A Probabilistic Optimization Framework for the Empty-Answer Problem

Davide Mottin
University of Trento
mottin@disi.unitn.eu

Alice Marascu*
IBM Research-Ireland
alice.marascu@ie.ibm.com

Senjuti Basu Roy
U of Washington Tacoma
senjutib@uw.edu

Gautam Das
UT Arlington & QCRI
gdas@uta.edu - gdas@qf.org.qa

Themis Palpanas
University of Trento
themis@disi.unitn.eu

Yannis Velegrakis
University of Trento
velgias@disi.unitn.eu

ABSTRACT

We propose a principled optimization-based interactive query relaxation framework for queries that return no answers. Given an initial query that returns an empty answer set, our framework dynamically computes and suggests alternative queries with less conditions than those the user has initially requested, in order to help the user arrive at a query with a non-empty answer, or at a query for which no matter how many additional conditions are ignored, the answer will still be empty. Our proposed approach for suggesting query relaxations is driven by a novel probabilistic framework based on optimizing a wide variety of application-dependent objective functions. We describe optimal and approximate solutions of different optimization problems using the framework. We analyze these solutions, experimentally verify their efficiency and effectiveness, and illustrate their advantage over the existing approaches.

1. INTRODUCTION

The web offers a plethora of data sources in which the user has the ability to discover items of interest by filling desired attribute values in web forms that are turned into conjunctive queries and get executed over the data source. Examples include users searching for electronic products, transportation choices, apparel, investment options, etc. Users of these forms often encounter two types of problems - they may over-specify the items of interest, and find no item in the source satisfying all the provided conditions (the *empty-answer problem*), or they may under-specify the items of interest, and find too many items satisfying the given conditions (the *many-answers problem*). This is because the majority of such searches are often of exploratory nature since the user may not have a complete idea, or a firm opinion of what she may be looking for.

In this paper we focus on the empty-answers problem. A popular approach to cope with empty-answers is *query relaxation*, which attempts to reformulate the original query into a new query, by removing or relaxing conditions, so that the result of the new query is likely to contain the items of interest for that user.

*Work primarily done while the author was with the University of Trento.

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. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.

Proceedings of the VLDB Endowment, Vol. 6, No. 14

Copyright 2013 VLDB Endowment 2150-8097/13/13... $ 10.00.

A typical approach in query relaxation is the *non-interactive* approach (e.g., [9, 15, 16, 17]), in which a set of alternative queries - with some of the original query conditions relaxed - is suggested to the user in order to select the one he or she prefers the most. This non-interactive approach has a number of limitations. The very large number of candidate alternative queries that may be generated, due to the numerous combinations of conditions that can be removed from it, makes the relevant systems hard to design, and cumbersome to use by naive users.

We advocate here that a different and more palatable approach for many application scenarios is the *interactive (or navigational)* approach, in which the user starts from an original empty-answer query, and is guided in a systematic way through several "small steps". Each step slightly changes the query, until it reaches a form that either generates a non-empty answer, or any further change in the query conditions does not result into a non-empty answer. In addition to being effective for naive users, such an interactive approach is meaningful for scenarios in which the user interacts with the data source via a small device, i.e., a mobile phone, where the size of the screen does not allow many choices to be displayed at once. Another kind of applications are the customer-agent interactions that take place over the phone, as it happens during the purchase of a travel insurance, an airline ticket, the reservation of a holiday house, a car, etc. When the original user specifications cannot be satisfied, the communication of all the different alternatives to the customer by the agent is not possible, thus, the agent will have to guide the customer through small steps to adapt to the original request into one for which there are satisfactory answers. Such interactions allows the user more control in the selection process.

Our proposed approach for suggesting query relaxations is driven by a novel and principled probabilistic framework based on optimizing a wide variety of application-dependent objective functions. For instance, when a user wants to purchase an airline ticket (and her initial query returns empty-answer), this framework may be used to suggest alternative queries that lead her to airline tickets that are most "relevant" to her initial preference, or are most economical, etc. The framework may be used to suggest queries that lead the user to the more expensive airlines tickets, or tickets in flights that have many unsold seats; thereby maximizing the revenue/profit of the airlines company. It may also be used to suggest queries that lead the user to valid tickets in the shortest possible number of interactive steps; i.e., the objective is to minimize the time/effort the user spends interacting with the system.

Most prior query relaxation approaches for the empty-answers problem have been non-interactive, and/or do not support a broad range of objectives, e.g., conflicting situations where the objective is to maximize profit of the seller. We provide a detailed discus-

	Make	Model	Price	ABS	MP3	Alarm	4WD	DSL	Manual	HiFi	ESP	Turbo
t_1	VW	Touareg	$62K	1	0	0	0	0	1	0	1	0
t_2	Askari	A10	$206K	0	1	0	0	1	1	1	1	0
t_3	Honda	Civic	$32K	1	0	0	0	0	0	0	0	0
t_4	Porsche	911	$126K	0	0	0	0	1	0	1	1	0

Figure 1: An instance of a car database.

sion of related work in Section 8. Optimization-based interactive approaches have been proposed for the many-answers problem [6, 18, 20], however these papers only consider one narrow optimization goal, that of minimizing user effort. Moreover, we note that there is a fundamental asymmetry between the empty-answer and many-answers problems that precludes a straightforward adaptation of previous optimization-based query tightening techniques to query relaxation.

We provide a brief overview of our approach. At each interaction step, a relaxation is proposed to the user[1]. In order to decide what should be the next proposed relaxation, our system has to first compute the *likelihood that the user will respond positively* to a proposal, as well as quantify the *effectiveness of the proposal* with respective to the optimization objective. Intuitively, the likelihood (or probability) that a user responds positively to a proposed relaxation depends on (a) whether the user believes that the proposal is likely to return non-empty results, and (b) whether some of the returned items are likely to be highly preferred by the user (even though they only partially satisfy the initial query conditions). Since our system cannot assume that the user knows the exact content of the database instance, nor does it precisely know whether the user will prefer the returned results, we resort to a probabilistic framework for reasoning about these questions. This probabilistic framework is one of the fundamental technical contributions of this paper.

To quantify the effectiveness of a proposed relaxation, one needs to consider the probability with which the user will accept (or reject) it, the *value* of the expected results of the specific query (this depends on the application-specific objective function as briefly discussed earlier, and discussed in more detail in Section 3.2), and the further reformulations (relaxations) that can be applied to it in case it turns out that it still produces no results. All these elements together form a factor that determines the *cost* of a proposal. The problem then is to find the sequence of query proposals with the *optimum total cost* (the actual cost function may be maximized or minimized based on the specific optimization objective and discussed in Section 3). This is in general a challenging task, as we have to consider an exponential number of possible sequences of query proposals.

To cope with the above challenges, we have developed different strategies for computing the sequences of query reformulations (relaxations) and have materialized these strategies into the respective algorithms. The first algorithm, FullTree, is a baseline and is essentially an exhaustive algorithm that pre-computes a complete tree of all the possible relaxation sequences, the possible user responses to each relaxation proposal, and the cost of the sequences. The second algorithm, FastOpt, is an innovative space pruning strategy that avoids computing the complete tree of all possible relaxations in advance. When deciding to relax a condition in order to generate a relaxation to propose to the user, it explores only part of the tree, maintaining upper and lower bounds of the costs of the candidate relaxations, until the one with the lowest cost can be unambiguously determined. While the above techniques always produce the optimal condition relaxations sequence, we also investigate an approximate solution with improved scalability characteristics, called CDR (or Cost Distribution Relaxation). This is a probabilistic method that looks ahead into the space of potential relaxations for the few next steps, estimates the probability distribution of the cost of further relaxations necessary before the entire relaxation sequence is constructed, and makes a maximum likelihood decision for the best next relaxation. The experimental evaluation demonstrates that this method is both effective in finding a solution close to the optimal, and efficient in producing this solution fast.

Our main contributions can be summarized as follows:

• We propose a principled probabilistic optimization-based interactive framework for the empty-answer problem that accepts a wide range of optimization objectives, and is based on estimation of the user's prior knowledge and preferences over the data. To the best of our knowledge, such a probabilistic framework has not been studied before.

• We propose novel algorithmic solutions using our framework. The algorithms FastOpt and CDR produce optimal and approximate relaxation sequences respectively, without having to explore the entire relaxation tree. In particular, to the best of our knowledge, a probabilistic method such as CDR for tree exploration has not been considered before.

• We perform a thorough experimental performance and scalability evaluation using different optimization objectives on real datasets, as well as a usability study on Mechanical Turk.

2. MOTIVATING EXAMPLE

Consider a web site like *cars.com*, where users can search for cars by specifying in a web-form the desired characteristics. An example instance of such a database is shown in Figure 1. A user is interested in a car that has anti-lock braking system (ABS), dusk-sensing light (DSL), and manual transmission. The data instance of Figure 1 reveals that there is no car that satisfies these three requirements.

The user is in an urgent need of a cheap car, and is therefore willing to accept one that is missing some of the desired characteristics. The system knows that cheapest car is a Honda Civic (i.e., tuple t_3) that has ABS, but no manual transmission and no DSL. So it proposes to the user to consider cars with no DSL. If the user accepts, the system next propose to the user to consider cars with no manual transmission. If she also accepts the second relaxation, then the cheapest car of the database, tuple t_3 would be returned. If the user does not accept the second relaxation, the second cheapest car of the database (tuple t_1) would be returned.

Instead, the system could also propose to the user cars with no ABS in the beginning. However, if the user accepts that suggestion, this would result in a match of the most expensive car of the database (Askari A10, tuple t_2). Since the user wishes to find the cheapest car, therefore, proposing first to relax the DSL requirement is more preferred.

Assume that when the user is first asked to relax DSL, the answer is no. In this case, the system needs to investigate what alternative relaxations are acceptable. If the system knew that most users prefer cars with DSL, it could have used this knowledge to propose a different relaxation in the first place. In the following sections, we present a framework that takes into account all the above issues, for different optimization objectives.

The query lattice of the query $ABS{=}1 \wedge DSL{=}1 \wedge Manual{=}1$ is graphically depicted in Figure 2. The original query can be modeled as $(1, 1, 1)$, depicted at the root of the lattice, while each of

[1]Instead of a single relaxation, a ranked list of top-k relaxations could also be suggested in one step. We discuss these extensions in Section 6.

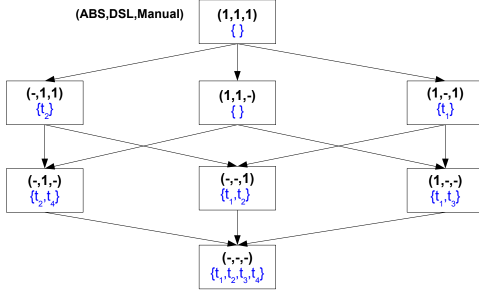

Figure 2: Query lattice of the query Q in Example 1.

the other nodes in the lattice represents a relaxed query. A relaxation on a specific attribute is depicted by a "$-$" (e.g., $ABS{=}1 \wedge DSL{=}1$ as $(1, 1, -)$). A directed edge from node p to node p' denotes that p' contains exactly one additional relaxation that is not present in p. For illustration, each relaxation contains the tuples in its answer set. Note that, given a query with k conditions, the number of possible relaxations is exponential in k.

3. PROBABILISTIC FRAMEWORK

In this section, we first discuss the generic probabilistic framework for optimization-based query relaxation, and then discuss specific application-dependent instantiations of the generic framework. This discussion considers Boolean databases for the ease of exposition, we refer to Section 6 for a further discussion on how to easily adapt categorical/numerical attributes with hierarchies inside our proposed framework.

3.1 Generic Probabilistic Framework

Let $\mathcal{A}$ be a collection $\{A_1, A_2, \ldots, A_m\}$ of m attributes, with each attribute $A_i{\in}\mathcal{A}$ associated with a finite domain Dom_{A_i}. The set of all possible tuples $\mathcal{U}{=}Dom_{A_1} \times Dom_{A_2} \times \ldots \times Dom_{A_m}$ constitutes the *universe*. A *database* is a finite set $\mathcal{D}{\subseteq}\mathcal{U}$. A tuple $t(v_1, v_2, \ldots, v_m)$ can also be expressed as a conjunction of conditions $A_i{=}v_i$, for $i{=}1..m$, allowing it to be used in conjunctive expressions without introducing new operators. Given $t{\in}\mathcal{U}$ we denote as $\mathfrak{Constrs}(t)$ the set of conditions of t.

A *query* Q is a conjunction of *atomic* conditions of the form $A_i{=}v_i$, where $A_i{\in}\mathcal{A}$ and $v_i{\in}Dom_{A_i}$. Each condition in the query is referred to as *constraint*. The set of constraints of a query Q is denoted as $\mathfrak{Constrs}(Q)$. A query can be equivalently represented as a tuple $(v_1, v_2, \ldots, v_m)$ where the value v_k corresponds to attribute A_k and models the condition $A_k{=}v_k$ if $v_k{\in}Dom_{A_k}$ or the boolean value "true" if v_k has the special value "-". Similarly, a tuple $(v_1, v_2, \ldots, v_m)$ can be represented as a query Q, i.e., a conjunction of conditions of the form $A_i{=}v_i$, one for each value v_i. Thus, by abuse of notation, we may write a tuple in the place of a query. A tuple t *satisfies* a query Q if $\mathfrak{Constrs}(Q) - \mathfrak{Constrs}(t){=}\emptyset$. The *universe* of a query Q, denoted as $\mathcal{U}_Q$, is the set of all the tuples in the universe $\mathcal{U}$ that satisfy the query Q. The answer set of a query Q on a database $\mathcal{D}$, denoted as $Q(\mathcal{D})$, is the set of tuples in $\mathcal{D}$ that satisfy Q. It is clear from the definition that $Q(\mathcal{D}){\subseteq}\mathcal{U}_Q$. An empty answer to a user query means that none of its satisfying tuples are present in the database.

Example 1. *The tuple* t_1 *in Figure 1 can be represented as* $Make{=}VW \wedge Model{=}Touareg \wedge Price{=}62K \wedge ABS{=}1 \wedge Computer{=}0 \wedge Alarm{=}0 \wedge 4WD{=}0 \wedge DSL{=}0 \wedge Manual{=}1 \wedge HiFi{=}0 \wedge ESP{=}1 \wedge Turbo{=}1$. *Given the set of attributes (ABS, DSL, Manual), the query* $ABS{=}1 \wedge DSL{=}1 \wedge Manual{=}1$ *can be modeled as* $(1, 1, 1)$ *while the query* $ABS{=}1 \wedge DSL{=}1$ *as* $(1, 1, -)$. ■

A relaxation is the omission of some of the conditions of the query. This results into a larger query universe, which means higher likelihood that the database will contain one or more of the tuples in it, i.e., the evaluation of the relaxed query will return a non-empty answer.

Definition 1. *A* relaxation *of a query* Q *is a query* Q' *for which* $\mathfrak{Constrs}(Q'){\subseteq}\mathfrak{Constrs}(Q)$. *The constraints in* $\mathfrak{Constrs}(Q) - \mathfrak{Constrs}(Q')$ *are referred to as* relaxed constraints *and their respective attributes as* relaxed attributes. ■

For the rest of this paper, since our goal is to provide a systematic way of finding a non-empty answer relaxation, we consider for simplicity only relaxations that involve one constraint at a time. Note that there are other forms of relaxations, relevant to categorical attributes with hierarchies, or numerical values. Our techniques can also handle these forms of relaxations. We discuss these cases further in Section 6.

The extra tuples that the query universe of a relaxation of a query Q has as opposed to query universe of Q is called a *tuple space*.

Definition 2. *The* tuple space *of a relaxation* Q' *of a query* Q, *denoted as* $\mathbb{TS}_Q(Q')$, *is the set* $\mathcal{U}_{Q'} {-} \mathcal{U}_Q$. ■

Among the constraints of a user query, some may be fundamental and the user may not be willing to relax them. We refer to such constraints as *hard constraints* and to all the others as *soft constraints*. Since the hard constraints cannot be relaxed, for the rest of this paper we focus our attention on the remaining constraints of the user query, which are initially considered to be soft.

In the tuple representation of a query, we use the "#" symbol to indicate a hard constraint, and "?" to indicate a question to the user for the relaxation of the respective constraint.

Example 2. *The expression* $(1, \#, -, 1, ?)$ *represents a relaxation query for which the user has already refused to relax the second constraint (i.e., she has kept the original query condition on the second attribute as is), has accepted to relax the third one, and is now being proposed to relax the last constraint.*

In order to quantify the likelihood that a possible relaxation Q' of a query Q is accepted by the user, we need to consider two factors: first, the *prior* belief of the user that an answer will be found in the database using the relaxed query Q', and second, the likelihood that the user will *prefer* (i.e., be satisfied with) the answer set of Q'. The relaxation Q' selected by the framework should have high values for both factors, and additionally should attempt to optimize application-specific objectives (e.g., try to steer the user towards highly profitable/expensive cars). We provide generic functional definitions of both factors next, and defer application-specific details to Section 3.2.

Since we cannot assume that the user knows the precise instance of the database, we resort to a probabilistic method for modeling that knowledge through a function called $prior(t, Q, Q')$. It specifically measures the *user belief* that a certain tuple t satisfying the relaxed query Q', i.e., a tuple from the tuple space of the relaxation, exists in the database. In order to estimate the likelihood that the user is satisfied with an answer set, we use a preference function $pref(t, query)$ that captures the probability that a user will like a tuple t, given the query. Section 3.2 discusses how specific prior and pref functions can be constructed for various applications.

Using the prior and the pref functions, we can compute the *relaxation preference function*, i.e., the probability that a user accepts a proposed relaxation Q' to a query Q (where Q evaluates to an empty answer). The probability to reject the relaxation is:

$$relPref_{no}(Q, Q') = \sum_{t \in \mathbb{TS}_Q(Q')} (1 - pref(t, Q')) * prior(t, Q, Q') \quad (1)$$

which represents the probability of not liking any of the tuples in the tuple space. Thus the probability of accepting the relaxation

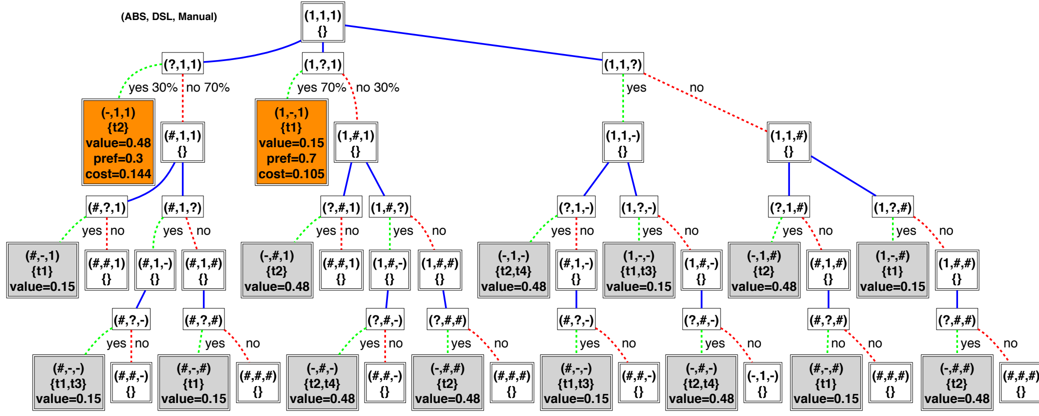

Figure 3: Query Relaxation tree of the query in Example 1.

is the probability that the user likes at least one tuple, which is the inverse of the probability of the user not liking any tuple (i.e., rejecting the relaxation), namely

$$relPref_{yes}(Q, Q') = 1 - relPref_{no}(Q, Q') \tag{2}$$

To encode the different relaxation suggestions and user choices that may occur for a given query Q that returns no results, we employ a special tree structure which we call the *query relaxation tree* (see Figure 3 for an example of such a tree). This is similar to tree structures used in machine learning techniques and games [23]. The tree contains two types of nodes: the *relaxation* nodes (marked with double-line rectangles in Figure 3) and the *choice* nodes (marked with single-line rectangles in Figure 3). Note that the children of relaxation nodes are choice nodes, and the children of choice nodes are relaxation nodes.

A *relaxation node* represents a relaxed query. The root node is a special case of a relaxation node that represents the original user query. A relaxation node does not have any children when the respective query returns a non-empty answer, or returns an empty-answer but cannot be relaxed further (either because all its constraints are hard, or because no further relaxation can lead to a non-empty answer). In every other case, relaxation nodes have k children, where k is the number of soft constraints in the query corresponding to the node. Each child represents an attempt to further relax the query. In particular, the i-th child represents the attempt to relax the i-th soft constraint (recall that in each interaction step we attempt to relax only a single constraint). These children are the choice nodes.

A *choice node* models an interaction with the user, during which the user is asked whether she agrees with the relaxation of the respective constraint. Each choice node has always two children: one that corresponds to a positive response from the user, and one that corresponds to a negative response. In the first case, the child is a relaxation node that inherits the constraints from its grandparent (i.e., the closest relaxation node ancestor), minus the constraint that was just relaxed (this constraint is removed). In the second case, the child is a relaxation node inheriting the constraints from the same grandparent, but now the constraint proposed to be relaxed has become a hard constraint (the relaxation was rejected). A choice node can never be a leaf. Thus, any root-to-leaf path in the tree starts with a relaxation node, ends with a relaxation node, and consists of an alternating sequence of relaxation and choice nodes.

Each path of the tree from the root to a leaf describes a possible relaxation sequence. Note that if the query Q consists of k constraints (i.e., attributes), there are an exponential (in k) number of possible relaxation sequences. In practice, the number of paths is significantly smaller, because they may terminate early: at relaxation nodes that have a non-empty answer, or at relaxation nodes for which no descendant corresponds to a non-empty answer.

Given an empty-answers query Q, let CONSTRUCTQRTREE(Q) be the procedure that generates the query relaxation tree for Q. This procedure will serve as a subroutine in our later algorithms for finding relaxation sequences with optimal cost.

Example 3. *Figure 3 illustrates the query relaxation tree for the query Q in the Example 1. Relaxation nodes are modeled with a double-line and choice nodes with a single-line border. The color-filled nodes are nodes corresponding to relaxations with a non-empty answer. The non-colored leaves correspond to relaxations that cannot lead to a non-empty answer, irrespective of further relaxations that may be applied. Notice how the "?" symbol is used to illustrate the proposal to relax the respective condition, and how this proposal is turned into a relaxed or a hard constraint, depending on the answer provided by the user.* ■

Next, we introduce and assign a *cost* value to every node of the query relaxation tree. Having the entire query relaxation tree that describes all the possible relaxation sequences, the idea is to consider the cost value of each relaxation node to determine which relaxation to propose during each interaction, based on the specific optimization objective, as we describe in Section 3.2.

Recall Equations (1) and (2) that describes the probability that a user will reject, or accept a specific relaxation proposal made by the system. Using these formulae, in general, the cost of a choice node n can be expressed as:

$$\begin{aligned} Cost(n) = relPref_{yes}(Q, Q') * (C_1 + Cost(n_{yes})) + \\ relPref_{no}(Q, Q') * (C_1 + Cost(n_{no})) \end{aligned} \tag{3}$$

where the n_{yes} and n_{no} are the two children (relaxation) nodes of n, Q is the query corresponding to the parent of n, and Q' corresponds to the suggested relaxation of Q at node n. In the formula, the variable C_1 is a constant, that is used to quantify any additional cost incurred for answering the current relaxation proposal.

The cost of a relaxation node, on the other hand, depends on the way the costs of its children are combined in order to decide the next relaxation proposal. To produce the optimal solution, at every step of the process, a decision needs to be made on what branch to follow from that point onward. This decision should be based on the selection of the relaxation that optimizes (maximizes or minimizes) the cost. Thus, the cost of a relaxation node n in the query relaxation tree is

$$Cost(n) = optimize_{c \in S} Cost(n_c) \tag{4}$$

where Q is the query that the node n represents, S is the set of soft constraints in $\mathfrak{Constrs}(Q)$, and n_c is the choice child node of n that

corresponds to an attempt to relax the soft constraint c. The optimization task is either maximization or minimization depending on the specific objective function.

The cost of a leaf node depends on a specific optimization objective, and the "value" of the tuples in that leaf node in contributing towards this objective. These details are presented in Section 3.2.

Therefore, the final *task* is to propose a sequence of relaxation suggestions interactively, such that the cost of the *root* node in the relaxation tree is optimized. An algorithm for that task using the relaxation tree is discussed in Section 4.1.

3.2 Application-Specific Instantiations of the Probabilistic Framework

In the previous subsection we developed a generic query relaxation framework. We emphasize that the main thrust of our paper's contribution is in developing the generic framework, which is largely agnostic to application-specific details. However, to illustrate its range of applicability, we take the opportunity here to discuss various specific instances of the framework, notably different instances of the *prior*, *pref*, and *objective functions*.

Recall that the *prior* function represents the user's prior knowledge of the content of the database. An implementation of the *prior* is to consider the data distribution in the case of known data domains. In our current implementation, we use the popular Iterative Proportional Fitting (IPF) [7] technique on the instance data (which can be thought as a sample of the subject domain) to estimate the required probabilities. IPF takes into account multi-way correlations among attributes, and can produce more accurate estimates than a model that assumes independence across attributes. However, we note that the independence model, or any other probability density estimation technique can be applied in the place of IPF.

The *pref* function is the probability/likelihood that a user will like a tuple t given a query. In simple instances, e.g., where the user in interested in cheap items in the query instances, the preference for a tuple can be modeled as any suitable function where the probability is dependent on the price of the item (higher the price, lower the probability). More generally, the approach is to use a tuple scoring function for calculating the *pref* of the tuples that imposes a non-uniform bias over the tuples in the tuple space. For example, instead of simple tuple scoring functions (such as price), one could also use more complex scoring functions such as assigning a *relevance* score [5] to each of the tuples. There exists a large volume of literature on such ranking/scoring functions [1, 10, 5]. Even though any of these functions are possible, in our implementation, we use a simple and intuitive measure, which is based on the Normalized Inverse Document Frequency [1].

$$pref(t, query) = \frac{\sum_{c \in \mathfrak{Constr}(query) \cap \mathfrak{Constr}(t)} idf(c)}{\sum_{c \in \mathfrak{Constr}(query)} idf(c)},$$

where $idf(c) = \log \frac{|\mathcal{D}|}{|\{t | t \in \mathcal{D}, t \text{ satisfies } c\}|}$.

However, the question remains - as the relaxation process progresses, does the preference of the user also evolve, i.e., the preference for a particular tuple changes? Note that the preference for a particular tuple may be computed in several different ways: (1) preference for a tuple is independent of the query and is always static - an example is where the preference is tied to a static property of the tuple, such as price, (2) preference for a tuple is query dependent, but only depends on the initial query and does not change during the interactive query relaxation session - e.g., when the preference is based on relevance score measured from the initial query, and (3) preference for a tuple is dependent on the latest relaxed query the user has accepted - this is a very dynamic scenario where after each step of the interactive session the preference can change. These different preference computation approaches are referred to as Static, Semi-Dynamic, and Dynamic respectively.

The generic probabilistic framework discussed in the previous subsection could be used to optimize a variety of *objective functions*, by appropriately modifying the preference computation approach of the tuples, and the cost computation of the leaf nodes, relaxation nodes, and the choice nodes of the relaxation tree. We illustrate this next.

Just as each tuple has a preference of being liked by a user, each tuple can also be associated with a *value* that represents its contribution towards a specific objective function. It is important to distinguish the value of a tuple from the preference for a tuple - e.g., if the objective is to steer the user towards overpriced, highly-profitable items, then the value of a tuple may be its price (higher the better), whereas the user may actually prefer lower priced items (lower is better) - although in most applications the value and the preference of a tuple are directly correlated. Thus in some applications, our query relaxation algorithms have to delicately balance the conflicting requirement of trying to suggest relaxations that will lead to high-valued tuples, but at the same time ensuring that the user is likely to prefer the proposed relaxation. The following example illustrates this situation:

Example 4. *Consider the example database in Figure-1, and assume that instead of steering the user towards cheap cars, the objective is to steer users towards expensive cars. In this case, the value/preference of a tuple is directly/inversely correlated with its price. For the purpose of illustration, let value of* $t_1 = 0.15, t_2 = 0.48, t_3 = 0.07, t_4 = 0.30$. *Let us also assume that the probability that the user will say "yes" to relaxing ABS is only* 0.3 *(e.g., she knows that most cars come with ABS systems, and relaxing ABS will not offer too many additional choice of cars), whereas the probability that she will say "yes" to relaxing DSL is much higher at* 0.7 *(e.g., it may be a relatively rare feature, and relaxing it may offer new choices). Of course, our system can only estimate these relaxation preference probabilities using Equations 1 and 2, which depend on the prior and tuple preference functions.*

Then, the cost of relaxing ABS is the expected value that can be achieved from it, which is $0.3 \times 0.48 = 0.144$, *while the cost of relaxing DSL is* $0.7 \times 0.15 = 0.105$. *The system would therefore prefer to suggest relaxing DSL to the user, since it has a higher cost (i.e., potential for greater benefit towards to overall objective), even though* t_2 *has lower preference than* t_1. ■

As with preferences, the value of a tuple may evolve as the user interacts with the system. Consequently three cases arise:

Static: In this case, the value of a tuple t is pre-calculated (statically) independently of the initial query Q, or subsequent relaxed queries Q'. The relaxation suggestions try to lead user to a leaf-node that has the highest cost (cost of a non-empty leaf is the maximum value of the tuples that represent that leaf[2])[3]. One can see that this is equivalent to guiding the users to the most-valued tuples. In such cases, the cost of a choice node is computed using Equation 3, by setting $C_1 = 0$. Finally, as the optimization objective is to maximize cost, then the cost of a relaxation node is the *maximum* cost of its children.

Semi-Dynamic: In this case, the value of a tuple t is calculated using the initial query Q, the first time it appears in the tuple space of a relaxation. Typical examples of such values are *relevance* score of the tuple to the initial query (here value is same as preference).

[2] Other aggregation functions (such as average) are also possible; the appropriate choice of the aggregation function is orthogonal to our problem.

[3] Cost of an empty-leaf node is 0.

This computed value of t is reused in all subsequent relaxations. The rest of the process is similar to that of Static.

Dynamic: In this case, the value of a tuple t at a relaxation node is calculated using the latest relaxed query Q' that the user has accepted. This value computation is fully dynamic, and the value of the same tuple t may change as the last accepted relaxed query changes. An example of such dynamically changing values are *relevance* of the tuple to the *most recent relaxed query*. Such dynamic value computation approach could be used inside the framework with the optimization objective of *minimizing user effort*, as it minimizes the expected number of interactions. In this case, any leaf node (empty or non-empty) has equal cost of 0. The cost of a choice node is computed using Equation 3, by setting $C_1 = 1$ (incurs additional cost of 1 with one more interaction). Finally, if the cost of a relaxation node is computed as the *minimum* cost of its children, then the underlying process will suggest relaxations that terminate this interactive process in minimum number of steps in an expected sense, thus minimizing the user effort.

More Complex Objective Functions: Interestingly, the framework could even be instantiated with more complex objective functions, such as those that represent a combination of the previous optimization objectives of relevance, price, user effort, etc. (e.g., most relevant results as quickly as possible, or cheapest result as quickly as possible). In such cases, the cost of a leaf node needs to be modeled as a function that combines these underlying optimization factors. After that, the cost computation of the relaxation nodes or the choice nodes in the relaxation tree would mimic either Semi-Dynamic[4] or Dynamic, depending upon the specific combined optimization objective. Further discussion on complex objective functions is omitted for brevity.

4. EXACT ALGORITHMS

4.1 The FullTree Algorithm

Given Equation 4, one can visit the whole query relaxation tree in a depth-first mode and compute the cost of the nodes in a bottom-up fashion. We refer to this algorithm as FullTree. Its steps are described in Algorithm 1. Note that the specific approach has the limitation that the whole tree needs to be constructed first, and then traversed, a task that is computationally expensive since the size of the tree can be exponential in k, where k is the number of the constraints in the query. Furthermore, for every positive response that the user provides to a relaxation request, the algorithm has to call the database to evaluate the relaxed query. Additionally, based on the specific score computation approach, for every response, it may have to make additional calls to recompute the *prior* and the *pref* value for the tuples in the relaxed query tuple space. This may lead to time complexity prohibitive for many practical scenarios.

4.2 The FastOpt Algorithm

To avoid computing the whole query relaxation tree, for each relaxation, we can compute an upper and a lower bound of the cost of its children. From the ranges of the costs that the computation provides, we can identify those branches that *cannot* lead to the branch with the optimal cost. When the specific optimization minimizes cost (i.e., effort), these are the branches starting with a node that has as a lower bound for its cost a value that is higher than the upper bound of the cost of another sibling node. Similarly, when the objective is to maximize the cost (i.e., lead user to most relevant answers/answers with highest static score), the branches starting with a node that has as a upper bound for its cost a value that is smaller

[4]choice node and relaxation node cost of Static is same as that of Semi-Dynamic.

Algorithm 1 FullTree

Input: Query Q
Output: Relaxation Cost of Q
1: $\mathcal{T} \leftarrow$ CONSTRUCTQRTREE(Q)
2: **return** COMPOPTCOST($\mathcal{T}$)

3: **procedure** COMPOPTCOST(Node n)
4: **if** n has no children **then**
5: **return** 0
6: **if** n is a ChoiceNode **then**
7: $Cost_{yes} \leftarrow$ COMPOPTCOST(n.yesChild)
8: $Cost_{no} \leftarrow$ COMPOPTCOST(n.noChild)
9: $Q_{yes} \leftarrow n.yesChild.Query$ ▷ query in the "yes" child
10: $P_{no} \leftarrow relPref_{no}(n.Query, Q_{yes})$ ▷ Equation (1)
11: $P_{yes} \leftarrow 1 - P_{no}$
12: **return** $P_{yes} * (1+Cost_{yes})+P_{no} * (1+Cost_{no})$ ▷ Equation (3)
13: **else if** n is a relaxation node **then**
14: **return** $\underset{c \in n.Children}{\text{optimum}}$ COMPOPTCOST(c)

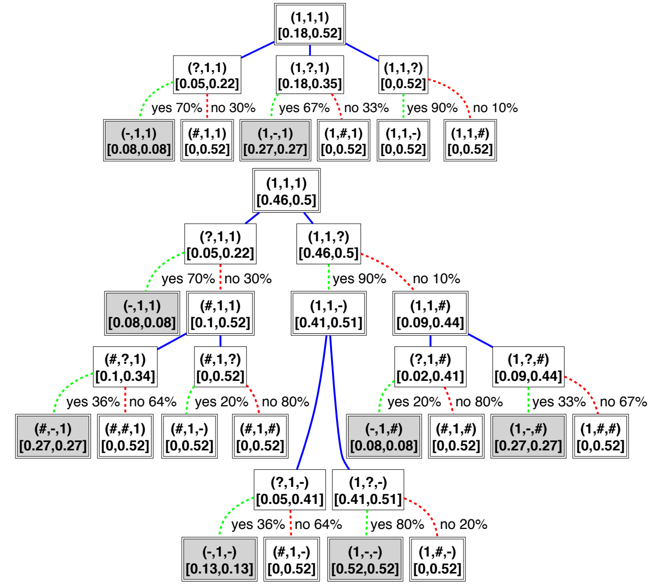

Figure 4: Ex. 5 Query Relaxation Tree after 1st and 2nd expansions

than the lower bound of the cost of another sibling node could be ignored. By ignoring these branches the required computations are significantly reduced. We refer to this algorithm as FastOpt.

Instead of creating the whole tree, FastOpt starts from the root and proceeds in steps. In each step, it generates the nodes of a specific level. A level is defined here as all the *choice* nodes found at a specific (same) depth, alongside the respective *relaxation* nodes they have as children. For the latter it computes a lower and upper bound of their cost and uses them to generate a lower and upper bound of the cost of the choice nodes in that level. When the cost is to be minimized (maximized), those choice nodes with a lower bound (upper bound) higher (lower) than the upper bound (lower bound) of a sibling node are eliminated along with all their subtree and not considered further. The computed upper and lower bounds of the choice nodes allow the computation of tighter upper and lower bounds for their parent relaxation nodes (compared to bounds that have already been computed for them in a previous step). The update of these bounds propagates recursively all the way to the root. If a relaxation node models a query that generates a non-empty answer, then it does not expand to its sub-children. Furthermore, after $|\mathfrak{Constrs}(Q)|$ repetitions, the maximum branch length is reached in which case the relaxation sequence with the optimum cost can be decided.

The upper and lower bounds of the cost of a node are computed

by considering the worst and best case scenario, and depends upon the specific score computation approach. Recall that the cost of a node is computed according to Equations (3) and (4). When the process seeks to optimize the cost using Semi-Dynamic or Dynamic score computation approach (corresponds to maximum relevance/maximize static score), the lowest cost of a node n at a level L could be as small as 0, because the remaining $|\mathfrak{Constrs}(Q)| - L$ relaxations accepted by the user may have a very small (almost zero) associated probability, resulting in the expected cost to be close to 0. This yields a lower bound $n.LB$=0. Alternately, the highest cost of a node n at a level L is achieved when the user is lead to the highest cost leaf with a "yes" probability of 100% immediately in the very next interaction. This yields an upper bound $n.UB$ = maximum cost of any leaf.

In contrast, when Dynamic score computation approach is used (corresponds to minimum effort objective), the lowest cost of a node n at a level L of the tree is achieved when the probability for the yes branch of the choice node is 100% and the $Cost(n_{yes})$ in Equation (3) is 0. This yields a lower bound $n.LB$=0. Similarly, the highest cost is achieved when all the remaining $|\mathfrak{Constrs}(Q)| - L$ negative responses have a probability of 100%. This yields an upper bound $n.UB$=$|\mathfrak{Constrs}(Q)| - L$.

At the end, when the computation reaches a level equal to the number of constraints in the query, $|\mathfrak{Constrs}(Q)|$, there is only one choice node to choose. Note that for the leaf nodes of the full tree, the upper and lower bounds coincide.

FastOpt is applicable to *any* cost function for which upper and lower bounds of the cost of a node can be computed even after *only part of the tree* below the node has been expanded. The efficiency of the algorithm relies on whether very tight bounds can be computed even after only a small part of the tree has been expanded. The cost function should also have the following monotonic property: the upper and lower bound calculations should get tighter if more of the tree is expanded. All aforementioned cost functions satisfy this property.

Example 5. *Consider the running example in Section 2, with the initial query (ABS, DSL, $Manual$), which aims to guide the user towards cheap cars. The value of a tuple is inversely proportional to its price. Let the normalized values for those tuples be* 0.27, 0.08, 0.52, *and* 0.13. *The objective is to select the relaxation node with the highest cost (i.e., expected value).*

Consider Figure 4. At the beginning the root node that is created represents the original query with 3 *conditions. Then, in the first iteration (L=1), the* 3 *possible choice nodes (corresponding to the* 3 *attributes of the query) along with their yes and no relaxation child nodes, will be generated (upper half of the figure). Since the relaxation nodes (-,1,1) and (1,-,1) give non-empty answers, they get lower bound (and upper bound) costs of .08 and 0.27 respectively (in the figure, the bounds of every node are denoted in square brackets "[...]"). The rest of the relaxation child nodes will be assigned a lower bound of 0 and an upper bound of* 0.52 *(price of the most expensive tuple in the database). Then the bounds of the choice nodes will be updated based on the expected value (considering respective preference probabilities), and the lower bound (resp. upper bound) of the root node will also be updated with the maximum of lower bound (resp. upper bound) cost of its child nodes. In the figure, the values of the quantities $relPref_{no}(Q, Q')$ and $relPref_{yes}(Q, Q')$ are illustrated next to the label of the no and yes edges, respectively.*

Let us now consider the expansion of the second level. For brevity, we only expand the first and the third child, as shown in the lower half of Figure 4. The newly generated relaxation nodes have new upper bounds, apart from those generating empty answers (or cannot be relaxed further) that have a 0 upper bound. This impacts the relaxation nodes of the previous (first) level, whose bounds are updated to [0.08,0.08], [0.1,0.52], [0.41,0.51] and [0.09,0.44]. The updates propagate all the way to the top. Notice that the first child of the root has now an upper bound (0.22) that is smaller than the lower bound of the third child (0.46), thus the first child is pruned and will not be considered further. ■

To further optimize the algorithm, we expand at each step only the node that has the tightest bounds, i.e., the smallest difference between its lower and upper bounds. The intuition is that the difference between these two values will become tighter (or even 0), and the algorithm will very soon decide whether to keep, or prune the node, with no effect on the optimal cost of the tree.

5. APPROXIMATE ALGORITHM

Although the FastOpt algorithm discussed in the previous section generates optimum-cost relaxations and builds the relaxation tree on demand, the effectiveness of this algorithm largely depends on the cost distribution properties between the participating nodes. In the worst case, FastOpt may still have to construct the entire tree first, even before suggesting any relaxation to the user. In fact, due to the exponential nature of the relaxation tree, even the FastOpt algorithm may be slow for an interactive procedure for queries with a relatively large number of constraints. Applications that demand fast response time (such as, online air-ticket or rental-car reservation systems) may not be able to tolerate such latency. On the other hand, these applications may be tolerant to slight imprecision. Thus, we propose a *novel approximate solution* that we refer to as the CDR (Cost Distribution Relaxation) algorithm. Like FastOpt, Algorithm CDR also constructs the query relaxation tree on demand, but the constructed part is *significantly smaller*. This is possible because it leverages the distributional properties of the nodes of the tree to probabilistically compute their cost. Of course for applications that are less tolerant to approximate answers, FastOpt may be more desirable, even though the response time may be higher.

Given a query Q, the algorithm CDR computes first the exact structure of the relaxation tree up to a certain level $L < |\mathfrak{Constrs}(Q)|$. Next, it approximates the cost of each L-th level choice nodes by considering the cost distributions of its children and proceeds with the bottom-up computation of the remaining nodes until the root. At the root node, the best relaxation child node is selected, and the remaining ones are pruned. Upon suggesting this new relaxation, the algorithm continues further based on the user's response.

There are three main challenges in the above procedure: (i) in the absence of the part of the tree below level L, how will the cost of level L nodes be approximated? (ii) How is the cost of the intermediate nodes approximated in the bottom-up propagation? and (iii) how is the best relaxation at the root selected? To address these challenges, we propose the use of the distributional properties of the cost of the nodes and the employment of probabilistic computations, as described next.

5.1 Computing Cost Probability Distributions

The algorithm computes the distribution of the cost of the nodes at level L (first the relaxation nodes, then the choice nodes), and higher by assuming that the underlying distributions are independent and by computing the *convolutions* [3] of the *probability density functions* (pdf for short).[5] We adopt convolution of distributions definitions from previous work [3] to compute the probability

[5]The independence assumption is heavily used in database literature, and as the experimental evaluation shows, it does not obstruct the effectiveness of our approach.

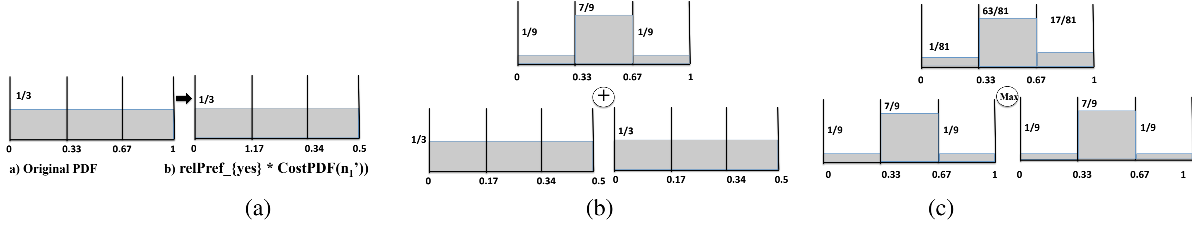

Figure 5: $CostPDF(n)$ for (a) the "yes" branch of a choice node, (b) choice node, and (c) non-leaf relaxation nodes.

distribution of the cost of the nodes in the partially built relaxation tree, as defined below. Then, in Section 5.2, we discuss how such convolution functions could be efficiently approximated using histograms.

Definition 3 (Sum-Convolution of Distributions)**.** *Assume that $f(x)$, $g(x)$ are the pdfs of two independent random variables X and Y respectively. The pdf of the random variable $X + Y$ (the sum of the two random variables) is the convolution of the two pdfs: $*(\{f, g\})(x) = \int_0^x f(z)g(x-z)\,dz$.*

Definition 4 (Max-Convolution of Distributions)**.** *Assume that $f(x)$, $g(x)$ are the pdfs of the two independent random variables X, Y respectively. The pdf of the random variable $Max(X, Y)$ (the maximum of the two random variables) is the max convolution of the two pdfs: ${}_{max}*(\{f, g\})(x) = f(x)\int_0^x g(z)\,dz + g(x)\int_0^x f(z)\,dz$.*

The Min-Convolution can be analogously defined, and moreover these definition can be easily extended to include more than two random variables.

We now describe how to estimate the cost distribution of each node using Sum convolution and Max(similarly Min) convolution. We denote as $CostPDF(n)$ the probability density function of the cost of a node n.

Cost distribution of a Relaxation Node: We first need to compute the cost distribution of nodes at level L and then propagate the computation to the parent nodes. We consider the pdf of each node at level L to be *uniformly distributed* between its upper and lower bounds of costs as described in FastOpt.

For relaxation nodes at higher levels, we need to compute the optimum cost over all the children nodes. Note that, optimization objectives associated with Semi-Dynamic and Static require Max-convolution as the score of the relaxation nodes are maximized in those cases. In contrast, Dynamic could be used to minimize effort - requiring Min-convolution to be applied to compute the minimum cost of the relaxation nodes.

Cost distribution of a Choice Node: The computation of the cost distribution involves the summation operation between two pdfs ($CostPDF(n_{yes})$ and $CostPDF(n_{no})$), and between a constant and a pdf (e.g., $C1 + CostPDF(n_{yes})$) (ref. equation (3)). Assuming independence, the former operation involves the sum convolution of two pdfs, whereas the latter requires the sum convolution between a pdf and a constant. In addition, $C1+CostPDF(n_{yes})$ (similarly $C1 + CostPDF(n_{no})$) needs to be multiplied with a constant $relPref_{yes}$ (similarly $relPref_{no}$). We note this multiplication operation between a constant and a pdf can be handled using convolution as well.

Selecting Relaxation at the Root: Given that the root node in the relaxation tree contains k children, the task is to select the best relaxation probabilistically. For each child node n_i of root with pdf $CostPDF(n_i)$, we are interested in computing the probability that the cost of n_i is the largest (resp. smallest) among all its k children when we want to maximize (resp. minimizes) the cost of the root. Formally, the suggested relaxation at the root (N_{rlx}) equals $N_{rlx} = (\arg\max)_{n_i}(\Pr(Cost(n_i) \geq \prod_{j=1,j\neq i}^{k} Cost(n_j)))$ (respectively $N_{rlx} = (\arg\min)_{n_i}(\Pr(Cost(n_i) \leq \prod_{j=1,j\neq i}^{k} Cost(n_j))))$

Given the user response, the above process is repeated for the subsequent nodes until the solution is found.

5.2 Efficient Computation of Convolutions

The practical realization of our methodologies is based on a widely adopted model for approximating arbitrary pdfs, namely *histograms* (we adopt equi-width histograms, however any other histogram technique is also applicable). In [3] it has been shown that we can efficiently compute the Sum, Max, and Min-convolutions using histograms to represent the relevant pdfs. In the following example, we illustrate how histograms may be used for representing cost pdfs at nodes of the relaxation tree.

Example 6. *Consider a query Q with $|\mathfrak{Constrs}(Q)|$=5 and empty answers, and assume that the approximation algorithm sets $L = 2$. Let us assume that cost is required to be maximized. Consider a choice node n at level 2, which has child relaxation nodes n'_1 (for a positive response to the relaxation proposal) and n'_2 (for a negative response); Wlog, let 1 be the upper bound of cost of n'_1[6]. Thus, the cost of each child is a pdf with uniform distribution between 0 and 1. $CostPDF(n'_1)$ is approximated using a 3-bucket equi-width histogram, and if we assume that $relPref_{yes}$ and $relPref_{no}$ of n are 0.5, the $CostPDF(n)$ can be computed using Equation 3 by approximating the pdf of the cost of each child with a 3-bucket histogram. Figures 5 (a) - (b) illustrate these steps.*

The algorithm continues its bottom-up computations, and considers relaxation nodes in the next higher level: at level 1, given a relaxation node (n') that has k children (each corresponds to a choice node in level 2), $CostPDF(n')$ is computed by using Equation 4 and applying max-convolution on its children (see Figure 5 (c)). Once the pdf of the cost of every relaxation node at level 1 has been determined, the algorithm next computes the pdf of cost of each level 1 choice nodes using sum-convolution, and so on.

6. EXTENSIONS

Managing databases with categorical / numerical attributes with hierarchies: Treatment of a categorical attribute is no different in our framework than that of its Boolean counterpart. A categorical attribute is treated Boolean by assuming its specified value (in the query) to be 1, and all other possible values as '*' (i.e., relaxed). Similarly, hierarchies in categorical attributes are readily accommodated by our approach: we just expand the query lattice (refer to Figure 2) using the hierarchies of each attribute, construct the query relaxation tree, and proceed with our algorithms as usual.

In the presence of *numerical attributes*, apart from choosing which attribute to relax, we also have to decide to what extent to relax the value of the chosen attribute. This can be formulated as a problem of relaxing hierarchical attributes, by defining buckets over data ranges of the numerical attributes, and hierarchies over those buckets. (This formulation makes sense in several practical situations,

[6] Recall that the lower bound is always 0.

where numerical ranges can be naturally bucketized: e.g., the age can be bucketized in "teens", "young", "middle-aged", "seniors", screen resolution in "VGA", "SVGA", "XGA", "QVGA", etc.).

Cardinality constraint on answer set: In several cases, the users are interested in non-empty answers that contain a certain minimum number of tuples (specified by a threshold). This is a generalization of the solutions we propose in this study: we build the query relaxation tree (with a simple adjustment to the relaxation preference function), only that this time we stop expanding a branch of the tree when we reach a node that corresponds to an answer set with the desired cardinality, and we then apply the relaxation algorithm as usual.

Suggesting top-k relaxations: In certain applications, it may be disconcerting for the user to get just one relaxation suggestion at a time. Our proposed framework could be easily extended to suggest a ranked list of k relaxations at a given interaction, by suggesting to the user the k best sibling relaxation nodes based on cost, at a given level in the relaxation tree.

7. EXPERIMENTAL EVALUATION

We present our experimental evaluations in this section, investigate the effectiveness and scalability of our proposed solutions, and compare our proposed framework with a number of related works and baseline methods. Our prototype is implemented in Java v1.6, on an i686 Intel Xeon X3220 2.40GH machine, running Linux v2.6.30. We report the mean values, as well as the 95% confidence intervals, wherever appropriate.

Datasets: We use two real datasets from diverse domains, namely, used cars and real estate. The Cars dataset is an online used-car automotive dealer database from US, containing 100,000 tuples and 31 attributes. The Homes dataset is a US home properties dataset extracted from a nationwide realtor website, comprising of 38,095 tuples with 18 boolean attributes. Based on the Cars dataset, we also generated datasets ranging between 20,000-500,000 rows (Cars-X), where we maintained the original (multidimensional) distribution of attribute values. This offers a realistic setting for testing the scalability of our algorithms.

Queries: We consider a workload of 20 random queries, initially containing only soft constraints. User preferences are simulated using our *relPref* value associated with each choice node.

7.1 Implemented Algorithms

Interactive Algorithm - Interactive: This algorithm is from our interactive query relaxation framework, and we implement three different instances of the preference computation: (i) Dynamic: a minimization of the user effort, with parameter $C_1 = 1$ and leaf cost 0; (ii) Semi-Dynamic: a maximization of the answers quality with parameter $C_1 = 0$ and leaf cost equal to the maximum value of the preference function in the result-set; and (iii) Static: a maximization of a randomly chosen static value for each tuple, with parameter $C_1 = 0$ and leaf cost the maximum profit of the result-set. Additionally, we implement FullTree, FastOpt, and our parameterized algorithm CDR.

Simple Baselines: We implement two simple baseline algorithms: Random always chooses the next relaxation to propose to the user at random, and Greedy greedily selects the first encountered non-empty relaxation, considering only the next level.

Related Works: We describe implemented related works next.

- top-k: This algorithm takes user-specific ranking functions as inputs (user provides weights for each attribute of the database), and we show the top-k tuples, ranked by the linear aggregation of the weighted attributes.
- Why-Not: This algorithm is from [24], non-trivially adapted for the empty-answer problem. We note that method [24] is primarily designed for numerical data, and inappropriate for empty-answer problem, since it assumes that the user knows her desirable answer (unlike empty-answer problems). We make the following adaptations: given a query with empty-answers, we apply our relevance-based pref ranking function (Section 3) to determine the most relevant tuple in the database that matches the query (non-exact match). We use that tuple as user's desirable answer, then convert the categorical database to a numeric one (in scale $0-1$), and apply [24] to answer the corresponding Why-Not query. The algorithm generates a set of relaxations that we present to the user.
- Multi-Relaxations: This algorithm is from [15], suggesting all minimal relaxations to the user.
- Mishra and Koudas: This algorithm is from [22]. Given a query, the method suggests a set of relaxations, such that the number of tuples in the answer set is bounded by a user specified input. The main differences and limitations of this work are discussed in Section 8. Our empirical study on the queries presented above exhibits that 81% of the queries with 8 constraints do not lead to a non-empty answer (i.e., failing queries), if the relaxations take place along each attribute independently (and ignore multiple attribute relaxations in conjunction). Figure 6 depicts the percentage of failing queries with queries of increasing size. Therefore, [22] largely fails to successfully address the empty-answer problem at hand.

Summary of Experiments: We implement the related works discussed above, and set up a user study comparing the related works with our proposed framework in Section 7.2. Additionally, we also present a separate study that validates the effectiveness of different cost-functions supported by our framework. An empirical comparison among the different objectives is presented in Section 7.3. Section 7.4 presents quality experiments to experimentally demonstrate the effectiveness of our proposed framework in optimizing the preferred cost function (by the cost of the root node of the relaxation tree). Section 7.5 presents the scalability studies. Section 7.6 reports the effectiveness of the approximation algorithm CDR based on the parameter (L).

Summary of Results: Our study concludes the following major points - (1) Existing methods are unable to address the same broad range of objectives (e.g., the case when the overall goal conflicts with user preference) as we do. (2) More than 60% of the users prefer "step-by-step" interactive relaxation suggestion to non-interactive top-k results based on user defined ranking functions (11%), or returning all relaxations suggestions in one step [24, 12] (20%). (3) User satisfaction is maximum (i.e., over 90%) with the returned results by our framework even for seller-centric optimization objectives. (4) Our proposed algorithms scale well with increasing dataset or query size (experiments up to 500k tuples). (5) Algorithm CDR can effectively balance between efficiency and the quality of the returned results (within a factor of 1.08 from the optimal).

7.2 User Study

We build a prototype of our system and use the Homes database to conduct two user studies with Amazon's Mechanical Turk (AMT).

User-Study 1: In this user study, we compare our proposed method Interactive (for seller-centric optimization) with top-k, Why-Not and Multi-Relaxations. We hire 100 qualified AMT workers to evaluate 5 different queries, and measure user satisfaction in a scale of 1 to 4 [7] independently and in comparison with other methods. We ask each worker, which method is most preferable (Favored),

[7] 1- very dissatisfied, 2 - dissatisfied, 3- satisfied, 4-very satisfied

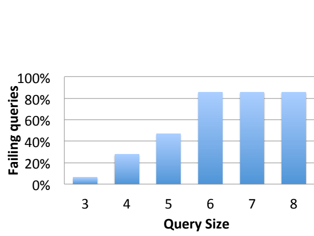

Figure 6: Failing queries in Mishra and Koudas vs query size.

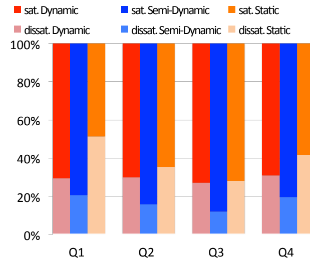

Figure 7: Percentage of satisfied and dissatisfied users.

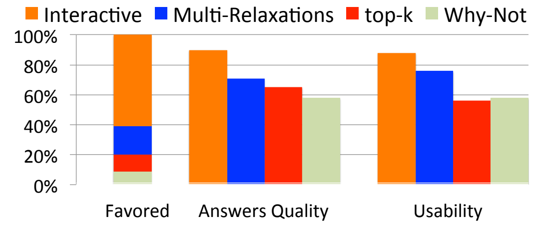

Figure 8: Comparison of user satisfaction with different related work.

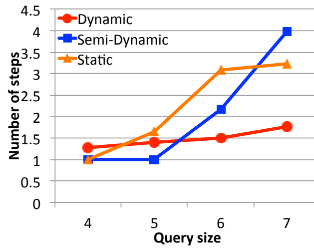

Figure 9: Number of steps vs query size in the user study.

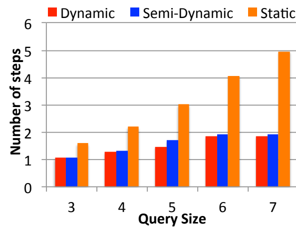

Figure 10: Effort vs Query Size of different preference objective functions (Homes dataset).

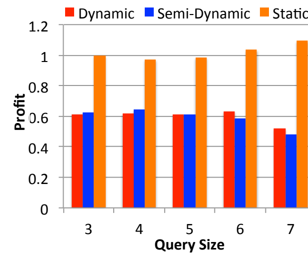

Figure 11: Profit vs query size of different preference objective functions (Homes dataset).

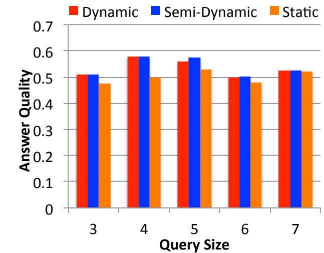

Figure 12: Quality of the results vs query size for different objective functions (Homes dataset).

rate her satisfaction with the quality of the returned results (Answers Quality) for each method, and rate her satisfaction with the effectiveness of each of methods (Usability). In addition we ask them the age range and the level of expertise with the use of computers and Internet (naive to IT professional user in the range 1-4). As depicted in Figure 8, more than 60% of the users prefer Interactive compared to other methods, and only 11% of users like to design ranking functions. With regard to result quality, more than 80% think that Interactive is appropriate for obtaining good quality results. At the other extreme, the adaptation of Why-Not algorithms produce good quality results only for 58% of the workers. With regard to method usefulness, the users are asked to independently evaluate the usefulness of each of the four methods in obtaining fast answers. 88% of the users prefer Interactive (i.e., give scores of 3 or 4), whereas 76% prefer Multi-Relaxations, 65% top-k, and 58% Why-Not. Finally, the users are also asked to score Interactive in terms of overall satisfaction: 91% workers are very satisfied with Interactive, out of which 49% are naive users (data is not shown in Figure 8).

User-Study 2: We set up three different tasks, hire a different set of 100 workers to test different optimization functions (without actually knowing them) in our framework. We propose five empty-answer queries per HIT, with $4-7$ attributes. The study uses the FastOpt algorithm, and the workers are asked to evaluate the suggested refinements (Q1), the system guidance (Q2), the time to arrive to the final result (Q3), and the system overall (Q4), in a scale of 1 (very dissatisfied) to 4 (very satisfied). We compare different optimization functions in terms of number of steps, profit, and answer quality (we only show results for the number of steps; the others exhibit a behavior similar to the ones described in the previous section, and are omitted for brevity). The analysis shown in Figure 9 shows that Dynamic guides users to non-empty results 2 times faster than the other approaches when the query size increases. The results (Figure 7) also show that the users express a favorable opinion towards our system. As expected, the Static method, being seller-centric, is the least preferred, yet satisfies 60% of the users on an average. The Semi-Dynamic approach is the most preferable overall, producing higher quality results faster, and highest user satisfaction (ranging between 72-89%) .

7.3 Preference Computation Comparison

Figure 10 shows how different cost functions behave with respect to the number of expected steps before we find a non-empty answer. We notice that the Static approach performs significantly worse than the other two. This is due to the fact that, in order to find more profitable tuples, the best option is to relax several constraints, which leads to producing long optimal paths. On the other hand, Figure 11 shows that Static achieves considerably better profit results, which means that the extra cost incurred pays off. Figure 12 measures the quality of the results, and indicates that the inclusion of the preference function in the probability computation tends to favor good quality answers. We also observe that the behavior of Static is very different from that of Dynamic and Semi-Dynamic, since it does not depend on the user preference, while the other two are highly user-centric, thus leading to (slightly) better answer quality.

7.4 Effectiveness

In the next set of experiments, we evaluate the effectiveness of the algorithms by measuring the cost of the relaxation for different query sizes. For brevity, we present only the results for Dynamic, since there are no significant differences among those objectives. In these experiments, we use queries of size up to 7, because this is maximum possible size for running FullTree in our experimental setup. Figure 13 depicts the results for the Homes dataset (normalized by the cost of FullTree for query size 3). The Cars dataset results are similar, and we omit them for brevity.

The graph confirms the intuition that the Random, Greedy and QueryRef algorithms are not able to find the optimal solution (i.e., the solution with the minimum expected number of relaxations). In addition, their relative performance gets worse as the size of the query increases, since the likelihood of making non-optimal choices increases as well. For query size 7, Random produces a solution that needs 2.4 times more relaxations than the optimal one.

On the other hand, CDR performs very close to FullTree, choosing the best path in most of the cases. The same observation holds for larger queries (upto 10 attributes, refer to Figure 14), where all values are normalized by the cost of FastOpt for query size 3). Our results also shows that CDR remains within a factor of 1.08 off the

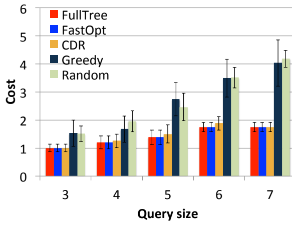

Figure 13: Relaxation cost vs query size (Homes dataset).

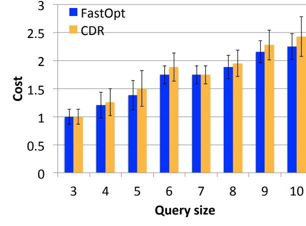

Figure 14: Relaxation cost vs query size, for FastOpt and CDR-3 (Homes dataset).

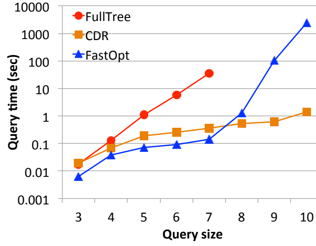

Figure 15: Query time (log scale) vs query size (Homes dataset).

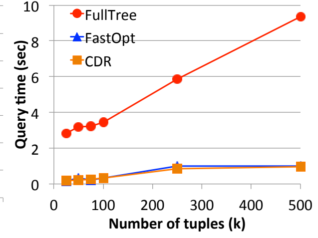

Figure 16: Query time vs dataset size, for query size 6 (Cars-X datasets).

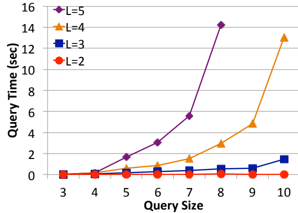

Figure 17: Query time vs query size, for CDR-L (Homes dataset).

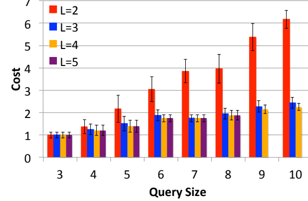

Figure 18: Relaxation cost vs query size, for CDR-L (Homes dataset).

optimal solution (expressed by FastOpt in the graph) corroborating its effectiveness to the empty-answer problem.

7.5 Scalability

Next we present experiments on the scalability properties of the top performers, FullTree, FastOpt and CDR, when both the size of the query and the size of the database increase.

In Figure 15, we illustrate the time to propose the next relaxation, as a function of the query size. We observe that the FastOpt algorithm performs better than CDR when the query size is small (i.e., less than 8).This behavior is explained by the fact that CDR is always computing all the nodes of the relaxation tree up to level L. In contrast, FastOpt is able to prune several of these nodes, leading to a significantly smaller tree. For query sizes larger than 8, CDR computes close to one order of magnitude less relaxation nodes than FastOpt. Finally, FullTree has an acceptable performance only for small query sizes (in our experimental setting we could only execute FullTree for query sizes up to 7).

The FastOpt algorithm remains competitive to CDR for small sized queries, but becomes extremely slow for large query sizes, requiring more than 1000 sec for queries of size 10. For the same queries, CDR executes three orders of magnitude faster, requiring 1.4 sec to produce the next relaxation, and significantly less than 1 sec for smaller queries.

We also experiment with varying dataset sizes between 25K-500K tuples, using the Cars-X datasets, having query size set to 6. Figure 16 indicates that the relaxation tree becomes smaller with increasing dataset size. This happens because more tuples in the dataset translate to an increased chance of a specific relaxation (i.e., node in the query relaxation tree) being non-empty. Note that, even though CDR involves more nodes than FastOpt, it has better time performance, since FastOpt has to build the entire tree before producing the first relaxation, whereas, CDR chooses the best candidate relaxation after computing the first L levels of the tree, which translates to reduced time requirements per iteration. Overall, we observe that FullTree and QueryRef quickly becomes inapplicable in practice, while FastOpt is useful only for small query sizes. In contrast, CDR is able to propose relaxations in less than 1 sec, even for queries with 10 attributes, and always produces a solution that is very close to optimal.

7.6 Calibrating CDR

Recall that the CDR-L algorithm starts by computing all the nodes of the relaxation tree for the first L levels (see Section 5), where L is a parameter.

Figure 18 shows the impact of L on the cost (the values have been normalized using as a base the cost of CDR-5 for query size 3). We notice that for $L = 2$ the algorithm behaves reasonably well only for very small query sizes. This behavior is expected, since for query sizes more than 4 the algorithm is trying to approximate the node cost distributions and then make decisions based on too little information. Increasing L always improves cost. $L = 3$ results in a considerable improvement in cost, but the results show that further increases have negligible additional returns. We also compare the time performance in Figure 17. The results show that CDR-5 quickly becomes expensive in terms of time, while CDR-4 is an efficient solution for queries with size up to 6. We conclude that using $L = 3$, CDR achieves the desirable trade-off between effectiveness and efficiency.

We also conduct experiments (omitted for brevity) with varying the number of the CDR histogram buckets between $5 - 40$, which has a negligible impact on time performance. For the rest of the experimental evaluation, we use CDR with $L = 3$, and 20 buckets.

8. RELATED WORK

Query Reformulation for Structured Databases: The closest related works are interactive optimization-based approaches studied for the *many-answers* problem, most notably [6, 18, 20], where given an initial query that returns *a large number of answers*, the objective is to design an effective drill-down strategy to help the user find acceptable results with minimum effort. We solve the complementary problem, where we have an *empty answer* and we need to relax the query condition to get a non-empty one. This fundamental asymmetry precludes a direct adaptation of one to another. In the former it is assumed that the user for sure prefers one of the tuples already in the result set, whereas, in our case the challenge is that we have no evidence what the user prefers, so we have to go with a probabilistic framework.

Most of the previous query relaxation solutions proposed for the empty-answer problem are non-interactive in nature. One of them [9] proposes query modification based on a notion of generalization, and identifies the conditions under which a generalization is applicable. In this framework there is no concept of leveraging user preferences in deciding the relaxation. Some other work [19] suggests alternative queries based on the "minimal" shift from the original query. In contrast, our method considers interactivity and additional goals that could be optimized during this interaction. "Why Not" queries are studied in [8, 24], where, given a query Q that did not return a set of tuples S that the user was expecting to be returned, to design an alternate query Q' that (a) is very "similar" to Q, and (b) returns the missing tuples S, however the rest of the returned tuples should not be too different from those returned by Q. "Why Not" queries are non-interactive, and it appears to be

non-trivial to extend these methods for the empty answers problem, because these problems require the user to be aware of some desired tuples S in the database, whereas in our case, no such set S is available to the user. Relaxation strategies for the empty-answer problem have been proposed as a recommendation service in [15, 16, 17]. All these methods are non-interactive and suggest relaxations with the objective to arrive at a non-empty answer that has the minimum number of attributes relaxed.

A few interactive query relaxation approaches have been proposed for the empty-answer problem. The paper [21] proposes an interactive relaxation strategy, and relaxes attributes one-by-one based on a deterministic partial/total order over attributes - no optimization criterion is explicitly described. A recent paper [22] proposes interactive query refinement to satisfy certain query cardinality constraints. The proposed techniques are designed to handle queries having range and equality predicates on numerical and categorical attributes. However, the technique is neither designed for optimizing any objective, nor does it consider any model of user preferences. We have provided an empirical study of this approach in Section 7.1.

An alternative to query reformulation approaches to solve empty/many-answers problems is the *ranked-retrieval* approach, where the task is to develop a *ranking function* that is able to score all items (even those that do not exactly match the query conditions) in the repository according to a "degree" of preference for the user, and return the top-k items. This approach can be very effective when the user is relatively sophisticated and knows what she wants, because the ranking function can be directly provided by the user. However, in the case of a *naive user* who is unable to provide a good ranking function, there have been many efforts to develop suitable system-generated ranking functions, both by IR [5] and database [1, 10, 11] researchers. At the same time, it has also been recognized [6, 14] that the rank-retrieval based approach has an inherent limitation for naive users - it is not an interactive process, and if the user does not like the returned results, it is not easy to determine how the system-generated ranking function should be changed. In this context, interactive approaches such as query reformulation are popular alternatives.

Query Reformulation in IR: Automatic query reformulation strategies for keyword queries over text data have been widely investigated in the IR literature [13, 12]. Various strategies have been used, ranging from relevance feedback to analyzing query logs and finding queries that are similar to the user queries. To find related queries, various strategies have been proposed, including measures of query similarity [4], query clustering [25], or exploiting summary information contained in the query-flow graph [2]. An alternative approach relies on suggesting keyword relaxations by relaxing the ones which are least specific based on their idf score [14].

9. CONCLUSIONS

In this work, we propose a novel and principled interactive approach for queries that return no answers by suggesting relaxations to achieve a variety of optimization goals. The proposed approach follows a broad, optimization-based probabilistic framework which takes into consideration user preferences. This is in contrast to prior query reformulation approaches that are largely non-interactive, and/or do not support such a broad range of optimization goals. We develop optimal and approximate solutions to the problem, demonstrating how our framework can be instantiated using different optimization goals. We have experimentally evaluated their efficiency and effectiveness, both by comparing to several techniques and by user studies.

As subsequent work, we intend to investigate extending our framework to handling join queries, as well as investigate a more holistic optimization-based query reformulation framework that considers simultaneous relaxations and tightening during interactive sessions.

10. ACKNOWLEDGMENTS

The work of Gautam Das was partially supported by NSF grants 0812601, 0915834, 1018865, and grants from Texas NHARP and Microsoft Research. Part of this work was done while the author was visiting the University of Trento and Qatar Computing Research Institute.

References

[1] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[2] A. Anagnostopoulos, L. Becchetti, C. Castillo, and A. Gionis. An optimization framework for query recommendation. In *WSDM*, pages 161–170, 2010.

[3] B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-*k* algorithms on exact and fuzzy data sets. *VLDB J.*, 18(2):407–427, 2009.

[4] R. A. Baeza-Yates, C. A. Hurtado, and M. Mendoza. Query recommendation using query logs in search engines. In *EDBT Workshops*, pages 588–596, 2004.

[5] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley New York, 2011.

[6] S. Basu Roy, H. Wang, G. Das, U. Nambiar, and M. Mohania. Minimum-effort driven dynamic faceted search in structured databases. In *CIKM*, pages 13–22, 2008.

[7] Y. M. M. Bishop, S. E. Fienberg, and P. W. Holland. *Discr. Multivariate Analysis: Theory and Practice*. MIT Press, 1975.

[8] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[9] S. Chaudhuri. Generalization and a framework for query modification. In *ICDE*, pages 138–145, 1990.

[10] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *VLDB*, pages 888–899, 2004.

[11] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31(3):1134–1168, 2006.

[12] S. Gauch and J. Smith. Search improvement via automatic query reformulation. *TOIS*, 9(3):249–280, 1991.

[13] S. Gauch and J. B. Smith. An expert system for automatic query reformulation. *JASIS*, 44(3):124–136, 1993.

[14] V. Hristidis, Y. Hu, and P. G. Ipeirotis. Ranked queries over sources with boolean query interfaces without ranking support. In *ICDE*, pages 872–875, 2010.

[15] D. Jannach. Techniques for fast query relaxation in content-based recommender systems. *KI'06: Advances in AI*, pages 49–63, 2007.

[16] D. Jannach and J. Liegl. Conflict-directed relaxation of constraints in content-based recommender systems. *Advances in Applied AI*, pages 819–829, 2006.

[17] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, volume 4, pages 167–172, 2004.

[18] A. Kashyap, V. Hristidis, and M. Petropoulos. Facetor: cost-driven exploration of faceted query results. In *CIKM*, pages 719–728, 2010.

[19] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[20] C. Li, N. Yan, S. B. Roy, L. Lisham, and G. Das. Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia. In *WWW*, pages 651–660, 2010.

[21] D. McSherry. Incremental relaxation of unsuccessful queries. In *EC-CBR*, pages 331–345, 2004.

[22] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873. ACM, 2009.

[23] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[24] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.

[25] J.-R. Wen, J.-Y. Nie, and H. Zhang. Query clustering using user logs. *ACM Trans. Inf. Syst.*, 20(1):59–81, 2002.