RECEIVED

JUL 26 1993

OSTI

RESTRICTING QUERY RELAXATION THROUGH USER CONSTRAINTS

by

Terry Gaasterland¹

Abstract

This paper describes techniques to restrict and to heuristically control relaxation of deductive database queries. The process of query relaxation provides a user with a means to automatically identify new queries that are related to the user's original query. However, for large databases, many relaxations may be possible. The methods to control and restrict the relaxation process introduced in this paper focus the relaxation process and make it more efficient. User restrictions over the database domain may be expressed as *user constraints*. This paper describes how user constraints can restrict relaxed queries. Also, a set of heuristics based on cooperative answering techniques are presented for controlling the relaxation process. Finally, the interaction of the methods for relaxing queries, processing user constraints, and applying the heuristic rules is described.

1 Introduction

In a knowledge-based system, a critical problem is the presentation of useful responses to users' queries. A query answering process should collaborate with users to find the information that they are seeking. Users do not always know enough about the data itself and how it is organized to ask accurate queries. Sometimes queries contain misconceptions and are bound to fail. Sometimes they are too general. Other times they are too specific.

When users ask queries that do not obtain the information that they seek, a technique called *relaxation* [5, 7] enables a knowledge-based system in the form of a deductive database to work interactively with a user to find alternative answers that are related to the answers of the original query. The relationships are defined through *taxonomy clauses*, which are first-order expressions about semantic relationships among predicates and constants in the deductive database. Taxonomy clauses are used to relax predicates, constants and variable dependencies in the user's query to expand the query's search space. With the relaxation technique, a user can ask a specific query and get related answers as well as direct answers. Without the relaxation method, users must figure out alternative queries for themselves, formulate them, and pose them to the system.

Alternatively, users may have restrictions on the knowledge domain that they would like to have addressed for every query that is asked. Users can express these restrictions through *user constraints* [5, 9]. User constraints express states that a user wants to disallow and states that a user wants to always persist. Each time a user asks a query, user constraints are applied to the query using semantic query optimization techniques. The resulting query produces answers that satisfy the user's restrictions. Thus, user constraints free users from the necessity of building all relevant restrictions into every query that they ask.

¹Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439.
EMAIL: gaasterland@mcs.und.edu

MASTER

The relaxation method and the user constraint method have been identified and formalized as independent processes. In this paper, we explore how user constraints can be used to restrict relaxation. Since several directions for relaxation may be available even with the restrictions, we introduce a set of heuristics for ordering possible relaxations of a query.

Section 2 gives some basic definitions about deductive databases. Section 3 discusses the relaxation process. Section 4 describes user constraints and reviews how semantic query optimization techniques are used to apply user constraints to a query. Section 6 shows how user constraints can be used to guide relaxation and eliminate relaxed queries that will not be of interest to the user. Section 7 describes heuristics for ordering alternative relaxations.

2 Definitions

A *term* is a constant, a variable, or a function over constants, variables, and functions. For example, the constant c and the variable X are terms, as is the expression $f(c, f(X))$. An *atom* is a predicate with terms in its arguments, as in $p(X, f(X))$. A *literal* is a positive or negative atom.

Deductive databases are comprised of syntactic information and semantic information [11]. The syntactic information consists of the *intensional database* (IDB) and the *extensional database* (EDB). The IDB is a set of clauses, or rules, of the form $A \leftarrow B_1, \dots, B_n$, $n > 0$, where A and each B_i is an atom. The EDB is a set of clauses, or facts, of the form $A \leftarrow$, where A is a ground atom.

Direct answers to database queries, which are clauses of the form $\leftarrow B_1, \dots, B_n$, are found by using SLD-resolution on the query, IDB, and EDB clause to produce a search tree. The root node of the search tree is the query clause; each node in the tree is produced by applying an IDB rule to the node above.

The semantic information in a deductive database consists of a set of *integrity constraints* (IC). Without loss of generality, we assume integrity constraints to have the form $\leftarrow C_1, \dots, C_n, E_1, \dots, E_m$, where each C_i is a literal whose predicate appears in an EDB fact or the head of an IDB rule.

An integrity constraint restricts the states that a database can take. For example, the integrity constraint *No person can be both male and female*, possibly written as $\leftarrow person(X), male(X), female(X)$, restricts people in a database from having two genders. Because the constraints on a database add no new deductive knowledge to the database, they are considered semantic information rather than syntactic information.

3 Relaxation

As noted by many researchers, including [1, 3, 4, 12, 13, 15, 16, 18], one form of cooperative behavior involves providing associated information that is relevant to a query. Generalizing a query in order to capture neighboring information is a means to obtain possibly relevant information.

Gaasterland, Godfrey, and Minker [7] have defined a method to *relax* a query in order to find neighboring information. A query can be relaxed in at least three ways: (1) rewriting a predicate with a more general predicate; (2) rewriting a constant (term) with a more general constant (term); and (3) breaking a join dependency across literals in the query. The first two relaxations are achieved in a general manner using *taxonomy clauses* that define hierarchical type relationships between predicates and constants in the database language. For example, the following clauses define relationships between the predicates *travel*, *flight*, and *train*:

```
T1: travel(From, To)  $\leftarrow$ 
    serves_area(A, From),
    serves_area(B, To),
    flight(A, B).
T2: travel(From, To)  $\leftarrow$ 
```

Figure 1: Type Taxonomy for Airports

```

serves_area(C, From),
serves_area(D, To),
train(C, D).

```

The following query Q1 contains a request to travel by plane from Washington, D.C., to Miami, Florida. It can be relaxed to produce an alternative query Q1':

```

Q1: ← flight('National', 'MIA').
Q1': ← travel(From, To),
      serves_area('National', From),
      serves_area('MIA', To).

```

After the relaxation step, SLD resolution can be used to find related answers.

3.1 Formal Definition

As described in [7], taxonomy clauses provide the basis for relaxation. They may be clauses that are already present in the database or they may be a separate set of clauses that are provide by the database designer as semantic information about the database. Either way, they describe an implicit taxonomy. For example, T1 and T2 describe a taxonomy in which flights and trains are modes of travelling.

For each taxonomy clause in the database, an additional *reciprocal clause* is added to the database. The body atom in a taxonomy clause that appears in the hierarchy is called a *key* atom. The head of a reciprocal clause contains the key atom of a taxonomy clause. The head atom in the taxonomy clause appears in the body of the reciprocal clause. For a taxonomy clause of the form $H \leftarrow B_1, \dots, B_n, K$, where K is the key atom, the reciprocal clause is $relax(K) \leftarrow B_1, \dots, B_n, H$. For T1 and T2 above, the reciprocal clauses are the following:

```

R1: relax(flight(A, B)) ←
    serves_area(A, From),
    serves_area(B, To),
    travel(From, To).
R2: relax(train(C, D)) ←
    serves_area(C, From),
    serves_area(D, To),
    travel(From, To).

```

Taxonomy clauses capture type relations over constants as well as over predicates. For example, the following taxonomy clauses represent some of the taxonomy relationships in Figure 1:

```
T3: airport(A) ← wash_dc_airport(A).
T4: airport(A) ← baltimore_airport(A).
T5: wash_dc_airport(A) ← A unifies 'National'.
T5: wash_dc_airport(A) ← A unifies 'BWI'.
T6: baltimore_airport(A) ← A unifies 'BWI'.
```

The reciprocal clause for T6 is the following:

```
R6: relax(A unifies 'BWI') ←
    baltimore_airport(A).
```

Prior to relaxation, a query is *variable substituted* so that the constants are moved out into separate atoms. Variable substitution replaces constants and repeated variables in a query with new unique variables. Then it equates the original constants and variables to the new variables. For relaxation, the equality expression uses the predicate *unifies*. For example, the variable substituted form of Q2 is Q2':

```
Q2: ← flight('Dulles', 'Orly').
Q2': ← flight(X, Y),
      X unifies 'Dulles',
      Y unifies 'Orly'.
```

In [7], the following meta-interpreter is described to perform deduction together with relaxation using reciprocal clauses:

```
relax_solve(As) ←
    relaxing(As, Bs),
    solve(Bs).

relaxing([A|As], [A|Bs]) ←
    relaxing(As, Bs).
relaxing([A|As], Bs) ←
    clause(relax(A), C's),
    relaxing(C's, Ds),
    relaxing(As, Es),
    append(Ds, Es, Bs).
relaxing([], []).
```

The second rule for the predicate *relaxing* calls the reciprocal clauses through the predicate *relax*. The predicate *solve* may perform deduction as in [19], or it may apply cooperative answering techniques or semantic optimization techniques to the query solving process as in [2, 14].

3.2 Controlling Relaxation

Several methods have been identified for controlling the relaxation process. One approach is to present each potential relaxation of a query to the user and allow the user to choose which relaxation to pursue. Another

approach is to pursue the relaxations breadth-first.

Section 5 describes how a set of possible relaxations can be restricted with user constraints prior to using either of these control mechanisms. Section 6 gives heuristics for ordering relaxations. Before describing these techniques, we present user constraints in the next section.

4 User Constraints

A user's restrictions on a database can be modeled as a set of constraints, called user constraints, and then used to alter a user's query. The query modification is performed by means of semantic query optimization [2]. First, we present an example. Next, we formalize the notion of user constraints.

4.1 An Example

Consider a query to a database *Which airline can I use to travel from Washington DC to Paris?*:

Q3: $\leftarrow \text{travel}(\text{Airline}, \text{Airport1}, \text{'CDG'}, \text{Time}),$
 $\text{near}(\text{Airport1}, \text{Washington}).$

and a user constraint *I refuse to fly into JFK*:

U1: $\leftarrow \text{flight}(\text{Airline}, \text{Num}, \text{From}, \text{To}, \text{Time}),$
 $\text{To} = \text{'JFK'}.$

with the following database information²:

I2: $\text{travel}(\text{Airline}, \text{From}, \text{To}, \text{Time}) \leftarrow$
 $\text{flight}(\text{Airline}, \text{Number}, \text{From}, \text{X}, \text{Time}),$
 $\text{travel}(\text{X}, \text{To}, \text{Time2}).$
I3: $\text{travel}(\text{From}, \text{To}, \text{Time}) \leftarrow$
 $\text{flight}(\text{Airline}, \text{Num}, \text{From}, \text{To}, \text{Time}).$
E1: $\text{flight}(\text{'Delta'}, 1001, \text{'National'}, \text{'JFK'}, 16:30).$
E2: $\text{flight}(\text{'Delta'}, 140, \text{'JFK'}, \text{'CDG'}, 20:30).$
E3: $\text{flight}(\text{'Air-France'}, 231, \text{'BWI'}, \text{'CDG'}, 17:30).$
E4: $\text{near}(\text{'National'}, \text{'Washington-DC'}).$
E5: $\text{near}(\text{'BWI'}, \text{'Washington-DC'}).$

With only IDB $\cup$ EDB, the query can be answered with *Take Delta to JFK and then to CDG*. However, such an answer is not desired by the user — it violates the user constraint of *I refuse to travel into JFK*.

An answer that satisfies the user's constraint provides the alternative that does not go through JFK. When Q3 is expanded with I2, Q3' occurs:

Q3': $\leftarrow \text{flight}(\text{Airline}, \text{Num}, \text{Airport}, \text{X}, \text{Time}),$
 $\text{travel}(\text{Airline}, \text{X}, \text{'CDG'}, \text{Time2}),$
 $\text{near}(\text{Airport}, \text{'Washington-DC'}).$

²Notice that these rules for *travel* differ slightly from the previous rules for *travel* in T1 and T2.

Merging Q3' with U1 produces the following:

Q3'': $\leftarrow \text{flight}(\text{Airline}, \text{Num}, \text{Airport}, X, \text{Time}),$
 $\text{travel}(\text{Airline}, X, \text{'CDG'}, \text{Time2}),$
 $\text{near}(\text{Airport}, \text{'Washington-DC'}),$
 $X \neq \text{'JFK'}.$

Subsequently, a solution obtained with E1 that substitutes JFK for X fails since $\{\text{'JFK'} \neq \text{'JFK'}\}$ is *false*. The answer obtained from E3 — *Take an Air France flight from BWI to CDG* — will suit the user more than an answer that does not consider the user's interests.

4.2 Formal Definition

To formalize the notion of user constraints, we assume that the EDB and IDB are function-free and contain definite Horn clauses. The closed world assumption is also assumed. In the following discussion, let DB be a set of IDB rules and EDB facts. Let L be the language of DB and let E be the following set of evaluable predicates: $\{=, <, >, \leq, \geq, \text{is with } +, -, *, \div\}$.

Definition 1 A user constraint is a clause of the form $\leftarrow P_1, \dots, P_m, E_1, \dots, E_n$, where each P_i is a literal in L and each E_j is an evaluable atom with a predicate in E. $\square$

In order to restrict the answer set of a query with a set of user constraints, the user constraints are semantically compiled into the query.

Definition 2 A user-constrained rule is defined to be of the form

$A \leftarrow B_1, \dots, B_m, \{R_1, \dots, R_n\}.$

where A is the procedure head of the rule, each literal B_i is either an extensional predicate or an evaluable predicate, and the R_j s are residues obtained from the rule $A \leftarrow B_1, \dots, B_m$ and some set of user constraints. $\square$

Definition 3 Let DBS be a semantically constrained database with IDB rules, IDB. Let UC be a set of user constraints in the language of DBS. The user-constrained database of DBS, DBS', is the set of user-constrained rules obtained from IDB and UC. $\square$

If a user constraint is semantically compiled into a query, then the answers for the new query reflect the restrictions of the user constraint. To show this property, we need a definition for an *answer*.

Definition 4 Let Q be a query, $\leftarrow Q_1 \dots Q_n$, where each Q_i is a literal.

Let $\{\theta_1 \dots \theta_m\}$ be a set of substitutions such that for each θ_i , $DB \vdash (Q_1, \dots, Q_n)\theta_i$.

Then each θ_i is an answer for Q. $\square$

An integrity constraint, IC, is always *true* in a database — that is, $DB \cup IC$ is consistent. This is not necessarily true of a user constraint. User constraints have a slightly different characterization since we wish to use user constraints to check answer substitutions that have been obtained for a query.

Intuitively, we can check each answer substitution θ by applying it to each variable substituted user constraint, U, that partially subsumes the query with the one-way substitution σ . If the database entails the instantiated user constraint, that is $DB \vdash U\sigma\theta$, then the answer substitution is acceptable. If the database does not entail the instantiated user constraint, then the answer substitution is considered to *violate* the user constraint.

The following theorem states this formally.

Theorem 1 Let Q be a flattened query³ $\leftarrow Q_1 \dots Q_n$, where each Q_i is an EDB literal and let U be a flattened, variable substituted user constraint, $\leftarrow P_1, \dots, P_m, E_1, \dots, E_r$, where each P_i is an EDB literal and each E_i is an evaluable atom. Let S be a subset of atoms in $\{P_1, \dots, P_m\}$ that subsumes some subset of atoms in $\{Q_1, \dots, Q_n\}$ with substitution σ . Let C be the conjunction of P_i s in $\{P_1, \dots, P_m\} - S$. Let Q' be a new query obtained by semantically compiling Q and U . Q' has the form:

$\leftarrow Q_1, \dots, Q_n, \neg(C\sigma, E_1\sigma, \dots, E_r\sigma)$. Let $\{\theta_1 \dots \theta_p\}$ be a set of answers for Q' , that is, $DB \vdash (Q_1, \dots, Q_n)\theta_i$, $1 \leq i \leq p$. Then $DB \vdash U\sigma\theta_i$ $1 \leq i \leq p$.

A proof for this theorem may be found in [5].

A short example helps to clarify the relationship between an answer substitution and a user constraint. Consider a query Q that says, *Which airlines fly from DCU to other airports?* and a variable substituted user constraint U that says, *I don't want to know about flights into JFK*:

$Q4: \leftarrow \text{flight}(A, N, \text{'DCU'}, X).$
 $U: \leftarrow \text{flight}(AA, NN, Y, Z), Z = \text{'JFK'}.$

U partially subsumes $Q4$ with the substitution $\sigma = \{A/AA, N/NN, \text{'DCU'}/Y, X/Z\}$. The semantically compiled query is the following:

$Q4': \leftarrow \text{flight}(A, N, \text{'DCU'}, X), X \neq \text{'JFK'}.$

Suppose that the database DB has three facts:

$\text{flight}(\text{united}, 401, \text{'National'}, \text{'LAX'}).$
 $\text{flight}(\text{united}, 914, \text{'National'}, \text{'CDG'}).$
 $\text{flight}(\text{united}, 1612, \text{'National'}, \text{'JFK'}).$

Then there are two answer substitutions for $Q4'$, θ_1 , and θ_2 :

$\theta_1 = \{\text{united}/A, 401/N, \text{'LAX'}/X\}$
 $\theta_2 = \{\text{united}/A, 914/N, \text{'CDG'}/X\}.$

Each answer substitution can be applied to the user constraint to obtain two instantiated constraints. $U\sigma\theta_1$ is the following:

$\leftarrow \text{flight}(\text{united}, 401, \text{'National'}, \text{'LAX'}),$
 $\text{'LAX'} = \text{'JFK'}.$

and $U\sigma\theta_2$ is the following:

$\leftarrow \text{flight}(\text{united}, 914, \text{'DCU'}, \text{'CDG'}),$
 $\text{'CDG'} = \text{'JFK'}.$

In this case, for $i=1$ and $i=2$, $DB \vdash U\sigma\theta_i$ trivially since the body of the user constraint evaluates to *false*, thus making the entire user constraint evaluate to *true*.

Now, we turn to the problem of how to restrict relaxation through user constraints.

³Flattening as defined in [5] is based on [17]. IDB rules are applied to a query until the query contains only EDB atoms, recursive atoms, or negated atoms.

5 Restricting Relaxation With User Constraints

Users may have a variety of concerns that they build into queries. When the queries are relaxed, those concerns may be dismissed. In this section, we demonstrate that the user constraints described in the preceding section provide a means to separate properties that a user wishes to hold for every query from properties that a user has assembled for a particular query.

Consider the following query which asks for flights from Dulles airport to Orly and also asks whether Dulles is a safe airport:

Q5: $\leftarrow \text{flight}(\text{'Dulles'}, \text{'Orly'}), \text{safe}(\text{'Dulles'})$.

Variable substitution, relaxation, and deduction with R1 and T1 produce a new relaxed query:

Q5': $\leftarrow \text{flight}(X, Y),$
 $\text{serves_area}(X, A1),$
 $\text{serves_area}(Y, A2),$
 $\text{serves_area}(\text{'Dulles'}, A1),$
 $\text{serves_area}(\text{'Orly'}, A2),$
 $\text{safe}(\text{'Dulles'}).$

If the user's intent in the original query was to make sure that the airport of departure was *safe*, then the intent is lost in the relaxed query. It would be better to allow the user to express this intent through the following user constraint which means *I want to know about flights from X to Y only if X is safe*:

U2: $\text{safe}(X) \leftarrow \text{flight}(X, Y).$

Given the query:

Q6: $\leftarrow \text{flight}(\text{'Dulles'}, \text{'Orly'}).$

semantic query optimization with U produces Q5 above.

Relaxation of Q6 produces the following query:

Q6': $\leftarrow \text{flight}(X, Y),$
 $\text{serves_area}(X, A1),$
 $\text{serves_area}(Y, A2),$
 $\text{serves_area}(\text{'Dulles'}, A1),$
 $\text{serves_area}(\text{'Orly'}, A2).$

Applying U2 to Q6' produces a restricted query that ensures that the airport of departure is *safe*:

Q6'': $\leftarrow \text{flight}(X, Y),$
 $\text{serves_area}(X, A1),$
 $\text{serves_area}(Y, A2),$
 $\text{serves_area}(\text{'Dulles'}, A1),$
 $\text{serves_area}(\text{'Orly'}, A2),$
 $\text{safe}(X).$

In other cases, user constraints can be used to eliminate relaxations that the user would not want to see. For example, suppose a user expresses that he or she does not want to take a train trip that is more than 300 miles:

U3: $\leftarrow \text{train}(X, Y),$
 $\text{distance}(X, Y, D), D > 300.$

If the user asks about flights from Washington, D.C., National to Boston's Logan Airport using the query Q7, a relaxed query using R1 and T2 would be Q7':

Q7: $\leftarrow \text{flight}(\text{'National'}, \text{'Logan'}).$
 Q7': $\leftarrow \text{train}(X, Y),$
 $\text{serves_area}(X, A1),$
 $\text{serves_area}(Y, A2),$
 $\text{serves_area}(\text{'National'}, A1),$
 $\text{serves_area}(\text{'Logan'}, A2).$

The user constraint U2 applies to the relaxed query to produce Q7''. Q7'' is eliminated by the EDB facts E6-E8 below:

Q7'': $\leftarrow \text{train}(X, Y),$
 $\text{serves_area}(X, A1),$
 $\text{serves_area}(Y, A2),$
 $\text{serves_area}(\text{'National'}, A1),$
 $\text{serves_area}(\text{'Logan'}, A2),$
 $\neg (\text{distance}(A1, A2, D), D > 300).$
 E6: $\text{serves_area}(\text{'National'}, \text{'Washington-DC'}).$
 E7: $\text{serves_area}(\text{'Logan'}, \text{'Boston'}).$
 E8: $\text{distance}(\text{'Washington-DC'}, \text{'Boston'}, 400).$

The first example about safe airports illustrates how user constraints can be used to express persistent properties separately and carry them into relaxed queries. The second example illustrates how user constraints can be used to express limiting information that can eliminate possible relaxations.

Each of these applications of user constraints makes the relaxation technique more tightly coupled with users' interests over the domain of the database.

6 Heuristics for Ordering Relaxations

In [10, 14], Gal described a method to use integrity constraints to correct misconceptions in user's queries. Gal and Minker provided heuristics for selecting which integrity constraints or false presuppositions to return to a user. In this section, we briefly describe their approach, and then we adapt the heuristics for the purpose of ordering the possible relaxations in a query.

6.1 Background: Identifying Cooperative Information

Integrity constraints specify unchanging states of the database like *Every entity that is a medical doctor must have a medical degree*, and *No entity is both a doctor and a lab technician*. The cooperative answering system of Gal and Minker [14] uses integrity constraints to provide alternative responses to users. If a user asks a query such as *Who are all of the employees at Metropolitan Hospital who are both lab technicians and doctors?* the query would be sure to fail to have any answers. A literal answer would be *none*. A better answer would contain the constraint information, *No one is both a doctor and a lab technician*. Alternatively, a user might ask, *Who are the doctors at Metropolitan Hospital with medical degrees?* A literal answer would provide a list of all the doctors at the hospital. A cooperative and informative answer would tell the user,

All doctors have medical degrees. The system might then go on and list all of the doctors, or it might ask whether the user still wants a list of the individuals. In both cases, the user's query has a misconception about the contents of the database, and the integrity constraint information corrects the misconception. In the first case, the misconception causes the query to fail; in the second, it causes the query to cover extra search space when seeking answers to the query.

The cooperative answering system [14] uses semantic query optimization techniques to identify interactions between integrity constraints and queries that reveal cooperative information to be included in an answer to a query. The system also searches for false presuppositions of failed queries. Letting a user know about a false presupposition is an alternative way of providing an informative answer.

A set of heuristic rules determines which pieces of cooperative information ought to be included when multiple pieces of information are available. In addition, the user is provided with a means to ask for terse or for highly explanatory answers.

6.2 Ordering Relaxations

We order relaxations by means of a set of heuristics that inspect any misconceptions in a query that have been identified through query optimization techniques. In the following heuristic rules, a misconception is considered to be a set of subgoals that ensures that a query will fail:⁴

1. Relaxation of misconceptions is given priority over other relaxations.
2. When several misconceptions are available for relaxation, precedence levels are set up as follows:
 - (a) Relaxation of a misconception detectable without an EDB search has priority over misconceptions detectable only with an EDB search.
 - (b) Relaxation of a misconception discovered through constraints takes precedence over those discovered as false presuppositions.
 - (c) Relaxation of a misconception that can be detected with a single constraint has priority over those detectable only with several constraints.
3. When several misconceptions have the same relaxation priority, the misconception identified by the constraint with the deepest derivation is selected for relaxation. Otherwise, the misconceptions are randomly ordered.

To illustrate these rules for ordering relaxations, consider the following integrity constraints which say that only military personnel can fly on military flights and that no flight leaves Andrews Air Force Base for Los Angeles after 11:00 p.m.:

IC1: $\neg \text{passenger}(\text{Person}, N),$
 $\neg \text{military_personnel}(\text{Person}),$
 $\text{flight}(\text{'military'}, N, \text{From}, \text{To}, \text{Time}).$
 IC2: $\neg \text{flight}(\text{Airline}, N, \text{From}, \text{To}, T),$
 $\text{From} = \text{'Andrews'},$
 $T > 23:00, \text{To} = \text{'LAX'}.$

Both constraints apply to the query, *Can Dr. Silva fly on a military flight from Andrews AFB to Los Angeles after midnight?*

⁴The cooperative answering system identifies three types of misconception: misconceptions that cause a query to fail, misconceptions that cause a query to cover redundant search space, and misconceptions that cause a query to have a very large number of answer substitutions. The first type of misconception is the most useful for ordering possible relaxations.

Q8: $\leftarrow$ *passenger*('Silva', *N*),
 flight('military', *N*, 'Andrews', 'LAX', *T*),
 $T \geq 24:00$.

IC1 would require an EDB search on *military_personnel*('Silva') to detect a violation. IC2 does not. Thus, relaxation of the atoms *flight*('military', *Number*, 'Andrews', 'LAX', *Time*) and $Time \geq 24:00$ would be preferred over relaxation of *passenger*('Silva', *Number*)

If no constraint identifies a misconception, but several false presuppositions are available, the most general false presupposition is selected for relaxation. Consider a variable substituted query that violates neither of the above constraints:

Q9: $\leftarrow$ *flight*(*Airline*, *Number*, *X*, *Y*, *Time*),
 X unifies Andrews, *Y unifies LAX*.

asking for flights from Andrews to LAX. It contains a presupposition that flights from Andrews exist; it also contains the presupposition that flights from Andrews to LAX exist. If the database contains no information about flights from Andrews, the query will fail and both presuppositions will be false. According to the relaxation ordering rules, the atom corresponding to the more general false presupposition would be relaxed — *X unifies Andrews*.

These heuristics for ordering the relaxations have several important properties. Rule 1 targets misconceptions to be relaxed first. The relaxation process formulates a new, corrected query for a user. The user then has the option to use the relaxed query or to pose an entirely new query.

Rules 2a, 2b, and 2c make the query answering process more efficient. When misconceptions can be found without EDB search, they are posed as sites of relaxation before any other misconceptions are sought. Identifying misconceptions via integrity constraint violations is much less expensive than searching for least failed subqueries of a query. Thus, any search for false presuppositions should be delayed until after integrity constraint violations are relaxed.

Rule 3 deals with derived constraints. When constraints take the form of a rule, as in IC3, it may be possible to derive new constraints like IC5 from IC3 and IC4:

IC3: *lunch_flight*(*X*) $\leftarrow$ *flight_times*(*X*, *T1*, *T2*),
 $T1 > 11:00$, $T1 < 13:00$,
 $Diff \text{ is } T2 - T1$, $Diff > 1:30$.
 IC4: $\leftarrow$ *lunch_flight*(*X*),
 airline(*X*, 'United Express').
 IC5: $\leftarrow$ *airline*(*X*, 'United Express'),
 flight_times(*X*, *T1*, *T2*),
 $T1 > 11:00$, $T1 < 13:00$,
 $Diff \text{ is } T2 - T1$, $Diff > 1:30$.

IC3 says that any flight longer than 1.5 hours that starts between 11:00 a.m. and 1:00 p.m. must be a lunch flight. IC4 says that no United Express flight can be a lunch flight. IC5, derived from IC3 and IC4, says that no United Express flight can be more than 1.5 hours long or it must not start between 11:00 a.m. and 1:00 p.m.

If all three of these constraints were violated by some query, Rule 3 would give higher relaxation priority to the atoms involved in the violation of IC5, since IC5 would reflect the misconceptions identified by all three constraints.

The heuristic rules for ordering relaxation can be used in two ways. They can order a set of possible relaxations before the relaxations are presented to the user as alternatives, or they can be used to automatically select a relaxed query. Selecting between these two modes of interaction is an open problem. It requires testing of users' ongoing interactions with a system. The better choice may be sensitive to the domain of

the database and the level of expertise of the user.

7 Processing Queries

To use user constraints to restrict relaxation and to order possible relaxations with the heuristic rules given above, a user's query is processed in the following steps:

1. A query Q is processed with deduction as usual or with cooperative answering techniques as described in [14] and [5].
2. Additional relaxed queries are found as follows:
 - (a) If the query succeeds but the user wants additional answers, relaxation is applied to find an unordered set of new related queries, $\{R_1, \dots, R_n\}$.
 - (b) If the query violates integrity constraints, the heuristic rules in Section 6 are applied to obtain an ordered list of relaxed queries, $\{R_1, \dots, R_n\}$.
 - (c) If the query fails and has no integrity constraint violations, false presuppositions are sought, and the rules in Section 6 are applied to obtain an ordered list of relaxed queries, $\{R_1, \dots, R_n\}$.
3. The set of user constraints are applied to each new relaxed query to form $R'_1, \dots, R'_m$. Any R_i that violates a user constraint is eliminated.
4. The remaining relaxed queries are either presented to the user so that the user can decide what to do next, or the highest priority relaxed query is selected for relaxation.

8 Summary

The process of query relaxation as introduced in [7] provides a user with a means to automatically identify new queries that are related to the user's original query. However, for large databases, many relaxations may be possible. The methods to control and restrict the relaxation process introduced in this paper focus the relaxation process and make it more efficient. We reviewed the process of relaxation and how to express user restrictions through user constraints. Then we showed how to restrict the relaxation process in two ways by applying the user constraints to relaxed queries. Next, we gave a set of heuristics for ordering a set of relaxed queries that are obtained from a query that has misconceptions. Finally, we presented how the relaxation process works together with user constraints and the heuristic rules.

The relaxation method, the user constraint method, and the misconception identification method have been implemented as separate cooperative devices within a Cooperative Answering MetaInterpreter called CARMIN [8]. The interaction of the two methods and the ordering heuristics described here are being implemented within CARMIN. The methods described in this paper complement a set of techniques for adding coherence and cohesion to a cooperative answer to a deductive database query [6].

The cooperative behaviors in CARMIN and the interactions of two of those behaviors as described in this paper reflect cooperative activity between users and knowledge-based systems. In general, we consider cooperative activity to consist of one entity helping another entity to achieve a goal or set of goals. In the context of knowledge-based systems, the goal of the user is to gain particular information by posing queries and receiving answers. The goal of the system is to answer not just the literal query but the intended query.

It is worthwhile to consider how these behaviors might apply in the context of two or more knowledge-bases exchanging information in order to achieve some goal. We are starting to investigate the generalization of the cooperative behaviors in CARMIN from user/system interactions to intrasystem interactions. Allow us to describe and discuss several scenarios at a hypothetical level.

- Consider two processors each with a separate knowledge base. In a situation in which one processor sends a subquery to another processor for solution, if integrity constraint violations are detected at the second processor, the misconception should be returned to the first processor. Whether the first processor should incorporate the misconception into the answer or use the information to “learn” that the second processor cannot answer queries that violate the IC in question, is an open question.

Detection of integrity constraint violations becomes a means to determine whether or not a particular processor can provide an answer to a subquery. This may be useful for guiding communication between processors or for building a more informative answer.

- Suppose that a first processor sends a query to a second processor that is not available and that the second processor is the only one with the definite information to solve the query. If a third processor could solve a relaxed version of the query, then the overall system could at least provide related answers to the original query.

Relaxation becomes a means to redirect subqueries.

- Suppose that a first processor divides a query into a local query and a remote query and that the link to the remote processor is slow. The first processor could formulate a “user constraint” to eliminate the remote portion and then include in the answer to the theory of user constraints [6].

User constraints become a means to return intermediate answers quickly and to characterize them so that the user knows the answer is incomplete.

In each of these scenarios, the critical point is that the cooperative behavior between two entities allows them to provide answers that they could not provide without cooperation.

Acknowledgements

This paper describes research supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38. I would like to thank Jack Minker for his guidance in the cooperative answering work and Parke Godfrey, Lev Novik, and Jorge Lobo for their insightful discussions on the material presented in this paper.

References

- [1] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–160, 1984.
- [2] U. Chakravarthy, J. Grant, and J. Minker. Logic based approach to semantic query optimization. *ACM Transactions on Database Systems*, 5(2):62–207, June, 1990.
- [3] W. W. Chu, Q. Chen, and R. Lee. Cooperative query answering via type abstraction hierarchy. Technical report, University of California at Los Angeles, October 1990.
- [4] F. Cappens and R. Demolombe. How to recognize interesting topics to provide cooperative answering. *Information Systems*, 14(2):163–173, 1989.
- [5] T. Gaasterland. *Cooperative Answers for Database Queries*. Ph.D. thesis, University of Maryland, Department of Computer Science, College Park, October 1992.
- [6] T. Gaasterland. Cooperative explanation in deductive databases. In A. Quilici, editor, *Working Notes: Symposium on Producing Cooperative Explanations*, AAAI Spring Symposium Series, Stanford University, Palo Alto, California, 1992.

- [7] T. Gaasterland, P. Godfrey, and J. Minker. Relaxation as a platform for cooperative answering. *Journal of Intelligent Information Systems*, 1993. To appear.
- [8] T. Gaasterland, P. Godfrey, J. Minker, and L. Novik. CARMIN: A Cooperative Answering MetaInterpreter. Technical Report to appear, University of Maryland, Department of Computer Science, College Park, Maryland 20742.
- [9] T. Gaasterland, J. Minker, and A. Rajasekar. Deductive Database Systems and Knowledge Base Systems. In *Proceedings of VIA 90*, Barcelona, Spain, October 1990.
- [10] A. Gal. *Cooperative Responses in Deductive Databases*. Ph.D thesis, University of Maryland, 1988.
- [11] H. Gallaire and J. Minker, editors. *Logic and Databases*. Plenum Press, New York, New York, April 1978.
- [12] J. M. Janas. On the feasibility of informative answers. In H. Gallaire, J. Minker, and J. M. Nicolas, editors, *Advances in Database Theory: Volume 1*, pages 397-414. Plenum Press, 1981.
- [13] K. McCoy. Reasoning on a highlighted user model to respond to misconceptions. *Computational Linguistics*, 14:52-63, September 1988.
- [14] J. Minker and A. Gal. Producing cooperative answers in deductive databases. In P. Saint-Dizier and S. Szpakowics, editors, *Logic and Logic Grammar for Language Processing*. L. S. Howard, Ltd., 1990.
- [15] A. Motro. SEAVE: A Mechanism for Verifying User Presuppositions in Query Systems. *ACM Transactions on Office Information Systems*, 4(4), October 1986.
- [16] M. Pollack. Generating expert answers through goal inference. Technical report, SRI International, Stanford, Calif., October 1983.
- [17] R. Reiter. Deductive question answering on relational databases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 149-177. Plenum Press, New York, 1978.
- [18] C. Shum and R. Muntz. Implicit representation for extensional answers. In L. Kershberg, editor, *Expert Database Systems*, Tysons Corner, Va, 1987.
- [19] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, 1986.

END

**DATE
FILMED**
9/17/93