A Framework for Query Refinement with User Feedback

Md. Saiful Islam[a,*], Chengfei Liu[a], Rui Zhou[a]

[a] *Faculty of Information and Communication Technologies, Swinburne University of Technology, VIC 3122, Australia*

Abstract

SQL queries in the existing relational data model implement the binary satisfaction of tuples. That is, a data tuple is filtered out from the result set if it does not satisfy the constraints expressed in the predicates of the user submitted query. Posing appropriate queries for ordinary users is very difficult in the first place if they lack knowledge of the underlying dataset. Therefore, imprecise queries are commonplace for many users. In connection with this, this paper presents a framework for capturing user intent through feedback for refining the initial imprecise queries that can fulfill the users' information needs. The feedback in our framework consists of both unexpected tuples currently present in the query output and expected tuples that are missing from the query output. We show that our framework does not require users to provide the complete set of feedback tuples because only a subset of this feedback can suffice. We provide the point domination theory to complement the other members of feedback. We also provide algorithms to handle both soft and hard requirements for the refinement of initial imprecise queries. Experimental results suggest that our approach is promising compared to the decision tree based query refinement approach.

Keywords: Imprecise Query, User Feedback, Point Domination Theory, Query Intent, Query Refinement

1. Introduction

Relational Data Model is one of the widely used data models for storing and retrieving information. The underlying query engine accepts requests from users through SQL, by which the non-answers are filtered out from the result set. To filter the non-answers from answers, users express their constraints in the form of predicates. Predicates are generally grouped in conjunctive or disjunctive normal form. For ordinary database users, posing appropriate constraints in the predicates without having a complete knowledge of the underlying dataset is a tedious job (Nandi and Jagadish, 2011). In most cases, users go for a number of trials before getting the ultimate or precise query. In the worst case, they reach an unsatisfactory one. To alleviate this problem in *cooperative database systems*, a dialog is established between a user and the system to understand the intent of the user (Sultana et al., 2009) by returning additional information as a response to a query (not only the answer set) in this regard (Motro, 1994). Recently, explanation of query results and their provenance information such as *why* and *how* a particular piece of information arrived (Cheney et al., 2009), (Green et al., 2007), (Glavic and Alonso, 2009) or even was missed (Huang et al., 2008), (Herschel and Hernández, 2010), (Chapman and Jagadish, 2009), (Tran and Chan, 2010) are suggested to be returned to the user in this endeavor. Supporting imprecise queries for inexperienced users (even for advanced users for explorative data analysis) could be thought of as a natural extension in cooperative database systems (Motro, 1988), (Nambiar and Kambhampati, 2003).

In connection with the execution of imprecise queries the following four situations may occur: (1) result set includes no unexpected tuples and misses no expected tuples, (2) result set includes some unexpected tuples and misses no expected tuples, (3) result set includes no unexpected tuples but misses some expected tuples, and (4) result set includes some unexpected tuples and similarly misses some expected tuples. Situation one is the perfect scenario where the user is happy about what she is doing with the data. That is, the query she developed matches perfectly with what she needs. Situation two and three matches a part of her needs but situation four disappoints her completely. In this regard, the user may wish to have explanation by asking particularly why a certain tuple is in the result set and why a certain tuple is not in the result set (Islam et al., 2012).

While *why* questions can be addressed by applying established why, where and how provenance techniques (Cheney et al., 2009), *why-not* questions have received very little attention. Three different models exist for explaining why-not questions: (1) Huang et. al. (Huang et al., 2008) and Herschel et. al. (Herschel and Hernández, 2010) explain a missing tuple by allowing modifications to the database so that the missing tuple appears in the query result with respect to the modified database. The intuition of this model is to explain in terms of how to modify some of the untrusted data in order to produce the missing tuple. However, this model may not be applicable in applications where all the data stored are trusted; (2) Chapman and Jagadish (Chapman and Jagadish, 2009) model explanation by identifying the operator(s) that filters out the tuple from the result set; and (3) Recently, Tran and Chan (Tran and Chan, 2010) model why-not explanation by refining the original query to include the missing expected tuples in the result. Also, Liu *et. al.* (Liu et al., 2010) collect *false positives* identified

[*] Corresponding author

Email addresses: mdsaifulislam@swin.edu.au (Md. Saiful Islam), cliu@swin.edu.au (Chengfei Liu), rzhou@swin.edu.au (Rui Zhou)

Table 1: Stores information about prospective PhD candidates

Sname	Uname	Npub	HD
John	Stanford	1	Bachelor
Craig	Berkley	1	Bachelor
Peter	CA Irvine	2	Master
Luke	Stanford	4	Incomplete
Noah	ANU	2	Master
Kevin	Melbourne	1	Master
Susan	Stanford	1	Master

Table 2: Stores information about candidates' graduating university

Uname	Rank
Stanford	1
ANU	3
Berkley	3
CA Irvine	4
Melbourne	5

by the users as a *feedback* to modify initial rules in information extraction settings. Both *why* and *why-not* questions are treated separately in the above models. We believe that a more helpful explanation should be one that can model both *why* and *why-not* questions asked at the same time for the same query. The following motivating examples illustrate the need for refining queries when query result includes unexpected tuples as well as misses some expected tuples.

Example 1. *Consider the two data tables that store information about the prospective PhD candidates as well as their graduating universities as shown in Table 1 and Table 2. Now, assume that Professor Michael wants to select two candidates from them and wishes to execute the following query against T_1 and T_2:* ***"find the name of all students who receive master degrees from universities with rank at least 4 and have at least 2 publications"****. The equivalent SQL statement of the query given above is as follows:*

SELECT Sname FROM T_1, T_2
WHERE HD='Master' AND Rank≤4 AND
Npub≥2 AND T_1.Uname=T_2.Uname;

The result set of the query consists of Peter and Noah as shown in Table 3. Unfortunately, Michael is expecting John rather than Peter in the result set and wants to have an explanation for this in connection with his imprecise query. One possible solution could be presenting provenance information of each tuple for his query (Cheney et al., 2009) as shown in Table 4 and could be thought of as the cooperative behavior of the database systems. Though the provenance information of Peter conveys much more detail, it is insufficient for Michael as he needs to examine it with great care. Besides of that Michael wants to refine his initial query to include John instead of Peter in the result set where John and Peter are expected and unexpected examples of tuples.

Table 3: Result set R_1

Sname
Peter
Noah

Table 4: Result set R_2 (enhanced query output)

Sname	HD	Npub	Rank
Peter	Master	2	4
Noah	Master	2	3

Example 2. *Consider a selection query given below which is issued by a user to a publication database and the corresponding result set as shown in Table 5 and Table 6. Assume that the user has a different image of the query output in her mind (not the one shown in Table 6) and the result set does not match her expectation. She may then ask "How can I exclude P1, P3 and P10 from my query output? How can I include P11 and P12 into my query output".*

SELECT PubID FROM Publication
WHERE CitationCnt ≥ 80 AND
PubYear ≥ 1986;

A promising approach to mitigate the above problem is to enable the user to submit an imprecise query and communicate the system by providing feedback as shown in Fig. 1. After collecting feedback on current query output, the system then needs to discover the query intent of the user. Finally, the system should refine the original query in a way so that the refined query better fulfills the user's information need. This paradigm of querying can also be both incremental and iterative. That is, the query constantly evolves during the query session and and the user goes through the *intent* → *query* → *execution* → *result* process many times (Nandi and Jagadish, 2011). In this paper, we show how to modify the initial query after receiving the feedback (unexpected and expected tuples) from the user. Once the feedback is given by the user, we first analyze the feedback to discover the query intent of the user by the point domination theory. Then, we show how to modify the initial query to include expected tuples and/or exclude unexpected tuples to/from the new query output.

The rest of the paper is organized as follows: Section 2 formally defines the problem targeted in this paper, challenges and our contributions; Section 3 describes how we model user feedback; Section 4 presents our query refinement algorithms; Sec-

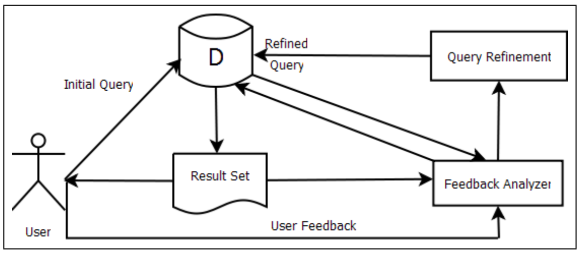


Figure 1: Query refinement with user feedback

Table 5: Publication database

PubID	CitationCnt	PubYear
$P1$	96	1989
$P2$	128	1986
$P3$	100	1989
$P4$	90	1993
$P5$	148	1986
$P6$	148	1990
$P7$	81	1996
$P8$	103	1986
$P9$	82	1994
$P10$	117	1987
$P11$	60	1995
$P12$	72	1996
$P13$	64	1996
$P14$	67	1995

Table 6: Result set (enhanced Query output)

PubID	CitationCnt	PubYear
$P1$	96	1989
$P2$	128	1986
$P3$	100	1989
$P4$	90	1993
$P5$	148	1986
$P6$	148	1990
$P7$	81	1996
$P8$	103	1986
$P9$	82	1994
$P10$	117	1987

tion 5 describes decision tree based query refinement; Section 6 presents experimental results; Section 7 describes related works and finally, Section 8 concludes our paper.

2. Problem Statement, Challenges and Contributions

We refer to queries that require the conditions to be slightly adjusted as imprecise queries and those requiring no adjustment as precise queries. Supporting imprecise queries over databases necessitates a system that collects feedback from the user to understand the submitted query intent and returns a new query to encounter both unexpected and expected tuples.

2.1. Problem Definition

Let Q be an imprecise query, D be the universe of discourse for Q and R be the set of resultant tuples. We use $Q(D)$ to denote all tuples that are satisfied by the predicates given in Q (and $R = Q(D)$) and $Q'(D)$ to denote all tuples that are not satisfied. We use R and $Q(D)$ alternatively in this paper. As Q is an imprecise query, $Q(D)$ may include some unexpected tuples as well as miss some expected tuples (as explained in Section 1). Let U be the set of unexpected tuples and E be the set of expected tuples. Therefore, $U \subseteq Q(D)$ and $E \subseteq Q'(D)$. The other tuples in $Q(D)$ except U are called truly positive tuples (TP). Similarly, tuples in $Q'(D)$ except E are called truly negative tuples (TN). We also call unexpected tuples *false positive* tuples (FP) and expected tuples *false negative* tuples (FN). The taxonomy of tuples in D with respect to Q is then shown in Fig. 2. Here, we adapt the original definitions of TP, FP, TN, and FN from information retrieval (IR) field (Mitchell, 1997) considering the imprecise user query Q as a tuple classifier.

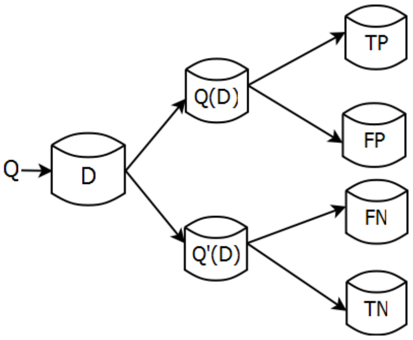


Figure 2: Taxonomy of tuples in D with respect to Q

We assume that both U and E is given by the user as feedback. Now, after getting U and E as feedback from the user, the system then makes necessary adjustment to Q to return a new query Q^* that includes as many as possible expected tuples as well as excludes as many as possible unexpected tuples. That is, the goal is to refine the initial query Q through query condition relaxation/restriction based on feedback to adapt better to the users information need. Each selection predicate c_i="a_i op v" in Q is treated as candidate for relaxation/restriction and by its relaxation/restriction we mean replacing v by a new value v^* if a_i is numeric. For categorical attributes, relaxation/restriction is performed by set operators UNION and EXCEPT. That is, we restrict and/or relax one or more c_i to refine the initial query Q to get the refined query Q^* to minimize the number of unexpected tuples in U and maximize the number of expected tuples from E in the result set.

2.2. Challenges

The difficulty of answering imprecise queries through query condition relaxation and/or restriction via user feedback is twofold.

(a) Our first challenge is modeling user feedback. Requiring a user to provide a complete set of feedback tuples is burdensome as she bears little understanding of the underlying dataset. Even for unexpected tuples clicking all of them in the result set is a tedious job and time-consuming too. The user may also be interested in defining the preferred values in a few dimensions instead of providing the complete tuple set. Therefore, an automatic approach for modeling and capturing the user intent is needed to complement the feedback.

(b) Our second challenge is modeling the query refinement. Once we get the feedback and capture the user intent we need to select the subset of query conditions for restriction and/or relaxation through which query refinement is achieved. But selection of subset of query conditions for relaxation or restriction is an NP-complete problem (see Section 4.1 for more detail). Therefore, we need good approximation algorithms that can run in polynomial time and still fulfill users' information needs.

2.3. Contributions

The main contributions of this paper are listed below:

(a) Our first contribution is modeling the user feedback and capturing the user intent. We present the point domination theory in this paper for complementing the feedback and capturing the user intent. We show how we can resolve user conflicts in defining the feedback and the query intent. For incomplete feedback, we also present a strategy for understanding the users' information needs.

(b) Our second contribution is the development of greedy-based approximation algorithms for the selection of subset of query conditions through which query refinement is achieved. We show how we can achieve query refinement when the selection predicates are overlapped for expected and unexpected tuples.

(c) The final contribution of our work is a thorough experimentation of the proposed algorithms, compared with the decision tree based query refinement.

3. User Feedback

In our framework, feedback is considered as annotations that a user provides to comment on query results, with the intention of informing the system of her information need and thereby, refining the submitted query to improve the quality of the refined query results. But annotating each tuple in the result set for the user is a very time-consuming job specifically for huge database and huge results. Moreover, providing complete feedback on false negatives (expected information currently not covered in the current result) is burdensome. Therefore, we allow the user to provide both concrete examples (tuples are marked explicitly by the user) and virtual tuples (an indication of the desired data tuples) in our framework. We argue that it is not necessary for the user to provide complete feedback but only for a subset. To find other members of the user feedback we can rely on the point domination theory proposed in this paper.

3.1. Point Domination Theory

Let G be the set of atomic predicate preference g_i, i.e., $G = \{g_1, g_2, ..., g_n\}$. We assume that each g_i is equally important to the user. We denote by $t \succ_{g_i} t'$ and $t \succeq_{g_i} t'$ the statements "tuple t satisfies preference g_i better than t'" and "tuple t satisfies preference g_i no worse than t'" respectively. If we plot the predicate preferences g_i in an n-dimensional space, each tuple will be an n-dimensional point. Then, we say a point t is no worse than another point t' in terms of G iff $\forall i \in [1, n]$, $t \succeq_{g_i} t'$ and point t dominates point t' iff $\forall i \in [1, n]$, $t \succeq_{g_i} t'$ and $\exists k \in [1, n]$, $t \succ_{g_k} t'$. The definition given above is also known as weak-pareto-dominance (Börzsönyi et al., 2001; Voorneveld, 2003).

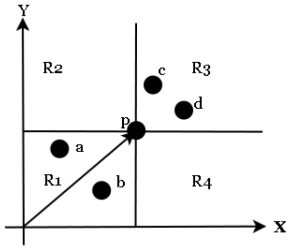


Figure 3: Point domination

We denote by $t \succeq t'$ and $t \succ t'$ the statements "tuple t is no worse than tuple t'" and "tuple t dominates tuple t'", respectively.

To make the above clear, consider there are two predicates we wish to consider (e.g., $att_1 \leq const_1$ and $att_2 \leq const_2$) and place these two predicates in x and y directions in a two-dimensional space as shown in Fig. 3. Also assume that a smaller value is preferred in each dimension (which is obvious from the given predicates). Consider there are five points 'a', 'b', 'p', 'c' and 'd' in this space. Now, both 'a' and 'b' dominate 'p' as both 'a' and 'b' are no worse than 'p' and satisfy at least one predicate better than 'p'. Similarly, 'p' dominates both 'c' and 'd'. If 'p' is our reference point, then 'p' divides the entire space into four regions as shown in Fig. 3. In general, any point from region R_1 is no worse than 'p' or dominates 'p'. Similarly, 'p' dominates any point from region R_3.

The above can also be generalized for predicates involving an arbitrary operator (i.e., '<', '<=', '>', and '>='). For operators '<' and '<=', the goal is to minimize the corresponding attribute value (i.e., a smaller value is preferred) and for operators '>' and '>=', the goal is to maximize the corresponding attribute value (i.e., a larger value is preferred). The point domination can also be generalized for predicates involving categorical attributes. However, if two tuples have different values for a categorical attribute, then they do not dominate each other in that dimension unless one of them satisfies the predicate or a partial order exists for it (Wong et al., 2008).

Example 3. *Consider the data given in Example 2. Now, let the user preference consists of the attributes 'CitationCnt' and 'PubYear'. Also assume that larger values are preferred for these two attributes. Then, we see that tuple $P6$ is no worse than tuple $P5$ in terms of 'CitationCnt' and $P6$ is better than $P5$ in terms of 'PubYear'. Therefore, we see that $P6$ dominates $P5$.*

3.2. Types of User Feedback

The query refinement problem targeted in this paper requires users to provide the unexpected and expected tuples as feedback. The unexpected tuple(s) are given by users as *why* question(s). On the other hand, the expected tuples are given by users as *why-not* question(s). The different types of this feedback are shown in Fig. 4.

3.2.1. Explicit Feedback

We define the feedback explicitly identified by the user as the *explicit feedback*. Explicit feedback can be of two types:

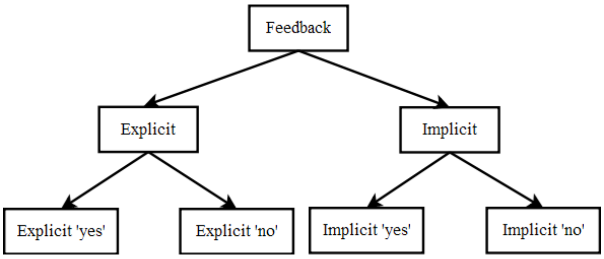


Figure 4: Different types of feedback

(1) explicit ‘yes’ tuples and (2) explicit ‘no’ tuples.

Definition 1. *(explicit ‘yes’ tuples)*
Data tuples given explicitly by the user as expected tuples are called explicit ‘yes’ tuples. Explicit ‘yes’ tuples are part of the non-answers, $Q'(D)$. That is, explicit ‘yes’ tuples are not included in the current result set.

Example 4. *(explicit ‘yes’ tuples)*
Consider the data given in Example 2. We see that tuples P11 and P12 are identified explicitly by the user as the expected ‘yes’ tuples. These tuples are not part of the current query output.

Definition 2. *(explicit ‘no’ tuples)*
Data tuples given explicitly by the user as unexpected are called explicit ‘no’ tuples. Explicit ‘no’ tuples are part of the answers, $Q(D)$. That is, explicit ‘no’ tuples are included in the current result set.

Example 5. *(explicit ‘no’ tuples)*
Consider the data given in Example 2. We see that tuples P1, P3, and P10 are identified explicitly by the user as the unexpected tuples. These tuples are part of the current query output.

3.2.2. Implicit Feedback

According to our point domination theory, any tuple $t' \in Q(D)$ dominated by any tuple $t \in U$ should be unexpected too. Similarly, any tuple $t' \in Q'(D)$ that is no worse than any tuple $t \in E$ should be expected as well. We term such t' as *implicit feedback*. Similar to the explicit feedback, implicit feedback can be of two types: (1) implicit ‘yes’ tuples and (2) implicit ‘no’ tuples.

Definition 3. *(implicit ‘yes’ tuples)*
Any tuple from non-answers that dominates one or more explicit yes tuples is called implicit yes tuple. That is, implicit yes tuples are no worse than explicit yes tuples. Similar to the explicit ‘yes tuples, implicit ‘yes’ tuples are not included in the current result set.

Example 6. *(implicit ‘yes’ tuples)*
Consider the data given in Example 2. We see that tuple P13 is no worse than tuple P11 according to the point domination theory. Therefore, if P11 is expected in the query output, P13 should be expected too.

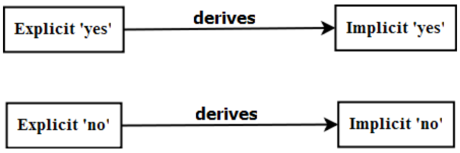


Figure 5: Feedback derivation

Algorithm 1 Computing Implicit ‘no’ Tuples

```
1: $U^+ \leftarrow U$;
2: for each $t \in U$ do
3:     if $\exists t' \in Q(D) \setminus U$ such that $t \geq_G t'$ then
4:         Add $t'$ to $U^+$;
5:     end if
6: end for
```

Definition 4. *(implicit ‘no’ tuples)*
Any tuple from answers that is no better than explicit no tuples is called an implicit ‘no’ tuple. Similar to the explicit ‘no’ tuples, implicit ‘no’ tuples are part of the current result set.

Example 7. *(implicit ‘no’ tuples)*
Consider the data given in Example 2. We see that tuple P8 is no better than tuple P10 according to the point domination theory. Though P8 is part of the current result set, it should be excluded too.

We compute the implicit feedback from the user given explicit feedback by applying the point domination theory. More specifically, explicit ‘yes’ tuples are used to derive implicit ‘yes’ tuples and explicit ‘no’ tuples are used to derive implicit ‘no’ tuples as shown in Fig. 5. We use implicit feedback to complement the user identified incomplete feedback. The advantage of this is that a user does not need to mention all of her feedback as long as other members of the feedback are no better than the currently provided unexpected tuples and no worse than the currently provided expected tuples.

3.2.3. Computing Implicit Feedback

To facilitate the computation of implicit feedback, we retain the values of each attribute that appear in the selection predicates for each tuple in $Q(D)$ as shown in Table 4 and Table 6. We call such a query output table an enhanced query output table. An enhanced query output table can be maintained by established provenance techniques (Cheney et al., 2009), (Glavic and Alonso, 2009). The main idea of computing implicit ‘no’ tuples is performing the dominance test for each tuple $t \in U$ against $Q(D)$. Let U^+ be the extended version of U that includes both explicit and implicit ‘no’ tuples and $U \subseteq U^+$. The computation of U^+ is then done as follows: (1) U^+ is initialized to U (Step 1 in Algorithm 1) and (2) for each tuple $t \in U$, if there exists any tuple $t' \in Q(D) \setminus U$ such that $t \geq_G t'$, then we add t' to U^+ (Steps 2-6 in Algorithm 1).

Example 8. *Consider the data given in Example 2. The user feedback for unexpected tuples consists of* $P1$*,* $P3$*, and* $P10$*.*

Algorithm 2 Computing Implicit 'yes' Tuples

```
$E^+ \leftarrow E$;
for each $t \in E$ do
    if $\exists t' \in Q'(D) \setminus E$ such that $t' \geq_G t$ then
        Add $t'$ to $E^+$;
    end if
end for
```

That is, $U = \{P1, P3, P10\}$. According to Algorithm 1, we compute U^+ as follows: (1) U^+ is initialized to $\{P1, P3, P10\}$ (Step 1 in Algorithm 1) and (2) tuple $P8$ is added to U^+ as $P10 \geq_G P8$ (Steps 2-6 in Algorithm 1). Therefore, we get $U^+ = \{P1, P3, P8, P10\}$.

To compute the implicit 'yes' tuples, we perform the dominance test between each tuples in E and tuples $Q'(D)$. Let E^+ be the extended version of E that includes both explicit and implicit 'yes' tuples and $E \subseteq E^+$. The computation of E^+ is then done as follows: (1) E^+ is initialized to E (Step 1 in Algorithm 2)and (2) for each tuple $t \in E$, if there exists any tuple $t' \in Q(D)' \setminus E$ such that $t' \geq_G t$, we add t' to E^+ (Steps 2-6 in Algorithm 2).

Example 9. *Consider the data given in Example 2. The user feedback for expected tuples consists of $P11$ and $P12$: $E = \{P11, P12\}$. According to Algorithm 2, we compute E^+ as follows: (1) E^+ is initialized to $\{P11, P12\}$ (Step 1 in Algorithm 2) and (2) tuple $P13$ and $P14$ are added to E^+ as $P13 \geq_G P11$ and $P14 \geq_G P11$ (Steps 2-6 in Algorithm 2). Therefore, we get $E^+ = \{P11, P12, P13, P14\}$.*

4. Query Refinement

In this section, we describe our query refinement algorithms. The purpose of our query refinement algorithms is to exclude as many as possible unexpected tuples as well as include as many as possible expected tuples in the new query output. Without any loss of generality, we consider simple queries for Section 4.1 and Section 4.2. In Section 4.1, we show that excluding unexpected tuples involves selection of a subset of query conditions for CNF type queries. To select this subset of conditions we provide a greedy approximation that can run in polynomial time. In Section 4.2, we show how we can include expected tuples in the refined query output. Finally, in Section 4.3 we provide solutions for both unexpected and expected tuples that occur for the same query output.

4.1. Exclusion of Unexpected Tuples

We know that unexpected tuples are part of the resultant tuples and generally stretch out near the boundaries of the predicates. For simple CNF type queries, let us assume that $R = \{t_1, t_2, ...t_m\}$, $C = c_1 \wedge c_2 \wedge ... \wedge c_l$, and $U \subseteq R$. To exclude an unexpected tuple $t \in U$, we need to dissatisfy at least one $c_i \in C$ (i.e., tight some predicates).This is because conditions are ANDed together in CNF type queries.

Example 10. *Consider the data given in Example 1. Now, if we want to exclude "Peter" from the result set, we can update the initial query by adopting one or more of the following: (a) HD='Master' to HD not in ('Master'); (b) Rank≤4 to Rank≤3; (c) Npub≥2 to Npub>2.*

As U represents a set of tuples, we may need to modify a set of conditions $\{c_i\}$ in C. This selection of query conditions offers a set-cover problem (i.e., selecting a set out of $2^{|C|} - 1$ sets) which is *NP-complete* and therefore, cannot be solved in polynomial time. Another goal is to find the minimum number of conditions which we need to modify to exclude U. In this paper, we give a greedy approximation for this selection process that can run in polynomial time. The greedy approach proposed in this paper is based on two different requirement types to indicate how users want to exclude the unexpected tuples. These requirement types are similar to those proposed by Kießling and Köstler (2002).

(a) **Soft Requirement**: excludes unexpected tuples only if their exclusion does not affect any truly positive tuple.

(b) **Hard Requirement**: excludes unexpected tuples whatever happens to truly positive tuples.

The greedy approach we propose in this paper has two phases. In the first phase, we go through each predicate and find the temporary updates that we may need to perform to exclude the unexpected tuples. In the second phase, a greedy selection strategy is given to select a subset of the temporary updates offered in the first phase. For this, we maintain two lists of tuples in our algorithms. These lists are: *pList* and *nList*. For each tuple $t \in R$, if it is truly positive tuple, we put it in the *pList*, otherwise we put it in the *nList*. Algorithm 3 and Algorithm 4 show how to offer the temporary updates for each condition $c_i \in C$ when the requirement is *soft* and *hard*, respectively. The run-time complexities of these algorithms are the same and they are $O(lm)$, where l is the number of predicates in the given query and m is the number of tuples (i.e., $|R|$).

The function $computeFitness(pList_q, nList_q)$ in Algorithm 3 and Algorithm 4 returns the fitness f_q of the temporary update c_q. Let n_{tp} be the number of truly positive tuples in $pList_q$, n_{fp} be the number of false positive tuples in $pList_q$, n_{tn} be the number of truly negative tuples in $nList_q$ and n_{fn} be the number of false negative tuples in $nList_q$. Then, we define fitness f_q in IR-style (Mitchell, 1997; Baeza-Yates and Ribeiro-Neto, 1999) as follows (f_q can be one of the following three measures):

(a) **Sensitivity**: Sensitivity measures the proportion of actual positive tuples which are correctly retained by the offered temporary update. The purpose of this measure is to retain as many as possible of the actual positive tuples in the refined query output.

$$f_q = \frac{n_{tp}}{n_{tp} + n_{fn}} \qquad (1)$$

(b) **Specificity**: Specificity measures the proportion of negative (unexpected) tuples which are correctly excluded by the offered temporary update. The purpose of this measure is to

Algorithm 3 Temporary Updates for Soft Exclusion

```
for $q = 1$ to $l$ do
    if $op_q$=‘$\geq$’ or ‘$>$’ then
        set $op_q \leftarrow$‘$\geq$’;
        set $v_q \leftarrow$ min $v_q$ from $pList_q$;
    else if $op_q$=‘$\leq$’ or ‘$<$’ then
        set $op_q \leftarrow$‘$\leq$’;
        set $v_q \leftarrow$ max $v_q$ from $pList_q$;
    else if $op_q$=‘$=$’ then
        set $op_q \leftarrow$‘$in$’;
        set $v_q \leftarrow values\_in\_pList_q$;
    end if
    update both $pList_q$ and $nList_q$;
    $f_q$=computeFitness($pList_q$, $nList_q$);
end for
```

Algorithm 4 Temporary Updates for Hard Exclusion

```
for $q = 1$ to $l$ do
    if $op_q$=‘$\geq$’ or ‘$>$’ then
        set $op_q \leftarrow$‘$>$’;
        set $v_q \leftarrow$ max $v_q$ from $nList_q$;
    else if $op_q$=‘$\leq$’ or ‘$<$’ then
        set $op_q \leftarrow$‘$<$’;
        set $v_q \leftarrow$ min $v_q$ from $nList_q$;
    else if $op_q$=‘$=$’ then
        set $op_q \leftarrow$‘$not\_in$’;
        set $v_q \leftarrow values\_in\_nList_q$;
    end if
    update both $pList_q$ and $nList_q$;
    $f_q$=computeFitness($pList_q$, $nList_q$);
end for
```

Algorithm 5 Greedy Selection

```
Sort $C$ (i.e., temporary updates) in descending order based
on $f_q$;
Set $T \leftarrow U$; // initialization
for $q = 1$ to $l$ do
    if $T = \emptyset$ then
        break;
    else if $T \cap nList_q \neq \emptyset$ then
        Make temporary $c_q$ permanent in $C$;
        $T = T - nList_q$;
    end if
end for
```

exclude as many as possible of the actual negative tuples in the refined query output.

$$f_q = \frac{n_{tn}}{n_{tn} + n_{fp}} \tag{2}$$

(c) **Accuracy:** Accuracy measures the proportion of true results (both true positives and true negatives), which are correctly retained (and excluded) by the offered temporary update. The purpose of this measure is to retain/exclude as many as possible of the actual positive/negative tuples in the refined query output.

$$f_q = \frac{n_{tp} + n_{tn}}{n_{tp} + n_{fp} + n_{tn} + n_{fn}} \tag{3}$$

The value of f_q is used to select the temporary updates c_q to exclude the unexpected tuples. Algorithm 5 shows how we can select the c_q progressively in greedy manner from C to exclude the set of unexpected tuples T ($T \subseteq U$). The complexity of this algorithm is $O(l \log l)$, where l is the number of predicates (i.e., $|C|$).

Example 11. *(Soft Exclusion) Consider the data and the query given in Example 1. The unexpected feedback is {‘Peter’}. Now, the pLists and nLists are:*

$pList_{HD}$={‘Noah’}, $nList_{HD}$={‘Peter’}
$pList_{Npub}$={‘Noah’}, $nList_{Npub}$={‘Peter’}
$pList_{Rank}$={‘Noah’}, $nList_{Rank}$={‘Peter’}

The temporary updates are:
HD in {‘Master’} // value ‘Master’ is selected from pList
Npub $\geq$ 2 // value 2 is selected from pList
Rank $\leq$ 3 // value 3 is selected from pList

The updated pLists and nLists are:
$pList_{HD}$={‘Peter’, ‘Noah’}, $nList_{HD}$={}
$pList_{Npub}$={‘Peter’, ‘Noah’}, $nList_{Npub}$={}
$pList_{Rank}$={‘Noah’}, $nList_{Rank}$={‘Peter’}

The fitness scores (accuracy) of the temporary updates are:
$f_{HD} = \frac{1+0}{1+1+0+0} = 0.5$
where $n_{tp} = 1$, $n_{fp} = 1$, $n_{tn} = 0$, $n_{fn} = 0$
$f_{Npub} = \frac{1+0}{1+1+0+0} = 0.5$
where $n_{tp} = 1$, $n_{fp} = 1$, $n_{tn} = 0$, $n_{fn} = 0$
$f_{Rank} = \frac{1+1}{1+0+1+0} = 1.0$
where $n_{tp} = 1$, $n_{fp} = 0$, $n_{tn} = 1$, $n_{fn} = 0$

The sorted list of the temporary updates are:
Rank $\leq$ 3 (1.0)
HD in {‘Master’} (0.5)
Npub $\geq$ 2 (0.5)

Greedy Selection:
T={‘Peter’} // set of unexpected tuples, U
Iteration#1:
Select Rank $\leq$ 3 to be permanent in the refined query, Q′
T=T-$nList_{Rank}$=$\emptyset$.

No more iterations as T is empty.

The refined query is: “SELECT Sname FROM T1, T2 WHERE HD=‘Master’ AND Rank≤3 AND Npub≥2 AND T1.Uname=T2.Uname”. We can easily verify that soft exclusion successfully excludes unexpected feedback ‘Peter’ and also does not affect any positive tuples in the new result.

Example 12. *(Hard Exclusion) Consider the data given in Example 1 and the modified query “SELECT Sname FROM*

T1, T2 WHERE HD='Master' AND Rank≤ 4 AND Npub≥1 AND T1.Uname=T2.Uname". Also, consider a new row in Table 1: {'Rui', 'CA Irvine', 3, 'Master'}. Hence, the result set consists of {'Peter', 'Rui', 'Noah', 'Susan'}. The unexpected feedback is {'Peter'}. Now, the pLists and nLists are:

$pList_{HD}$={'Rui', 'Noah', 'Susan'}, $nList_{HD}$={'Peter'}
$pList_{Npub}$={'Rui', 'Noah', 'Susan'}, $nList_{Npub}$={'Peter'}
$pList_{Rank}$={'Rui', 'Noah', 'Susan'}, $nList_{Rank}$={'Peter'}

The temporary updates are:
HD not in {'Master'} // value 'Master' is selected from nList
Npub > 2 // value 2 is selected from nList
Rank < 4 // value 4 is selected from nList

The updated pLists and nLists are:
$pList_{HD}$={}, $nList_{HD}$={'Peter', 'Rui', 'Noah', 'Susan'}
$pList_{Npub}$={'Rui'}, $nList_{Npub}$={'Peter', 'Noah', 'Susan'}
$pList_{Rank}$={'Noah', 'Susan'}, $nList_{Rank}$={'Rui', 'Peter'}

The fitness scores (accuracy) of the temporary updates are:
$f_{HD} = \frac{0+1}{0+0+1+3} = 0.25$
where $n_{tp} = 0, n_{fp} = 0, n_{tn} = 1, n_{fn} = 3$
$f_{Npub} = \frac{1+1}{1+0+1+2} = 0.50$
where $n_{tp} = 1, n_{fp} = 0, n_{tn} = 1, n_{fn} = 2$
$f_{Rank} = \frac{2+1}{2+0+1+1} = 0.75$
where $n_{tp} = 2, n_{fp} = 0, n_{tn} = 1, n_{fn} = 1$
The sorted list of the temporary updates are:
Rank < 4 (0.75)
Npub > 2 (0.50)
HD not in {'Master'} (0.25)

Greedy Selection:
T={'Peter'} // set of unexpected tuples, U
Iteration#1:
Select Rank < 4 to be permanent in the refined query, Q'
T=T-$nList_{Rank}$=∅.

No more iterations as T is empty.

The refined query is: "SELECT Sname FROM T1, T2 WHERE HD ='Master' AND Rank<4 AND Npub≥1 AND T1.Uname=T2.Uname".

We can easily verify that hard exclusion affects the positive tuple 'Rui' in the new result set (though it successfully excludes 'Peter' in the new result set). However, for this example, soft exclusion would not make any change in the original query. In soft exclusion, we exclude the unexpected tuples in the new result set only if it does not affect any positive tuple.

For simple DNF type queries, to exclude an unexpected tuple t we need to dissatisfy all conditions $c_i \in C$ that are satisfied by t. This is because, the conditions c_i are ORed together in DNF type of queries, i.e., $C = c_1 \vee c_2 \vee ... \vee c_l$. Therefore, the method is straightforward. First, we run Algorithm 3 and Algorithm 4. Then, we make the temporary updates offered by Algorithm 3 and Algorithm 4 permanent, based on soft and hard user requirement, respectively.

Algorithm 6 Inclusion of Expected Tuples

```
for $q$ = 1 to $l$ do
    if $op_q$='≥' or '>' and $\exists t \in E$ such that $t \cdot v_q < v_q$ then
        set $op_q$ ←'≥';
        set $v_q$ ← min $v_q$ from $E$;
    else if $op_q$='≤' or '<' and $\exists t \in E$ such that $t \cdot v_q > v_q$ then
        set $op_q$ ←'≤';
        set $v_q$ ← max $v_q$ from $E$;
    else if $op_q$='=' and $\exists t \in E$ such that $t \cdot v_q \neq v_q$ then
        set $v_q \leftarrow v_q \cup t \cdot v_q$;
    end if
end for
```

4.2. Inclusion of Expected Tuples

To include the expected tuples in the result set we need to relax some constraints for some predicates in the query. For simple CNF type queries, let us again assume that $C = c_1 \wedge c_2 \wedge ... \wedge c_l$, and $E \subseteq Q'(D)$, where D represents the universe of discourse for Q. For a predicate $c_q \in C$ if the operator is '≥' or '>' we need to set v_q to the minimum value from E. If the operator is '<' or '≤' we need to set v_q to the maximum value from E and finally, if the operator is '=' we need to set v_q to the bag of values from E. The pseudocode for this is given in algorithm 6. The run-time complexity of this algorithm is $O(lm)$, where l is the number of predicates and m is the number of expected tuples.

Example 13. *Consider the data given in Example 1. Now, if we want to include "John" in the new result set, we need to update the initial query by adopting all of the following: (a) HD='Master' to (HD in ('Master', 'Bachelor')) and (b) Npub≥2 to Npub≥1. This is because the intial query is of CNF type and 'John' fails to satisfy both HD='Master' and Npub≥2.*

For simple DNF type of queries, the inclusion of expected tuples is similar to the exclusion of unexpected tuples in CNF queries. That is, to include an expected tuple in the result set, we have multiple choices. But rather than selecting a subset, we rely on the distance metric for the inclusion. Let us assume that $C = c_1 \vee c_2 \vee ... \vee c_l$. To include a tuple $t \in E$ in the result set we need to satisfy at least one $c_i \in C$. So, to include tuple t in the result set we need to measure the distance of $t.v_q$ to the corresponding binding value v_q of $c_q \in C$. Then, we can update v_q to the closest match. If there are more than one closest matches found, then we can break the tie by the following assumptions:

(a) Users are more confident and less flexible about the equality ('=' and '! =') operators.

(b) Users are more flexible and less confident about the inequality ('>', '≥', '<' and '≤') operators.

Therefore, we define the distance metric as follows:

(a) if a_q is numeric

$$d(t.v_q, c.v_q) = \frac{|t.v_q - c.v_q|}{max(t.v_q, c.v_q)} \tag{4}$$

(b) if a_q is categorical

$$d(t.v_q, c.v_q) = \begin{cases} 1 & \text{if } t.v_q! = c.v_q \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

4.3. Handling both Unexpected and Expected Tuples

The query refinement problem for handling both unexpected and expected tuples in the same query is difficult compared to handling them separately. This could be further complicated if we fail to capture the user intent. To include an expected tuple, we know that we need to relax certain predicates. On the other hand, to exclude an unexpected tuple, we need to restrict certain predicates. It is generally assumed that restriction and relaxation cannot occur in the same predicate. That is, we can relax certain predicates for expected tuples as long as it does not hinder the exclusion of unexpected tuples.

The inclusion of expected tuples and exclusion of unexpected tuples can be done by following the approaches given in sections 4.1 and 4.2. To handle both of them in the same query, we can include the expected tuples in the first phase and in the second phase we can exclude the unexpected tuples, and vice versa. Let us assume that C_u denotes the set of conditions $c_q \in C$ that we need to restrict to exclude the unexpected tuples and C_e denotes the set of conditions $c_q \in C$ that we need to relax to include the expected tuples. We propose two strategies to handle both unexpected and expected tuples in the same query output: (a) Non-Overlapping Strategy (NOS) and (b) Overlapping Strategy (OS).

4.3.1. Non-Overlapping Strategy

In non-overlapping strategy (NOS), we relax all conditions $c_q \in C_e$ for including the expected tuples E^+ in the new query output. Then, we run Algorithm 3, Algorithm 4 and Algorithm 5 on $C - C_e$. That is, after relaxing the conditions in C_e, the remaining predicates of C are restricted to exclude the unexpected tuples. Therefore, in NOS we get $C_u \cap C_e = \emptyset$. The pseudocode of this strategy is given in Algorithm 7.

4.3.2. Overlapping Strategy

In overlapping strategy (OS), we relax all conditions $c_q \in C_e$ for including the expected tuples E^+ in the new query output. Then, we run Algorithm 3, Algorithm 4 and Algorithm 5 on C (not C_u which is unlike the NOS) to exclude the unexpected tuples U^+. That is, all predicates that appear in C are considered for the exclusion of U^+. Therefore, it may happen that $C_u \cap C_e \neq \emptyset$. The pseudocode of this strategy is given in Algorithm 8.

Algorithm 7 Non-Overlapping Strategy

```
if $exe\_order$ is '$ExFUL$' then
    Run Algorithm 6 on $C$ to compute $C_e$;
    Run Algorithm 3 & 4 on $C - C_e$ to compute temporary updates;
    Run Algorithm 5 to compute $C_u$;
else if $exe\_order$ is '$UnFEL$' then
    Run Algorithm 3 & 4 on $C$ to compute temporary updates;
    Run Algorithm 5 to compute $C_u$;
    Run Algorithm 6 on $C - C_u$ to compute $C_e$;
end if
Relax each $c_q \in C_e$ to include $E^+$ in $Q'(D)$;
Restrict each $c_q \in C_u$ to exclude $U^+$ in $Q'(D)$;
```

Algorithm 8 Overlapping Strategy

```
if $exe\_order$ is '$ExFUL$' then
    Run Algorithm 6 on $C$ to compute $C_e$;
    Relax each $c_q \in C_e$ to include $E^+$ in $Q'(D)$;
    Run Algorithm 3 & 4 on $C$ to compute temporary updates;
    Run Algorithm 5 to compute $C_u$;
    Restrict each $c_q \in C_u$ to exclude $U^+$ in $Q'(D)$;
else if $exe\_order$ is '$UnFEL$' then
    Run Algorithm 3 & 4 on $C$ to compute temporary updates;
    Run Algorithm 5 to compute $C_u$;
    Restrict each $c_q \in C_u$ to exclude $U^+$ in $Q'(D)$;
    Run Algorithm 6 on $C$ to compute $C_e$;
    Relax each $c_q \in C_e$ to include $E^+$ in $Q'(D)$;
end if
```

4.3.3. Non-Overlapping Strategy vs. Overlapping Strategy

As we see in overlapping strategy the target predicates for expected and unexpected tuples are overlapped. It may happen that some expected tuples dominated by the unexpected tuples in $C_e \cap C$ dimensions will be excluded too. But in non-overlapping strategy it can not happen as we consider only the remaining predicates after relaxing on C_e. Similar scenarios can arise if we restrict the predicates after relaxation. Therefore, the user should direct in which order he/she prefers to go. We name these orders as follows:

(a) **Expected First Unexpected Last (ExFUL)**: include expected tuples first, then exclude unexpected tuples.

(b) **Unexpected First Expected Last (UnFEL)**: exclude unexpected tuples first, then include expected tuples.

Example 14. *(Non-Overlapping Strategy)*
Consider the data given for Example 1. Suppose both unexpected and expected tuples occur for the same query as follows:
U={'Peter'} and
E={'John'}
Suppose ExFUL order is chosen. We see in Example 13 that (HD='Master' OR HD='Bachelor') and Npub≥1 are selected to include 'John' in the refined query output. Therefore, the only attribute left for exclusion of 'Peter' in the refined query output is Rank. We see in Example 11 that Rank≤3 is selected by our greedy selection strategy to exclude 'Peter'. Hence, the refined query that can include 'John' as well as exclude 'Peter' in the refined query output is as follows: "SELECT Sname FROM T1,

T2 WHERE HD in ('Master', 'Bachelor') AND Rank≤3 AND Npub≥1 AND T1.Uname=T2.Uname;".

Example 15. *(Overlapping Strategy)*
Consider the data and the query given for Example 2. Suppose both unexpected and expected tuples occur for the same query as follows:
U={P1, P3, P10} and E={P11, P12}.
Suppose ExFUL order is chosen.

Inclusion of Expected Tuples: The expected tuples fail to satisfy 'CitationCnt ≥ 80'. We fix this failed predicate as 'CitationCnt ≥ 60' according to steps 2-4 of Algorithm 6. Therefore, the new query for this part becomes as follows:

SELECT PubID FROM Publication
WHERE CitationCnt ≥ 60 AND
PubYear ≥ 1986;

Exclusion of Unexpected tuples: We consider both attributes CitationCnt and PubYear (unlike to non-overlapping strategy which would consider PubYear only) for excluding the tuples in U.

The pLists and nLists are:
$pList_{CitationCnt}$ *= {P2, P4-P7, P9, P11-P14},*
$nList_{CitationCnt}$ *={P1, P3, P8, P10},*
$pList_{PubYear}$ *= {P2, P4-P7, P9, P11-P14} and*
$nList_{PubYear}$ *={P1, P3, P8, P10}.*

The temporary updates (for Hard Exclusion) are:
CitationCnt > 117
where value 117 is selected from $nList_{CitationCnt}$
PubYear> 1989
where value 1989 is selected from $nList_{PubYear}$

The updated pLists and nLists are:
$pList_{CitationCnt}$ *= {P2, P5, P6},*
$nList_{CitationCnt}$ *= {P1, P2, P4, P7-P14 },*
$pList_{PubYear}$ *= {P4, P6, P7, P9, P11-P14} and*
$nList_{PubYear}$ *= {P1-P3, P5, P8, P10}.*

The fitness scores (accuracy) of the temporary updates are:
$f_{CitationCnt} = \frac{3+4}{3+0+4+7} = 0.5$
where $n_{tp} = 3$, $n_{fp} = 0$, $n_{tn} = 4$, $n_{fn} = 7$
$f_{PubYear} = \frac{8+4}{8+0+4+2} = 0.857$
where $n_{tp} = 8$, $n_{fp} = 0$, $n_{tn} = 4$, $n_{fn} = 2$

The sorted list of the temporary updates are:
PubYear > 1989 (0.857)
CitationCnt > 117 (0.5)

Greedy Selection:
T={P1, P3, P10}// set of unexpected tuples, U
Iteration#1:
Select PubYear > 1989 to be permanent in Q'
T=T-$nList_{PubYear}$*=∅*

No more iterations as T is empty.

The refined query is: "SELECT PubID FROM Publication WHERE CitationCnt > 60 AND PubYear > 1989;".

Algorithm 9 Feedback Table Construction

```
Input: Enhanced Query Output Table (OT), Expected tuples (E)
Output: Feedback Table (FT)
Copy OT to FT;
Add each tuple∈ E to FT;
Add a new column'Feddback' to FT;
for each row in FT do
    if tuple ∈ row is expected or truly positive tuple then
        Set row.Feedback to 'yes';
    else
        Set row.Feedback to 'no';
    end if
end for
```

5. Decision Tree Based Query Refinement

A decision tree (DT) is a tree-like model of decisions and probably the most popular classification model. Given an input with well-defined attributes, the DT can classify the input entirely based on making choices about each attribute. But the problem of learning an optimal DT is known to be NP-complete under several aspects of optimality and even for simple concepts (Hyafil and Rivest, 1976).

We use the best-known and most widely-used C4.5 decision tree learning algorithm (Mitchell, 1997) and WEKA implementation of it (Hall et al., 2009). Given labeled data the algorithm initializes a decision tree with one leaf node that represents the whole data. Then it tests each attribute, and chooses the best attribute (A) and value (v) pair, which, if used to split the data into two portions, one with values in attribute $A \geq v$, another with values $\leq v$, results in maximum entropy gain. The algorithm recursively tests and splits the leaves until either, all points in each partition belong to one class, or it becomes statistically insignificant to split further. We set the minimum number of instances in the two most popular branches to one (default 2 in WEKA) to improve its precision and accuracy. In the learned tree, each path P from root to a leaf ('yes') node forms a CNF expression. All such paths are ORed together to form a DNF expression which is ANDed to the original query. Finally, Q^* is simplified to incorporate the learned DNF expression. The pseudocode of the decision tree based query refinement is given in Algorithm 10. The feedback table (FT) for Algorithm 10 is constructed by adding a class-column 'Feedback' to the enhanced query output table (as well as expected tuples). Then, we label each row by putting value 'yes' in this class column if it is a truly positive or expected tuple otherwise we label it 'no'. The pseudocode for constructing the feedback table is given in Algorithm 9.

Table 7: Labeled data for decision tree learning for Example 1

Sname	HD	Npub	Rank	Feedback
John	Bachelor	1	1	yes
Peter	Master	2	4	no
Noah	Master	2	3	yes

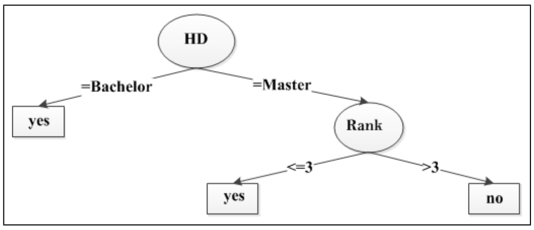


Figure 6: Decision tree learning: Q_{DT}= (HD=‘Bachelor’) OR (HD=‘Master’ AND Rank≤ 3)

Example 16. *Consider the data given in Example 1. The feedback table consists of ‘John’, ‘Peter’ and ‘Noah’ as shown in Table 7. We see that feedback column is labeled with ‘yes’ for ‘John’ and ‘Noah’ as they are expected in the refined query output whereas the same column is labeled with ‘no’ for ‘Peter’ as he is unexpected. The learned decision tree is given in Fig. 6. The query returned by the DT based Query Refinement is as follows:*

SELECT Sname FROM T_1, T_2
WHERE ((HD=‘Bachelor’ AND Rank≤4 AND Npub≥2)
OR (HD=‘Master’ AND Rank≤3 AND Npub≥2))
AND T_1.Uname=T_2.Uname;

We can easily verify from the above refined query that DT fails to learn the appropriate relaxation(s) for including ‘John’ in the refined query output. DT finds it appropriate to separate ‘John’ from other ‘no’ examples based on only ‘HD’ as we see in Fig. 6. No other ‘no’ example exists to learn appropriate relaxation for the attribute ‘NPub’ for ‘John’.

Example 17. *Consider the data given in Example 2. The feedback table and the learned DT for this example data is given in Table 8 and Fig. 7. The query returned by the DT based Query Refinement is as follows:*

SELECT PubID FROM Publication
WHERE (CitationCnt ≥ 80 AND PubYear > 1989)
OR (CitationCnt > 117 AND PubYear ≤ 1989);

6. Experiments

In this section, we demonstrate the effectiveness of our query refinement framework. We first present our experimental setup including the dataset we use and user feedback. We then evaluate our query refinement algorithms in different perspectives as well as against DT based query refinement.

Table 8: Labeled data for decision tree learning for Example 2

PubID	CitationCnt	PubYear	Feedback
P1	96	1989	no
P2	128	1986	yes
P3	100	1989	no
P4	90	1993	yes
P5	148	1986	yes
P6	148	1990	yes
P7	81	1996	yes
P8	103	1986	no
P9	82	1994	yes
P10	117	1987	no
P11	60	1995	yes
P12	72	1996	yes
P13	64	1996	yes
P14	67	1995	yes

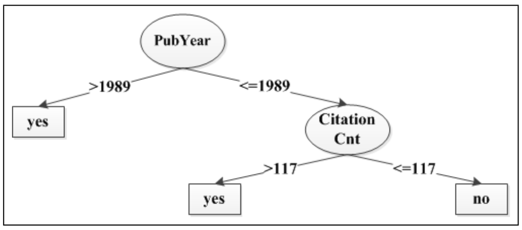


Figure 7: Decision tree learning: Q_{DT}= (PubYear > 1989) OR (PubYear ≤1989 AND CitationCnt > 117)

6.1. Experimental Setup

6.1.1. Environment

We run all our experiments on an Intel(R) Core(TM) Duo E8400 3.0 GHz PC with 3.49 GB RAM. The refinement algorithms are implemented in Java along with MySQL server 5.1. For C4.5 decision tree learning algorithm, we use WEKA class library (Hall et al., 2009).

6.1.2. Datasets and Queries

We use two datasets: DBLP[1] of size 456 MB and UCI Automobile[2] of size 592 KB. We convert the XML based

[1] http://dblp.uni-trier.de/xml/

[2] http://archive.ics.uci.edu/ml/datasets.html

Algorithm 10 DT Based Query Refinement

```
Input: Initial Query($Q$), Feedback Table ($FT$)
Output: Refined Query ($Q^*$)
Run DT learning algorithm on $FT$;
$Q_{DT} \leftarrow \emptyset$;
Let $P$ is a path from the root to a leaf (‘yes’) of DT
  {Depth-first search suffices to find $P$ in DT}
  for each path $P$ do the following:
    (a) Convert each path $P$ into CNFs
    (b) Combine $P$ with $Q_{DT}$: $Q_{DT} \leftarrow Q_{DT}$ OR $P$
$Q^* \leftarrow Q$ AND $Q_{DT}$;
Simplify $Q^*$;
```

Table 9: Dataset Statistics

Dataset	size	#tables	#attributes	Types of attributes
DBLP	456 MB	6	2 to 5	categorical and numerical
UCI Automobile	592 KB	1	26	categorical and numerical

Table 10: DBLP dataset queries of CNF type derived from the base query

Query	SQL Statement	#size
Q_1	SELECT pubid FROM publication, book WHERE citationcnt ≥ 60 AND pubyear ≥ 1980 AND pages ≥ 12 AND rank ≤ 1 AND book = acronym;	15
Q_2	SELECT pubid FROM publication, book WHERE citationcnt ≥ 40 AND pubyear ≥ 1990 AND pages ≥ 8 AND rank ≤ 2 AND book = acronym;	30
Q_3	SELECT pubid FROM publication, book WHERE citationcnt ≥ 50 AND pubyear ≥ 1950 AND pages ≥ 10 AND rank ≤ 2 AND book = acronym;	51
Q_4	SELECT pubid FROM publication, book WHERE citationcnt ≥ 40 AND pubyear ≥ 1980 AND pages ≥ 8 AND rank ≤ 1 AND book = acronym;	82
Q_5	SELECT pubid FROM publication, book WHERE citationcnt ≥ 35 AND pubyear ≥ 1980 AND pages ≥ 10 AND rank ≤ 2 AND book = acronym;	100
Q_6	SELECT pubid FROM publication, book WHERE citationcnt ≥ 30 AND pubyear ≥ 1980 AND pages ≥ 10 AND rank ≤ 2 AND book = acronym;	143
Q_7	SELECT pubid FROM publication, book WHERE citationcnt ≥ 30 AND pubyear ≥ 1980 AND pages ≥ 8 AND rank ≤ 2 AND book = acronym;	161
Q_8	SELECT pubid FROM publication, book WHERE citationcnt ≥ 30 AND pubyear ≥ 1950 AND pages ≥ 8 AND rank ≤ 2 AND book = acronym;	178

Table 11: DBLP dataset queries of CNF type for different attribute groups

Query	SQL Statement	#size
Q_9	SELECT pubid FROM publication WHERE citationcnt ≥ 60 AND pages ≥ 8 AND pubyear ≥ 1980;	33
Q_{10}	SELECT pubid FROM publication, book WHERE citationcnt ≥ 30 AND pages ≥ 8 AND rank ≤ 2 AND book = acronym;	178
Q_{11}	SELECT pubid FROM publication, book WHERE citationcnt ≥ 20 and rank ≤ 1 and pubyear ≥ 1960 and book = acronym;	364
Q_{12}	SELECT pubid FROM publication, book WHERE citationcnt ≥ 18 and rank ≤ 1 and book = acronym;	430

Table 12: Automobile dataset queries of CNF Type

Query	SQL Statement	#size
Q_1	SELECT ID FROM automobile WHERE symboling ≥ 3 AND normalizedlosses ≥ 150 AND bore ≥ 3 AND stroke ≥ 3 AND horsepower ≥ 100;	11
Q_2	SELECT ID FROM automobile WHERE stroke ≥ 2 AND horsepower ≥ 90 AND bodystyle='sedan' AND citympg ≥ 20;	30
Q_3	SELECT ID FROM automobile WHERE compressionratio ≥ 9 AND wheelbase ≥ 90 AND price ≥ 9500 AND bore ≥ 3 AND aspiration = 'std';	56
Q_4	SELECT ID FROM automobile WHERE stroke ≥ 2 AND horsepower ≥ 90 AND fuelsystem = 'mpfi' AND highwaympg ≥ 19 AND price ≥ 3000;	81
Q_5	SELECT ID FROM automobile WHERE length ≥ 150 AND width ≥ 50 AND height ≥ 40 AND curbweight ≥ 200 AND enginesize ≥ 120 AND peakrpm ≥ 4000;	100

Table 13: Automobile dataset queries of DNF Type

Query	SQL Statement	#size
Q_1	SELECT ID FROM automobile WHERE price ≥ 16500 OR compressionratio ≥ 20 OR highwaympg ≥ 42;	71
Q_2	SELECT ID FROM automobile WHERE compressionratio ≥ 10 OR wheelbase ≥ 100 OR bore ≥ 3.5;	112
Q_3	SELECT ID FROM automobile WHERE length ≥ 170 OR width ≥ 70 OR height ≥ 54 OR curbweight ≥ 3200;	127
Q_4	SELECT ID FROM automobile WHERE stroke ≥ 4 OR horsepower ≥ 85 OR citympg ≥ 35;	145

Table 14: Comparison of NOS, OS and DT for DBLP dataset queries of CNF type derived from the base query

Query	NOS			OS			DT		
	Acc(%)	Prec(%)	Time	Acc(%)	Prec(%)	Time	Acc(%)	Prec(%)	Time
Q_1	42.06	49.40	35.85	18.59	35.88	28.57	-3.66	33.99	49.52
Q_2	48.70	67.67	38.34	23.27	44.57	26.11	-4.94	41.48	70.75
Q_3	48.75	62.62	28.95	46.14	61.33	41.62	40.11	50.99	58.76
Q_4	65.36	76.79	47.40	62.18	74.66	27.61	48.84	55.83	89.84
Q_5	56.27	71.41	30.22	54.62	70.52	28.76	51.82	64.59	116.02
Q_6	61.68	76.13	31.45	60.75	75.53	29.09	58.92	70.10	143.93
Q_7	68.01	77.59	28.45	67.86	77.54	29.73	65.15	70.96	157.65
Q_8	70.42	79.79	29.13	70.36	79.74	29.67	68.74	76.84	155.34
Avg	**57.66**	**70.17**	**33.72**	**50.47**	**64.97**	**30.14**	**40.62**	**58.10**	**105.23**

DBLP dataset into relational data format which consists of six tables with attributes 2 to 5. We also incorporate ERA conference/journal ranking information[3] into the DBLP dataset. The Automobile dataset has a single table with 26 attributes. Some statistics about these datasets are shown in Table 9. To create test queries of CNF type for DBLP dataset, we use the following base query to create 8 different queries of different result size (i.e., queries that return 15 to 178 tuples in the initial result set for DBLP dataset as shown in Table 10):

$$Q_i = \pi_{pubid}\sigma_{citationcnt \geq const_1 \wedge pubyear \geq const_2 \wedge pages \geq const_3 \wedge rank \leq const_4 \wedge book=acronym}(publication \times book)$$

where $const_i$ is a constant for the i^{th} attribute. We also create another set of queries for DBLP dataset, by having conditions on different group of attributes, that return 52 to 430 tuples in the initial result sets for DBLP dataset as shown in Table 11. The purpose of having different attribute groups in the test queries is to diversify the resultant tuples. For the Automobile dataset, we create queries of CNF type that return 11 to 100 tuples in the initial result sets as given in Table 12 (by having conditions on different group of attributes again). We also create queries of DNF type for different attribute groups that return 71 to 145 tuples in the initial result sets for the Automobile dataset as given in Table 13.

6.1.3. User Feedback

We randomly pick a set of tuples from the initial result set as the unexpected tuples and limit the threshold to no more than 30% of the resultant tuples. To get expected tuples, we pick tuples from the non-answers (both virtual and complete tuples). Once we get the expected tuples and the user preferences for unexpected tuples, we then run point dominance to complement the feedback by following the approaches explained in Section 3.

6.1.4. Evaluation Process

We evaluate our algorithms proposed in this paper in two different aspects: (1) Quality of Results: precision ($=n_{tp}/(n_{tp} + n_{fp})$) and accuracy ($=(n_{tp} + n_{tn})/(n_{tp} + n_{fp} + n_{tn} + n_{fn})$) measures (Mitchell, 1997; Baeza-Yates and Ribeiro-Neto, 1999) and (2) Efficiency: time taken to refine a query. We compare the performances of the proposed query refinement algorithms with the decision tree based tuple classifier as follows: (1) We use the queries given in Table 10, Table 11, Table 12, and Table 13; (2) We make 100 different input instances for each query by randomly picking the result tuples as the unexpected tuples (no more than 30%) and setting the virtual tuples (randomly relaxing the predicates in the given query) from non-answers as the expected tuple set; (3) Then, we update the implicit feedback; and (4) Finally, we calculate the average of accuracy and precision improvement (in %) and the time (in $10^5\times$nano-seconds) taken by our methods (NOS and OS with ExFUL order) and the decision tree based query refinement.

6.2. Experimental Results for CNF Type of Queries

In this subsection, we present the experimental results for CNF type of queries. We compare our methods (NOS and OS) with DT based query refinement for both DBLP and Automobile datasets, and queries given in Table 10, Table 11, and Table 12.

6.2.1. Overlapping Strategy vs. Non-overlapping Strategy

On average, non-overlapping strategy (NOS) performs better than overlapping strategy (OS) in terms of both accuracy and precision improvement as it is evident from Table 14, Table 15, and Table 16. This is because we cannot restrict and relax the same predicates in a query for different purposes at the same time. NOS takes more time to refine a query compared to the OS on average as we see from Table 14, Table 15, and Table 16. Table 17, Table 18, and Table 19 show for how many input instances (out of 100) for each query, NOS provides better accuracy and precision improvement; and takes less time compared to the OS. NOS gives better accuracy and precision improvement compared to the OS for more than 98% of input instances for both datasets as we see from Table 17, Table 18, and Table 19.

6.2.2. DT vs. Other Methods

Table 14, Table 15, and Table 16 compare the performances of NOS, OS and DT based query refinements. It is evident from Table 14, Table 15, and Table 16 that our methods (both NOS and OS) outperform the DT based query refinement in terms of both accuracy and precision improvement in average. Our methods also take less time in average for all queries than the DT based query refinement except Q_1 for the Automobile dataset. Table 20, Table 21, and Table 22 show for how many input instances (out of 100) for each query, our methods provide better accuracy and precision improvement; and take less time compared to the DT based query refinement. We observe that our methods perform better compared to the DT based query refinement for more than 94% input instances in terms of precision improvement, on average, for both datasets. This is because we retain the truly positive tuples in the answer set as many as possible in our approach ("*soft*" requirement). Our methods also give better accuracy improvement for more than 63% of input instances, on average, for both DBLP and Automobile datasets as we see from Table 20, Table 21, and Table 22. It is quite promising that our methods run faster for more than 80% of input instances, on average, for both datasets. Finally, Table 14, Table 15, Table 16, Table 20, Table 21, and Table 22 manifest that our methods (NOS and OS) outperform the DT based query refinement for both the base query (instances are given in Table 10) and queries having conditions on different attribute groups (queries given in Table 11 and Table 12).

6.3. Experimental Results for DNF Type of Queries

In this subsection, we present the experimental results for DNF type of queries. We compare our method (OS in ExFUL order) with DT based query refinement for the Automobile dataset, and queries given in Table 13. To do so, we first include

[3] http://www.core.edu.au/

Table 15: Comparison of NOS, OS and DT for DBLP dataset queries of CNF type for different attribute groups

Query	**NOS**			**OS**			**DT**		
	Acc(%)	Prec(%)	Time	Acc(%)	Prec(%)	Time	Acc(%)	Prec(%)	Time
Q_9	41.73	46.98	21.10	32.58	43.06	20.21	16.48	32.32	36.14
Q_{10}	50.65	62.43	22.03	50.37	62.31	22.07	44.65	49.23	131.55
Q_{11}	53.73	67.68	22.44	53.49	67.58	22.53	53.46	57.56	301.57
Q_{12}	85.76	86.67	18.14	85.67	86.63	18.44	1.87	2.46	187.70
Avg	**57.97**	**65.94**	**20.93**	**55.53**	**64.89**	**20.81**	**29.12**	**35.39**	**164.24**

Table 16: Comparison of NOS, OS and DT for Automobile dataset queries of CNF type

Query	**NOS**			**OS**			**DT**		
	Acc(%)	Prec(%)	Time	Acc(%)	Prec(%)	Time	Acc(%)	Prec(%)	Time
Q_1	39.00	45.55	44.81	31.77	42.63	36.08	30.23	41.11	16.39
Q_2	65.85	65.20	26.01	23.85	34.87	25.70	-17.07	29.16	64.72
Q_3	32.68	68.90	34.46	29.74	65.66	27.31	26.98	47.89	70.90
Q_4	48.91	64.45	27.40	45.62	62.36	26.76	50.79	54.41	49.75
Q_5	14.60	47.43	32.19	8.16	43.01	34.37	16.47	36.30	88.34
Avg	**40.21**	**58.31**	**32.97**	**27.83**	**49.71**	**30.04**	**21.48**	**41.77**	**58.02**

Table 17: DBLP dataset queries of CNF type derived from base query: NOS vs. OS

Query	Better Accuracy #instances (Out of 100)	Better Precision #instances (Out of 100)	Less Time #instances (Out of 100)
Q_1	100	100	41
Q_2	100	100	33
Q_3	100	100	44
Q_4	100	100	47
Q_5	100	99	50
Q_6	100	96	48
Q_7	100	100	46
Q_8	97	99	61
Avg	**99.63**	**99.25**	**46.25**

Table 18: DBLP dataset queries of CNF type for different attribute groups: NOS vs. OS

Query	Better Accuracy #instances (Out of 100)	Better Precision #instances (Out of 100)	Less Time #instances (Out of 100)
Q_9	99	99	33
Q_{10}	98	99	50
Q_{11}	99	99	62
Q_{12}	99	99	54
Avg	**98.75**	**99.00**	**49.75**

Table 19: Automobile dataset queries of CNF type: NOS vs. OS

Query	Better Accuracy #instances (Out of 100)	Better Precision #instances (Out of 100)	Less Time #instances (Out of 100)
Q_1	100	100	45
Q_2	100	100	54
Q_3	98	100	40
Q_4	100	100	42
Q_5	100	100	54
Avg	**99.60**	**100.00**	**47.00**

Table 20: DBLP dataset queries of CNF type derived from base query: Our Methods vs. DT

Query	Better Accuracy #instances (Out of 100)	Better Precision #instances (Out of 100)	Less Time #instances (Out of 100)
Q_1	98	100	88
Q_2	100	100	100
Q_3	78	98	100
Q_4	89	93	100
Q_5	51	93	100
Q_6	100	94	100
Q_7	46	97	100
Q_8	48	96	100
Avg	**76.25**	**96.38**	**98.50**

Table 21: DBLP dataset queries of CNF type for different attribute groups: Our Methods vs. DT

Query	Better Accuracy #instances (Out of 100)	Better Precision #instances (Out of 100)	Less Time #instances (Out of 100)
Q_9	99	99	99
Q_{10}	54	99	99
Q_{11}	51	99	99
Q_{12}	99	99	99
Avg	**75.75**	**99.00**	**99.00**

Table 22: Automobile dataset queries of CNF type: Our Methods vs. DT

Query	Better Accuracy #instances (Out of 100)	Better Precision #instances (Out of 100)	Less Time #instances (Out of 100)
Q_1	77	100	7
Q_2	100	80	100
Q_3	58	99	100
Q_4	46	99	100
Q_5	34	95	100
Avg	**63.00**	**94.60**	**81.40**

Table 23: Comparison of our method with DT for Automobile dataset queries of DNF Type

Query	**Our Method**				**DT**			
	Acc(%)	Prec(%)	Time	#NSub	Acc(%)	Prec(%)	Time	#NSub
Q_1	45.63	68.21	8.69	1	35.86	54.91	50.54	2.55
Q_2	34.09	67.96	13.48	1	26.17	46.24	66.00	1.38
Q_3	40.00	70.58	12.92	1	41.56	60.87	82.18	1.79
Q_4	61.30	65.42	17.05	1	64.03	67.24	69.82	3.70
Avg	**45.26**	**68.04**	**13.03**	**1**	**41.91**	**57.32**	**67.13**	**2.35**

Table 24: Automobile dataset queries of DNF Type: Our Method vs. DT

Query	Better Accuracy #instances (Out of 100)	Better Precision #instances (Out of 100)	Less Time #instances (Out of 100)
Q_1	93	100	99
Q_2	94	100	100
Q_3	66	100	100
Q_4	48	96	98
Avg	**75.25**	**99.00**	**99.25**

Table 25: Statistical significance (p-value): Our Methods vs. DT

Query^{CNF}	Acc.	Prec.	Time
Q_1	2.087e-07	<2.2e-16	0.005111
Q_2	2.087e-07	<2.2e-16	<2.2e-16
Q_3	2.087e-07	1.905e-07	5.432e-05
Q_4	2.087e-07	1.187e-07	3.286e-06
Q_5	2.087e-07	0.004171	1.759e-14
Q_6	2.087e-07	0.09648	1.451e-10
Q_7	2.087e-07	0.001221	3.391e-11
Q_8	2.087e-07	0.08659	4.942e-12
Q_9	2.087e-07	4.284e-12	0.01464
Q_{10}	2.087e-07	1.397e-11	<2.2e-16
Q_{11}	2.087e-07	< 2.2e-16	<2.2e-16
Q_{12}	2.087e-07	< 2.2e-16	<2.2e-16

(a) Queries of CNF type in DBLP dataset

Query^{CNF}	Acc.	Prec.	Time
Q_1	2.087e-07	< 2.2e-16	1.000000
Q_2	2.087e-07	< 2.2e-16	0.007973
Q_3	2.087e-07	1.816e-10	0.0975
Q_4	2.087e-07	2.063e-13	2.042e-14
Q_5	2.087e-07	5.68e-08	< 2.2e-16

(b) Queries of CNF type in Automobile dataset

Query^{DNF}	Acc.	Prec.	#NSub	Time
Q_1	2.07e-07	6.579e-08	< 2.2e-16	2.992e-10
Q_2	2.087e-07	2.2e-16	1.446e-09	<2.2e-16
Q_3	2.087e-07	< 2.2e-16	< 2.2e-16	5.246e-08
Q_4	2.087e-07	0.8698	< 2.2e-16	2.209e-11

(c) Queries of DNF type in Automobile dataset the expected tuples in the new query result set by following the distance based approach described in Section 4.2 and then, we exclude the unexpected tuples in the new query result set by following the approach described in Section 4.1. Table 23 and Table 24 compare the performances of our method and the DT based query refinement. From Table 23, we see that the performance of our method, on average, is comparable to the DT based query refinement. That is, our method performs better than or similarly to DT in most cases, on average. From Table 24, we see that our method gives better accuracy and precision improvement for more than 75% and 99% of input instances on average, respectively. Our method also does not change the structure of the original query at all, it only modifies the binding values of the attributes involved in the query predicates. On the other hand, the DT changes the structure of the original query by introducing sub-queries (equal to the number of rules learned by DT) in the new query. As we see from Table 23 that DT introduces 2.35 sub-queries (#NSub) on average for the tested Automobile queries.

6.4. Discussion

6.4.1. Implication of Results

We observe that DT based query refinement gives poor performance, on average, compared to the other methods proposed in this paper. Now, the question is why does DT achieve these poor results? DT is mainly an information-gain-theory based linear classifier and its success relies heavily on underlying data distribution (Hyafil and Rivest, 1976), (Mitchell, 1997). The FT constructed by Algorithm 9 does not guarantee any statistical data distribution. As an example, consider the FT given in Table 7. We see that DT fails to learn appropriate relaxations to include 'John' in the refined query output (see Fig. 6). DT finds it appropriate to separate 'John' from other 'no' examples based on 'HD'. No other 'no' example exists to learn appropriate relaxation for attribute 'NPub' for 'John'. The effects of this fact are observed for queries with fewer resultant tuples. For example, we achieve negative improvement in ACC for CNF type of queries $Q1$ and $Q2$ in DBLP dataset (see in Table 14) and for query $Q2$ in the Automobile dataset (see Table 16). That is, the DT based query refinement favors queries having balanced negative and positive examples only. If there are no negative examples available, DT fails to learn any relaxation to include the expected tuples in the refined queries (i.e., DT does not work at all for this case). Other methods proposed in this paper are data-driven (consider only the sample data available and do not depend on any data-distribution). Therefore, these methods perform better compared to the DT based query refinement for CNF type of queries.

For DNF Type of queries, the performance of our method is better than or similar to the DT based query refinement. However, our method does not introduce subqueries in the refined queries. On the other hand, the DT based query refinement introduces subqueries in the new query (i.e., changes the structure of the original query), which is its major disadvantage. For example, consider the query Q_2 of DNF type given in Table 13. The refined query returned by our method for including the expected tuples for a sample run is given below:

SELECT id FROM automobile WHERE (compressionratio ≥ 10 OR wheelbase ≥ 103 OR bore ≥ 4);

The refined query for excluding the unexpected tuples from the above query via soft and hard updates are:

SELECT id FROM automobile WHERE (compressionratio ≥ 10 OR wheelbase ≥ 103 OR bore ≥ 4); and

SELECT id FROM automobile WHERE (compressionratio > 23 OR wheelbase > 116 OR bore > 4);

Finally, the refined query returned by the DT based query refinement is given below:

SELECT id FROM automobile WHERE (compressionratio ≥ 10 OR wheelbase ≥ 103 OR bore ≥ 4) AND ((wheelbase ≤ 102 AND bore > 3 AND wheelbase > 100 AND wheelbase > 101) OR (wheelbase > 102 AND bore > 3));

We conclude that DT based query refinement approach tends to make the refined query very complex, which may not be desirable always.

6.4.2. Statistical Significance

To show that the performance of our methods is statistically significant, we perform 2-sample t-test (David and Gunnink, 1997) on our achieved results. Table 25 shows the statistical significance (p-value) of our results. We observe that there is more than 99% probability that our methods give better accuracy and take less time compared to the DT based query refinement for both CNF and DNF type of queries in both DBLP and Automobile dataset, as we see from Table 25(a), Table 25(b) and Table 25(c). There is also more than 99% probability that our methods give better precision for all queries except Q_4 of DNF type in automobile dataset (see in Table 25(a), Table 25(b) and Table 25(c)). These tests ensure that our methods are more likely to achieve better results compared to the DT based query refinement. However, we believe that an extensive study is needed for making such a general statement if the tested queries are complex (i.e., queries involving nested subqueries), which we plan to study in our future work.

7. Related Work

The contributions mostly related to this paper are by Ma et al. (2006), Tran and Chan (2010), and Liu et al. (2010). Ma et al. (2006) model query refinement as learning the structure of the query as well as learning the relative importance of query components. They consider only DNF type of queries and feedbacks on the initial result set. They do not consider what new information a user expects to see (i.e. what is missing). Tran and Chan (2010) model query refinement by collecting missing tuples as feedback from the user. Authors exploit the idea of *skyline queries* to report the closest refined query w.r.t. the original one to minimize the distance between the refined and the original query. In the refined query, they also consider new predicates to add/drop. However, this approach can add/drop wrong predicates to/from the query and their refined queries may introduce more false positives (unexpected tuples) in the result set. Liu et al. (2010) collect *false positives* identified by the users to modify the initial rules in information extraction settings to exclude unexpected results.

Both why (unexpected) and why-not (expected) tuples are treated separately in the above models. In our approach, we collect both *false positives* (i.e., unexpected tuples) and *false negatives* (i.e., expected tuples) to refine the initial imprecise query. We propose query refinement as a framework to modify the original query for minimizing the number of unexpected tuples as well as maximizing the number of expected tuples. We emphasize our work is on only modifying the query conditions (not ranking the database tuples, nor adding/dropping of predicates). Still, our methods achieve improved accuracy and precision on average compared to DT based query refinement.

There are three main research areas, closely related to query refinements and these are: (1) too few/many answers problem, (2) top-k query processing and (3) cardinality assurance problem. The too few answers problem is addressed through the relaxation of query conditions by utilizing the domain-specific distance metrics and returning nearby records (Kadlag et al., 2004). Histogram information, showing how records are distributed over each attributed range of values, as well as a count of the total number of records are also exploited to encounter this problem. The too many answers problem is analogous to the too few answers problem and is generally addressed through the restriction of query conditions. We can view the too many/few answers problem as a predicate transformation problem as mentioned by Mishra and Koudas (2009). In top-k query processing, top ranked tuples are returned according to some ranking function and with respect to the submitted query. The primary problem with this approach is the requirement of a scoring function which may not be readily available as mentioned by Ilyas et al. (2003). The cardinality assurance problem deals with the execution of an SQL query that must ensure a result cardinality when executed on a given database. Users may also have preferences on how to transform the original query to increase/decrease the result size. Such cases occur in Machine Intelligence applications. Mishra and Koudas (2009) address the cardinality assurance problem (i.e., too many/few answers problem) by incorporating user feedback to best capture user preferences.

Nambiar and Kambhampati (2003), Kadlag et al. (2004), Nambiar and Kambhampati (2005), Ma et al. (2006) and Koudas et al. (2006) place great importance on the the need of supporting exploratory or imprecise queries in relational and web databases. This has particular application to the many/few answers problem often faced by database users. Mishra and Koudas (2009) address this problem by incorporating user feedback to best capture user preferences, whereas Koudas et al. (2006) propose a lattice based framework for minimum amount of query conditions relaxation. User feedback is also considered as first-class citizen in other areas of database systems such as semi-structured data (Cao et al., 2010), information integration (Belhajjame et al., 2011) and schema mappings (Belhajjame et al., 2010), and extensively studied in information retrieval techniques (Moon et al., 2010) including image (Hoi and Wu, 2011) and video retrieval (Vrochidis et al., 2010). In this paper, we show how one can capture user intent by collecting feedback (a sample set of unexpected and expected tuples) and modify the initial imprecise query to better fulfill users' information needs. This kind of cooperative behavior of database systems is discussed by Amer-Yahia et al. (2005) to bridge the gap between database systems and information retrieval techniques to increase the usability of databases (Jagadish et al., 2007).

8. Conclusion and Future Work

This paper presents a framework to capture the user intent through feedback for refining the initial imprecise queries. For this, we require the user to provide a sample set of expected and unexpected tuples. Then, we show how one can complement it by automatically suggesting the complete set by applying our proposed point domination theory. Finally, we show how we can refine the initial imprecise query to exclude the unexpected tuples as well as to include the expected tuples. We provide a greedy selection approach for selecting the subset of conditions through which the query refinement is achieved. The overall complexity of our algorithms are $O(lm + l \log l)$, where m is the number of tuples to be considered and l is the number of predicates in the given query. Experimental results suggest that our methods are quite promising compared to the naïve decision tree based query refinement approach. We plan as a future work to extend/adapt the proposed framework for complex queries as well as to solve many and/or few answers problems together with the user given attribute and/or predicate preferences.

9. Acknowledgement

This work is supported by the grants of ARC Discovery Projects DP120102627 and DP110102407. We would like to thank anonymous reviewers for their constructive comments.

References

Amer-Yahia, S., Case, P., Rölleke, T., Shanmugasundaram, J., Weikum, G., 2005. Report on the db/ir panel at sigmod 2005. SIGMOD Record 34, 71–74.

Baeza-Yates, R.A., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Belhajjame, K., Paton, N.W., Embury, S.M., Fernandes, A.A.A., Hedeler, C., 2010. Feedback-based annotation, selection and refinement of schema mappings for dataspaces, in: EDBT, pp. 573–584.

Belhajjame, K., Paton, N.W., Fernandes, A.A.A., Hedeler, C., Embury, S.M., 2011. User feedback as a first class citizen in information integration systems, in: CIDR, pp. 175–183.

Börzsönyi, S., Kossmann, D., Stocker, K., 2001. The skyline operator, in: ICDE, pp. 421–430.

Cao, H., Qi, Y., Candan, K.S., Sapino, M.L., 2010. Feedback-driven result ranking and query refinement for exploring semi-structured data collections, in: EDBT, pp. 3–14.

Chapman, A., Jagadish, H.V., 2009. Why not?, in: SIGMOD Conf., pp. 523–534.

Cheney, J., Chiticariu, L., Tan, W.C., 2009. Provenance in databases: Why, how, and where. Foundations and Trends in Databases 1, 379–474.

David, H.A., Gunnink, J.L., 1997. The paired t test under artificial pairing. The American Statistician 51, pp. 9–12.

Glavic, B., Alonso, G., 2009. The perm provenance management system in action, in: SIGMOD Conf., pp. 1055–1058.

Green, T.J., Karvounarakis, G., Tannen, V., 2007. Provenance semirings, in: PODS, pp. 31–40.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The weka data mining software: an update. SIGKDD Explorations 11, 10–18.

Herschel, M., Hernández, M.A., 2010. Explaining missing answers to spjua queries. PVLDB 3, 185–196.

Hoi, S.C.H., Wu, P., 2011. Sire: a social image retrieval engine, in: ACM Multimedia, pp. 817–818.

Huang, J., Chen, T., Doan, A., Naughton, J.F., 2008. On the provenance of non-answers to queries over extracted data. PVLDB 1, 736–747.

Hyafil, L., Rivest, R.L., 1976. Constructing optimal binary decision trees is np-complete. Inf. Process. Lett. 5, 15–17.

Ilyas, I.F., Aref, W.G., Elmagarmid, A.K., 2003. Supporting top-k join queries in relational databases, in: VLDB, pp. 754–765.

Islam, M.S., Liu, C., Zhou, R., 2012. On modeling query refinement by capturing user intent through feedback, in: Zhang, R., Zhang, Y. (Eds.), Australasian Database Conference, ACS, Melbourne, Australia. pp. 11–20.

Jagadish, H.V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A., Yu, C., 2007. Making database systems usable, in: SIGMOD Conf., pp. 13–24.

Kadlag, A., Wanjari, A.V., Freire, J., Haritsa, J.R., 2004. Supporting exploratory queries in databases, in: DASFAA, pp. 594–605.

Kießling, W., Köstler, G., 2002. Preference sql - design, implementation, experiences, in: VLDB, pp. 990–1001.

Koudas, N., Li, C., Tung, A.K.H., Vernica, R., 2006. Relaxing join and selection queries, in: VLDB, pp. 199–210.

Liu, B., Chiticariu, L., Chu, V., Jagadish, H.V., Reiss, F., 2010. Automatic rule refinement for information extraction. PVLDB 3, 588–597.

Ma, Y., Mehrotra, S., Seid, D.Y., Zhong, Q., 2006. Raf: An activation framework for refining similarity queries using learning techniques, in: DASFAA, pp. 587–601.

Mishra, C., Koudas, N., 2009. Interactive query refinement, in: EDBT, pp. 862–873.

Mitchell, T.M., 1997. Machine learning. McGraw Hill series in computer science, McGraw-Hill.

Moon, T., Li, L., Chu, W., Liao, C., Zheng, Z., Chang, Y., 2010. Online learning for recency search ranking using real-time user feedback, in: CIKM, pp. 1501–1504.

Motro, A., 1988. Vague: A user interface to relational databases that permits vague queries. ACM Trans. Inf. Syst. 6, 187–214.

Motro, A., 1994. Cooperative database systems, in: FQAS, pp. 1–16.

Nambiar, U., Kambhampati, S., 2003. Answering imprecise database queries: a novel approach, in: WIDM, pp. 126–133.

Nambiar, U., Kambhampati, S., 2005. Answering imprecise queries over web databases, in: VLDB, pp. 1350–1353.

Nandi, A., Jagadish, H.V., 2011. Guided interaction: Rethinking the query-result paradigm. PVLDB 4, 1466–1469.

Sultana, K.Z., Bhattacharjee, A., Amin, M.S., Jamil, H.M., 2009. A model for contextual cooperative query answering in e-commerce applications, in: FQAS, pp. 25–36.

Tran, Q.T., Chan, C.Y., 2010. How to conquer why-not questions, in: SIGMOD Conf., pp. 15–26.

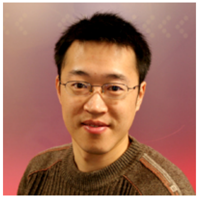
Voorneveld, M., 2003. Characterization of pareto dominance. Oper. Res. Lett. 31, 7–11.

Vrochidis, S., Kompatsiaris, I., Patras, I., 2010. Optimizing visual search with implicit user feedback in interactive video retrieval, in: CIVR, pp. 274–281.

Wong, R.C.W., Fu, A.W.C., Pei, J., Ho, Y.S., Wong, T., Liu, Y., 2008. Efficient skyline querying with variable user preferences on nominal attributes. Proc. VLDB Endow. 1, 1032–1043.

Mr. Md. Saiful Islam received his BSc (Hons) and MS degree in Computer Science and Engineering from University of Dhaka, Bangladesh, in 2005 and 2007, respectively. He is currently a third year PhD student in the Faculty of Information and Communication Technologies at the Swinburne University of Technology, Australia. He has also been serving as a faculty member in the Institute of Information Technology, University of Dhaka since January 2008. He has published over 30 peer-reviewed papers in various journals and conference proceedings. His research interests are in the areas of data management, information retrieval, machine learning and computer architecture.

Dr. Chengfei Liu is a Professor and the head of the Web and Data Engineering research group in the Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia. He received the BS, MS and PhD degrees in Computer Science from Nanjing University, China in 1983, 1985 and 1988, respectively. Prior to joining Swinburne, he taught at the University of South Australia and the University of Technology Sydney, and was a Research Scientist at Cooperative Research Centre for Distributed Systems Technology, Australia. He also held visiting positions at the Chinese University of Hong Kong, the University of Aizu in Japan, and IBM Silicon Valley Lab in USA. He has published over 160 peer-reviewed papers in various journals and conference proceedings and has served on technical program committees and organizing committees of over 100 international conferences or workshops in the areas of database systems, Web information systems, and workflow systems. His current main research interests include keywords search on structured data, query processing and refinement for advanced database applications, query processing on uncertain data and big data, and data-centric workflows.



Dr. Rui Zhou is a research fellow currently working at Swinburne University of Technology, Australia, where he finished his PhD. in 2010. He received his BS. and MS. at Northeastern University, China in 2004 and 2006, respectively. His research interest is mainly about query processing on XML data and graph data.