Efficient GCD functions in SQL

Joachim Nielandt[1] **Antoon Bronselaer**[1] **Guy de Tré**[1]

[1] Department of Telecommunications and Information Processing, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium

Abstract

Querying a database in an intuitive way is becoming more important. Users expect their tools to present an easy-to-use interface that can interpret complex search criteria. These criteria can be optional, mandatory, unwanted or grouped together, according to what the developer of a tool wishes to offer to the user. This research focuses on making it possible to interpret a complex aggregation of criteria and retrieve results in the standard language of the target database. The translation of user query to database query is presented and some experiments regarding performance are given.

Keywords: PostgreSQL, criteria aggregation, weighted preferences, GCD, LSP, SQL

1. Introduction

Flexible querying. When a user wants to query a database using SQL in a non-trivial way he is faced with a couple of problems. Traditionally, a user will build a query where he/she filters out what is not needed, using a WHERE clause. This clause will contain a couple of expressions which are aggregated with conjunctions ($\wedge$), disjunctions ($\vee$) and negations ($\neg$).

This causes problems on a fundamental level when the user needs to specify something intuitive (and thus more closely related to human reasoning). How would the user for example specify houses with a price that is acceptable (e.g. between 200k and 300k, the lower the better), with at least 2 bedrooms (the more the better), yet preferably not close to a train station (the farther away the better). When the preferences are nested (i.e., an aggregation is needed to combine some attributes and some results of other aggregations), building a binary logical filter becomes even more difficult. Intuitively it is readily apparent that there is a strong need for ordering the results, in order to show the best match first, followed by results that respectively degrade in the quality of their match with the query.

Ordering results is especially important these days, as most tools that retrieve data according to a user's query have to present their results quickly and accurately. Users do not look far down the list of results and expect to see the desired result at the top of the list. A strong example for this behaviour is an internet search engine, where people pay most of their attention to the first page and, more specifically, the area around the first result[1, 2].

This is why flexible querying has become a strong tool that has sparked a healthy interest. In what follows we will use a *property* database as a working example, containing over 200.000 records detailing houses (properties) and their characteristics (e.g. price, construction year, renovation year, number of bedrooms ...). This is our own dataset and not publicly available.

Objectives. Because of the fact that building flexible queries using standard logical operators is non-intuitive and complex, there is a strong need for simplification in that regard. We propose, as a first step in our research regarding this topic, a grammar that allows specifying fuzzy queries. These queries can then be translated into the language of the database of choice, allowing for fine-grained control over the test conditions and manipulation and optimization of the query.

Improving the performance is a strong goal, as aggregating preferences can take a lot of computational power. Usually, this entails a full scan of the considered table, which is expensive, timewise. One of the objectives is to avoid this full scan and see whether we can optimize any way we can.

Overview. In Section 2 a couple of concepts are given that are needed throughout the paper. Fuzzy querying is situated and methods for scoring components are given, with special attention given to GCD functions as they will take a central role. In Section 3 the problem we are facing is described, comprising the aggregation of preferences using GCD from within a database. Afterwards, Section 4 handles the proposed solution to the problem, which details the composition of a custom query grammar and query preprocessing. Results are presented in Section 5, giving performance measurements using various queries and parameters. To finish, Sections 6 and 7 handle prospects for future work and conclusions respectively.

2. Preliminaries

In the following preliminaries attention is given to the field of fuzzy querying and how scores are used and aggregated to allow ranking components of a system. More specifically, GCD (Generalised Conjunction / Disjunction) functions are situated and

the advantage they could bring to fuzzy querying is highlighted.

2.1. Fuzzy querying

Querying any regular database can become difficult when considering complex needs of a user. Fuzzy querying offers tools to alleviate these difficulties, and is a useful concept to explain what follows in the paper (for an overview, see [3]). A regular database consists of rows (tuples of the relation) and columns (attributes of the relation), together forming the table (relation). We can write this relation R as the following schema:

$$R(A_1 : T_1, ..., A_n : T_n)$$

with R the relation, A_i the ith attribute and T_i the datatype of A_i. T_i determines the allowed values, or domain dom_{T_i}, for attribute A_i. A single tuple can then be described as follows:

$$t_j(A_1 : v_1, ..., A_n : v_n)$$

with t_j the jth tuple and $v_i \in dom_{T_i}, 1 \leq i \leq n$ the value for the ith attribute.

What fuzzy querying allows is to define a query using linguistic terms, which makes it intuitive to use and reason about. These linguistic terms are expressed in terms of fuzzy sets, which are defined on one single attribute $A : T$. They express the desirability of values of the domain dom_T, using a membership function μ, so we can determine how desirable a given value is. This desirability can later be used to aggregate and process.

Example 2.1. An example of a fuzzy set, in the context of our example concerning properties (see 5.1), could be the fuzzy set with membership function μ_{cheap}, shown in Figure 1. This example shows the desirability of a house according to a user , based on its price, where any house cheaper than 100k is perfect and any house more expensive than 200k is unacceptable. Values in between show a gradual decrease in desirability.

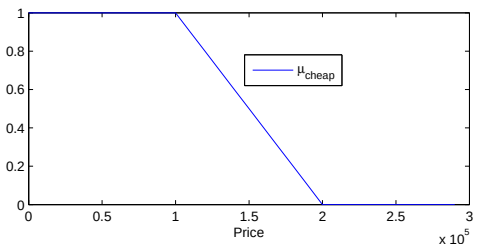

Figure 1: Example membership function μ_{cheap} for property price

2.2. Scoring components

Traditional scoring. In traditional databases, scoring records is done by assigning a boolean value to every considered attribute. 0, 1 and *NULL* respectively indicate that the attribute is unwanted, wanted and that there is no opinion about it (unknown). Aggregating these scores can be done with ternary logic, which falls out of the scope of this paper.

Fuzzy querying. Evaluating a system that consists of n components, using fuzzy querying, is approached as follows. Each component c_i is given a score e_i, with $1 \leq i \leq n$ and $e_i \in [0, 1]$. A weight $w_i \in [0, 1]$ can be given to c_i, to differentiate between the importance of individual components. Weights are normalized and sum to 1 ($w_1 + w_2 + ... + w_n = 1$). A possible global score can than be calculated as follows:

$$e = w_1e_1 + w_2e_2 + ... + w_ne_n, 0 \leq e \leq 1 \qquad (1)$$

GCD-based fuzzy querying. GCD functions are, for example, used in the LSP (Logic Scoring of Preference) [4] method. The LSP soft computing technique is used for multi-criteria decision support and allows evaluating a system during all phases of its lifecycle. This generally means that evaluation occurs from the moment the initial feasibility study is performed, up until the final acceptance tests. What we need, specifically, from this method, is the *specification of the preference aggregation structure* [5], which is performed using GCD.

The above mentioned traditional way of aggregating scores into a global score (see Equation 1) does not allow an important aspect which is present in LSP: modelling a mandatory requirement, a component that is essential to the system, which means that, if component c_i does not qualify at all ($e_i = 0$), the overall score of the system will be 0 as well, regardless of the other components [5].

This results in GCD being a tool that allows making decisions in a flexible and natural (as a human would reason) way: it is possible to define a complex array of components with individual scores, and aggregate everything with a high degree of control. This makes it a valuable tool to be applied in the field of fuzzy querying.

GCD functions allow modelling a broad range of aggregators, and form a superset for bipolar queries with mandatory and optional requirements. This makes it a powerful tool, which influenced the choice to investigate this approach first. A detailed comparison and discussion about definitions can be found in literature [6, 7].

GCD functions. In its most general form, the aggregation function is the so-called *weighted power mean* (WPM) [5, 8, 9]:

E	r	Name
$min(e_1, e_2)$	$-\infty$	Minimum
$1/(w_1/e_1 + w_2/e_2)$	-1	Harmonic Mean
$(e_1)^{w_1} \cdot (e_2)^{w_2}$	0	Geometric Mean
$w_1 e_1 + w_2 e_2$	1	Arithmetic Mean
$\sqrt{w_1 e_1^2 + w_2 e_2^2}$	2	Square Mean
$max(e_1, e_2)$	$+\infty$	Maximum

Table 1: Special cases for value r in WPM

$$e = \begin{cases} (\sum_{i=1}^{n} w_i e_i^r)^{1/r} & \text{, if } |r| \in]0, +\infty[\\ \prod_{i=1}^{n} e_i^{w_i} & \text{, if } r = 0 \\ \min(e_1, \ldots, e_n) & \text{, if } r = -\infty \\ \max(e_1, \ldots, e_n) & \text{, if } r = +\infty \end{cases} \quad (2)$$

with $r \in [-\infty, \infty]$ a parameter (real number) that influences the characteristics of the aggregation the WPM will perform and $\{e_i \in [0, 1] | 1 \leq i \leq n\}$ the set of criteria that will be aggregated by the WPM.

The more r approaches $-\infty$, the more the WPM will aggregate the criteria in a conjunct fashion. The reverse is also true; the more r approaches $+\infty$, the more the aggregation will be performed in a disjunct fashion.

As an illustration of the method, a couple of special cases for the value r are useful, more specifically in the case of $n = 2$ (as defined by [9]), shown in Table 1.

The cited authors also proposed the following values for r (see Table 2), to indicate different gradations between conjunction and disjunction. These values were calculated by relating a quantity $c \in [0, 1]$, called the conjunction degree, to r, for certain values of n. Variable c indicates the degree of conjunctivity of an aggregate function [9], where $c = 0$ and $c = 1$ will respectively result in a disjunction (i.e., the maximum) and conjunction (i.e., the minimum).

Aggregating multiple criteria is intuitively done in small steps. Small groups of preferences are each aggregated into an aggregated preference. These can be grouped again, in an analogue way, creating a tree with preferences as leaves and aggregated preferences as inner nodes. Approaching the preferences in this way makes sense, as it is hard to reason about an aggregation that takes into account a high number of preferences, without grouping them and making the problem more manageable [10].

3. Problem description

Consider the database we used for tests (see Section 5.1), containing over 200k records describing houses, with a large number of attributes (numbers, booleans, text ...):

$$\{a_j | 1 \leq j \leq m\}$$

property	nearschool	e
p_1	true	1
p_2	false	0
p_3	true	1

Table 3: Example boolean preference

property	price	e
p_1	140000	0.6
p_2	90000	1.0
p_3	250000	0.0

Table 4: Example property prices and preferences

Our aim is to launch a query on this database, which expresses a number of preferences of a user (in the context of this paper we are targetting a developer, not an end-user of a commercial product).

Consider a subset of attributes in which the user is interested in:

$$\{a_i | 1 \leq i \leq n \wedge n < m\}$$

As the columns of the property database contain data of different types, formats and restrictions, a first step is to preprocess the data into n preferences $e_i | e_i \in [0, 1] \wedge 1 \leq i \leq n$, one for each attribute a_i the user has an opinion about.

Example 3.1. As a most simple example, we can calculate a preference e based on the boolean attribute *nearschool* as shown in Table 3.

Example 3.2. It becomes more interesting when we are dealing with more complex preferences. If we for example base ourselves on the membership function of the fuzzy set that is shown in Figure 1 to calculate preference e, it is apparent there is a gradual nuance between liking the price and not liking the price. Everything below 100k is perfect, everything above 200k is unacceptable.

Based on the membership function μ_{cheap} shown in Figure 1 it is now possible to calculate preferences based on prices, with some examples shown in Table 4.

After transforming attributes $\{a_i | 1 \leq i \leq n\}$ into preferences $\{e_i | 1 \leq i \leq n\}$, we can start considering the importance of each preference (by adding weights) and how they relate to one another (by determining a conjunctive degree, thus influencing the parameter r in the WPM).

4. Proposal

In this section we propose the use of a novel query grammar to describe the aggregation of preferences in a database. Queries that implement this grammar are transformed in the native language of a target database, enabling a user to execute complex fuzzy queries. More importantly, optimizations are suggested and implemented that take advantage of

Mandatory requirement	Name of Operation	Symbol of operation	Conjunctive degree (c)	Value of r			
				$n = 2$	$n = 3$	$n = 4$	$n = 5$
No	Disjunction	D	0.000	$+\infty$	$+\infty$	$+\infty$	$+\infty$
No	Strong QD	D+	0.125	9.52	11.09	12.28	13.16
No	Medium QD	DA	0.250	3.93	4.45	4.82	5.09
No	Weak QD	D-	0.375	2.02	2.19	2.30	2.38
No	Arithmetic Mean	A	0.500	1.00	1.00	1.00	1.00
No	Weak QC	C-	0.625	0.26	0.20	0.17	0.16
Yes	Medium QC	CA	0.750	-0.72	-0.73	-0.71	-0.67
Yes	Strong QC	C+	0.875	-3.51	-3.11	-2.18	-2.61
Yes	Conjunction	C	1.000	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Table 2: GCD functions [9]

the properties of the required aggregation technique (in this paper, GCD is considered).

4.1. Query grammar

Why is it needed?. It is difficult for a user to visualize and build complex fuzzy queries in standard SQL. There is however a lot to say for the use of the language: it can be optimized by whichever database it implements, it is portable and it is easy to debug and understand.

More specifically, by using a custom query language that translates in a target language of choice, it is easy to add optimizations with regards to the execution of the query, according to the target database. This is important, as one of the main goals of this research is to find faster ways to perform the aggregation of preferences in a database.

To investigate the use of GCD functions in a database we decided to use, as a first step, a loosely coupled [11] preprocessing step. In this preprocessing step a custom defined query is transformed into the target database's SQL syntax, after which the query is executed. In future work this will be integrated more closely into the database itself, resulting in better performance.

Input / Output. First off, we consider what input we need to process and what we want as output. As far as the input is considered it can be limited to the following:

1. A table t_i containing preferences and weights (e_i and w_i), and other optional data that might be useful for interpretation.
2. References to columns in t_i that contain values in the unit interval $[0, 1]$, that should be aggregated as preferences.
3. Optional minimum / maximum limit to be imposed on the results. This allows for further optimisations.
4. A value for the parameter r, dictating the conjunctive degree to which the aggregation will be performed.

For output we expect a result table, containing at least the following data:

1. Every row from t_i, unless it was filtered out by MIN / MAX.
2. The minimum set of columns should at least contain the preferences passed as input, together with a column that contains the aggregated preference (to be used in the ranking of the records) for the considered row.

BNF notation. The above information was compiled into the notation shown in Fragment 1. It is concise, allows for nesting and makes a rudimentary filter (MIN and MAX values to dictate which results have to be shown in the result of the query) possible. This filter can be used in later research to define optimizations. For the sake of compactness, atomic types have been assumed as defined and are shown in capital letters.

$$
\begin{array}{lcl}
< gcd > & := & \text{“}GCD\text{”} < source > \\
 & & \text{“(”} < wplist > \text{“)”} < r > \\
 & & [< min >][< max >] \\
< wplist > & := & < wp > \text{“,”} < wplist > \mid < wp > \\
< wp > & := & \text{“(”} < pref > \text{“,”} < weight > \text{“)”} \\
< source > & := & < tablename > \mid < subquery > \\
< min > & := & \text{“}MIN(\text{”} < FLOAT > \text{“)”} \\
< max > & := & \text{“}MAX(\text{”} < FLOAT > \text{“)”} \\
< tablename > & := & < STRING > \\
< subquery > & := & < STRING > \\
< preference > & := & < FLOATCOLUMN > \mid < gcd > \\
< weight > & := & < FLOAT > \\
< source > & := & < STRING > \\
< r > & := & < FLOAT >
\end{array}
$$

Fragment 1: BNF notation for GCD query

Example queries. To make the previous syntax easier to digest, a couple of examples are given in Listing 1.

```
GCD (SELECT 1 AS T, 0 AS F FROM table)
  ((T,0.3),(F,0.7)) 9.52 MIN(0.1);

GCD
  (SELECT
    CASE WHEN nearschool THEN 1 ELSE 0 END
    AS nearschoolpref,
    1 - CASE WHEN price > 300000 THEN 0
        WHEN price < 100000 THEN 1 ELSE (
         price - 100000.0) / 200000.0 END
    AS pricepref
  )
  ((nearschoolpref, 0.2),(pricepref, 0.8))
      -0.72;
```

Listing 1: 2 example GCD queries

4.2. Query preprocessing

General. Now that we have defined a grammar for our custom GCD query we can describe our transformation. Transforming the following GCD query (with preferences $\{e_1, e_2\}$ and weights $\{w_1, w_2\}$ and $n = 2$, ignoring optional parameters for now):

$$\text{GCD subquery } ((e_1, w_1), (e_2, w_2)) \text{ r}$$

assumes that *subquery* is a string that contains an SQL query, which, when executed, will return a temporary relation that contains, at minimum, attributes e_1 and e_2. In our test relation *property* there are available, among others, the following attributes: price and nearschool.

Preprocess data. As we can not use an integer (price) or a boolean (nearschool) as a preference we need to preprocess this data into preferences. This process needs to be performed, at one point or another, as it describes the opinion of the user about the considered attributes. As an example we give the elementary query shown in Listing 3.

This query results in all the preferences $\{e_i | e_i \in [0, 1], 1 \leq i \leq n\}$ that we are interested in: we like prices under 100k and dislike prices above 300k, while being near a school is described as positive.

Transformation. Now we have valid input for our problem we can use it to complete it (*subquery* is the SQL query we have constructed above):

```
SELECT sub.*, power(
  weight1*power(e1,r)+
  weight2*power(e2,r),
  1/r)
FROM (subquery) as sub
```

Optimizations. Because of the automatization of the query construction we can ensure that users get a couple of optimizations for free (in the sense that, if the conditions are right, they do not have to do anything extra to get a benefit). A couple of situations can be easily optimized when constructing the target query:

1. If $r = 1$ we can remove all power calculations from the select clause. This reduces calculation time (what is essentially calculated is the weighted mean).
2. If $r = 1 \wedge \forall i \neq j | w_i = w_j$ we can replace the weighted power mean by $avg(e_1, e_2)$.
3. If $r = +\infty$ or $r = -\infty$ we can replace the weighted power mean by $max(e_1, e_2)$ or $min(e_1, e_2)$ respectively
4. If $r < 1$ we have a mandatory situation: if $\exists e_i \in [0, 1] | 1 \leq i \leq n \wedge e_i = 0$ then the result of the WPM is always 0.

Some optimizations can only be done when the desired resultset should be filtered. There are two identified cases studied here:

1. If $\forall i \in \{1, ..., n\} : r = +\infty \wedge \text{MAX} < 1 \Rightarrow e_i \neq 1$, with MAX a value that dictates the upper limit of desired preference in the given results, n the amount of preferences e to aggregate, we can filter on the base preferences in advance: no preference should be equal to 1.
2. If $\forall i \in \{1, ..., n\} : r = -\infty \wedge \text{MIN} > 0 \Rightarrow e_i \neq 0$. Analogous to the previous case, for $r = -\infty$, no preference should be equal to 0 when demanding a minimum aggregated preference in the resultset.

Optimizations are not restricted to this method alone. Other operators with similar properties can benefit from this approach too, such as the OWA operators or the Sugeno integral. Generally, optimizing is possible if special cases can be defined that can be handled in a more efficient way.

5. Results

In this section experimental results are given. The used test platform is explained and results are given with regard to the execution times for various queries, as well as improvements that were made by applying optimalizations.

5.1. Test platform

In all tests and investigations performed in this paper a PostgreSQL database was used with a single relation, *property*, containing over 200 thousand tuples. The relation contains a large number of attributes (58) and is thus reasonably useful to see how real-life queries are handled by the system.

5.2. Measurements of optimizations

In this subsection we will report some experiments with regard to optimizations we have done. Two *subqueries* (see Listings 2 and 3) have been used as a source for the aggregation. The first one focuses on boolean preferences only (3, in this case), which should lead to fast processing due to the fact that only ones or zeros are possible values. The second

```
select propertyid,
  CASE WHEN nearschool
    THEN 1 ELSE 0
  END as nearschoolpref,
  CASE WHEN nearshops
    THEN 1 ELSE 0
  END as nearshopspref,
  CASE WHEN newestate
    THEN 1 ELSE 0
  END as newestatepref
from property
```

Listing 2: Query q_1: 3 boolean preferences

```
select propertyid,
  CASE WHEN nearschool
    THEN 1 ELSE 0
  END as nearschoolpref,
  CASE WHEN price > 300000
    THEN 0 WHEN price < 100000
    THEN 1 ELSE (price - 100000.0) /
        200000.0
  END as pricepref
from property
```

Listing 3: Query q_2: 1 boolean preference + 1 gradual preference

query however contains, apart from a boolean preference, a gradual preference as well (with possible values between 0 and 1), allowing to demonstrate the computational complexity that arises when not only boolean values have to be considered.

Impact datatype. The difference between queries q_1 and q_2 lies of course in the data being generated: the first only creates boolean values while the latter also creates a gradual value between 0 and 1. Calculating the WPM for this gradual value will be, intuitively, slower, as tested below. To allow negative r values in this test the CASE optimization was added (to prevent the following power calculations in the WPM: $x^y | y = 0 \wedge x < 0$).

To put things in perspective the execution times of q_1 and q_2 were respectively 337ms and 417ms. Timing results can be found in Table 5. Here it is still clear that q_2 takes more time to compute than q_1, which becomes obvious when calculating the WPM with a negative r: the optimization that builds on $e_i = 0$ has less of an impact, as the gradual preference in q_2 is almost never equal to 0.

Infinity optimization. When dealing with $r = +\infty$ and $r = -\infty$ the WPM can be deduced [9] to the functions MAX and MIN respectively. This offers a quick and easy optimization in the SQL query, especially when taking into account query q_2, which contains a non-boolean value as one of the preferences. Execution times can be found in Table 6. For $r \in \{-\infty, +\infty\}$, the time gained is substantial with regards to $r \notin \{-\infty, +\infty\}$.

r	Query	time (ms)
$+\infty$	q_1	421
	q_2	582
0.25	q_1	1235
	q_2	11802
1	q_1	873
	q_2	951
-3.51	q_1	417
	q_2	14256
$-\infty$	q_1	366
	q_2	565

Table 5: Execution times for varying r and query

r	Query	time (ms)
0.25	q_1	1132
	q_2	11682
$+\infty$	q_1	392
	q_2	568
$-\infty$	q_1	334
	q_2	567

Table 6: Infinity optimization: execution times

6. Future work

This work was a first step in the process of looking for a generic framework that can process any aggregation function into a SQL query, in a user friendly way. The target audience of this framework are developers, of whom can be expected to understand the inner workings of the system. Once we have established this developer framework we can work towards an interface that allows users from all levels to operate the system. This includes investigating *presets* for the framework that sacrifice functionality for transparancy.

When the framework is finished it will be interesting to see how it can be integrated into an existing database system in a tightly coupled way [11], by implementing it on a lower level. Statistical analysis will be performed in a following step as well, to check the impact of the nature and structure of the data on execution times.

Investigation of other methods within this framework will be performed, such as the OWA operators, the Sugeno integral, PostgreSQLf etc.

7. Conclusions

A query grammar was suggested that allows a user to formulate aggregations of preferences using GCD functions, according to the way the LSP method aggregated preferences. This grammar gets interpreted and translated to PostgreSQL, which can be executed on a standard database install. Nested aggregation was made possible, which allows the technique to perform complex aggregations. Optimizations were suggested and tested out, which show a definite improvement with respect to execution

times.

8. Thanks

This work is supported by the Flemish Fund for Scientific Research (FWO-Vlaanderen).

References

[1] Lorigo, L., Haridasan M., Brynjarsdóttir, H., Xia L., Joachims T., Gay G., Granka L., Pellacini F. and Pan B. Eye tracking and online search: Lessons learned and challenges ahead. *J. Am. Soc. Inf. Sci. Technol.*, 59(7):1041–1052, May 2008.

[2] Jansen B.J. and Spink A. How are we searching the world wide web?: a comparison of nine search engine transaction logs. *Inf. Process. Manage.*, 42(1):248–263, January 2006.

[3] Zadrozny, S., De Tré, G., De Caluwe, R., Kacprzyk, J. *Handbook of Research on Fuzzy Information Processing in Databases*, chapter An Overview of Fuzzy Approaches to Flexible Database Querying. In [12], 2008.

[4] Dujmović, J.J. Continuous preference logic for system evaluation. *Fuzzy Systems, IEEE Transactions on*, 15(6):1082–1099, 2007.

[5] Dujmović, J.J. A method for evaluation and selection of complex hardware and software systems. In *CMG 96 Proceedings*, pages 368–378, 1996.

[6] De Tré, G., Dujmovic, J.J., Bronselaer, A. and Matthé, T. On the applicability of multicriteria decision making techniques in fuzzy querying. In Salvatore Greco, Bernadette Bouchon-Meunier, Giulianella Coletti, Mario Fedrezzi, Benedetto Matarazzo, and Ronald R. Yager, editors, *Communications in Computer and Information Sciences*, volume 297, pages 130–139. Springer-Verlag, 2012.

[7] Bosc, P. and Pivert, O. On three fuzzy connectives for flexible data retrieval and their axiomatization. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1114–1118, New York, NY, USA, 2011. ACM.

[8] Dujmović, J.J. Characteristic forms of generalized conjunction/disjunction. In *Fuzzy Systems, 2008. FUZZ-IEEE 2008. (IEEE World Congress on Computational Intelligence). IEEE International Conference on*, pages 1075–1080, 2008.

[9] Su, S.Y.W., Dujmović, J.J., Batory, D.S., Navathe, S.B. and Elnicki, R. A cost-benefit decision model: analysis, and selection of data management. *ACM Trans. Database Syst.*, 12(3):472–520, September 1987.

[10] Dujmović, J.J., De Tré, G. and Dragicevic, S. Comparison of multicriteria methods for land-use suitability assessment. In JP Carvalho, DU Kaymak, and JMC Sousa, editors, *Proceedings of the joint 2009 International Fuzzy Systems Association world congress and 2009 European Society for Fuzzy Logic and Technology conference*, pages 1404–1409. European Society for Fuzzy Logic and Technology (EUSFLAT), 2009.

[11] Urrutia, A., Tineo L. and Gonzalez C. *Handbook of Research on Fuzzy Information Processing in Databases*, chapter FSQL and SQLf: Towards a Standard in Fuzzy Databases. In [12], 2008.

[12] J. Galindo. *Handbook of Research on Fuzzy Information Processing in Databases*. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2008.