ReqFlex: Fuzzy Queries for Everyone

Grégory Smits[1], Olivier Pivert[1], Thomas Girault[2]
[1]IRISA-University of Rennes 1, France
{gregory.smits | olivier.pivert}@irisa.fr
[2]Freelance Engineer
toma.girault@gmail.com

ABSTRACT

In this demonstration we present a complete fuzzy-set-based approach to preference queries that tackles the two main questions raised by the introduction of flexibility and personalization when querying relational databases: i) how to efficiently execute preference queries? and, ii) how to help users define preferences and queries? As an answer to the first question, we propose PostgreSQL_f, a module implemented on top of PostgreSQL to handle fuzzy queries. To answer the second question, we propose ReqFlex an intuitive user interface to the definition of preferences and the construction of fuzzy queries.

1. INTRODUCTION

The last decade has witnessed an increasing interest in expressing preferences inside DataBase (DB) queries. As a matter of fact, the first research works on this topic date back to the late 80s, see for instance [3]. Motivations for such a concern are manifold. First, it has appeared to be desirable to offer more expressive query languages that can be more faithful to what a user intends to ask. Second, the introduction of preferences in queries provides a basis for rank-ordering the retrieved items, which is especially valuable in case of large sets of items satisfying a query. Third, a classical query may also have an empty set of answers, while a relaxed (and thus less restrictive) version of the query might be matched by some items.

Approaches to database preference queries may be classified into two categories according to their qualitative or quantitative nature. In the latter, preferences are expressed quantitatively by a monotone scoring function (the overall score is positively correlated with partial scores), often taken as a weighted linear combination of attribute values. Since the scoring function associates each tuple with a numerical score, tuple t_1 is preferred to tuple t_2 if the score of t_1 is higher than the score of t_2. Typical representatives of this first category are fuzzy set-based approaches [2], which use membership functions that describe the preference profiles of the user on each attribute domain involved in the query.

In the qualitative category of approaches, preferences are defined through binary preference relations. Typical representatives of this category are approaches based on Pareto order, aimed at computing non-dominated answers (viewed as points in a multidimensional space, their set constitutes a so-called skyline), starting with the works of Bőrzsőnyi *et al* [1].

In this paper, we focus on the fuzzy-set-based approach to database preference queries, which benefits from the great expressivity of fuzzy set theory when it comes to modeling various types of preferences. As a typical representative of a fuzzy query language, we consider SQLf [4], a fuzzy extension of SQL initially proposed in the 90s and completed by different add-ons since then. This language incorporates many fuzzy features and is thus a powerful tool for expressing database preference queries. However, two questions remain somewhat open, that respectively concern: i) the efficient implementation of a fuzzy querying system based on SQLf, ii) the way a nonexpert user may be guided in his/her elicitation of fuzzy queries (which are intrinsically more complex to specify than regular queries since membership functions come into play). As illustrated by Figure 1, this demonstration proposes a semi-integrated implementation of a module with fuzzy querying capabilities on top of PostgreSQL as an answer to the first question (Section 2), and for the second question, we introduce ReqFlex, an intuitive interface for nonexpert users to the definition of preferences and queries (Section 3).

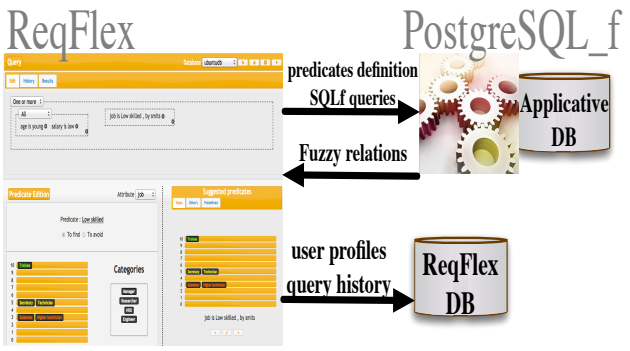

Figure 1: Global view of the fuzzy querying process

Proceedings of the VLDB Endowment, Vol. 6, No. 12

2. FUZZY QUERIES

2.1 Preference Model

Fuzzy sets are convenient tools to model vague criteria and user's preferences [2]. Using this paradigm, a preference is represented by means of a set whose boundaries are gradual. Thus, the satisfaction of a tuple t regarding such a fuzzy set F is a matter of degree in the unit interval denoted by $\mu_F(t)$. The underlying fuzzy set theory offers a large panoply of connectives to aggregate these preferences from classical conjunction (min) and disjunction (max) to quantified statements (most of, at least two, around a dozen, ...) and weighted averaging operators.

In our context of DB querying, users define fuzzy sets to model their preferences that are associated with linguistic labels like '*recent*', '*low*', '*very cheap*', etc. Moreover, in accordance with the imprecise nature of the concepts they represent, membership functions associated with the fuzzy sets behind these properties introduce some graduality when checking the satisfaction of the items wrt. the user's preferences. The satisfaction degree in $[0, 1]$ provides the necessary information to rank-order the items that somewhat satisfy the user's requirements. As illustrated in Figure 2 (top), membership functions defined over numerical attributes are most of the time of a trapezoidal shape, that clearly exhibit a range of ideal values (maximal satisfaction degree of 1) and another of acceptable values (satisfaction degree greater than 0). Over categorical attributes, categories of interest are individually associated with satisfaction degrees taken in the unit interval as illustrated in Figure 2 (bottom).

2.2 Fuzzy DB Queying

The language called SQLf described in [4] extends SQL so as to support fuzzy queries. The general principle consists in introducing gradual predicates wherever it makes sense. The three clauses *select*, *from* and *where* of the base block of SQL are kept in SQLf and the *from* clause remains unchanged. The principal differences concern mainly two aspects: i) the calibration of the result since it is made with discriminated elements, which can be achieved through a number of desired answers (k), a minimal level of satisfaction (α), or both, and ii) the nature of the authorized conditions as mentioned previously. Therefore, the base block is expressed as:

select [**distinct**] [$k \mid \alpha \mid k$, α] *attributes*
from *relations* **where** *fuzzy-cond*;

where *fuzzy-cond* may involve both Boolean and fuzzy predicates.

The operations from the relational algebra — on which SQLf is based — are extended to fuzzy relations by considering fuzzy relations as fuzzy sets on the one hand and by introducing gradual predicates in the appropriate operations (selections and joins especially) on the other hand [4]. As an illustration, the fuzzy selection operator is defined as:

$$\mu_{select(r,\, cond)}(t) = \top(\mu_r(t),\, \mu_{cond}(t))$$

where r denote a fuzzy relation, *cond* is a fuzzy predicate and $\top$ is a triangular norm (most usually, *min* is used). A typical example of a fuzzy query addressed to a database containing information about employees is:

select 20 0.6 * **from** *employees* **where** *age* **is** 'young' **and** *job* **is** 'low_skilled';

where 'young' and 'low_skilled' are two fuzzy terms defined by the membership functions that appear in Figure 2.

2.3 PostgreSQL_f

In the absence of a commercial RDBMS capable of interpreting fuzzy queries, it was necessary to perform a so-called *derivation step* [4] in order to generate a regular Boolean query used to prefilter the relation concerned and then to use a third party programming language to build the fuzzy relations on the basis of the results returned by the derived Boolean query. We now propose PostgreSQL_f[1], an extension of PostgreSQL that implements the functionalities necessary to the evaluation of fuzzy queries. The interests of such a mild coupling architecture lies in the fact that the fuzzy resulting relation is directly computed during the tuple selection phase which improves the overall performance of the fuzzy query execution process. Moreover, the definition of PostgreSQL_f as an external module that can be loaded at runtime by PostgreSQL makes easier the maintenance and code distribution. Implemented with functions and procedures written in C or PL/PGSQL, the current version of PostgreSQL_f offers the following functionalities:

Definition of fuzzy predicates over numerical attributes using the function create_numerical_predicate that takes as arguments the attribute concerned, the linguistic label and the bounds of the trapezoidal membership function,
select create_numerical_predicate('year', 'recent', 2005, 2006, 2008, 2010);

Definition of fuzzy predicates over categorical attributes using the function create_categorical_predicate that takes as arguments the attribute concerned, the linguistic label and associations between categories of interest and satisfaction degrees:
select create_categorial_predicate('job', 'high_skilled', ['seller', 'technician', engineer'], [0.4, 0.3, 0.5]);

Introduction of fuzzy conditions in the selection clause using the operator $\sim=$:
select * **from** *cars* **where** *year* $\sim=$ 'recent';

Application of modifiers to alter the definition of fuzzy predicates. Pre-defined modifiers available by default are *very* (strengthening modifier) and *rather* (weakening modifier). They are defined as follows: $\mu_{mod\,P}(x) = (\mu_P(x))^n$ where $n = 2$ (resp. 0.5) if *mod* is *very* (resp. *rather*):
select * **from** *cars* **where** year $\sim=$ 'very recent';

Conjunctions and disjunctions. As in the classical case, a selection condition can be defined as a conjunction or a disjunction of fuzzy or Boolean predicates. The SQL connectives **and** and **or** have been respectively extended by means of the operators && and || . The triangular norm and conorm underlying these operators can be selected among predefined ones. A function that converts Booleans to real numbers is used to combine Boolean and fuzzy predicates:
select * **from** *cars*
where *year* $\sim=$ 'very recent' && km $\sim=$ 'low' && brand = 'BMW';

[1] `https://github.com/postgresqlf/PostgreSQL_f`

Calibration of the results with thresholds. Specified in the *select* clause, qualitative (α) and quantitative (k) thresholds can be defined so as to control the satisfaction level or cardinality of the answers returned:

select set_alpha(0.4); **select** set_k(20);

Aggregation of predicates with fuzzy quantifiers. Besides conjunction and disjunction, fuzzy quantifiers such as *most* or *at_least_two* may be used to combine fuzzy (and/or Boolean) predicates. Different interpretations of fuzzy quantifiers are available:

select * **from** *cars*
where most(*year* ∼= 'very recent', km ∼= 'low, *brand* = 'VW');

Gradual operators. Provided that distance functions have been defined on the attribute domains, gradual operators can be used to perform comparisons with scalars (operator ∼) or sets of values (operator *in* ∼):

select * **from** *cars*
where *year* ∼ 2008 && *brand in* ∼ ('Peugeot', 'Renault');

3. REQFLEX

Obviously, formal languages such as SQL and SQLf for fuzzy queries are not so easy to use for novice users. This is why it is necessary to propose intuitive user interfaces to help them formulate their queries. All the functionalities offered by PostgreSQL_f cannot be easily managed graphically, especially by inexperienced users, this is why ReqFlex focuses on the intuitive definition of queries composed of a projection and a selection part addressed to a universal relation that may be a view over joined tables.

As illustrated in Figure 1, the research prototype named ReqFlex[2] we developed acts as an RDBMS query interface. This interface generates SQLf queries that can be directly executed by the PostgreSQL_f extension. The user interface is connected to an application DB that stores user profiles. A user profile is composed of authentication information and also stores the previously defined fuzzy sets and queries. This way, it is possible to build a personal vocabulary composed of fuzzy sets and previously executed fuzzy queries. This DB also stores the connection details of the accessible databases and the list of searchable attributes.

3.1 An Intuitive Definition of Fuzzy Predicates

Since the semantics of a fuzzy term relies on its membership function, any fuzzy querying system must provide the users with a convenient way to define the membership functions of the fuzzy terms that they wish to include in a query.

The query interface we have developed proposes an intuitive and user-friendly method to define personalized fuzzy sets. As illustrated in Figure 2, the fuzzy predicate edition panel lets users easily define their fuzzy predicates on numerical (top) and categorical attributes (bottom). For predicates on numerical attributes, the membership function can be defined using the two sliders to set the ideal and acceptable value intervals or using its graphical representation directly by dragging the boundaries of the trapezoidal function. For categorical attributes, users can drag the different values of the concerned attributes to a scale of satisfaction

[2] `http://thomas.girault.fr/reqflex/` using the login *smits*

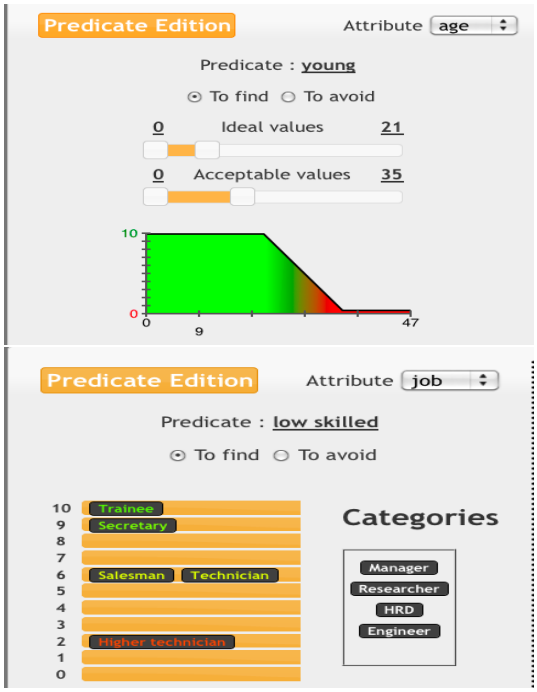

Figure 2: Fuzzy predicates on numerical (top) and categorical (bottom) attributes

degrees. To help the users construct their own predicates, the interface suggests to refine fuzzy predicates defined by other users on the concerned attribute (bottom-right part of Figure 3). Users can save the fuzzy predicates they have defined in their profile in order to reuse or refine them in future queries.

3.2 A Drag and Drop Query Construction

Fuzzy predicates can then be dragged and dropped into the fuzzy query edition panel. When users move their predicates into the query edition panel, they can create new groups of predicates or insert them into an existing one. As soon as a group contains more than one predicate, the user has to choose the connective he/she wants to use to aggregate the satisfaction degrees related to the different predicates. A list of connectives is proposed, which ranges from classical conjunction/disjunction to quantified statements ('*most*', '*at least 2*', ...). A tooltip containing a short description of the semantics of each connective is displayed on demand. Previously executed queries are also stored in the user's profile as XML documents.

Figure 3 illustrates how the drag and drop system may be used to drag the predicate *age is 'young'* to a box already containing the predicate *salary is 'low'*. As this box now contains more than one predicate, a connective has to be chosen among the conjunction 'All', the disjunction 'One or more' and some predefined quantifiers, here one considers that the conjunction is used to aggregate the predicates of this box. Then, after choosing the disjunctive connective *One or more* to combine the two conjunctive boxes, one obtains the query illustrated in Figure 4 corresponding to *(job is 'low_skilled' and age is 'young') or (age is 'young' and salary is 'low')*.

Figure 3: Intuitive drag-and-drop fuzzy querying

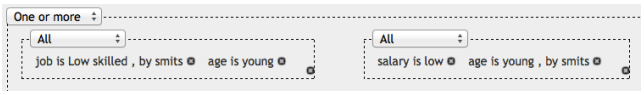

Figure 4: Edition panel of the selection part of the query

The top right part of the interface proposes a set of buttons to reset the query, logout, execute the query or set some additional parameters (Figure 5) such as: i) the list of attributes to introduce into the projection clause; ii) a qualitative threshold α to retrieve the items whose satisfaction with respect to the query is greater than α, or iii) a quantitative threshold k aimed at retrieving the k most satisfactory items only.

When the execution button is clicked, the new fuzzy predicates are translated into *PL/PGSQL* functions and the query is translated into SQLf. The code is then submitted to the RDBMS. The execution of a fuzzy query Q returns a fuzzy relation where each tuple, say t, is associated with a satisfaction degree denoted by $\mu_Q(t)$ or simply mu in the result panel illustrated in Figure 6. Thanks to the fact that the returned tuples are presented in a decreasing order of their satisfaction degree, users may easily identify the tuples that best satisfy their queries and may also adjust the quantitative parameter k to reduce the size of the result set.

4. DEMONSTRATION

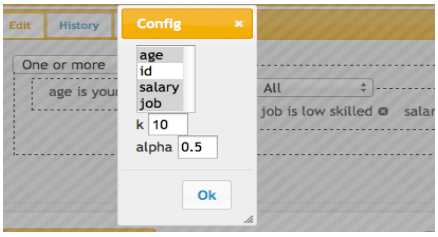

Figure 5: Additional parameters for a fuzzy query

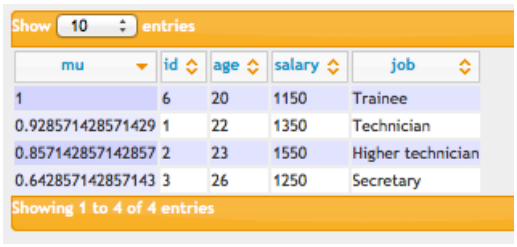

Figure 6: Visualization of the fuzzy relation returned by the query *age* $\sim=$ *'young'* && *job* $\sim=$ *'low skilled'* with the definition of the predicate *'young'* and *'low_skilled'* given in Figure 2

In order to prove the intuitiveness of ReqFlex, we will let participants inexperienced in fuzzy theory to query two datasets. The first one of a human-manageable size (about 100 tuples) concerns the description of employees and is of a particular interest to discover the main features of ReqFlex as well as the relevance of the returned tuples. The second data set, composed of 80.656 ads about second hand cars described on 15 attributes, will be used to assess the expressivity of the fuzzy queries that can be defined through the user interface of ReqFlex but also to measure the efficiency of the mild coupling strategy implementation of PostgreSQL_f.

To the best of our knowledge, the coupling of ReqFlex with PostgreSQL_f offers the first unified approach to fuzzy-set-based preference queries that is accessible to nonexpert users, hence to real applicative contexts.

5. REFERENCES

[1] S. Bőrzsőnyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of the 17th IEEE Inter. Conf. on Data Engineering*, pages 421–430, April 2001.

[2] D. Dubois and H. Prade. Using fuzzy sets in flexible querying: Why and how? In *Proc. of the 1996 Workshop on Flexible Query-Answering Systems*, pages pp. 89–103, 1996.

[3] T. Ichikawa and M. Hirakawa. ARES: a relational database with the capability of performing flexible interpretation of queries. *IEEE Transactions on Software Engineering*, 12:624–634, 1986.

[4] O. Pivert and P. Bosc. *Fuzzy Preference Queries to Relational Databases*. Imperial College Press, London, UK, 2012.