# Query Generalization: A Method for Interpreting Null Answers

Amihai Motro

Department of Computer Science
University of Southern California
Los Angeles, CA 90089

## Abstract

A frustrating event in the course of interaction with a database management system is query failure: a query is submitted to the database, but instead of the anticipated printout, the system responds with an empty set of data items. While such null answers are always correct from a technical point of view, quite often they are unsatisfactory. Most efforts to deal with this problem have been in the context of natural language interfaces. In this paper we outline a simple mechanism for handling query failures in a typical database management system, which has only formal language interfaces, and only limited knowledge on the data it stores (such as types and relationships). The mechanism is demonstrated with the Loose Structure data model, which adopts an object-oriented, logic-based approach. Its principles, however, may be implemented with other data models and user interfaces.

*Expert Database Systems; Larry Kerschberg, Editor.* 

## 1. Introduction

A frustrating event in the course of interaction with a database management system is *query failure*: a query is submitted to the database, but instead of the anticipated printout, the system responds with an empty set of data items (a *null answer*).

Query failure occurs when no data items satisfy the condition expressed in the query. While such null answers are always correct from a technical point of view, quite often they are unsatisfactory. Many null answers reflect undetected errors in the queries. Obviously, if such errors are recognized, interaction with the database will be improved. But even when null answers reflect genuine failures, more informative answers can be provided.

In this paper we outline a mechanism for providing *interpretations* for null answers. Each null answer is accompanied with an interpretation, which classifies the failure as either a user error or as a genuine null. In the former case it attempts to point out the error; in the latter case it attempts to delimit the scope of the failure and provide partial answers.

In interpreting failures the database management system demonstrates cooperative behavior and thus improves the interaction between the user and the database system. Improving man-machine interaction through cooperative systems is an active research area, where much of the effort focuses on the interface between natural language users and database management systems. An example of cooperative behavior is monitoring of changing data[1]. When a query generates a negative answer that may possibly change to a positive answer in the future, a cooperative system will offer to monitor the situation and inform the user when the change occurs. For example, the query "Has flight 909 landed yet?" will be answered with "No. Shall I let you know when it does?" Additional examples of cooperative behavior may be found in [3, 4, 5, 2, 7]. Of particular relevance to this paper is the system CO-OP, designed by Kaplan [3], which implements some of the conventions of cooperation in human conversation. These include *corrective*

[1]See the paper by Eric Mays elsewhere in this book.

responses, that detect erroneous presuppositions, and *suggestive* responses, that anticipate follow up queries. CO-OP was designed for natural language interaction, and relied on domain specific knowledge. In contradistinction, our approach here is to obtain similar effects in a typical database management system, which has only formal language interfaces, and only standard knowledge on the data it stores (e.g. types, attributes, relationships). Thus, the mechanism outlined here is less ambitious, but also less expensive. It is envisioned as a *help* key that users may press after a query fails. Our approach is similar to that adopted by Corella et al [1]. But while we share many basic principles, the research reported here takes a more general approach. For example, a limitation of Correla's technique is that it handles only a limited class of queries: conjunctive queries with a single variable, where each conjunct specifies a simple matching requirement. Also, since their data model is extremely simple, only limited analysis is possible. This analysis cannot detect, for example, user misconceptions about the *structure* of the data.

The next section discusses the sources of null answers and classifies them into two types. This classification is the basis for the failure interpretation mechanism, described in Section 3. While the principles of this mechanism hold for all data models and user interfaces, this mechanism is implemented most naturally within a logic-based, object-oriented data model, such as the Loose Structure data model [6]. This model is summarized in Section 4, and the details of the failure interpretation mechanism for this particular model are described in Section 5. Section 6 concludes with a short summary and some remaining problems.

## 2. Sources of Null Answers

A query submitted to a database can either be *rejected* or *evaluated*. However, a query that is evaluated is not necessarily "correct": a query may be acceptable to the system but somehow not model correctly the intentions of the user. Such queries are said to have *mistakes*; queries that are rejected by the system are said to have *errors*. The distinction between errors and mistakes is based, therefore, on the ability of the system to detect a problem in the query. Queries that contain errors or mistakes will be referred to collectively as *incorrect*.

### 2.1. Null Answers that Are Results of Mistakes

Experience shows that quite often null answers are interpreted by users as indication of mistakes. This happens when the user believes that the query should have matched some data, and therefore concludes that something went wrong. The user's reaction then is to examine the query for a possible cause: perhaps a misspelled name, or an operation used incorrectly, or simply insufficient understanding of the database. Clearly, the user interprets such answers as "mistake messages". Needless to say, null answers are very unsatisfactory mistake messages. Consequently, the user may end up trying different versions of this query in an attempt to understand the reason for its failure.

In contradistinction, when the database management system actually detects an incorrect query, it rejects it with an informative message. After considering such a message the user can correct the query promptly. It follows that a database management system that rejects a query is superior, in terms of cooperation, to systems that evaluate this query.

A principal source of mistakes in queries is *misconception*: as users form queries on the basis of their conceptions of the data stored in the database[2], inaccurate conceptions may lead to queries that do not implement the intentions of the users correctly. Other sources of mistakes are insufficient command of the query language, misspellings, etc.

While mistakes may result in non-null answers as well, because null answers tend to occur more frequently, and since they carry the least information on the error that caused them, analysis of null answers is a promising technique.

Mistakes occur more frequently when the organization of the data is not known to the user (or when the data model does not enforce organization). In addition, naive database users usually make more mistakes. In general, user interfaces based on *procedural* queries, in which the user specifies action-by-action how to obtain the target data, are less susceptible to such situations, as the user can monitor the progress of the evaluation

[2]These conceptions usually reflect they way these users perceive the real life environment which is modelled with this database.

of the query. On the other hand, user interfaces based on *specification* queries, in which the user qualifies the target data by a condition, tend to detect less mistakes, returning more null answers instead.

### 2.2. Genuine Null Answers

Even *genuine* null answers, i.e. null answers that are not the result of mistakes, are sometimes disturbing, as often, the information they provide amounts to a "shrug".

This is in contrast with human behavior, where a negative answer is usually accompanied with some kind of additional information. For example, when presented with the question "Do you know of a nearby supermarket with a good selection of wines at low prices?", a person without a satisfactory answer may still respond with something like "No, but I know several that are not around here".

In general, such answers are helpful, as they inform the person asking that the question was indeed meaningful, and that its failure was genuine, not the result of some misconception. More importantly, such answers tend to delimit the scope of the failure; in the previous example a negative answer could have been caused by a more fundamental inability to satisfy the question (it may be that the person asked does not even know of any nearby supermarket) and the person asking could be left wondering what is the real cause for the negative answer. Finally, sometimes such answers anticipate subsequent questions, as often negative answers trigger follow up questions.

Similarly, a database management system can be programmed so that genuine null answers are always accompanied with interpretations that achieve the very same benefits: assure the user that the query was meaningful, delimit the scope of its failure, and anticipate possible future queries.

## 3. The Failure Interpretation Mechanism

The failure interpretation mechanism attempts to detect mistakes in failed queries (they become errors...). Failed queries are thus classified into mistakes and genuine failures. This classification is then used to produce an interpretation of the failure (and

provide further assistance). The method used to detect mistakes can be regarded as an extension of methods already in use. Following is an examination of the source of query rejection in typical database management systems.

One obvious cause of query rejection is *syntactic*: the query does not obey the syntax rules of the query language. Other rejections may be described as *schematic*. For example, assume a relational database and the query "select all tuples from relation **R** where attribute **A** has value **v**". If there is no relation **R** in the database, or relation **R** does not include attribute **A**, the query is rejected at the "schema" level. For an example of a rejection which is neither syntactic nor schematic, assume a functional database with the function **WORKS-FOR** from **EMPLOYEE** into **PROJECT**, and the query "list all values of **PROJECT** for **EMPLOYEE=Smith** by the function **WORKS-FOR**". If **Smith** is not in the domain **EMPLOYEE**, the query is rejected after the "content" of the database had been examined.

Except perhaps for syntax-based rejections, these examples of rejections can all be classified as misconceptions. To qualify their target data, all queries provide certain information, such as navigation paths, names of relationships, names of data items, and so on. This information reflects the user's conception of the database. During query processing the validity of this conception is checked against the actual database, and, if found incorrect, the system is able to reject the query with an appropriate message. The previous queries, for example, were rejected because, contrary to the user's conceptions, there is no relation **R**, relation **R** does not have attribute **A**, **Smith** is not an **EMPLOYEE**, etc.

Assume now a database on currently available recordings of music, with information on the composer, the title and the artist. Consider the query "List all different recordings of Chopin's Piano Concerto No. 3 by Rubinstein". As Chopin did not compose a third piano concerto, there will be no such recordings. Still, most database management systems will simply return a null answer without rejecting the query, although this query

too reflects a misconception.[3] By detecting such misconceptions many queries, that would otherwise fail, can be rejected with appropriate messages.

### 3.1. Definitions

A *database* is a set of values. A *(retrieval) query* $Q$ against a database $D$ is a function that evaluates to a subset of $D$, called the *answer* to $Q$ and denoted $Q(D)$. A query $Q$ *fails* if $Q(D)$ is the empty set; $Q$ is also said to have a *null answer*. The mechanism that evaluates queries is part of the *database management system*.

Consider the queries "List all different recordings of Chopin's Piano Concerto No. 2" and "List all different recordings of Chopin's Piano Concerto No. 2 by Rubinstein". Clearly, the former query is more general than the latter. This query relationship is defined as follows: given two queries $Q$ and $Q'$, $Q$ is *more general* than $Q'$ if $Q(D)$ contains $Q'(D)$. As it is based on containment, *generalization* is a partial order among the different queries.

As another example, consider the query "**1** List all female employees who earn more than $30000". Some more general queries are "**2** List all female employees who earn more than $20000" or "**3** List all employees who earn more than $30000". The latter can be generalized further by the query "**4** List all employees". "**5** List all persons" is still more general. The most general, of course, is "**6** List everything". This example demonstrates some different methods to generalize queries: weaken a condition (from **1** to **2**), remove a condition altogether (from **1** to **3**, or from **3** to **4**), and substitute a more general concept (from **4** to **5**, or from **5** to **6**).

The database management system incorporates a *query generalizor*. Given a query $Q$ this component generates a set of queries $\mathbf{Q}$, all more general than $Q$, and none related through generalization relationships. Queries in $\mathbf{Q}$ are all *minimally* more general than

[3] Admittedly, to identify this misconception positively, it must be assumed that the database includes complete information on composers and their compositions. But even when this so-called "closed world assumption" cannot be made, it should always be possible to reject this query with a message "there are no recordings of Chopin's Piano Concerto No. 3 by *any* artist".

$Q$; that is, all other queries more general than $Q$ that can be generated by this mechanism, are also more general than some query in $\mathcal{Q}$.

To perform this task the query generalizor incorporates various strategies, all based on information stored in the database. Possible strategies are discussed in more detail in a Section 5. The query generalizor is the main component of the *failure interpretation mechanism.*

### 3.2. Principles

Consider again the query "List all female employees who earn more than $30000". We can assume the user who formulated it believes there may be female employees who earn more than $30000 (otherwise why bother ask). It is reasonable to assume that this user is even more confident that some employees (either males or females) indeed earn more than $30000. Furthermore, this user is quite certain that employees earn salaries. In other words, while a query that is presented to a database conveys the conceptions of the user about the database with some uncertainty, its more general queries convey user conceptions with a greater degree of confidence. This suggests a simple heuristic to estimate the conceptions of a user from the query.

This heuristic, which assumes most queries risk only minimal uncertainty, states that *while users expect their queries may possibly have null answers, they tend to be confident that every more general query would not fail.*

Under this heuristic, the generalizations of a query become indicators of the conceptions of the user. When a query fails, these conceptions can be tested by evaluating the generalizations. Each generalization that fails suggests a misconception. If all succeed, the original failure is interpreted as a genuine failure, and the answers obtained are offered as "partial answers" (i.e. "the best the system could do to satisfy the query").

Assume $Q'$ and $Q''$ are both generalizations of $Q$, but $Q''$ is more general than $Q'$. If both succeed, then the partial answer returned by $Q'$ is better. If both fail, then the

misconception indicated by $Q''$ is stronger. This suggests that after a failure we should look for *minimal generalizations that succeed*, or *maximal generalizations that fail.*

The failure interpretation mechanism applies these principles in the following way. When a query $Q$ fails the query generalizor is called to generate its set $\mathcal{Q}$ of minimally more general queries, and this set is evaluated. Then

- If all queries in $\mathcal{Q}$ succeed ($Q$ is a maximal failure), then the failure of $Q$ is genuine, and the answers to the generalizations are offered as partial answers. The interpretive message **partial answers available (yes/no)** accompanies the null answer. Responding **yes** the user gets a list of possible partial answers.

- If some queries in $\mathcal{Q}$ fail, then the failure of $Q$ is due to misconceptions. In a recursive process, each of the failed queries in $\mathcal{Q}$ is generalized, until a set of maximal failures is obtained (every query in this set fails, but all its generalizations succeed). The interpretive message **possible misconceptions (yes/no)** accompanies the null answer. Responding **yes** the user gets a list of the possible misconceptions.

This procedure can be summarized as follows: when a query fails, its associated set of maximal failures is derived. If this set contains only the query itself, then it is a genuine null; otherwise, each maximal failure describes a misconception. Therefore, the interpretation of query failure is always provided by its associated maximal failures. This leads to the conclusion that *only maximal failures are significant.*

## 4. The Loose Structure Data Model

The Failure Interpretation mechanism can be implemented with different data models and user interfaces. Its main component, the query generalizor, should employ strategies that take advantage of the features of the particular data model. An object-oriented, logic-based data model is a convenient environment to demonstrate a very general strategy. The Loose Structure data model provides such an environment, and will be used here.

While most data models emphasize structure, thereby requiring substantial investment in their design and maintenance (update and reorganization), the Loose Structure model

permits databases that are unstructured heaps of facts. These facts can generate further facts through inference rules, and are monitored by integrity constraints.

The appropriate mechanism for retrieval from a Loosely Structured database is *browsing*: exploratory searching that does not assume any knowledge about the database and its organization (or even the very existence of such organization). While the Loose Structure model supports a standard retrieval language based on predicate logic, this language is intended primarily to help formalize different browsing techniques.

This model is particularly suitable for modelling environments which are subject to constant evolution (or of which our conception is continuously evolving). It is a natural tool for modelling those environments that do not lend themselves to "massive" classifications. In general, this alternative model can be employed whenever we prefer to trade retrieval efficiency, for minimal investment in organization.

The Loose Structure model may be classified as a binary, object-oriented, logic-based data model. However, as it formalizes the notion of a unit of information (a fact) and allows one to describe such units "one by one", it does not require "modelling", as this activity is usually understood. A brief description follows. For more details see [6].

The most basic units of data are *entities*. Let $\mathcal{E}$ be a universe of distinctly named entities. This universe is partitioned into three sets, which are not necessarily disjoint: a set of *types* $\mathcal{T}$, a set of *tokens* $\mathcal{V}$ and a set of *relationships* $\mathcal{R}$. Relationships between entities are represented with *facts*, which are pairs of types or tokens named with a relationship. Thus, facts are elements of $(\mathcal{T} \cup \mathcal{V}) \times \mathcal{R} \times (\mathcal{T} \cup \mathcal{V})$. Some examples of facts are *(JACK,BROTHER-OF,JILL)*, *(JACK,$\in$,BOY)* and *(BOY,LIKES,DOG)*[4].

In particular, $\mathcal{V}$ includes all the numbers, and $\mathcal{R}$ includes the relationships $=$, $\neq$, $<$, and $>$. We assume that for every two different number entities *N1* and *N2* exactly one of the following facts is included: either *(N1,<,N2)* or *(N1,>,N2)*, depending on whether *N1* is smaller than *N2* or not. In addition, we assume that for every two

[4]The relationship $\in$ describes membership; it is discussed in the next section.

entities *E1* and *E2* (not necessarily numbers) exactly one of these two facts is included: either *(E1,=,E2)* or *(E1,≠,E2)*, depending on whether *E1* and *E2* are identical or not.

When an entity of a fact is substituted with a *variable* the result is a *template fact*. A template fact is a restriction on its variables to entities that form existing database facts. Template facts are then used to construct formulas. A *formula* is constructed from template facts using negation, conjunction and disjunction operations, and universal and existential quantifiers.[5]

Closed formulas (i.e. all variables are bound) are used to express *integrity constraints*. For example, the following constraint guarantees the transitivity of the $<$ relationship:

$$(\forall x)\,(\forall y)\,(\forall z)\,(((x,<,y) \wedge (y,<,z)) \Rightarrow (x,<,z)).$$

As another example, the constraint that a child cannot be older than his father is expressed with the formula:

$$(\forall p_1)\,(\forall p_2)\,(\forall a_1)\,(\forall a_2)\,(((p_1,\in,PERSON) \wedge (p_1,AGE,a_1) \wedge (p_2,\in,PERSON) \wedge (p_2,AGE,a_2) \wedge (p_1,FATHER\text{-}OF,p_2)) \Rightarrow (a_1,>,a_2)).$$

Formulas can also be used to express inference. *Inference rules* require closed formulas of the form: $(u)\,(v \Rightarrow w)$, where $v$ is a subformula, $w$ is a template and $u$ is universal quantification. For example, the following rule inserts every student with GPA greater than 3.5 into the honor category:

$$(\forall x)\,(\forall y)\,(((x,\in,STUDENT) \wedge (x,GPA,y) \wedge (y,>,3.5)) \Rightarrow (x,\in,HONORS))$$

If the database includes the facts *(JOHN,∈,STUDENT)* and *(JOHN,GPA,3.7)*, then, using this rule the fact *(JOHN,∈,HONORS)* may be inferred. An inference rule may therefore be regarded as a collective representation of facts.

Finally, a Loosely Structured database is a set of facts $\mathcal{P}$, a set of inference rules $\mathcal{I}$, and a set of integrity constraints $\mathcal{C}$, such that the closure of $\mathcal{P}$ under $\mathcal{I}$ does not falsify any of the constraints of $\mathcal{C}$.

Formulas are also instrumental in retrieval. A formula with free variables is a *query*.

[5] Although not necessary, implication is often used for clarity.

Let $Q$ be such a query, and let $x_1,...,x_n$ be its free variables. The *value* of $Q$, denoted $\{Q\}$, is the set of tuples $(c_1,...,c_n)$ that satisfy it.

For example, the query

$$Q(x) = (x,\in,BOY) \wedge ((\exists y)\ (y,\in,GIRL) \wedge (x,BROTHER\text{-}OF,y))$$

lists all boys who have a sister. Assuming a database that includes all previously mentioned facts, the value of $Q$ includes the entity *JACK*.

## 5. Query Generalization in the Loose Structure DBMS

Through the use of inference rules and integrity constraints the Loose Structure model permits the database designer to introduce as much structure as desired. In this section we discuss several rules that are necessary for proper query generalization.

The fundamental relationship between tokens and types is *membership*: a token is an instance of a type. To express this relationship with facts a special entity $\in$ is used. Example are $(JACK,\in,GIRL)$ and $(2.5,\in,REAL\text{-}NUMBER)$.

A frequent relationship between types is *generalization*: the concept described by one type is more general than the concept described by the other type. To express this relationship with facts a special entity $\prec$ is used. Examples are $(GIRL,\prec,FEMALE)$ and $(REAL\text{-}NUMBER,\prec,NUMBER)$.

A third basic relationship is between relationships and it is called *consequence*: one relationship always implies another relationship between the same two entities. To express this relationship with facts a special entity $\Rightarrow$ is used. Examples are, $(LOVE,\Rightarrow,LIKE)$ and $(<,\Rightarrow,\neq)$.

The relations $\in$, $\prec$ and $\Rightarrow$ must be *disjoint* and their union must be *cycle-free*: two entities should not be related via more than one of these relationships, and an entity should not be related to itself through a chain of memberships, generalizations and consequences.

The following rules express part of the semantics of membership, generalization and

consequence (existential quantifiers, as well as some parenthesis, are omitted):

$(x,\in,a)\ (a,\prec,b) \Rightarrow (x,\in,b)$,
$(x,r,y)\ (r,\Rightarrow,r') \Rightarrow (x,r',y)$,
$(a,\prec,b)\ (b,\prec,c) \Rightarrow (a,\prec,c)$, and
$(a,\Rightarrow,b)\ (b,\Rightarrow,c) \Rightarrow (a,\Rightarrow,c)$.

The first rule ensures that if a token is a member of a type, then it is also a member of every more general type. The second ensure that when two entities maintain a relationship, they also maintain every consequential relationship. The last two rules state the transitivity of generalizations and consequences.

A fact such as *(BOY,LOVES,DOG)* could have different meanings. For example, it could mean that *every* boy loves *every* dog, or that *every* boy loves *some* dogs, or that *some* boys love *some* dogs, etc. The desirable semantics can be enforced with appropriate inference rules. We adopt the weakest interpretation ("*some-some*") as the "standard" semantics. According to this interpretation, if *(JACK,LOVES,FIDO)* and *(FIDO,∈,DOG)* are facts, then *(JACK,LOVES,DOG)* should also be a fact. If *(JACK,∈,BOY)* is a fact then also *(BOY,LOVES,DOG)*. If *(BOY,≺,PERSON)* then also *(PERSON,LOVES,DOG)*. If *(DOG,≺,ANIMAL)* then also *(PERSON,LOVES, ANIMAL)*. Similarly, if *(LOVES,⇒,LIKES)* then also *(PERSON,LIKES,ANIMAL)*. These inferences are described by the following set of rules:

$(x,r,y) \wedge (x,\in,x') \Rightarrow (x',r,y)$,
$(x,r,y) \wedge (y,\in,y') \Rightarrow (x,r,y')$,
$(x,r,y) \wedge (x,\prec,x') \Rightarrow (x',r,y)$,
$(x,r,y) \wedge (y,\prec,y') \Rightarrow (x,r,y')$ and
$(x,r,y) \wedge (r,\Rightarrow,r') \Rightarrow (x,r',y)$.

The query generalizor in the Loose Stucture database management system receives a query $Q$ that failed, and returns a set $\mathcal{Q}$ of queries that are minimal generalizations of $Q$. If $Q$ specifies a total of $n$ entities, then $\mathcal{Q}$ contains $n$ queries: each is obtained by applying the generalization procedure to a different entity of $Q$. The mechanics of this procedure are different for types (or relationships) and for tokens.

### 5.1. Generalizing Types and Relationships

The previous rules guarantee that if a type specified in a query is substituted with a more general type, or if a relationship is substituted with a consequence relationship, a more general query is obtained. The generalization procedure substitutes a type or a relationship by the immediate generalization or consequence (or a conjunction of substitutions, if several immediates exist). Thus, among all more general queries that may be formed by substitutions this procedure selects the minimal one.

Consider this query to list all beer lovers:

$$Q(x) = (x,\in,PERSON) \wedge (x,LOVES,BEER),$$

and assume the following facts represent the closest generalizations of the *PERSON*, *BEER* and *LOVES*:

$(PERSON,\prec,LIVING\text{-}THING)$,
$(BEER,\prec,ALCOHOLIC\text{-}BEVERAGE)$ and
$(BEER,\prec,FERMENTED\text{-}BEVERAGE)$.
$(LOVES,\dot{\Rightarrow},LIKES)$,

The query generalizor produces three different queries:

$$Q_1(x) = (x,\in,LIVING\text{-}THING) \wedge (x,LOVES,BEER),$$
$$Q_2(x) = (x,\in,PERSON) \wedge (x,LIKES,BEER) \text{ and}$$
$$Q_3(x) = (x,\in,PERSON) \wedge (x,LOVES,ALCOHOLIC\text{-}BEVERAGE) \wedge (x,LOVES,FERMENTED\text{-}BEVERAGE).$$

These queries return, respectively, all living things that love beer, all persons who merely like beer, and all persons who love beverages which are alcoholic and fermented.

When queries produced by the query generalizor fail, the generalization procedure is applied again. In the case of type or relationship generalization, the very same process is repeated.

### 5.2. Generalizing Tokens

Each token specified in $Q$ required *strict* matching by a database token. The controlled relaxation of this requirement is the principle that governs the generalization procedure for tokens. For each token of $Q$ a *neighborhood* is defined, and in the more general query matching is satisfied by *any* entity in this neighborhood. Defining a

neighborhood of a token is like defining an ad-hoc type to which the token belongs.

Assume $Q$ specifies a token $E$. Let $N_E$ designate the neighborhood of $E$ and let $y$ be an existential variable not used in $Q$. The generalization of $Q$ on $E$ is obtained by substituting $E$ with $y$ and requiring that $y \in N_E$. Thus, if a query that requests a listing of all programmers who know PASCAL fails, and ALGOL, PL/1 and ADA are in PASCAL's neighborhood, then a more general query would be to list all programmers who know at least one of these languages. The more general query is then satisfiable by any other instance of this type. Obtaining a satisfactory neighborhood $N_E$ is the major point to consider in this process.

Every database fact can be regarded as a characterization of each of its participating entities. Thus, two database entities that appear in similar facts (i.e. facts that are identical except for these two entities) share a common characterization. The more such common characterizations exist, the more similar the two entities are perceived. This leads to the following definitions of *neighborhood*, *immediate neighborhood* and *relevant neighborhood*.

Let $X$ be an entity, and let *(X,Y,Z)* be a fact. The set of all database entities that match the query $Q(x) = (x,Y,Z)$ is called a *neighborhood* of $X$. An intersection of neighborhoods is also a neighborhood.

By intersecting all the neighborhoods to which $X$ belongs we obtain a neighborhood of entities that share all the characterizations of $X$. As an example, if *(JOHN,∈,CITIZEN)*, *(JOHN,DRINKS,BEER)* and *(JOHN,LIKES,MARY)* are database facts, then *JOHN* belongs to three neighborhoods: the citizens, the beer drinkers, and those who like Mary. The intersection of these neighborhoods is the set entities that are most like John: the beer-loving citizens who like Mary.

Depending on the facts which describe an entity, the size of intersection neighborhoods may vary widely. If an entity is described by a unique fact, such as *(JOHN,TELEPHONE,743-6710)*, then the intersection neighborhood will include itself

only. For the purpose of generalization, neighborhoods that do not encompass any additional entities are useless.[6] Consequently, when intersecting neighborhoods to obtain the set of entities most similar to the given entity, single element neighborhoods should not participate. While this, of course, does not guarantee an intersection with more than a single element, single element neighborhoods formed this way are more plausible: an entity is "truly unique" when no other entities share all its *non-identifying* characteristics. The *immediate neighborhood* of $X$ is the intersection of all the multiple element neighborhoods to which $X$ belongs.

Assume now a database on apartments available for rent in Los Angeles County is presented with the following query to list all apartments in Santa Monica.

$Q(x) = (x,\in,APARTMENT) \wedge (x,LOCATED\text{-}IN,SANTA\text{-}MONICA).$

When $Q$ fails, it may be desirable to generalize it on the entity *SANTA-MONICA*. However, this entity may have many characterizations in the database, only some of which are relevant to its role in this query as apartment location (for example, the fact *(SANTA-MONICA,MAYOR,CLARK)* is a characterization of Santa Monica which is irrelevant to its present role). The role of *SANTA-MONICA* in $Q$ is defined as *LOCATED-IN*, which is the relationship in the template in which it appears. Assume that the following characterizations of *SANTA-MONICA* are relevant to the role *LOCATED-IN*:

*(SANTA-MONICA,RENT-CONTROL,STRICT),*
*(SANTA-MONICA,POLLUTION-LEVEL,LOW)* and
*(SANTA-MONICA,BEACH-ACCESS,YES).*

Using a relationship called *relevant*, denoted @, this information is stored in three facts:

*(LOCATED-IN,@,RENT-CONTROL),*
*(LOCATED-IN,@,POLLUTION-LEVEL)* and
*(LOCATED-IN,@,BEACH-ACCESS).*

When the neighborhood of *SANTA-MONICA* is formed, *LOCATED-IN* is taken as its role, and the relationship *relevant* is used to identify *RENT-CONTROL*, *POLLUTION-LEVEL* and *BEACH-ACCESS* as relevant characterizations. The relevant neighborhood of *SANTA-MONICA* then includes all locations with the same

[6] And neighborhoods that are extremely wide are ineffective.

kind of rent control, pollution level and beach access. Consequently, the query generalizor outputs the following query:

$$Q'(x) = (x,\in,APARTMENT) \wedge ((\exists y)\ (x,LOCATED\text{-}IN\ y) \wedge (y,RENT\text{-}CONTROL,STRICT) \wedge (y,POLLUTION\text{-}LEVEL,LOW) \wedge (y,BEACH\text{-}ACCESS,YES)).$$

It is easy to verify that queries generated by immediate or relevant neighborhoods are indeed generalizations of the input queries (i.e. their answers contain the answers to the input queries). Of course, relevant neighborhoods can be formed only when the role of this entity is specified (i.e. not a variable), and the relationship @ provides some relevant characterizations. Otherwise, immediate neighborhoods are formed. Further generalization of a query that underwent immediate or relevant neighborhood substitution is done best by expanding the neighborhood, through removal of one of the characterizations used to form the neighborhood (this corresponds to the deletion of one of the conjuncts introduced in the substitution). Thus, if the neighborhood was single element, it immediately goes into further generalization.

### 5.3. Generalizing Numbers

Consider the following query to list all employees with four children:

$$Q(x) = (x,\in,EMPLOYEE) \wedge (x,NO\text{-}OF\text{-}CHILDREN,4)$$

and assume generalization on the entity *4* is attempted. In the case of number tokens, neighborhoods are readily available in the form of symmetric intervals around the number. A possible generalization of $Q$ is:

$$Q'(x) = (x,\in,EMPLOYEE) \wedge (\exists y) \wedge (x,NO\text{-}OF\text{-}CHILDREN,y) \wedge (y,\geq,3) \wedge (y,\leq,5))).$$

$Q'$ relaxes the requirement that the number of children must be *4*; instead, this number should be in the interval *[3,5]*.

In this example, the lower and upper bounds chosen to replace *4* were the closest integers that create an interval with more than *4* itself. This seems suitable when substituting a value that represents number of children; when substituting, say, yearly salary, broader intervals should be used. For a meaningful substitution, the query generalizor looks for a fact that describes the appropriate *step*. This step relationship is

denoted with the symbol $\Delta$. Thus, facts such as *(NO-OF-CHILDREN,$\Delta$,1)*, *(SALARY,$\Delta$,1000)*, or *(BUDGET,$\Delta$,10%)* describe the steps that should be used in enlarging the scope of a query involving this attribute. If a step fact is not available, then a system constant (such as 10%) may be used.

If the relationship involving the number is an arithmetic comparator, such as $\leq$ or $=$, then a simple reduction may be performed. Consider this input query to list all employees with salaries over $30,000:

$$Q(x) = (x,\in,EMPLOYEE) \wedge (\exists y)\,((x,SALARY,y) \wedge (y,\geq,30000))$$

and assume *(SALARY,$\Delta$,1000)*. The previous procedure generates the output query

$$Q'(x) = (x,\in,EMPLOYEE) \wedge ((\exists y)\,(x,SALARY,y) \wedge (\exists z)\,((y,\geq,z) \wedge (z,\leq,31000) \wedge (z,\geq,29000))).$$

$Q'$ may be reduced to

$$Q''(x) = (x,\in,EMPLOYEE) \wedge (\exists y)\,((x,SALARY,y) \wedge (y,\geq,29000).$$

Since we assume that all valid mathematical facts are present, all queries generated by substituting numbers with intervals around these numbers are generalizations of the original queries. When further generalization is necessary, queries that were generalized by number substitution go into the very same process again.

## 6. Conclusion

Whether they reflect mistakes or are genuine failures, null answers are rarely satisfactory. By a simple analysis of null answers, we showed how every null answer can be provided with an interpretation.

We defined the concepts of query generalization and maximal failure, and adopted the assumption that a user who formulates a query expects all its generalizations to succeed. Consequently, these generalizations become indications of the user's conceptions. When a query fails these conceptions are tested in the computation of the maximal failures. If the query is already a maximal failure, the failure is classified as genuine, and the answers to the immediate generalizations are offered as partial answers. Otherwise each maximal failure is reported to the user as a possible misconception. In either case, an

appropriate interpretation is provided.

The principles of the failure interpretation mechanism are independent of the data model used. Each implementation exploits the features of the particular data model. For example, in a relational database system, query generalization may be done by weakening mathematical conditions, or by deleting conjuncts from queries. In data models that incorporate a type hierarchy (these are often referred to as "semantic" data models) type substitution can be performed. In object-oriented data models ad-hoc neighborhoods may be created to replace tokens that cannot be matched.

Thus, our mechanism uses information already included in the database, without requiring additional meta-information to help the system understand queries (the @ and $\Delta$ facts are possible exceptions). In this respect it can be incorporated inexpensively into available systems. For more profound reasoning about queries, additional meta-information must be stored (and appropriate mechanisms to manipulate this information must be defined).

One possible drawback of our mechanism is that the number of possible generalizations is proportional to the number of entities specified in the query. For complex queries this could limit the utility of the mechanism. A pruning strategy, perhaps based on the cardinalities of the answers, may be desirable. Another important consideration is the cost of failure interpretation. As this is mainly the cost of evaluating the follow up queries, a retrieval strategy that avoids the need to evaluate each follow up query from scratch is very desirable.

## References

[1] Corella et al.
Cooperative Responses to Boolean Queries.
In *Proceedings of the First International Conference on Data Engineering*, pages 77-85. Los Angeles, California, 1984.

[2] A. K. Joshi.
Mutual Beliefs in Question Answering Systems.
In N. Smith (editor), *Mutual Belief*. Academic Press, 1982.

[3] J. Kaplan.
*Cooperative Responses from a Portable Natural Language Data Base Query System.*
PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1979.

[4] E. Mays.
Failures in Natural Language Systems: Application to Data Base Query Systems.
In *Proceedings of the First Meeting of the American Association for Artificial Intelligence*. Stanford, CA, 1980.

[5] E. Mays et al.
Natural Language Interaction with Dynamic Knowledge Bases: Monitoring as Response.
In *Proceedings of 8-IJCAI*. Vancouver, BC, 1981.

[6] A. Motro.
Browsing in a Loosely Structured Database.
In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 197-207. Boston, Massaccusetts, 1984.

[7] B. L. Webber and E. Mays.
Varieties of user Misconceptions: Detection and Correction.
In *Proceedings of IJCAI-8*. Karlsruhe, Germany, 1983.