Approximating Sensor Network Queries with In-Network Summaries

Alexandra Meliou
Carlos Guestrin
Joseph M. Hellerstein

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-137
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-137.html

October 21, 2008

Copyright 2008, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Approximating Sensor Network Queries Using In-Network Summaries

Alexandra Meliou [#1], Carlos Guestrin [*2], Joseph M. Hellerstein [#3]

[#] *EECS Department, UC Berkeley*

[1] ameli@cs.berkeley.edu

[3] hellerstein@cs.berkeley.edu

[*] *SCS, Carnegie Mellon University*

[2] guestrin@cs.cmu.edu

ABSTRACT

In this work we present new in-network techniques for communication-efficient approximate query processing in wireless sensornets. We use a model-based approach that constructs and maintains a spanning tree within the network, rooted at the basestation. The tree maintains compressed summary information for each link that is used to "stub out" traversal during query processing. Our work is based on a formal model of the in-network tree construction task framed as an optimization problem. We demonstrate hardness results for that problem, and develop efficient approximation algorithms for subtasks that are too expensive to compute exactly. We also propose efficient heuristics to accommodate a wider set of workloads, and empirically evaluate their performance and sensitivity to model changes.

1. INTRODUCTION

Query processing has received significant attention in recent research on wireless sensor networks ("sensornets") [13]. As is well known, communication is one of the most expensive operations in a sensornet [2], so various query processing techniques have been proposed to minimize communication. Model-based data acquisition is a particularly promising approach to address this issue [6]. It uses historical information gathered from the network to predict rough query answers from a probabilistic model, and decides which data is worth gathering from the network at query time to augment that model sufficiently to meet a desired accuracy bound. Given a chosen set of data to gather, path planning algorithms [17] compute an efficient distributed strategy to retrieve the data needed to augment the model. This model-based approach relies upon information stored at a central location. Centralized models can make accurate predictions – assuming that the centrally maintained model is accurate – but suffer from unexpected events like node and link failures in the network. Given the distributed nature of sensornets, a natural advance upon the early model-based work is to move this processing into the network.

In this model-based setting, the focus is on providing approximate answers to queries. Approximate queries are well suited to sensornet settings because perfect accuracy is both hard to achieve, and typically unnecessary. Physical sensors provide a spatially discrete and thus approximate view of the continuous physical phenomena they are used to monitor. Physical sensing also induces measurement errors, due to device imperfections, calibration problems, and physical stresses like dirt and heat. As a result, most users of sensornet applications understand that total accuracy of reported values is unnecessary. Approximate queries allow some uncertainty in the reported result, typically characterized by two parameters: a window w of accuracy for the answer and a confidence limit δ that is expected to be satisfied. An example of such a query is the following: "Return the temperature at each sensor node, within $\pm 1°C$ with 95% confidence". Our focus will be on these "`SELECT *`" queries, that return a reading from each of the sensors in a field.

Our work adopts a model-based approach to approximation [6], but moves the implementation to an in-network setting, where the models can stay more easily up-to-date, pre-processing of routing paths at query runtime can be eliminated, and failures on routing paths are no longer crucial. We propose the use of a carefully designed in-network spanning tree to minimize the communication required to return a robust estimate of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.

value reported by each sensor node. This tree is rooted at a base station, with Gaussian models stored at each node in the tree, one per child of the node. Queries are answered by traversing to a depth in the tree where the summaries provide sufficient information to answer a query within a specified window of accuracy.

We break this problem into individual tasks which we discuss in the corresponding sections. First, we provide a formal definition of the problem of optimal in-network summaries. Second, we present an efficient Gaussian-based compression scheme that is geared towards minimizing erroneous reporting of values, which can be optimized based on query workload. Third, we present query traversal algorithms that utilize the compression scheme to make routing decisions and give value estimates with limited communication. Our work is grounded in formal hardness results for the optimal in-network summary problem, along with approximation algorithms for a basic query processing scenario. Given this formal basis, we expand the set of scenarios we consider to a more practical setting, using a set of intuitive heuristics. We conclude with an experimental evaluation of the sensitivity of our approach to changes in the data and query workload.

1.1 Paper Outline

This paper is organized as follows: In Section 2 we define our in-network summaries, discuss their optimality and move on to Section 3 to talk about how they should be compressed to meet storage constraints. Section 4 discusses how sensornet queries can benefit from a given in-network summary scheme to run more efficiently, and our query traversal algorithm is evaluated against an optimal strategy. In Section 5, we use the results of Sections 2, 3 and 4, to explore the space of in-network summaries, formally define the problem of finding the optimal one, and give hardness results and approximation algorithms. The algorithms are extended to a distributed setting and are evaluated against their centralized versions. Section 5.4 further expands the results to optimize the summary structure for a variety of workloads, and an evaluation of the scheme is given against a larger workload range. Finally, Section 6 presents an evaluation of the sensitivity of in-network summaries to changes in the underlying data.

1.2 Related Work

The idea of Semantic Routing Trees was proposed in [15] as an overlay index in the network, to allow for routing decisions to those leaves relevant to the query. Our in-network summaries go further, maintaining summaries of the data at different tree levels to allow for reduction in communication. The use of summaries for query processing has been examined in different settings. GHTs ([19]) propose storing and retrieving network information using Geometric Hash Tables, and Distributed Quadtrees ([5]) overlay quadtree structures over WSNs to satisfy distance sensitive spatial queries.

Our in-network summaries aim on using models distributed in the network to make query processing more cost efficient. The same objective was tackled by centralized approaches: In [6], the BBQ system is presented, which proposes a model-driven scheme to provide approximate answers to queries posed in a sensor network, satisfying some information guarantees. [17, 18] also focus on a centralized approach, where all the decisions and planning are performed at a basestation node, and heuristics are given to cope with unexpected network behavior.

The TinyDB system [16], which is largely used for data collection in sensor networks, uses spanning trees for the data retrieval, but does not rely on any other in-network data to optimize queries. Maintaining data in the network is the focus of distributed storage ([7]). Several different coding schemes are developed for storing information in the network, but the goal in this case is to make the data resilient to failures and not to optimize query execution. A similar tactic is employed by the SPIN protocol [10, 12], which disseminates data in the network, so that a user posing a query at different locations can immediately get back results.

In terms of data gathering, directed diffusion ([11]) is a data centric approach that sets up gradients from data sources to the basestation, forming paths of information flow, which also perform aggregation. Rumor Routing ([4]), uses long lived agents that create and redirect paths to events they encounter.

In this work we will focus on "`select *`" type queries, but there has been significant work on in-network aggregation. TAG ([14]) implements simple aggregates using the initializer, merging and evaluator functions. Aggregation in adversarial settings is examined in [9].

2. OPTIMAL IN-NETWORK SUMMARIES

In this section we focus on the type of data summaries that we need to maintain to optimize queries in a sensornet setting and give a definition of optimality for that scheme. We then define the problem of data compression and treat it from the standpoint of the query workload.

Our workload consists of "`SELECT *`" style queries that request the approximate values of multiple sensors, without aggregation. The approximation bounds are defined by an accuracy window w and confidence limit δ. For a query that has parameters (w, δ), and cardinality k, then on expectation, δk true answers will be within $\pm w$ of the reported answers. Depending on the values of w and δ, a query can range from being very loose to very

strict in terms of accuracy. The smaller the window, and the higher the confidence, the stricter the query becomes, demanding more accurate results.

In order to answer queries in a sensor network setting using information residing in the network, we introduce "in-network summaries", or "spanning summary trees". An in-network summary is a spanning tree of the network, in which every node stores a model of the data in each of the subtrees it points to. As the subtrees become smaller in the lower levels of the hierarchy, the models become finer and more precise. A query using that hierarchy can explore the structure starting from the root and going only as deep as is necessary to provide answers of good quality. We want to optimize this tree structure with the objective of minimizing the communication cost of answering queries. The guarantee that we want to achieve is that for n data nodes the query will produce at least δn answers that are within w distance of their actual value.

DEFINITION 1. *An* in-network summary *(or spanning summary tree) over a network graph $G(V, E)$ is a spanning tree $T(V, E')$, $E' \subseteq E$, augmented with models M_v, $\forall v \in V$. M_v is stored in the node p that is the parent of v, i.e., $(p, v) \in E$.*

We are given a network graph $G(V, E)$ and a query workload $W = \{Q_i(w_i, \delta_i)\}$. We will use M_v to symbolize the model information kept for node v. Then the optimization problem of finding the optimal in-network summary can be defined as follows:

Given: Graph $G(V, E)$, query workload $W = \{Q_i(w_i, \delta_i)\}$
Find: Tree $T = G'(V, E')$ and models M_v, $\forall v \in V$, such that the average communication cost required to retrieve values to respond to $Q_i \in W$ is minimized

Our design choices for the *in-network summaries* should preserve the following requirements:

Compactness: Due to storage limitations at the sensor level summary models M_v need to be small.

Informativeness: The summaries should be as informative as possible so that queries can be answered by accessing as few of them as possible.

Accuracy: The summaries should not lead to inaccurate query answers, i.e. the confidence bounds returned by examining summaries should correctly reflect the probability of the answers falling within the accuracy window.

3. MODEL COMPRESSION

To get started, in this section we focus on one part of the in-network summary problem: the construction of models at each node. For this discussion, we assume temporarily that the structure of the tree T is given, and we want to pick the best model M_v, $\forall v \in T$. We will revisit the structure of T in Section 5.

Many models could be maintained in a spanning summary tree, but in keeping with prior work we assume a gaussian (normal) model. For the modeling of input *readings* gaussian models are typically appropriate, since they successfully capture the nature of noisy measurements of physical phenomena ([3]). Therefore, for all leaves in T the model M_v will be a gaussian distribution based on observations of that node's measurements. Now, when a data summary needs to represent multiple sensors, a natural extension is to use gaussian mixtures, which can be of restricted size to comply with the storage limitations that sensor nodes have. A gaussian k-mixture refers to a mixture of k gaussians.

Assuming for simplicity that T has fixed fanout F, data resides at the leaves represented by the single gaussian distributions, and every internal node keeps F pairs of (childptr, gaussian k-mixture), then our "data structure" closely resembles a database index. Given a specified spanning tree T, we can imagine "bulk loading" the models M_v with a bottom-up construction, combining gaussian mixtures as we climb up the hierarchy. In this approach, mixtures high in the tree will be quite large. Since we need to keep the size restricted, we need to have a method for compressing the mixtures, otherwise called "collapsing".

In the problem of compression, our input is a mixture (set) of l gaussian distributions, and our output a k-size mixture, $k < l$. The parameter k is dictated by the amount of storage space assigned to the model on every node, and it is not required to be the same on each one of them. We will first address the problem for the simple case of $k = 1$, assuming that the summary hierarchy keeps a single gaussian distribution as a model of all the sensors in each subtree. In Section 5.5 we will revisit the compression model and explore generalizations to larger k.

During query execution, the only information that we have for a certain subtree is its collapsed distribution. In the case of a single-gaussian compression model ($k = 1$) the answers that this summary in the tree can provide is to report the mean μ of the summary as the answer for all nodes represented by the subtree. Intuitively, our goal is:

Given: A set of distributions $\mathcal{S} = \{N(\mu_i, \sigma_i^2)\}$
Find: Distribution $N(\mu, \sigma^2)$ that maximizes informativeness and accuracy over the original set $\mathcal{S}$.

In the above description, informativeness and accuracy are not clearly defined. Before we can proceed, we need

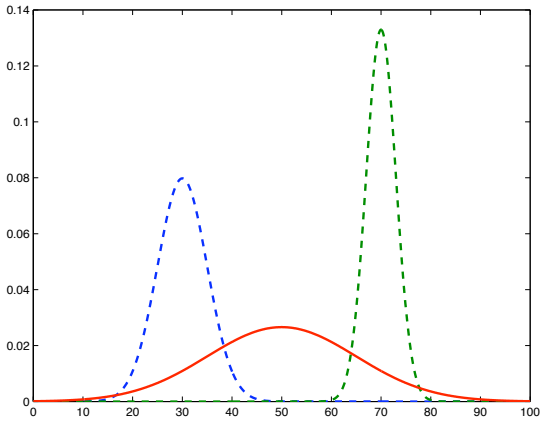

Figure 1: Two distributions (dashed lines) representing values of 2 sensor nodes, that happen to have no overlap. Collapsing using KL divergence produces a distribution (solid line) that contains significant mass in an interval that the original distributions contained almost none.

to formalize these concepts in a way that captures the approximation guarantees we provide for queries.

To understand the meaning of these terms in this context, some intuition is useful. Consider the following natural but misleading example. A common approach in collapsing gaussian mixtures is to minimize some distance function (e.g. KL divergence) of the collapsed distribution from the original mixture. While intuitively this might seem to capture the quality of the compression, such an approach can actually lead to very bad decisions for our problem.

In the example of Figure 1 we want to collapse the 2 distributions depicted by dashed lines into one. If our objective is the minimization of a metric like KL divergence, the result would be the solid line in Figure 1. The resulting distribution minimizes the distance from the original ones, but has the following pitfall: it falsely reports some significant mass on the interval $[45-60]$ where the original distributions contained almost none. In our setting this can in some cases lead to false query results for both of the original sensors involved. A query with a large window may not suffer from this side effect, but one with a window small enough to fit in the problematic interval (e.g. width 10) could produce erroneous results. Thus we have loss of accuracy.

Looking at the case above, we would have gotten better results had we just kept one of the original distributions intact and tossed the other one. In that case we would have known that at least one of the nodes would always be reported correctly. The important observation here is that whether the answer would be wrong – and how wrong – depends on the specific query. This observation suggests that the collapsing be targeted to a specific query workload. This is the approach we follow below.

Alternatively, one can argue that in order to avoid loss of accuracy, the variance of the collapsed gaussian could be adjusted so that it will not contain any "fake" mass. But such an approach would result in an extremely wide and flat distribution, which would be too close to the x-axis of Figure 1 to be visible. Such high uncertainty would be useless in query execution, as this summary could not produce answer estimates that would fall in a plausible query's accuracy window. In this case we have loss of informativeness.

Now that we gave an intuition of what informativeness and accuracy mean in this definition, we can proceed to give an appropriate compression scheme.

3.1 Simple Collapsing

During compression we want to preserve as much information from the original distributions as possible. This means that the new distribution should contain as much "real mass" as possible, and this will happen if it is centered at the location that contains the most mass from the underlying distributions. This location however depends on the window in which we compute the mass. In Figure 1, if the window used is fairly large, then the location that maximizes the total mass will be centered somewhere in between the two original distributions. On the other hand, if the window is small, then the location that maximizes the total mass would be centered at the narrowest of the original distributions. Therefore, compression of a given set of distributions will have a different result depending on the window assumed.

This also directly translates to query answers. In order to better understand our requirements for the collapsing, we need to take a look at how the collapsed distribution will be used to answer a given query. Assume a collapsed distribution $N(\mu, \sigma^2)$. $Q(w, \delta)$ is a query with error allowance inside a window w and confidence requirement δ. Q can be answered in a satisfactory way by distribution N, if the mass of N in the interval $[\mu - w, \mu + w]$ is greater or equal to the confidence requirement δ, i.e. $M_{[\mu-w,\mu+w]} \geq \delta$. In that case the query will report the value μ for both nodes.

As it is obvious from Figure 1, if we choose N as depicted by the solid line, then for small values of w the query may apparently pass the mass test, but the response will be wrong, as neither of the original nodes has value $\mu \pm w$ with δ confidence. However, if the window w is large enough, the distribution N may be sufficient to answer Q without problems. In order to make sure that that answer will be correct based on the query requirements we need to make sure that the mass

of the collapsed distribution in the interval $[\mu-w, \mu+w]$ is the same as the total mass of the original distributions in the same interval. This observation leads directly to a simple approach for collapsing.

The two requirements that we have for collapsing are that it should not introduce "fake mass" (high accuracy), and it should retain as much "real mass" as possible (high informativeness). Of course that optimization makes sense for a specific choice of window, and it gives the guarantee that queries with the same window will get accurate response. Mathematically, the problem we want to solve is as follows:

Given: One dimensional function f representing a probability distribution (in our case a gaussian)

Find: z such that the mass of f in the interval $[z - w, z + w]$ is maximized.

$$\max_z \int_{z-w}^{z+w} f(x)dx \tag{1}$$

Once the optimal location for z is chosen, this becomes the mean of the collapsed distribution. The variance of the new distribution will be calculated based on the mass of the original distributions in the same interval:

$$\int_{\mu-w}^{\mu+w} N(\mu, \sigma^2)dx = \sum_i \int_{\mu-w}^{\mu+w} N_i(\mu_i, \sigma_i^2)dx$$

3.1.1 *Location Of Maximum Mass*

The solution to the maximization given in equation (1) cannot always be determined analytically. One approach is to numerically evaluate it through the use of sliding windows: given window w, "slide" the interval $[z_i - w, z_i + w]$ across the x-axis calculating the mass of the distribution for every location z_i and pick the one with the maximum value. Obviously the quality of the result will depend on the fineness of the discretization $D = \{z_i\}$ used for the sliding.

Another approach is to use gradient ascent with the means of the original distributions as starting points. The reason to try this is that for continuous functions , the optimal location of the window is required to contain at least one local maximum. (It is easy to show that if it doesn't, sliding the window to one direction would increase the contained mass.) However this approach is not guaranteed to find the optimal solution either, and one can construct adversarial examples where that happens. In Figure 2 we compare the two approaches of sliding window and gradient ascent, which are shown to perform equivalently, with the sliding window algorithm winning by a small margin.

For this experiment we used real data from the Intel Berkeley Lab deployment [6] to compute gaussian models for 54 nodes, and used them to compare the two algorithms for different window sizes. The discretization used for the sliding window algorithm was a $D = \{z_i\}$ where $z_{i+1} - z_i = 0.001$ In conclusion, the sliding window technique with a modest discretization produces equivalent results as gradient ascent, when at the same time being a lot more computationally efficient.

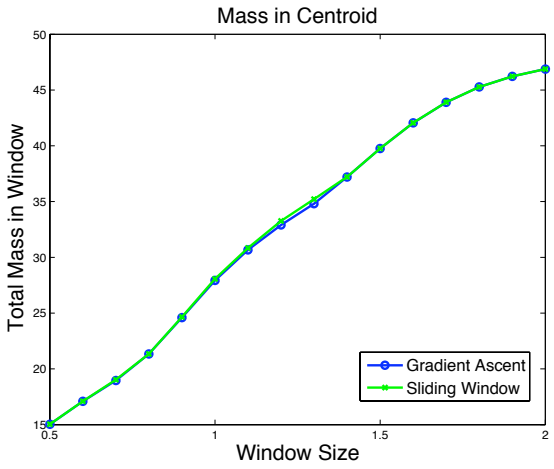

Figure 2: Comparison of the sliding window and gradient ascent algorithms

3.2 Tail-aware Collapsing

One thing to note is that the compression algorithm described in Section 3.1 suffers from accuracy problems in the case of recursive collapsing. Recursively applying simple collapsing to compress distributions with one another, we run the risk of introducing "fake mass" and thus reducing accuracy. Even though simple collapsing does guarantee that the mass inside window w of its mean ($[\mu - w, \mu + w]$) is accurate (i.e. corresponds to real mass from the original distributions), there is no guarantee for parts of the interval $I \subset [\mu - w, \mu + w]$.

The same is true for the tails of the distribution: the intervals $[-\infty, \mu - w] \cup [\mu + w, \infty]$. Recursively collapsing distributions can therefore cause errors due to the introduced fake mass.

Even though the mass inconsistencies inside subintervals of $[\mu - w, \mu + w]$ are not simple to deal with, the possibly problematic tails are easy to fix with *tail-aware collapsing*. Tail-aware collapsing performs compression exactly the same way as simple collapsing, but simply disregards any mass that exists in the tails of the distributions. Note however, that this approach can err in the opposite direction: it avoids the introduction of fake mass through the tails, but may disregard real mass that may have actually existed in the tails. Therefore, compared to simple collapsing, tail-aware is more con-

servative.

4. QUERY TRAVERSAL

In the previous section we discussed optimal compression of a set of input distributions based on an expected query workload. Sensor nodes organized in a tree structure can create an in-network summary by recursively performing compression bottom up, from leaves to root. In this section, we focus on the problem of routing queries using an in-network summary, making routing decisions at each node based on the local model M_v. First we will discuss what the optimal traversal would be for query $Q(w, \delta)$ on tree $T = G(V, E)$. A traversal is a connected component of G that contains the basestation. Since G is a tree, every connected component in it will also be a tree. The optimal traversal is the traversal of minimum total cost that can *satisfy* query Q on expectation, as defined in Definition 2

DEFINITION 2 (QUERY SATISFACTION). *In a network of n nodes, a response $R = \{r_1 \dots r_n\}$ to a query $Q(w, \delta)$ is said to satisfy Q if the actual values of at least δn nodes fall in their respective interval $[r_i - w, r_i + w]$.*

Due to linearity of expectations, assuming that the underlying distributions are correct, then the query will be satisfied if $\sum_i \int_{r_i-w}^{r_i+w} f_i(x)dx \geq \delta n$. So the optimal traversal problem can be defined as follows:

Given: tree $T = G(V, E)$ and node models $M_v, \forall v \in V$

Find: $G'(V', E')$, $E' \subseteq E$, such that $\sum_e Mass(M_u, w) \geq \delta n$, where $e = (u, v)$ with $u \in V'$ and $v \in V \backslash V'$

In the above problem statement, $Mass(M_u, w)$ is the maximum mass that can be contained inside window w from model M_u. In the case where the mixture model is compressed down to a single gaussian, $Mass(M_u, w) = \int_{\mu-w}^{\mu+w} f(x)dx$, where f is the normal probability density function.

4.1 DP Traversal

We will solve the optimal traversal problem using a dynamic programming algorithm. Every node will keep a DP table (in our case it's just a vector) which will hold information on forwarding decisions based on assigned budget. For example, if v is the root of subtree T_v and C_{T_v} is the cost of traversing the entire subtree T_v, then v will keep a vector of length C_{T_v} where every entry will be the maximum mass that can be collected by assigning the corresponding budget to that subtree. For a tree of fanout F, children $u_1, \dots, u_F$ of node v and a budget assignment $B = \{b_1, \dots, b_F\}$ among the F children, the DP function for computing the entry $J_v(c)$ will be:

$$J_v(c) = \max_B \sum_{i=1,b_i>0}^{F} J_{u_i}(b_i) + \sum_{i=1,b_i=0}^{F} Mass(M_v, w)|T_{u_i}| \quad (2)$$

where $|T_{u_i}|$ corresponds to the number of nodes represented by the subtree of node u_i. The second summation in (2) gives a mass estimate for the unvisited children ($b_i = 0$) based on the model of node v. Also, $\sum_i b_i = c - \sum_{i \ s.t. \ b_i>0} w_{(v,u_i)}$, where $w_{(v,u_i)}$ the weight of the edge (v, u_i), and the sum subtracted in this formula adjusts the available budget by the cost of reaching those children of v with non-zero budget assignments. The weight of each (v, u_i) edge can be simply defined as the number of hops needed to reach that child. Along with the DP vector, the choice of best budget assignment for each cell should also be kept. At the leaves of the tree, the DP vector is a single element, $J(0) = Mass(M_v, w)$.

Algorithm 1 gives the outline of this dynamic program implemented for the case where $F = 2$, a binary tree, and unit edge weights. The code can be easily extended to the more general case of fanout F or even unrestricted fanout.

Algorithm 1 DPConstruct(v,w,B)

```
1:  if B < 0 then
2:      return 0
3:  end if
4:  if B == 0 then
5:      J(B) = Mass(M_v, w)
6:  else
7:      for k = 0 ... B do
8:          lMass(k) = DPConstruct(u_1, w, k − 1)
9:          rMass(k) = DPConstruct(u_1, w, B − k − 1)
10:     end for
11:     J(B) = max_k(lMass(k) + rMass(k))
12:     leftBudget(B) = argmax_k(lMass(k) + rMass(k)) − 1
13:     rightBudget(B) = B − leftBudget(B) − 1
14: end if
```

With the DP in place, routing decisions can easily be made depending on the δ parameter of the query. Given δ we can compute the desirable mass as δn, where n is the total number of nodes in the network. Using the DP table at the root, we can find the smallest budget that achieves that mass, say b_i, and the values of $leftBudget(b_i)$ and $rightBudget(b_i)$ will give the budget assignments for the left and right child respectively. The traversal descends until the budget is exhausted. The DP vector for each node is equal to the number of nodes in that node's subtree, so if n are all the nodes in the tree, the space needed would be $O(n)$. Constructing the DP vector for each node has complexity $O(n^2)$, so the whole algorithm has complexity $O(n^3)$.

The described DP approach can compute the optimal

traversal solution for a query of window w, on a tree model compressed using the same window. One problem is that the DP tables can become large at nodes high in the hierarchy, which could violate our limited storage principle.

As an alternative, we proceed to propose a simple greedy traversal algorithm that makes decisions locally at every node, without the requirement of keeping extra information, like the DP tables.

4.2 Greedy Algorithm

Our Greedy descent algorithm is quite straightforward. Basically the query is initiated at the root of the tree, and every node makes a decision of whether to descend or not based on the satisfiability of the query by the local model:

$$\int_{\mu-w}^{\mu+w} f(x)dx \geq \delta \tag{3}$$

If the model at the current node satisfies (3), then no descent is necessary. Otherwise the query is forwarded to the node's children. In this simple version of the algorithm, if a decision is made to forward, then all of the children will receive the query. One could extend this to a scheme where children are given different priorities and forwarding can happen selectively. This however would require extra communication, as decisions to forward might depend on traversal results on other subtrees, and cannot be made only locally. Hence we adopt the simple policy of forwarding the query to all children or none.

Algorithm 2 GreedyDescend(node$_i$,w,δ)

```
1: Compute I = ∫_{μ−w}^{μ+w} f(x)dx
2: if I ≥ δ then
3:     return μ_i
4: else
5:     GreedyDescend(children_i,w,δ)
6: end if
```

Note that Algorithm 2 is more conservative than the DP approach, as it applies the satisfiability definition on every subtree and not just the global tree. The algorithm will terminate the recursion at a set of nodes each of which satisfy the query according to Definition 2 in their local subtree. If k_i is the number of nodes represented by every subtree i where the recursion terminated, then in total at least $\delta \sum_i k_i = \delta n$ have accurate within w values, which in turn means that globally the query is always satisfied. So every solution of the greedy algorithm is guaranteed to satisfy query Q, but the satisfiability definition only requires it to hold globally and not for every subtree, and that is why Algorithm 2 is conservative.

We evaluate the performance of the greedy algorithm comparing it with the DP solution for different sizes of the window w, and against different values of the confidence parameter δ depicted on the x-axis. Both algorithms in this case are executed on the same binary tree with data gathered from the Intel Berkeley Lab deployment [1]. To construct the initial leaf distributions, data from one hour is analyzed, and nodes are grouped together in the tree based on spatial proximity. The results are given in Figures 3 and 4. As we pointed out, the greedy algorithm demonstrates a more conservative behavior in terms of cost, which also results in a smaller number of reported errors, but stays a competitive alternative, without requiring maintaining state in the nodes.

5. THE TREE CONSTRUCTION PROBLEM

In the previous sections we discussed methods for performing compression and query traversal. However, the algorithms were applied over a predefined tree, and even though compression is optimally done given a specific workload expectation (i.e. a set window), the way the nodes are grouped has a big effect on the quality of compression. Grouping of dissimilar nodes into the same subtree will inevitably lead to bad compression. Therefore a bad tree topology can lead to bad performance. In this section we will discuss the problem of finding the optimal tree for a specific query workload.

We will solve the problem for the case of graphs with unit edge weights. From a practical perspective, a communication link between two nodes is considered to exist if the nodes have a packet loss rate bounded by a threshold.

The metric we wish to minimize is communication cost during query execution. Assume T_{OPT} is an optimal tree that produces the minimum possible cost when traversed by query $Q(w, \delta)$. A traversal of T_{OPT} using the greedy algorithm (Algorithm 2) will stop at nodes on various levels that all satisfy inequality (3). Define the cut C as the set of nodes at which Alg 2 stops. Note that there is no path from root to leaf in tree T_{OPT} that does not "hit" the cut C. The structure of the tree below the cut, i.e. all nodes that have an ancestor in C, is irrelevant to the cost of query Q, as the query traversal will never descend that far.

A further observation shows that in the case of trees with constant fanout, the structure above the cut is irrelevant as well. This is derived from Theorem 1:

THEOREM 1. *In a tree with fixed fanout F, the cost to traverse from the root to cut, i.e. the cost to reach all nodes in the cut from the root, is $\frac{F}{F-1}(|C| - 1)$, where $|C|$ is the size of the cut.*

PROOF. First we will prove that the size of a cut C_k

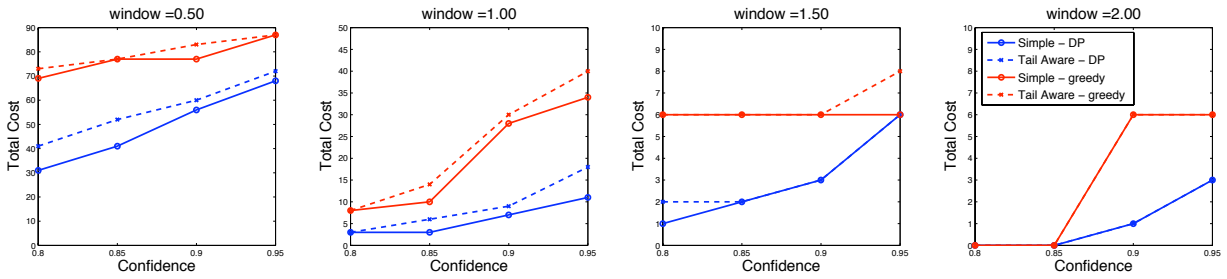

Figure 3: Evaluation of cost of Greedy against optimal cost found by the DP algorithm. The "Simple" and "Tail-aware" schemes refer to the type of compression deployed (Section 3)

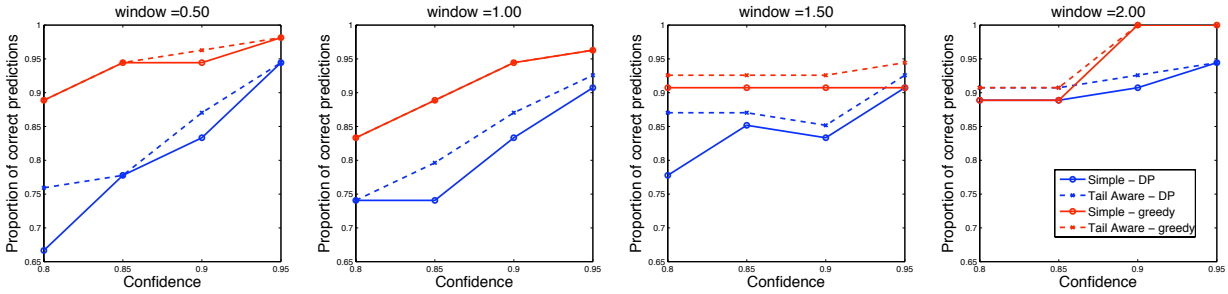

Figure 4: Comparison of the proportion of correct responses for the greedy and the optimal cost traversal chosen by the DP algorithm. The "Simple" and "Tail-aware" schemes refer to the type of compression deployed (Section 3)

is $|C_k| = (k-1)F - (k-2)$, where k integer. What this implies is that depending on the fanout, cuts cannot be of any size. For example a tree of fanout 3 cannot have a cut of size 2.[1] In this discussion C_k will be a cut of size order k. This doesn't mean that C_k is of size k, but rather that it is one order bigger than a cut of order $k-1$. The exact size as we will shortly prove is given by

$$|C_k| = (k-1)F - (k-2) \tag{4}$$

The proof will be done by induction.

- For k=1 the cut only contains the root and it has size $|C_1| = (1-1)F - (1-2) = 1$.
- Assume that for order k a cut C_k has $(k-1)F - (k-2)$ nodes.
- We will now show that a cut C_{k+1} has $kF-(k-1)$ nodes:
  Every cut of order k can be transformed into a cut of order $k+1$ by replacing a node in the cut by its F children. That would increase the cardinality of the cut by F. Since all cuts of order k have the same number of nodes, then all cuts of size $k+1$ that are constructed this way will also have the same number of nodes. The only way that a cut C_{k+1} may not have the same cardinality would be if there is no cut of order k that can produce C_{k+1} with the above method. That means that C_{k+1} does not contain any F nodes that are siblings, otherwise these could be replaced by their parent to form a cut of order k. Now let us see if that is possible. Assume that node $v \in C_{k+1}$ is the deepest node in the cut, i.e. the path from root to v is the longest path from root to the cut. v's siblings is nodes $u_1 \dots u_{F-1}$. If $\exists u_i \notin C_{k+1}$ then this means that there is a node in C_{k+1} that is deeper than node v because there is no path from root to leaf that does not cross C_{k+1}. This is impossible because we assumed that v is the deepest node in the cut. Therefore $u_i \in C_{k+1}$ and thus there is a cut of order k that can produce C_{k+1}. That means that C_{k+1} has the same cost as any other cut of order $k+1$. Now, since $|C_{k+1}| = |C_k| + (F-1)$ we can derive that $|C_{k+1}| = kF - (k-1)$. QED

Using the exact same type of induction we can show that the cost to traverse to a cut of order k is $(k-1)F$. Solving equation (4) in terms of k and replacing to the cost formula, we get that the cost is equal to $F\frac{(|C|-1)}{F-1}$. □

5.1 Optimal Tree Problem

Theorem 1 shows that if C is the cut that the greedy algorithm picks for query $Q(w, \delta)$, then the structure

[1] Remember that with the greedy algorithm, two nodes that are in the same cut cannot be in an ancestor-descendant relationship, as the greedy algorithm forwards the query to either all children or none

of the spanning summary tree above and below the cut are irrelevant to the cost of answering the query. That means that knowing what the cut is would be sufficient to determine query cost. Every node in the cut acts as a representative for a whole subtree, so the cut really is a set of connected components each of which contains a representative node with model M_v that can satisfy query Q.

With that observation, the Optimal Tree Problem with respect to query $Q(w, \delta)$, is the problem of finding a tree that responds to Q using Algorithm 2 with minimum communication cost, and is formally defined as follows:

DEFINITION 3 (OPTIMAL TREE PROBLEM).
Given: *Graph $G(V, E)$ with the cost function $c : E \rightarrow \mathcal{R}_+$ and query $Q(w, \delta)$*
Find: *Set of components $S = \{C_1 \dots C_k\}$ such that*

- *$\forall i$ C_i is connected in G.*
- *$\forall i$ $\frac{1}{|C_i|} \sum_{j \in C_i} Mass_j(C_i) \geq \delta$.*
- *$\cup C_i = V$ and $C_i \cap C_j = \emptyset$ for $i \neq j$*
- *With the objective to minimize the cost of the minimum cost subtree T of G that contains at least one vertex from each component C_i.*

In the above definition $Mass_j(C_i)$ represents the total mass that node j contributes to component C_i.

The objective of the Optimal Tree Problem (4th bullet) is exactly the Group Steiner Tree Problem which is defined as follows: we are given a graph $G(V, E)$, with the cost function $c : E \rightarrow \mathcal{R}_+$ and sets of vertices $g_1, g_2, \dots, g_k \subset V$. The sets $g_1, g_2, \dots, g_k$ are called groups. The objective is to find the minimum cost subtree T of G, that contains at least one vertex from each set g_i. In our case, apart from the GSTP component we also have the problem of finding the clusters that would lead to the tree with the best cost.

Since the Group Steiner Tree Problem is NP-hard, the Optimal Tree Problem is also hard. The GST problem has a polylogarithmic approximation [8], so we will attempt to address the selection of components C_i as a separate problem.

5.2 Optimal Clustering

From Definition 3, the division of nodes into the components C_i can be viewed as a clustering problem. Each cluster will correspond to a subtree rooted at a node chosen by the query cut. According to the definition, each cluster is of limited diameter (the mass inside appropriately centered window w should satisfy confidence δ) and using the intuition provided by Theorem 1, to find the optimal clustering we should try to minimize the number of clusters, or in other words find the minimum size cut for the specific query.

Even though this subproblem is also hard, we will provide a greedy algorithm that approximates the optimal solution by a logarithmic factor. The pseudocode is given in Algorithm 3.

Algorithm 3 GreedyClustering(V, w, δ)

```
1:  Clusters = ∅
2:  Pick discretization D = {z_1, ..., z_k}
3:  repeat
4:    for z ∈ D do
5:      S_z = ∅
6:      for v_i ∈ V do
7:        Mass_z(v_i) = ∫_{z-w}^{z+w} f_i(x)dx //f_i the model of
          node v_i
8:      end for
9:      while Σ_{v_i ∈ S_z} Mass_z(v_i) ≥ δ|S_z| do
10:       v* =argmax_i Mass_z(v_i)
11:       remove v* from Mass_z
12:       if Σ_{v_i ∈ S_z ∪ {v*}} Mass_z(v_i) ≥ δ|S_z| then
13:         S_z = S_z ∪ {v*}
14:       else
15:         break;
16:       end if
17:     end while
18:   end for
19:   S*_z =argmax_i |S_{z_i}|
20:   Clusters = Clusters ∪ S*_z
21:   V = V \ S*_z
22: until V = ∅
```

PROPOSITION 1. *Algorithm 3 provides a factor $\log(n)$ approximation to the optimal clustering, which minimizes the number of clusters.*

Proposition 1 is easily derived when one notices that Algorithm 3 is equivalent to greedy Set Cover. In practice the algorithm behaves a lot better than this guarantee, with results always close to the best solution. The graphs from those experiments were omitted due to space constraints.

We observe that in practice, the greedy clustering performs extremely well. It is important to note however that Algorithm 3 may violate one of the conditions of Definition 3, that each cluster has to be connected. Greedy Clustering greedily adds nodes to each fixed cluster center, which are the intervals given by the discretization. For a fixed center, the weight of every vertex is constant and independent of which other vertices join the same cluster. If we do not require that each cluster is connected, then by picking vertices from the one with the highest weight and decreasing we will get a cluster with the largest cardinality for that interval. However, if we enforce the connectivity requirement, the problem of picking the set of vertices of the highest cardinality that forms a valid cluster, becomes NP-hard as we proceed to show.

Maximum Connected Subgraph of Limited Diameter Problem:

Given: Graph $G(V, E)$, vertex weight W_u for vertex u, maximum diameter D [2]

Find: $G'(V', E')$ s.t:

- G' is connected
- $\frac{1}{|V'|} \sum_{u \in V'} W_u \leq D$
- with the objective to maximize $|V'|$

THEOREM 2. *The Maximum Connected Subgraph of Limited Diameter Problem (MCSLD) is NP-hard*

PROOF. The hardness result is based on a fairly natural reduction from set cover. Assume that there exists a polynomial algorithm that solves MCSLD. We will show that we would then be able to solve any instance of Set Cover in polynomial time.

Assume an instance of Set Cover, with a set of sets $\mathcal{C} = \{C_1, \ldots, C_k\}$. Each C_i is a set containing some elements $\{s_j\}$. We will transform it into an instance of MCSLD as follows: Construct a graph G by creating a node for each C_i, a node for each element s_i and one extra node, let's call it H. Connect H with all the C_is. Connect each C_i with all the elements s_i that are contained in C_i. Give all the s_i and vertex H a vertex weight of $\frac{1}{n+1}$, where n the total number of elements s_i. Assign to each C_i a vertex weight of $A >> 1$.

We will call the MCSLD algorithm on graph G with $D = 1 + A$. If the algorithm returns a solution, then this is equivalent to a set cover of size 1, as the only way to get a connected component with total cost at most $1 + A$ is to include exactly one of the C_i. It is equally easy to see that when MCSLD is called with $D = 1 + rA$, its solution is equivalent to a set cover of size r. Therefore calling MCSLD at most k times, which is polynomial in terms of the input of Set Cover, would give us the optimal solution to Set Cover. But since Set Cover is NP-hard, then MCSLD has to be hard as well. □

The reason why we require each component/cluster to be connected is because the participating nodes should be able to form a tree. To address the connectedness issue, we can follow either of 2 approaches: (a) augment the clusters chosen by the greedy algorithm with extra communication nodes to force connectedness, or (b) adjust the greedy algorithm so it only augments the clusters from accessible nodes. The problem with option (b) is that the algorithm will no longer have the logarithmic guarantee. Option (a) may be more appealing in most cases, as the extra cost for intra-cluster communication would only have to be inflicted once during the cluster formation and determination of the common model, and does not inflict extra cost during query time.

[2]With vertex weight W_u being the mass of u's distribution outside the window interval, then D would be $1 - \delta$

In Figure 5 we see a comparison based on communication cost between the different approaches. The greedy with Intra-Cluster cost, has higher cost than the Connected Greedy for some choices of window size, but as we mentioned the fact that the extra cost cost is only inflicted during cluster formation should be taken into consideration.

5.3 Distributed Clustering

Greedy clustering is a good way to approximate the minimum number of clusters, but Algorithm 3 provides a centralized approach. In this section we will give a distributed clustering algorithm, and we will experimentally compare it with the centralized one.

Algorithm 4 Distributed Expanded Neighborhood

```
1: Basestation broadcasts clustering msg
2: Each node picks a random wait time.
3: if wait time passes without cluster requests then
4:     node initiates clustering
5: end if
6: repeat
7:     node randomly selects v from neighbor table
8:     if d ≤ D_max then
9:         node sends cluster request to v
10:        Augment neighbors table with neighbors of v
11:    else
12:        Remove v from neighbors
13:    end if
14: until no more neighbors
```

The reason for introducing Algorithm 4 is because a centralized approach has the overhead of communicating all the nodes' data to a centralized location to perform the clustering. Also a distributed approach will be more suitable for performing selective reclustering locally if the hierarchical summary needs to be updated. Algorithm 4 basically initiates at a single node level where a cluster of size 1 is created. The node looks up neighbors on its neighborhood table and attempts to augment its cluster size by inviting them to join the cluster. Once a new neighbor joins, the current neighborhood is augmented by the newcomer's neighbors. The cluster stops growing when no more neighbors can be added without exceeding the maximum allowed cluster diameter D_{max}. Note that the algorithm is distributed but requires some bookkeeping and messages to communicate neighborhood tables. Hence we also consider a much simpler version of greedy clustering, using a random walk to pick new nodes, instead of selecting from the total neighborhood. Basically the difference with algorithm 4 is that a new node can only be selected from the neighbors in the network connectivity graph of the last node that was added to the cluster. The advantage of this approach is that it does not require communicating any neighborhood tables.

In Figure 6 we demonstrate comparison experiments

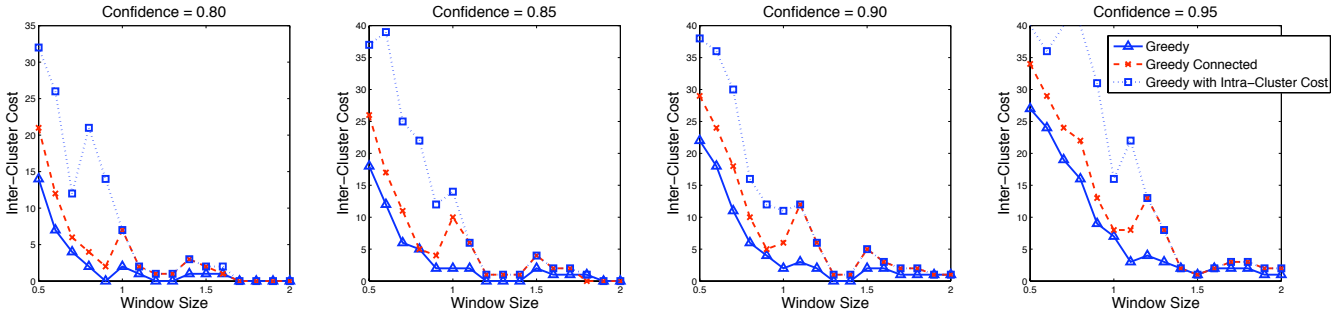

Figure 5: Comparing the different clustering approaches, based on the communication cost for varied parameters of window size and confidence for the query workload.

between the 2 centralized approaches, greedy clustering and centralized greedy, and the 2 distributed approaches, expanded neighborhood and random walk. The simulation experiments were performed using Matlab, and real data from the Intel Berkeley Lab deployment. The objective of all the algorithms is to create a minimum cardinality clustering. An important thing to remember in these graphs is that centralized greedy can usually achieve better results because it doesn't enforce the connectivity requirement, whereas all the other algorithms do. It is possible that centralized greedy does not always give the best result, as it is not optimal, but has a logarithmic approximation guarantee.

From these experiments we observe that the results of distributed clustering are very comparable to the centralized ones, especially to connected greedy. On average the distributed expanded neighborhood and random walk heuristics performed within a factor of 1.4 and 2.8 respectively of the centralized greedy algorithm which has a guaranteed logarithmic factor approximation of the optimal solution.

5.4 Building Trees for Varied Workload

In Section 5 we have discussed the problem of constructing the optimal tree for a specific query. We connected the problem with a clustering problem, we showed hardness results and gave approximation algorithms and heuristics. In this section we will extend our approach to the setting of a more varied query workload. We will assume a baseline confidence δ, and query workload that includes various different windows $\{w_1, \ldots, w_k\}$

A natural but heuristic way to extend our approach to account for the case of different windows would be to recursively cluster in decreasing order of window size. Clustering will start with the largest window size which represents the less strict query in the workload. The clusters produced are further divided into smaller clusters using the next largest window in the set, and the process continues in that fashion until the smallest window size in our set. This algorithm, sketched in Algorithm 5 produces a tree in which every level corresponds to a different window size, from larger in the higher parts of the hierarchy, to smaller in the deeper levels. This also complies with the intuition that models high in the hierarchy present a coarse view of the data, while moving into deeper levels provides more detail.

Algorithm 5 TreeConstruction(G,wRange)

```
1: sort(wRange)
2: G(k+1) = G;
3: for i = k downto 1 do
4:     w = wRange(i)
5:     G(i) = cluster(G(i+1),w)
6:     Connect G(i) with G(i + 1)
7: end for
```

Since the tree construction approach for multiple values of windows that we are proposing is heuristic, we want to experimentally evaluate its performance by comparing its communication cost for queries of the different window sizes, versus a tree that was geared only towards the specific window at hand. We design a single tree for window sizes 0.5, 1, 1.5 and 2, and confidence 0.9, as well as 4 other trees each one geared towards a single one of the previous window values. The communication cost of a query workload of windows [0.5,1,1.5,2] on the first tree is evaluated against the corresponding cost of the tree specifically designed for that window. The results are shown in Figure 7.

We observe that Algorithm 5 approximates well the best single clustering solution.

Notice that the tree construction is not specific to the type of clustering we use, and any of the algorithms that we proposed can be used for the clustering process, centralized or greedy.

Figure 8 presents experiments on a tree designed for a workload of window sizes 0.5, 1, 1.5 and 2, and confidence of 0.9 using Algorithm 3. The hierarchy is evaluated for a query workload of varied confidence and

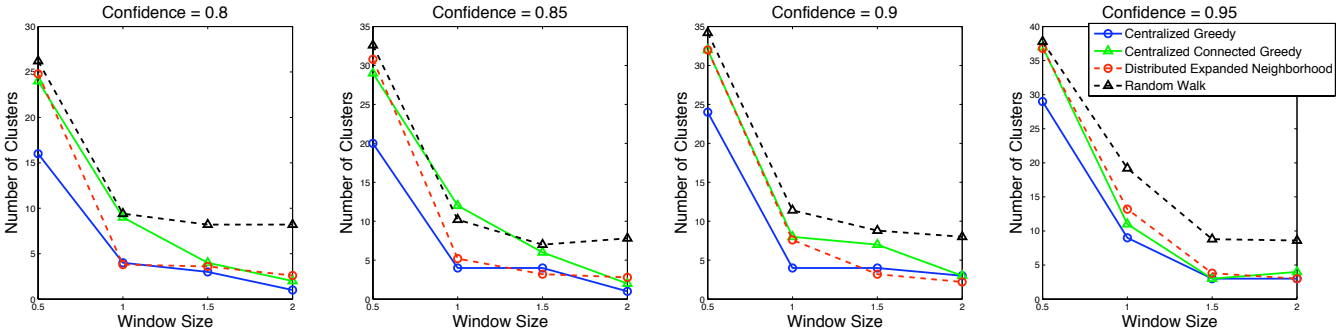

Figure 6: Comparison of the distributed and centralized clustering algorithms.

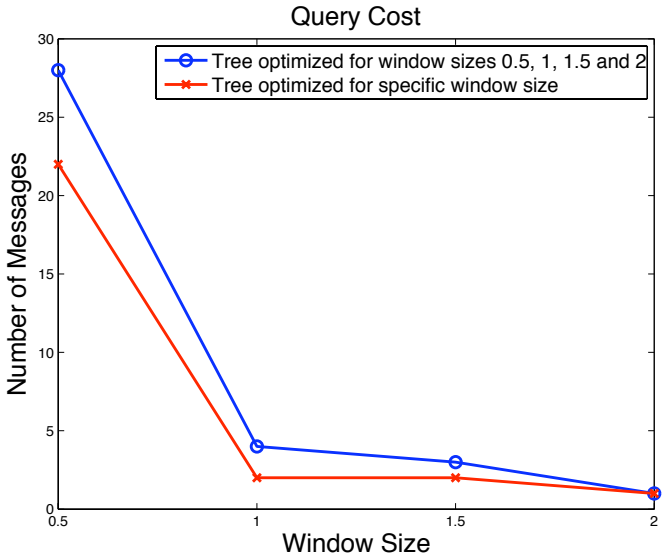

Figure 7: Comparing the performance of a tree produced by Algorithm 5 with the performance that a single window optimal clustering would give.

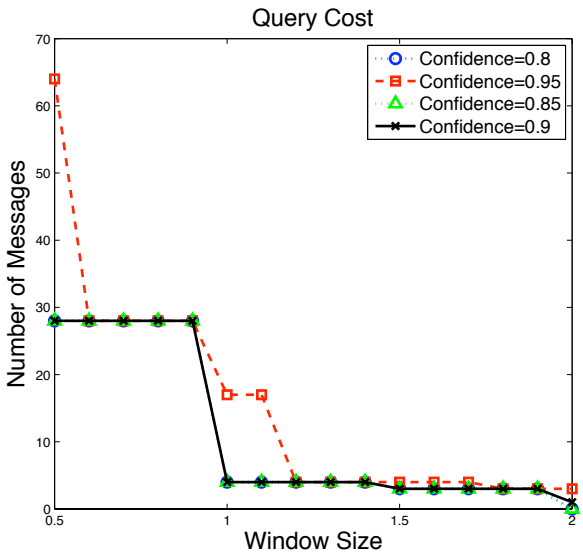

Figure 8: Query experiments on an in-network summary created using the set of window sizes [0.5 1 1.5 2].

window sizes ranging from 0.5 to 2 with a step of 0.1 (windowRange=$[0.5, 0.6, \ldots, 1.9, 2]$). The communication cost behavior resembles a step function, because depending on the strictness of the query (confidence and accuracy window) the traversal usually picks for the most part one level of the hierarchy. Queries with windows that match the ones used in the design space descend to the cluster level corresponding to that window size, or higher, if their confidence in less than what the hierarchy was designed for. In the error graph the horizontal lines represent the respective confidence limits.

In Figure 9 we perform the same experiments on trees constructed using the other 3 clustering approaches, centralized connected greedy (Algorithm 3), distributed expanded neighborhood (Algorithm 4) and random walk. The distributed algorithms are somewhat outperformed by the centralized approach, but they still beat traditional data gathering approaches like TinyDB ([16]) which gathers data along a spanning tree. The TinyDB cost is constant and independent of the query parameters.

5.5 Enriched Models

Our analysis up to this point focused on Single Gaussian Model (SGM) compression schemes, which have minimal requirements of storage space from the nodes. In this section we will evaluate the performance of SGMs against more complex models which preserve more information by utilizing extra space. The size of each model is represented by the parameter k (for SGMs $k = 1$),

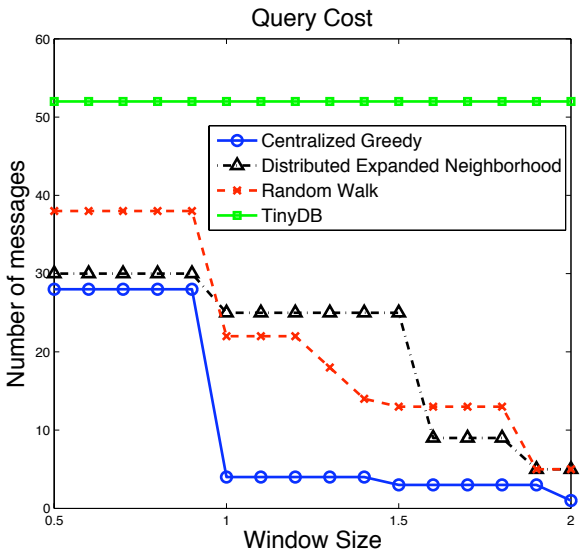

Figure 9: Query experiments on trees constructed by different clustering algorithms

and the amount of space that each model uses is proportional to k.

We will compare 3 different types of models against SGMs:

k-mixture: A k-size mixture is maintained instead of a size 1 mixture. The compression of a mixture of size l to size k is performed by clustering the l distributions of the original mixture into k sets using a modified k-means algorithm. Then each set is compressed to a single gaussian as described in Section 3.1.

Virtual nodes: The l-size mixture is reduced to a set of k mixtures in the same fashion as in the k-mixture model and SGMs are computed for each set. Note the difference with the previous approach: there, a single k-mixture represents all of the nodes in the subtree, whereas here in the virtual nodes approach each separate SGM represents only that portion of the nodes used to construct it. Essentially this is equivalent to splitting each node into k virtual nodes where an SGM is maintained for each one. Note that this approach requires some extra bookkeeping space to record the actual sensor nodes represented by each virtual node.

SGMs on multiple windows: In this case the extra space is used to maintain additional SGMs for different window sizes than the ones that the tree was designed for. Depending on the query window the appropriate model is used at every node.

The generalized models described above are evaluated against simple SGM compression on a tree built for a design workload of window sizes $[0.5, 1, 1.5, 2]$ and confidence of 0.9. The models are then tested on a query workload of a finer range and the results are depicted in Figure 10. The results depicted are for enriched models of size $k = 4$. Experiments with larger k were also performed, with no change in the results.

An initially surprising observation is that enriched models demonstrate minimal to none performance gains. Specifically, k-size mixtures are not any better than a size 1 mixture, and the same goes for virtual nodes. Even though this may initially seem counterintuitive, the reasoning behind this is the criterion used during tree construction. Nodes are divided into clusters ensuring that the total mass contributed by the participating nodes to the interval of the design window size is enough to satisfy the confidence that the tree was designed for. Also, the design of SGMs ensures that this mass is preserved in the resulting compressed model. A more elaborate model is therefore not necessary, as a SGM is perfectly sufficient to represent that information. Thus it is natural that those approaches provide no gain for queries of equal or less confidence than the tree was designed for. We observe some minimal gains on queries of higher confidence – we can now receive some benefit from the additional information – but still the gains are not significant. These results are actually evidence of the quality of our tree construction method. Basically queries have requirements for normal error bounds and thus a normal distribution is the appropriate model when the underlying nodes are clustered based on its properties.

The only actual gains that we observe from an enriched model approach come from our 3rd design: SGMs for multiple windows. This approach keeps several SGMs for multiple windows from the design workload. The gains that we observe from this approach are not due to modeling improvements, i.e. better representation of the underlying data, but due to better tuning to the design workload. Our simple SGM hierarchy attempts to minimize the average communication cost for the given query workload. In the case of multi-window SGMs the benefit arises from keeping models of several window sizes on higher levels, and thus giving the opportunity to queries to terminate higher in the tree than they were initially expected to.

In sight of this observation we also want to evaluate our intuitive decision to assign window sizes to tree levels in decreasing order from root to leaves. We exhaustively enumerate all the possible assignments of window sizes from the design workload to different tree levels, and we compute their average communication cost on queries from that workload. As Figure 11 shows, the intuitive choice of decreasing window sizes along the tree levels is also the right one. The experiment was repeated

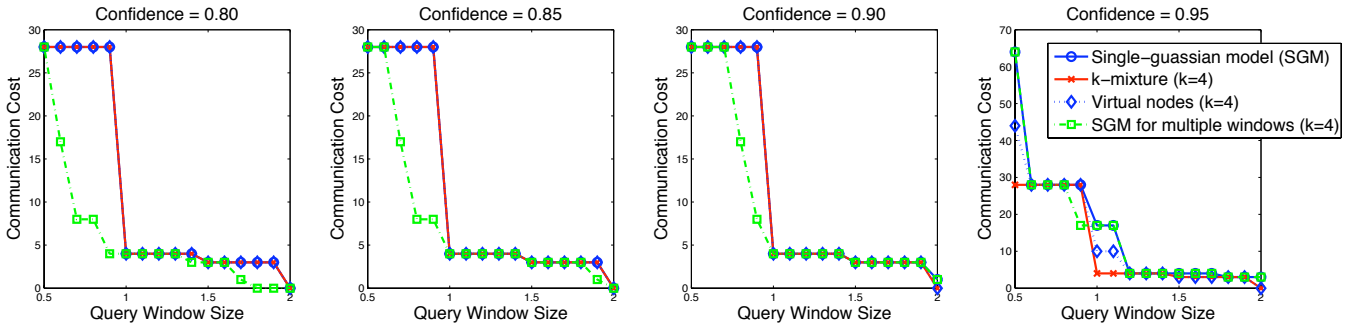

Figure 10: Evaluation of SGMs and enriched models

with several different design workloads, with the same result.

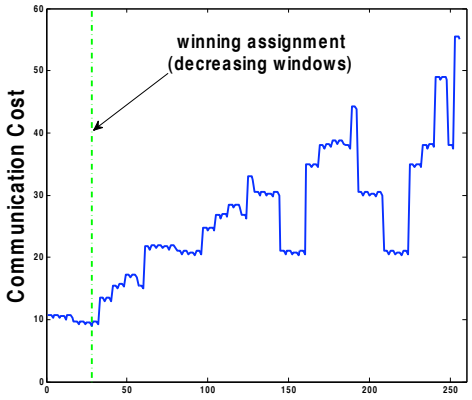

Figure 11: Evaluation of window assignments across tree levels

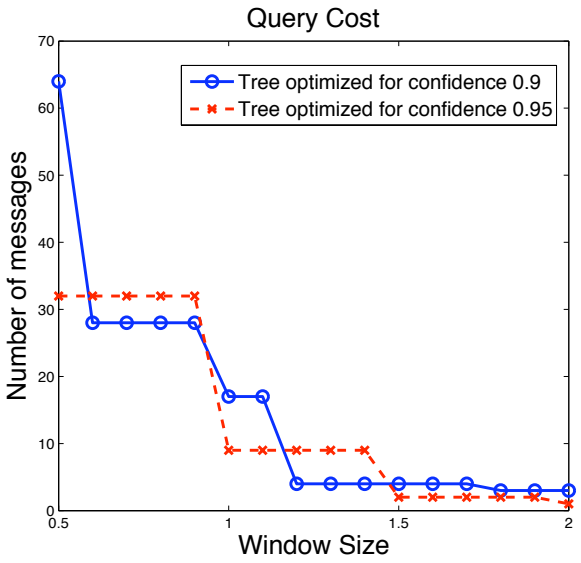

Figure 12: Comparison of hierarchies built on different confidence. The query workload is of confidence 0.95.

6. SENSITIVITY ANALYSIS

In this section we evaluate the sensitivity of our tree construction algorithms to different parameters using simulation experiments. As with all the previous experiments, these are also done using real data from the Intel Berkeley Lab deployment.

Our first set of experiments examine how the value chosen for the confidence parameter during tree construction affects the cost during query execution. We built 2 trees for different confidence limits, 0.9 and 0.95, on the same data, and executed on them the same workload of queries which have a confidence requirement of 0.95. The results of this experiment are in Figure 12. We observe that the tree built on 0.9 confidence has similar performance to the tree that was built for a 0.95 workload, so the baseline parameter δ used in the tree construction does not have a big impact.

Another parameter that could potentially affect performance is the choice of window sizes used for the tree construction. In our second set of experiments, we compare the communication performance during query evaluation on two different trees built using different ranges of window sizes. An interesting observation on Figure 13 is that the tree constructed using the full window range does not always perform better in terms of communication cost. The reason for that is that the hierarchy is forced to have more levels, and to get to a specific one, e.g. the 0.5 level, a query has to be forwarded across more levels on the full range tree. Another reason is that our heuristic of clustering for largest windows first, while natural, can damage the chances of smaller window sizes to form good clusters. That means that the strategy of including all of the possible windows of the expected workload in the tree construction may not always be the best choice. It would be interesting to explore how to optimize the window range for the tree design, and incorporating that range as a parameter in

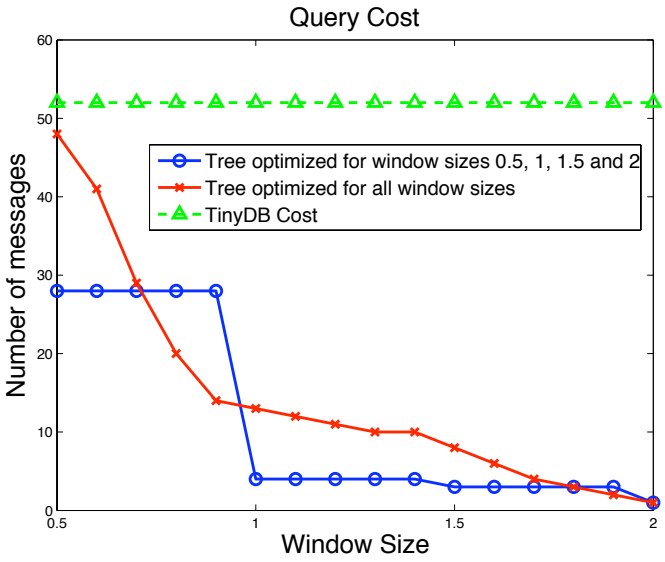

Figure 13: Comparing tree construction with a few vs a broader range of windows

the optimization may require revisiting our tree construction heuristic.

In our final performance experiments, we wish to evaluate the performance of in-network summaries over time. Data is taken for 48 hours, starting around midday of the first day. The in-network summary is built based on data from the first hour, for window sizes 0.5, 1, 1.5 and 2, and confidence 0.9. The evaluation is done using queries with the same parameters.

First we want to see how the hierarchical summary responds to having its models updated every hour, but without changing the structure of the tree. That means that when the distributions of values change at the leaf level, those are propagated to the rest of the levels updating the models, but it is all done on the initial tree structure, without allowing any regrouping of nodes based on the new models. The results are given in Figure 14.

The case of model updates is compared with the case of full tree reconstruction at every time step. We observe that even in the full reconstruction case we get bad performance when variance of the underlying data is high, as happens around timestep 20 and timestep 40. For the case of model updates without restructuring of the tree, we can see that even though for large windows the results are reasonable, for small windows they are very disappointing. An important observation though is that performance seems to deteriorate at different times for the different window sizes. From observation of the experimental results, larger sizes seem to be able to follow the changes until about 40 hours later , clusters with window size 1 seem to become unusable after about 6 hours, and the smallest window clusters seem to not be able to be reused at all.

Our solution is to augment the model updates with *escalated* clustering. What this means is that clusters of different sizes get restructured with different frequencies: clusters corresponding to small window size will need to get reclustered more frequently, whereas those of larger window size much less often. In Figure 15 escalated restructuring is performed every hour, 6 hours and 24 hours, for clusters of size 0.5, 1, and 1.5 respectively. The choice of reclustering frequencies at this point is ad hoc and primarily based on observations of the behavior of Figure 14, but the purpose of this experiment is really to demonstrate that escalated reclustering provides big performance benefits. An interesting part of future work would be to discuss how to automatically derive the appropriate frequencies or determine on the fly when reclustering is necessary.

From Figure 15 we observe that now the in-network summary follows closely the best result. Note that at time 42 and afterwards though, the model deteriorates. This is because it coincidentally happens that the size 1 clusters get updated on a moment with high variance of the underlying data, and thus the resulting clusters are bad and cannot be useful to queries even when the underlying data recovers. To deal with that an easy solution would be to detect data of high variance, and avoid performing restructuring during those times.

7. CONCLUSIONS

In this paper we introduced in-network summaries to improve the efficiency of approximate query processing in sensornets. Summaries can be used to make routing decisions and provide answers to queries without paying the communication cost to access the whole network, and without requiring centralized planning and model maintenance. We formally defined the problem of optimal summary construction, and broke it into several different components that we addressed. We presented efficient compression schemes for the summaries which are optimized based on query workload, and provided traversal algorithms that utilize the summary structures to produce query results. We related the problem of tree optimization to a clustering problem which proved to be NP-hard, and gave a centralized approximation algorithm and distributed heuristics. We experimentally evaluated our algorithms using data from a real world deployment and performed comparisons across multiple parameters. Finally we tested the sensitivity of our scheme to variations of the design decisions and variations of the data, and we showed that it is quite robust to changes.

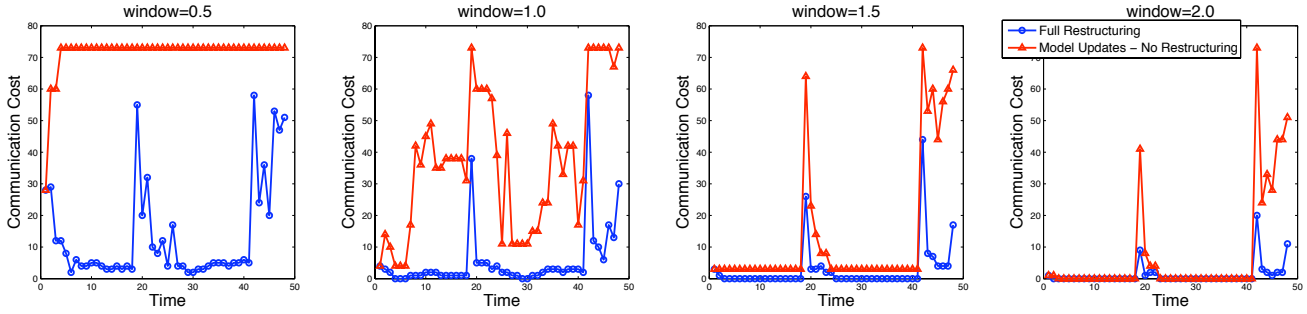

Figure 14: Time progression of in-network summaries with model updates.

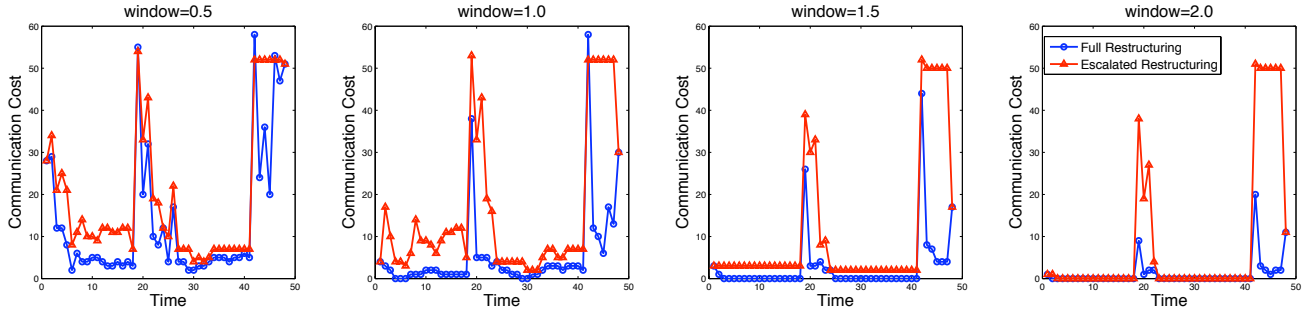

Figure 15: Time progression of in-network summaries with model updates and escalated restructuring.

8. REFERENCES

[1] https://mirage.berkeley.intel-research.net/.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 2002.

[3] Analog Devices. Data sheet: Small, low power 3-axis ±3g imems accelerometer adxl 330.

[4] David Braginsky and Deborah Estrin. Rumor routing algorithm for sensor networks. In *1st ACM international workshop on Wireless sensor networks and applications*. ACM Press, 2002.

[5] M. Demirbas and Xuming Lu. Distributed quad-tree for spatial querying in wireless sensor networks. *Communications, 2007. ICC '07. IEEE International Conference on*, pages 3325–3332, June 2007.

[6] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[7] Alexandros G. Dimakis, Vinod Prabhakaran, and Kannan Ramchandran. Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 15, Piscataway, NJ, USA, 2005. IEEE Press.

[8] Naveen Garg, Goran Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. *J. Algorithms*, 37(1):66–84, 2000.

[9] Minos N. Garofalakis, Joseph M. Hellerstein, and Petros Maniatis. Proof sketches: Verifiable in-network aggregation. In *ICDE*, pages 996–1005. IEEE, 2007.

[10] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks, 1999.

[11] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *6th annual international conference on Mobile computing and networking*, 2000.

[12] Joanna Kulik, Wendi R. Heinzelman, and Hari Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless Networks*, 2002.

[13] S. Madden and J. Gehrke. Query processing in sensor networks. *Pervasive Computing*, 3(1), January-March 2004.

[14] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[15] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2003. ACM.

[16] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[17] Alexandra Meliou, David Chu, Carlos Guestrin, Joseph Hellerstein, and Wei Hong. Data gathering tours in sensor networks. In *IPSN*, 2006.

[18] Alexandra Meliou, Andreas Krause, Carlos Guestrin, and Joseph M. Hellerstein. Nonmyopic informative path planning in spatio-temporal models. In *National Conference on Artificial Intelligence (AAAI)*, July

2007.

[19] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. Ght: a geographic hash table for data-centric storage. In *1st ACM international workshop on Wireless sensor networks and applications*. ACM Press, 2002.