Noname manuscript No.
(will be inserted by the editor)

A Holistic and Principled Approach for the Empty-Answer Problem

Davide Mottin · Alice Marascu · Senjuti Basu Roy · Gautam Das[1] · Themis Palpanas · Yannis Velegrakis[2]

Received: date / Accepted: date

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 fewer 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. Moreover, we discuss two important extensions to the base framework: the specification of a minimum size on the number of results returned by a relaxed query and the possibility of proposing multiple conditions at the same time. We analyze the proposed solutions, experimentally verify their efficiency and effectiveness, and illustrate their advantages over the existing approaches.

The work was primarily done while D. Mottin and A. Marascu were at the U. of Trento.

[1]Partially supported by Texas NHARP, Microsoft Research, and NSF grants 0812601, 0915834, 1018865. Work done while at the U. of Trento and the Qatar Computing Research Inst.

[2]Partially supported by the ERC grant Lucretius and the KEYSTONE Cost Action.

D. Mottin, E-mail: davide.mottin@hpi.de
Hasso Plattner Institute

Y. Velegrakis, E-mail: velgias@disi.unitn.eu
University of Trento

A. Marascu, E-mail: alice.marascu@ie.ibm.com
IBM Research-Ireland

S. Basu Roy, E-mail: senjutib@njit.edu
Department of Computer Science, New Jersey Institute of Technology

G. Das, E-mail: gdas@uta.edu - gdas@qf.org.qa
University of Texas Arlington & QCRI

T. Palpanas, E-mail: themis@mi.parisdescartes.fr
University of Paris Descartes

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 way 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. Typically, query relaxation is *non-interactive* (e.g., [13,27,28,29]). 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. 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 for many application scenarios is more palatable the *interactive (or navigational)* approach 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 a further change in the query is not possible, given the restrictions set forward by the user. 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, a holiday house reservation, 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 provide the user more control with the selection process.

Our approach for selecting 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, our framework can be used to suggest alternative queries that do return some airline tickets, and these tickets that are the most "relevant" to her initial preferences, or are the 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 of the 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 discussion of related work in Section 8. Optimization-based interactive approaches have been proposed for the many-answers problem [7, 30, 32], however these papers only consider one narrow optimization goal, that of minimizing user effort. 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.

The approach works as follows. 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 3.3), 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 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).

[1] Instead of a single relaxation, a ranked list of top-k relaxations could also be suggested in one step.

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. In this paper we extend our previous work [41] adding the support top-k relaxations, limits on the cardinality of the results, a complexity analysis, support for non boolean databases, and a more efficient algorithm that combines the approximate and the pruning technique. The framework has also been demonstrated in a research prototype [40].

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.

• 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.

• We propose an extension of the framework that returns top-k relaxations at each step, and also we allow the possibility to specify a cardinality constraint on the size of the results. Enabling top-k relaxations is a critical step that affects time. As such, we introduce a new algorithm, FastCDR, that embeds both the optimal FastOpt pruning and the approximate cost computed by CDR. Our framework also accommodates previous Pareto-optimal solutions that find the maximally succeeding subqueries. To this end, we devise a combined method called FastOptMFS that includes maximally succeeding queries to our optimization criteria.

• We explain how our technique can be used for different cases such as categorical attributes with hierarchies, numerical attributes, or constraints on the expected answer set.

• We perform a thorough experimental performance and scalability evaluation using different optimization objectives on real datasets, as well as a usability study on real users, and we report our findings.

The rest of this paper is organized as follows. We start by presenting a motivating example in Section 2, and our probabilistic framework in Section 3. Section 4 describes exact algorithms that solve our problem, including efficient algorithms that can prune the search space, and Section 5, fast approximate algorithms. We discuss extensions of our basic framework in Section 6, and present the experimental evaluation in Section 7. Finally, we provide an overview of the related work in Section 8, and conclude in Section 9.

	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

Fig. 1: An instance of a car database.

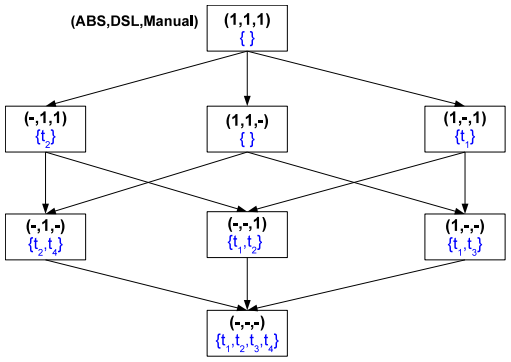


Fig. 2: Query lattice of the query Q in Example 1.

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 the 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 Manual transmission. If the user accepts, the system next proposes to the user to consider cars with no DSL. If she also accepts the second relaxation, then the cheapest car of the database, tuple t_3 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 preferable.

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 set of possible relaxations of the query $ABS{=}1 \wedge DSL{=}1 \wedge Manual{=}1$ is graphically depicted in Figure 2 as a lattice where each node represents a query. The query of a node is a relaxation of the query modeled on the node above. The query is expressed as a triple where each value of the triple means that the respective condition is ignored if "-" or considered if "1"(e.g., $ABS{=}1 \wedge DSL{=}1$ as $(1, 1, -)$). The original query can be modeled as $(1, 1, 1)$, depicted at the root of the lattice, while each of the other nodes in the lattice represents a relaxed query. 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

This section introduces the proposed probabilistic framework for interactive query relaxation. The framework is based on a cost associated to each user interaction with the system. Given the generality of the cost model, application-specific instantiations are presented (such as, proposing relaxations for which the results have the maximum price).

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 attribute A_k takes the value v_k if $v_k{\in}Dom_{A_k}$, or A_k takes any value 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 $\mathfrak{Constrs}(Q) \setminus \mathfrak{Constrs}(t){=}\emptyset$. 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 $\mathfrak{Constrs}(Q) \setminus \mathfrak{Constrs}(Q')$ 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 $\mathcal{U}_{Q'} \setminus \mathcal{U}_Q$.

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 the respective constraint to be relaxed.

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.

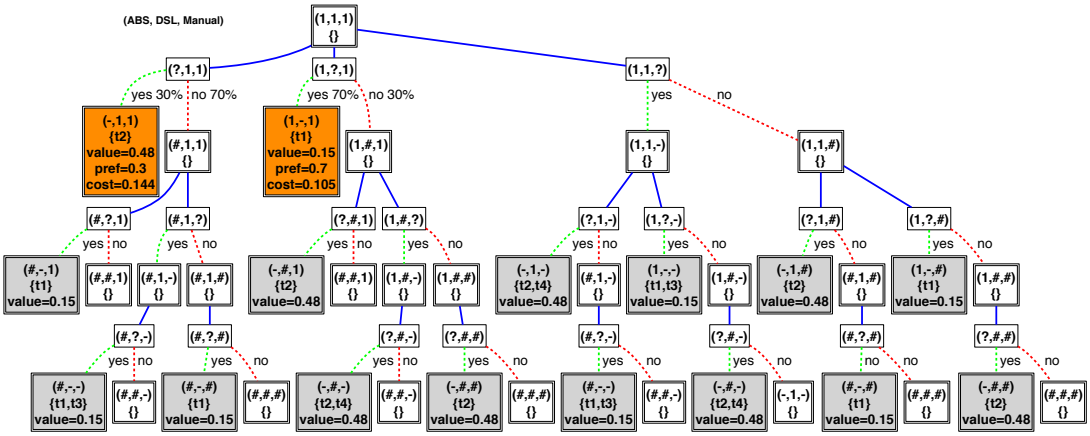


Fig. 3: Query Relaxation tree of the query in Example 1.

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 3.3.

Since we cannot assume that the user knows any tuple in 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 $pref(t, Q)$ that captures the probability that a user will like a tuple t, given the query. Section 3.3 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 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

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

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 [36]. 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 a number of children equal to 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.

Definition 3 (Relaxation tree) Given a query Q_s that returns no results, the *relaxation tree* $\mathcal{T}$ is recursively defined as follows. (1) The root node is a relaxation node representing the query Q_s; (2) the children of a relaxation node are all the choice nodes representing a relaxation Q' along one non-hard constraint; (3) the children of a choice node are two relaxation trees rooted at Q_{yes} and Q_{no} that represent the query Q' after the user choice (*yes*, or *no*, respectively); (4) a subtree has no children (leaf node) if the query is not relaxable or non-empty.

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 further relaxation of any non-hard constraint leads to a non-empty answer.

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. 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 3.3.

Recall Equations (1) and (2) that describe 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 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 objective.

The cost of a leaf node, returned by a function $CostLeaf$, depends on a specific optimization objective, and the "value" of the tuples in that leaf node that contribute towards this objective (the details are presented in Section 3.3).

Definition 4 Given a node n the $Cost$ is

$$Cost(n) = \begin{cases} CostLeaf(n) & \text{if } n \text{ is a leaf} \\ \text{Equation 3} & \text{if } n \text{ is a choice node} \\ \text{Equation 4} & \text{if } n \text{ is a relaxation node} \end{cases}$$

Recall that a relaxation tree encodes all possible relaxation sequence, among which we want to find the best one. A *solution tree* is a subtree of the complete tree containing only complete paths from the root to the leaves. Therefore, we aim at solving the following problem.

Problem 1 Given an initial empty answer query Q_s on a database D and a specific instance of $prior, pref, CostLeaf$ functions for the relaxation tree $\mathcal{T}$ built on Q_s, find the solution subtree $\mathcal{T}' \subseteq \mathcal{T}$ such that the $Cost(\mathcal{T}')$ is optimum (maximum or minimum depending on the optimization criteria), where $Cost(\mathcal{T}')$ is the cost of the root computed with Equation 4.

Example 4 Assume the objective is to minimize the number of steps, then the cost of the root in Figure 3 is the children with the shortest expected length. Even though the left and the middle children have the same structure, the middle has 70% probability of choosing the yes-child. Since the yes-path is the shortest, the child with the minimum cost is the middle one, and the query (1,?,1) is returned to the user.

3.2 Theoretical analysis

Consider the relaxation problem for an empty answer query in which the aim is to minimize the user effort, i.e., the number of user interactions needed. If one is able to find that

minimum cost relaxation, the simpler problem of deciding whether a relaxation has a cost at most n, should have the same or smaller complexity. Unfortunately, the latter problem can be shown to be **NP**-complete.

Theorem 1 *Given a database D and an empty answer query Q, deciding whether there is a query relaxation tree such that the cost of its root node is less than or equal to a constant n is* **NP***-complete.*

Proof. For the proof we can assume that we have a boolean database, i.e., a database where each attribute takes a boolean value of 0 or 1. We will show that even for that special case, the problem is still **NP**-complete. To do so we reduce the known to be **NP**-complete exact cover by 3-set problem (denoted as X3C) [20] to ours. Given a finite set U over $3n$ elements, and a collection S of 3-elements subsets of U, X3C finds an exact cover for U, i.e., a sub-collection $C \subseteq S$ of subsets, such that every element of U occurs in exactly one member of C.

Consider an instance X3C(U, S) of the X3C problem that consists of a finite set $U = \{x_1, x_2, \ldots, x_{3n}\}$ defined over $3n$ elements, and a family $S = \{S_1, S_2, \ldots, S_q\}$ of subsets of U, such that, $|S_i| = 3, \forall 1 \leq i \leq q$ and requires a yes or no answer on whether there exists a cover $C \subseteq S$ of n pairwise disjoint sets, covering all elements in U.

Given a specific instance $\mathcal{I}$ of the X3C problem, we create a database D with q boolean attributes $A = \{A_1, \ldots, A_q\}$ and $3n$ tuples $T = \{t_1, \ldots, t_{3n}\}$. For each $S_i = \{x_j, x_k, x_l\}$, A_i contains boolean 1 for tuples $\{t_j, t_k, t_l\}$, while the remaining tuples get boolean value 0 for attribute A_i. This way, every attribute is present (i.e., corresponds to 1 value for that attribute) in only three tuples. For such a database, we consider the query $Q \in \{0\}^q$ (i.e., all q constraints of Q are set to 0). We also construct a simple "black-box" ranking function that assigns a preference score $pref(t, Q)$ to each tuple t in the database. $pref(t, Q) = 1$, when the tuple exactly matches all the query predicates; otherwise, $pref(t, Q) = 0$. Therefore, for our instance, it is easy to see that $pref(t, Q) = 0$ for all the tuples in the database, since Q returns no answer in the first place. The next step is to generate a relaxation tree, and compute the cost of its root node based on the minimum cost strategy. Interestingly, using the black-box preference function described above, we have $relPref_{no} = 100\%$ and $relPref_{yes} = 0$ in each choice node. This is indeed true, because, $pref(t, Q) = 0$ for every tuple t, which results in $relPref_{yes} = 0$ for every choice node. The above steps achieved to create an instance $\mathcal{J}$ of the query relaxation problem from an instance I of the X3C problem that we initially considered. We consider the Dynamic objective, with $C1 = 1$. Moreover, since $pref(t, Q) = 0$ for each tuple t any choice of the prior leads to the same result.

We claim that $\mathcal{I}$ is a YES-instance of X3C iff $\mathcal{J}$ is a YES-instance of our problem.

($\Rightarrow$): Suppose $C = \{S_{j_1}, ..., S_{j_n}\} \subseteq S$ is an exact (disjoint) cover of U. Then consider each node in the optimal path of the query relaxation tree, where tuple $t_i \in A_k$ iff $x_i \in S_{j_k}$. Notice that each node A_k consists of only three tuples. Since C is an exact cover of U, each element $x \in U$ appears in exactly one subset $S_k \in C$. Thus, $Cost(root) = 100\% * (1 + Cost(n_{no})) = n$, that indicates that $\mathcal{J}$ is a YES-instance.

($\Leftarrow$): Let π be a the optimal path in the query relaxation tree. $Cost(root) = n$, witnesses the fact that $\mathcal{J}$ is a YES-instance. Observe that any node in π only contributes to 3 tuples to the database. Since the overall cost is n, it follows that every node contains exactly 3 tuples and they are disjoint. Now, if we consider the collection S, it is easy to verify that $|C| = n$ and that every element $x \in U$ appears in exactly one set $S \in C$. Therefore, $\mathcal{I}$ is a YES-instance.

□

3.3 Application-Specific Instantiations of the Probabilistic Framework

The generic query relaxation framework presented in the previous section 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. One possible implementation, which is the one we use, is the popular Iterative Proportional Fitting (IPF) [8, 44, 45] 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. This is achieved by considering the information on the correlations among some attributes in the various multi-dimensional marginal distributions of these attributes. This technique is based on the well recognized and widely applicable information theoretic principle of maximum entropy [8]. 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 [6] to each of the tuples. There exists a large volume of literature on such ranking/scoring functions [1, 6, 14]. 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].

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

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

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 5 Consider the example database in Figure 1, and assume that instead of steering the user towards cheap cars, the objective may steer the 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. Three cases can also be considered here.

Static: In this case, the value of a tuple t is pre-calculated (statically) independently of the initial query Q_s, 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_s, 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). This computed value of t is reused in all subsequent relaxations. The rest of the process is similar to that of Static.

[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.

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 proposed 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.

Combining Maximal Succeeding Subqueries. The framework can also combine previously studied strategies, e.g., those based on maximal succeeding subqueries, in order to propose interactive results that maximize one of the objectives and the minimality of the answer. The combination of such objectives considers only paths in the tree that lead to a maximal succeeding subquery and discards all the others. We integrate one adaptation of the well-known QuickXPlain [29] from Jannach [27] that includes user preferences to find the minimally failing subquery. The minimally failing subquery is the biggest empty-answer query containing the least preferred database attributes, such that any subquery is non failing. Therefore the constraints in the minimally failing subquery are the only constraints to be relaxed. We combine this strategy with the Semi-Dynamic and Dynamic objective, and use the algorithm in [27] as a subroutine to generate the relaxations.

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

3.4 Cardinality constraint

In several applications, the user is interested in non-empty answers that contain a certain minimum number of tuples (specified by some cardinality constraint). Our framework assumes that the user is interested in at least one result. A cardinality constraint introduces a cutoff criteria on the size of the results, and it might be integrated in the framework.

The simplest approach to constrain the number of tuples is to consider as empty any query that returns a number of tuples less than the required cardinality. This approach is used in most of the previous works [12, 35], but while the interactive method in [35] assumes that the user is interested in a fixed number of results, the why-not approach [12, 51] assumes that the user is interested in some specific tuple.

Alternatively, a positive score is assigned to any non-empty query. This approach ensures that any non-empty query can be selected, even with small probability. Interestingly, our framework can naturally model such cardinality constraints by assigning either a fixed minimum score to non-empty leaves, or a penalty on the $pref$ function on each result of a query with less than the required cardinality. Therefore, we assume the existence of a user defined scoring function that assigns a value to any non-empty relaxed query Q'. Notice that this value is then propagated up, through the recursive application of the $Cost$ function.

Since any scoring function is a valid choice, we have implemented the simplest approach.

3.5 Top-k relaxations

In certain applications, it may be disappointing for the user to get just one relaxation suggestion at a time. Our proposed framework can be readily 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 the cost at a given level in the relaxation tree. In a top-k model, the user chooses among the k relaxations without being asked to refuse/accept them individually. This has the substantial drawback that no feedback is provided in the navigation phase and the user is left blind on the interaction.

There are two ways of interpreting the choice of the user in a top-k model: the No-bias model, and the ranked or Skip-above model.

No-bias. Under the no bias model the user selects one relaxation and the system simply proposes the next relaxations, or the results of the relaxed query (if it is non-empty). This model has been extensively adopted to present refinements of queries in the many-answers problem [7,30], and in query recommendation [3, 11] in the web, since the order in which the reformulations are presented is irrelevant.

Skip-above. The alternative model assumes that the relaxations are proposed in the order of decreasing importance

(like we do in our framework), which is the approach employed in the click-chain model [48]. Then, once the user selects a relaxation, this implicitly means that she refuses to relax those that rank higher. Therefore, all the attributes that rank higher are implicitly deemed as hard and not proposed to the user anymore.

Note that one can easily use the No-bias model to materialize the Skip-above model since the former considers all the possible remaining attributes after the user selected one to relax. The opposite does not hold. However, generating more than one relaxations at a time results in higher computation times, because we need to explore more relaxation alternatives simultaneously. Section 4.3 shows how to efficiently modify algorithms that propose only the single best relaxation to obtain the desired results, without substantially changing the framework.

Another complementary approach to present top-k relaxations requires to recompute the $pref$ function on the selected/non-selected attributes. Therefore, the relaxation tree changes on demand to consider the change in the preference. This adaptive setting is not considered in this work.

4 Exact Algorithms

4.1 FullTree

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. This algorithm is referred to 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 by the procedure CONSTRUCTQRTREE, and then traversed, a task that is computationally expensive since the size of the tree can be exponential in the number of the constraints in the query. Procedure CONSTRUCTQRTREE constructs the whole tree starting on the root node representing input query Q_s, and then it recursively adds child nodes until the query in the node is non-empty or cannot be relaxed further (i.e., no more constraints to relax). 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.

Algorithm 1 FullTree

Input: Initial query Q_s
Output: Relaxation Cost of Q_s

1: $\mathcal{T} \leftarrow$ CONSTRUCTQRTREE(Q_s)
2: **return** COMPOPTCOST($\mathcal{T}$)

3: **procedure** CONSTRUCTQRTREE(Query Q)
4: $n_{relax} \leftarrow$ new RelaxationNode(Q) ▷ construct the root
5: $\mathcal{C}=\{c \mid c \in \mathfrak{Constrs}(Q) \wedge c$ is hard$\}$
6: $Q_{tmp} \leftarrow$ new Query($\mathcal{C}$)
7: **if** $Q_{tmp}(\mathcal{D})=\emptyset \vee Q = \emptyset$ **then** ▷ non-relaxable query
8: **return** n_{relax}
9: **for** $c \in \mathfrak{Constrs}(Q)$ **do**
10: **if** c is not hard **then**
11: $n_{resp} \leftarrow$ new ChoiceNode()
12: n_{relax}.addChild(n_{resp})
13: $Q_{yes} \leftarrow$ new Query($\mathfrak{Constrs}(Q) \backslash \{c\}$) ▷ remove c
14: n_{resp}.yesChild $\leftarrow$ (CONSTRUCTQRTREE(Q_{yes}))
15: $c_h \leftarrow Hard(c)$ ▷ c_h is the hard version of c
16: $Q_{no} \leftarrow$ new Query($(\mathfrak{Constrs}(Q) \backslash \{c\}) \cup \{c_h\}$)
17: n_{resp}.noChild $\leftarrow$ (CONSTRUCTQRTREE(Q_{no}))
18: **return** n_{relax}

19: **procedure** COMPOPTCOST(Node n)
20: **if** n has no children **then**
21: **return** $CostLeaf(n)$
22: **if** n is a ChoiceNode **then**
23: $Cost_{yes} \leftarrow$ COMPOPTCOST(n.yesChild)
24: $Cost_{no} \leftarrow$ COMPOPTCOST(n.noChild)
25: $Q_{yes} \leftarrow n.yesChild.Query$ ▷ query in the "yes" child
26: $P_{no} \leftarrow relPref_{no}(n.Query, Q_{yes})$ ▷ Equation (1)
27: $P_{yes} \leftarrow 1 - P_{no}$
28: **return** $P_{yes} * (C1+Cost_{yes}) + P_{no} * (C1+Cost_{no})$
29: **else if** n is a relaxation node **then**
30: **return** $\underset{c \in n.Children}{\text{optimum}}$ COMPOPTCOST(c)

4.2 FastOpt

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 the 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 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. Algorithm 2 shows the FastOpt pseudocode.

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

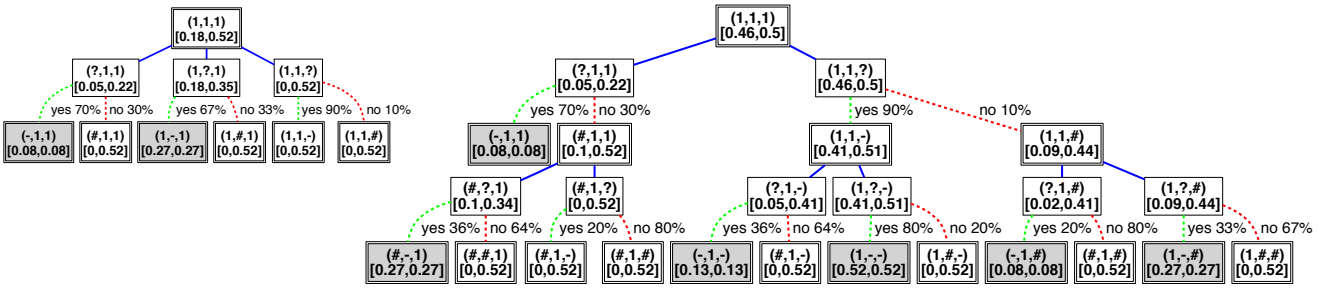


Fig. 4: Ex. 6 Query Relaxation Tree after 1st and 2nd expansions

Algorithm 2 FastOpt

Input: Initial query Q_s
Output: Relaxation Cost of Q_s

1: NextBranch ← new RelaxationNode(Q_s)
2: NextBranch.L ← 1
3: $\mathcal{T}$ ← NextBranch
4: **repeat**
5: Build tree at level NextBranch.L
6: N_{rlx} ← relaxation nodes in level NextBranch.L
7: **for each** $n \in N_{rlx}$ **do**
8: **if** n is a leaf **then** ▷ i.e., no further relaxation is possible
9: $n.UB$,$n.LB$ ← CostLeaf(n)
10: **else**
11: Compute $n.UB$, $n.LB$ for the specific objective
12: UPDATENODES(NextBranch.L)
13: PRUNE ($\mathcal{T}$)
14: $\mathcal{T}_c$ ← $\mathcal{T}$.Children ▷ children of the root node
15: NextBranch.L ← $L + 1$
16: **for each** $n \in \mathcal{T}_c$ **do**
17: **if** $n.L = |\mathfrak{Constrs}(Q_{start})|$ **then**
18: $\mathcal{T}_c \leftarrow \mathcal{T}_c \setminus$ {NextBranch}
19: NextBranch ← $\arg\min_{n \in \mathcal{T}_c}$ {n.UB - n.LB} ▷ min ub-lb strategy
20: **until** NextBranch not NULL
21: **return** COMPOPTCOST($\mathcal{T}$)

22: **procedure** UPDATENODES(Level L)
23: $\mathcal{N}_L$ ← nodes at level L
24: **for each** $n \in \mathcal{N}_L$ **do**
25: **if** n is a choice node **then**
26: Compute probabilities as in Algorithm 1
27: r_{yes}, r_{no} ← "yes" and "no" children of n
28: n.LB ← $P_{yes} * (C1 + r_{yes}.LB) + P_{no} * (C1 + r_{no}.LB)$
29: n.UB ← $P_{yes} * (C1 + r_{yes}.UB) + P_{no} * (C1 + r_{no}.UB)$
30: **else if** n is a relaxation node **then**
31: n.UB ← $\underset{c \in n.Children}{\text{optimize}}$ (n.UB)
32: n.LB ← $\underset{c \in n.Children}{\text{optimize}}$ (n.LB)

33: **procedure** PRUNE(Node r)
▷ r.UB < n.LB if the objective is to maximize
34: **if** exists $n \in r$.Siblings s.t. r.LB > n.UB **then**
35: n.Father.Children ← n.Father.Children $\setminus \{r\}$
36: **else**
37: **for each** $n \in$ r.Children **do**
38: PRUNE(n)

bound higher than the upper bound (or respectively, with an upper bound lower than the 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 and 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 Static 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 the 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 6 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.

4.3 FastOpt for top-k

Unlike the FullTree algorithm that constructs the entire tree and returns all the possible relaxations, the FastOpt does not guarantee to maintain at least k branches for each level. It computes the worst and best case by means of bounds and prunes a branch if it does not participate for sure to the optimal solution. This condition has to be adapted to the top-k scenario. In the case of top-k relaxations, a priority queue P of k best children has to be maintained for each subtree. A node can be safely pruned if there exist k elements with a lower (upper) bound greater (lower) than the kth biggest (smallest) upper bound.

Algorithm 3 shows the PRUNE procedure of FastOpt adapted for top-k case with Dynamic objective. All other objectives are easily computed with minimal changes in the code. The algorithm first computes a priority queue P containing sibling nodes in decreasing order of lower bound and the kth biggest lower bound (Lines 34-35). Then it removes from P any node with a lower bound greater than the kth biggest lower bound (Lines 37-39). Finally, the PRUNE procedure is called on the remaining nodes in P. It is easy to see that the algorithm preserves the completeness of the solution, given that the pruned nodes are those that for sure cannot eventually become part of the final solution.

Under the Skip-above model, FastOpt can be optimized further by reasoning on the values of the bounds of each node. Recall that the Skip-above model assumes that once an attribute is selected for relaxation, any attribute that is ranked higher cannot be relaxed. Therefore, in order to be certain that the removed attributes are better than the selected ones we have to reason on the bounds. In particular, given a choice node, any other sibling node having an upper (lower) bound greater (smaller) then the lower (upper) bound of that node is selected. Then, the attributes that should be relaxed by the selected sibling nodes cannot be relaxed any further.

Example 7 Consider the right tree in Figure 4. The node's (1,1,?) lower bound (0.46) is higher than the upper bound (0.22) of (?,1,1). Therefore, the first attribute cannot be relaxed anymore in the sub-tree rooted at (1,1,?), and the query becomes (#,1,?).

Algorithm 3 shows the pruning part involved in the Skip-above model. The PROPAGATE subroutine simply applies

Algorithm 3 FastOpt for top-k (Dynamic)

Input: Initial query Q_s, number of relaxations k
Output: Relaxation Cost of Q_s

```
    ⋮
34: procedure PRUNE(Node r)
35:     P ← {n} ∪ r.Siblings
36:     kUB ← kth highest UB
37:     for i = 1...k, p_i ∈ P do
38:         if p_i.LB > kUB then
39:             P ← P \ {p_i}
40:             p_i.Father.Children ← p_i.Father.Children \{p_i}
41:     if Skip-above then
42:         for i = k...1, p_i ∈ P do
43:             for j = (i − 1)...1, p_j ∈ P do
44:                 if p_i.UB < p_j.LB then
45:                     c ← attribute relaxed by p_j
46:                     Constrs(p_i.Q) ← Constrs(p_i.Q) ∪ Hard(c)
47:                     PROPAGATE(c, p_i)
48:     for each p ∈ P do
49:         PRUNE(p)
```

the same constraint on the children nodes, and prune the branches corresponding to the relaxed constraint if already constructed.

The performance of Algorithm 3 is clearly affected by the value k: a large k diminishes the chances to prune branches in advance. We show the relation between k and time in the experimental section.

5 Approximate Algorithms

5.1 CDR

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, the 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.

5.1.1 Cost Probability Distributions Computation

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* [4] of the *probability density functions* (pdf for short).[5] We adopt convolution of distributions definitions from previous work [4] to compute the probability 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 5 (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 6 (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)

[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.

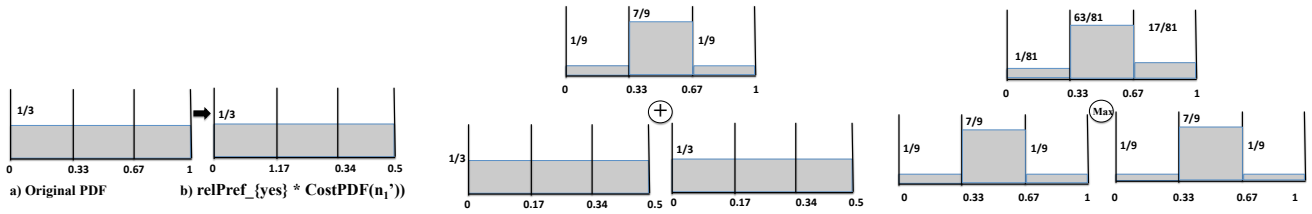


Fig. 5: $CostPDF(n)$ for (a) the "yes" branch of a choice node, (b) choice node, and (c) non-leaf relaxation nodes.

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

$$N_{rlx} = \arg\max_{n_i} \left(\Pr(Cost(n_i) \geq \prod_{j \neq i}^{k} Cost(n_j))\right)$$

($N_{rlx} = \arg\min_{n_i} \Pr(Cost(n_i) \leq \prod_{j \neq i}^{k} Cost(n_j))$, respectively)

Algorithm 4 CDR (Dynamic)

Input: Initial query Q_s, level L
Output: Relaxation node to be proposed at the root

```
for l = 1 ... L do
    N_rlx ← relaxation nodes in level l
    for each n∈N_rlx do
        if n is a leaf then                ▷ n is non-empty or not relaxable
            CostPDF(n) ← uniform in [1, |Constrs(Q_s)| − l]
for l = L ... 1 do
    UpdateRelaxationNodes(l)
    UpdateChoiceNodes(l)
N_u ← child nodes of root
return arg max_{n∈N_u} (Pr(Cost(n) ≤ ∏_{j=1, j≠n}^{|N_u|} Cost(n_j)))

procedure UpdateChoiceNodes(Level l)
    N_rsp ← choice nodes at level l
    for each n∈N_rsp do
        Compute probabilities as in Algorithm 1
        r_yes, r_no ← "yes" and "no" child of n
        CostPDF(n) ← P_yes * (C1 + CostPDF(r_yes)) +
                     P_no * (C1 + CostPDF(r_no))

procedure UpdateRelaxationNodes(Level l)
    N_rlx ← relaxation nodes at level l
    for each n∈N_rlx do
        N_u ← choice child nodes of n
        CostPDF(n) = min_{i=1..|N_u|} (CostPDF(n_i))
```

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

5.1.2 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 [4] 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 8 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.

5.2 FastCDR

Completely constructing L levels as required by the CDR when it needs to return top-k relaxations becomes computationally expensive. Since the FastOpt behaves like the FullTree when k is close to $|\mathfrak{Constrs}(Q)|$ we need a different algorithm to efficiently solve the problem while guaranteeing quality close to optimal.

CDR can be further optimized by removing from the search space non promising branches in the first L levels, and continuing the exploration over the remaining nodes. This can be naturally achieved using FastOpt in the first L levels and then expanding the the remaining tree using the CDR. Clearly, the solutions produced applying this strategy are as good as those of CDR, and hopefully better if the removed branch is the one that CDR would select for expansion. Furthermore, if the optimal solution is found within the first L levels, the time performance will be the same as FastOpt, eliminating the requirement of CDR that further expands one of the branches if the optimal is not detected in the first iteration. We refer to this new hybrid algorithm as Fast Cost Distribution Relaxation (of FastCDR). Algorithm 5 describes its steps in pseudocode. It works exactly as the CDR except that first it generates all the candidate nodes using FastOpt for top-k (see Line 13-27).

Algorithm 5 FastCDR (Dynamic)

Input: Initial query Q_s, level L, number of relaxations k
Output: top-k relaxations at root

```
1: GENERATENODES(L)
2: UPDATERELAXATIONNODESLOWEST(L)
3: for l = L..1 do
4:     UPDATERELAXATIONNODES(l)                ▷ see Algorithm 4
5:     UPDATECHOICENODES(l)                    ▷ see Algorithm 4
6: N_u ← child nodes of root
7: P ← N_u ordered by Pr(Cost(u) ≤ ∏_{∀j=1,j≠u}^{|N_u|} Cost(j))
8: return p_1, ..., p_k ∈ P

9: procedure UPDATERELAXATIONNODESLOWEST(Level L)
10:    N_rlx ← relaxation nodes at level L
11:    for each relaxation node n∈N_rlx do
12:        CostPDF(n) ← uniform in [1, |Constrs(Q)| − L]

13: procedure GENERATENODES(Level L)
14:    for l = 1 ... |L| do
15:        N_rlx ← relaxation nodes in level l
16:        for relaxation node u∈N_rlx do
17:            if u represents an empty answer query then
18:                UPDATERELAXATIONNODESLOWEST(u)
19:                u.LB, u.UB ← CostLeaf(n)
20:            else
21:                u.LB = 0
22:                u.UB = |Constrs(Q_s)| − L
23:        UPDATECHOICENODES(L)
24:        for TLev = (L − 1)..0 do
25:            UPDATERELAXATIONNODES(TLev)
26:            UPDATECHOICENODES(TLev)
27:        PRUNE(L)                            ▷ see Algorithm 2
```

6 Extensions

6.1 Databases with categorical and numerical attributes with hierarchies

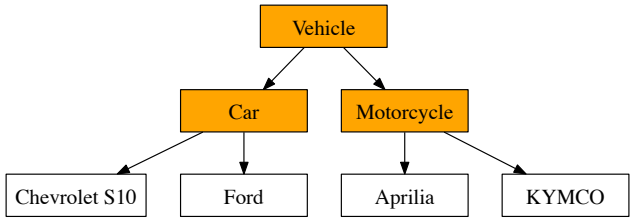
Our framework can ingest categorical data, provided that the data is organized and stored in a specific format. If not, a preprocessing step needs to be executed to bring the in that format, and then the algorithm can run as in the boolean database case.

Categorical attributes. The categorical data reorganization is performed attribute by attribute. Each value of a categorical attribute is an attribute itself in a boolean database. For instance, attribute *price*={*low, average*} is represented as two attributes *price-low* and *price-high* in the corresponding boolean database and a tuple has value 1 on *price-low* if the value of the attribute *price* is *low*. Given a query on categorical attributes, we convert it into boolean using the converted attributes. The rest of the computations in the framework remain unchanged.

The framework can also accommodate the case in which an order or a preference is induced on the categorical attribute values, by changing the $pref$ function to steer the user towards a specific attribute value.

Hierarchical attributes. The same idea applies to conceptual hierarchies (i.e., hierarchies that have a partial or total order of the attributes). In order words, we expand the query lattice (refer to Figure 2) using the hierarchies of each attribute, construct the query relaxation tree, and finally compute the cost. Additional priorities on the hierarchy attributes may be embedded in the *pref* computation. Figure 6 contains an illustration of the categorical data in a Boolean database. *Vehicle* contains two categories {*Car, Motorcycle*}. Each of

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



Vehicle	Car	Motorcycle	Chevrolet S10	Ford	Aprilia	KYMCO
1	1	0	0	1	0	0
1	0	1	0	0	0	1

Fig. 6: Categorical data representation in a Boolean database.

these categories contains the following subcategories {*Chevrolet S10, Ford*} and {*Aprila, KYMCO*}. An attribute is created for each category. Each tuple has value 1 or 0 for an attribute depending on whether the tuple represents an object belonging to that category or not. For instance, the attribute corresponding to category named "car" receives the value 1 for a tuple modeling a "Chevrolet S-10" car (because it belongs to the category "car") and 0 for "motorcycle" (because it does not belong to the category "motorcycle"). The table at the bottom of Figure 6 illustrates how a *Ford* car and a *KYMCO* motorcycle are stored in the Boolean database.

From an implementation point of view, to navigate through the hierarchies levels during the execution, the algorithm keeps an additional structure that stores information about the categorical attributes. During the interactive process with the user, and when the next best relaxation is computed on the fly, the algorithm will also check if any relaxation can be applied in the hierarchy levels (i.e., if the current attribute is a "car", then it is evaluated whether relaxing one level up to "vehicle" makes sense).

Numerical attributes. The *numerical attributes* are more elastic from a relaxation point of view as the numerical value itself of an attribute can be extended over large ranges of values. If we consider buckets over data ranges and hierarchies on top of them, this case is reduced to the hierarchical case and the above methods applies.

Our algorithms can equally ingest databases containing all these kinds of attributes simultaneously. However, we recall that the focus of the current work is not on optimized ways of operating with different types of data, but on the optimization of the interactive process with the user.

6.2 Drill down / Roll back

Proposing to the user the relaxations that at any given moment seem to be the most promising (according to the criteria we have already discussed) and the responses that the user has until that moment provided, may end up into a query that no more relaxations are possible or even if there are relaxations, they will not lead into a query with a non-empty answer. This type of relaxations are represented as a leaf in the query relaxation tree. If this happens, then instead of simply terminating the process, which may not be the best option for the user, we can retract one or more relaxations and follow an alternative path, allowing the process to continue.

There are many options that can be used to decide which alternative path to follow. For instance, one could: (i) go 1-relaxation back, and continue with the next best sibling; (ii) go t-relaxations back, and continue with the next best sibling, with t having some predefined value; (iii) go t-relaxations back, and continue with the next best sibling, with t being the minimum number of levels back s.t. the corresponding subtree has at least $m > 0$ tuples, and m being some parameter; (vi) ask the user to which of the attributes she selected so far she is willing to give up; ban that attribute, go back to the level where that attribute was relaxed and continue with the next best sibling; (v) same, but allow the user to select $t > 1$ attributes that she is willing to give up, and discard all of them at once. Go to the attribute at the highest level in the tree, and recompute from there and by not considering the other $t - 1$ attributes relaxations; (vi) same as before, but prioritize all other attributes that are not in the set of attributes under that attribute, and consider the rest only after this set of attributes is exhausted; and as a final alternative, (vii) once we go back to another attribute (selected by one of the previous ways), make zero all probabilities of all attributes the user selected after that attribute, and continue with the next best sibling.

Roll back algorithms. The FastOpt, CDR, and FastCDR can be adapted to accept roll-back operations. More specifically, when the user requires a roll-back operation, the algorithm locates the node the user wants to roll-back to, and computes the second best subtree if not already constructed. This operation is done on-demand, and it assumes that the new query contains all the attributes at that point in the tree, without the attribute that the user has just discarded. The algorithm then calls the procedure that constructs the new subtree with the new query, irrespective of the rest of the tree previously computed.

7 Experimental Evaluation

We present our experimental evaluation 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 16 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 empty-answer queries, initially containing only soft constraints. User preferences are simulated using our *relPref* value associated with each choice node.

Implementation. All the algorithms are implemented in a java framework that can be easily extended. The boolean database is loaded in the main memory and is represented as an array of integers. The array is sorted, therefore queries are efficiently answered in logarithmic time while using native boolean operations.

7.1 Implemented Algorithms

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. (Note that for Semi-Dynamic and Static we do not report the results using average as the leaf cost function, since they are similar to those obtained using the maximum.) Additionally, we implement FullTree, FastOpt, and our two parameterized algorithms CDR and FastCDR, as well as top-k variants of FastOpt and FastCDR that interactively propose k relaxations at each step. Finally, we implement a method (described in Section 3.3) that combines FastOpt with [27], using Dynamic and Semi-Dynamic objectives, which we refer to as FastOptMFS.

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

Related Works. In this group of algorithms we have considered a number of approaches from the related literature.

• 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 [51], non-trivially adapted for the empty-answer problem. We note that method [51] 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 an empty-answer query, we apply our $pref$ 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 [51] to answer the corresponding Why-Not query. The algorithm generates a set of relaxations that we present to the user.

• MFS and InteractiveMFS: These algorithms are from [27], and they propose all the maximally succeeding subqueries to the user. Based on the paper mentioned above, we implement two algorithms: InteractiveMFS is the MFS algorithm with the backtrack functionality, presented in Figure 9 of [27]. In addition, the algorithm requires as user input a preference order on the attributes; we used our preference function to compute an ordering and compare the results. Even though a thorough experimental assessment is missing from the original paper, we include such a comparison in Section 7.3.

• QueryRef: This algorithm is from [35]. 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, depicted in Figure 7, 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). Therefore, [35] largely fails to successfully address the empty-answer problem at hand. In order to remedy this situation, we also experiment with an adaptation of the algorithm that proposes additional minimal relaxations if the relaxation along one attribute is non-empty.

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 other two user studies: a separate study that validates the effectiveness of different cost-functions supported by our framework, and a comparison between single and top-k relaxations produced by our methods. Pareto-optimal solutions are considered and compared in Section 7.3. An empirical comparison among the different objectives is presented in Section 7.4. Section 7.5 presents quality assessment to experimentally demonstrate the effec-

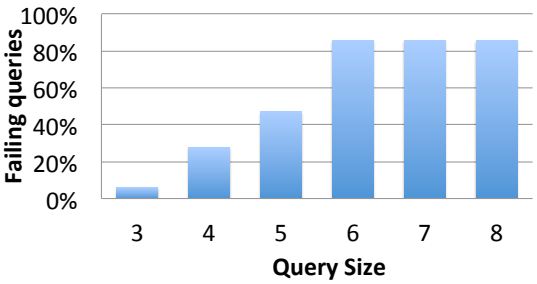


Fig. 7: Failing queries in QueryRef vs query size.

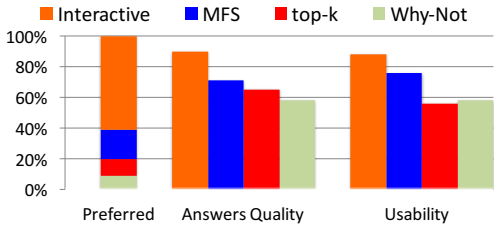


Fig. 8: Comparison of user satisfaction with different related work.

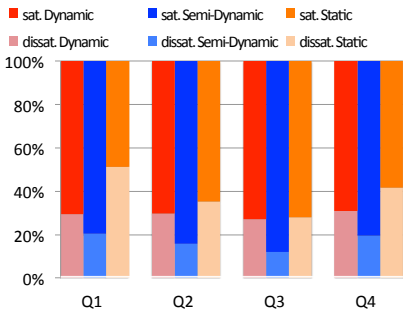


Fig. 9: Percentage of satisfied and dissatisfied users.

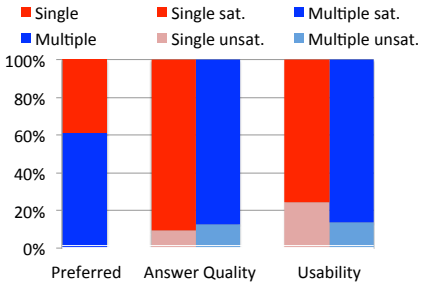


Fig. 10: Comparisons of single ($k = 1$) and multiple ($k > 1$) relaxations in terms of user satisfaction.

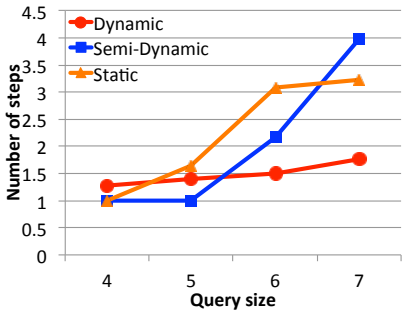


Fig. 11: Number of steps vs query size in the user study.

tiveness of our proposed framework in optimizing the preferred cost function (by the cost of the root node of the relaxation tree). Section 7.6 presents the scalability studies. Section 7.7 and Section 7.8 present the results at increasing k and cardinality, respectively. Section 7.9 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 [21,51] (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). (6) FastCDR preserves quality close to optimal and real-time performance at increasing k.

7.2 User Study

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

Comparison to previous work. 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), 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

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

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).

Optimization goals comparison. 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 Human Intelligence Task (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 11 shows that Dynamic guides users to non-empty results 2 times faster than the other approaches when the query size increases. The results (Figure 9) 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%) .

Top-k quality. In this user study we compare the quality of the FastOpt when the number of relaxations proposed changes. The user does not provide any yes/no answer if multiple relaxations are shown, assuming only yes answers in that case. We ask the users to evaluate 5 different queries with single or multiple (2,3) relaxations proposed. At the end we propose to: choose among single and multiple relaxations (Preferred), evaluate the overall quality of the answers for single and multiple (Answer quality) and, evaluate the easiness of use (Usability) in a scale of 1 to 4. Users found both techniques useful and appreciated their features, with a slight preference for Multiple, mainly because of its small advantage in terms of usability.

7.3 Comparison to Pareto-optimal solutions

A Pareto-optimal solution proposes non-empty relaxed queries that are maximal, i.e., any super-query of a maximal non-empty query is empty. This notion has been extensively used in the query relaxation literature, and forms the basis of the two other interactive methods in the literature, namely QueryRef and InteractiveMFS. Figure 15 shows the comparison among the FastOpt, FastCDR, QueryRef, and MFS methods. We also compare with the FastOptMFS method that combines our FastOpt algorithm with MFS. We note that while QueryRef only allows the user to choose among the different proposed relaxations, InteractiveMFS includes a backtrack strategy, in which the user may refuse a relaxation.

Figure 15a shows that our methods relax more than InteractiveMFS. This is expected since our Dynamic objective captures the likelihood that the user accepts a relaxation in order to have fewer interactions (as depicted in Figure 15b). Therefore, with high probability it will relax one attribute to return non-empty answers. Instead, the random model implemented in InteractiveMFS does not ensure this property, leading often times to an empty answer. QueryRef on the other hand, relaxes between 0.8 (query size 3) to 3.5 (query size 7) attributes more, since it does not incorporate any preference model and forces the user in selecting any of the maximal succeeding queries. Furthermore, our model (including the combined FastOptMFS) requires up to 1.2 less steps than InteractiveMFS and up to 2.3 less steps than QueryRef to get to an answer. QueryRef exhibits an answer quality close to our method: since it explores the whole query space, the probability to get high quality results is high, while our framework proposes one relaxation at a time.

7.4 Preference Computation Comparison

Figure 12 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 13 shows that Static achieves considerably better profit results, which means that the extra cost incurred pays off. Figure 14 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.

7.5 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 and we compare algorithm based on the cost function value at the root (see Equation 3). In these

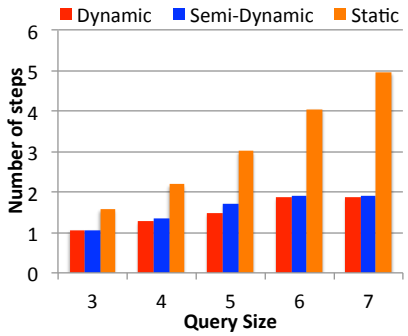


Fig. 12: Effort vs Query Size of different preference objective functions (Homes dataset).

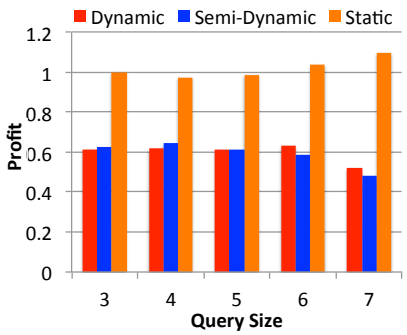


Fig. 13: Profit vs query size of different preference objective functions (Homes dataset).

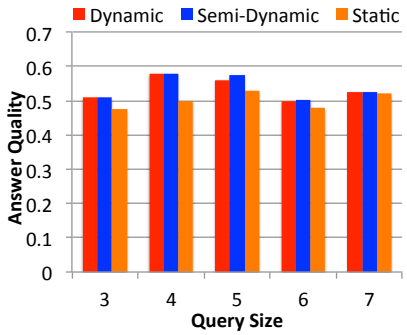
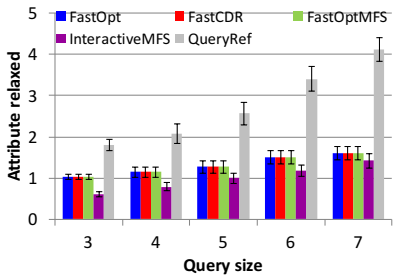
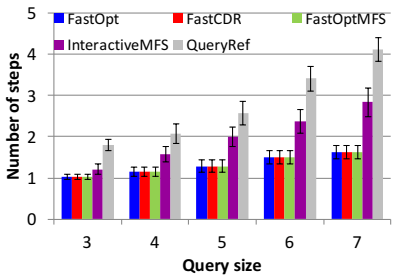


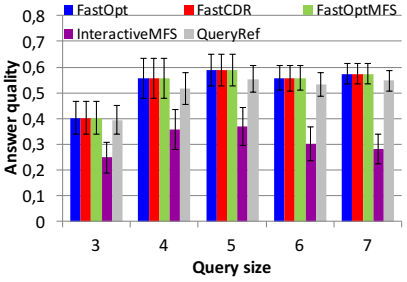
Fig. 14: Quality of the results vs query size for different objective functions (Homes dataset).



(a) Number of attribute relaxed vs query size (Homes dataset), Dynamic objective.



(b) Number of steps vs query size (Homes dataset), Dynamic objective.



(c) Quality of the results vs query size (Homes dataset), Semi-Dynamic objective.

Fig. 15: Comparison with the interactive Pareto-optimal approaches FastOptMFS, QueryRef and MFS.

experiments, we use queries of size up to 7, because this is maximum possible size for running FullTree in our experimental setup. Figure 16a 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.

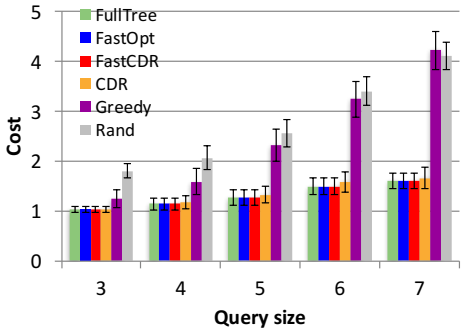
The graph confirms the intuition that the Random and Greedy algorithms are not able to find the optimal solution (i.e., the solution with the minimum expected number of relaxations). In addition, their relative performance gets worse as the size of the query increases, since the likelihood of making non-optimal choices increases as well. For query size 7, Random produces a solution that needs 2.5 times more relaxations than the optimal one and 2.6 for Greedy. As expected, they also exhibit the largest variance in performance.

On the other hand, CDR and FastCDR perform 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 16b), 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 optimal solution (expressed by FastOpt in the graph) corroborating its effectiveness to the empty-answer problem. Interestingly, FastCDR, due to the pruning strategy of FullTree, guarantees nearly optimal answers even for large queries. The results demonstrate that CDR and FastCDR are effective solutions to the empty-answer problem, even when the query size grows much larger than L (set to 3 for all the experiments).

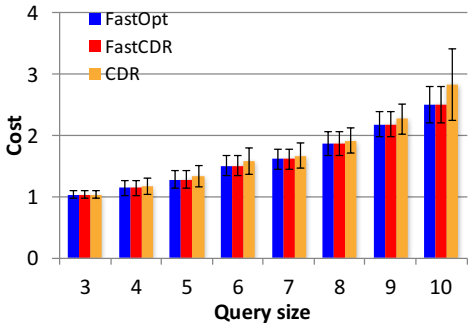
7.6 Scalability

Figure 17 illustrates experiments on the scalability properties of the top performers, FullTree, FastOpt, CDR and FastCDR, when both the size of the query and the size of the database increase.

Figure 17a shows 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), but worse than FastCDR that combines the characteristics of both. This behavior is explained by the fact that CDR is always computing all the nodes of the relaxation tree up to level L, while FastCDR applies pruning rules to speed up the computation. In contrast, FastOpt is able to prune several of these nodes, leading to a significantly smaller tree. For query sizes larger than 8, CDR and FastCDR compute more than two order of magnitudes less relaxation nodes than FastOpt. Moreover, as the query size increase CDR performs close to FastCDR since the

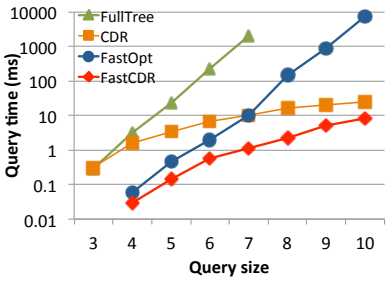


(a) Proposed algorithms and baselines comparison

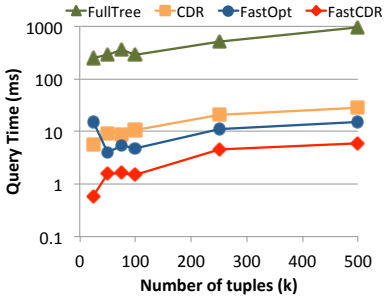


(b) FastOpt, (Fast)CDR-3 comparison for big queries

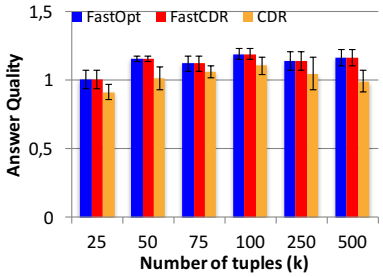
Fig. 16: Relaxation cost vs query size (Homes dataset).



(a) Query time (log scale) vs query size (Homes dataset).



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



(c) Relaxation cost vs query size, for FastOpt, (Fast)CDR-3 (Homes dataset).

Fig. 17: Results with increasing query size.

time to compute the levels below L dominates the computation. 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 almost 10 sec for queries of size 10. For the same queries, FastCDR executes three orders of magnitude faster, requiring 8.3 ms to produce the next relaxation, and significantly less than 1 ms for smaller queries.

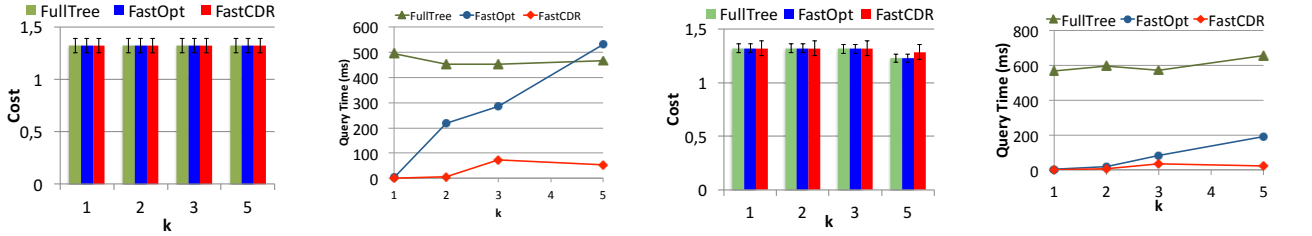
We also experiment with varying dataset sizes between 25K-500K tuples, using the Cars-X datasets, having query size set to 6. Figure 17b indicates that query time is moderately affected by size of the database, since relaxation tree becomes smaller with increasing dataset size even though the database access time increase. 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 involve more nodes than FastOpt, it performs similarly, 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.

We also show in Figure 17c the impact to the answer quality (Semi-Dynamic preference computation) as a function of database size. As expected, we obtain more qualitative answers with bigger databases, since the likelihood of having non-empty queries with good results also increase. As per quality comparison, FastCDR performs as well as FastOpt with different optimization criteria.

Overall, we observe that FullTree quickly becomes inapplicable in practice, while FastOpt is useful only for small query sizes. In contrast, CDR and FastCDR are able to propose relaxations in less than 10 ms, even for queries with 10 attributes, and always produces a solution that is very close to optimal.

7.7 Impact of k in the top-k relaxations

In the next set of experiments, we show the impact of the k in the top-k relaxations, with the No-bias and Skip-above models. The FastOpt algorithm is compared with the FastCDR in terms of quality and time.



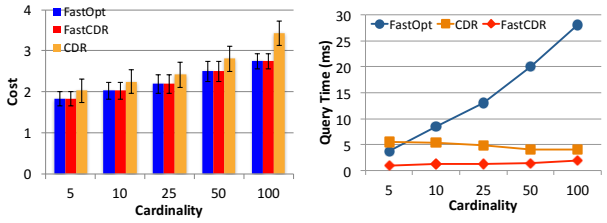
(a) Cost vs number of relaxations, when algorithms use the No-bias model

(b) Time vs number of relaxations, when algorithms use the No-bias model

(c) Cost vs number of relaxations, when algorithms use the Skip-above model

(d) Time vs number of relaxations, when algorithms use the Skip-above model

Fig. 18: CDR and FastCDR behavior with increasing number of relaxations proposed at each step.



(a) Cost vs cardinality

(b) Time vs cardinality

Fig. 19: CDR behavior with increasing cardinality in Homes dataset.

Figure 18a shows that in terms of quality, FastCDR for all practical purposes performs as good as optimal (the differences in performance are so small that they are not visible in the graphs), for different values of k.

Figure 18b shows the query time of each method at increasing k varying the number of reformulations and averaging the results obtained with query size 3-7. FastCDR takes nearly constant time in k and returns results one order of magnitude faster than the optimal. On the other hand, FastOpt query time increases linearly with k performing worse than FullTree with $k = 5$. The reason is that the pruning power of FastOpt is affected by k, since the number of branches in the relaxation to be constructed is bigger. Moreover, the overhead induced by the pruning procedure negatively affects the query time.

Figures 18(c-d) show the results with the Skip-above model. FastCDR produces slightly worse results in this case, since the choice of the subtree to expand is made before knowing which the best choice is (and therefore without knowing which attribute cannot be relaxed further). Unsurprisingly, the presentation bias pruning introduced in FastOpt positively affects the query time.

7.8 Cardinality Impact

We analyze the impact of cardinality on our different methods. A high cardinality value tends to produce deeper trees, affecting time and quality. Figure 19b shows that FastOpt takes 6 times more when the cardinality moves from 5 to 100. The approximate algorithms CDR and FastCDR are constant in the cardinality value. Moreover, as depicted in Figure 19a, the quality of the results produced by FastCDR is not affected by the cardinality value.

7.9 Calibrating (Fast)CDR

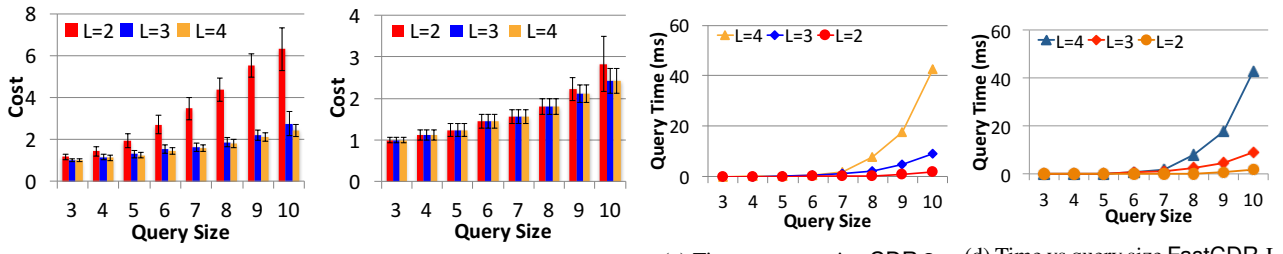
Recall that the (Fast)CDR-L algorithms start by computing all the nodes of the relaxation tree for the first L levels (see Section 5), where L is a parameter.

Figures 20a,20b show the impact of L on the cost (the values have been normalized using as a base the cost of (Fast)CDR-4 for query size 3). We notice that for $L = 2$ the CDR behaves reasonably well only for very small query sizes, while FastCDR produces qualitative relaxations up to query size 8. This behavior is expected, since for big query sizes 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 Figures 20c,20d. The results show that (Fast)CDR-4 quickly becomes expensive in terms of time. We conclude that using $L = 3$, CDR and FastCDR achieve the desirable trade-off between effectiveness and efficiency.

We also conduct experiments in Figure 21b with varying the number of the FastCDR histogram buckets between $5-40$, which has a negligible impact on time performance. We experience in Figure 21a that with more than 20 buckets the effect on the quality is minimal. The algorithm is also stable with respect to the data used. For the rest of the experimental evaluation, we use (Fast)CDR with $L = 3$, and 20 buckets.

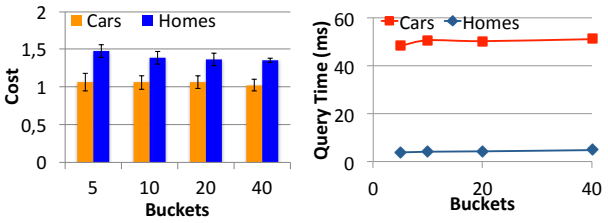
8 Related Work

Query Reformulation for Structured Databases. The closest related works are interactive optimisation-based approaches



(a) Cost vs query size CDR-L (b) Cost vs query size FastCDR-L (c) Time vs query size CDR-L (d) Time vs query size FastCDR-L

Fig. 20: CDR and FastCDR behavior for different values of L at increasing query size.



(a) Cost vs number of buckets (b) Time vs number of buckets

Fig. 21: CDR behavior with increasing number of buckets in Homes and Cars datasets.

studied for the *many-answers* problem, most notably [7,30, 32], 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 proposes query modification based on a notion of generalisation, and identifies the conditions under which a generalisation is applicable [13]. Alternatively, machine learning can be used to identify the reasons, i.e., combination of conditions, for an empty-answer query and eliminate those that invalidate as many such combinations as possible [42, 43]. The latter approach is data driven, but a similar service can be achieved by exploiting schema information and in particular integrity constraints, which in turn can be used to inform the user about the query failing conditions [26]. All these frameworks do not leverage user preferences in deciding the relaxation. Koudas et al. [31] suggest alternative queries based on the "minimal" shift from the original query. In contrast, our method considers additional goals that could be optimised during the interaction.

"Why Not" queries are studied in [12, 51], where, given a query Q that did not return a set of tuples S that were expected to be returned, the goal is to design an alternate query Q' that (a) is very "similar" to Q, and (b) returns the missing tuples S, and (c) has the rest of the returned tuples not so different from those returned by Q. "Why Not" queries are non-interactive, and it is non-trivial to extend these methods for the empty answers problem, because they require the user to be aware of some desired tuples S in the database. In our case though, no such set S is available to the user.

Relaxation strategies for the empty-answer problem have been proposed as a recommendation service in [27, 28, 29]. Motro [37] proposed that the system may decide to relax a query if it is suspected to produce an empty answer due to some misspelled words, some wrong table/attribute selection, or some badly selected condition in the query. With this assumption, the relaxation is unavoidably based on a syntactic similarity function on schema elements [38]. This solution has the advantage that it can be easily explained to the user [39]. A common characteristic of all the above approaches is that they are non-interactive.

A main challenge in all the relaxation works is how to select the best relaxation among all alternatives. Bosc et al. [9] make use of a parameterized proximity relation, in order to drive the relaxations. While this is a meaningful framework for query relaxation, in our work, we additionally take into account the user's prior and preference functions, while providing the ability to use a wide range of objective functions to quantify relaxations. Furthermore, this approach assumes empty flexible (fuzzy) queries, while our work focuses on relaxing empty crisp queries.

A few interactive query relaxation approaches have been proposed for the empty-answer problem. McSherry [34] proposes an interactive strategy, where attribute are relaxed one-by-one, based on a deterministic partial/total order over the attributes. Nevertheless, no optimization criterion is explicitly described. A recent paper [35] 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 some objective, nor does it consider a model for user preferences. We have provided an empirical study of this approach in Section 7.1.

An alternative to query reformulation approaches for solving the empty/many-answers problem is the *ranked-retrieval* approach. The task is to develop a *ranking function* that scores all items (even those that do not exactly match the query conditions) in the repository, according to a "degree" of preference of 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. 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 [6] and database [1, 14, 15] researchers. At the same time, it has also been recognized [7,25] 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 ranking function should be changed. In this context, interactive approaches such as query reformulation are popular alternatives.

Our approach can be considered as an instance of associative query answering [17] that returns data related to, but not explicitly asked by the query. The relatedness may be based on external information, e.g., a domain ontology, while our approach is data- and user-driven. In the same spirit, taxonomies have been used in deductive databases to drive the query relaxation process [19], using greedy strategies. At large scale, the system may require techniques for cooperative query answering that are suited for a distributed environment [49]. No matter what approach is used for the query relaxation, performance is always an issue due to the many alternatives that need to be investigated. Performance optimization techniques vary depending on the approach. Examples include those we presented in this paper, efficient lattice traversal techniques [49], subquery enumerations [23], or monotone CNF formulas [18], even query caching which allows a-priori testing for empty-answer situations [33].

Most of the works on query relaxation do not consider past knowledge. Bosc et al. [10] describe an incremental approach, yet, based on past failed queries. The past knowledge our approach considers does not only include the previously failed queries, but also the previous choices of the user during the interaction phase. Thus, the user preferences are fully taken into consideration. The work in [50] consults a database of historical items that match the query, and in combination with predefined taxonomies, it decides which relaxation is likely to yield results that align well with the user's original intent.

Skyline queries is a popular problem with many variations in the literature. In principle, the goal is to find a set of tuples that are not dominated by any other tuple. A tuple t dominates another tuple s if t is strictly better than s in all attribute values (assuming an ordering on the values). The complexity of finding dominating tuples is $\mathcal{O}(n^2)$ in the worst case; though, in practice they perform better. This problem has also been studied for subspaces (i.e., subsets of the database attributes) [16, 46]. Highly related are also the preference queries that try to satisfy conditions and preference ordering on sets by selecting a set of items as an answer [53]. However, skyline queries do not deal in principle with the empty-answer problem, which is tackled instead by maximal succeeding subqueries methods [27, 35].

Query Reformulation in IR. Automatic query reformulation strategies for keyword queries over text data have been widely investigated in the IR literature [21,22]. 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 [5], query clustering [52], or exploiting summaries contained in the query-flow graph [3]. An alternative approach relies on suggesting keyword relaxations by relaxing the ones which are least specific based on their idf score [25].

Zero-hit queries & HCI. The empty-answer problem has also been studied from the HCI point of view, under the name of *zero-hit queries*. The main idea is to provide the user with the visual tools to adjust an empty-answer query so that it generates data. The Aplhaslider [2] is one such example. Using key values sampled from a specific attribute is providing the user with an overview of the value distribution, and is allowing her to drill down and adjust the query conditions in order to reach a non-empty answer query. Instead of giving the user a summary of the values, it is possible to provide some meta-data information [24]. The user can then perform some selection based on the meta-information before having the actual data retrieved. The latter approach works not only for the case of the empty-answer, but also in the case of the too-many answers [47]. Both approaches can benefit from our framework, so as to better reason on the possible relaxations and avoid unsuccessful trials by the user aiming at achieving a non-empty answer.

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. The holistic framework is adapted to return top-k reformulations

at each step and to include cardinality constraint. Moreover, any database can be translated into boolean and the same techniques applied. 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, by comparing to several techniques and user studies.

References

1. S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.
2. C. Ahlberg and B. Shneiderman. The alphaslider: a compact and rapid selector. In *CHI*, page 226, 1994.
3. A. Anagnostopoulos, L. Becchetti, C. Castillo, and A. Gionis. An optimization framework for query recommendation. In *WSDM*, pages 161–170, 2010.
4. 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.
5. 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.
6. R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley New York, 2011.
7. 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.
8. Y. M. M. Bishop, S. E. Fienberg, and P. W. Holland. *Discr. Multivariate Analysis: Theory and Practice*. MIT Press, 1975.
9. P. Bosc, A. HadjAli, and O. Pivert. Empty versus overabundant answers to flexible relational queries. *Fuzzy Sets and Systems*, 159(12):1450–1467, 2008.
10. P. Bosc, A. HadjAli, and O. Pivert. Incremental controlled relaxation of failing flexible queries. *JIIS*, 33(3):261–283, 2009.
11. Y. Chang, I. Ounis, and M. Kim. Query reformulation using automatically generated query concepts from a document space. *Information processing & management*, 42(2), 2006.
12. A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.
13. S. Chaudhuri. Generalization and a framework for query modification. In *ICDE*, pages 138–145, 1990.
14. S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *VLDB*, pages 888–899, 2004.
15. S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic information retrieval approach for ranking of database query results. *TODS*, 31(3):1134–1168, 2006.
16. J. Chomicki, P. Ciaccia, and N. Meneghetti. Skyline queries, front and back. *SIGMOD Record*, 42(3):6–18, 2013.
17. W. W. Chu and Q. Chen. Neighborhood and associative query answering. *J. Intell. Inf. Syst.*, 1(3/4):355–382, 1992.
18. C. Domingo, N. Mishra, and L. Pitt. Efficient read-restricted monotone CNF/DNF dualization by learning with membership queries. *Machine Learning*, 37(1):89–110, 1999.
19. T. Gaasterland. Cooperative answering through controlled query relaxation. *IEEE Expert*, 12(5):48–59, 1997.
20. M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.
21. S. Gauch and J. Smith. Search improvement via automatic query reformulation. *TOIS*, 9(3):249–280, 1991.
22. S. Gauch and J. B. Smith. An expert system for automatic query reformulation. *JASIS*, 44(3):124–136, 1993.
23. P. Godfrey. Minimization in cooperative response to failing database queries. *Int. J. Cooperative Inf. Syst.*, 6(2):95–149, 1997.
24. S. Greene, E. Tanin, C. Plaisant, B. Shneiderman, L. Olsen, G. Major, and S. Johns. The end of zero-hit queries: Query previews for nasa's global change master directory. *Int. J. on Digital Libraries*, 2(2-3):79–90, 1999.
25. 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.
26. J. M. Janas. On the feasibility of informative answers. In H. Gallaire, J. Minker, and J. Nicolas, editors, *Advances in Data Base Theory*, pages 397–414. Springer, 1981.
27. D. Jannach. Techniques for fast query relaxation in content-based recommender systems. *KI'06: Advances in AI*, pages 49–63, 2007.
28. D. Jannach and J. Liegl. Conflict-directed relaxation of constraints in content-based recommender systems. *Advances in Applied AI*, pages 819–829, 2006.
29. U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, volume 4, pages 167–172, 2004.
30. A. Kashyap, V. Hristidis, and M. Petropoulos. Facetor: cost-driven exploration of faceted query results. In *CIKM*, 2010.
31. N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.
32. 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.
33. G. Luo. Efficient detection of empty-result queries. In *VLDB*, pages 1015–1025, 2006.
34. D. McSherry. Incremental relaxation of unsuccessful queries. In *ECCBR*, pages 331–345, 2004.
35. C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873. ACM, 2009.
36. T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
37. A. Motro. Seave: A mechanism for verifying user presuppositions in query systems. *TOIS*, 4(4):312–330, Dec. 1986.
38. A. Motro. Vague: A user interface to relational databases that permits vague queries. *TOIS*, 6(3):187–214, July 1988.
39. A. Motro. Flex: a tolerant and cooperative user interface to databases. *TKDE*, 2(2):231–246, Jun 1990.
40. D. Mottin, A. Marascu, S. Basu Roy, G. Das, T. Palpanas, and Y. Velegrakis. Iqr: An interactive query relaxation system for the empty-answer problem. In *SIGMOD*, pages 1095–1098, 2014.
41. D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis. A probabilistic optimization framework for the empty-answer problem. *PVLDB*, 6(14):1762–1773, 2013.
42. I. Muslea. Machine learning for online query relaxation. In *KDD*, pages 246–255, 2004.
43. I. Muslea and T. J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.
44. T. Palpanas and N. Koudas. Entropy based approximate querying and exploration of datacubes. In *SSDBM*, pages 81–90, 2001.
45. T. Palpanas, N. Koudas, and A. O. Mendelzon. Using datacube aggregates for approximate querying and deviation detection. *IEEE Trans. Knowl. Data Eng.*, 17(11):1465–1477, 2005.
46. J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.
47. C. Plaisant, B. Shneiderman, K. Doan, and T. Bruns. Interface and data architecture for query preview in networked information systems. *ACM Trans. Inf. Syst.*, 17(3):320–341, 1999.
48. F. Radlinski and T. Joachims. Query chains: learning to rank from implicit feedback. In *KDD*, pages 239–248. ACM, 2005.
49. Z. W. Ras and A. Dardzinska. Solving failing queries through cooperation and collaboration. *WWW*, 9(2):173–186, 2006.
50. G. Singh, N. Parikh, and N. Sundaresan. Rewriting null e-commerce queries to recommend products. In *WWW*, 2012.
51. Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.
52. J.-R. Wen, J.-Y. Nie, and H. Zhang. Query clustering using user logs. *TOIS*, 20(1):59–81, 2002.
53. X. Zhang and J. Chomicki. Preference queries over sets. In *ICDE*, pages 1019–1030, 2011.