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 (\(\land\)), disjunctions (\(\lor\)) 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 $i$th 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 $j$th tuple and $v_i \in dom_{T_i}, 1 \leq i \leq n$ the value for the $i$th 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 $\mu$, 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 $\mu_{cheap}$, 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 $\mu_{cheap}$ for property price](image)

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 $e_i$, to differentiate between the importance of individual components. Weights are normalized and sum to 1 ($w_1 + w_2 + \ldots + w_n = 1$). A possible global score can then be calculated as follows:

$$e = w_1e_1 + w_2e_2 + \ldots + w_ne_n, 0 \leq e \leq 1 \quad (1)$$

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]:

$$w_{\text{mean}}(WPM) [5, 8, 9]:$$
3. Problem description

Consider the database 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\}

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.

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 \(c\) to \([0,1]\), 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].

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
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{align*}
< \text{gcd} > & \quad := \quad "GCD" \ < \text{source} > \\
& \quad \quad "(" \ < \text{wplist} > ")" \ < \ r > \\
& \quad \quad [ < \ \text{min} > ] [ < \ \text{max} > ] \\
< \text{wplist} > & \quad := \quad < \ \text{wp} > \ | \ < \text{wplist} > \ | \ < \ \text{wp} > \\
< \ \text{wp} > & \quad := \quad "(" \ < \ \text{pref} > \ "\)," \ < \ \text{weight} > \ "\)" \\
< \ \text{source} > & \quad := \quad < \ \text{tablename} > \ | \ < \ \text{subquery} > \\
< \ \text{min} > & \quad := \quad "\ \text{MIN}(" \ < \ \text{FLOAT} > \ "\)" \\
< \ \text{max} > & \quad := \quad "\ \text{MAX}(" \ < \ \text{FLOAT} > \ "\)" \\
< \ \text{tablename} > & \quad := \quad < \ STRING > \\
< \ \text{subquery} > & \quad := \quad < \ STRING > \\
< \ \text{preference} > & \quad := \quad < \ \text{FLOATCOLUMN} > \ | \ < \ \text{gcd} > \\
< \ \text{weight} > & \quad := \quad < \ \text{FLOAT} > \\
< \ \text{source} > & \quad := \quad < \ STRING > \\
< \ r > & \quad := \quad < \ FLOAT > \\
\end{align*}
\]

Table 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.
The SQL query we have constructed above):

Now we have valid input for our

while being near a school is described as positive.

As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

4.2. Query preprocessing

General. Now that we have defined a grammar for

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Preprocess data. As we can not use an integer

Optimizations. Because of the automatization of

Optimizations. Because of the automatization of

Optimizations. Because of the automatization of

Optimizations. Because of the automatization of

Optimizations. Because of the automatization of

Optimizations. Because of the automatization of

5. Results

In this section experimental results are given. The

5.1. Test platform

In all tests and investigations performed in this paper

5.2. Measurements of optimizations

In this subsection we will report some experiments

659
The difference between queries 
Impact datatype. The difference between queries 
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.

Listing 2: Query $q_1$: 3 boolean preferences

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

$\text{select propertyid,}$ 
$\quad \text{CASE WHEN nearschool THEN 1 ELSE 0 END}$ 
$\quad \text{CASE WHEN nearshops THEN 1 ELSE 0 END}$ 
$\quad \text{END as nearschoolpref,}$ 
$\quad \text{CASE WHEN price > 300000 THEN 0 WHEN price < 100000 THEN 1 ELSE (price - 100000.0) / 200000.0 END}$ 
$\quad \text{END as pricespref}$ 
$\text{from property}$

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, PostgreSQL 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


