Exploiting Correlated Attributes in Acquisitional Query Processing

Amol Deshpande, Carlos Guestrin, Wei Hong, and Samuel Madden

IRB-TR-04-008

June, 2004

Intel Research Berkeley

Exploiting Correlated Attributes in Acquisitional Query Processing

Amol Deshpande
amol@cs.berkeley.edu
UC Berkeley

Carlos Guestrin Wei Hong
{*guestrin,whong*}*@intel-research.net*
Intel Research, Berkeley

Samuel Madden
madden@csail.mit.edu
MIT

ABSTRACT

Sensor networks and other distributed information systems (such as the Web) must frequently access data that has a high per-attribute *acquisition* cost, in terms of energy, latency, or computational resources. When executing queries that contain several predicates over such expensive attributes, we observe that it can be beneficial to use correlations to automatically introduce low-cost attributes whose observation will allow the query processor to better estimate the selectivity of these expensive predicates. In particular, we show how to build *conditional plans* that branch into one or more sub-plans, each with a different ordering for the expensive query predicates, based on the runtime observation of low-cost attributes. We frame the problem of constructing the optimal conditional plan for a given user query and set of candidate low-cost attributes as an optimization problem. We describe an exponential time algorithm for finding such optimal plans, and describe a polynomial-time heuristic for identifying conditional plans that perform well in practice. We also show how to compactly model conditional probability distributions needed to identify correlations and build these plans. We evaluate our algorithms against several real-world sensor-network data sets, showing several-times performance increases for a variety of queries versus traditional optimization techniques.

1. INTRODUCTION

The past decade has seen the emergence of a number of massive-scale distributed systems, including the Web, the Grid, sensor networks [18], and PlanetLab [1]. Unlike many earlier distributed systems that consisted of a few tens of locally connected nodes, these networks are characterized by their large size, broad geographic distribution, high-latency, and ability to interface users to huge amounts of remote data.

One of the novel challenges associated with query processing in such environments is managing the high cost associated with obtaining the data to be processed. While traditional disk-based DBMSes work very hard to reduce the number of disk I/Os, the high costs of disk access are usually amortized by reading or writing pages that contain many records each. In contrast, in this new class of distributed systems, the cost of fetching a single field of a single tuple is frequently measured in seconds and this effort usually cannot be spread across several fields. For example, in a sensor network, the time to acquire a single reading from a calibrated digital sensor can be long as 1.3 seconds [17]. Following the nomenclature of Madden et al. [16], we refer to such systems as *acquisitional*, to reflect the high costs of accessing data and the fact that it must be actively captured when a query is issued rather than being available on disk at the time of query execution.

In this paper, we explore a class of techniques aimed at mitigating the costs of capturing data in acquisitional systems. These techniques are based on the observation that, in such systems, locally available information with a low cost of acquisition is often highly correlated with much more expensive information. If these correlations are perfect (i.e., there exists a one-to-one mapping from cheap attribute values to expensive attribute values), then a cheap attribute can simply be used in place of an expensive one. However, even in the absence of perfect correlation, a cheap attribute can provide information that can aid in selectivity estimation in query optimization.

For example, Figure 1 shows a scatter-plot of light values[1], versus time of day from a single sensor in sensor network deployed in our lab. In this network, light is an expensive attribute, requiring the processor to be on for almost a second to acquire each sample. However, time of day is readily available in just a few microseconds. Furthermore, looking at this figure, it is clear that light and time of day are correlated: given a time of day, light values can be bound to within a fairly narrow band, especially at night (e.g., during hours 0-5 and 16-24).

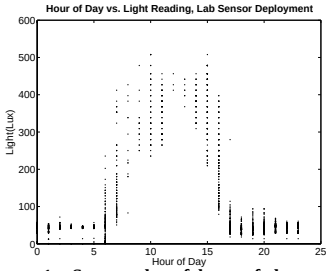


Figure 1: Scatter-plot of hour of day vs. light values at a single sensor within the authors' lab.

We focus on using correlations of this sort to assist in cost-estimation and query processing for multi-predicate range queries, of the form:

```
(1) SELECT a1, a2, ...  , an
 WHERE l1 ≤ a1 ≤ r1  (P1)
 AND ...
 AND lk ≤ ak ≤ rk  (Pk)
```

Where $k \leq n$ and P_n is shorthand for the range predicate in the associated `WHERE` clause. We are particularly interested such queries in acquisitional systems where the cost of acquiring some of the attributes is non-negligible and where correlations exist between one or more attributes. In this paper, we illustrate many examples of such correlations in a variety of real world data sets. A simple way to account for such correlations is to generate a multi-dimensional probability distribution over attribute values and exhaustively search for a single predicate order that will minimize the expected cost. Unfortunately, due to the presence of correlations, expected case performance is far from optimal. For example, given Figure 1, the selectivity of a predicate like `light < 100 Lux` is substantially lower at night than during the day. Thus, if the user issues a multi-predicate query with one such predicate over light, the optimal

[1]In this Figure, light is measured in Lux. Lux is a measure of light received per m^2 of surface area. 100,000 Lux corresponds to full sunlight; 500 - 1000 Lux are typical of a well-lit room; a rural moonlight night is 1-2 Lux.

order of the predicates may vary depending on the time of day. To take advantage of this insight, a query optimizer can *condition* on the time and choose the most optimal plan depending on whether it is night or day.

This observation may seem counter-intuitive: query evaluation can become cheaper by observing additional attributes. If, however, such additional observations are low-cost and allow the query processor to determine with high confidence that a query predicate, P, will reject a particular tuple, this can offer substantial performance gains. The query processor can immediately apply P, avoiding the cost of acquiring unnecessary and expensive attributes. In this paper, we show how to identify such correlations in data, and search for query plans that take advantage of those correlations. We demonstrate that the benefit of this technique can be substantial, offering a several-times performance speedup for a wide range of queries and query workloads, with best-case gains approaching an order of magnitude reduction in running time. Our approach is *adaptive*, allowing a different query plan to be selected on a per-tuple basis, depending on the values of these conditioning attributes. To avoid the overhead of re-running the optimizer on every tuple, we pre-compute these conditional plans based on correlations observed before the query was initiated.

Throughout this paper, we use examples and data sets from sensor networks, though the techniques are general enough to apply to many different acquisitional environments; we return to a discussion of such environments in Section 7.

In summary, the contributions of this paper are:

- We show that correlations are prevalent in sensor network data, and that they can be exploited in multi-predicate range queries by introducing observations over inexpensive attributes correlated with query predicates.
- We introduce the notion of using *conditional plans* for query optimization with correlated attributes.
- We present an optimal, exponential-time algorithm for finding the best possible conditional plan given a multi-dimensional probability distribution over a set of attributes.
- We present a polynomial-time heuristic algorithm for reducing the running time of this algorithm and bounding the size of the generated plans.
- We present efficient ways of computing conditional probabilities over a set of predicates that allow us to avoid the exponential cost of representing an n-dimensional distribution over query variables.
- We evaluate the performance of our algorithms on several real-world data sets, showing substantial benefits over a large class of range queries.

2. CORRELATED ATTRIBUTES AND CONDITIONAL PLANS

In this section, we describe the problem we are seeking to address with our correlation-based techniques and the architecture of our query processing system.

2.1 Problem statement

We are concerned with the problem of generating a conditional plan that will minimize the expected execution cost of a user-supplied range query of the form shown in query (1) above. Here, we assume that there are n attributes in the query table, $X_1, \ldots, X_n$, and that the first m of them, $X_1, \ldots, X_m$ are referenced in the query (so $m \leq n$). We assume that each attribute X_i takes on a value $x_i \in \{1, \ldots, K_i\}$. Note that this definition requires real-valued attributes to be discretized appropriately. In sensor networks, for example, this discretization is natural, as the resolution of the sensors is limited (e.g., to 10 bits when using the internal analog-to-digital (ADC) on the Berkeley Motes [6, 12].) We use the tuple $\mathbf{x}$ to denote a particular assignment to the all attributes $X_1 \ldots X_n$. Each attribute is also associated with an *acquisition cost*, where C_i denotes the cost of acquiring the value of X_i. In a sensor network, this cost ranges from a few microjoules to tens of millijoules [17]; according to Madden et al. [16] the cost of acquiring a sensor reading once per second on a mote can be comparable to the cost of running the processor.

We denote the predicates in the query by $\{\phi_1, \ldots, \phi_p\}$, and the logical `where` clause by φ. For simplicity, we use $\varphi(\mathbf{x})$ to denote the truth value of φ for the tuple $\mathbf{x}$. Our goal is, thus, to find a minimum expected cost plan for determining whether $\varphi(\mathbf{x})$ is true or false, where the expectation is taken over the possible assignments to the tuple $\mathbf{x}$.

We focus our conditional plans Γ on simple binary decision trees[2], where each interior node n_j specifies a binary *conditioning predicate*, $T_j(X_i \geq x_i)$, that splits the plan into two alternate conditional plans, $\Gamma_{T_j(\mathbf{x})=\mathtt{T}}$ and $\Gamma_{T_j(\mathbf{x})=\mathtt{F}}$, where $T_j(\mathbf{x})$ is the truth value of T_j on the tuple $\mathbf{x}$. At node n_j, the query processor will evaluate the predicate $T_j(\mathbf{x})$ and choose to execute one of these two subplans depending on the value of the predicate. Each conditioning predicate depends only on the value of a single attribute. It is important to note that, in addition to our query predicates $\{\phi_1, \ldots, \phi_p\}$, our plan can potentially include conditioning predicates T_j over attributes that do not appear in the query. For example, due to correlations, we may condition our plan on the value of an attribute X_j with low acquisition cost, if X_j gives us information about one or more query attributes of high acquisition cost.

Each leaf node of the binary tree Γ is annotated with `T` or `F` to indicate whether φ has been determined to be true or false.

Figure 2 shows a simple illustrative example of a conditional query plan, where two introduced attributes (voltage and time of day) are used to condition the plan. Edges of the conditional plan are labeled with the value of their conditioning predicates and the conditional probabilities of those predicates being satisfied. In this simple example, rather than annotating each leaf of the plan with `T` or `F`, we annotate them with simple sequential plans over the query predicates, *e.g.*, if the hour of the day is between 19 and 24, and the voltage is less than $2.5V$, then we execute Plan4, which first measures the GPS position, then the light value, and finally the temperature. This conditional plan can be cheaper than any single sequential ordering of the predicates, because predicate selectivities vary depending on the value of the conditioning predicates, as shown in the table in the lower right corner of the figure.

2.2 Evaluating plan cost

During execution of a plan Γ, we simply traverse the binary tree defined by Γ, acquiring any attributes we need to evaluate the conditioning predicates. For a particular tuple $\mathbf{x}$, *i.e.*, an assignment to all attributes, our plan will traverse a single path to a leaf of the binary tree Γ, which is labeled with `T` or `F` indicating the truth value of $\varphi(\mathbf{x})$. The cost of this traversal is the sum of the cost of the attributes that are acquired by plan Γ

[2]Our approach can, of course, be extended to more general decision trees, as discussed in Section 7.

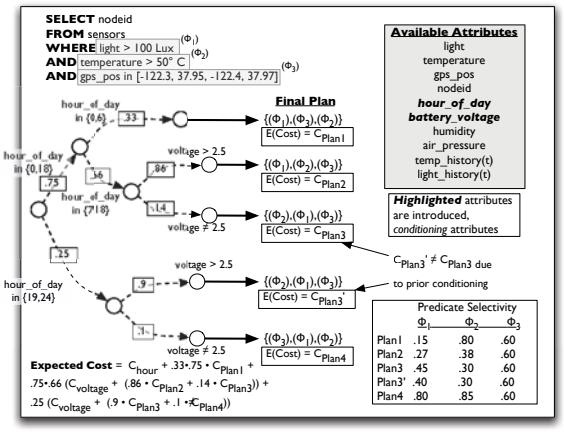


Figure 2: A conditional query plan. Notice that hour_of_day and voltage are used as conditioning attributes. Different orderings of the query predicates are used in the final evaluation of the query plan depending on the value of these attributes. Edges of the conditional plan are labeled with their conditional predicates as well as their probabilities.

in this traversal. Specifically, at each node n_j in this traversal, if the attribute in the predicate T_j has already been acquired, then this node has zero *atomic cost.* However, if the attribute X_i in T_j has not yet been acquired, then the atomic cost of this node is C_i. For simplicity, we annotate each node n_j of our plan with this atomic cost $C(n_j)$. Note that the atomic cost of a leaf is 0, as no attributes are acquired at this point. We can now formally define the *traversal cost* $C(\Gamma, \mathbf{x})$ of applying plan Γ to the tuple $\mathbf{x}$ of the plan recursively by:

$$C(\Gamma, \mathbf{x}) = \begin{cases} 0 & \text{if } |\Gamma| = 1, \\ C(\texttt{Root}(\Gamma)) + C(\Gamma_{T_j(\mathbf{x})}, \mathbf{x}), & \text{otherwise,} \end{cases} \quad (1)$$

where $\texttt{Root}(\Gamma)$ is the root node of the tree for plan Γ, $C(\texttt{Root}(\Gamma))$ is the atomic cost of this root node as defined above, and the cost is 0 when we have reached a leaf, *i.e.*, when $|\Gamma| = 1$.

Optimal planning is an optimization problem that involves searching the space of available conditional plans that satisfy the user's query for the plan Γ^* with minimal expected cost:

$$\begin{aligned} \Gamma^* &= \arg\min_{\Gamma} C(\Gamma), \\ &= \arg\min_{\Gamma} E_{\mathbf{x}}\left[C(\Gamma, \mathbf{x})\right], \\ &= \arg\min_{\Gamma} \sum_{\mathbf{x}} P(\mathbf{x}) C(\Gamma, \mathbf{x}). \end{aligned} \quad (2)$$

Using the recursive definition of the cost $C(\Gamma, \mathbf{x})$ of evaluating a tuple $\mathbf{x}$ in Equation (1), we can similarly specify a recursive definition of the expected cost $C(\Gamma)$ of a plan. For this recursion, we must specify, at each node n_j of the plan, the conditional probability that the associated predicate T_j will be true or false, given the predicates evaluated thus far in the plan. We use $\mathbf{t}$ to denote an assignment to this set of predicates. Using this notation, the expected plan cost is given by:

$$C(\Gamma, \mathbf{t}) = \begin{cases} 0 & \text{if } |\Gamma| = 1, \\ C(\texttt{Root}(\Gamma)) + \\ \quad P(T_j \mid \mathbf{t}) C(\Gamma_{T_j}, \mathbf{t} \wedge T_j) + & \text{otherwise,} \\ \quad P(\neg T_j \mid \mathbf{t}) C(\Gamma_{\neg T_j}, \mathbf{t} \wedge \neg T_j), \end{cases} \quad (3)$$

where $C(\Gamma, \mathbf{t})$ is the expected cost of the (sub)plan Γ starting from its root node $\texttt{Root}(\Gamma)$, given that the predicates $\mathbf{t}$ have

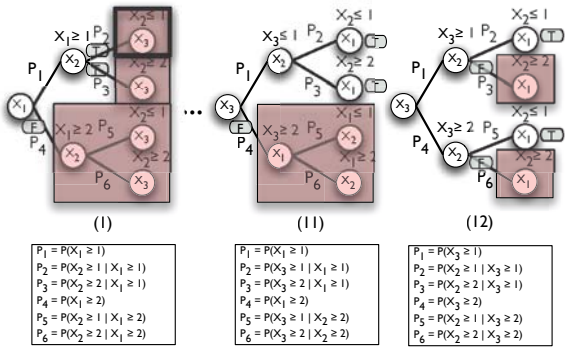


Figure 3: Set of possible plans for the two predicate query $X_1 = 1 \wedge X_2 = 1$, with three attributes, X_1, X_2, and X_3 available for use in the query. The labels in the nodes indicate the attribute acquired at that point, and the labels on the edges denote the probability of the outcome along that edge. Each P_i expands into the conditional probability expansion given at the bottom of the figure. Terminal points in the tree are labeled with their outcome: `T` is the tuple passes the query or `F` if it fails. Grayed out regions do not need to be explored because a predicate fails or all predicates are satisfied before are reached.

been observed. At this point, the expected cost depends on the value of the new predicate T_j. With probability $P(T_j \mid \mathbf{t})$, T_j will be true and we must solve the subproblem $C(\Gamma_{T_j}, \mathbf{t} \wedge T_j)$, *i.e.*, the subplan Γ_{T_j} after observing the original predicate values $\mathbf{t}$ and the new predicate value $T_j = \texttt{T}$. Similarly, with probability $P(\neg T_j \mid \mathbf{t}) = 1 - P(T_j \mid \mathbf{t})$, T_j will be false and we must solve $C(\Gamma_{\neg T_j}, \mathbf{t} \wedge \neg T_j)$, *i.e.*, the subplan $\Gamma_{\neg T_j}$ after observing $\mathbf{t}$ and the $T_j = \texttt{F}$. As before, when we reach a leaf ($|\Gamma| = 1$), the cost is 0. Now the expected cost of a plan Γ is defined using Equation (3) by $C(\Gamma, \emptyset)$.

We present several search algorithms for finding minimal cost plans in Section 3. In the remainder of this section, we illustrate the concepts presented thus far by considering the naive generate-and-test algorithm that enumerates all possible conditional plans over a set of attributes.

Consider the simple example of exhaustively enumerating the plans for three attributes, $\{X_1, X_2, X_3\}$, each with binary domain $\{1, 2\}$. Our query in this example is simply $\varphi = (X_1 = 1 \wedge X_2 = 1)$. Figure 3 shows three possible plans in this enumeration; there are 12 total possible plans in this case. Each node in this figure is labeled with the attribute acquired at that node, and the P_i values denote the conditional probability of the outcome along each edge occurring, given the outcomes already specified in the plan branch. Terminal outcomes are denoted as `T` or `F`, indicating that a tuple that reaches this node is output or rejected.

Note that, if at some branch of the plan we can prove the truth value of φ we do not need to further expand the tree. For example, in Figure 3, the grayed-out regions represent branches of the plan that do not need to be evaluated since a predicate has failed. The outlined box at the top of Plan (1) indicates that all query predicates have been evaluated on this branch, so X_3 does not need to be acquired.

Given the plans in Figure 3, it is straightforward to read off the expected cost as defined in Equation (3). For example, for Plan (11) the cost is:

$$\begin{aligned} C(\text{Plan (11)}) = C_3 + \\ P(X_3 = 1)(C_2 + P(X_2 = 1 \mid X_3 = 1)C_1) + \\ P(X_3 = 2)(C_1 + P(X_1 = 1 \mid X_3 = 2)C_2), \end{aligned}$$

where, for example, when branching on X_2, we do not need to consider the branch for the assignment $X_2 = 2$ as this assign-

ment makes φ false and we replace the grayed-out box by a leaf with value F.

At this point, the observation from the introduction bears repeating: the cheapest possible plan is not always the one that immediately acquire the query predicates. In our example, plan (12) could be cheaper than plan (1), if observing X_3 has low cost and dramatically skews the probabilities of the attributes X_1 and X_2. In particular, if $X_3 = 1$ increases the probability of $X_2 = 2$, then observing X_3 may allow us to select the particular attribute that is more likely to determine if $\varphi =$ F. Thus, if $X_3 = 1$, the query processor may avoid acquiring X_1, which it does first in plan (1).

2.3 Estimating event probabilities

We now look at the question of determining the values of the conditional probabilities on the edges of a plan Γ. The problem is to determine the probability that a predicate T_j of the form $T_j(X_i \geq x_i)$ will be satisfied, given a set of conditioning predicates, or *a priori* conditions, $\mathbf{t}$. We can write this probability as $P(T_j \mid \mathbf{t}) = P(T_j|t_0, \ldots, t_{|\mathbf{t}|})$, where t_i is the truth assignment to a predicate T_i that appears in $\mathbf{t}$. Bayes rule tell us that this quantity is equal to:

$$\frac{P(T_j, t_0, \ldots, t_{|\mathbf{t}|})}{P(t_0, \ldots, t_{|\mathbf{t}|})}.$$

Given this definition, a naive method for computing such probabilities is to scan the historical data, computing the rows r that satisfy the predicates in $\mathbf{t}$, and the subset of r, r_{sat}, that satisfies $T_j =$ T. The probability $P(T_j =$ T $\mid \mathbf{t})$ is simply $|r_{sat}|/|r|$.

The disadvantage of this approach is that it requires a linear scan of the entire data set for each probability computation (although the space requirements are small, as it is trivial to compute the sizes of r and r_{sat} without actually materializing the rows.) As we will see, our algorithms can require tens of thousands of such computations, so linear scans can be costly, especially for large data sets. Furthermore, if the number of predicates in $\mathbf{t}$ is large, then the number of records that satisfy these predicates may be very small. In such cases, our estimate of $P(T_j =$ T $\mid \mathbf{t})$ will have very high variance. We describe more efficient (and potentially more accurate) techniques for estimating conditional probabilities in Section 3 below.

2.4 Communications model and plan size

Once a plan is generated, it is sent into the network and executed locally at each of the sensor nodes. In a sensor network, in addition to the expected acquisition cost, we are also concerned with plan size, as larger conditional plans with more branches require more communication, and the overhead of transmitting these larger plans could possibly outweigh the benefit gained by executing them.

Plan size is also important due to the limited storage available in sensor nodes. A complete conditional binary tree of depth d, contains $O(2^d)$ nodes. Each node in this tree requires at least two bytes. Thus, the 4K of RAM available on motes can only store complete trees of depth 11. Of course, our plans are not complete binary trees, as illustrated by Figure 3. Nonetheless, this issue still needs to be addressed.

A simple approach is to bound the plan size to be under some fixed size, where that size can be selected to easily fit into device RAM. Alternatively, we can modify our optimization problem to include both the cost of acquiring predicates and the cost of communicating the plan. In this case, the optimization becomes:

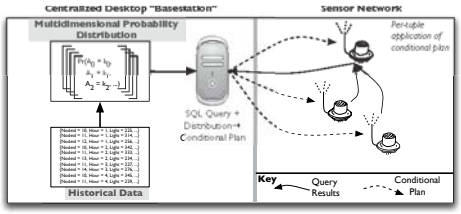


Figure 4: Query processing architecture. Conditional plans are generated from multi-dimensional probability distributions on the basestation and sent into the network. Sensors execute the plans locally, producing results which are routed back to the basestation for display.

$$\arg\min_{\Gamma} \left(C(\Gamma) + \alpha\zeta(\Gamma)\right),$$

where $\zeta(\Gamma)$ denotes size in bytes of Γ, and α is a scaling factor equal to $\frac{\text{cost to transmit a byte}}{\text{\# tuples processed in query lifetime}}$ that allows us to capture the intuition that, as the running time of a continuous query gets large, the time spent in query execution will dominate the cost of sending the plan. We focus our presentation on limiting plan sizes, though this joint optimization problem could be addressed with an extension of our approach.

2.5 Architecture

Before detailing our algorithms for plan generation, we briefly discuss the architecture in which queries are executed. Current sensor network hardware [6], with 4K of RAM and a 4Mhz processor does not have sufficient power to run our optimization algorithms. However, once the plan is generate, the online plan execution step (*i.e.*, a simple traversal of a binary tree) requires minimal computational power. As a result, we build the conditional plans offline, on a well-provisioned *basestation*, using historical readings collected over time. Once generated, these plans are distributed to the sensors (or other query processing nodes) for execution. Just as optimizer statistics are periodically collected and re-analyzed in a traditional DBMS, we envision that conditional plans may be re-generated at the user's request, or when the query processor detects substantial changes in the correlations.

Figure 4 illustrates the major components of our architecture. The basestation collects historical data, computes conditional probabilities over that data, using either a multidimensional probability distribution or by repeatedly scanning the data (as described above), and uses those probabilities plus the user's query to generate a conditional plan. This plan is sent into the network, executed by the sensors, and the results are transmitted back to the basestation.

3. OPTIMAL SOLUTION

This section focuses on obtaining the minimal cost plan for determining the truth value of a logical clause φ as defined in Equation (2). We first determine the complexity class of this problem, and then present an exhaustive algorithm that can compute such optimal plan.

3.1 Problem complexity

The optimization problem outlined in the previous section is an instance of the general *minimum cost resolution strategy problem (MRSP)* [9], which seeks to find a minimum cost strategy for evaluating a boolean formula φ. The complexity of MRSP has been analyzed for various special cases. In this section, we analyze a case that is particularly relevant for our query

optimization task. We focus on simple boolean formulas φ that are formed by the conjunction of unary predicates over a subset $X_1, \ldots, X_m$ of our n correlated attributes. This simple case includes our example Query (1), and, of course, if we were to include disjunctions the complexity will usually not decrease. For this problem class, we prove that:

THEOREM 3.1. *Let $X_1, \ldots, X_n$ be a set of attributes, and let φ be a conjunction of unary predicates on $X_1, \ldots, X_m$, where $m \leq n$, then:*

1. *even if we have an oracle that instantaneously computes the conditional probabilities $P(T_j \mid \mathbf{t})$, for any predicate T_j and any assignment to any set of predicates $\mathbf{t}$, then, for some cost K, the problem of deciding whether a plan for determining the truth value of φ with expected cost lower than K exists is #P-hard;*
2. *if $\mathcal{D}$ is a set of d tuples, and the expected cost of a plan Γ is approximated by:*

$$C(\Gamma) = E_{\mathbf{x}}\left[C(\Gamma, \mathbf{x})\right] \approx \frac{1}{d} \sum_{\mathbf{x} \in \mathcal{D}} C(\Gamma, \mathbf{x}), \qquad (4)$$

then, for some cost K, the problem of deciding whether a plan for determining the truth value of φ with approximate expected cost (as defined in Equation (4)) lower than K exists is NP-complete. □

Item 1 addresses a very general exact optimization case. Here, we decouple the problem of optimizing the plan from that of computing the conditional probabilities required to evaluate a plan, by adding an instantaneous oracle that can provide these probabilities. Even with this oracle, the complexity of the problem we would like to solve is still #P-hard (reduction from #3-SAT), indicating that efficient solution algorithms are unlikely to exist. Alternatively, as addressed in Item 2, we may wish to optimize our plan with respect to a particular dataset $\mathcal{D}$. This is a simpler case, as we only need to consider the tuples in $\mathcal{D}$, allowing us to ignore exponentially-many possible tuples in the optimization that are not in $\mathcal{D}$. Unfortunately, this simpler problem is still NP-complete (reduction from the complexity of binary decision trees [13]).

3.2 Exhaustive algorithm

The hardness results in the previous section indicate that a polynomial time algorithm for obtaining an optimal plan Γ^* is unlikely to exist. In this section, we describe an optimal depth-first search algorithm that includes caching and pruning techniques to attempt to find Γ^*. Even with caching and pruning, the worst case complexity of this algorithm is still exponential in the number of attributes. However, as we are computing our plans off-line, and then downloading these plans onto the sensor nodes, we expect that, for query tables over a small number of attributes, this exhaustive algorithm should be sufficient. problems. These methods are then compared in Section 6.

Our dynamic programming algorithm (Figure 5) is based on the observation that once a predicate is used to split the conditional plan, it sub-divides the original problem into two independent subproblems. Each subproblem covers a disjoint subspace of the attribute-domain space covered by the original problem, and as such, can be solved independently of the other subproblem. A subproblem is thus defined by the attribute-domain space covered by it, or, specifically, by the ranges of the values that the attributes can take given the conditioning predicates used before this subproblem was generated.

We will denote a subproblem where each attribute X_i can take values in the range $R_i = [a_i, b_i]$ by Subproblem(φ,$R_1 = [a_1, b_1], \ldots, R_n = [a_n, b_n]$). Our initial problem is denoted by Subproblem(φ,$R_1 = [1, K_1], \ldots, R_n = [1, K_n]$). We will specify the dynamic programming algorithm by defining the expected completion cost of a problem in terms of the costs of its subproblems. Here, we denote the cost of the cost of the optimal plan for Subproblem(φ,$R_1, \ldots, R_n$) by J(φ,$R_1, \ldots, R_n$).

The subproblems that are formed from a particular problem are defined by the predicate used to split this problem. The potential predicates that can be used to further divide Subproblem(φ,$R_1, \ldots, R_n$) are of the form $T(a_i \leq X_i < x_i)$ and $T(x_i \leq X_i \leq b_i)$. That is, the range $R_i = [a_i, b_i]$ of X_i is divided into $[a_i, x_i - 1]$ and $[x_i, b_i]$. Our goal is to choose the optimal splitting attribute X_i and the optimal assignment from the range $[a_i, b_i]$.

The expected cost of a problem can now be defined recursively by the sum of three terms: the cost of acquiring the splitting attribute, plus the cost of each subproblem weighted by the probability that the observed attribute value will lead to this particular subproblem. Specifically, if we choose to split on attribute X_i, we must first pay the cost of acquiring X_i, which we now denote by C_i'. If the original problem has not yet acquired this attribute, *i.e.*, if the range R_i of X_i still spans all possible values ($R_i = [1, K_i]$), then we must pay the acquisition cost $C_i' = C_i$. However, if X_i has already been acquired (in which case, $[a_i, b_i]$ will be a strict subset of $[1, K_i]$), then $C_i' = 0$.

Now, we must consider the recursive cost of the subproblems. The particular choice of subproblem depends on the actual observed value of X_i. Thus, the probability we will need to solve Subproblem(φ,$R_1, \ldots, [a_i, x_i - 1], \ldots, R_n$) depends on the probability that X_i will take on a value in $[a_i, x_i - 1]$ given the ranges $R_1, \ldots, R_n$ observed thus far, *i.e.*, $P(X_i \in [a_i, x_i - 1] \mid R_1, \ldots, R_n)$. The optimal choice of splitting attribute and value is now the one that minimizes the expected cost of the resulting subproblems.

EXHAUSTIVEPLAN(φ,$R_1, \ldots, R_n$,$\overline{C}$)
- IF THE RANGES $R_1, \ldots, R_n$ ARE SUFFICIENT TO DETERMINE TRUTH OF φ, OR ALL QUERY ATTRIBUTES HAVE BEEN OBSERVED:
  - RETURN $[0, \emptyset]$.
- IF SUBPROBLEM $R_1, \ldots, R_n$ HAS BEEN CACHED:
  - RETURN CACHED RESULT.
- LET $C_{min} \leftarrow \overline{C}$, AND $\Gamma \leftarrow \emptyset$.
- FOR $i \leftarrow 1$ TO n:
  - IF R_i IS $[1, K_i]$, THEN:
    - LET $C' \leftarrow C_i$.
  - ELSE:
    - LET $C' \leftarrow 0$.
  - IF $C' < C_{min}$, THEN:
    - FOR $x_i \leftarrow a + 1$ TO b, WHERE $R_i = [a, b]$:
      - LET $[C_{<x_i}, \Gamma_{<x_i}] \leftarrow$ EXHAUSTIVEPLAN(φ, $R_1, \ldots, [a, x_i - 1], \ldots, R_n$,$C_{min} - C'$).
      - LET $P_{<x_i} \leftarrow P(X_i \in [a, x_i - 1] \mid R_1, \ldots, R_n)$.
      - LET $C' \leftarrow C' + P_{<x_i} C_{<x_i}$.
      - IF $C' < C_{min}$, THEN:
        - LET $[C_{\geq x_i}, \Gamma_{\geq x_i}] \leftarrow$ EXHAUSTIVEPLAN(φ, $R_1, \ldots, [x_i, b], \ldots, R_n$,$C_{min} - C'$).
        - LET $P_{\geq x_i} \leftarrow (1 - P_{<x_i})$.
        - LET $C' \leftarrow C' + P_{\geq x_i} C_{\geq x_i}$.
        - IF $C' < C_{min}$, THEN:
          - LET $C_{min} \leftarrow C'$.
          - LET $\Gamma \leftarrow \left\{ \begin{array}{l} T(X_i < x_i) \rightarrow \Gamma_{<x_i}, \\ T(X_i \geq x_i) \rightarrow \Gamma_{\geq x_i} \end{array} \right\}$.
- IF $C_{min} < \overline{C}$, THEN:
  - CACHE $[C_{min}, \Gamma]$ AS THE OPTIMAL PLAN FOR $R_1, \ldots, R_n$.
- RETURN $[C_{min}, \Gamma]$.

Figure 5: Exhaustive planning algorithm.

$$J(\varphi, R_1, \ldots, R_n) = \min_i \min_{x_i \in [a_i+1, b_i]} C'_i + P(X_i \in [a_i, x_i - 1] \mid R_1, \ldots, R_n) J(\varphi, R_1, \ldots, [a_i, x_i - 1], \ldots, R_n) + P(X_i \in [x_i, b_i] \mid R_1, \ldots, R_n) J(\varphi, R_1, \ldots, [x_i, b_i], \ldots, R_n). \quad (5)$$

To complete this recursion, we must define the base cases. First, if all of the attributes in φ have already been observed, then the cost is 0. Specifically, $J(\varphi, R_1, \ldots, R_n) = 0$, if each $R_i = [a_i, b_i]$ is a strict subset of $[1, K_i]$, for each query attribute $i = 1, \ldots, m$. Similarly, we may reach a point where the ranges $R_1, \ldots, R_n$ are sufficient to evaluate the truth value of φ. For example, if φ is a conjunctive query and the jth predicate in the query is $\phi_j(X_i \geq 10)$, and we reach a subproblem $J(\varphi, R_1, \ldots, R_n)$ where $R_i = [1, 9]$, then the cost of this subproblem is 0. These base cases now complete the recursion. The optimal plan is obtained by caching, for each subproblem $J(\varphi, R_1, \ldots, R_n)$, the attribute X_i and the assignment x_i that minimizes Equation (5).

Figure 5 shows the pseudo-code of our dynamic programming algorithm based on this recursion, that searches the plan space in a depth-first manner. Other than the caching optimization inherent in dynamic programming, we also use pruning to cut down on the search space explored. Our pruning strategy is simple: when exploring possible splitting attribute values, we store the cost of the best plan explored thus far ($\overline{C}$), if the current branch exceeds this cost, we no longer need to explore. More elaborate pruning techniques, such as branch-and-bound,could potentially be effective in this problem.

The complexity of EXHAUSTIVEPLAN depends on the total number of subproblems that may be generated during the execution. This number is bounded by $\prod_{j=1}^{n} \frac{K_i \times (K_i - 1)}{2}$, *i.e.*, the number of possible ranges $R_1, \ldots, R_n$. For each subproblem, we have to consider (1) each possible splitting attribute and assignment, in the worst case, $\sum_{j=1}^{n}(K_i - 1)$, and (2) compute the conditional probabilities required by the algorithm. Hence the running complexity of this algorithm in the worst case (when pruning does not help in reducing the search space) is $O(nKK^{2n} + K_p K^{2n})$, where $K = \max_i K_i$ is the maximum number of possible assignments for an attribute, and K_p denotes the complexity of computing the conditional probabilities — we will discuss this in detail in Section 5. Thus, this optimal algorithm is only feasible for a small number of attributes that take on a relatively small number of assignments.

4. HEURISTIC SOLUTIONS

The complexity of the exhaustive algorithm described in the last section is prohibitive except for the simplest of problems. In this section, we present heuristic algorithms for finding good conditional plans.

4.1 Optimal Sequential Plan

An *optimal sequential plan* is defined to be the optimal execution plan that does not use *any* conditioning predicates to *split* the plan, choosing instead a sequential order on predicates that is followed regardless of the observed values, until a stopping condition is reached. Sequential plans are the building block used in our heuristic algorithm. We will use $\widehat{\Gamma} = \text{OPTSEQUENTIAL}(\varphi, R_1, \ldots, R_n)$ to define the optimal sequential plan $\widehat{\Gamma}$ given that we have already observed the ranges $R_1, \ldots, R_n$ to the attributes $X_1, \ldots, X_n$.

4.1.1 Optimal Sequential Plan for Conjunctive Queries

Interestingly, the exhaustive algorithm described in Section 3 can be used to compute the optimal sequential plan for a conjunctive query. Note that any conjunctive query φ over $X_1, \ldots, X_m$ can be written as $varphi = \bigwedge_{i=1}^{m} \phi_i(l_i \leq X_i \leq r_i)$. If we observe the value of some query attribute X_i, then either $varphi$ is proven to be false, or we need to observe other query attributes. In general, the choice of next query attribute will depend on the particular observed value of X_i, yielding a complex conditional plan. However, we can redefine our optimization problem to restrict potential conditioning predicates to be only the query predicates $\phi_i(l_i \leq X_i \leq r_i)$. If ϕ_i is observed to be false, we can terminate the search as above. Otherwise, we continue by choosing another predicate ϕ_j, and thus observing attribute X_j. If ϕ_j is false, we have proven that φ is false, otherwise we simply recurse. Clearly, this procedure will lead to a sequential order over attributes. The same optimal dynamic programming algorithm presented in the previous section can be applied in this redefined version of the problem to obtain this optimal sequential plan. This redefined problem is specified as follows:

- We redefine each query attribute X_i by $X'_i \in [1, 2]$, where $X'_i = 2$ if $X_i \in [l_i, r_i]$ and $X'_i = 1$ otherwise. Non-query attributes $X_{m+1}, \ldots, X_n$ are removed.
- In terms of these new attributes, our query now becomes $varphi' = \bigwedge_{i=1}^{m} \phi'_i(X'_i = 2)$. Note that φ is true if and only if φ' is true.

We can view this redefinition of the problem as a rediscretization of the possible values of the attributes into 2 bins, *i.e.*, $X_i \in [l_i, r_i]$ and $X_i \notin [l_i, r_i]$. We must also define the cost and probabilities in this redefined problem, which, of course, depend on the input ranges $R_1, \ldots, R_n$ of OPTSEQUENTIAL:

- As in the original problem, the cost for acquiring each X'_i is equal to C_i, if $R_i = [1, K_i]$, and 0 otherwise.
- The joint probability for the new attributes $X'_1, \ldots, X'_m$ $P(X'_1, \ldots, X'_m)$ is initialized to be the joint probability of the truth value of the query predicates, given the input ranges: $P(X'_1, \ldots, X'_m \mid R_1, \ldots, R_m)$.

The running time complexity of finding the optimal base plan, therefore, is $O(m2^m)$. Considering that the typical number of predicates in a query is likely to be small (less than 10), obtaining an optimal sequential plan is significantly more practical algorithm than solving the general optimal dynamic programming problem.

4.1.2 Optimal Sequential Plan for Arbitrary Queries

Due to lack of space, we will only present a brief sketch of our dynamic programming algorithm for computing an optimal sequential plan for arbitrary queries. We first note that, as all of our query predicates are unary, *i.e.*, ranges over single attributes, we can define a generalization of the rediscretization technique, which was only presented for conjunctive queries in the previous section: in general queries, for each attribute X_i, we now define v_i bins for X_i, where each bin is defined by the (sorted) extreme points of the ranges of each predicate over X_i. If we were to apply our optimal algorithm from Section 3 here, we would end up with a conditional plan over these v_i values of each new attribute, and not the desired sequential plan. However, we can write a dynamic programming recursion over possible orderings to attributes, using local optimal-

GREEDYSPLIT(φ,$R_1, \ldots, R_n$)
LET $C_{min} \leftarrow \infty$.
FOR $i \leftarrow 1$ **TO** n:
  IF RANGE OF X_i **IS** $[1, K_i]$, **THEN**:
    LET $C' \leftarrow C_i$.
  ELSE:
    LET $C' \leftarrow 0$.
  IF $C' < C_{min}$, **THEN**:
    FOR $x_i \leftarrow a+1$ **TO** b, WHERE $R_i = [a, b]$:
      LET $\left[\widehat{C}_{<x_i}, \widehat{\Gamma}_{<x_i}\right] \leftarrow$ OPTSEQUENTIAL(φ, $R_1, \ldots, [a, x_i - 1], \ldots, R_n$).
      LET $P_{<x_i} \leftarrow P(X_i \in [a, x_i - 1] \mid R_1, \ldots, R_n)$.
      LET $C' \leftarrow C' + P_{<x_i}\widehat{C}_{<x_i}$.
      IF $C' < C_{min}$, **THEN**:
        LET $\left[\widehat{C}_{\geq x_i}, \widehat{\Gamma}_{\geq x_i}\right] \leftarrow$ OPTSEQUENTIAL(φ, $R_1, \ldots, [x_i, b], \ldots, R_n$).
        LET $P_{\geq x_i} \leftarrow (1 - P_{<x_i})$.
        LET $C' \leftarrow C' + P_{\geq x_i}\widehat{C}_{\geq x_i}$.
        IF $C' < C_{min}$, **THEN**:
          LET $C_{min} \leftarrow C'$.
          LET $T \leftarrow T(X_i \geq x_i)$, $\widehat{\Gamma}_{<} \leftarrow \widehat{\Gamma}_{<x_i}$, $\widehat{\Gamma}_{\geq} \leftarrow \widehat{\Gamma}_{\geq x_i}$.
RETURN $\left[C_{min}, T, \widehat{\Gamma}_{<}, \widehat{\Gamma}_{\geq}\right]$.

Figure 6: Greedy selection of binary splitting point.

ity on the problems defined by sub-sequences of attributes. The rediscretization now allows us to compute the probability that observing a particular attribute is sufficient to prove the truth value of the query φ, by only considering v_i values of each attribute. The complexity of this algorithm is dominated by the complexity of computing these probabilities, which, in the worst case, is $O(v^m)$, where v is the maximum of the ranges of the attributes after rediscretization.

4.2 Greedy binary splits

Recall that the exhaustive algorithm described in Section 3 finds the optimal split point for each problem, by considering the optimal values of each resulting subproblem. Instead, we focus on locally optimal greedy splits. For a problem, we define the *locally optimal binary split* to be the split point that results in maximum *immediate benefit* over the optimal sequential plan. More formally, for a problem $Subproblem(\varphi, R_1, \ldots, R_n)$, the locally optimal binary split, GREEDYSPLIT$(\varphi, R_1, \ldots, R_n)$ is defined as a greedy version of Equation (5):

$$\begin{aligned}\text{GREEDYSPLIT}(\varphi, R_1, \ldots, R_n) = \min_i \min_{x_i \in [a_i+1, b_i]} \quad & C'_i + \\ & P(X_i \in [a_i, x_i - 1] \mid R_1, \ldots, R_n) \\ & \quad \widehat{J}(R_1, \ldots, [a_i, x_i - 1], \ldots, R_n) + \\ & P(X_i \in [x_i, b_i] \mid R_1, \ldots, R_n) \\ & \quad \widehat{J}(R_1, \ldots, [x_i, b_i], \ldots, R_n), \end{aligned} \tag{6}$$

where $\widehat{J}(R_1, \ldots, R_n)$ is the expected cost of the optimal sequential plan starting from the ranges $R_1, \ldots, R_n$. Figure 6 shows the complete pseudo-code of the algorithm that finds the locally optimally split point, given a sub-routine OPTSEQUENTIAL that computes the optimal sequential plan for a subproblem.

4.3 Greedy planning algorithm

Our greedy planning algorithm uses the greedy splits described above to efficiently find a good conditional plan for the query. The algorithm maintains a current decision list plan Γ, which is initially defined to be just a leaf with the root node. Each leaf n_i in the current plan stores an optimal sequential plan $\widehat{\Gamma}$ for its subproblem, and the optimal greedy split plan

GREEDYPLAN(φ,MAXSIZE)
LET $\widehat{\Gamma} \leftarrow$ OPTSEQUENTIAL(φ,$[1, K_1], \ldots, [1, K_n]$).
LET $\left[\overline{C}, T(X_j \geq x_j), \widehat{\Gamma}_{<x_j}, \widehat{\Gamma}_{\geq x_j}\right] \leftarrow$ GREEDYSPLIT(φ, $[1, K_1], \ldots, [1, K_n]$).
LET $n_1 \leftarrow \left[\widehat{\Gamma}, T(X_j \geq x_j), \widehat{\Gamma}_{<x_j}, \widehat{\Gamma}_{\geq x_j}, [1, K_1], \ldots, [1, K_n]\right]$.
LET $\Gamma \leftarrow \{n_1\}$.
ADD TO QUEUE n_1 WITH PRIORITY $C(\widehat{\Gamma}) - \overline{C}$.

WHILE $|\Gamma| <$ MAXSIZE:
  REMOVE TOP OF QUEUE:
    $n_i = \left[\widehat{\Gamma}, T(X_j \geq x_j), \widehat{\Gamma}_{<x_j}, \widehat{\Gamma}_{\geq x_j}, R_1, \ldots, R_n\right]$, WHERE $R_j = [a, b]$.
  LET $\left[\overline{C}, T(X_u \geq x_u), \widehat{\Gamma}_{<x_u}, \widehat{\Gamma}_{\geq x_u}\right] \leftarrow$ GREEDYSPLIT(φ, $R_1, \ldots, [a, x_j - 1], \ldots, R_n$).
  LET $n'_i \leftarrow \left[\widehat{\Gamma}_{<x_j}, T(X_u \geq x_u), \widehat{\Gamma}_{<x_u}, \widehat{\Gamma}_{\geq x_u}, R_1, \ldots, [a, x_j - 1], \ldots, R_n\right]$.
  ADD TO QUEUE n'_i WITH PRIORITY $P(R_1, \ldots, [a, x_j - 1], \ldots, R_n)\left(C(\widehat{\Gamma}_{<x_j}) - \overline{C}\right)$.
  LET $\left(\overline{C}, T(X_v \geq x_v), \widehat{\Gamma}_{<x_v}, \widehat{\Gamma}_{\geq x_v}\right) \leftarrow$ GREEDYSPLIT(φ, $R_1, \ldots, [x_j, b], \ldots, R_n$).
  LET $n''_i \leftarrow \left[\widehat{\Gamma}_{\geq x_j}, T(X_v \geq x_v), \widehat{\Gamma}_{<x_v}, \widehat{\Gamma}_{\geq x_v}, R_1, \ldots, [x_j, b], \ldots, R_n\right]$.
  ADD TO QUEUE n''_i WITH PRIORITY $P(R_1, \ldots, [x_j, b], \ldots, R_n)\left(C(\widehat{\Gamma}_{\geq x_j}) - \overline{C}\right)$.
  LET $\Gamma \leftarrow \Gamma \cup \{n_i \rightarrow n'_i, n_i \rightarrow n''_i\}$.
RETURN Γ.

Figure 7: Greedy planning algorithm.

defined by Equation (6). The benefit of applying the greedy binary split at n_i over the optimal sequential plan is given by the expected cost of the sequential plan, $C(\widehat{\Gamma})$ minus the expected cost of the optimal greedy split $\overline{C}$ computed by Equation (6). We maintain a priority queue over leaves of the current plan. A particular leaf n_i, whose subproblem is $R_1, \ldots, R_n$, is inserted in the queue with a priority given by the difference $C(\widehat{\Gamma}) - \overline{C}$ weighed by the probability that our plan will reach the subproblem in leaf n_i, $P(R_1, \ldots, R_n)$, as the expected gain of expanding n_i is $P(R_1, \ldots, R_n)\left(C(\widehat{\Gamma}) - \overline{C}\right)$.

Our greedy algorithm chooses the next leaf to expand from the queue based on this priority.

Figure 7 illustrates the complete greedy algorithm. We start with a sequential plan for the root node, and a single split on this node. We then iterate through the queue. We remove the top leaf n_i, and create two children leaves n'_i and n''_i, based on the greedy split from n_i. These new leaves are then added to the queue with appropriate priorities.

The running complexity of this algorithm can be seen to be $O(nKC_{optseq}N + nKK_p)$, where C_{optseq} denotes the cost of computing optimal sequential plan for a subproblem, N denotes the number of splits performed by the algorithm, and K_p denotes the complexity of computing the probabilities required by this algorithm. In particular, for conjunctive queries, the complexity of this algorithm is $O(nKm2^mN + nKK_p)$.

5. EFFICIENT COMPUTATION OF PROBABILITY DISTRIBUTIONS

Until now, we have ignored the issue of computing the event probabilities required by our planning algorithms. In particular, the algorithms require the computation of conditional probabilities of predicates given a set of predicates that are already known to be satisfied. In this section, we will discuss how these

probability computations can be performed efficiently, given a historical dataset of samples.

Consider the case where each required conditional probability is estimated from counts from a dataset $\mathcal{D}$ of d tuples as described in Section 2.3. Consider a subproblem generated during the execution of either the EXHAUSTIVEPLAN algorithm or the GREEDYSPLIT algorithm, $Subproblem(\varphi, R_1 = [a_1, b_1], \ldots, R_n = [a_n, b_n])$, and let the part of the dataset that satisfies the conditions $X_i \in R_i, \forall i$, be $\mathcal{D}(R_1, \ldots, R_n)$.

5.1 Probabilities for exhaustive planning

While solving a subproblem, the probabilities that EXHAUSTIVEPLAN (Figure 5) needs to compute are:

$$P_{<x_i} = P(X_i \in [a_i, x_i - 1] \mid R_1, \ldots, R_n),\ \forall i,\ \forall a_i < x_i \leq b_i.$$

As we can see, all the probabilities required by this step of the algorithm (ignoring any recursive calls for subproblems) are probabilities conditioned on $(X_1 \in R_1) \wedge \ldots \wedge (X_n \in R_n)$. Note that $P(x_i \mid R_1, \ldots, R_n)$ is simply an independent normalized histogram of X_i for the data in $\mathcal{D}(R_1, \ldots, R_n)$. We can compute these histograms for every attribute $X_1, \ldots, X_n$ with one pass over the dataset. Once we have $P(x_i \mid R_1, \ldots, R_n)$, we can compute the probabilities of the required ranges incrementally by noting that:

$$P_{<(x_i+1)} = P_{<x_i} + P(x_i \mid R_1, \ldots, R_n). \tag{7}$$

Thus, we can compute the probabilities of all required ranges in time only $O(|\mathcal{D}|\, nK + nK)$, where $K = \max_i K_i$.

We must also consider generating the dataset for each subproblem called recursively from $R_1, \ldots, R_n$. Here, we can create an index independently for each attribute, for each value of the attribute in time $O(|\mathcal{D}|\, nK)$. We can now use a similar technique to the one used for range probabilities in Equation (7): the set of indices for the range $[1, x_i]$ is equal to the set of indices for $[1, x_i - i]$ union with the indices for x_i. Thus, selecting the dataset for all resulting subproblems is also $O(|\mathcal{D}|\, nK + nK)$. Therefore, the complexity of the exhaustive algorithm using the dataset to estimate probabilities is $O(|\mathcal{D}|\, (nK)^2 K^{2n})$.

5.2 Probabilities for greedy planning

When the GREEDYSPLIT is considering a conditioning predicate $X_i \in [a_i, x_i - 1]$, it requires computation of two types of distributions:

- $P_{<x_i} = P(X_i \in [a_i, x_i - 1] \mid R_1, \ldots, R_n)$ as above,
- a joint probability distribution over the rediscretized attributes in the query predicates, conditioned on $X_1 \in R_1 \wedge \ldots \wedge X_n \in R_n$ and on the particular choice of splitting attribute and assignment.

The conditional probabilities $P_{<x_i}$ can be computed incrementally as in Equation (7). The joint distribution over the rediscretized attributes can be similarly computed from a normalized joint histogram over each attribute X_i and over the rediscretized attributes $X'_1, \ldots, X'_m$. We can then use an incremental rule similar to the one in Equation (7) to compute the required joint probabilities. Thus, the total computation required in at each split is $O(|\mathcal{D}|\, nK)$ to create the histograms and $O(nK + nKv^m)$ to incrementally update the joint distributions. Using these optimizations, if the number of subproblems solved by GREEDYSPLIT is N, then the total running time is $O(NnK(C_{optseq} + v^m))$, or, for conjunctive queries, $O(NnKm2^m)$.

6. EVALUATION

We now experimentally evaluate the algorithms described in the above sections. We demonstrate that our algorithm offers a substantial performance increase over a non-conditional query execution engine for a wide class of queries over real-world data sets. We also show that our GreedySplit heuristic closely approximates the exhaustive algorithm and that our techniques scale with respect to the number of query predicates, the domain size of the attributes in the query, and the amount of historical data. Finally, we show that our heuristic performs well with a small number of splits, corresponding to a small plan size that may be required in memory-constrained environments.

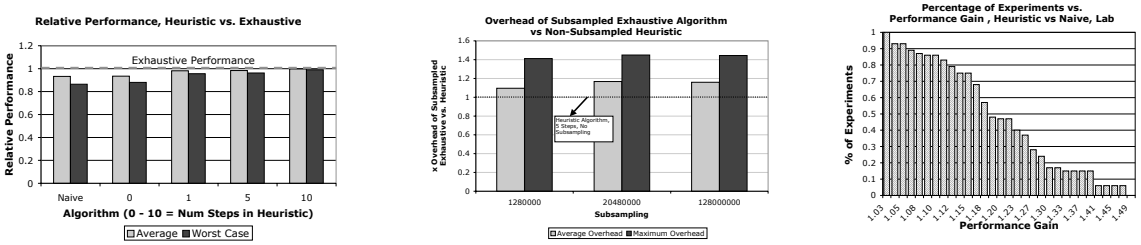
We implemented our algorithms in Java on traditional PC hardware. We evaluate their performance by costing and running plans on this centralized PC; we reserve implementing a plan executor that runs on sensor network hardware for future work. We believe our techniques are readily implementable on sensor networks, as the energy overhead required to execute a conditional plan, especially when compared to the costs of acquiring sensor readings, will be small.

We present results for several real world data sets collected from TinyOS-based sensors. We use two data sets: a *lab* data set of 400,000 light, temperature, and humidity readings collected every two minutes for several months from about 45 motes. These three environmental attributes are quite expensive to acquire. These readings also include the *id* of the generating sensor, time of day, and battery voltage; we assign these cheap attributes a marginal sampling cost. Our second data set, *garden* was collected from a set of 20 Berkeley motes deployed in a forest; each device was equipped with two temperature and light sensors, as well as a single sensor for humidity and air pressure.

We also refer to two variants of the *lab* data set. To generate *lab8* we concatenated the light, humidity, and temperature readings from 8 of these sensors together into a single table. We also generated a *historical* version of this data set where historical readings from twenty minutes ago concatenated to each of the current sensor readings. We set the cost of accessing these historical attributes to be 1/10th the cost of accessing the attribute itself, to reflect the fact that for a sensor to use a reading of this sort, it must be acquired at least once every ten sample periods.

Due to the complexity of the exhaustive algorithm, we were required to reduce the domain size of the attributes significantly by *subsampling* each attribute by some integer factor. We measure the amount of subsampling as the product of the reduction factors of all of the query attributes. This is relative to the non-subsampled domain size, which we also measure as the product of the sizes of the attribute domains. For our lab data set with six attributes, this is approximately 10^{14}.

Except as noted below, our plans generated by our algorithms are built using a set of *training* data, and evaluated on a disjoint set of *test* data. Readings from these two sets are from non-overlapping time windows, simulating the case where historical data is used to generate a model that is later run for days or weeks within a sensor network. Our experiments were conducted by automatically generating a variety of range queries. As we explain below, we found that queries with predicates with very low selectivity do not exhibit a significant performance gain. For this reason, most predicates generated for our experiments are satisfied by a large (approximately 50%) portion of the data set. For each query, we select, uniformly and at random, the left endpoint of the range of the query; the width of each predicate is chosen to be two standard deviations of the



(a) (b) (c)

Figure 8: Quality of plans: (a) exhaustive algorithm versus the naive and heuristic algorithms on a subsampled version of the lab data set, the three heuristic bars represent different numbers of splits in the heuristic algorithm; (b) impact of using a subsampled version of data to train the exhaustive algorithm (versus the heuristic algorithm with 10 splits); (c) cumulative frequency of performance gain for experiments over the lab data set, the frequency of at a particular x-coordinate indicates the fraction of experiments that did at least that well.

attribute which it is over.

We compare the performance of our algorithms to a traditional, non-conditional query planner. We refer to the traditional algorithm *naive* algorithm, used by most traditional query optimizers[15, 10, 2]. Naive simply orders predicates by $\frac{cost}{1-selectivity}$, where selectivity is simply the marginal probability that the predicate does not output a tuple as computed from the historical data.

6.1 Exhaustive versus heuristic algorithms

In our first experiment, we compare the performance of the exhaustive algorithm to our greedy heuristic. Our test queries consist of three predicate queries (generated as specified above) over the lab data. To allow the exhaustive algorithm to run, we subsample by a factor of 1.2×10^7 for the experiments in this section. We note that subsampling significantly reduces the improvement that our algorithms can obtain over a traditional optimizer; our results in the next section illustrate much more substantial gains when it is disabled.

Figure 8(a) shows the average performance relative to the exhaustive algorithm. We vary the number of split points in our heuristic plans from 0 to 10. Each bar represents the average of 95 different queries. Notice that in all cases, our algorithms outperform naive, and that the both the worst case and average performance of the heuristic with 10 splits is very close to the performance of exhaustive. We tried this experiment with a variety of subsamplings and data sets and obtained similar results; space constraints preclude the inclusion of these graphs. We also note that we did not separate test data from training data to avoid possible performance variations that a disjoint test and training set might introduce.

The space and time complexity of the exhaustive algorithm are very high. On a 2.4GHz Pentium 4 with 1 gigabyte of RAM, the largest problems we could solve were still several orders of magnitude smaller than the smallest of our real-world data sets. We show complete scalability results for all of our algorithms in Section 6.3.

As an alternative to the heuristic algorithm presented above, we considered a simple modification to the exhaustive algorithm where we ran it over a subsampled data set and used the generated plan for query evaluation (over the unmodified data set). We compared the performance the this variant of the exhaustive algorithm on the lab data set to the performance of our heuristic on the unmodified data; the results are shown in Figure 8(b). In this experiment, we varied the amount of subsampling (e.g. product of bucket sizes). As before, each bar represents the average (or maximum) of 95 trials. Notice that the subsampled optimal algorithm performs substantially worse than our heuristic on this data set, even with large subsamplings.

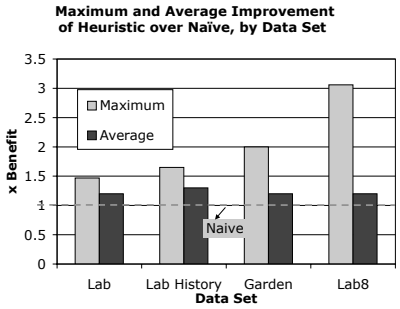


Figure 9: Average and maximum performance gain of the 10-split heuristic algorithm versus the naive algorithm.

In general, as these results show, subsampling of this sort is probably bad idea, because it tends to obscure interesting correlations in the data by grouping together parts of the attribute space that may be partially correlated with other attributes, diluting the benefit our algorithms can obtain by exploiting such correlation.

6.2 Heuristic algorithm performance

Next, we ran experiments with our heuristic algorithm on the two data sets described above (including the lab8 and historical variants of the lab data), running 100 queries (generated as above) in each experiment. In each case, we used no subsampling, and run the heuristic algorithm for 10 splits. The queries were as follows: for the lab data set, we had three predicates over light, temperature and humidity; we also allowed the algorithm to condition on sensor id, battery voltage, and time of day. For the historical version of this data set, we added historical values of light and temperature as conditional variables. In the lab8 data set, we had eight predicates over temperature on the eight different sensors, with sensor id, voltage, and time of day available for conditioning. In the garden data set, we used four predicates over light, both temperature sensors, and humidity, with time of day and location (e.g., altitude within the forest) available as conditioning predicates. Figure 9 shows the average and maximum performance gain over naive for each of these experiments. Notice that in all cases the average gain is over 20%. The maximum gain is a factor of three in the lab8 data set; in the garden data set, it is a factor of two.

Figure 8(c) shows a cumulative frequency of experiment performance over the lab data set, where each bin indicates the number of experiments that did at least as well as the performance at that bin. Notice that the gains are fairly uniformly

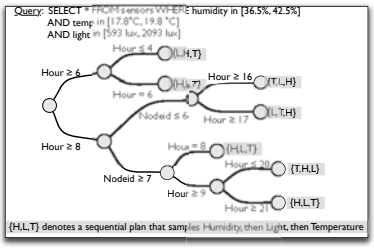


Figure 10: Example of a conditional plan generated by the heuristic algorithm for a query over our lab data set looking for instances when it is both bright, cool, and dry.

distributed, and that there is some gain in every case. This latter point suggests that our models are not overfitting on the training data, since that would result in some experiments with a negative performance gain. Due to space constraints, we omit CDFs for other data sets; they show similar patterns and distributions of gains, except that there are very few experiments that demonstrate gains in excess of a factor of two (when such gains are to be had at all).

6.2.1 Detailed Plan Study

Figure 10 illustrates an example of a real plan produced by our system for one of our test queries from the lab data set. The total performance gain for this plan is about 20% over naive. In this case, the query looks for sensors that are reading cool temperatures, and relatively high light intensities – indicating, perhaps, that someone is working in the lab at night when it is typically cold and dark. None of the predicates in this query have a particularly low marginal selectivity, but the total selectivity of the query will be low, as there there are few cases when it is both cold and bright in our lab. Notice that in this case, our algorithm first conditions on the hour in the day: when it is morning (hours 0 - 6), it prefers to sample light first, as very early in the morning the lab is unused and it is dark, so this predicate will fail. In the afternoon, additional conditional predicates on nodeid are introduced. The split at nodeid ≤ 6 represents a group of sensors (1-6) in a common part of the lab that is not used at night, so darkness is highly correlated with time of day. At the other sensors (nodeid ≥ 7), the lab is sometimes in use until late into the night. Thus, in this part of the lab, light may not be correlated with time of day. Interestingly, it appears that humidity is correlated with time of day, as the generated plan samples humidity first late at night. This is likely because the temperature control system in our building is turned off at night, and air conditioning and heating tends to keep the humidity low.

6.2.2 Best-case Analysis

We note that in several data sets, Figure 9 shows a dramatic best case performance gain of several times over the naive algorithm. This is due to the inability of the naive algorithm to account for correlations; for example, with the lab8 data set, the light and temperature readings from each of the sensors are highly correlated. In this case, the query contained eight predicates, each with selectivities of about 50%. Seven of the predicates selected for high temperatures from a sensor; the eighth was looking for low temperatures, so the total selectivity of the query was low. It happened that this eighth predicate was less selective than the others, so naive placed it at the end of the query. In a non-conditional plan, this is a particularly bad decision, as if a tuple passes one of the seven correlated predicates, it will pass all of them, requiring the acquisition of seven attributes, only to fail on the eighth predicate. The heuristic algorithm was able to use the correlation between the sensors to observe that when one of the correlated attributes passes a tuple, they all will, and so its second step should be to check the eighth predicate. Though this example makes naive look remarkably bad, we believe it will occur frequently in monitoring scenarios. For example, a user may wish to determine if a particular sensor is malfunctioning by selecting sets of readings where that sensor reads differently from a set of (supposedly well correlated) neighbors.

6.2.3 Discussion

Finally, we observe that, for a number of queries, the *naive* approach works quite well. It does best in scenarios when there is one predicate of very low selectivity that it can place early in the plan. It does poorly when most predicates have approximately the same selectivity, as it cannot choose a good order, whereas our techniques can use correlations to distinguish predicates in such cases. Naive performs particularly badly in cases like the one we identified with the multi-sensor lab data set, where the query contains a sequence of correlated predicates of about the same selectivity.

To quantify the effect of selectivity on the relative performance of our algorithms versus the naive algorithm, we studied the best and worst performing 5% of queries in the lab data set (whose performance distribution is shown in Figure 5.2). The average variance of the *apriori* probability of query predicates being satisfied in the best case was 0.00076, whereas it was 0.095 in the worst case. In all of the best cases, the probability of each predicate being satisfied was approximately 50%, and in the worst cases, there was at least one predicate with very low selectivity (less than 5%.)

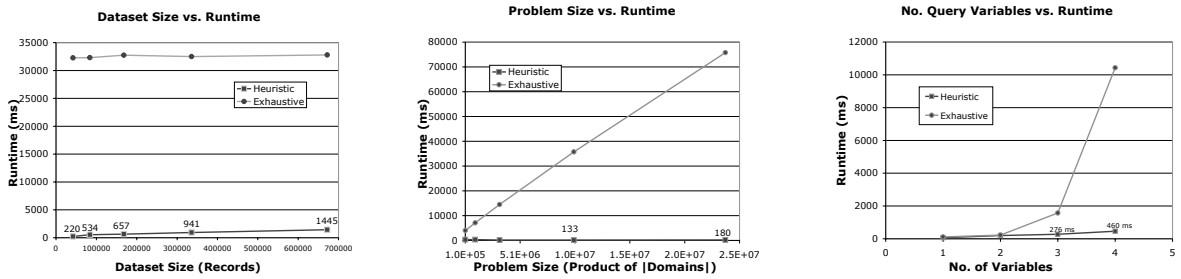
6.3 Scalability

In this section, we study the scaling of the exhaustive and heuristic algorithm versus number of predicates, attribute domain size, and amount of historical data. Recall from our discussion above that we expect the performance of our heuristic algorithm to scale linearly in the size of the data set, linearly with domain size, and exponentially (base 2) in the number of query variables. The exhaustive algorithm is also linear in the size of the data set, but is polynomial in the size of the domain size and exponential in the number of query variables, where the base of the exponent is the domain size. Our scalability experiments verify that our implementation obeys the complexity bound given above. We briefly summarize these results in this section, omitting a detailed discussion of experimental parameters due to space constraints.

Figure 11(a) shows total runtime versus the the amount of historical data, comparing the heuristic algorithm running for 10 steps to the exhaustive algorithm (over the garden data set with three query predicates.) In this experiment we varied the amount of data by replicating the lab data set from 1-16 times. The linear scaling with respect to data set size of both algorithms makes them able to work effectively even when the data sets are quite large.

Figure 11(b) shows the scaling of the runtime of our algorithms versus the product of the domain sizes of the query attributes (i.e., amount of subsampling) on the (unreplicated) garden data set, show how the polynomial scaling of the the exhaustive algorithm slows it down for larger domain sizes.

Figure 11(c) shows the performance of our algorithms with respect to the number of variables in the query. The runtime of both algorithms increases with the number of variables, but that the exhaustive algorithm scales much more dramatically, due to



(a) (b) (c)

Figure 11: Running time versus: (a) amount of historical data; (b) size of attribute domains; (c) number of query variables.

the larger exponent base.

We also looked at the runtime of the heuristic algorithm with respect to the number of splits, again a three variable version of the data set. Due to space constraints, we omit the graph, but the runtime ranged from 176ms with 1 split to 841ms with 20 splits. Plan size (in extremely verbose, human readable form), ranged from 357 characters for the a plan with no splits to 2373 bytes for a plan with 20 splits, representing a six times increase in size.

Finally, we measured the memory requirements of our algorithms, but again do not plot the results due to space constraints. In general, the heuristic algorithm's memory requirements scale with size of the data set, ranging from about 3 megabytes on the basic garden data set to 49 megabytes on same data set replicated 16 times. The exhaustive algorithm scales with domain size, ranging from 34 megabytes on the smallest domain we tested with to 510 megabytes in the largest.

7. APPLICATIONS AND EXTENSIONS

This paper focuses on building the basic data-structures and algorithms required for obtaining good conditional plans for complex acquisitional queries. This section outlines several extensions built on this basic framework.

Approximate answers: Thus far, we have shown how to use our conditional planning techniques that generate the exact truth value of the query φ. Our techniques also provide an effective method for obtaining approximate query answers. In particular, every time our recursive algorithms visit a subproblem Subproblem(φ,$R_1, \ldots, R_n$), we first evaluate whether the truth value of φ can be determined from the ranges $R_1, \ldots, R_n$. In approximate queries, we, in addition, evaluate the probability that φ is true given the ranges, $P(\varphi \mid R_1, \ldots, R_n)$. If, for example, the user defines a 5% tolerance, then we can stop as soon as this probability is lower than 5%, returning $\varphi = \texttt{F}$, higher than 95%, returning $\varphi = \texttt{T}$. Additionally, the user can define different tolerances for false positives and false negatives appropriately. These approximate framework allows us to answer queries over very expensive attributes, without ever evaluating them, if observing low-cost attributes is sufficient to estimate $P(\varphi \mid R_1, \ldots, R_n)$ within the required tolerances.

Distributed query processing: Although we have focused our presentation on conditional plans that are expected to run on a single node of a network, an interesting application of our approach is the processing of queries over attributes distributed in the network. For example, we may have a query involving the temperature in multiple sensor nodes. Our planning approach can now take into account the cost acquiring an attribute locally, and the cost of forwarding a query to another node. In this case, our cost function will depend on the previous location of the query. If we need to acquire an attribute in the same location, we do not pay the communication cost, and the location variable remains unchanged. If we decide to acquire an attribute in another location, then we must pay the communication cost (depending on the route, network quality, etc.) plus the attribute acquisition cost, at which point the location variable is set to the new location. Given a model of network connectivity and routing costs, this distributed extension of our (exhaustive and heuristic) algorithms is straightforward.

Complex acquisition costs: We have focused on a very simple cost function: the independent cost of acquiring a single variable. In general, however, we may have significantly more complex cost functions. For example, motes have sensor boards with multiple sensors that are powered up simultaneously. Thus, the cost of acquiring a reading can be decomposed as the high cost of powering up the board, plus a low cost for a reading of each sensor in the board. In our algorithms, after we first split on an attribute, the cost of adding another split is 0. Similarly, we can make the cost of the first attribute in a sensor board high, but after some attribute is observed, the others will have low cost.

Graphical Models: The methods presented in the previous section, using historical data to estimate conditional probabilities, suffer from two issues: First, these algorithms are linear in the size of the dataset, which can be very large. Second, after each split on a predicate, each subproblem will be consistent with at most half of the data. Thus, the amount of data available to estimate probabilities decreases exponentially with the number of splits. After several splits, our probability estimates will thus have very high variance. This can result in choosing arbitrary plans, that may turn out to be significantly worse in reality than in the training data.

An alternative is to build a compact model of the data. *Probabilistic graphical models* provide a compact representation of complex, exponentially-large distributions, by exploiting conditional independences between attributes. These models offer two significant advantages: First, there are several algorithms that allow us to compute the conditional probabilities efficiently by exploiting structure in the graphical model [5]. Second, by exploiting conditional independences, we can often represent the required joint distribution with a polynomial number of parameters. Thus, this representation is significantly less susceptible to the overfitting observed when estimating probabilities from a dataset directly. Due to lack of space, we refer the reader to the book by Cowell *et al.* [5] for details.

Query processing in other environments: Up to this point, we have focused on the application of conditional plans in sensor networks.They are equally applicable in other wide-area environments; for example, on the web, the latency to acquire individual data items in quite high, and the data may may exhibit correlations that can be exploited using conditional

plans. Similarly, in compressed databases [3], the cost of acquiring attributes may include the cost of decompression, which can be be very high. Conditional plans can reduce the amount of decompression required to execute a query.

Our techniques can also be applied to traditional database query processing. For example, our techniques generalize easily to star queries containing only key-foreign key join predicates, can be thought of as expensive "selections" on the relation at the center of the star (commonly referred to as the *fact* table), and conditional plans can be used to exploit correlations between the *dimension* tables.

Generalization to other types of queries: Thus far, we have focused our attention on conjunctive queries. Other interesting types of queries in sensor network queries are existential queries, or queries that use the "LIMIT" clause. For example, we may only be interested in finding out if there exists a sensor that is recording high values of light and temperature. We can use conditional plans to significantly reduce the number of acquisitions made, by "guessing" which of the sensors are most likely to satisfy the predicates, possibly using other low-cost attributes in the process.

8. RELATED WORK

Our idea of conditions plans is quite similar to *parametric query optimization* [14, 8, 4, 7], where part of the query optimization process is postponed until the runtime. Typically, these techniques choose a set of query plans at query optimization, and a set of conditions that are used to select a plan at runtime. This earlier work differed substantially from ours in two essential ways: First, in these traditional approaches, the plan to be run for an entire query is selected dynamically at run-time; thus, even if correlations were taken into account by these approaches, per-tuple variations, which we have seen to be prevalent and widely exploitable, could not be accounted for. Second, these approaches did not generate nodes in the choose plan by observing correlations in the data, so could not exploit the same relationships in the data as our approach.

Adaptive query processing techniques [11] attempt to reoptimize query execution plans during query execution itself. We believe that the idea of conditional plans that we propose is both orthogonal and complementary to adaptive query processing. If sufficiently accurate information about the data is available (as we assume in this work), then conditional plans can reap many of the benefits of adaptive query processing technique apriori. However, in many cases, such information may not be available, and adaptive techniques must be used.

Earlier work on expensive predicates [10, 2] talks about how to optimize queries with expensive predicates. All these techniques however produce a single sequential plan in the end, and to our knowledge, none of this work has considered correlated predicates. Shivakumar *et al.*, [19] propose using low-cost predicates to avoid evaluating expensive predicates. Their approach also constructs a sequential plan in the end, and the final query output may contain false positives, or may miss certain answers. Our approach, on the other hand, guarantees correct execution of the original query in all cases.

Madden *et. al.*, [16] propose the idea of acquisition query processing where the cost of acquiring attributes is explicitly modeled, though their focus was entirely on sensor network query processing. In this paper, we have generalized their basic idea, and have proposed an approach to significantly speed up query processing in such environments. Web querying is another domain where the idea of acquisitional query processing, and the techniques we propose in this paper, can be very useful. Chen et al., [3] propose techniques to perform query optimization in compressed database systems, and also model the cost of acquiring attributes explicitly. The techniques we propose in this paper can be extended in straightforward manner to their scenario.

9. CONCLUSIONS

In this paper, we showed how to exploit correlations between attributes in a database system by modifying the query optimizer to produce *conditional* plans that significantly outperform plans produced by traditional database optimizers. We showed specifically how these correlations can be used to optimize the performance of multi-predicate selection queries with attributes that have high acquisition costs [16] which frequently occur in distributed systems such as sensor networks and the Internet. We developed planning algorithms for generating such conditional plans given a historical data set with correlations. Our experimental results demonstrate the significant performance gains if the query optimizer is modified to take into account these correlations.

10. REFERENCES

[1] Planetlab. http://www.planet-lab.org.

[2] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *TODS*, 24(2):177–228, 1999.

[3] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *ACM SIGMOD*, 2001.

[4] R. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD*, 1994.

[5] R. Cowell, P. Dawid, S. Lauritzen, and D. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Spinger, New York, 1999.

[6] I. Crossbow. Wireless sensor networks (mica motes). http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.

[7] S. Ganguly. Design and analysis of parametric query optimization algorithms. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases*, 1998.

[8] G. Graefe and K. Ward. Dynamic query evaluation plans. In *SIGMOD*, 1989.

[9] R. Greiner, R. Hayward, and M. Molloy. Optimal depth-first strategies for and-or trees. In *AAAI/IAAI*, 2002.

[10] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *TODS*, 23(2):113–157, 1998.

[11] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, and D. C. K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

[13] L. Hyafil and R. Rivest. Constructing optimal binary decision trees is np-complete. *IPL*, 1976.

[14] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *18th International Conference on Very Large Data Bases*, 1992.

[15] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, 1986.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *ACM SIGMOD*, 2003.

[17] J. Polastre. Design and implementation ofwireless sensor networks for habitat monitoring. Master's thesis, UC Berkeley, 2003.

[18] G. Pottie and W. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51 – 58, May 2000.

[19] N. Shivakumar, H. Garcia-Molina, and C. Chekuri. Filtering with approximate predicates. In *VLDB*, 1998.