A Sampling-Based Approach to Optimizing Top-k Queries in Sensor Networks

Adam Silberstein Rebecca Braynard Carla Ellis Kamesh Munagala Jun Yang
Department of Computer Science, Duke University, Durham, NC 27708, USA
{adam,rebecca,carla,kamesh,junyang}@cs.duke.edu

Abstract

Wireless sensor networks generate a vast amount of data. This data, however, must be sparingly extracted to conserve energy, usually the most precious resource in battery-powered sensors. When approximation is acceptable, a model-driven approach to query processing is effective in saving energy by avoiding contacting nodes whose values can be predicted or are unlikely to be in the result set. However, to optimize queries such as top-k, reasoning directly with models of joint probability distributions can be prohibitively expensive. Instead of using models explicitly, we propose to use samples of past sensor readings. Not only are such samples simple to maintain, but they are also computationally efficient to use in query optimization. With these samples, we can formulate the problem of optimizing approximate top-k queries under an energy constraint as a linear program. We demonstrate the power and flexibility of our sampling-based approach by developing a series of top-k query planning algorithms with linear programming, which are capable of efficiently producing plans with better performance and novel features. We show that our approach is both theoretically sound and practically effective on simulated and real-world datasets.

1 Introduction

Technology advances in wireless sensor networks have opened up new opportunities for collecting data from all sorts of environments. The task of effectively and efficiently querying these networks is an important and challenging problem. Because sensors are often battery-powered, the lifetime of the network is tied to the rate at which it consumes energy. In particular, radio communication is a primary source of energy consumption in sensor networks. Hence, minimizing communication in query execution can save significant amount of energy and help prolong the lifetime of the network. This goal is complicated by the topology of the network, which dictates how nodes are contacted during querying.

The seminal work by Deshpande et al. [4] proposes *model-driven data acquisition*, which suggests using models such as multivariate Gaussians to predict sensor readings. Models let us avoid visiting nodes whose readings can be accurately predicted or are unlikely to contribute to the final result. This approach can dramatically reduce the energy consumed by the network, but of course makes results approximate. Such approximation is acceptable in many scenarios, where users are often not interested in exact answers, and small errors usually have little bearing on how the results are interpreted and used. At any rate, sensors already need to cope with errors that are inevitable due to the stochastic nature of measurements.

Much work is still needed to fully realize the potential of model-driven data acquisition. As we will demonstrate, reasoning directly with models—even those as simple as independent Gaussians—can be prohibitively expensive when planning complex queries. Energy cost models and network topology further complicate this effort. Our goal is to develop a powerful, flexible, and computationally tractable framework for optimizing queries that considers energy constraints and network topology, and realizes the benefits of the model-driven approach while avoiding the complexity involved in reasoning with models.

A Closer Look at Top-k Sensor Queries In this paper we focus on optimizing top-k queries in sensor networks. A top-k query returns the k nodes with the highest sensor readings in the network. Top-k queries are both useful in practice and theoretically interesting. They serve as a case study to illustrate the challenges and difficulties in reasoning with models of joint distributions, and demonstrate the advantages of our proposed optimization framework.

Consider the simple example of ornithologists studying various bird species in a forest who do not know where birds are most likely to be found at any given time. They place bird feeders in various locations in the forest, each of which includes a sensor that detects weight changes to count the number of times birds land at the feeder. Before entering the forest, the ornithologists want to know the best locations to find birds. They run a top-k query over the network to determine the feeders that have attracted the most birds recently, and therefore are likely to be good points of ob-

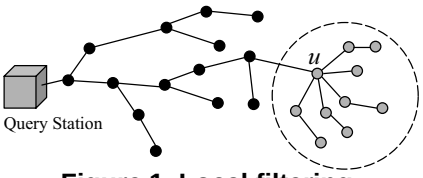

Figure 1. Local filtering.

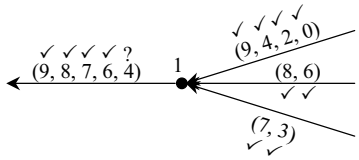

Figure 2. Constructing a proof.

servation. The ornithologists are not as interested in feeders attracting few birds; therefore, the unpopular feeders should be omitted from the query result. A top-k query meets exactly this goal. Of course, in practice, the basic top-k query may need to be more tailored to this setting, e.g., the researchers might want to group nearby feeders into clusters for purposes of observation, and obtain the top k clusters ordered by average bird count. Nevertheless, the basic form of the query remains top-k.

Optimization of top-k sensor queries is significantly more complex than selection queries (e.g., return all readings greater than x). Suppose we have a joint probability distribution over all sensor readings. A cursory approach for selecting sensors most likely in the top k is to choose the k sensors whose means are highest. Unfortunately this approach does not work. Consider an example where all readings hover near μ (with low variance), except a group of, say, $3k$ sensors whose readings are drawn from a distribution with mean $\mu' < \mu$, but with high enough variance such that each has a $1/3$ probability of being greater than μ. In this case, we expect most of the top k to come from the group of $3k$. However, the naive approach will choose other sensors simply because their means are higher.

It is possible to calculate the probability that a reading is in the top k directly from a model of the joint distribution. However, this task is computationally very expensive even for extremely simple cases, such as when each reading is drawn from an independent Gaussian (see Section 3 for details). Direct calculation involves summing over an exponential number of cases, and there does not appear to be a close-form solution. The key point here is that unlike in simple selection queries, whether a reading is in the result of a top-k query depends on the readings from other sensors.

Suppose we can efficiently estimate the likelihood of each reading being in the top k. The problem of building an efficient query plan still remains tricky and challenging. A seemingly reasonable approach is to acquire readings from the most promising sensors, and send all of their readings up to the query station. This strategy is still far from optimal. Consider the scenario depicted in Figure 1. Suppose all readings in the dotted region have an equal likelihood of ranking in the top k, yet no more than a fifth can actually do so. It is therefore suboptimal to send any more than 2 out of the 10 readings to the query station. This form of negative correlation is not uncommon in real-world situations. Consider our ornithology example. Certain birds are known to be increasingly territorial as the quality of food resources in an area increases, and it becomes more worth the effort to defend that area. Such a trend in our example would result in certain parts of the forest having less bird traffic. In these areas, although some feeders, chosen arbitrarily, might still be heavily frequented by the few birds there, the other feeders would be largely unused. In this situation, the best strategy is to obtain readings from all sensors, pass them all to the root of the subtree spanning this area, and select and pass on only the highest readings. We call this technique *local filtering*. The approach of acquiring readings and passing each of them all the way to the root falters here: It would have a hard time choosing which readings to acquire, and end up passing up many readings (mostly unneeded) in order to increase the chance that one of them is in the actual result. The challenge here is to how to build query plans that use local filtering effectively.

Finally, we face the issue of assessing the result of an approximate answer using a model-based approach. How does the user know if the result is acceptable? The typical approach is to provide a confidence measure, but that assumes the model (built on past observations) accurately reflects current conditions. We propose approximate top-k queries with "proofs," which are a stronger guarantee that does not assume the accuracy of the model. The idea is to acquire and pass up some additional readings for the purpose of proving that others are in the top k. As an example, consider the sensor in Figure 2, who has local reading of value 1 and receives three lists of top values from child subtrees. Suppose that this sensor is charged with the task of returning five top values to its parent. Among these five values, the first four can be proven because other child subtrees have returned smaller values. However, the fifth value, 4, cannot be proven because the middle subtree could possibly contain a value between 4 and 6. Note that certain values returned by the subtrees, such as 3, will not make it into the top k, but are useful in constructing proofs. We expect the cost of proof-carrying plans to be higher than those without proofs, because every reading needs to be acquired—any sensor not visited could potentially produce the highest value. How efficient can we make proof-carrying plans? And how can we do any better than a simple plan that always returns the exact answer by having each sensor return the top k values in its subtree?

Contributions We have briefly motivated a host of chal-

lenges in optimizing top-k query plans in sensor networks: complexity in using models directly, complex plans integrating local filtering and proofs, energy-constrained and topology-aware optimization. The goal of our work is to develop an optimization framework that is efficient in exploiting past behavior, powerful enough to produce plans with local filtering and proofs, and flexible enough to incorporate energy and topology considerations.

- We propose keeping samples of past sensor readings rather than maintaining models explicitly. Such samples are not only simple to maintain, since no model is explicitely constructed, but are also computationally efficient to use in query optimization. We explain how to exploit the predictive power of samples while avoiding the complexity of maintaining and reasoning with models explicitly. We show that the sampling-based approach is theoretically sound.
- We develop a query optimization framework for our sampling-based approach using linear programming. Linear programming complements sampling well and is able to encode energy constraints, topology, and complex plan features such as local filtering and proofs. Encoding energy constraints gives us a powerful ability. We can apply an energy budget the query plan building process. The resulting plan is designed to achieve the highest possible accuracy while exhausting no more energy than allocated it.
- We apply the above concepts and techniques to the top-k problem in our PROSPECTOR class of query planning algorithms, so named because query plans, like prospectors (e.g., for gold), must carefully choose where to dig given limited resources, whether into the network or the ground. We evaluate these algorithms using simulated and real-world data. We show that approximate query plans obtained using PROSPECTORs achieve good approximation results given energy constraints. We confirm the savings in using approximate query plans over exact query plans. Interestingly, by extending the PROSPECTOR algorithm for generating proof-carrying approximate plans, we obtain a two-phase exact top-k algorithm that outperforms traditional, non-model-driven exact algorithms.

2 Preliminaries

A sensor network consists of n nodes $u_1, \ldots, u_n$, each measuring a numeric value. We assume the network is organized as a spanning tree $\mathcal{T}$ rooted at a root node. $\mathcal{T}$ is initially constructed and modified over time as needed to cope with topology changes using techniques such as those in [5]. Let *parent*(u_i) denote the parent of node u_i in $\mathcal{T}$, and *children*(u_i) denote the set of u_i's children. Let e_i denote the edge (communication link) between u_i and *parent*(u_i). A top-k query ($1 \leq k \leq n$) returns the k largest values measured by the nodes. Following the convention in TAG [10], we assume that the query is pushed down into the network from the root node in a *distribution phase*, and the result values are routed up from children to parents and eventually to the root in a *collection phase*.

The primary source of energy consumption in a sensor network is radio communication. Therefore, we use the total amount of energy spent on communication as the primary yardstick for measuring the cost of a query. In a collection phase, messages are unicast from children to their parents. The total amount of energy spent in sending and receiving a unicast message with x bytes of content is given by $\sigma + \delta x$, where σ and δ represent the per-message and per-byte costs, respectively. In the table below, we show how to calculate a typical value for δ from the specification of MICA2 motes [2]:

sending cost (s)	36 mJ/sec
receiving cost (r)	14.4 mJ/sec
byte rate (b)	2500 bytes/sec
per-byte cost ($\delta = (s + r)/b$)	0.02016 mJ/byte

The per-message cost σ stems from the "handshake" between the sender and the receiver required by a reliable communication protocol, and the additional message header preceding the actual message content. Our use of a reliable protocol comes into play when we deal with failure in Section 4.4. For MICA2 motes, a typical value for σ in our query scenario is .645 mJ, which is high compared with δ. In other words, merely communicating with a node, regardless of how few values are transmitted, incurs a significant cost. This observation motivates approximation algorithms that reduce energy consumption by visiting only a subset of the nodes; on the other hand, such algorithms necessarily give up the exactness of results because any node not visited could produce a value in the actual top k.

We differentiate between an initial distribution phase and subsequent distribution phases. In the initial distribution phase, a new query plan is installed into the network. In this case, each node sends a subplan to each of its children using a unicast message, whose energy cost can be estimated as discussed previously. We assume that each node stores its part of plan (i.e., how many values it expects from each of its children and how many values need to be returned to its parent). Once the plan has been stored, subsequent distribution phases can use broadcast messages to trigger query execution. A "re-execute" message with an empty body is recursively broadcast to subtrees where some values are expected. Thus, the total energy spent in a subsequent distribution phase is much less than a collection phase and the initial distribution phase. To simplify discussion, we focus only on the collection phase, and omit the details of handling the energy costs of the distribution phases. Nonetheless, our experimental results do account for the costs of distribution phases.

Naive Exact Top-k Algorithms We begin with two naive algorithms that guarantee exact answers to top-k queries without using any predictive models. These algorithms provide a tradeoff between the number of messages used and the number of values transmitted.

The first algorithm, NAIVE-k, computes the answer bottom-up in one pass over the network. Each node simply collects the top k values from each of its children, computes the top k among all such values and its own, and passes them on to its parent. If the subtree rooted at a node has fewer than k nodes, then all values from the subtree are passed up to the parent. This algorithm uses the minimum possible number of messages, since every node must be visited in order to guarantee an exact answer. However, the sizes of the messages used by NAIVE-k are quite large. A node with fan-out f receives fk values from its children, but at least $(f-1)k$ of them will not be in the final result, representing a significant waste of bandwidth.

The second algorithm, NAIVE-1, computes the answer in a pipelined fashion, where each node requests only one value at a time from its children. More precisely, each node maintains a heap containing its own value and the last value requested from each of its children. When the node receives from its parent a request for a value, the node first ensures that the heap has a value from each of its children (unless the child has no more values to return); if not, a new value is requested from that child. Then, the largest value in the heap is removed and returned to the parent. Intuitively, NAIVE-1 attempts to minimize the number of values transmitted. A node with fan-out f receives no more than $f + k' - 1$ values from its children, where $k' \leq k$ is the number of values requested from the node by its parent. However, since each value is transmitted in a separate message, NAIVE-1 requires a large number of messages, and the overhead incurred is prohibitive.

Approximate Top-k Query Plans An approximate top-k query plan allows for a continuous tradeoff between the accuracy of the result and the cost of querying. Formally, a single-pass approximate plan is an assignment of "bandwidth" $b(e_i)$ to each edge e_i in the network. This bandwidth represents the number of values that should be transmitted on e_i from u_i to *parent*(u_i) in a collection phase. Upon receiving the lists of values from its children, node u_i sorts these values along with its own, and sends the top $b(e_i)$ values up to its parent. Naturally, we require that $0 \leq b(e_i) \leq 1 + \sum_{n_j \in \textit{children}(n_i)} b(e_j)$. The answer to the query consists of those values returned by the root node.

Given a limit on the total amount of energy the plan is allowed to spend on communication, it is conceivable that the answer produced by the plan does not exactly contain the top k values in the network. The goal is to find a plan that minimizes the expected number of top k values not returned. The expectation is taken over the joint distribution of all values in the network.

3 Sampling-Based Query Planning

As mentioned in Section 1, reasoning directly with models of sensor values can be computationally intractable. Recall that we cannot order nodes' likelihood of ranking in the top-k simply by sorting them by expected value. We must calculate the probability that each node ranks anywhere in the top-k. To illustrate the complexity of this task, consider the problem of calculating the probability that node u_n has the k-th ranked value in the network. This occurs in $\binom{n-1}{k-1}$ disjoint cases, each with a different subset of $k-1$ nodes whose values rank higher than that of u_n. For simplicity, assume that the value at u_i is drawn independently from a distribution with probability density function $P_i(x)$ and cumulative distribution function $D_i(x)$. The following integral computes the probability that u_n's value ranks lower than those from nodes in a set U but higher than all others:

$$\int P_n(x) \prod_{u_i \in U} (1 - D_i(x)) \prod_{u_j \notin U \cup \{u_n\}} D_j(x)\, dx.$$

Even for the simple, common case where all distributions are normal, the above integral has no closed-form solution and is expensive to evaluate numerically. Moreover, there are $\binom{n-1}{k-1}$ such integrals for each of k ranks. The calculations must be repeated for each node. If we were to drop the independence assumption, or try to reason about more complex scenarios such as local filtering, the complexity would get even more out of hand.

Our solution is to base optimization on a set of samples from the joint distribution of all values, where each sample is an assignment of value for each node in the network. These samples can be easily generated at runtime using the "exploration/exploitation" framework from machine learning literature [9]. At randomly chosen timesteps, we spend more energy to collect all values in the network and use them as a sample. The most recent samples are maintained and used in optimization. This approach naturally adapts to changes in the joint distribution over time. Maintaining a model encoded in samples, then, is extremely simple to maintain, since no model is explicitly built. Alternatively, if a model of the joint distribution is already available, we can use it to generate random samples directly.

We translate each sample from a set of sensor values into a Boolean vector, whose i-th component is 1 if the value from node u_i is among the top k in this sample, and 0 otherwise. The vectors from a set of samples form a Boolean matrix $\mathbf{S}$, where $\mathbf{S}[s,i] = 1$ if and only if u_i produces one of the top k values in the s-th sample. The optimization goal can be restated as follows for sampling-based query planning: Given a network-wide limit for energy consumption, find an approximate top-k query plan that minimizes, over all samples, the total number of top-k values (i.e., 1's in $\mathbf{S}$)

that fail to be returned by the plan.

Note that this approach can be easily generalized to queries that return subsets of all sensor values, e.g., selection and quantile queries. In the general case, we would set $\mathbf{S}[s,i] = 1$ if u_i contributes to the answer in s-th sample, and 0 otherwise. The optimization goal would still be to minimize the total number of 1's in $\mathbf{S}$ missed by the plan.

The intuition behind sampling-based query planning is that given a large enough set of samples, patterns of nodes contributing to answers will emerge. Such patterns include obvious ones where certain nodes appear most often in answers, as well as more subtle ones where a subtree may consistently contain the same number of answer values, even though in each sample they may come from different nodes in the subtree. Looking at the samples, we can optimize an approximate plan by encouraging acquisition of values from nodes and subtrees that contribute most to the answers.

We call our sampling-based query planning algorithms PROSPECTORS. To warm up, we first present a simple PROSPECTOR that uses a greedy strategy. Then, in the remainder of this section, we provide the theoretical foundation for our sampling-based approach. More sophisticated PROSPECTORS that consider network topology, local filtering, and proofs will be presented in Sections 4.

PROSPECTORGREEDY This simple algorithm constructs a query plan incrementally by greedily adding values to be obtained, one at a time. The intuition is to give priority to nodes who are contributing the most to the top k values over all samples. As long as the energy cost of the plan constructed so far does not exceed the prescribed budget, the algorithm greedily picks the node u_i (among all nodes not visited by the current plan) for which $\sum_s \mathbf{S}[s,i]$ is the largest, and expands the current plan to obtain the value from u_i.

3.1 Theoretical Foundation of Sampling

As discussed earlier, the objective function of our optimization is to minimize the expected "error" over the entire joint distribution of all sensor values. The size of this distribution is exponential in the number of values. The essence of the sampling-based approach is to avoid enumerating the entire joint distribution, and instead to approximate the joint distribution by drawing a number of samples over which optimization will be carried out. Shmoys and Swamy [13] have recently shown that this approach works for solving a large class of *two-stage stochastic optimization problems with recourse* by solving a linear relaxation of the integer program on the sample and converting the resulting solution into an integer solution. For such problems, only a polynomial number of samples are needed to achieve arbitrary degree of accuracy in the solution of the linear relaxation.

We present an example of such a problem, called STOCHASTIC-STEINER-TREE, which arises in demand forecasting applications. Consider a resource provisioning algorithm whose goal is to purchase links to connect a set of demands to a central hub; the goal being to minimize the cost of links purchased. On day 1, the algorithm only has access to the probability distribution $\mathbf{D}$ over *scenarios*. Each scenario is a set of nodes which requires connectivity on day 2, and represents one possible forecasted event. On day 2, the actual node-set that requires connectivity is revealed (this set of nodes is a scenario drawn from $\mathbf{D}$). The cost of any link is cheaper on day 1 than on day 2 by a factor α. The algorithm can purchase some links on day 1 cheaply to construct a subtree, but with knowledge only of $\mathbf{D}$. Then, on day 2, when the actual node-set is revealed, the algorithm has to purchase additional links to ensure that the entire revealed node-set is connected to the hub. Since link costs are lower on day 1, it makes sense for the algorithm to purchase some links with just knowledge of $\mathbf{D}$. The goal is to choose links to purchase on day 1 so that the expected total cost of the solution is minimized, where the expectation is over the probability distribution $\mathbf{D}$ of node-sets revealed on day 2.

This problem is a two-stage stochastic optimization problem because the algorithm has to allocate resources in two stages, with only knowledge of the *distribution* of possible scenarios in the first stage. Shmoys and Swamy [13] show a polynomial time algorithm based on linear programming for approximately solving such problems using samples from the distribution of scenarios. They first formulate the problem as an integer program over the entire joint distribution, which is linearized by relaxing the integrality constraints. Sampling from the distribution then yields a polynomial sized linear program which is an arbitrarily good approximation to the original program. Their main result shows that to achieve $1+\epsilon$ accuracy in the solution to the linear program, the number of samples needed is poly$(\frac{1}{\epsilon}, \frac{1}{\alpha}, n)$, where α is the factor by which a link is cheaper on day 1 than day 2, and n is the number of vertices in the graph. Note that their algorithm works even if the size of the distribution $\mathbf{D}$ is exponential. It is also easy to extend their algorithm to the case where there is a bound on the first-stage cost, and the goal is to minimize the expected cost of the second stage. They further show how solving the linear program yields an approximate solution to the integer program; for this step, we require a much simpler approach for our problem, which we outline in the next section.

Consider now the following simpler formulation of the top-k query planning problem called SIMPLE-TOP-K. The root can access any node at unit cost, and the values at nodes follow some joint distribution. The algorithm is given a bound R on the number of nodes it can query. The goal is to choose these nodes to minimize the expected number of values in the top-k which are not queried. We show the following:

Theorem 1 SIMPLE-TOP-K *is a special case of*

STOCHASTIC-STEINER-TREE.

Proof: Each node is connected by a link of unit cost to the root. The cost of purchasing this link is the same on day 1 and day 2. The nodes that require connectivity are the nodes in the top k for any sample from the joint distribution. We treat these samples as the scenarios. The algorithm only has access to the distribution on day 1. The R nodes that the root queries are the links which are purchased on day 1, and their cost is upper-bounded by R. On day 2, the links that need to be purchased correspond to the top k values that are not queried by the root. Therefore, minimizing the expected cost on day 2 corresponds exactly to minimizing the expected number of top k values that are not present at the root when it queries the R nodes in SIMPLE-TOP-K. Note that the purchase of links on day 2 is just a "thought-experiment" in SIMPLE-TOP-K; the root does not actually query the missing nodes on day 2. All we claim is that this cost, had the root known they were in the top k and queried them on day 2, is the same as the number of top k values missed by the root on day 1. □

We can therefore apply the STOCHASTIC-STEINER-TREE framework to solve this problem by integer programming. This integer program can be approximately solved by sampling, writing a linear relaxation on the sample, and converting the solution to the resulting linear program into an integer solution. Since $\alpha = 1$ in the case of SIMPLE-TOP-K, the number of samples needed is poly$(\frac{1}{\epsilon}, n)$, independent of the size of the joint distribution of the top k values. In Section 5, we show through experiments that the number of samples required is indeed low in practice.

We present the actual solution in the next section. The problem we actually solve is a generalization of SIMPLE-TOP-K, but still falls within the class of two-stage stochastic optimization problems. The only issue is that the cost of the first stage is actually the cost of the bandwidth allocation and the cost of the second stage is the number of top k values missed by the root. Their ratio, $\frac{1}{\alpha}$ need not be polynomially bounded. Since we bound the first stage cost and optimize the second stage cost, their being incomparable does not affect the quality of the solution and keeps the samples polynomially bounded. Theoretically, this is achieved by scaling the first stage costs so that they lie in a polynomial range. We omit proof of this claim, since we have not performed any scaling in our implementation and the number of samples required is still small (Section 5).

We believe several other problems arising in sensor network optimization, when suitably formulated, might be amenable to solutions in this framework.

4 Linear Programming Formulations

In this section, we describe how to optimize approximate top-k queries over samples using linear programming. We present a series of PROSPECTOR algorithms, which use linear programs of increasing complexity to obtain plans with better performance or additional features. PROSPECTORLP−LF exploits the topology of the sensor network in optimization, but does not consider plans with local filtering. PROSPECTORLP+LF is not only topology-aware, but also considers local filtering. PROSPECTOR-PROOF further adds proofs to plans, and we show how to extend it to a two-phase exact top-k algorithm, PROSPECTOREXACT, which can outperform naive exact algorithms described in Section 2.

The PROSPECTORs also demonstrate our ability to use linear programming to constrain query processing to a set energy limit. The algorithms construct plans to return the highest accuracy given that limit.

For the discussion below, we introduce some additional notations. Let *anc*(u_i) denote the set of ancestors of node u_i and itself, and *desc*(u_i) denote set of descendants of u_i and itself. Let the cost of sending a message with x values along edge e_i be $\sigma_i + \delta_i x$. Recall that typically values for σ_i and δ_i are given in Section 2 (δ_i needs to be scaled by the number of bytes per value). The algorithms are given a cost limit of C.

4.1 PROSPECTORLP−LF

PROSPECTORLP−LF, our first formulation of PROSPECTOR using linear programming, considers the network topology in optimization. Unlike PROSPECTORGREEDY, this algorithm encourages plans to obtain promising values clustered under the same subtree, because these values can be transmitted together in one message from the subtree to the root, without incurring separate per-message costs. On the other hand, like PROSPECTORGREEDY, this algorithm does not support local filtering; any value that it chooses to obtain will travel all the way to the root.

We use one integer variable $x(u_i)$ for each node u_i; a value of 1 means that the value at u_i is chosen by the plan and will be transmitted to the root, while a value of 0 means the plan will ignore this value. We also use an integer variable $a(e_i)$ for each edge e_i to indicate whether e_i is ever used for communication by the plan. The linear program is shown below, followed by explanation:

$$\text{(1)}\quad \text{Minimize } \sum_i \Big((1 - x(u_i)) \sum_s \mathbf{S}[s,i]\Big), \text{ subject to:}$$

$$\text{(2)}\qquad x(u_i) \le a(e_j) \quad \forall i,j \text{ s.t. } u_j \in \textit{anc}(u_i);$$

$$\text{(3)}\quad \textstyle\sum_i \big(a(e_i)\cdot\sigma_i\big) + \sum_i \big(x(u_i)\cdot\sum_{u_j\in \textit{anc}(u_i)}\delta_j\big) \le C;$$

$$\text{(4)}\qquad 0 \le x(u_i), a(e_i) \le 1 \quad \forall i.$$

Line (1) captures the optimization goal of minimizing the total number of top-k values missed by the plan over all samples. The term $\sum_s \mathbf{S}[s,i]$ counts the number of times that u_i's value is in the top k.

Line (2) captures the constraint that if the plan chooses the value at u_i, then all edges above u_i are used for communication (to transmit this value).

Line (3) upper-bounds the total cost by the prescribed budget. The first summation calculates the total per-message costs incurred on all edges used by the plan. The second summation calculates the total cost of sending the chosen values to the root, ignoring any per-message overhead. If u_i is chosen, then the plan pays the per-value cost on all edges above u_i, totaling $\sum_{u_j \in anc(u_i)} \delta_j$.

Line (4) captures the integrality constraints on variables $x(u_i)$ and $a(e_i)$. Strictly speaking, these constraints should be expressed as $x(u_i), a(e_i) \in \{0, 1\}$. Since most solvers work on linear constraints, we replace these constraints with linear ones on Line (4). The resulting program can then be efficiently solved by a linear solver such as the simplex method. The solution produced by the solver may have some of the $x(u_i)$ and $a(e_i)$ variables set to fractional values, though it is unlikely in practice. In the event that a fractional solution happens, we can construct a feasible integer solution from it by setting $x(u_i) = 1$ if $x(u_i) \geq 0.5$ in the fractional solution, and setting $x(u_i) = 0$ otherwise; variables $a(e_i)$ are handled in the same way. It can be easily shown that the resulting integer solution increases the objective function value by at most a factor of 2, and costs at most $2C$. We omit the detailed proof here. In practice, we find that the linear relaxation performs much better than what the theoretical bound guarantees.[1]

Finally, note that the only place where we use the Boolean sample matrix $\mathbf{S}$ is in calculating the total number of 1's in a column. Therefore, instead of storing the entire $\mathbf{S}$, we can simply maintain a vector of column sums (except in situations where we want to maintain a "window" of recent samples and expire old ones from the window). The same optimization also applies to PROSPECTORGREEDY.

4.2 PROSPECTORLP+LF

In this version of PROSPECTOR, we consider approximate plans with local filtering. We use two integer variables for every edge in the network: $b(e_i)$ denotes the "bandwidth" assigned by the plan to edge e_i, representing the maximum number of values that can be transmitted along that edge; $a(e_i)$ indicates whether e_i is ever used for communication by the plan, just as in PROSPECTORLP−LF. The assignment of bandwidth to edges completely specifies an approximate top-k query plan, as described in Section 2.

Local filtering will take place when $b(e_i) < \sum_{u_j \in children(u_i)} b(e_j)$, because in this case u_i receives more values than it sends up. For example, in Figure 1, PROSPECTORLP+LF may allocate a bandwidth of 3 to the edge out of node u, even though this node is going to receive 9 values from its children. As motivated in Section 1, local filtering can greatly improve the performance of a plan by allowing it to examine more values and make more informed decisions at runtime on which values to pass on. The challenge to query optimization is how to determine the appropriate amount of filtering, i.e., what amount of bandwidth to assign to each edge. We will now address this challenge with PROSPECTORLP+LF.

To capture local filtering in a linear program, we need some more integer variables. For the s-th sample, let $ones(s)$ denote the set of nodes that provide the top k values for this sample. In other words, $u_i \in ones(s)$ if and only if $\mathbf{S}[s, i] = 1$. We use an integer variable $x_s(u_i)$ for each such entry. Setting $x_s(u_i)$ to 1 means that the plan returns the value at u_i when executed on the s-th sample; otherwise, setting $x_s(u_i)$ to 0 means that the plan misses the value at u_i (which should have been returned) for the s-th sample. The linear program is shown below, followed by explanation:

$$\text{(5)} \quad \text{Minimize} \sum_s \sum_{u_i \in ones(s)} (1 - x_s(u_i)), \text{ subject to:}$$

$$\text{(6)} \quad \sum_i \big(a(e_i)\sigma_i + b(e_i)\delta_i\big) \leq C;$$

$$\text{(7)} \quad x_s(u_i) \leq a(e_j) \quad \forall s, i, j \text{ s.t. } u_i \in ones(s) \cap desc(u_j);$$

$$\text{(8)} \quad \sum_{u_i \in ones(s) \cap desc(u_j)} x_s(u_i) \leq b(e_j) \quad \forall s, j;$$

$$\text{(9)} \quad 0 \leq a(e_i), x_s(u_i) \leq 1 \quad \forall s, i.$$

Line (5) specifies the optimization objective: to minimize the total number of misses over all samples.

Line (6) constraints the total cost of the plan.

Line (7) encodes the constraint on when an edge needs to be used for communication, analogous to Line (2) of PROSPECTORLP−LF. Here, the constraint is slightly more complicated. Intuitively, if the plan returns the value of u_i for the s-th sample (i.e., $x_s(u_i) = 1$), then the plan must use each edge e_j above u_i for communication (i.e., $a(e_j) = 1$).

Line (8) encodes the bandwidth constraint, which captures the fact that the bandwidth allocated to an edge e_j necessarily limits the number of top k values that can be returned from the subtree rooted at u_j, since all such values must be go through e_j.

Line (9) captures the integrality constraints on variables $a(e_i)$ and $x_s(u_i)$ with linear relaxation, analogous to Line (4) in PROSPECTORLP−LF.

The biggest difference between PROSPECTORLP+LF and PROSPECTORLP−LF is that the former uses one variable $x_s(u_i)$ for each 1 entry in the Boolean sample matrix $\mathbf{S}$, while the latter uses one variable $x(u_i)$ for each column

[1] We also note that PROSPECTORLP−LF with integrality constraints might be solvable to an arbitrarily good approximation factor by dynamic programming. In particular, our NP-hardness proof for this problem reduces from the KNAPSACK problem for which such a guarantee is achievable. However, the other algorithms in this section may not be amenable to dynamic programming; therefore, we use the linear programming framework, which is generally applicable to all our algorithms.

of $\mathbf{S}$. The additional variables allow PROSPECTORLP+LF to capture plans that make finer-grained decisions using local filtering. In contrast, having only one variable $x(u_i)$ for each node u_i implies that the plan must return u_i's value either for all samples or not at all; in other words, the plan cannot make a run-time decision on whether to pass on u_i's value based on local filtering.

4.3 PROSPECTORPROOF

The previous approximate algorithms are unable to guarantee the accuracy of their results unless we assume that the model or past samples indeed reflect the current behavior of sensor values. As motivated in Section 1, if a stronger guarantee is desired, we can augment an approximate top-k query plan with a "proof" constructed in a bottom-up fashion by passing up additional non-top-k values necessary for proving the correctness of top-k values returned. We now present PROSPECTORPROOF, an algorithm capable of generating such plans. We begin by describing a proof-carrying top-k query plan in more detail.

As in a regular top-k query plan, each edge in the network is allocated a bandwidth $b(e_i)$ specifying the maximum number of values that node u_i can pass up. Additionally, in a proof-carrying plan, u_i will ensure that among the $b(e_i)$ values that it returns, a subset of size l indeed consists of the top l values in the subtree rooted at u_i. We say that these l values are *proven* by u_i. During the execution of a proof-carrying plan, each node u_i proceeds in the following sequence of steps:

1. *Receive values from child subtrees.* From each child u_j of u_i, u_i receives $b(e_j)$ values, among which the top k_j values are proven by u_j.
2. *Sort.* Node u_i sorts all values received from its children as well as its own value, and determines the top $b(e_i)$ values to be passed up.
3. *Prove values.* Consider each value y among the top $b(e_i)$ values to be passed up from u_i. This value is proven by u_i if and only if for every child u_j of u_i, one of the following three conditions holds:

   (c.1) y comes from u_j and is proven by u_j.

   (c.2) There is a value y' proven by u_j, and $y' < y$.

   (c.3) $b(e_j) = |\mathit{desc}(u_j)|$, i.e., u_j passes up all values at or below it in the network.

   From the above, it is not difficult to see that if y is proven, then all values greater than y in the top $b(e_i)$ are proven as well.
4. *Return to parent.* The top $b(e_i)$ values are passed up to $\mathit{parent}(u_i)$. The number of values proven by u_i is also passed up if this number is less than $b(e_i)$; otherwise, all $b(e_i)$ values are assumed to be proven. This optimization saves the cost of transmitting this number from small subtrees that will likely prove all values they pass up, and completely eliminates the need for leaf nodes to transmit such numbers.

At the end of the execution, the root node returns a set of values as the answer, and additionally proves that the top k' among them are indeed the top k' values in the entire network. The rest of the values returned by the root may or may not be in the actual top k, but they may still be useful to the user. Furthermore, they can help us construct a more efficient "mop-up" plan to retrieve the missing actual top k values, as we will show later in this section.

The following lemma is immediate from the description of a proof-carrying plan:

Lemma 1 *The set of l values proven by a node in a proof-carrying top-k query plan are indeed the top l values in the subtree rooted at this node.*

We now describe how to formulate a linear program to optimize the bandwidth allocation for a proof-carrying query plan, so that in expectation, the largest number of top k values can be proven by the root. As before, we have the Boolean sample matrix $\mathbf{S}$. In addition, however, we need to keep the actual values in samples. Again, $\mathit{ones}(s)$ denotes the set of nodes with the top k values in the s-th sample. We also denote by $\mathit{smaller}_s(u_i, u_j)$ the set of nodes in the subtree rooted at u_j whose values are smaller than the value at u_i in the s-th sample.

As in PROSPECTORLP+LF, we use a variable $b(e_i)$ for every edge e_i in the network. However, we do not need a variable $a(e_i)$ indicating whether an edge is used by the plan. The reason is that a proof-carrying plan must use all edges in the network, or else we cannot even prove the top 1 value because any node not visited by the plan may actually have the largest value. We use an integer viable $x_s(u_i, u_j)$, where $u_i \in \mathit{desc}(u_j)$, which indicates whether the value of u_i is proven by u_j when the plan runs on the s-th sample. Let r denote the root node, and $\mathit{sibling}(u_i, u_j)$, where $u_i \in \mathit{desc}(u_j)$, denote the set of u_j's children who are not u_i's ancestors. The linear program is shown below:

$$(10) \quad \text{Maximize} \sum_s \sum_{u_i \in \mathit{ones}(s)} x_s(u_i, r), \text{ subject to:}$$

$$(11) \quad \textstyle\sum_i \big(\sigma_i + b(e_i)\delta_i\big) \le C;$$

$$(12) \quad \textstyle\sum_{u_i \in \mathit{desc}(u_j)} x_s(u_i, u_j) \le b(e_j) \quad \forall s, j;$$

$$(13) \quad x_s(u_i, \mathit{parent}(u_j)) \le x_s(u_i, u_j) \quad \forall s, i, j \text{ s.t. } u_i \in \mathit{desc}(u_j) \wedge u_j \ne r;$$

$$(14) \quad x_s(u_i, u_j) \le \textstyle\sum_{v \in \mathit{smaller}_s(u_i, u_k)} x_s(v, u_k) \quad \forall s, i, j, k \text{ s.t. } u_i \in \mathit{desc}(u_j) \wedge u_k \in \mathit{sibling}(u_i, u_j) \wedge \mathit{smaller}_s(u_i, u_k) \ne \varnothing;$$

$$(15) \quad 0 \le x_s(u_i, u_j) \le 1 \quad \forall s, i, j \text{ s.t. } u_i \in \mathit{desc}(u_j).$$

Line (10) specifies the optimization objective of maximizing the expected number of top k values proven by the root, over all samples.

Line (11) constraints the total amount of energy spent by the plan. The cost of sending the number of proven values is small and can be incorporated by conservatively reserving a fixed amount of energy for each non-leaf edge. We omit the details here.

Line (12) encodes the bandwidth constraint, analogous to Line (8) of PROSPECTORLP+LF.

Line (13) encodes the constraint that for any value to be proven by a node, that value must be proven by all nodes between this node and the node who owns the value.

Line (14) encodes the proof constraint. Namely, for a value y (from u_i) to be proven by a node u_j, every child u_k of u_j (except the child from which y comes) must prove some smaller value. The only exception to this constraint is when the child u_k actually returns all values in the subtree rooted at u_k (in which case all these values are proven by u_k), and none of these values is smaller than y. This exception is captured by the condition $\mathit{smaller}_s(u_i, u_k) \neq \varnothing$, which prevents the constraint to be generated in this case.

Line (15) captures the integrality constraints on $x_s(u_i, u_j)$ variables with linear relaxation (as previously discussed for Line (4) of PROSPECTORLP−LF).

Compared with PROSPECTORLP+LF, where the number of $x_s(u_i)$ variables it uses is on the order of the product of the number of samples and the size of the network, PROSPECTORPROOF uses one variable $x_s(u_i, u_j)$ for each sample and descendent-ancestor node pair, thereby increasing the number of variables in the linear program by another factor equal to the height of the network. Nevertheless, the overall size of the linear program remains polynomially bounded by the problem size. In our experiments, we have never run into situations where our linear solver takes too much time on realistic problem sizes.

From PROSPECTORPROOF to PROSPECTOREXACT We now show how to use PROSPECTORPROOF to construct a two-phase algorithm, PROSPECTOREXACT, which always returns the exact answer to a top-k query, just as does NAIVE-k. We show in Section 5 that PROSPECTOREXACT is performs considerably better than NAIVE-k. The basic idea is to use PROSPECTORPROOF as the first phase of PROSPECTOREXACT. If all top k values are proven by the root, we are done. Otherwise, we carry out a second, "mop-up" phase to retrieve the missing (or unproven) top-k values, using the information collected in the first phase to narrow down the search. The hope is that the first phase will produce a reasonable answer, so the second phase will be cheap or even unnecessary.

We assume that each node u_i remembers its own value as well as all values passed to it from its children in the first phase; let $\mathit{retrieved}(u_i)$ denote this set of values. Node u_i also remembers which of these values are proven at u_i; let $\mathit{proven}(u_i)$ denote this set of proven values. The second phase of PROSPECTOREXACT executes as follows at each node u_i:

1. *Receive request from parent.* Node u_i receives from $\mathit{parent}(u_i)$ a triple $(l, y_\vdash, y_\dashv)$, which requests u_i to return the top l values at or below u_i in the network within the range $(y_\vdash, y_\dashv)$.
2. *Forward requests to children.* Node u_i constructs a request $(l', y'_\vdash, y'_\dashv)$ as follows:
   - $l' = l - |\mathit{proven}(u_i) \cap (y_\vdash, y_\dashv)|$. Intuitively, u_i can use the proven values in the range requested by u_i's parent to service (at least partially) that request. Only the remaining portion of the request needs to be forwarded to u_i's children.
   - $y'_\vdash = \max(y_\vdash, y)$, where y is the l-th ranked value in $\mathit{retrieved}(u_i) \cap (y_\vdash, y_\dashv)$ if this value exists, and $-\infty$ otherwise. Intuitively, any other value must at least beat y in order to make it into the top l values in range $(y_\vdash, y_\dashv)$.
   - $y'_\dashv = \min(y_\dashv, \min(\mathit{proven}(u_i)))$. Intuitively, u_i should not need to request from its children any value larger than $\min(\mathit{proven}(u_i))$ because all such values must be in $\mathit{proven}(u_i)$ already.

   If $l' > 0$ and $(y'_\vdash, y'_\dashv) \neq \varnothing$, then u_i will broadcast this request to its children.
3. *Receive and return.* If u_i has requested from its children in the previous step, it receives the requested values and merge them with $\mathit{retrieved}(u_i)$. Finally, u_i returns the top l values in $\mathit{retrieved}(u_i) \cap (y_\vdash, y_\dashv)$ to its parent.

The root node r initiates the second phase by broadcasting the request $(k - k', y, \min(\mathit{proven}(r)))$ to its children, where y is the k-th largest value in $\mathit{retrieved}(r)$.

Note that we can further improve the algorithm in the second phase by sending to children requests with different bounds and numbers of desired values. The details are rather involved and we omit them here. In our experiments, we find the version presented above to be sufficient and effective in practice; further improvements will bring only marginal benefits for the problems we tested.

Note that PROSPECTOREXACT uses the knowledge about the distribution of sensor values for a different purpose from other PROSPECTORS. PROSPECTOREXACT uses this knowledge strictly for optimizing an algorithm that always returns an exact answer, similar to the purpose for which traditional database query optimizers use statistics. This knowledge does not need to be accurate in any way to guarantee correctness, although better accuracy yields better performance. In contrast, the other PROSPECTORS

cannot guarantee the accuracy of results produced by their plans, if their knowledge about the sensor values is inaccurate.

4.4 Other Features

Coping with Failures For simplicity, we have described the PROSPECTORs assuming a static network topology where all nodes and edges are always available. We now present techniques for coping with failures. We assume that permanent node failure is possible, but rare. As mentioned in Section 2, the spanning tree $\mathcal{T}$ can change over time as necessary. If a node is non-functioning for an extended period of time, $\mathcal{T}$ adjusts to exclude the node. The plan is then re-optimized based on the new topology.

On the other hand, we expect transient failures to happen frequently. Recomputing a new topology every time is expensive and ineffective, since the latest topology will always lag behind the latest failures. Instead, we can incorporate failures into optimization. Recall that we use a reliable communication protocol; in the event that a message is not delivered, we attempt to route it around the failure to the intended destination or its parent (which requires some additional state to be maintained at the nodes). We collect statistics on the frequency with which each edge fails, as well as the cost of re-routing around such edges. To incorporate this information into our optimization framework, we simply increase the cost of each edge by the product of its failure probability and the extra cost incurred by re-routing.

An alternative is to develop query plans that directly cope with transient failures during execution without using a reliable communication protocol. This approach has the potential of delivering better performance, and would be an interesting problem for future research.

Modeling Other Costs We have focused on radio transmission costs, because these dominate energy consumption. We have ignored non-radio costs, such as sensor reading acquisition and computation. It is straightforward to integrate them into our optimization framework. For example, we can add acquisition to the PROSPECTORs by adding its cost to the overall energy constraint and enforcing that in order for the root to acquire a node, the node must acquire a measurement.

Plan Dissemination To simplify presentation, we have ignored the costs of installing a query plan in the initial distribution phase and triggering its execution in subsequent distribution phases. Such costs can be incorporated in the linear programming framework. Furthermore, our experimental results do account for these costs.

Plan Re-calculation It is prohibitively expensive to disseminate a new plan every time conditions change. We can, however, often re-calculate the optimal plan at the base station. Only if this plan performs considerably better than the current one, do we disseminate it.

Re-sampling We need to sample the entire network periodically to gather input needed for optimization. When to re-sample depends on how confident we are in the accuracy of the current model for predicting top k. This confidence can be measured by periodically running PROSPECTORPROOF or PROSPECTOREXACT (instead of PROSPECTORs without proofs), which can tell us the accuracy of our approximate solutions. If the accuracy is not acceptable, the rate of re-sampling is increased.

5 Experimental Evaluation

We evaluate our algorithms using our own simulator of a network of Crossbow MICA2 motes [2]. Our simulator uses a generic MAC-layer protocol. We model only communication costs, as discussed in Section 2. The cost of queries that we report include both the cost of triggering query execution (assuming that the plan has already been stored in the network in the initial distribution phase) and the cost of the collection phase.

To create a sensor network, we start with a given rectangular space and a root node, place a number of nodes randomly within the space, and then, while adhering to mote radio distance limits, build a spanning tree over them where each node is as few hops from the root as possible.

Approximate PROSPECTORs We perform a comparison of a number of algorithms, plotting energy cost in mJ against accuracy of the query plan. Accuracy is measure as the percentage of actual top-k values returned by the query. Sensor values in this synthetic data experiment are drawn from independent normal distributions whose means and variances are chosen randomly from small ranges. The results are shown in Figure 3. ORACLE is a non-plausible algorithm that knows the exact location of the top k values beforehand; its cost serves a baseline for comparison of the approximate algorithms. For approximate top-k algorithm, we set $k = 40$ and vary the amount of energy budget. For exact algorithms (ORACLE, NAIVE-k, and NAIVE-1), we vary accuracy by changing k (e.g., $k = 36$ corresponds to 90% accuracy), and then measuring the cost of the plan. The relative order of algorithms is as predicted. NAIVE-k performs much worse than the others, supporting the case for approximate over exact solutions. We do not plot NAIVE-1 in this figure; its cost at $k = 4$ already matches the cost of NAIVE-k at $k = 40$, and grows linearly more expensive with increasing k. The three approximate algorithms in increasing order of performance are PROSPECTORGREEDY, PROSPECTORLP$-$LF, and PROSPECTORLP+LF. This ranking incrementally shows the benefits of topology-awareness and local filtering. We explore these features further in later experiments.

Variance We next explore the affect of variance in the distribution of sensor values on PROSPECTORLP$-$LF and PROSPECTORLP+LF. The means are again chosen within

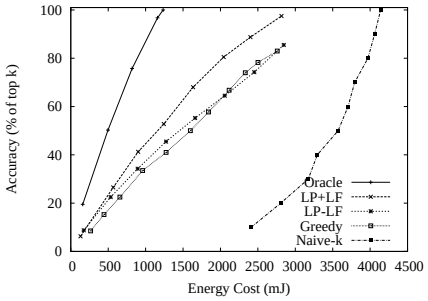

Figure 3. Comparison of algorithms.

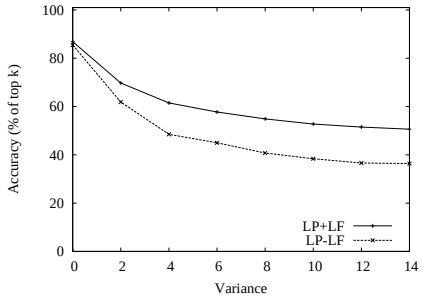

Figure 4. Effect of variance.

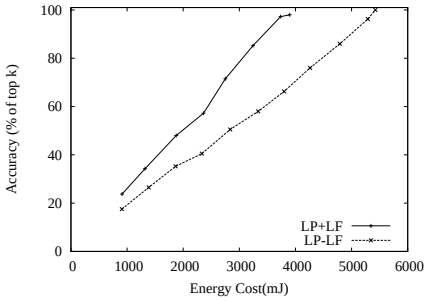

Figure 5. Contention zones.

a small range, while variance moves from low, where the top-k is very predictable, to high, where nodes have almost equal probability of ranking in the top k. The energy limit is fixed at a sufficiently high level over all trials to allow PROSPECTORLP+LF to achieve near perfect accuracy when variance is negligible. The results are shown in Figure 4. When variance is low, the algorithms perform similarly since they essentially know where the top k values are, and can visit just those nodes. As variance increases, performance degrades for both algorithms, but PROSPECTORLP−LF degrades faster. The reason is that it must still choose a limited number of values to acquire. In contrast, PROSPECTORLP+LF reacts to the increasing uncertainty by visiting more nodes and using local filtering to pass on the most promising ones. While it pays for visiting more nodes by not being able to send as many values all the way to root, the values it does send up are much more likely to be in the top k than those sent by PROSPECTORLP−LF. Finally, both algorithms level out once variance rises to the point that the mean values are diluted and all nodes have roughly equal probability of making the top k.

Local Filtering To better understand the impact of negative correlation on query planning, as introduced in our ornithology example from Section 1, we introduce a scenario where the network contains six "contention zones" spaced evenly around its perimeter with the query root in the center, as depicted in Figure 6. Each contention zone contains k nodes. Nodes outside the zones have fixed means, μ, and low variances. Nodes inside the zones have lower means than μ, but higher variances such that each of these has an identical $1/6$ chance of exceeding μ. This means the total expected number of zone nodes above μ is k. Each zone is expected to provide $k/6$ of the top k values. We again measure energy cost against accuracy, as shown in Figure 5. PROSPECTORLP+LF greatly outperforms PROSPECTORLP−LF. PROSPECTORLP−LF makes the mistake of acquiring all values from a single zone before visiting another. Remember that all zone nodes have equal probability of making the top k, and appear to be of equal benefit in sampling. If PROSPECTORLP−LF pays

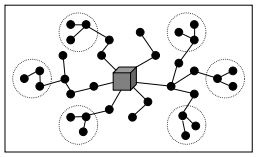

Figure 6. Contention zones.

the initial cost of traversing the network to reach a particular zone, it is cheaper to continue accessing its nodes than those in another zone. For the same amount of energy, PROSPECTORLP+LF can visit several zones and all nodes within them, but locally filter out only the highest $k/6$ values from each. PROSPECTORLP+LF gains advantage as allocated energy increases. This intuitively makes sense; each additional value PROSPECTORLP−LF acquires has only a $1/6$ chance of making the top k. PROSPECTORLP+LF allocates additional energy more intelligently using local filtering for greater benefit.

We continue this experiment by starting with the above scenario and varying the number of zones, and adjust the fraction of nodes within each that are greater than μ accordingly. For example, when there are 4 zones, each still has k nodes, but the probability of a node being greater than μ rises to $k/4$. The energy limit is set to a level shown to create a large performance gap between PROSPECTORLP−LF and PROSPECTORLP+LF, as determined in the previous experiment. The result is shown in figure 7. As the number of zones increases, the penalty that PROSPECTORLP−LF suffers for acquiring an entire zone increases, since a lower percentage of the top k values are in any zone. Both algorithms degrade in performance because as the number of zones increases, each one must visit more zones and pay a higher cost to acquire the same number of top k values.

Intel Lab Data We evaluate our algorithms on a dataset built from the Intel Berkeley Research Lab [7], consisting of environmental readings over a series of time epochs collected by 54 motes spread around their lab, from which we have extracted temperature readings. In addition, because we are interested in network topology, and the lab area is not large enough to force much hierarchy on the spanning tree, we shorten radio range to 6 meters, the minimum that

still allows for a fully connected tree. Lastly, the data set is missing values in various epochs. We have filled in a missing value at a given node and epoch with the average of the node values read at the prior and subsequent epochs.

We test approximation algorithms using the first 100 epochs as samples and run queries on the following data with $k = 15$. We plot energy cost versus accuracy for the approximate PROSPECTORs. The results for PROSPECTORLP−LF and PROSPECTORLP+LF are nearly identical. Therefore, in Figure 9, we only show results for PROSPECTORGREEDY and PROSPECTORLP−LF. This result shows topology considerations have an impact on performance, but local filtering does not. PROSPECTORGREEDY does not perform as well as PROSPECTORLP−LF until they reach energy levels where both acquire nearly all of the top k values. The lack of impact for local filtering stems from the fact that the locations of the top k values are fairly predictable in the dataset. In this case, PROSPECTORLP−LF is clearly sufficient to build an efficient query plan. As would be expected with this data set, approximation methods greatly outperform NAIVE-k. Although not plotted here, to acquire the top 9 values (60%), NAIVE-k requires over 800mJ, more than three times what the approximates need for near 100% accuracy.

PROSPECTORPROOF We next evaluate the effectiveness of PROSPECTOREXACT by comparing it to the exact algorithms, NAIVE-k and ORACLEPROOF. ORACLEPROOF, similar to Oracle, knows the locations of the top k values, but still accesses all nodes to provide a proof for the solution. It serves as a baseline for exact algorithms. A new baseline is needed here because exact algorithms must visit all nodes, and therefore cannot compete with the energy savings of the approximate ones. So while we propose PROSPECTOREXACT as a very efficient algorithm for getting an exact solution, we acknowledge that due to unavoidable high overhead cost, the savings versus NAIVE-k are of a much smaller magnitude than those achieved by the approximate PROSPECTORs.

Results for this experiment are shown in Figure 8. PROSPECTOREXACT is a two-phase algorithm, and we plot the breakdown of costs in two phases. NAIVE-k and ORACLEPROOF are single-phase algorithms, so their costs are fixed in this setting and plotted as horizontal lines. Recall that the goal of PROSPECTOREXACT is to plan the first phase so that the second one is unneeded or requires very little energy. When the first phase is allocated smaller amounts of energy, the second phase is necessary and expensive, resulting in poor performance. When energy is higher in the first phase, the cost of the second phase is low, but the first phase acquires more values than needed. The optimal point falls in the middle. Accepting the limited potential savings we can achieve, we do find that the optimal point realizes roughly 70% of the possible improvement between NAIVE-k and ORACLEPROOF.

Other Results We have omitted the full details of our investigation on the performance impact of sampling size. Under conditions similar to those for Figure 3, we find that, as expected, using just a single sample results in very poor accuracy. With only a small increase to 4–5 samples, accuracy increases dramatically and then essentially levels out, only increasing very slightly up to 25–30 samples. Experiments on the Intel Lab data show a similar trend, with benefit of additional sampling leveling out around 5 samples.

We also mention here that cost of installing the query plan in the initial distribution phase, which requires unicasting instructions to each node involved in the plan, is on the order of the cost of one collection phase. Following our assumption on install-once, run-many-times usage, this cost is reduced in significance when amortized over many subsequent runs of the same query.

While time spent solving linear programs had no impact on our cost measurements, it is worth mentioning these results. We ran our experiments using ILOG CPLEX 8.1 on a 650MHz desktop computer. Running time was very dependent on the energy constraint, and was significantly slower at levels where there were many plans of similar value. In the worst cases, given a program with 200 nodes and 50 samples, PROSPECTORLP+LF finished in 30 seconds and PROSPECTOREXACT finished in 120 seconds. In better cases, these often took just a few seconds.

6 Related Work

A substantial amount of work has been done on querying sensor networks. The general question of what type of queries should be done on networks, such as event-based or periodic, is explored in [11] and [15]. Many papers focus on saving energy and extending the life of the network. One main strategy is to do at least some query work in-network, as suggested by [12] and [15]. In-network aggregation reduces overall energy consumption by performing computation within the network and reducing the size of transmissions propagated upward.

A second main strategy for conserving energy is using approximation. The idea of using models to encode the relationships between sensor values at different motes, as well as between different types of sensors, is introduced in [4]. Techniques for using such models are further explored in [3]. These models, however, are mainly directed at selection-type queries. Several approaches for extending the useful lifetime of the network are based on deploying a redundant amount of motes, where a subset of them can represent the whole collection. The "connected k-coverage" problem, presented in [17], finds a minimal set of nodes that are fully connected and can stand for the entire set of nodes, given approximation limitations. Snapshot queries take a similar approach by choosing a subset of nodes to stand if

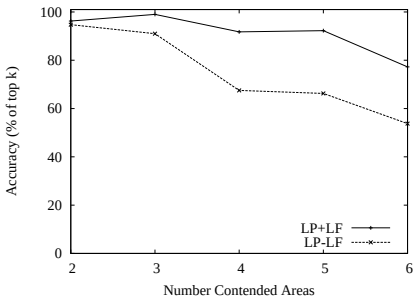

Figure 7. Varying # of zones.

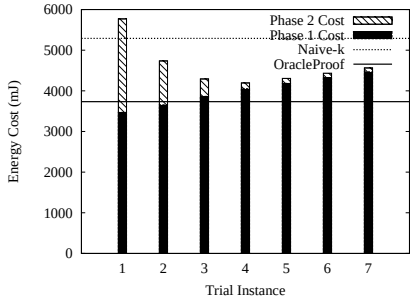

Figure 8. PROSPECTOREXACT**.**

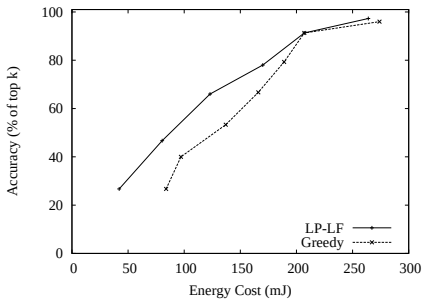

Figure 9. Intel Lab data.

for their neighbors [8]. The resolution of the query can be adjusted by involving more nodes. The set of nodes that is used in querying can be adjusted depending on which nodes have the highest remaining energy resources.

Other papers also investigate specific strategies for solving more complicated queries than selection. [14] proposes methods for computing approximations of quantile aggregates such as median and mode . They present a structure, q-digest, which nodes use to summarize the data they have receive into a fixed size message, with approximation bounds. [6] suggests linear regression as a means to develop a model covering measurements at all nodes. When a query is performed, the network does not transmit large numbers of values to convey measurements from all nodes, but instead transmits small numbers values that constrain the parameters of the model, thereby providing information that can be used to derive the measurements.

There is considerable work on other top-k variations. Babcock and Olston discuss the problem of monitoring the top-k data objects within a distributed system [1]. In their setting, an object's score is the sum of its local scores (one from each node). Our problem ranks the values from each node. Additionally, theirs is a continuous query, while ours is periodic. A closer parallel to [1] for sensor networks is [16]. Scores for the same object can accumulate at different sources, and they leverage that an object might easily rank in the top k in one part of the network to prop up its ranking in other parts. This technique can be used to maintain an overall top-k with some confidence, and avoid recomputing it as often as would otherwise be necessary.

7 Conclusions

We have presented sampling-based query planning as a means to leverage the advantages of modeling network behavior, while avoiding the sometimes prohibitive costs of dealing explicitly with models. Sampling maintenance is also quite simple. Sampling and linear programming complement each other well to form an optimization framework. This framework lets us constrain query plans to a user-defined energy budget. We apply the framework to the top-k query problem and create the Prospector series of algorithms, which integrate topology-awareness and local filtering into planning, and provide approximation solutions, as well as a "proof" version that can be extended to an exact solution. Our evaluation shows the impact of various conditions on our algorithms. Among the points we make are that we can realize large energy savings with approximate over exact solutions while achieving high accuracy levels, and that local filtering is a more advantageous feature the more a network contains areas of contention in which subsets of nodes have top-k values. These confirm that the sampling-based framework does achieve the energy-saving benefits of the modeling-based approach, while affording advantages in ease of calculation.

The basic techniques and framework in this paper provide an innovative way of interacting with sensor networks. We have also presented a number of extensions for coping with some of the realistic issues that arise with sensors, such as frequent failure. We foresee two interesting lines of research for making our techniques more adaptable to real-world conditions. The first is to model failure and other complications and integrate them into optimization. The second is to build more flexible plans that leverage actual network conditions once they are observed during query execution.

References

[1] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, San Diego, California, USA, June 2003.

[2] Crossbow Inc. *"MPR-Mote Processor Radio Board User's Manual"*.

[3] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting correlated attributes in acquisitional query processing. In *Proc. of the 2005 Intl. Conf. on Data Engineering*, Tokyo, Japan, Apr. 2005.

[4] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, Toronto, Canada, Aug. 2004.

[5] R. Gallager, P. Humblet, and P. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Languages and Systems*, 5(1):66–77, 1983.

[6] C. Guestrin, P. Bodik, R. Thibaux, M. Paskin, and S. Madden. Distributed regression: an effi cient framework for mod-

eling sensor network data. In *Proc. of the 2004 Intl. Conference on Information Processing in Sensor Networks*, Berkeley, California, USA, Apr. 2004.

[7] Intel Berkeley Research Lab. `http://berkeley.intel-research.net/labdata/`.

[8] Y. Kotidis. Snapshot queries: Towards data-centric sensor networks. In *Proc. of the 2005 Intl. Conf. on Data Engineering*, Tokyo, Japan, Apr. 2005.

[9] N. Littlestone and M. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, 1994.

[10] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *Proc. of the 2002 USENIX Symp. on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, Dec. 2002.

[11] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, San Diego, California, USA, June 2003.

[12] M. Sharaf, J. Beaver, A. Labrinidis, and P. Chrysanthis. Balancing energy effi ciency and quality of aggregate data in sensor networks. *The VLDB Journal*, 13(4):384–403, 2004.

[13] D. Shmoys and C. Swamy. Stochasitc optimization is (almost) as easy as deterministic optimization. In *Proc. of the 2004 IEEE Symp. on Foundations of Computer Science*, Rome, Italy, Oct. 2004.

[14] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *Proc. of the 2004 SenSys*, Baltimore, Maryland, USA, Nov. 2004.

[15] Y. Yao and J. Gehrke. Query processing for sensor networks. In *Proc. of the 2003 Conf. on Innovative Data Systems Research*, Ansilomar, California, USA, Jan. 2003.

[16] D. Zeinalipour-Yazti, S. Neema, D. Gunopulous, V. Kalogeraki, and W. Najjar. Data acquisition in sensor networks with large memories. In *Proc. of the 2005 IEEE Intl. Workshop on Networking Meets Databases*, Tokyo, Japan, Apr. 2005.

[17] Z. Zhou, S. Das, and H. Gupta. Connected k-coverage problem in sensor networks. In *Proc. of the 2004 IEEE Intl. Conf. on Computer Communications and Networks*, Chicago, Illinois, USA, Oct. 2004.