of tuples falling in each histogram bucket. A node may decide to initiate histogram collection either based on a timer or based on some heuristic (see below), and broadcasts a histogram collection request. This request effectively constructs a tree rooted at the collector [16]. The local histograms from all nodes are then delivered back to the collector along the tree, aggregated at intermediate nodes. At the collector, the local histograms are assembled to form a global histogram. If the collector decides that the global histogram has changed from the old one beyond a configured threshold to warrant a rebalancing, it will disseminate the new global histogram to the entire network.

One approach to trigger histogram collection is by setting a timer, *e.g.*, once per day or once per hour depending on the application and the knowledge about the global data distributions. This approach works well in the cases where there are apparent *cycles* such as daily temperature readings in summer. Alternatively, DIM rebalancing can also be triggered dynamically by *the most loaded* nodes. The measure of load can be one of the following: the number of tuples stored since the last rebalancing, the number of transmissions and/or receptions since the last rebalancing, or the combinations thereof. If a node finds itself overloaded, *i.e.,* beyond some threshold based on the measures above, it broadcasts a histogram collection request. It is possible that more than one node might broadcast the request at the same time.This conflict is resolved by selecting the root based on some functions, *e.g.*, node id. Other nodes suppress their advertisements if they have received one. We omit the details for lack of space.

Once the global histogram has been built up, heuristics can be used for determining if a histogram dissemination is needed, *i.e.*, if the global data distribution has changed significantly. We describe two energy-efficient heuristics. First, if the buckets are fixed, certain threshold can be placed for monitoring bucket height changes. Second, suppose that the longest zone code is known, all the data space partition points can be calculated and compared with the old ones to decide changes beyond the configured threshold.

As with other multi-dimensional histogram schemes, the cost to handle high-dimensional histograms is expensive. Since we adopted fixed buckets, for a given dimensionality, the cost to store and calculate histograms for rebalancing is bounded. The most expensive part is histogram collection and dissemination as each of them will require $\Theta(N)$ messages for a network of N nodes[5].

3.2. Remapping from Data Space to Zones

To achieve rebalancing, each node uses the new histogram to remap the data space to zones. The intuition behind our approach is to have each node redefine all its data space division points such that after each division the amount of data stored in each half is approximately the same.

Consider the 3-dimensional data space in Figure 2. Upon receiving the new histogram disseminated by the histogram collector, all nodes independently run the remapping algorithm as follows. First, find a point h on the humidity dimension such that the number of (H, T, L) pairs with $0 \leq H < h$ is equal to that with $h \leq H < 1$. If h is within some bucket, divide that bucket into two halves and reduce the bucket height accordingly (within each histogram bucket, we assume data uniformly distributed). Point h is the first division point to the entire data space. If the zone code of the node starts with 0, then drop all buckets with $h \leq H < 1$; otherwise, drop all buckets with $0 \leq H < h$. Repeat the above operations on all three dimensions alternately until the number of points computed for all three dimensions equals the bit-length of the zone code of the node. The resulting divisions comprise the new hash function for the rebalanced DIM. Figures 4 and 5 illustrate the data space partitions before and after applying this approach.

It is worth pointing out that the remapping does not change the zones and zone codes which are decided by the network physical topology. Unlike the original DIM design, after re-

5 The constant factor depends on the bucket size and the data ranges.

```
REMAP()
1  Initialize rect
2  l ← code.length
3  for i ← 1 to l
4  do d ← i mod DIMENSIONS
5     if histogram is empty
6        then new_cut[i] ← (rect[d].lower + rect[d].upper)/2
7        else h ← Number of tuples in all buckets/2
8             new_cut[i] ← Cut point to get h
9             Divide the bucket where h is located
10            Drop histograms out of rect
11            var ← |new_cut[i] − old_cut[i]| / (rect[d].upper − rect[d].lower)
12            if var < THRESHOLD
13              then new_cut[i] ← old_cut[i]
14    if code[i] = 0
15       then rect[d].upper ← new_cut[i]
16       else rect[d].lower ← new_cut[i]
```

Figure 6: DIM rebalancing algorithm. $rect$ **is the hyper-rectangle mapped to the zone indicated by** $code$**.** $rect[d]$**.lower and** $rect[d]$**.upper represent the lower and upper bound of** $rect$ **on dimension** d**, respectively.** old_cut **and** new_cut **are the data space division points before and after rebalancing, respectively.**

balancing, nodes will have different local views of the data space partition, which in the original DIM was implicitly the same for all nodes. For example, as shown in Figure 4, in the original DIM, after the first division on the humidity dimension, the second division on the temperature dimension will be the same for all nodes. In the rebalanced DIM, however, this is no longer the case, as illustrated in Figure 5. Since the two sets of data after the first division (on the humidity dimension) are distributed differently (see Figure 2), the second division (on the temperature dimension) for each half *sub*-space is different from each other.

Nevertheless, a consistent global mapping is guaranteed by the fact that the common code prefix always identifies the same hyper-rectangle in the data space. In other words, the length of the common code prefix of two nodes decides how much common global view they share. This shared global view guarantees the correctness of tuple insertion and query processing in DIM.

The pseudo code of the median-based DIM rebalancing algorithm is shown in Figure 6.

3.3. Handling DIM Transition

In the transition state, due to the change of the hash function, every node needs to check the data it has stored. If a tuple is now mapped to a hyper-rectangle different from the one the node currently owns, that tuple will be transferred to its new owner. Also, during transition, new data may be generated or queries may be issued. A problem with transition is that nodes in different areas may have different DIM versions, because of histogram dissemination delay, assuming that histograms are delivered via a reliable protocol.

In order to consistently deal with queries and insertions during transition, we tag every DIM message with a *version* number which records the latest DIM version number of all the nodes this message has gone through. Upon receiving a message m, a node, say A compares the version number $v(m)$ of m with its local DIM version number $v(A)$. If $v(A) > v(m)$, *i.e.,* node A has a more recent version than message m, node A will update m by setting $v(m) \leftarrow v(A)$. On the other hand, if $v(A) < v(m)$, node A holds message m and send a *version request* to the node, say B that is the previous hop of message m[6]. Node B responds to the version request of node A by sending its latest histogram $h(B)$ to A, *i.e.,* the histogram dissemination is repeated from B to A locally. Upon receiving $h(B)$, A replaces its latest local histogram with $h(B)$ and calls algorithm REMAP() as in Figure 6 to build a local remapping from the data space to its zone.

3.3.1. Data Migration While a tuple is transferred from its current storage site to the new one, that tuple is retained at the original node until the transfer is complete. We do this for consistency reasons, reclaiming the storage thereafter. Data migration needs to be rate limited in order to not congest the network.

3.3.2. Query Processing To process queries consistently during the transition, we need to consider two cases:

1. All nodes on the query path have not updated their DIM versions, *i.e.,* they stay in the DIM before rebalancing (henceforth called the *old DIM*). In this case, the query will be processed safely without knowing the existence of the new DIM.
2. At least one node on the query path has started using the DIM after rebalancing (henceforth called the *new DIM*). In this case, the version request will be used to retrieve the new DIM and the query will be delivered to the new DIM storage sites, as has been described.

DIM's query semantics requires that a query retrieve all qualified data inserted before the query time, *i.e.,* no matter which path the query has gone along, it should obtain the same set of responses. To satisfy these semantics, we introduced a *PULL* protocol to the original DIM design.

When entering the new DIM, node A checks the changes to the hyper-rectangles mapped to its zone and to all of the empty zones it owns. If any of the hyper-rectangles has *increased* in any of the dimensions, a PULL request is sent. The PULL request is merely a DIM query that covers the hyper-rectangle in the *old* DIM. A node processes the PULL request in exactly the same way as it would process a query, except that the version number is locked to be that of the old DIM. If the node

6 We assume that the previous hop information is available in the underlying network protocols.

has tuples matching the query, it responds to the PULL request with a PULL reply. For each received PULL reply, the requester adds the reply sender to its list, called *pull set*. The pull set serves two purposes:

1. It converts the PULL request forwarding/receiving nodes to the new DIM.
2. It associates the storage of the same data in the old DIM and the new DIM.

Thereafter, whenever node A resolves a query, it forwards the query to all nodes in its pull set to retrieve tuples stored there if the pull set is not empty. If a PULL replier has no more old data because it has completed the data transfer, it sends an END-OF-TRANSFER message to the requester to remove itself from the pull set of the requester. When the pull set becomes empty, node A will no longer forward queries to the old DIM since all data has been transferred to A.

3.4. Discussion

There are obvious trade-offs between the rebalancing performance and the granularity of histogram buckets. Within each bucket, tuples are assumed to be distributed uniformly. Thus, a smaller bucket size might enable much better rebalancing at the cost of energy in collecting and disseminating larger histograms. Various histogram compaction techniques can, of course, alleviate these problems: dropping empty buckets, merging neighboring buckets, thresholding buckets in skewed data distributions, *etc.*

The performance of DIM rebalancing is affected by several factors. First, the frequency of DIM rebalancing affects DIM performance as well as data migration costs. The scenario for which our DIM rebalancing scheme performs best is a skewed and slow-changing data distribution. In this case, DIM will quickly converge to the right data space division and change little after the first several rounds of rebalancing. In addition to data distribution, queries also contribute to traffic hotspots. For example, if the majority of queries are focused on a small area in the data space, then the node storing data for that area will become a hotspot. Query distributions, if available and converged, can in principle be used in the same way as we have used data distributions in order to avoid hotspots. However, it is less obvious that the query workload for a sensor network database will change slowly enough for our scheme to be applicable.

Second, the cost of data migration can be removed by keeping temporal data-space partitions, *i.e.*, *multi-hash* functions, one for each rebalancing. Each hash function is timestamped and tagged with a DIM version number. When issuing a query, the query source node injects the query into the network multiple times once per hash function. Intuitively, we are building multiple DIM overlays on a single network. All the overlays share the same zones and zone codes and are different by timestamps and version numbers. While saving the cost of data migration, this scheme has its own expenditure. In addition to the extra storage needed at each node to maintain the status of different hash functions, the overall query cost is multiplied by the number of overlays involved in this query, since now each query needs to be injected multiple times.

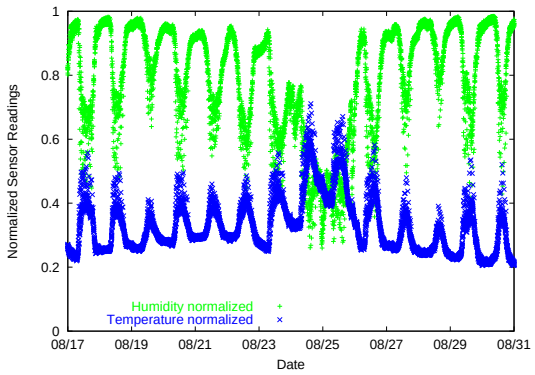

Figure 7: Redraw of the humidity and temperature sensor readings for the forest data set.

Finally, our rebalancing scheme does not affect other aspects of the original DIM design such as the local replication and the mirror replication.

4. Evaluation

In this section, we evaluate our rebalancing scheme with real world data as well as data generated by our simulator. We start with a study of data collected from real deployed sensor networks. We then describe our metrics and simulation scenarios and present the results.

4.1. Data Analysis

As shown in Figures 2 and 3, we have analyzed two data sets, *forest* and *office*. In the forest deployment, each of the 10 nodes collected sensor readings once per half an hour over 15 consecutive days. While the light readings are distributed randomly, the other two readings — temperature and humidity — are highly correlated and actually follow certain cycles over time. To show this, Figure 7 replots the temporal variation of humidity and temperature. Two observations can be drawn from our *forest* data set:

1. The overall data distribution is highly skewed. Most of the humidity and temperature readings are in a small range, *e.g.*, temperature readings are mostly found in between $20°C$ and $40°C$.
2. There is a common pattern of daily cycle, *esp.* for humidity and temperature; except for a couple of days, the same pattern was repeated with a little variation every 24 hours.

The office data set was obtained from a network of 52 nodes deployed in a large office over 9 consecutive days. A tuple (temperature, humidity, light) was retrieved from each node

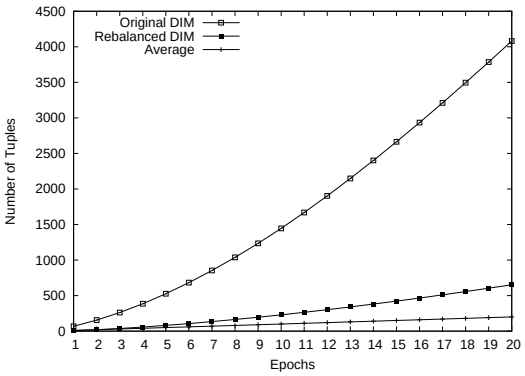

Figure 8: The standard deviation of storage usage when network size is 100 with the small bucket size.

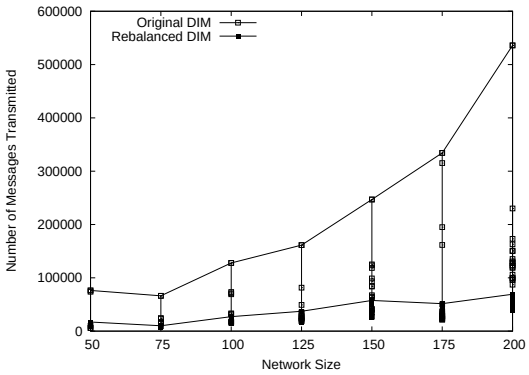

Figure 9: Comparison of the "hottest" nodes when query sizes are fixed.

once per half an hour on average. As shown in Figure 3, the office data set is also highly skewed. Unlike the forest data set, however, the light readings are more volatile at some locations in the office due to human activity.

As we can see, skewed distributions with relatively small volatile changes are the common feature of both the data sets. Later in this section we will show how DIM performs on them.

4.2. Metrics and Simulation Framework

We evaluated our DIM rebalancing approach using *ns*-2 (a network simulator) and compared it to the original DIM using the following four metrics:

1. **Standard deviation of storage load** measures the efficacy of rebalancing.
2. **Traffic hotspot** measures the number of messages transmitted by the "hottest" nodes and is a coarse indicator of network lifetime.
3. **Rebalancing overhead** measures the amount of data tuples transferred by each DIM rebalancing.
4. **Average query cost** measures the average number of transmissions per query (not including reply messages).

ns-2 is an event-driven network simulator that provides a wireless network simulation environment with user-configurable parameters. In our simulation, we used dense networks with size from 50 nodes up to 200 nodes uniformly placed in square areas. The radio range is set to 40m and the node density is set to be 1 node per 250m^2.

The data we generated for simulations were drawn from a 3-dimensional data space. We used tri-variate Gaussian distributions with the means oscillating within a small interval, 0.1 times the data range on each dimension, and the standard deviations 0.3 times the data range on each dimension. Note that the reason we chose the tri-variate Gaussian distributions is to generate *skewed* data. Our approach does not require the multiple dimensions to be correlated and merely attempts to mitigate hotspots arising from skewed distributions.

All nodes inserted the same amount of data with the same frequencies. In our simulation the ratio of insertions to queries is 2. We used two different query size distributions — exponential where "large" queries are relatively rare and fixed sized where all queries have the same size. The average for both query size distributions is the same — 10% of the range on each dimension. Queries were uniformly randomly placed in the 3-d data space. In our simulation, query sources were randomly chosen among all nodes and on average, each node issued the same amount of queries. The histogram buckets we used are cubical in the data space, *i.e.*, they have the same edge size on each dimension. We used two different bucket sizes for histograms: large size, *i.e.*, $\frac{1}{8}$ on each dimension and small size, *i.e.*, $\frac{1}{32}$ on each dimension.

4.3. Results

First we examined the standard deviation of storage usage. The result is shown in Figure 8 where DIM rebalances itself once per epoch for a total of 20 epochs. In each epoch, 1000 tuples were inserted. The tuples used in each epoch were drawn from a distribution whose mean differs slightly from the mean used in the previous epoch. Clearly, with rebalancing, nodes were able to share the storage load more evenly among them and the standard deviation increases with the same order of average increase. At the end of the last epoch, the standard deviation of storage usage without rebalancing is nearly 7 times that of the rebalanced DIM. Similar results were also shown by other network sizes we have used.

Of greater interest to us is the network hotspot, the metric that indicates network lifetime. With a balanced storage load, we expect that the rebalanced DIM networks will reduce their hotspots significantly since insertion and query reply traffic will be better distributed across nodes. Our experiments point out the rather dramatic impact of rebalancing.

Figure 9 illustrates the top 10% "hottest" nodes for networks of different sizes, 20 epochs in time, and with fixed query sizes. As can be seen, in 50-node networks the "hottest" node of the original DIM sent 4 times as many messages as the "hottest" node of the rebalanced DIM. This difference tends

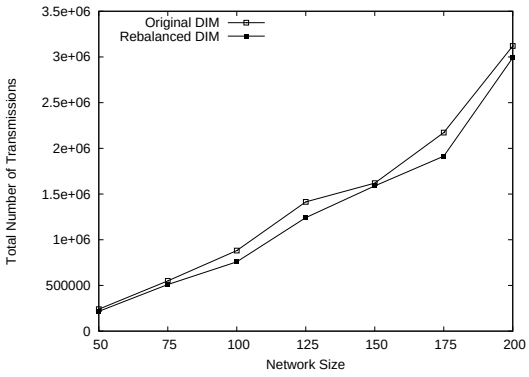

Figure 10: The overall energy consumption when query sizes are distributed exponentially.

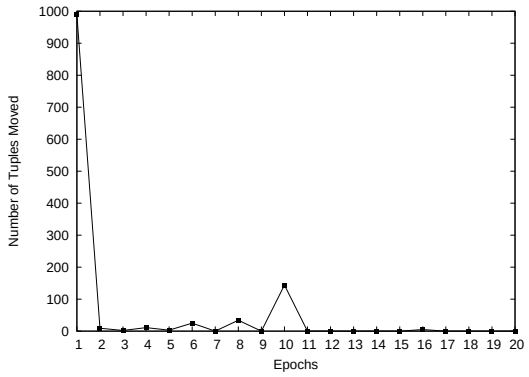

Figure 12: Data migration overhead of rebalancing a DIM network with 100 nodes and 20 epochs.

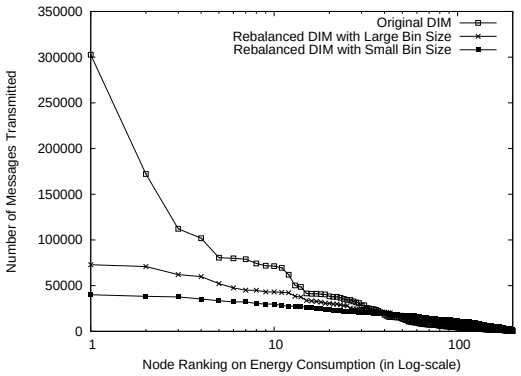

Figure 11: Comparison of the "hottest" nodes under different bucket sizes in a network of 200 nodes.

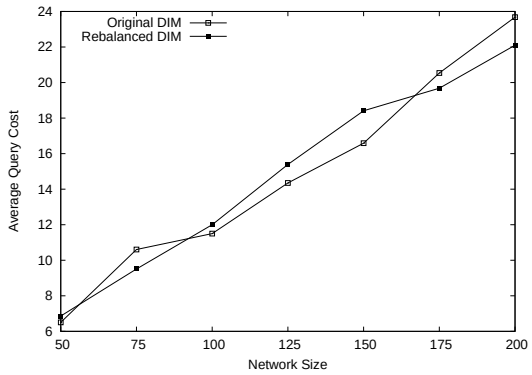

Figure 13: Average query cost comparison. The query sizes follow the exponential distribution.

to increase as the network size increases since the total number of insertions and queries has increased. When the network has 200 nodes, the ratio is nearly 8. Similar results also hold when query sizes follow the exponential distribution. Related to hotspots is the overall energy consumption measure that is shown in Figure 10 where we can see that the overall energy consumption of the rebalanced DIM, including the cost of histogram collection and dissemination and that of data migration, is comparable to the original one.

The resolution of histograms also affects the network hotspots. Intuitively, large bucket sizes, *i.e.*, coarser resolutions, will make DIM node storage load less balanced than small bucket sizes, *i.e.*, finer resolutions. Figure 11 shows the impact of bucket size on the DIM rebalancing performance in networks of 200 nodes. Clearly, large buckets move the hotspots towards the original DIM, *e.g.*, when the bucket size equals to the entire data space, the rebalanced DIM will degrade to the original DIM. However, as one can see, even with large bucket as we used here, the "hottest nodes" in the rebalanced DIM transmitted 4 times fewer messages than the original DIM. In Figure 11, we can also see the comparable average energy consumption among the the three cases, which is at the node ranking 100^{th}.

The data migration overhead measurement is shown in Figure 12 where the curve represents the amount of tuples transferred upon each rebalancing over a total of 20 epochs when network size is 100. As we have claimed, the rebalancing overhead will be amortized over time when the global data distribution is stable (as our real-world data sets show, they indeed are).

Finally, we show in Figure 13 that the average query cost, *i.e.*, the average number of transmissions per query, will not be affected by rebalancing. This is encouraging; to some extent rebalancing might be expected to reduce data locality, but this figure shows that the effect of this reduction is insignificant, given that queries are randomly placed in the data space; otherwise, query distributions can also be taken to build histograms as discussed in Section 3.4.

4.4. Results from Real-World Data Set

We also tested our rebalancing scheme by conducting trace-driven simulations on the two real world data sets (Section 4.1) with the same query distributions and bucket sizes. For the forest data set, we used a network of 10 nodes, the same size as the deployment. Every day, each node generates 480 tu-

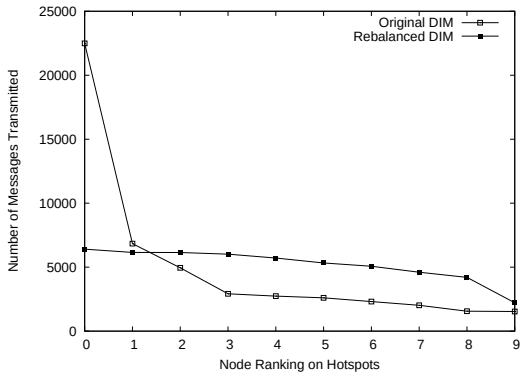

Figure 14: Hotspots measured with the forest data set shown in Figures 2.

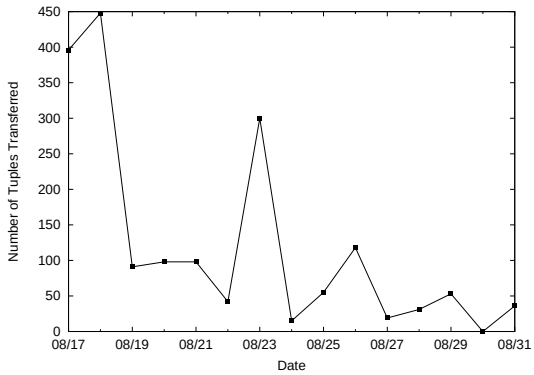

Figure 15: Data migration overhead of the forest data set shown in Figures 2.

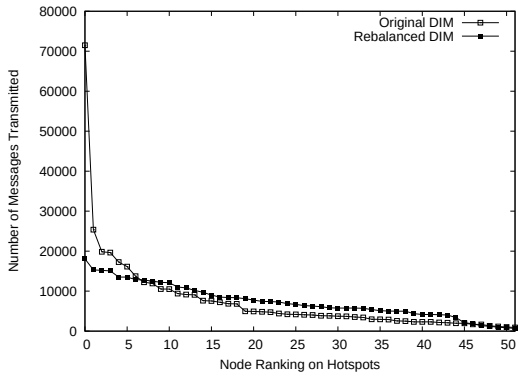

Figure 16: Hotspots measured with the office data set shown in Figures 3.

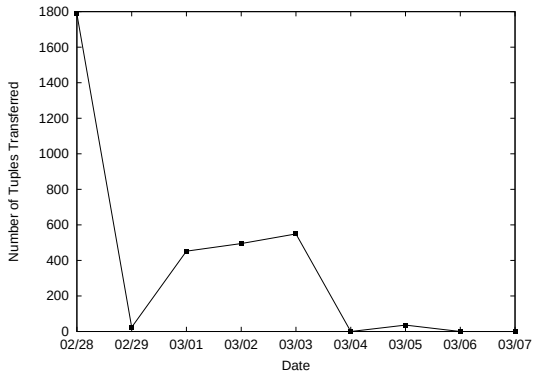

Figure 17: Data migration overhead of the office data set shown in Figures 3.

ples as explained in Section 4.1. Rebalancing is scheduled at the end of each day due to the apparent data cycles.

Figure 14 compares the hotspots of the original DIM with the rebalanced DIM under exponential query size distributions and the small bucket size. What is more interesting is Figure 15. Referring to Figure 7, we can see that our rebalancing scheme can tolerate short-term (1-2 days in this case) data distribution changes. Note that the data migration spike on 8/23 was triggered by the accumulated histogram changes[7]. The climate changes that occurred on 8/24 and 8/25 did not trigger data migration bursts since they did not significantly impact the global data distribution. One can expect that if the climate change was long-lasting, *e.g.*, season alternation, then they would finally affect the global data distribution and be reflected via DIM's data space partitions. This result is encouraging since it shows that our rebalancing scheme can converge to the global data distribution while being stable in the face of local volatility.

Similar results were also observed with the office data set. Again, we used a 52-node network, the same size as the deployment, and scheduled DIM to rebalance once a day due to the apparent daily cycles. Figure 16 illustrates the node hotspots with and without rebalancing when the query sizes were exponentially distributed. Figure 17 shows the data migration overhead for each rebalancing.

5. Implementation

We have implemented DIM and our rebalancing scheme on the Berkeley motes. The software architecture of DIM on TinyOS [12] is shown in Figure 18. The rebalancing functionality is implemented in the core DIM components (mainly in the `Zone` module) as well as in the dispatcher. The code size on the motes is over 3000 line NesC [9] code. We also implemented a Java GUI that provides a DIM front-end on PC's. The GUI allows users to create DIM, set sensor sampling rates, check node status, issue queries, and initiate rebalancing (in our current implementation rebalancing is user-triggered for simplicity). The mote connected with user's PC functions as the histogram collector.

Figure 19 shows the results of our experiments with a network of 10 MICA2 motes in an indoor deployment. The maximum path length between any pair of nodes is 3 hops. We collected light and temperature readings from each node, *i.e.*, the

7 As stated before, the histogram change is decided with the comparison threshold. We leave the further exploration of finding appropriate thresholds to the future work.

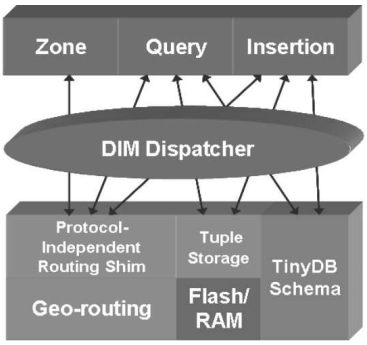

Figure 18: The software architecture of DIM as implemented on Berkeley Motes.

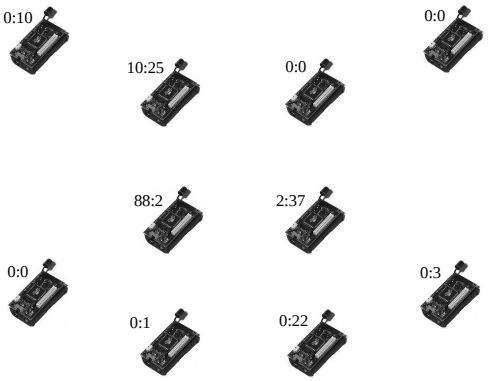

Figure 19: MICA2 mote experiment scenario with the storage usage before and after one rebalancing.

data space is 2-dimensional. The total number of tuples generated is 100, 10 from each node and no query was issued.In Figure 19, the label adjacent to each node indicates its storage usage. The first number shows storage at each node without rebalancing. Clearly, the distribution of (light, temperature) pairs is skewed with 88 tuples stored in a single node. The second number is the storage usage after a single rebalancing with the bucket size of 1/10 span on each dimension. The standard deviations of the storage usage are 26.1687 for the original DIM and 12.6174 for the rebalanced one. In our experiment, due to the skewed data distribution, each histogram needs only one TinyOS message. In general, smaller bucket size involves more traffic; half the bucket size can double the amount of histogram messages in the worst case.

6. Related Work

Existing sensor network database systems (*e.g.*, Cougar [3] and TinyDB [15]) tend to focus on processing queries inside a sensor network with query results delivered to a host computer outside the network. The default mode of operation is for a user from the host computer to issue a query which gets flooded throughout the sensor network. Query operators (e.g., sampling, selection, aggregation, etc.) are evaluated on all the nodes inside the network and eventually, query results are streamed back to the host computer outside the network. [15] has proposed extensions to this default mode of operation with per-node data buffers for storing local query results as well as a notion of a semantic routing tree (SRT) which provides heuristics for limiting the scope of query dissemination based query predicates.

In contrast, DIM extends sensor network databases with a different mode of operation where sensor nodes can autonomously generatetuples and *store* them within the network while queries can be issued from any node in the network over the data that have been generated so far. The availability of a host computer is not required. The location where a data tuple is stored is defined by a locality-preserving hash function that maps the multidimensional data space to physical network space. With DIM's hash function, queries do not need to be flooded throughout the network but can be routed precisely to the nodes that are responsible for storing the data within the query range.

Other data storage schemes in sensor networks have been proposed with different goals in mind: GHT [24] supports exact match queries (note that this is a special case of the multi-dimensional range queries supported by DIM), DIMENSIONS [8] efficiently provides spatio-temporal aggregates, and DIFS [10] provides efficient range querying on a single attribute.

DIM's design draws inspiration from decades of database research. Partitioning data spaces in order to facilitate queries have long been adopted in database indices such as B-trees [1], R-trees [11], and k-d trees [2], and their variants. Furthermore, index rebalancing has long been a concern of this line of research. Section 1 already discussed the differences between DIM's approach to rebalancing, and that adopted by traditional database indices.

Our rebalancing scheme as presented in this paper makes use of histograms to estimate skewed data distribution by counting the number of tuples falling into each bucket. Similar applications of histograms have long been studied by the database community, mainly for query optimization [19, 13, 20, 21]. The research on multi-dimensional histograms has proposed various approaches [17, 22, 18, 4, 25, 26, 23]. The application of them in a distributed context like the sensor network is open for further research.

7. Conclusion

In this paper, we have described a novel re-balancing technique for DIM, a distributed indexing and storage structure in sensor networks. Our rebalancing technique reduces network hotspots by a factor of 4 or more for skewed data distributions. Such skewed distributions are common in the real-world sensor network deployments we have analyzed. Our proof-of-concept implementation clearly indicates the feasibility of im-

plementing these mechanisms even on impoverished devices like the Berkeley motes.

Of course, the ultimate viability of these mechanisms (and of DIM itself), will be judged by successful deployments in real world scenarios. Towards this end, we intend to try deploying DIM in a real deployment very soon. Longer-term, we need to find viable mechanisms that will alleviate query hotspots; a promising approach in this direction is caching, which we intend to explore.

References

[1] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communicaions of the ACM*, 18(9):475–484, 1975.

[3] 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.

[4] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A Multidimensional Workload-Aware Histogram. In *Proceedings of the ACM SIGMOD*, Santa Barbara, CA, May 2001.

[5] Crossbow Technology, Inc. MTS Data Sheet. http://www.xbow.com/Products.

[6] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.

[7] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync Protocol for Sensor Networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, Los Angeles, CA, November 2003.

[8] D. Ganesan, D. Estrin, and J. Heidemann. DIMENSIONS: Why do we need a new Data Handling architecture for Sensor Networks? In *Proceedings of the First Workshop on Hot Topics In Networks (HotNets-I)*, Princeton, NJ, October 2002.

[9] 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 *Proceedings of f Programming Language Design and Implementation (PLDI) 2003*, San Diego, CA, June 2003.

[10] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. DIFS: A Distributed Index for Features in Sensor Networks. In *Proceedings of 1st IEEE International Workshop on Sensor Network Protocols and Applications*, Anchorage, AK, May 2003.

[11] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD*, Boston, MA, June 1984.

[12] 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.

[13] Y. E. Ioannidis and V. Poosala. Histogram-based solutions to diverse database estimation problems. *IEEE Data Eng. Bull.*, 18(3):10–18, 1995.

[14] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional Range Queries in Sensor Networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, Los Angeles, CA, November 2003.

[15] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of ACM SIGCMOD*, San Diego, CA, June 2003.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGregation Service for Ad-Hoc Sensor Networks. In *Proceedings of 5th Annual Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[17] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 28–36. ACM Press, 1988.

[18] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In C. Beeri and P. Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 236–256. Springer, 1999.

[19] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Rec.*, 14(2):256–276, 1984.

[20] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the ACM SIGMOD*, Montreal, Canada, June 1996.

[21] V. Poosala and Y. E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 448–459. Morgan Kaufmann Publishers Inc., 1996.

[22] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 486–495. Morgan Kaufmann Publishers Inc., 1997.

[23] L. Qiao, D. Agrawal, and A. E. Abbadi. Rhist: adaptive summarization over continuous data streams. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 469–476. ACM Press, 2002.

[24] 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 *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, Atlanta, GA, September 2002.

[25] N. Thaper, P. Indyk, S. Guha, and N. Koudas. Dynamic Multidimensional Histograms. In *Proceedings of the ACM SIGMOD*, Madison, WI, June 2002.

[26] H. Wang and K. C. Sevcik. A multi-dimensional histogram for selectivity estimation and fast approximate query answering. In *Proceedings of the 2003 conference of the Centre for Advanced Studies conference on Collaborative research*, pages 328–342. IBM Press, 2003.