Extending the Relational Database Model to Support Goal Queries

Amihai Motro

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

Abstract

Requests for data can be classified roughly into two kinds: *specific* requests and *goals*. A specific request establishes a rigid qualification, and is concerned only with data that matches it precisely. A goal, on the other hand, establishes a target qualification and is concerned with data which is *close* to this target. Many database queries are in effect goals, but because database management systems lack the mechanisms to handle goals, they must be emulated with specific requests. Usually, this means the user is forced to retry a particular query repeatedly with minor modifications, until it matches something satisfactory. In this paper we show how to handle goals directly, in the environment of relational databases. We define the concept of *distance* between data values that are from the same domain and show how to incorporate it into relational systems. Based on this distance, we define three types of goals: *neighborhood goals*, *optimum goals* and *priority goals*. We then demonstrate how a typical query language can be extended to express goals, and give guidelines for implementation.

1. Introduction

Requests for data can be classified roughly into two kinds: *specific* requests and *goals*. A specific request establishes a rigid qualification, and is concerned only with data that matches it precisely. Some examples of specific requests are "How much does Jones earn?" or "When does flight 909 depart?" If the database does not contain salary information on Jones or departure time for flight 909, null answers should be returned; the user is not interested in the earnings of somebody else or in the departure time of a different flight. A goal, on the other hand, establishes a target qualification and is concerned with data which is *close* to this target. As an example, consider "List the inexpensive French restaurants in the downtown area". If there are none, a moderately priced Continental restaurant a few blocks south of downtown may have to do. Similarly, when a project is started which calls for experienced Pascal programmers with background in applied mathematics, we may want the personnel database to mention also that there is an engineer with some knowledge of Algol.

Although goals account for much of the use of databases, currently, database management systems cannot handle them directly. Consequently, they must be emulated with specific requests. Usually, this means the user is forced to retry a particular query repeatedly with minor modifications, until it matches something satisfactory. In this paper we attempt to correct this situation by showing how query languages and database management systems can be extended to handle goals directly.

To satisfy goals we need the notion of *distance* between data values. For example, which restaurant is *closer* to being an inexpensive French restaurant in the downtown area: a moderately priced Continental restaurant a few blocks south, or an inexpensive Chinese restaurant located in the heart of downtown? The need for distances becomes clearer if we examine *numeric* data. Because such data already embeds a notion of distance, goals that involve numbers are easier to handle. For example, the goal "List all courses with 4 units of credit" is satisfied first with courses of 4 units, then with courses of 3 or 5 units, and so on.

As the opening examples illustrate, a user interface to databases which is capable of handling goals appears to be more "intelligent". This is because answering questions with information which is only close to what was requested, or somehow related to it, is a common feature of human interaction. Such interaction is known as *cooperative behavior* and there have been much focus on how to improve man-machine interaction by emulating such behavior through various techniques; some examples are [5, 7, 8, 13, 6, 1, 10]. Not surprisingly, this added intelligence is made possible by including additional semantic information in the database, namely information about distances.

Our work is done in the context of the relational data model. This is primarily because of the simplicity of its structures and the advantages of a formal query language such as the relational calculus. However, the notions of distances and goals can be implemented with other data models, as well.

Our first task is to extend the relational data model to include data distances (Section 2). We describe four different options for defining distances between values of a given attribute. Distances may be based on information about values of this attribute which is already available in other relations; a new relation may be added to the database to specify the distance between every two values of this attribute; one of several standard built-in distance functions may be used; or a new distance function may be defined (if an adequate definition of distance cannot be provided, then a standard default function is used).

Distance definitions express important semantic information about the attributes. This information permits certain manipulations of the data (such as "safe" substitutions of some values by other values), and it can be used as evidence that two differently named attributes are similar. Thus, the concept of attribute distance is in the same general class as attribute *ordering* [4], and attribute *domains* [9].

Having defined the concept of distance, we then proceed to define three types of goals: *neighborhood goals*, *optimum goals* and *priority goals* (Section 3). While the definitions

are in terms of relational calculus, there should be no difficulty in extending them to other relational languages. In particular, we show how the language QUEL may be extended to express these goals.

Successful implementation of goals depends largely on the proper selection of certain parameters when the database is designed. Section 4 discusses this issue, and gives some guidelines for implementation. We conclude with a brief summary.

2. Distance

Each database relation has a designated attribute which is its *key*. Technically, this key provides a means for unique identification of tuples of the relation. Semantically, it may be regarded as the *topic* of the relation. In other words, the non-key attributes can be regarded as providing a *description* for the key attribute. For example, consider the following relation about restaurants:

RESTAURANT: NAME,TYPE,LOCATION,PRICE,RATING,TEL-NO

where NAME is the key, and let

(Le-Phoney,French,Westwood,Expensive,Awful,395-0297)

be a tuple from this relation. The last five values of this tuple *describe* the value **Le-Phoney**.

If the relation RESTAURANT is our only source of information on restaurants, then differences between restaurants must be based on differences between their descriptions. Consequently, the distance between two values of NAME is based on the difference between the quintuples that describe them, which is then defined as a combination of the distances between their corresponding components. Our definition of distance is, therefore, recursive. For example, given another RESTAURANT tuple,

(Jasmine-Gardens,Chinese,Chinatown,Inexpensive,Good,477-5113)

the distance between **Le-Phoney** and **Jasmine-Gardens** is a combination of the distances between **French** and **Chinese**, **Westwood** and **Chinatown**, **Expensive** and **Inexpensive**, etc. Similarly, the distance between **French** and **Chinese** (both values of attribute TYPE) may be based on another relation which describes the different restaurant types. In general, the distance between two values may be based on the difference between

their descriptions, in a relation in which their attribute is a key. Note that if one relation is to provide distances for an attribute of another relation, then the values of its key attribute must *include* all the values of the attribute of the other relation. Such a requirement is usually referred to as a *referential integrity constraint* [2], and the database management system must be capable of enforcing such constraints.

If a relation is not available to describe a particular attribute, we may construct a special distance relation, which will be keyed on a *pair* of such attributes, giving the distance between every two values of this attribute. In our example, distances between **Chinatown** and **Westwood** (both values of attribute LOCATION) may be obtained from another relation which specifies the distance between every two locations.

For some attributes distances may be obtained from standard built-in distance functions. For example, the distance between two numbers may be calculated by a function called NUMBER to be the absolute value of their difference. Similarly, the difference between two strings may be calculated by STRING to be 0 if they are identical and 1 otherwise.

Finally, new distance functions may be defined. For example, the STRING function mentioned above may be redefined to implement some scheme that bases the distance between two strings on their degree of similarity. Notice that the last two options are suitable for attributes that range over infinite (or very large) domains. Distance functions are computations that complement the data. They recall other computations which are incorporated into database management systems, such as database procedures for monitoring integrity constraints, or database transactions which are triggered automatically under certain predefined situations (see [3] for a review of this topic).

The option selected to define distances between values of a given attribute will be referred to as the *measure* of that attribute. In addition to the measure, each attribute is also assigned three additional parameters: a *relative weight*, a *scaling factor* and a *neighborhood radius*.

When comparing two restaurants, one may wish to ignore their different telephone numbers; or one may wish to give higher consideration to whether or not they are of the same type. For this purpose we shall associate with each attribute a *relative weight*, to indicate the relative importance of this attribute as a descriptor of the key attribute.

Usually, the attributes that participate in a description involve distances of different scales. For example, the description of an employee may involve the attributes YEARS_OF_EXPERIENCE, where distances are measured with numbers which are relatively small, and SALARY, where distances are measured with numbers that are substantially larger. Before combining these distances one may want to correct them; for example, by dividing the salary distance by 1000. For this purpose we shall associate with each attribute a *scaling factor*, to correct distances between its values, for their role here. Scaling factors have the effect of adjusting individual distances into units of comparable significance[1].

Given a value of some attribute, a frequent objective will be to determine the values that are *close* to it. For this purpose we shall also associate with each attribute a *neighborhood radius*. This radius will establish standard neighborhoods of this size around values of this attribute. When the distance function is STRING, then by selecting the neighborhood radius to be 0, the effect is that of isolation: given a value of that attribute, no other value is close to it. As we shall see later, this combination will serve well as our default measure, when no adequate distance function can be provided.

To summarize, relational databases are defined as follows. A database schema is a set of relation schemas. Each relation schema is a sequence of attribute names, one of which is designated as the key attribute. Every attribute is defined with the following parameters:

[1]Note that it is possible to combine the scaling factor and the relative weight into a single parameter. The benefit of two separate parameters is primarily in the process of conceptual design (this issue is discussed further in Section 4).

- *Measure*: the name of the measure by which distances between values of this attribute are calculated. It is either the name of another relation (possibly a distance relation), or the name of a distance function (possibly a standard built-in function).

- *Scaling Factor*: a number for correcting distances between values of this attribute after they are obtained from the measure.

- *Relative Weight*: a number in the range that designates the relative importance of this attribute in the description of the key of this relation, for calculating distances between values of the key.

- *Neighborhood Radius*: a number in the range used to establish standard neighborhoods of this (scaled) distance around values of this attribute.

All distances and all parameters are from the range $[0,\infty)$, except scaling factors which may not be 0. Key attributes always have the relation in which they appear as their measure. Also, as they do not participate in producing distances, they do not specify a weight. As the definition of measures is recursive, care must be taken to avoid *cyclicity* in the definitions. Sometimes, relations have *composed keys*: keys that constitute more than one attribute. Here, we group them together as one structured attribute that is designated as key. For example, A(B,C) denotes that attributes B and C are combined into the structured attribute A.

An example of a schema for a database on restaurants is given in Figure 2-1. Each attribute name is followed by its measure, its scaling factor, its relative weight and its neighborhood radius. Key attributes are indicated by a star. Consider first the relation RESTAURANT. NAME is its key, and TYPE, LOCATION, PRICE, RATING and TEL-NO are the descriptors. TYPE, PRICE and RATING rely on three other relations, called CUISINE, PRICE and RATING, to describe the different restaurant types, price categories and restaurant ratings, and thus provide distance information. LOCATION relies on a special relation, called NEIGHBORHOOD, which specifies the distance between every two locations. The distance between two values of TEL-NO is derived from the standard function STRING. When calculating distances between restaurants, TYPE and RATING are more important than LOCATION or PRICE, and the attribute TEL-NO is to be ignored. The next relation, CUISINE, describes the different cuisines. It uses two attributes to

relation	attribute	key	measure	s	w	r
RESTAURANT						
	NAME	*	RESTAURANT	1	-	1
	TYPE		CUISINE	1	2	1
	LOCATION		NEIGHBORHOOD	10	1	1
	PRICE		PRICE	1	1	1
	RATING		RATING	1	2	1
	TEL-NO		STRING	1	0	0
CUISINE						
	NAME	*	CUISINE	1	-	2
	CATEGORY		STRING	1	2	1
	CALORIES		NUMBER	500	1	0.5
NEIGHBORHOOD						
	NAME(A,B)	*	NEIGHBORHOOD	1	-	1
	MILES		NUMBER	1	1	5
PRICE						
	DESCRIPTION	*	PRICE	1	-	1
	RANKING		NUMBER	1	1	1
RATING						
	DESCRIPTION	*	RATING	1	-	1
	RANKING		NUMBER	1	1	1

Figure 2-1: Schema of a Database on Restaurants

describe each cuisine: CATEGORY groups different restaurant types into more general classes (such as Oriental, Middle-Eastern, etc.) and CALORIES gives the number of calories in an average meal. Distances between values of CATEGORY or CALORIES are to be computed by standard functions, and CATEGORY is assumed to be more important than CALORIES. Next, relation NEIGHBORHOOD describes the distance between every two locations A and B. Note that it is keyed on a structured attribute. Finally, relations PRICE and RATING simply map the non-numeric scales used for price categories and ratings into numeric scales. All scaling factors in this schema are selected to be 1, except for LOCATION and CALORIES. Distances between locations and between calories are scaled by 10 miles and 500 calories, respectively. All neighborhood radiuses are selected to be 1, except for MILES and CALORIES. These radiuses define neighborhoods of 5 miles

RESTAURANT

NAME	TYPE	LOCATION	PRICE	RATING	TEL-NO
Le-Phoney	French	Westwood	Expensive	Awful	395-0297
Cafe-Truque	Nouvelle	Downtown	Moderate	Fair	243-2323
Jasmine-Gardens	Chinese	ChinaTown	Inexpensive	Good	477-5113
Garabanzos	Mexican	Downtown	Moderate	Fair	395-9480
Flower-of-China	Chinese	Hollywood	Expensive	Very-Good	493-8246
Mikonos	Greek	Westwood	Prohibitive	Excellent	828-2104
Lotsapasta	Italian	Hollywood	Inexpensive	Good	424-6942
Havana	Cuban	Fairfax	Moderate	Fair	624-0483
Ala-Kefak	Israeli	Fairfax	Moderate	Very-Good	743-6710
Nippon	Japanese	Downtown	Expensive	Excellent	391-3797

CUISINE

NAME	CATEGORY	CALORIES
Chinese	Oriental	800
Cuban	Latino	1800
French	Continental	2900
Japanese	Oriental	950
Greek	Middle-Eastern	1500
Israeli	Middle-Eastern	1250
Italian	Continental	1800
Mexican	Latino	2000
Nouvelle	Continental	2500

NEIGHBORHOOD (partial)

NAME(A)	NAME(B)	MILES
Westwood	Chinatown	24
Westwood	Downtown	20
Downtown	Fairfax	12
Westwood	Westwood	0
Chinatown	Downtown	4
Chinatown	Fairfax	14
Westwood	Hollywood	8
Chinatown	Chinatown	0
Downtown	Hollywood	10
0	0	0

PRICE

DESCRIPTION	RANKING
Prohibitive	5
Very-Expensive	4
Expensive	3
Moderate	2
Inexpensive	1

RATING

DESCRIPTION	RANKING
Exceptional	5
Excellent	4
Very-Good	3
Good	2
Fair	1
Awful	0

Figure 2-2: Instance of the Database on Restaurants

and 500·0.5 = 250 calories, respectively, around values of MILES and CALORIES. A small instance of this database is shown in Figure 2-2.

So far we have discussed distances informally. Here are our formal definitions. Let x and y be two values of some attribute A with measure M. The *distance between x and y under measure M*, denoted $d_M(x,y)$, is defined as follows:

If M is a distance function (standard or user-defined), then the distance is resolved by some known procedure. For example, if M=STRING, then

$$d_M(x,y) = d_{\text{STRING}}(x,y) = \begin{cases} 0 & \text{if } x=y \\ 1 & \text{if } x \neq y \end{cases}$$

Or, if M=NUMBER, then

$$d_M(x,y) = d_{\text{NUMBER}}(x,y) = |\, x-y \,|$$

If M is not a distance function, it must be a relation. The distance is then defined as

$$d_M(x,y) = \begin{cases} d'_M((x,y),(0,0)) & \text{if } M \text{ is a distance relation} \\ d'_M(x,y) & \text{otherwise} \end{cases}$$

Where d' is a distance calculated from relation M, as follows. Assume M has attributes $A_0,A_1,\ldots,A_n$, and let A_0 be its key. For each i=1,...,n let M_i, s_i and w_i be, respectively, the measure, the scaling factor and the weight of attribute A_i, and denote $w = w_1 + w_3 + \ldots + w_n$. Let x and y be values of attribute A_0, and let $x_1,\ldots,x_n$ and $y_1,\ldots,y_n$ be their respective descriptors. Then:

$$d'_M(x,y) = \sum_{i=1}^{n} d_{M_i}(x_i,y_i) \cdot (1/s_i) \cdot (w_i/w)$$

If either x or y are not values of A_0, then we define:

$$d'_M(x,y) = \infty$$

As the definitions indicate, the distance between x and y under measure M is either calculated by a distance function M, or is derived from a relation M. In the latter case, the distance obtained from M is either the distance between x and y, or the distance

between (x,y) and (0,0). The former is used when M is a relation with a key that contains values of attribute A; the latter is used when M is a relation with a composed key that contains *pairs* of values of attribute A. A relation keyed on a pair of identical attributes is called a *distance* relation. It is assumed that its key attribute always includes an "origin" value (0,0), and the distance between two values x and y is interpreted as the distance between (x,y) and (0,0). The latter is calculated in the usual way, as the weighted and scaled sum of the distances between the values that describe (x,y) and (0,0). Each pair of values may be described with a single attribute (as in relation LOCATION), but in general any number of descriptors may be used.

As an example, $d_{\text{RESTAURANT}}$(**Le-Phoney**,**Cafe-Truque**) is calculated as follows. The measure for TYPE is based on relation CUISINE. There, the distance between **French** and **Nouvelle** is a combination of the distance between their CATEGORY and the distance between their CALORIES. The former distance is calculated by the standard function STRING to be 0 (both are **Continental**); the latter is calculated by the standard function NUMBER to be 400. These distances are combined with weights and scaling factors to a distance of $(0/1)\cdot(2/3) + (400/500)\cdot(1/3) = 0.27$. Next, the measure for LOCATION is based on the distance relation NEIGHBORHOOD. Therefore, the distance between **Westwood** and **Downtown** is given by the distance between (**Westwood**,**Downtown**) and (**0**,**0**), which is calculated by the standard function NUMBER to be 20. The measures for PRICE and RATING are based on the relations by the same names. Each relation has only one describing attribute with the standard function NUMBER. From these relations the distances between **Expensive** and **Moderate**, and between **Awful** and **Fair** are both calculated to be 1. Finally, TEL-NO has the standard function STRING, which calculates the distance between **395-0297** and **243-2323** to be 1. The five individual distances are now combined with weights and scaling factors to produce the following distance between **Le-Phoney** and **Cafe-Truque**:

$$(0.27/1)\cdot(2/6) + (20/10)\cdot(1/6) + (1/1)\cdot(1/6) + (1/1)\cdot(2/6) + (1/1)\cdot(0/6) = 0.92.$$

Similarly, the distance between **Le-Phoney** and **Nippon** is calculated to be 3.63.

3. Goals

Distances enable us to specify *goals*. In this section we introduce goals of three kinds: *neighborhood goals*, *optimum goals* and *priority goals*. Our formal treatment of goals will be done in the context of tuple relational calculus, but other languages should present no problems.

Our definitions for tuple calculus are taken, with minor changes, from [12]. A tuple relational calculus query is an expression of the form $\{ t \mid \psi(t) \}$, where t is a tuple variable and ψ is a formula in predicate logic with t as its only free variable. Except for t, every other tuple variable of ψ must be associated with exactly one relation. The i^{th} component of a tuple variable u is denoted $u.i$. If u is a tuple variable associated with relation R, and A is an attribute of R, then $u.A$ denotes the component of u for the attribute A. The atomic formulas of ψ may be of three kinds:

1. $(u \in R)$, where R is a relation name and u is a tuple variable. These atomic formulas are used to associate variables with relations, as discussed above.

2. $(u.i \ \theta \ v.j)$, where u and v are tuple variables, and θ is one of the following comparators: $=, \neq, <, \leq, >, \geq$.

3. $(u.i \ \theta \ a)$, where u and θ are as above, and a is a constant.

Assume now that x and y are values, to which a measure M applies, and let s and r be two numbers. We define a new arithmetic comparator, called *similar-to* and denoted $\sim$, as follows:

$$x \sim y \ \text{ if } \ d_M(x,y) \cdot (1/s) \ \leq \ r$$

Thus, two values are *similar*, if the distance between them, according to measure M scaled by s, is not greater than r. We extend the definition of atomic formulas to allow θ to be $\sim$. Note that when a formula ψ incorporates several *similar-to* comparators, the particular M, s and r used with each comparator must be known.

3.1. Neighborhood Goals

A *neighborhood goal* is a tuple calculus query that incorporates *similar-to* comparators. We assume that the two operands of each *similar-to* comparison are associated with attributes that have the same measure, scaling factor and neighborhood radius, and that the comparator derives its parameters (i.e. M, s and r) from these values[2].

Every specific request can now be relaxed into a neighborhood goal, by substituting any of its *equal-to* comparators with *similar-to* comparators. As the answer to a neighborhood goal always contains the answer to the specific request from which it was derived, the goal is more *general* (in the sense of [10]) than the specific request.

Consider, for example, the following tuple calculus query to retrieve all locations that have inexpensive restaurants that serve 2000 calories meals:

$$Q = \{\, t \mid (\exists r)\ (\exists c)\ (r \in \text{RESTAURANT}) \wedge (c \in \text{CUISINE}) \wedge (t{=}r.\text{LOCATION}) \wedge (r.\text{PRICE}{=}\textbf{Inexpensive}) \wedge (r.\text{TYPE}{=}c.\text{NAME}) \wedge (c.\text{CALORIES}{=}\textbf{2000}) \,\}$$

As the only inexpensive restaurants are **Jasmine-Gardens** and **Lotsapasta**), and they serve, respectively, 800 and 1800 calories meals, Q will return a null answer.

Assume now that we change Q so that the price constraint becomes r.PRICE $\sim$ **Inexpensive**. Both operands are of the same attribute RESTAURANT.PRICE, which has the measure PRICE (with factor 1 and radius 1). Therefore, the new constraint is satisfied whenever $d_{\text{PRICE}}(r.\text{PRICE},\textbf{Inexpensive}) \leq 1$; which allows r.PRICE to be either **Inexpensive** or **Moderate**. Consequently, the new query will return the location **Downtown**, which has a moderately priced restaurant that serves 2000 calories meals. If, instead, we changed the calories constraint to c.CALORIES $\sim$ **2000**, then all locations that have inexpensive restaurants that serve meals of between 1750 and 2250 calories, will be retrieved (in our example, **Hollywood**). And if both changes are made, then **Fairfax**, which has a moderately priced restaurant that serves 1800 calories meals, will be retrieved in addition to the previous two locations.

---

[2]The requirement that scaling factors and neighborhood radiuses be identical may be relaxed, if we adopt new common parameters, such as the average of the scaling factors, or the minimum of the neighborhood radiuses. However, both operands must have the same measure.

The *similar-to* comparator can also be used between two variables. Assume that Q is changed so that the constraint that joins the RESTAURANT and CUISINE relations becomes r.TYPE $\sim$ c.NAME. The two operands are of attributes RESTAURANT.TYPE and CUISINE.NAME, but both have the same measure CUISINE (with factor 1 and radius 1). Locations will be retrieved, if they have an inexpensive restaurant whose type is *close* to a cuisine with 2000 calories meals. Similarly, if the constraint that binds the free variable t is changed to $t \sim r$.LOCATION, then all locations are retrieved, which are *close* to where there are inexpensive restaurants that serve 2000 calories meals.

3.2. Optimum Goals

Neighborhood goals may result in several answers (and possibly none). Optimum and priority goals may be regarded as two ways to prune the answers to a neighborhood goal, for an answer which is deemed as most suitable.

While neighborhood goals may be based on any tuple calculus query, optimum and priority goals are based on a more restricted family of tuple calculus queries, defined as follows:

$$\{ t^{(n)} \mid (\exists u_1) \ldots (\exists u_m)\, (u_1 \in R_1) \wedge \ldots \wedge (u_m \in R_m) \wedge (t.1 = u_{i_1}.j_1) \wedge \ldots \wedge (t.n = u_{i_n}.j_n) \wedge \phi \}$$

where ϕ is any tuple calculus formula without quantifiers or negations, which is in conjunctive normal form (i.e. ϕ is a chain of subformulas connected with *and* operators, where each subformula is a chain of atomic formulas connected with *or* operators; note that negation can always be effected by changing atomic formulas to use complementary comparators). While this family of queries is a strict subset of the queries of tuple calculus, it is a powerful subset[3].

Let $Q = \{ t \mid \psi(t) \}$ be a neighborhood goal that is based on a query from this restricted family. Let α_i, i=1,...k be the subformulas of ϕ that involve *similar-to*

[3]This family of queries corresponds to the queries that can be expressed with the RETRIEVE statement of the query language QUEL

comparisons, and let $(x_{i,j} \sim y_{i,j})$, $j=1,...,n_i$ be the *similar-to* comparisons in α_i. Assume that $x_{i,j}$ and $y_{i,j}$ always have a common measure $M_{i,j}$, a common scaling factor $s_{i,j}$ and a common neighborhood radius $r_{i,j}$. Let T denote the answer to this goal.

The *optimum goal* Q_{op} is defined as the values of T that minimize the sum

$$\sum_{i=1}^{k} \min\{ d_{M_{i,j}}(x_{i,j}, y_{i,j}) \cdot (1/s_{i,j}) \mid j=1,...,n_i \}$$

As the definition indicates, an optimum goal retrieves all the tuples t which satisfy ψ, while minimizing the total distance used in *similar-to* comparisons.

For example, consider a user who is interested in a Chinese restaurant in Chinatown or in Westwood, whose price is moderate, and whose rating is very good. Except for the type Chinese, the user is willing to relax all the other constraints, but is interested in the restaurant that is *closest* to his description. This request is expressed with the following optimum goal:

$$\begin{aligned} Q_{op} = \{\, t \mid (\exists r)\, (r \in \text{RESTAURANT}) \wedge (t{=}r.\text{NAME}) \wedge (r.\text{TYPE}{=}\textbf{Chinese}) \wedge \\ ((r.\text{LOCATION} \sim \textbf{Chinatown}) \vee (r.\text{LOCATION} \sim \textbf{Westwood})) \wedge \\ (r.\text{PRICE} \sim \textbf{Inexpensive}) \wedge (r.\text{RATING} \sim \textbf{Very-Good}) \,\} \end{aligned}$$

3.3. Priority Goals

Again, let $Q = \{\, t \mid \psi(t) \,\}$ be a neighborhood goal that is based on a query from the restricted family. Let α_i, $i=1,...,k$ be the subformulas of ϕ that involve *similar-to* comparisons, and let $(x_{i,j} \sim y_{i,j})$, $j=1,...,n_i$ be the *similar-to* comparisons in α_i. Assume that in each α_i, for all j, $x_{i,j}$ and $y_{i,j}$ all have a common measure M_i, a common scaling factor s_i, and a common neighborhood radius r_i. Let T_0 denote the answer to this goal.

The *priority goal* Q_{pr} is defined by the following process. From T_0 we select the values, for which the smallest of the distances $d_{M_1}(x_{1,j}, y_{1,j}) \cdot (1/s_1)$ is minimal, and denote this set T_1. Next, from this smaller set T_1 we select the values, for which the smallest of the distances $d_{M_2}(x_{2,j}, y_{2,j}) \cdot (1/s_2)$ is minimal, and denote this set T_2. And so on, until, finally, from the set T_{k-1} we select the values for which the smallest of the distances $d_{M_k}(x_{k,j}, y_{k,j}) \cdot (1/s_k)$ is minimal and denote this set T_k. T_k is the final answer.

Thus, a priority goal relaxes some *equal-to* constraints into *similar-to* constraints, but insists that they are satisfied in a particular order. For example, consider a user who is interested in an inexpensive French restaurant in Downtown. Knowing that there may be none, the user is willing to relax his specific request into a goal. However, the user emphasizes that he is less willing to compromise on the type of the restaurant; between the other two constraints, he is more willing to compromise on the price than on the location. This request is expressed with the following priority goal:

$$Q_{pr} = \{\, t \mid (\exists r)\,(r \in \text{RESTAURANT}) \wedge (t{=}r.\text{NAME}) \wedge (r.\text{TYPE} \sim \textbf{French}) \wedge (r.\text{LOCATION} \sim \textbf{Downtown}) \wedge (r.\text{PRICE} \sim \textbf{Inexpensive}) \,\}$$

Optimum or priority goals too may result in several answers (and possibly none). But if the neighborhood goal, on which an optimum or priority goal is based, has an answer, then the optimum or priority goal will have an answer.

3.4. Expressing goals in QUEL

To demonstrate how goals would be expressed in an actual query language, we choose QUEL [11]. As already mentioned, QUEL uses a RETRIEVE statement which corresponds to the restricted family of queries defined in Section 3.2. As an example, the query Q of Section 3.1 is expressed in QUEL as follows:

```
range of r is RESTAURANT
range of c is CUISINE
retrieve (r.LOCATION) where
        r.PRICE = 'Inexpensive' and
        r.TYPE = c.NAME and
        c.CALORIES = '2000'
```

Only minor additions to the syntax of QUEL are required to extend it so it can express goals. The symbol =? is used for the *similar-to* comparator. Neighborhood goals are specified simply by using =? in the **where** part of the RETRIEVE statement. Optimum or priority goals require the keyword **optimum** or **priority** to be mentioned after the keyword **retrieve**. Thus, a request to retrieve the restaurant in Downtown that resembles most an inexpensive French restaurant is expressed with the following goal:

range of *r* **is** RESTAURANT
retrieve optimum (*r*.NAME) **where**
r.TYPE =? **'French' and**
r.LOCATION = **'Downtown' and**
r.PRICE =? **'Inexpensive'**

As evident from the syntax, a neighborhood goal is expressed like a regular query. As optimum and priority goals are always satisfied from answers to neighborhood goals, the keywords **optimum** and **priority** instruct the system to prune these answers[4]. Notice that goals tend to be short: when trying to express goals in a system that supports only specific requests, queries often tend to use many disjunctions.

4. Issues of Design and Implementation

The process of defining a new database is usually referred to as *database design*. A database designer models a real-world environment with elements of the data model such as relations, keys, constraints, etc. In our extended model this process now includes also the determination of the appropriate measures and parameters (scaling factors, relative weights and neighborhood radiuses) for each database attribute. Proper selection of these measures and parameters is critical to the successful handling of goals. Some guidelines follow.

If an attribute is described by another relation, we may use that relation as a measure. Otherwise, assuming this attribute varies over a finite domain, we may choose to add to the database a special distance relation. For attributes which vary over infinite (or very large) domains we must use a distance function. If a standard built-in function such as NUMBER or STRING is not satisfactory, a new distance function may be defined. As an example of a new distance function, consider a relation that associates license plates with car owners. We may define a new function to compute the distance between two different license plates. This distance can then be used in a goal query to try and locate the owner of a car, based on an approximation of the true license plate. STRING (with radius 0) is also the default measure. When an appropriate measure cannot be provided,

[4]This agrees nicely with the style of QUEL, which uses another retrieve keyword, called **unique**, to prune away all replications.

the use of this measure will defeat all *similar-to* comparisons with values of this attribute. Obviously, by using STRING measures throughout, the database reverts to the situation where only specific requests are permitted.

Scaling factors may be chosen to grade the distances obtained from the measures by what is considered a significant difference (in the context in which these distances are to be applied). When comparing restaurants, a distance of 10 miles between their locations may be considered significant; when comparing cuisines, a difference of 500 calories in the average calorie contents of their meals may be considered significant. Often, rankings like those used for pricing or rating restaurants, do need not to be scaled.

In selecting neighborhood radiuses, the radius 1 is often acceptable (this is because the distances which are compared to the radiuses are already scaled). Given a value, it creates around it a neighborhood which incorporates values that are similar to it, as well as the closest values that are already significantly different from it. Alternatively, if the values of this attribute are scattered arbitrarily, we may choose a radius that is some fraction of the maximum distance between values of this attribute. For example, if the maximum distance between locations is 80 miles, we may choose to create 8 miles neighborhoods, so that approximately 10% of all locations will be incorporated into each neighborhood (if the scaling factor is 10, then the radius will be 0.8).

After some experience, scaling factors and neighborhood radiuses may be "fine-tuned" to values that are usually satisfactory. However, selecting the weights that combine different attributes of a description into a measure, is much more difficult. Weights determine what is important in a description; and this may not have a unique answer. For example, in a description of a restaurant, some may consider TYPE as the most important attribute; others may consider RATING or PRICE as more important. Thus, a particular combination of weights can only represent a compromise. However, when a given combination of weights is unsatisfactory, the user should be able to specify his own priorities. This should become part of the definition of the user's *view* of the database, which is designed to accommodate the needs of this user. The use of scaling factors tends to simplify the specification of relative weights; weights can usually be expressed

with integers from a limited range (e.g. between 1 and 20). When the default weights prove unsatisfactory for a particular user, selecting new weights is relatively simple.

To implement goal retrieval, we must modify the DBMS in several places. While the nature and extent of the necessary modifications depend on the system, we can observe four groups of changes, as follows. (1) The schema constructor must be modified to accept the extended definitions of attributes (i.e. the measures and parameters). (2) Similarly, the query processor must be modified to accept the extended query language. (3) A new recursive procedure called DISTANCE must be written, which accepts two values, the name of a measure, a scaling factor and a neighborhood radius, and determines whether the scaled distance between the values is within the radius. (4) All procedures that iterate over database values, comparing them to a particular value, must be changed to call this new procedure for the comparisons.

The only computations that may affect the performance of query processing are the repeated calls to the DISTANCE procedure. If the measure is a distance function, then DISTANCE will not require additional retrievals. If the measure is based on another relation, then two additional retrievals will be necessary to obtain two new descriptions, and then DISTANCE will be called recursively on each pair of description components. Thus, each call to DISTANCE will require at most two retrievals[5]. The total number of retrievals necessary to resolve a distance between two values depends on the extent of its dependence on other measures. If n relations are involved in the recursive definition of a particular measure, then $2n$ retrievals will be necessary to resolve a single distance between two values of this measure. However, in most cases n is expected to be a very small number. Also, when assessing this cost, one should compare it to the cost of repeating the whole query many times (with minor modifications), in systems that cannot handle goals.

When answering goals, it may be beneficial to include in the answers the values of the *similar-to* comparators, upon which each answer is based. For example, the optimum

[5]These retrievals are relatively fast, as key values are provided.

goal that retrieves the name of the restaurant in Downtown that resembles most an inexpensive French restaurant, should return, in addition to the name of the restaurant, its type and price.

We mentioned that goals may still return null answers (of course, not as often as specific requests). When a goal evaluates to a null answer, it may be desirable to *repeat* it automatically (or after user approval) with wider neighborhoods (e.g. double all neighborhood radiuses).

5. Conclusion

Goals account for much of the usage of databases: if one compares database retrieval to telephone directory lookup, then specific requests resemble "white pages" lookup, while goals resemble "yellow pages" lookup. Systems that allow users to express goals directly (rather than require them to iterate through numerous specific requests) are more cooperative, and possibly more efficient.

A legitimate concern is that goals will be satisfied by meaningless values, and we already emphasized the importance of selecting all measures and parameters carefully. Still, we claim that the use of our mechanism is mostly risk-free. First, it is obvious that users who specify a *similar-to* comparator are well aware of its semantics, and would probably examine answers to goals more carefully than answers to specific requests. Also, if the values which satisfy the *similar-to* comparators are displayed along with the answers (as suggested above), then users can always monitor the decisions made by the system.

References

[1] F. 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] C. J. Date.
An Introduction to Database Systems, Volume I (Third Edition).
Addison Wesley, 1982.

[3] C. J. Date.
An Introduction to Database Systems, Volume II.
Addison Wesley, 1983.

[4] S. Ginsburg and R. Hull.
Order Dependency in the Relational Model.
Theoretical Computer Science 26:149-195, 1983.

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

[6] S. J. Kaplan.
Cooperative Responses from a Portable Natural Language Query System.
Artificial Intelligence 19, 1982.

[7] 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, California, 1980.

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

[9] D. J. McLeod.
High Level Definition of Abstract Domain in a Relational Data Base System.
Journal of Computer Languages 2(3):61-73, July, 1977.

[10] A. Motro.
Query Generalization: A Technique for Handling Query Failure.
In *Proceedings of the First International Workshop on Expert Database Systems*, pages 314-325. Kiawah Island, South Carolina, 1984.

[11] M. Stonebraker et al.
The Design and Implementation of INGRES.
ACM Transactions on Database Systems 1(3):189-222, September, 1976.

[12] J. D. Ullman.
Principles of Database Systems.
Computer Science Press, 1982.

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