Machine Learning for Online Query Relaxation

Ion Muslea
SRI International
333 Ravenswood
Menlo Park, CA 94025
ion.muslea@sri.com

ABSTRACT

In this paper we provide a fast, data-driven solution to the *failing query* problem: given a query that returns an empty answer, how can one relax the query's constraints so that it returns a non-empty set of tuples? We introduce a novel algorithm, LOQR, which is designed to relax queries that are in the disjunctive normal form and contain a mixture of discrete and continuous attributes. LOQR discovers the implicit relationships that exist among the various domain attributes and then uses this knowledge to relax the constraints from the failing query.

In a first step, LOQR uses a small, randomly-chosen subset of the target database to learn a set of decision rules that predict whether an attribute's value satisfies the constraints in the failing query; this *query-driven* operation is performed *online* for each failing query. In the second step, LOQR uses nearest-neighbor techniques to find the learned rule that is the most similar to the failing query; then it uses the attributes' values from this rule to relax the failing query's constraints. Our experiments on six application domains show that LOQR is both robust and fast: it successfully relaxes more than 95% of the failing queries, and it takes under a second for processing queries that consist of up to 20 attributes (larger queries of up to 93 attributes are processed in several seconds).

Categories and Subject Descriptors

I.2.6 [**Artificial intelligence**]: Learning

General Terms

Algorithms, Experimentation

Keywords

online query relaxation, failing query, rule learning, nearest neighbor, Web-based information sources

1. INTRODUCTION

The proliferation of online databases lead to an unprecedented wealth of information that is accessible via Web-based interfaces. Unfortunately, exploiting Web-based information sources is non-trivial because the user has only *indirect access* to the data: one cannot browse the whole target database, but rather only the tuples that satisfy her queries. In this scenario, a common problem for the casual user is coping with *failing queries*, which do not return any answer. Manually relaxing failing queries is a frustrating, tedious, time-consuming task because, in the worst case, one must consider an exponential number of possible relaxations (i.e., various combinations of values for each possible subset of attributes); furthermore, these difficulties are compounded by the fact that over-relaxing the query (i.e., weakening the constraints so that there are many tuples that satisfy them) may lead to prohibitive costs in terms of bandwidth or fees to be paid per returned tuple. Researchers have proposed automated approaches to query relaxation [12, 9, 7], but the existing algorithms have a major limitation: the knowledge used in the query relaxation process is acquired *offline*, independently of the failing query to be relaxed.

We introduce here an *online, query-guided* algorithm for relaxing failing queries that are in the disjunctive normal form. Our novel algorithm, LOQR[1], uses a small, randomly-chosen subset $\mathcal{D}$ of the target database[2] to discover implicit relationships among the various domain attributes; then it uses this extracted domain knowledge to relax the failing query.

Given the small dataset $\mathcal{D}$ and a failing query, LOQR proceeds as follows. In the first step, it uses $\mathcal{D}$ to learn decision rules that predict the value of *each* attribute from those of the other ones. This learning step is *on-line* and *query-guided*: for each attribute *Attr* in the failing query, LOQR uses the other attributes' values to predict whether the value of *Attr* satisfies the query's constraints on it. Then, in a second step, LOQR uses nearest-neighbor techniques to find the learned rule that is the most similar to the failing query. Finally, LOQR relaxes the query's constraints by using the attribute values from this "most similar" rule.

For example, consider an illustrative laptop purchasing scenario in which the query

$$Q: \; Price \leq \$2,000 \bigwedge Display \geq 17'' \bigwedge Weight \leq 3\,lbs$$

[1] LOQR stands for Learning for Online Query Relaxation

[2] As we will show in the empirical validation, as long as the examples in $\mathcal{D}$ are representative of those in the target database $\mathcal{TD}$, $\mathcal{D}$ and $\mathcal{TD}$ can be - in fact - disjoint.

KDD'04, August 22–25, 2004, Seattle, Washington, USA.

fails because large-screen laptops weigh more than three pounds. In the first step, LOQR uses a small dataset of laptop configurations to learn decision rules such as

$$R: \quad Price \leq \$2,300 \bigwedge Cpu < 2.1\,GHz \bigwedge Display \leq 13'' \\ \Rightarrow Weight \leq 3\,lbs$$

The right-hand side (i.e., the consequent) of this rule consists of a constraint from the failing query, while the left-hand side specifies the conditions under which this constraint is satisfied in the dataset $\mathcal{D}$. After learning such rules for each constraint in the query, LOQR uses nearest-neighbor techniques to find the learned rule that is the most similar to the failing query (for the time being, let us assume that the rule R is this "most similar" rule).

To relax the failing query, LOQR "fuses" the constraints on the attributes that appear in *both* Q and R. In our example, the relaxed query

$$Q_R: \quad Price \leq \$2,300 \bigwedge Weight \leq 3\,lbs$$

is obtained as follows: R's constraint on CPU is ignored because CPU does not appear in Q. Q's constraint on the screen size is dropped because it conflicts with the one in R ($Display$ cannot be simultaneously smaller that 13" and larger than 17"). Finally, Q's price limit is increased to the value in R, which is less constraining than the original amount of $\$2,000$. Note that this price increase is crucial for ensuring that Q_R does not fail: as long as the constraints in Q_R, which are a subset of R's, are *at most* as tight as those in R, Q_R is *guaranteed* to match *at least* the tuples covered by R (i.e., the examples from which R was learned).

In summary, the main contribution of this paper is a novel, data-driven approach to query relaxation. Our online, query-guided algorithm uses a small dataset of domain examples to extract implicit domain knowledge that is then used to relax the failing query. The rest of the paper is organized as follows: after discussing the related work, we present a detailed illustrative example. Then we describe the LOQR algorithm and discuss its empirical validation.

2. RELATED WORK

Query modification has long been studied in both the fields of databases and information retrieval [15, 18, 10, 13, 17, 4, 19, 2, 5, 3]. More recently, with the advent of XML, researchers have proposed approaches for approximate pattern matching [25], answer ranking [11], and XML-oriented query relaxation [1, 16, 20].

CO-OP [18] is the first system to address the problem of *empty answers* (i.e., failing queries). CO-OP is based on the idea of identifying the failing query's *erroneous presuppositions*, which are best introduced by example. Consider the query ``get all people who make less than $15K and work in the R & D department at the Audio, Inc. company''. Note that this query may fail for two different reasons: either nobody in Audio, Inc.'s R & D department makes less than $15K or the company does not have an R & D department. The former is a *genuine* null answer, while the latter is a *fake* empty answer that is due to the *erroneous presuppositions* that the company has an R & D department. CO-OP, which focuses on finding the erroneous presuppositions, transforms the original query into an intermediate, graph-oriented language in which the connected sub-graphs represent the query's presuppositions; if the original query fails, then CO-OP tests each presupposition against the database by converting the subgraphs into sub-queries.

The FLEX system [23] can be seen as generalizing the ideas in CO-OP. FLEX reaches a high tolerance to incorrect queries by iteratively interpreting the query at lower levels of correctness. FLEX is also *cooperative* in the sense that, for any failing query, it provides either an explanation for the failure or some assistance for turning the query into a non-failing one. Given a failing query, FLEX generates a set of more general queries that allow it to determine whether the query's failure is *genuine* (in which case it suggest similar, but non-failing queries) or *fake* (in which case it detects the user's *erroneous presuppositions*).

The main drawback of systems such as FLEX is their high computational cost, which comes from computing and testing a large number of presupposition (to identify the significant presupposition, a large number of queries must be evaluated *on the entire database*). In order to keep the running time acceptable, Motro [22] suggests heuristics for constraining the search. A related approach is proposed by Gaasterland [12], who controls the query relaxation process by using heuristics based on semantic query-optimization techniques. Finally, on the theoretical side, Godfrey [14] proves complexity results for finding all/some *minimal failing* and *maximal succeeding* sub-queries: finding all of them is NP-hard, while finding a subset takes polynomial time.

The CoBase system [8, 6, 9, 7] is the closest approach to our LOQR algorithm. Central to the CoBase approach is the concept of Type Abstraction Hierarchies (TAHs), which synthesize the database schema and tuples into a compact, abstract form. In order to relax a failing query, CoBase uses three types of TAH-based operators: generalization, specialization, and association; these operations correspond to moving up, down, or between the hierarchies, respectively. CoBase automatically generates the TAHs by clustering all the tuples in the database [7, 21].

CoBase is similar to LOQR in the sense that it uses machine learning techniques for relaxing the queries. However, the two approaches are radically different: in CoBase, the CPU-intensive clustering is performed only once, in an off-line manner, on all the tuples in the database. CoBase then uses this resulting TAH to relax *all* failing queries. In contrast, LOQR customizes the learned decision rules to each failing query by applying - online - the C4.5 learner [24] to a small, randomly-chosen subset of the database.

3. THE INTUITION

Let us consider again the illustrative laptop purchasing domain, in which the query

$$Q_0: \quad Price \leq \$2,000 \bigwedge CPU \geq 2.5\,GHz \bigwedge \\ Display \geq 17'' \bigwedge Weight \leq 3\,lbs \bigwedge HDD \geq 60GB$$

fails because of two independent reasons:

- laptops that have large screens (i.e., $Display \geq 17''$) weigh more than three pounds;
- fast laptops with large hard disks ($CPU \geq 2.5GHz \bigwedge HDD \geq 60GB$) cost more than \$2,000.

In order to relax Q_0, LOQR proceeds in three steps: first, it learns decision rules that express the implicit relationships among the various domain attributes; then it uses nearest-neighbor techniques to identify the learned rule that is most

Brand	Price	CPU	HDD	Weight	Screen
Sony	\$2899	3.0 GHz	40 GB	3.9 lbs	18“
Dell	\$1999	1.6 GHz	80 GB	3.6 lbs	12“
...	...	...	...	...	...

Table 1: The original dataset $\mathcal{D}$.

Brand	Price	CPU	HDD	Weight	Screen
Sony	\$2899	**YES**	40 GB	3.9 lbs	18“
Dell	\$1999	**NO**	80 GB	3.6 lbs	12“
...	...	...	...	...	...

Table 2: The newly created dataset D_1.

similar to the failing query; finally, it uses the attribute values from this most-similar learned rule to relax the constraints from the failing query.

3.1 Step 1: Extracting domain knowledge

In this step, LOQR uses the small, randomly-chosen subset $\mathcal{D}$ of the target database to discover knowledge that can be used for query relaxation. LOQR begins by considering Q_0's constraints independently of each other: for each constraint in Q_0 (e.g., $CPU \geq 2.5\,GHz$), LOQR uses $\mathcal{D}$ to find patterns that predict whether this constraint is satisfied. Intuitively, this corresponds to finding the "typical values" of the other attributes in the examples in which $CPU \geq 2.5\,GHz$.

For example, consider the dataset $\mathcal{D}$ that consists of the laptop configurations from Table 1. In order to predict, for any laptop configuration, whether $CPU \geq 2.5\,GHz$ is satisfied, LOQR uses $\mathcal{D}$ to create the additional dataset D_1 shown in Table 2. Note that each example in $\mathcal{D}$ is duplicated in D_1, except for the value of its CPU attribute. The original CPU attribute is replaced in a binary one (values **YES** or **NO**) that indicates whether $CPU \geq 2.5\,GHz$ is satisfied by the original example from $\mathcal{D}$. The new, binary CPU attribute is designated as the class attribute of D_1.

LOQR extracts the potentially useful domain knowledge by applying the C4.5 learner to the dataset D_1, thus learning a set of decision rules such as

$$\begin{aligned} R_1: \quad & Price \leq \$2,900 \bigwedge Display \geq 18'' \bigwedge Weight \leq 4\,lbs \\ & \Rightarrow IsSatisfied(CPU \geq 2.5\,GHz) == \textbf{YES} \\ R_2: \quad & Price \geq \$3,500 \\ & \Rightarrow IsSatisfied(CPU \geq 2.5\,GHz) == \textbf{YES} \\ R_3: \quad & Price \leq \$2,000 \bigwedge HDD \geq 60 \bigwedge Weight \leq 4\,lbs \\ & \Rightarrow IsSatisfied(CPU \geq 2.5\,GHz) == \textbf{NO} \end{aligned}$$

Such rules can be used for query relaxation because they describe *sufficient conditions* for satisfying a particular constraint from the failing query. In our example, the rules above exploit the values of the other domain attributes to predict whether $CPU \geq 2.5\,GHz$ is satisfied.

Besides D_1, LOQR also creates four other datasets $D_2 - D_5$, which correspond to the constraints imposed by Q_0 to the other domain attributes (i.e., *Price*, *HDD*, *Weight*, and *Display*). Each of these additional datasets is used to learn decision rules that predict whether Q_0's constraints on the corresponding attributes are satisfied.

Note that this learning process takes place *online*, for each individual query; furthermore, the process is also *query-guided* in the sense that each of the datasets $D_1 - D_5$ is created *at runtime* by using the failing query's constraints. This *online, query-guided* nature of the process is the key feature that distinguishes LOQR from existing approaches.

3.2 Step 2: Finding the "most useful" rule

At this point, we must emphasize that the rule R_1, R_2, and R_3 that were learned above can be seen as the existentially-quantified statements. For example, R_1 can be interpreted as the statement *"there are some examples in $\mathcal{D}$ that satisfy the condition*

$$\begin{aligned} Q_1: \quad & Price \leq \$2,900 \bigwedge Display \geq 18'' \bigwedge \\ & Weight \leq 4\,lbs \bigwedge CPU \geq 2.5\,GHz \end{aligned}$$

Consequently, if we apply Q_1 to $\mathcal{D}$, Q_1 is guaranteed *not* to fail because it certainly matches the examples covered by R_1 (i.e., the ones from which R_1 was learned). Furthermore, as $\mathcal{D}$ is a subset of the target database, it also follows that Q_1 is guaranteed *not* to fail on the target database.

In this second step, LOQR converts all the learned rules into the corresponding existential statements. Then it identifies the existential statement that is the *"most useful"* for relaxing the failing query (i.e., the one that is the *most similar* to Q_0). This "most similar" statement is found by nearest-neighbor techniques. For example, the statement Q_1 above is more similar to Q_0 than

$$\begin{aligned} Q_2: \quad & Price \leq \$3,000 \bigwedge Display \geq 18'' \bigwedge \\ & Weight \leq 4\,lbs \bigwedge CPU \geq 2.5\,GHz \end{aligned}$$

because Q_1 and Q_2 differ only on their constraint on *Price*, and Q_0's $Price \leq \$2,000$ is more similar (i.e., closer in value) to Q_1's $Price \leq \$2,900$ than to Q_2's $Price \leq \$3,000$. Likewise, Q_1 is more similar to Q_0 than

$$Q_3: \quad Brand == \text{“}Sony'' \bigwedge CPU \geq 2.5\,GHz$$

which shares only the CPU constraint with the failing query.

3.3 Step 3: Relaxing the failing query

For convenience, let us assume that of all learned statements from the datasets $D_1 - D_5$, Q_1 is the one most similar to Q_0. Then LOQR creates a relaxed query Q_r that contains only constraints on attributes that appear both in Q_0 and Q_1; for each of these constraints, Q_r uses the less constraining value of those in Q_0 and Q_1. In our example, the resulting relaxed query is

$$\begin{aligned} Q_r: \quad & Price \leq \$2,900 \bigwedge CPU \geq 2.5\,GHz \bigwedge \\ & Display \geq 17'' \bigwedge Weight \leq 4\,lbs \end{aligned}$$

which is obtained by dropping the original constraint on the hard disk (since it appears only in Q_0), keeping the constraint on CPU unchanged since (Q_0 and Q_1 have identical constraints on CPU), and setting the values in the constraints on *Price*, *Display*, and $Weight$ to the least constraining ones (i.e., the values from Q_1, Q_0, and Q_1, respectively).

The approach above has two advantages. First, as Q_1 is the statement the most similar to Q_0, LOQR makes minimal changes to the original failing query. Second, as the constraints in Q_r are a subset of those in Q_1, and they are *at most* as tight as those in Q_1 (some of them may use the looser values from Q_0), it follows that all examples that satisfy Q_1 also satisfy Q_r. In turn, this implies that Q_r is *guaranteed* *not to fail* on the target dataset because Q_1 satisfies *at least* the examples covered by R_1.[3]

Let us now briefly consider a more complex example of query relaxation. Suppose that instead of Q_1, the existential statement that is the most similar to Q_0 is

$$\begin{aligned} Q_4 : \quad & Price \geq \$2,500 \bigwedge Display \geq 18'' \bigwedge \\ & Weight \geq 2.5\, lbs \bigwedge CPU \geq 2.5\, GHz \end{aligned}$$

Note that for both $Price$ and $Weight$, the constraints in Q_0 and Q_4 use the "opposite" operators $\leq$ and $\geq$ (e.g., $\leq$ in Q_0 and $\geq$ in Q_4, or vice versa). When relaxing Q_0 based on the constraints in Q_4, LOQR creates the relaxed query

$$Q'_r : CPU \geq 2.5\, GHz \bigwedge Display \geq 17'' \bigwedge Weight \in [2.5, 3]$$

in which

- the $Weight$ is constrained to the values that are common to both Q_0 and Q_4 (i.e., $Weight \geq 2.5\, lbs$ and $Weight \leq 3\, lbs$ implies $Weight \in [2.5, 3]$);
- the constraint on $Price$ is dropped because there are no values of this attribute that can simultaneously satisfy the corresponding constraints from Q_0 and Q_4 (i.e., $Price \leq \$2,000$ and $Price \geq \$2,500$);
- the other constraints are obtained exactly in the same way as they were computed for Q_r.

3.4 Beyond the illustrative example

At this point we must make two important comments. First, the failing query Q_0 is just a conjunction of several constraints on the various attributes. In order to relax queries that are in *disjunctive normal form* (i.e., disjunctions of conjunctions similar to $Q_0 - Q_4$), LOQR simply considers the conjunctions independently of each other and applies the three steps above to each individual conjunction.

Second, the query-guided learning process above creates a dataset D_i for each attribute that is constrained in the failing query (e.g., the five datasets $D_1 - D_5$ for the query Q_0). This idea generalizes in a straightforward manner for the scenario in which the user is allowed to specify "hard constraints" that should *not* be relaxed under any circumstances: for each example in $\mathcal{D}$, the entire set of hard constraints is replaced by a single binary attribute that specifies whether or not *all* the hard constraints are *simultaneously* satisfied in that particular example. This is an additional benefit of our online, query-guided approach to query relaxation, and it is not shared by other approaches.

4. THE LOQR ALGORITHM

In this section we present a formal description of the LOQR algorithm. We begin by briefly describing the syntax of the input queries, after which we discuss the algorithm itself.

4.1 The query syntax

LOQR takes as input queries in the disjunctive normal form (DNF). Consequently, a failing query Q_0 consists of a disjunction of conjunctions of the form $Q_0 = C_1 \bigvee C_2 \bigvee \ldots \bigvee C_n$. In turn, each C_k is a conjunction of constraints imposed on (a subset of) the domain attributes:

$$C_k = Constr(A_{i_1}) \bigwedge Constr(A_{i_2}) \bigwedge \ldots \bigwedge Constr(A_{i_k}).$$

When the context is ambiguous, the notation $Constr_{C_k}(A_j)$ is used to denote the constraint imposed by the conjunction C_k on the attribute A_j.

```
Given:
  - a failing query $Q$ in Disjunctive Normal Form
  - a small, randomly-chosen subset $\mathcal{D}$ of the target database

$RelaxedQuery = \emptyset$
FOR EACH of $Q$'s failing conjunctions $C_k$ DO
  - Step 1: $Rules$ = ExtractDomainKnowledge($C_k$, $\mathcal{D}$)
  - Step 2: $Refiner$ = FindMostSimilar($C_k$, $Rules$)
  - Step 3: $RelaxedConjunction$ = Refine($C_k$, $Refiner$)
  - add $RelaxedConjunction$ to $RelaxedQuery$
```

Figure 1: LOQR successively relaxes each conjunction independently of the other ones.

Each constraint consists of a domain attribute, an operator, and one or several constants. For the discrete attributes, LOQR accepts constraints of the type $=$, $\neq$, $\in$, or $\notin$ (e.g., $Color = black$ or $Manufacturer \in \{Sony, HP\}$). For the continuous attributes, the constraints use the inequality operators $\leq$, $<$, $\geq$, or $>$ (e.g., $Price < 2000$).

For a DNF query to fail, each of its conjunctions C_k must fail (i.e., the query consists of *failing conjunctions* only); conversely, by successfully relaxing *any* of its failing conjunctions, one turns a failing query into a non-failing one. Based on this observation, LOQR successively relaxes the failing conjunctions independently of each other (see Figure 1); consequently, without any loss of generality, we focus here on the three steps used to relax a failing conjunction C_k: extracting the implicit domain knowledge (expressed as learned decision rules), finding the decision rule that is the most similar to C_k, and using this decision rule to actually relax C_k.

4.2 Step 1: Extracting domain knowledge

In this first step, LOQR uses a subset of the target database to uncover the implicit relationships that hold among the domain attributes. This is done by the following strategy: for each attribute A_j that appears in the failing conjunction C_k, LOQR uses the values of the other domain attributes to predict whether A_j's value satisfies $Constr_{C_k}(A_j)$.

As shown in Figure 2, LOQR starts with a randomly-chosen subset $\mathcal{D}$ of the target database and creates one additional dataset D_j for each attribute A_j that is constrained in C_k. Each dataset D_j is a copy of $\mathcal{D}$ that differs from the original only by the values of A_j: for each example in D_j, if the original value of A_j satisfies $Constr_{C_k}(A_j)$, LOQR sets A_j to YES; otherwise A_j is set to NO. For each D_j, the binary attribute A_j is designated as the class attribute.

After creating these additional datasets, LOQR applies C4.5-rules [24] to each of them, thus learning decision rules that, for each attribute A_j in C_k, predict whether A_j satisfies $Constr_{C_k}(A_j)$. In other words, these learned decision rules represent patterns that use the values of some domain attributes to predict whether a particular constraint in C_k is satisfied.

4.3 Step 2: Finding the "refiner statement"

After the learning step above, LOQR converts each learned decision rule into the equivalent existentially-quantified state-

[3] Note that this guarantee holds only if $\mathcal{D}$ is a subset of the target database. If these two datasets are disjoint (see our experimental setup), Q_R may fail, even though it is highly unlikely to do so.

```
ExtractDomainKnowledge (conjunction $C_k$, dataset $\mathcal{D}$)
- $Rules = \emptyset$
FOR EACH attribute $A_j$ that appears in $C_k$ DO
    - create the following binary classification dataset $D_j$:
        - FOR EACH example $ex \in \mathcal{D}$ DO
            - make a copy $ex'$ of $ex$
            - IF $ex'.A_j$ satisfies $Constr_{C_k}(A_j)$
                THEN set $ex'.A_j$ to "yes"
                ELSE set $ex'.A_j$ to "no"
            - add $ex'$ to $D_j$
        - designate $A_j$ as the (binary) class attribute of $D_j$
        - apply the C4.5-RULES algorithm to $D_j$
        - add these learned rules to $Rules$
- return $Rules$
```

Figure 2: *Step 1:* query-guided extraction of domain knowledge expressed as decision rules.

ment. This is done by simply replacing "$\Rightarrow$" by "$\bigwedge$" in each rule (remember than any decision rule "$a \bigwedge b \bigwedge c \Rightarrow d$" can be interpreted as the existential statement "there are examples in $\mathcal{D}$ such that $a \bigwedge b \bigwedge c \bigwedge d$").

Note that the resulting existential statements have the same syntax as the failing conjunctions; i.e., they both represent a conjunction of constraints on the domain attributes. Consequently, LOQR can detect the existential statement that is the most similar to C_k by performing an attribute-wise comparison between the constraints in C_k and those in each of the learned statements (i.e., LOQR finds C_k's "nearest neighbor" among the existential statements).

In order to evaluate the similarity between a conjunction C_k and a statement S, LOQR use the function

$$dist(C_k, S) = \sum_{A_j \in C_k \bigwedge A_j \in S} w_j \times Dist(C_k, S, A_j)$$

in which w_j denotes the user-provided (relative) weight of the attribute A_j. $Dist(C_k, S, A_j)$ is a measure of the similarity between the constraints imposed on A_j by C_k and S; it has the range $[0, 1]$ (the smaller the value, the more similar the constraints) and is defined as follows:

- if the attribute A_j does *not* appear in *both* C_k and S, then $Dist(C_k, S_i, A_j) = 1$; i.e., C_k and S are highly dissimilar with respect to A_j.

- if A_j takes discrete values, then $Dist(C_k, S_i, A_j) = \begin{cases} 0 & \text{if } Constr_{C_k}(A_j) \bigcap Constr_S(A_j) \neq \emptyset \\ 1 & \text{otherwise} \end{cases}$

  In other words, C_k and S are highly similar with respect to A_j if there is at least a value of A_j that is valid for both $Constr_{C_k}(A_j)$ and $Constr_S(A_j)$.

- if A_j takes continuous values, then $Dist(C_k, S_i, A_j) = \frac{|Value(Constr_{C_k}(A_j)) - Value(Constr_S(A_j))|}{Max_{A_j} - Min_{A_j}}$ where $Value()$ returns the constraint's numeric value, while Max_{A_j} and Min_{A_j} are the largest and the smallest value of A_j in $\mathcal{D}$, respectively. Intuitively, the smaller the relative difference between the values in the two constraints, the more similar the constraints.

```
Refine ( conjunction $C_k$, statement $S$)
 - $C_{Relax} = \emptyset$
 FOR EACH attribute $A_j$ that appears in both $C_k$ and $S$ DO
   IF $A_j$ is discrete THEN
    - $C_{Relax} = C_{Relax} \bigwedge (Constr_{C_k}(A_j) \bigcap Constr_S(A_j))$
   ELSE /*———— $A_j$ is continuous ————*/
    IF $Constr_{C_k}(A_j)$ and $Constr_S(A_j)$ are of "same type" THEN
     - $C_{Relax} = C_{Relax} \bigwedge Least(Constr_{C_k}(A_j), Constr_S(A_j))$
    ELSE
     - $C_{Relax} = C_{Relax} \bigwedge (Constr_{C_k}(A_j) \bigcap Constr_S(A_j))$
 - return $C_{Relax}$
```

Figure 3: *Step 3:* relaxing a failing conjunction.

4.4 Step 3: Refining the failing conjunction

After finding the statement S that is the most similar to the failing conjunction C_k, LOQR uses the domain knowledge synthesized by S to relax the constraints in C_k. As shown in Figure 3, the relaxation works as follows:

- the relaxed conjunction C_{Relax} includes only constraints on attributes that are present in *both* C_k *and* S;

- if A_j is discrete, C_{Relax} constrains its values to the ones common to both $Constr_{C_k}(A_j)$ and $Constr_S(A_j)$. If $Values(Constr_{C_k}(A_j)) \bigcap Values(Constr_S(A_j)) = \emptyset$, then C_{Relax} does not impose any constraint on A_j.

- if A_j is continuous, there are two possible scenarios:

  1. if $Constr_{C_k}(A_j)$ and $Constr_S(A_j)$ use the same type of inequality (e.g., one of $>$ or $\geq$), then C_j contains the *least constraining*[4] of $Constr_{C_k}(A_j)$ and $Constr_S(A_j)$. For example, if the constraints are $Price <$ \$1,000 and $Price \leq$ \$799, then C_{Relax} contains the former.
  2. if $Constr_{C_k}(A_j)$ and $Constr_S(A_j)$ use different types of inequalities (e.g., one of them uses $>$ or $\geq$, while the other one uses $<$ or $\leq$), then C_{Relax} contains the *intersection* of the two constraints. For example, if the constraints are $Price <$ \$1,000 and $Price \geq$ \$799, then C_{Relax} contains $Price \in$ [799, 1000); if the constraints are $Price \geq$ \$1,000 and $Price <$ \$799, their intersection is empty, which means that C_{Relax} imposes no constraint on $Price$.

5. EXPERIMENTAL RESULTS

In this section we begin with a brief overview of the five algorithms to be evaluated, followed by the description of the datasets, the experimental setup, and the actual results.

5.1 The Algorithms

We empirically compare the performance of LOQR with that of the following four algorithms: LOQR-50, LOQR-90, s-NN, and r-NN. The first two are variants of LOQR, while the other ones represent two baselines.

The baselines work as follows: for each failing conjunction C_k, they use the distance measure from Section 4.3 to find the example $Ex \in \mathcal{D}$ that is the most similar to C_k. Then

[4] For each continuous attribute, LOQR requires the user to specify whether *larger* or *smaller* values are more desirable. In our laptop scenario, the larger the hard disk, the better; conversely, the smaller the *Price* the better, too.

they use Ex to create a conjunction C'_k that has the same constraints as C_k, except that the original attribute values are replaced by the corresponding values from Ex. The difference between S-NN and R-NN is that the former simply returns C'_k as the relaxed query, while the latter uses C'_k to relax C_k as explained in Section 4.4.

LOQR-50 and LOQR-90 represent variants of LOQR that illustrate the trade-offs between the following strategies: "generate over-relaxed queries that are highly unlikely to fail, but return a (relatively) large number of tuples" *vs* "create under-relaxed queries that return fewer tuples, but are more likely to fail". Both algorithms assume that the user designates one of the attributes as being *the most relevant* (ideally, the constraint on this attribute should not be relaxed at all).

For each failing conjunction C_k, LOQR-50 and LOQR-90 run LOQR, which computes the relaxed conjunction C_{Relax}. After the user selects an attribute A_j from C_{Relax} as the *most relevant*, the algorithms

- apply Q_R to the dataset $\mathcal{D}$ and obtain the set $\mathcal{CT}$ of compliant tuples, which consists of the examples covered by the decision rule used to relax the failing query.

- determine the set $\mathcal{V}$ of all values taken by the attribute A_j over the dataset $\mathcal{CT}$.

- replace the value in $Constr_{C_k}(A_j)$ by "the most constraining" 50- or 90- percentile value, respectively.[5] For example, if $Constr_{C_k}(A_j)$ is $Price < \$2000$, LOQR-90 replaces \$2000 by a value $v \in V$ such that exactly 90% of the values in V are *worse* (i.e., *smaller*) than v. Similarly, if $Constr_{C_k}(A_j)$ is $CPU > 2.5\,GHz$, LOQR-90 replaces $2.5\,GHz$ by a value $v \in V$ such that exactly 90% of the values in V are *larger* then v.

For all five algorithms above, we used equal weights ($w_j = 1$) in the formula for $dist(C_k, S)$, which measures the similarity between two conjunctive formulas (see Section 4.3).

5.2 The Datasets and the Setup

We evaluate the algorithms above on six different datasets. The first one, LAPTOPS, is the original motivation for this work. It consists of 1257 laptop configurations extracted from `yahoo.com`; each laptop is described by five numeric attributes: price, CPU speed, RAM, HDD space, and weight. The other five domains are taken from the UC Irvine repository: breast cancer Wisconsin (BCW), low resolution spectrometer (LRS), Pima Indians diabetes (PIMA), water treatment plant (WATER), and waveform data generator (WAVE).

In order to evaluate the performance of the five algorithms above, we proceed as follows. Given a failing query Q and a dataset $\mathcal{D}$, each algorithm uses $\mathcal{D}$ to generate a relaxed query Q_R. In order to estimate its adequacy, Q_R is then evaluated on a *test set* that consists of all examples in the target database *except* the ones in $\mathcal{D}$. We have chosen to keep $\mathcal{D}$ and the test set *disjoint* (which may lead to the relaxed query failing on the test set) because in our motivating Web-based domains the owner of a database may be unwilling to provide the dataset $\mathcal{D}$. By keeping $\mathcal{D}$ and the test set disjoint, we can simulate (up to a certain level) the scenario in which the relaxation algorithm exploits an alternative information source that is somewhat representative of the data in the target database.

For each of the six domains, we have seven distinct failing queries. We also consider various sizes of the dataset $\mathcal{D}$: 50, 100, 150, ..., 350 examples; for each of these sizes, we create 100 arbitrary instances of $\mathcal{D}$, together with the 100 corresponding test sets. For each size of $\mathcal{D}$ and each of the seven failing queries, each query relaxation algorithms is run 100 times (once for each instance of $\mathcal{D}$); consequently, the results reported here are the average of these 700 runs.

5.3 The Results

In our experiments, we focus on two performance measures:

- *robustness*: what percentage of the failing queries are successfully relaxed (i.e., they don't fail anymore)?

- *coverage*: what percentage of the examples in the test set satisfy the relaxed query?

Figures 4 and 5 show the *robustness* and *coverage* results on the six evaluation domains. In terms of *robustness*, LOQR obtains by far the best results: independently of the size of $\mathcal{D}$, on all six domains LOQR's robustness is above 90%; in fact, most of the robustness results are close or above 99% (i.e., about 99% of the queries are successfully relaxed).

In contrast, the two baselines, S-NN and R-NN, display extremely poor robustness: on LRS, PIMA, WATER, and WAVE their robustness is below 10%. The baselines' best results are obtained on small-sized $\mathcal{D}$ (i.e., $Size(\mathcal{D}) = 50$), where the scarcity of the training data makes it unlikely to find a domain example that is highly-similar to the failing query; in turn this leads to an over-relaxation of the query, which improves the robustness. However, as $Size(\mathcal{D})$ increases, the performance of the two baselines degrades rapidly.

The second measure of interest, *coverage*, must be considered in conjunction with the robustness results: even though we are interested in low-coverage results[6], a low-coverage, non-robust algorithm is of little practical importance. Consequently, the low-coverage results of the two baselines (see Figure 5) must be put in perspective: after all, the vast majority of the queries relaxed by these algorithms still fail.

LOQR scores less spectacularly in terms of *coverage*: when learning from datasets of 350 examples, its coverage on the six domains is between 2% and 19%. However, by trading-off robustness for coverage, LOQR-90 obtains excellent overall results: when using 350 training examples, on all domains but BCW, LOQR-90 reaches robustness levels between 69% and 98%, while also keeping the coverage under 5%.

Last but not least, we are also interested in the amount of time spent relaxing a query: given that each query is processed online, it is crucial that LOQR quickly relaxes the incoming queries. In Table 3, we show the CPU time (in seconds) that is spent refining the queries in the six application domains. Our results show that LOQR is extremely fast:

[5] Obviously, this strategy applies only to the continuous attributes. For the discrete ones, the user is asked to select a correspondingly small subset of the most desirable values.

[6] Our motivation comes from Web-based information sources, for which high-coverage queries may be unacceptable because of (1) the database's owners unwillingness to return large chunks of the data; (2) the bandwidth problems associated with transmitting huge datasets over the Internet; (3) the fee that one may have to pay for each returned tuple. Consequently, we are interested in query relaxations that return only a few, highly-relevant tuples.

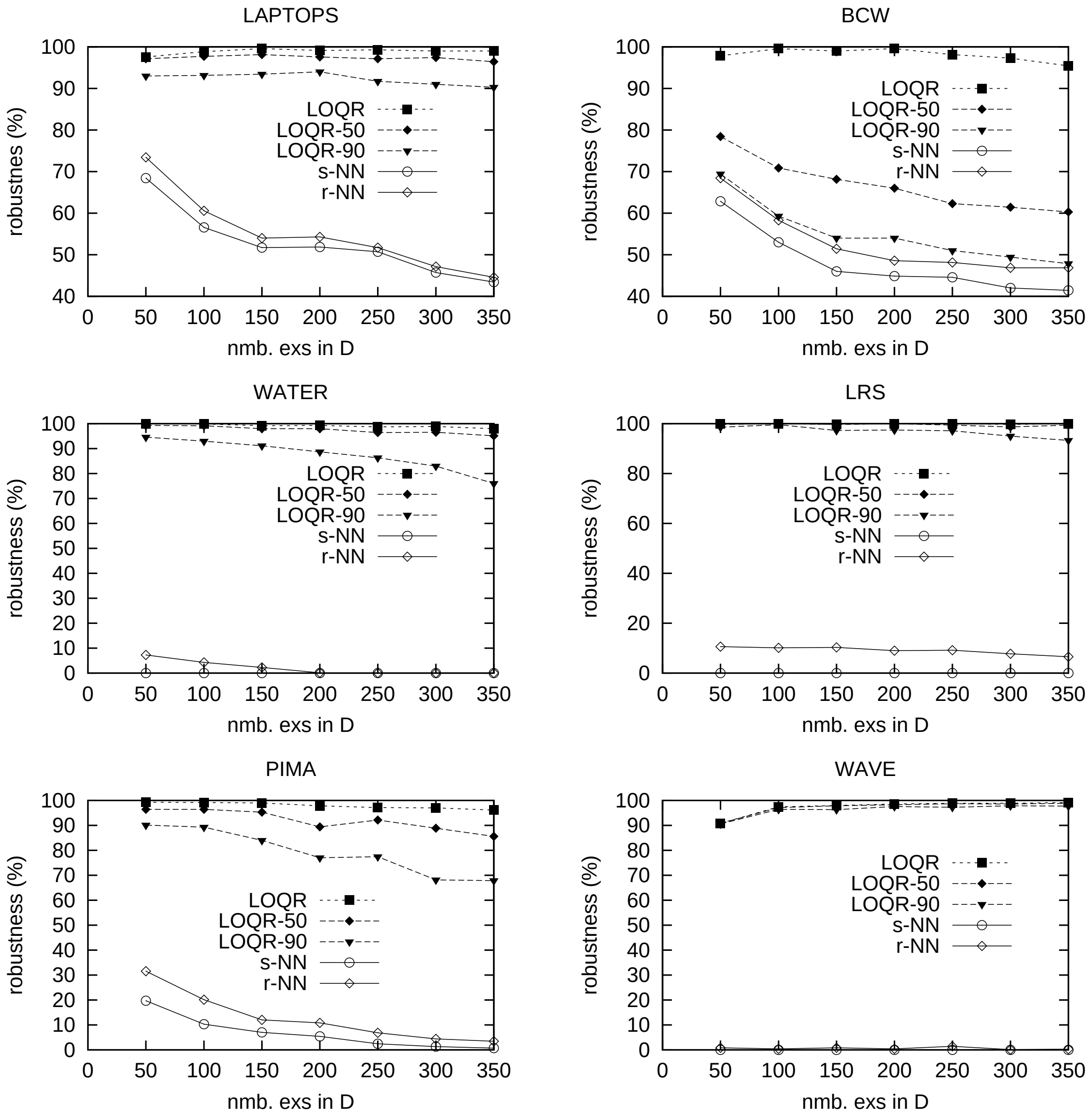

Figure 4: Robustness: what percentage of the relaxed queries are not failing?

queries that consist of at most 21 attributes are processed under 1.40 seconds; for queries that consist of 93 attributes, it may take up to 30 seconds. To put things into perspective, a human cannot even read and comprehend a 93-attribute query in this amount of time; in fact, it is highly unlikely that humans are even willing to write 93-attribute queries.

Note that the running time is influenced both by the size of the dataset $\mathcal{D}$ and the number of attributes in the query. The former relationship is straightforward: the larger the dataset $\mathcal{D}$, the longer it takes to learn the decision rules. The latter is more subtle: as LOQR creates a new dataset for each attribute in the query, it follows that the more attributes in the query, the longer it takes to process the query. However, not all constraints are equally time consuming; for example, if an attribute A is constrained to take a value that is out of the range of values encountered in $\mathcal{D}$, then it is superfluous to learn from the corresponding dataset, which consists solely of "negative examples" (the constraint on A is not satisfied in *any* of the examples from $\mathcal{D}$).

6. DISCUSSION

Before concluding this paper, we must discuss two important design choices that heavily influence LOQR's performance: the *online* and the *query-driven* nature of the learning process. The former refers to the fact that the learning step is performed at run-time, for each failing query. The latter specifies that the learning task is designed as a binary classification problem in which the class attribute represents the boolean value "does A_j's value satisfy the constraints imposed onto it by the failing query?"

6.1 Online *vs* offline learning

In order to illustrate the advantages of our online approach, we re-use the experimental setup above to compare

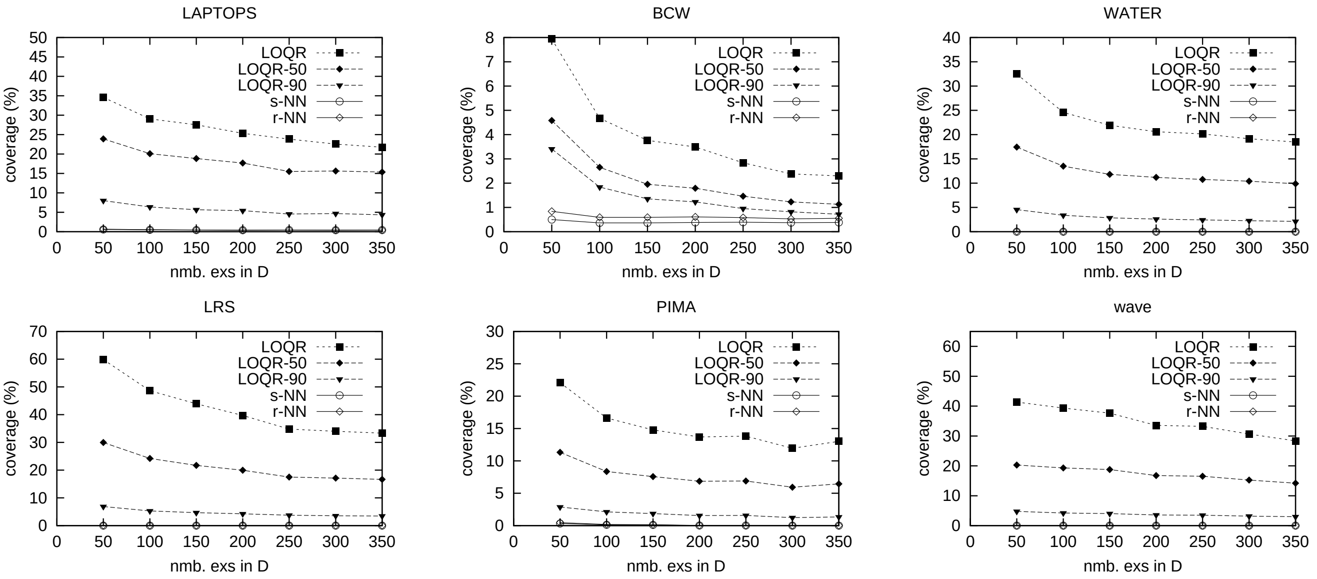

Figure 5: Coverage: what percentage of the examples in the test set match the relaxed query?

Dataset	Attributes per query	CPU time (seconds per query)	
		$\|\mathcal{D}\| = 50$	$\|\mathcal{D}\| = 350$
LAPTOPS	5	0.06	0.15
PIMA	8	0.15	0.47
BCW	10	0.14	0.37
WAVE	21	0.46	1.39
WATER	38	0.87	4.40
LRS	93	4.32	30.71

Table 3: Running times for LOQR, averaged over the runs on each of the 100 instances of $\mathcal{D}$ that are created for 50 and 350 examples, respectively.

LOQR with an offline variant, OFF-k. The only difference between the two algorithms is that OFF-k performs the learning step only once, independently of the constraints that appear in the failing queries. Similarly to LOQR, OFF-k also tries to predict an attribute's value from those of the other attributes; however, because it does not have access to the constraints in the failing query, OFF-k proceeds as follows: for discrete attributes, it learns to predict each discrete value from the values of the other attributes; for continuous attributes, it discretizes the attribute's range of values in $\mathcal{D}$ so that it obtains a number of k intervals that have equal size.

In this empirical comparison, we use two offline versions (i.e., $k = 2$ and $k = 3$) for both LOQR and LOQR-90. Figures 6 and 7 show the *robustness* results[7] for LOQR and LOQR-90, respectively (OFF-2 and OFF-3 denote the two offline versions of each algorithm). The graphs show that both LOQR and LOQR-90 clearly outperform their offline variants,[8] thus demonstrating the superiority of the online, query-guided approach.

6.2 Query-driven learning

As we have already seen, guiding the learning process by the constraints that appear in the failing query leads to dramatic robustness improvements. However, the query-driven approach used by LOQR is by no means the only possible approach. In fact, we can distinguish four main scenarios:

- *no constraints:* this approach corresponds to *offline* learning, in which none of the query constraints are used to guide the learning process.
- *class-attribute constraints:* this is the approach used by LOQR, in which we create a dataset $\mathcal{D}$ for each attribute in the query. Each such dataset uses *exactly one* of the failing query's constraints to guide the learning; more precisely, one of the constrained attributes becomes the *designated class-attribute* that takes two discrete values (i.e., does/does-not satisfy the constraint).
- *set of hard constraints:* this scenario represents a straightforward generalization of the previous one. If the user specifies a subset of M of the failing query's constraints that *must* be satisfied (i.e., *hard constraints*), one can then replace the corresponding M attributes by a single discrete one that represents the conjunction of the M hard constraints and then apply LOQR.
- *all constraints:* this final scenario correspond to *simultaneously* replacing the original values of *all* the attributes

[7] Because of space limitations, we do not show the *coverage* results. However, they can be summarized as follows: on all six domains, the coverage of the online and offline algorithms are extremely close. On WATER and LRS, both LOQR and LOQR-90 outperform their offline counterparts, while on BCW and WAVE the performance is virtually the same.

[8] Figures 6 and 7 also show that OFF-3 does not always outperform OFF-2. This is because the discretization of the continuous attributes is made independently of the values from the failing query: as the discretization takes place offline, the values that appear in query's constraints may lay anywhere within a discretized interval. Consequently, the “purity” of the discretized class that includes the query value may vary wildly (e.g., almost all values in that interval may or may not satisfy the constraint from the query), which - in turn - dramatically affects the quality of the relaxed query.

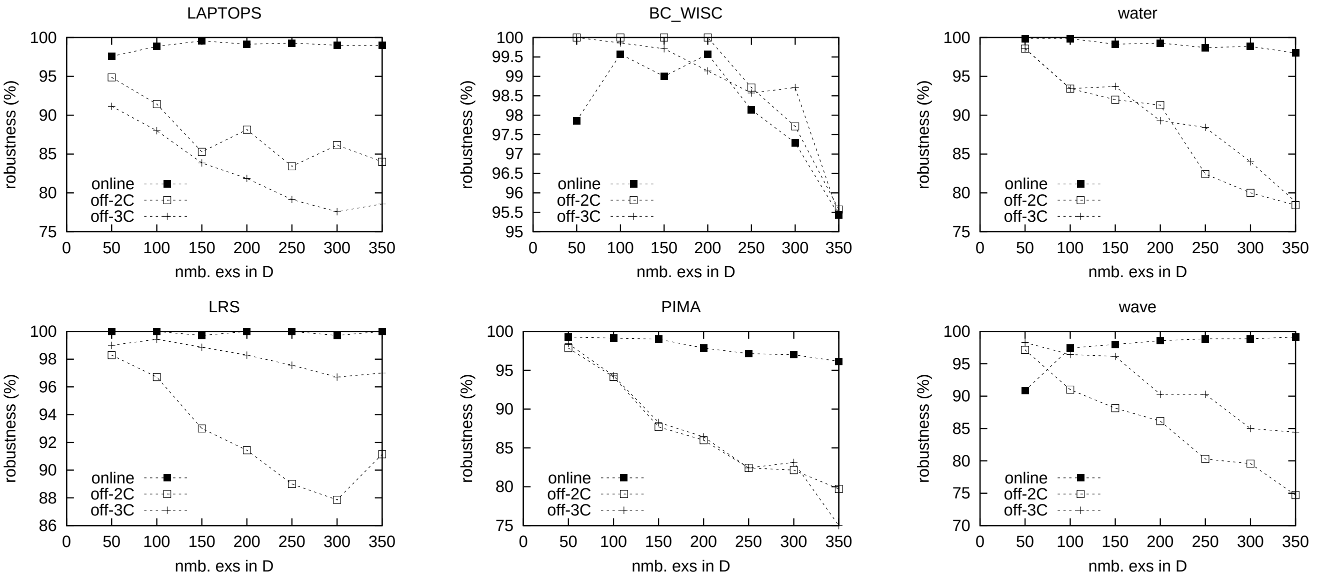

Figure 6: Online vs Offline: robustness results for LOQR.

in the query with the corresponding boolean value (i.e., does/does-not satisfy the constraints in the query).

In this paper we have analyzed the first two of the scenarios above. The third one represents a straightforward extension of LOQR that was not addressed here because of space limitations. Finally, the last scenario has both advantages and disadvantages over LOQR. On one hand, by simultaneously replacing the values of all attributes with the corresponding boolean values, the "*all constraints*" scenario creates a single dataset $\mathcal{D}$ instead that one per attribute; in turn this leads to a considerable gain in terms of processing speed. On the other hand, in the "*all constraints*" scenario there is only one way to relax a query, namely by dropping constraints. In contrast, LOQR permits both constraint dropping and constraint relaxation (i.e., replacing the original value by a less constraining one), thus providing a significantly more flexible solution to the query relaxation problem.

7. CONCLUSIONS & FUTURE WORK

In this paper we have introduced a novel, data-driven approach to query relaxation. Our algorithm, LOQR, performs online, query-driven learning from a small subset of the target database. The learned information is then used to relax the constraints in the failing query. We have shown empirically that LOQR is a fast algorithm that successfully relaxes the vast majority of the failing queries.

We intend to continue our work on query relaxation along several directions. First, we plan to extend our data-driven approach by also exploiting user preferences that are learned as the system is in use. Second, we are interested in a query visualization algorithm that would allow a user to explore the trade-offs between the various possible query relaxations. Finally, we plan to integrate the query visualization module in a mixed initiative system in which the user interacts with the query relaxation algorithm by expressing various preferences over the domain attributes.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

We would like to thank Melinda Gervasio, Karen Myers, Maria Muslea, and Tomas Uribe for their helpful comments on this paper. We also thank Steven Minton and Fetch Technologies, Inc. for providing us with the Laptops dataset.

9. REFERENCES

[1] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *International Conference on Extending Database Technology EDBT*, 2002.

[2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[3] K. Chakrabarti, M. Ortega, S. Mehrotra, and K. Porkaew. Evaluating refined queries in top-k retrieval systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(5), 2003.

[4] S. Chaudhuri. Generalization and a framework for query modification. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*, pages 138–145. IEEE Computer Society, 1990.

[5] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proceedings of the Conference on Very Large Databases*, pages 397–410, 1999.

[6] W. Chu, Q. Chen, and A. Huang. Query answering via cooperative data inference. *Journal of Intelligent Information Systems*, 3(1):57–87, 1994.

[7] W. Chu, K. Chiang, C.-C. Hsu, and H. Yau. An error-based conceptual clustering method for providing approximate query answers. *Communications of* ACM, 39(12):216–230, 1996.

[8] W. Chu, R. C. Lee, and Q. Chen. Using type interfaces and induced rules to provide intentional

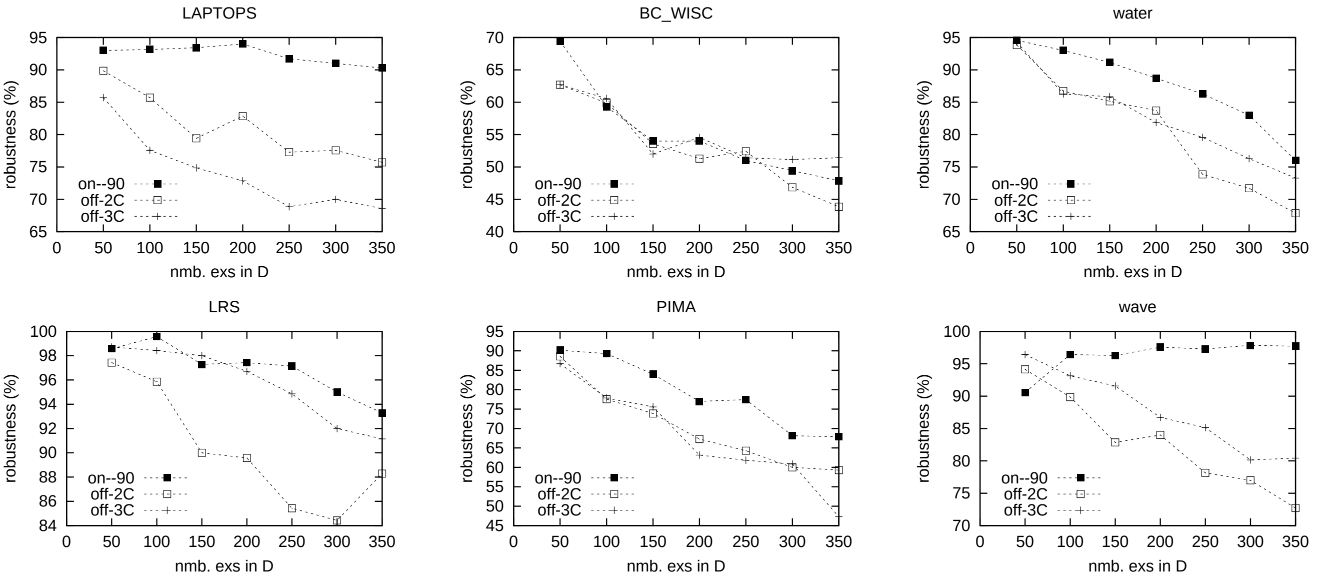

Figure 7: Online vs Offline: robustness results for LOQR and LOQR-90.

answers. In *Proceedings of the Seventh International Conference on Data Engineering*, 1991.

[9] W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson. Cobase: A scalable and extensible cooperative information system. *Journal of Intelligence Information Systems*, 6(2/3):223–59, 1996.

[10] F. Corella, S. J. Kaplan, G. Wiederhold, and L. Yesil. Cooperative responses to boolean queries. In *Proceedings of the 1st International Conference on Data Engineering*, pages 77–85, 1984.

[11] N. Fuhr and K. Grosjohann. XIRQL: A query language for information retrieval in XML documents. In *Research and Development in Information Retrieval*, pages 172–180, 2001.

[12] T. Gaasterland. Cooperative answering through controlled query relaxation. *IEEE Expert*, 12(5):48–59, 1997.

[13] A. Gal. *Cooperative responses in deductive databases.* PhD thesis, Department of Computer Science, University of Maryland, College Park, 1988.

[14] P. Godfrey. Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems*, 6(2):95–149, 1997.

[15] J. M. Janas. Towards more informative user interfaces. In A. L. Furtado and H. L. Morgan, editors, *Fifth International Conference on Very Large Data Bases, October 3-5, 1979, Rio de Janeiro, Brazil, Proceedings*, pages 17–23. IEEE Computer Society, 1979.

[16] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 40–51, 2002.

[17] M. Kao, N. Cercone, and W.-S. Luk. Providing quality responses with natural language interfaces: The null value problem. *IEEE Transactions on Software Engineering*, 14(7):959–984, 1988.

[18] S. Kaplan. Cooperative aspects of database interactions. *Artificial Intelligence*, 19(2):165–87, 1982.

[19] D. A. Keim and H. Kriegel. VisDB: Database exploration using multidimensional visualization. *Computer Graphics and Applications*, 1994.

[20] D. Lee. *Query Relaxation for XML Model.* PhD thesis, Department of Computer Science, University of California Los Angeles, 2002.

[21] M. Merzbacher and W. Chu. Pattern-based clustering for database attribute values. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases*, 1993.

[22] A. Motro. SEAVE: a mechanism for verifying user presupositions in query system. ACM *Transactions on Information Systems*, 4(4):312–330, 1986.

[23] A. Motro. Flex: A tolerant and cooperative user interface databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):231–246, 1990.

[24] R. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann Publishers, 1993.

[25] A. Theobald and G. Weikum. Adding relevance to XML. *Lecture Notes in Computer Science*, 1997:105–131, 2001.