Novi Sad J. Math.
Vol. 41, No. 2, 2011, 161-179

THE PFSQL QUERY EXECUTION PROCESS[1]

Srdjan Škrbic[2], Miloš Racković[3], Aleksandar Takači[4]

Abstract

Usable implementations of fuzzy relational database systems are very rare considering the long time of research on the subject. The existing solutions are usually either obsolete or related to a specific application. The research group that authors belong to aims at producing a general system capable of using priority fuzzy logic with databases. The system consists of a fuzzy extension of a relational model, a fuzzy query language PFSQL for this data model, a CASE tool that allows easy creation of fuzzy relational database schemas, and an extension to JDBC driver that incorporates possibilities to use PFSQL queries.

In this paper we present details of the implementation of the PFSQL interpreter. We describe a model that we devised and point out decisions we made and tools we used in the implementation process. The result is a working implementation of the first priority fuzzy logic extension of the SQL language.

AMS Mathematics Subject Classification (2010): 68P15,68P20,68N20

Key words and phrases: PFSQL, Fuzzy database

1 Introduction

The idea to use fuzzy sets and fuzzy logic to extend the existing database models and query languages has been utilized for a long time and in different variations. Even after thirty years of research, working implementations usable in the real world systems are rare, if not non-existent.

The main goal of the five-year long research conducted at the University of Novi Sad was the implementation of a system capable of using priority fuzzy logic with databases. The system can be divided into four main parts:

- new data model based on fuzzy extensions of the relational model,
- fuzzy query language PFSQL (Priority Fuzzy Structured Query Language) for this data model and interpreter for this new language,

[1]Authors would like to acknowledge the support of the Serbian Ministry of Science, project OI–174023 – Intelligent techniques and their integration into wide-spectrum decision support and project III–47003 – Infrastructure for technology enhanced learning in Serbia.

[2]University of Novi Sad, Faculty of Science, Serbia, e-mail: shkrba@uns.ac.rs

[3]University of Novi Sad, Faculty of Science, Serbia, e-mail: milos.rackovic@dmi.uns.ac.rs

[4]University of Novi Sad, Faculty of Technology, Serbia, e-mail: stakaci@tf.uns.ac.rs

- CASE (Computer Aided Software Engineering) tool for fuzzy relational database modelling and
- JDBC (Java DataBase Connectivity) driver extensions allowing PFSQL querying from Java programs.

In this paper we give details related to the PFSQL interpreter implementation mechanisms and techniques. The first task in the process of building the PFSQL interpreter was to design the language itself. This task is described in previous papers [12, 15, 19, 21, 22] that we mention later in the text with more details. One of the results of this sequence of papers – the EBNF (Extended Backus-Naur Form) specification of the PFSQL is used here as the most important input parameter. The second input parameter was the new fuzzy relational data model. In this paper we explain how the actual implementation was done using the previous work. We give pointers to the most important decisions made in order to successfully implement the interpreter. The main result is a working implementation of the PFSQL language. We also give a comparison of our approach to the most successful competitor - the FSQL (Fuzzy Structured Query Language) language and the interpreter for it.

In the next section of this paper we give a detailed description of a number of references that describe research in the use of fuzzy logic in relational databases. This overview covers the topic from its beginnings, to the most recent approaches. The third section contains description of the existing PFSQL system. We omit numerous details and subtle points, but give references that contain complete description of the existing solution and its parts. The PFSQL interpreter details are given through three additional sections. In the first one, we describe details about the parser of PFSQL queries using JavaCC compiler compiler. After that we describe the mechanisms for PFSQL query parse tree transformation to SQL (Structured Query Language) query parse tree. In the next section, we describe details about fuzzy membership degree calculations. We finish by giving a comparison to FSQL and some concluding remarks.

2 Related work

The literature contains references to several models of fuzzy knowledge representation in relational databases. For example, the Buckles-Petry model [1] is the first model that introduces similarity relations in the relational model. The GEFRED (Generalized Model of Fuzzy Relational Databases) model [8] is a probabilistic model that refers to generalized fuzzy domains and admits the possibility distribution in domains. It includes the case where the underlying domain is not numeric, but contains scalars of any type. Also, it contains the notion of unknown, undefined and null values. It experienced subsequent expansions, such as those presented in [4, 5]. Zvieli and Chen [23] offered a first approach to incorporate fuzzy logic in the ER (Entity-Relationship) model. Their model allows fuzzy attributes in entities and relationships. Chen and Kerre [3] introduced the fuzzy extension of several major EER (Extended

Entity-Relationship) concepts. Chaudhry, Moyne and Rundensteiner [2] proposed a method for designing fuzzy relational databases following the extension of the ER model of Zvieli and Chen. They propose a way to convert a crisp database into a fuzzy one. Galindo, Urrutia and Piattini [6] describe a way to use fuzzy EER to model the database and how to represent modeled fuzzy knowledge using relational databases. In addition, they described specification and implementation of the FSQL - an advanced SQL language with fuzzy capabilities.

In [11], authors studied the possibilities to extend the relational model with fuzzy logic capabilities. The subject was elaborated in [14], where a detailed model of fuzzy relational database was given. Moreover, using the concept of Generalized Priority Constraint Satisfaction Problem (GPFCSP) from [7, 9, 16] the authors found a way to introduce priority queries into FRDB (Fuzzy Relational DataBase), which resulted in the PFSQL (Priority Fuzzy Structured Query Language) query language [15, 22]. In [12], [21] and [19], the authors introduce similarity relations on the fuzzy domain which are used to evaluate the FRDB conditions. They describe a solution for a fuzzy relational database application development and give the architecture of the PFSQL interpreter, and the data model that this implementation is based on. A brief overview of this system is given in the next section.

3 Existing solution

In order to allow the use of fuzzy values in SQL queries, we extended the classical SQL with several new elements. In addition to fuzzy capabilities that make the fuzzy SQL - FSQL, we add the possibility to specify priorities for fuzzy statements. We named the query language constructed in this manner priority fuzzy SQL - PFSQL.

The basic difference between SQL and PFSQL interpreters is in the way the database processes records. In a classical relational database, queries are executed so that a tuple is either accepted in the result set if it fulfills the conditions given in a query, or removed from the result set if it does not fulfill the conditions. In other words, every tuple is given a value true (1) or false (0). On the other hand, as the result set, the PFSQL returns a fuzzy relation on the database. Every tuple considered in the query is given a value from the unit interval. This value is calculated using the fuzzy logic operators.

3.1 The PFSQL

Details about the PFSQL syntax can be found in [14] and [19]. Here we give some pointers to what elements of the classical SQL are extended. The variables in a query should be allowed to have both crisp and fuzzy values, hence it is necessary to introduce relational operators between different types of fuzzy values as well as between fuzzy and crisp values. Next, we introduced and used the possibility based ordering on the set of fuzzy numbers in the queries.

This possibility gives grounds for the introduction of set functions like MIN, MAX and COUNT in the PFSQL. The Classical SQL includes the possibilities to combine conditions using logical operators. This possibility is, of course, also the part of our fuzzy extensions, thus combining fuzzy conditions is also a feature of our implementation. Values are calculated using t-norms, t-conorms, and a "strict" negation. Queries are handled using priority fuzzy logic based on the GPFCSP (Generalized Priority Fuzzy Constraint Satisfaction Problem) systems [7, 9, 10, 14].

In the classical SQL it is clear how to assign a truth value to every elementary condition. With the fuzzy attributes, the situation becomes more complex. At first, we assign a truth value from the unit interval to every elementary condition. The only way to do this is to implement a set of algorithms that calculate truth values for every possible combination of values in a query and values in the database. For instance, if a query contains a condition that compares a fuzzy quantity value with a triangular fuzzy number in the database, we must have an algorithm that calculates the compatibility degree of the two fuzzy sets. After the truth values from the unit interval are assigned, they are aggregated using fuzzy logic. We use a t-norm in case of the operator AND, and its dual t-conorm in case of the operator OR. For negation we use the strict negation: $N(x) = 1 - x$. In case of priority statements, we use the GPFCSP system rules to calculate the result.

We will now describe the processes that allow PFSQL queries to be executed. The basic idea is to first transform a PFSQL query into an SQL query. Namely, the conditions with fuzzy attributes are removed from the WHERE clause and moved up in the SELECT clause. In this way, the conditions containing fuzzy constructs are eliminated, so that the database will return all the tuples - ones that fulfill fuzzy conditions as well as the ones that do not. As a result of this transformation, we get a classical SQL query. Then, when this query is executed against the database and results are interpreted using fuzzy mechanisms. These mechanisms assign a value (membership degree) from the unit interval to every tuple in the result set. If a threshold is given, all the tuples in the result set that have satisfaction degree below the threshold are removed.

3.2 Fuzzy-Relational Database

It is clear now that the PFSQL implementation has to rely upon a metadata that describe fuzzy attributes that reside inside the database. For these purposes, a FRDB data model has been defined. In this section we give a brief description of this model.

Our FRDB data model allows data values to be any fuzzy subset of the attribute domain. User only needs to specify a membership function of a fuzzy set. Hypothetically, for each fuzzy set we should have an algorithm on how to calculate the values of its membership function. This would lead to a large spatial complexity of the database and that is why only a well known standard types of fuzzy sets (triangular, trapezoidal, etc.) are allowed as attribute values. If a type of a fuzzy set is introduced, then we only need to store the parameters

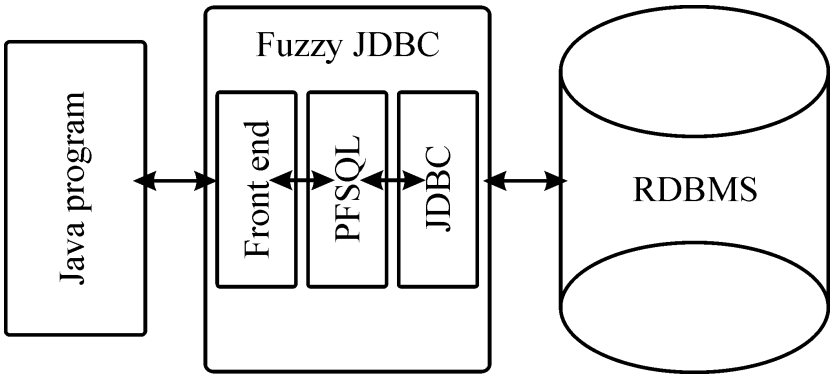

Figure 1: Fuzzy JDBC driver.

that are necessary to calculate values of its membership function.

Also, we introduced one more extension of the attribute value, the linguistic label. Linguistic labels are in fact named fuzzy values from the domain. Considering these extensions, we can define a domain of a fuzzy attribute as $D = D_C \cup F_D \cup L_L$, where D_C is a classical attribute domain; F_D is a set of all fuzzy subsets of the domain, and L_L is the set of linguistic labels.

In order to represent these fuzzy values in the database, we extend this model with additional tables that make fuzzy metadata model. Several tables are introduced to cover all described needs. The structure of this model is not important for this paper, so we do not give it here, but it can be found in [14].

3.3 Fuzzy JDBC Driver and FRDB CASE Tool

The need to allow easy PFSQL usage from Java programs and still keep the database independence is resolved with the implementation of the fuzzy JDBC driver. This driver acts as a wrapper for the PFSQL processing mechanisms described earlier and for the JDBC API implemented by the driver for a specific RDBMS (Relational DataBase Management System). The JDBC driver for the database used simply becomes a parameter that the fuzzy JDBC driver uses to access the database. The architecture of the system built in this way is shown at Fig. 1.

Java program uses interfaces offered by the fuzzy JDBC driver as a front end component. These interfaces include possibilities to: initialize driver class, create database connection, create and execute PFSQL statements, and read result set.

When executed, PFSQL statements are pre-processed in the way described before, and sent to the database as ordinary SQL statements using a JDBC driver. Result returned from the database is processed again by the PFSQL mechanisms (membership degrees are added), and returned to the Java program using front end classes.

The Fuzzy JDBC driver with PFSQL mechanisms and the FRDB data model described above offer a solution to develop database applications when a database schema exists in a database. In order to ease the development of database schemas enriched with fuzzy elements, a CASE tool is implemented too. Again, the description of the CASE tool is omitted because it is not important for the goals set in this paper.

4 The PFSQL parser

This section contains details about how the PFSQL query parser was built. The EBNF specification is presented in [19], and we do not give it in full here. Instead, we give a few examples that illustrate main PFSQL features. The queries are executed against a hypothetical student database. The first query returns names and surnames of students whose GPA is greater than the given triangular fuzzy number:

```
SELECT msd.name, msd.surname
FROM MainStudentData msd
WHERE (msd.GPA>#triangle(9,1,0.4)#)
```

The # symbol is chosen to mark fuzzy constants. If we defined a linguistic label average GPA that has value triangle(9,1,0.4), the previous query could be simplified like this:

```
SELECT msd.name, msd.surname
FROM MainStudentData msd
WHERE (msd.GPA>#ling(averageGPA)#)
```

Queries like these can be enriched with additional constraints. The next query returns names and surnames of students that have GPA greater than average with priority 0.7, and GPA on the fourth year greater than 8.50 with priority 0.4. The query also contains the threshold clause that limits the results and removes tuples with fuzzy satisfaction degree smaller than 0.2.

```
SELECT msd.name, msd.surname
FROM MainStudentData msd
WHERE (msd.GPA>#ling(averageGPA)# PRIORITY 0.7)
  AND (msd.GPA4>8.5 PRIORITY 0.4)
THRESHOLD 0.2
```

The aggregate functions MAX, MIN and COUNT can take fuzzy value as an argument. The next query illustrates the usage of aggregate function MIN. It returns the minimal GPA.

```
SELECT MIN(msd.GPA)
FROM MainStudentData msd
```

If we assume that the variable msd.GPA is fuzzy, the execution of this query becomes complex because it includes the ordering of fuzzy values. As a result, for example, we could get this value: triangle(6.9,0.4,0.7).

The parser is constructed using JavaCC compiler compiler tool. This tool automatically generates a parser for a given EBNF grammar. However, in this case, it is not enough to simply specify a parser, it is necessary to use mechanisms offered by JavaCC through JJTree preprocessor. Using the JJTree, it is possible to generate a code that creates a parse tree without losing any information contained in the original query. It is necessary to:

- alter the parse tree nodes generated automatically by the JJTree so that they can store additional information contained in the query, and
- alter the JavaCC specification so that it includes semantic actions that will copy all the information contained in the query to these structures.

Listing 1 contains the JavaCC code for the main part of the SELECT clause enriched with additional semantic actions.

Listing 1: A snippet from the PFSQL JavaCC specification enriched with JJTree semantic actions.

```
void SelectWithoutOrder():
{Token select;
 Token distinct;}{
  select=<SELECT>{jjtThis.setSelect(select.image);}
  [distinct=<DISTINCT>{jjtThis.setDistinct(distinct.image);}]
  SelectList()
  FromClause()
  [WhereClause(){jjtThis.setHasWhere(true);}]
  [GroupByClause(){jjtThis.setHasGroup(true);}]
  [ThresholdClause(){jjtThis.setHasThreshold(true);}]
}
void SelectList():{}{
  <MULT>{jjtThis.setMult(true);}
  | (ObjectName()|AggregateFunction())(","(ObjectName()
    |AggregateFunction()){jjtThis.setMultipleHead(true);})*
}
void FromClause():{}{  <FROM>FromItem()(","FromItem(){jjtThis.
    setMultipleHead(true);})*
}
void FromItem():
{Token t1,t2;}{
  t1=<IDENTIFIER>{jjtThis.setIdent1(t1.image);}
  [t2=<IDENTIFIER>{jjtThis.setIdent2(t2.image);}]
}
void WhereClause():{}{
  <WHERE>SQLExpression()
}
void SQLExpression():
{Token t;}
{
SQLAndExpression()(t=<OR>{jjtThis.setOrStr(t.image);}SQLAndExpression()
    )*
}
```

```
void SQLAndExpression():
{Token t;}
{  SQLUnaryLogicalExpression()(t=<AND>{jjtThis.setAndStr(t.image);}
    SQLUnaryLogicalExpression())*
}
void SQLUnaryLogicalExpression():
{Token t;}
{
  LOOKAHEAD(2)ExistsClause()
  |([t=<NOT>{jjtThis.setNotStr(t.image);}]SQLRelationalExpression(){
      jjtThis.setExistStatement(false);})
}
```

Let us observe the beginning of the code, around the DISTINCT keyword. Construction shown in the listing puts the DISTINCT keyword into the corresponding node of the parse tree if, during the parsing, this keyword is encountered. This is achieved by calling the setDistinct() method over the current node in the tree, denoted by jjtThis. Similarly, with WHERE clause, the method setHasWhere() is called over the current node, that specifies that WHERE clause exists in the query. These methods are not generated automatically, they are added to the generated code by hand.

The parse tree is built of different types of nodes that are instances of classes that describe the corresponding constructions of the JavaCC specification. Except empty nodes of different types, these nodes contain all the information from the query that they represent.

For example, let us observe the PFSQL query shown in Listing 2.

Listing 2: An example the PFSQL query.

```
SELECT msd.name, msd.surname
FROM MainStudentData msd
WHERE msd.GPA>'#triangle(4,1,0.4)#' PRIORITY 0.8 AND msd.IDnumber<200
```

This query returns names and surnames of the students that have a GPA greater than the triangular fuzzy number triangle(4,1,0.4) with priority 0.8 and have an ID number smaller than 200. A complete parse tree for this query is given in Fig. 2.

The obtained PFSQL parser successfully checks the PFSQL query syntax and returns appropriate error messages as needed. In the process, begging with the query string, it builds the parse tree that preserves every information contained in the original query. Moreover, an important feature of transforming the parse tree back to query string is also implemented.

5 Parse Tree Transformation

The possibility to transform the parse tree back to a query string mentioned in the previous section is vital for the PFSQL query processing. As mentioned before, the PFSQL query processing includes transformation operations on the parse tree, and its conversion back to the query string at the end of the process. The obtained query string is then sent to the relational database for further

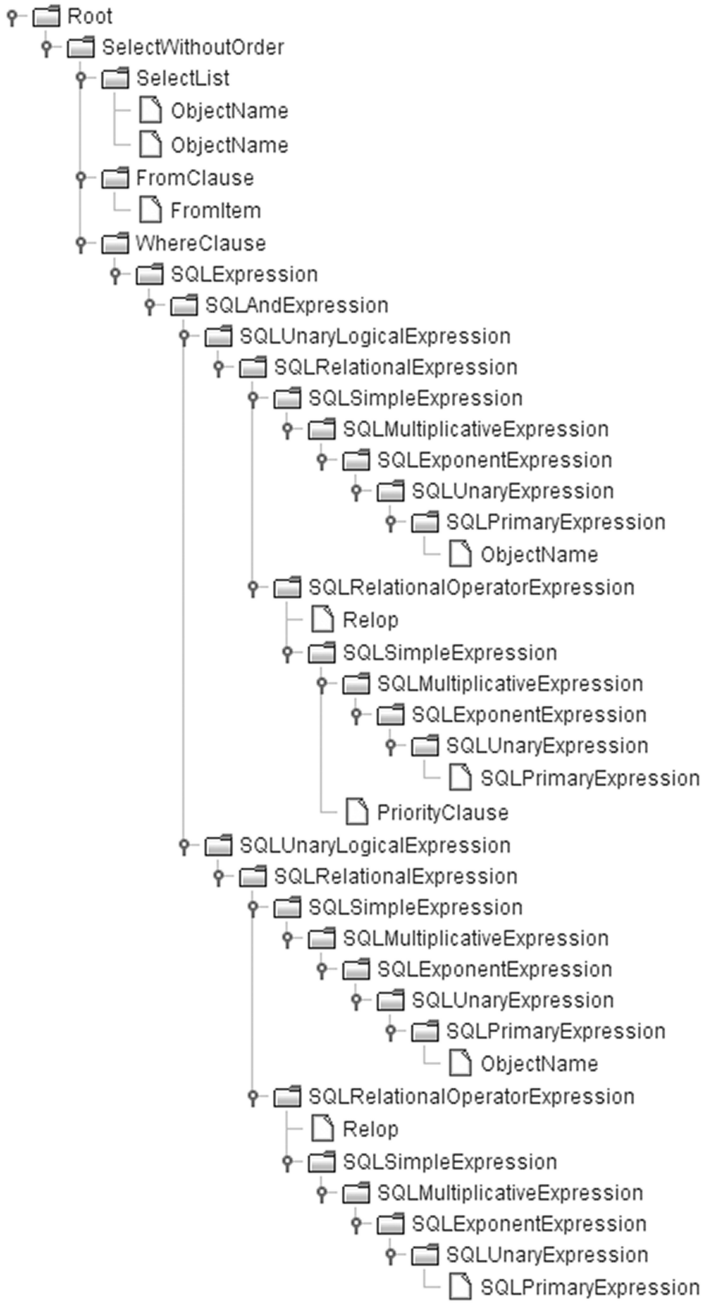

Figure 2: Parse tree for the PFSQL query from Listing 2.

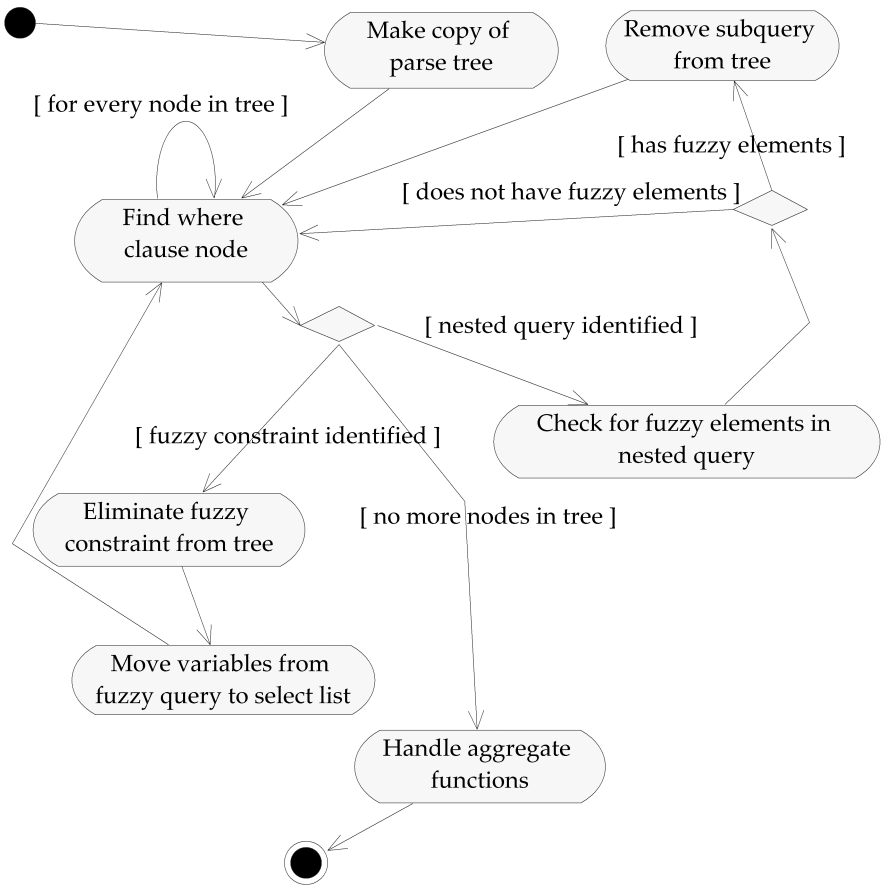

Figure 3: Activity diagram describing transformation of a PFSQL query parse tree.

processing and returned results are interpreted using priority fuzzy logic. More details on the parse tree transformation method is given in the next section.

The PFSQL query parse tree transformation and fuzzy membership degree calculations are the two most demanding topics from the implementation point of view. In this section we show the UML activity diagrams that clearly define which actions have to be taken to achieve specific goals. More details about the implementation and complete source code may be downloaded from [17].

The activity diagram that describes the process of PFSQL query transformation to SQL query is shown in Fig. 3. Since an initial parse tree is needed in the process of fuzzy membership degree calculations, we are transforming its copy only. The transformation consists of a recursive tree traversal that includes the modification of certain elements when the need to modify them is detected.

Let us observe the parse tree node that represents a WHERE clause. If it is identified that some node in this node's subtree represents a fuzzy constraint, as mentioned before, it is removed from the tree. It is not enough only to remove such a node from the tree, it is necessary to inspect if it contains fuzzy variables as well. If it does contain fuzzy variables, it is necessary to add them to the SELECT list in the same tree, in order to obtain their value from the database. The fuzzy constraint must have a value–relational operator–value structure, as defined by the PFSQL EBNF syntax. This means that the operations with fuzzy values are not supported by the PFSQL language at this time.

Nested queries are constructions that need to be identified and treated separately. If a nested query is found, then we inspect if it contains any fuzzy variables. If it does not contain them, then there is no need for special treatment, and it is left as it is. But, if it contains fuzzy variables, it is removed from the tree. It will be executed recursively as part of the fuzzy membership degree calculation algorithm. The PFSQL query language currently supports only non-correlated nested queries (independent from the rest of the query). If a correlated nested query is identified, an appropriate error message is returned.

If we do not detect a fuzzy constraint or nested query while processing a node, we continue with the analysis of this node's children, until the complete tree is traversed.

In the end, it is also necessary to process the appearance of aggregate functions in the select list. If it is detected that some aggregate function contains fuzzy variable as argument, then the whole aggregate function construction is replaced by that fuzzy variable. In this way, we obtain values for that fuzzy variable from the database, and calculate the value of the aggregate function in the next processing step. At this time, only MIN, MAX and COUNT aggregate functions are supported. If the fuzzy variable appears under any other aggregate function, an appropriate error message is returned. Also, the GROUP BY construct with fuzzy values is not allowed in the current implementation, although various possibilities to do this are discussed in [18].

In accordance to the described parse tree transformation method, the query from Listing 2 and its corresponding parse tree from Fig. 2 would take the form shown in Listing 3 and Fig. 4.

Listing 3: SQL query obtained after the transformation of the example of PFSQL query.

```
SELECT msd.name, msd.surname, msd.GPA
FROM MainStudentData msd
WHERE msd.IDnumber<200
```

6 Fuzzy Membership Degree Calculations

After the parse tree transformation described in the previous section, an SQL string is acquired and sent to the database for execution. The Database returns a result set that, in general, contains multiple tuples. At this moment it is

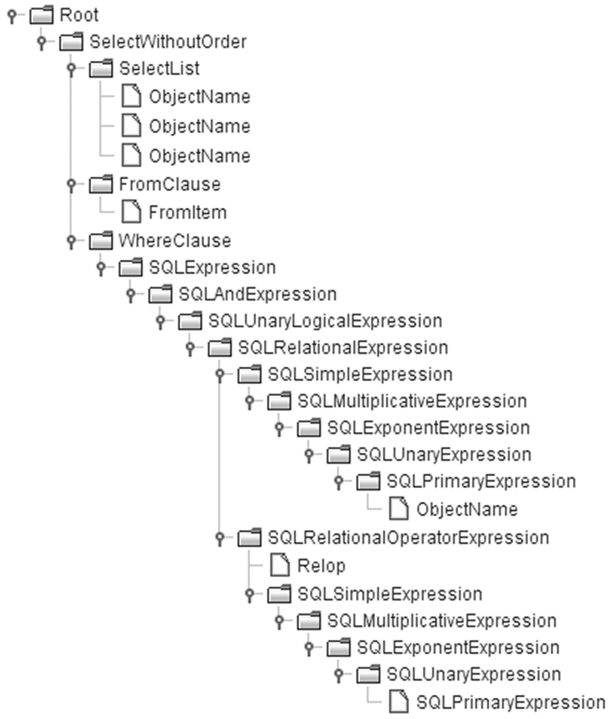

Figure 4: Parse tree after transformation.

necessary to evaluate all of the tuples, one by one, and assign fuzzy membership degree to each of them using information from the original PFSQL query. In the case that the original PFSQL query contained an aggregate function with fuzzy argument, instead of assigning a fuzzy membership degree to the tuples, we use them to calculate the value of the aggregate function.

The activity diagram that specifies this process is given in Fig. 5.

In the case that the original PFSQL query contains an aggregate function with a fuzzy attribute, the result set is analyzed. If the aggregate function is COUNT, then simply, the number of tuples in the result set is determined. In the case that the aggregate functions MIN or MAX are found, the fuzzy values are compared in order to find a minimum or maximum. This minimum or maximum may consist of more than one value, having in mind that we use a partial order for the fuzzy sets [14, 18].

If there is no aggregate function, we enter the second branch of the algorithm that assigns a fuzzy constraint satisfaction degree to every tuple returned by the database using the original parse tree. At the beginning, we search the parse tree and locate a node that corresponds to the WHERE clause. We start the calculations from that point.

We calculate the fuzzy membership degree of all subnodes recursively. In

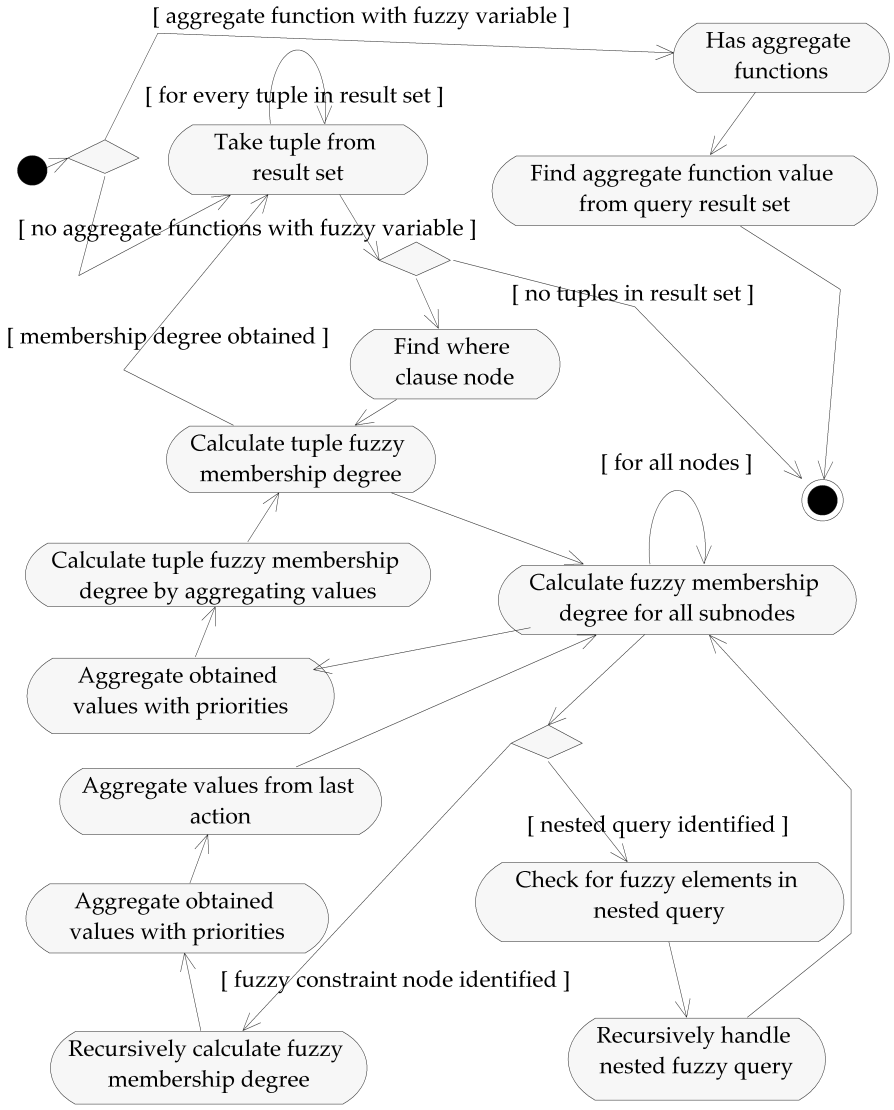

Figure 5: Activity diagram specifying calculations of fuzzy membership degree for result set tuples.

this process we may encounter a node with a fuzzy condition, as well as a node that contains a nested query with fuzzy elements. The fuzzy condition value calculation is shown in a separate activity diagram later in the text. If a priority is assigned, the calculated value is first aggregated with it. This aggregation is done using the S_p ("product") triangle conorm in the following way. If we

denote the obtained value of the fuzzy condition as x, and its assigned priority as p, then the aggregated value is given by $S_p(x, 1-p)$. The values obtained in that way are then further aggregated using triangle norm T_L (Łukasiewicz) in the case of the operator AND, and its dual triangle conorm S_L in the case of the operator OR. If a negation is encountered, we use the so-called strict negation: $N(x) = 1 - x$. Details about these calculations that include definition of basic terms and formal explanations why we calculate values in this way are given in [10, 16, 18].

In the case that a nested query with fuzzy elements is encountered, it is executed independently, recursively, in the same way as the main query. The acquired results are then used in further calculations related to the node where this kind of content is encountered. According to the PFSQL syntax, it is possible that the operators EXISTS, ALL, ANY and IN appear before a nested query. In the case of the operator EXISTS, simply, we check whether the set returned from a nested query is empty or not. In the case of the operator IN, in general, it is necessary to check whether a fuzzy value on the left side can be found in the result set on the right side. Finally, in the case of the operators ALL and ANY, we compare the value from the left side of the operator to every value from the nested query result set. This task is reduced to the process of evaluation of fuzzy conditions described below.

The need to evaluate fuzzy satisfaction degrees demanded that the methods that calculate triangular norm and conorm values be implemented also. However, in the case a user needs to use some other t-norm or t-conorm, not implemented in the system, it is enough to write the corresponding methods by hand and replace the existing methods with the new ones.

We still need to clarify the way to calculate fuzzy satisfaction degrees for fuzzy conditions. Fuzzy conditions are expressions that consist of two (possibly) fuzzy operands with a relational operator between them. Relational operators defined in the PFSQL syntax can be divided into two groups:

- operators $=, != , <>$ and
- operators $<, >, \leq, \geq$.

The allowed operands for these operators are fuzzy or crisp variables and constants. In accordance with the PFSQL syntax, fuzzy constants have the following structure:

- $triangle(max, leftOffset, rightOffset)$ – triangle fuzzy number with maximum in max and left and right offsets denoted by $leftOffset$ and $right - Offset$,
- $trapezoid(leftMax, rightMax, leftOffset, rightOffset)$ – trapezoidal fuzzy number with maximum on the $[leftMax, rightMax]$ interval and left and right offsets denoted by $leftOffset$ and $rightOffset$,
- $interval(left, right)$ – interval with left and right boundaries given by $left$ and $right$,

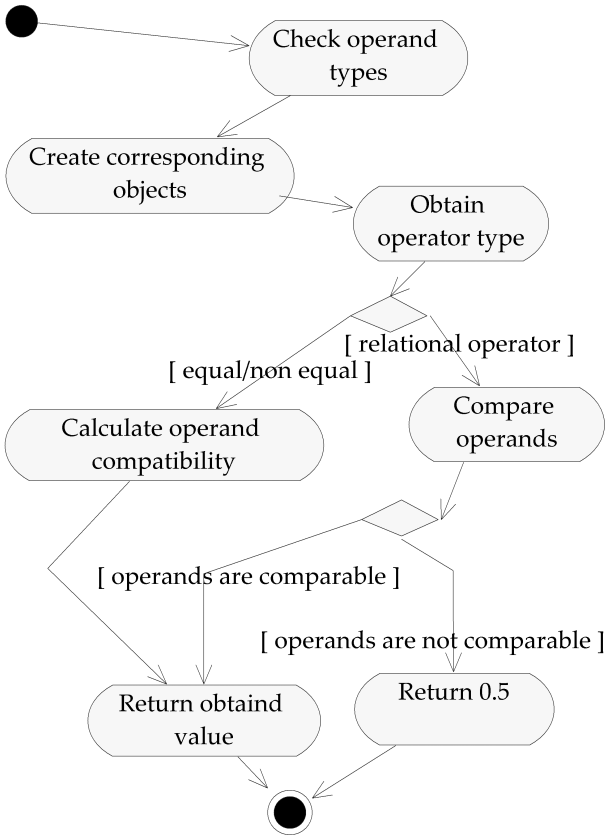

Figure 6: Activity diagram specifying the process of fuzzy condition fuzzy membership degree calculation.

- $fq(left, right, isIncreasing)$ – fuzzy shoulder that ascends or descends from $left$ to $right$,
- $ling(name)$ – linguistic label with name given by $name$.

The activity diagram shown in Fig. 6 specifies the process of fuzzy condition fuzzy membership degree calculation under presented assumptions.

At the beginning of the process, we check what types of operands are present, to determine if there are any fuzzy constants or variables. Based on this analysis, in the next step, the corresponding objects that represent those operands are created. In essence, for every operand type, we create a class that enables storage of the needed values for that operand type. For example, in the case of a triangular fuzzy number, we store the values $leftOffset$, max and $rightOffset$. In the case of constants, these objects are created by parsing the string, while in the case of variables, the values are taken from the result set. If a value of a fuzzy

variable is obtained from the database, the obtained result set contains only a reference to the specific fuzzy value that resides inside the fuzzy metamodel part of the database. So, the fuzzy metamodel has to be consulted first, in order to obtain a fuzzy value.

When the corresponding objects for the operands are created, we check the type of the operator. In the case the operator belongs to the first group (equals/nonequals), it is necessary to apply an algorithm for calculation of fuzzy compatibility of the two sets. If the operator belongs to the other group, then we apply an algorithm for comparison of two fuzzy sets. These algorithms are explained in detail in [14, 18].

It is conceptually simple to calculate a compatibility value for two fuzzy sets, but the implementation is technically demanding. It is necessary to implement algorithms that calculate the intersection of every pair of types of fuzzy sets that we support in our system. For instance, a class that serves for the representation of a triangular fuzzy number also contains methods that calculate a compatibility degree of a triangular fuzzy numbers and trapezoidal fuzzy numbers, intervals, fuzzy shoulders and crisp values. The same holds for all other supported types of values. Result of these calculations is always a number from the unit interval.

In the case of comparison of two fuzzy sets, the algorithm can return:

- value 1, if the first set is less than or equal to the second one,
- value 0, if the first set is not less than or equal to the second one and
- value 0.5, if the two sets are not comparable.

7 PFSQL and FSQL

At the end, we give a brief comparison of the PFSQL interpreter implementation mechanisms and FSQL implementation techniques. The FSQL is considered to be one of the most advanced fuzzy SQL languages today. The fuzzy database query language FSQL is built on top of the FIRST-2 model using Oracle DBMS and PL/SQL stored procedures [6]. Similarly, we used our own fuzzy-relational data model described in [13] and [14] to build an interpreter for the PFSQL language. The PFSQL query language allows priority statements to be specified for query conditions, which is a new feature that FSQL does not contain. We use the GPFCSP concept for calculating membership degrees of query tuples when priorities are assigned to the conditions. The GPFCSP is a theoretical concept developed just for these purposes. Although the FSQL language has more features than PFSQL, it does not allow usage of priority statements. The PFSQL is the first query language that does. Moreover, the PFSQL is implemented using Java, outside the database, which makes our implementation database independent.

8 Concluding remarks

The main concern of this paper is the interpreter for the priority fuzzy logic enriched SQL language – PFSQL. We gave details related to its implementation and described main problems and decisions made during its construction. In an effort to compare our approach to the FSQL – the most advanced competitor, we concluded that the PFSQL is the first language that adds priority fuzzy logic extensions to SQL.

The described implementation is tested on the fuzzified database segment of the student affairs information system of the Faculty of Science in Novi Sad. This segment is related to the entrance examination management. It consists of about 30 relational tables with more than 5000 records in most of them. PFSQL has not been used in practice so far. We plan to do that in the future. Moreover, in order to offer a more complete solution for the fuzzy relational database application development, it is necessary to enrich the PFSQL language with more features of a regular SQL. Authors intend to study and solve these problems in the future.

References

[1] Buckles, B., Petry, F., A fuzzy representation of data for relational databases. Fuzzy Set. Syst. 7, 3 (1982), 213–226.

[2] Chaudhry, N., Moyne, J., Rundensteiner, E., A design methodology for databases with uncertain data. In: Proc. 7th Intl. Working Conf. Scientific Statistical Database Management (Charlottesville, VA, 1994), pp. 32–41.

[3] Chen, G., Kerre, E., Extending ER/EER concepts towards fuzzy conceptual data modelling. In: Proc. IEEE Intl. Conf. Fuzzy Syst. (Anchorage, AK, 1998), pp. 1320–1325.

[4] Galindo, J., Medina, J., Aranda, M., Querying fuzzy relational databases through fuzzy domain calculus. Int. J. Intell. Syst. 14, 4 (1999), 375–411.

[5] Galindo, J., Medina, J., Cubero, J., Garcia, M., Relaxing the universal quantifier of the division in fuzzy relational databases. Int. J. Intell. Syst. 16, 6 (2001), 713–742.

[6] Galindo, J., Urrutia, A., Piattini, M., Fuzzy Databases: Modelling Design and Implementation. IDEA Group, Hershey, PA, 2006.

[7] Luo, X., Lee, J., Leung, H., Jennings, N., Prioritized fuzzy constraint satisfaction problems: Axioms, instantiation and validation. Fuzzy Set. Syst. 136, 2 (2003), 151–188.

[8] Medina, J., Pons, O., Vila, M., GEFRED: A generalized model of fuzzy relational databases. Inform. Sciences 76, 1–2 (1994), 87–109.

[9] Takači, A., Schur-concave triangular norms: Characterization and application in PFCSP. Fuzzy Set. Syst. 155, 1 (2005), 50–64.

[10] Takači, A., Perović, A., Jovanović, A., Measuring uncertainty with priority based logic. In: Proc. 12th Intl. Conf. Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU 2008) (Malaga, Spain, 2008), pp. 1490–1496.

[11] Takači, A., Škrbić, S., How to implement FSQL and priority queries. In Proc. 3rd Serbian-Hungarian Joint Symposium Intelligent Systems and Informatics (Subotica, Serbia, 2005), pp. 261–267.

[12] Takači, A., Škrbić, S., Measuring the similarity of different types of fuzzy sets in FRDB. In: Proc. 5th Conf. EUSFLAT (Ostrava, Czech Republic, 2007), pp. 247–252.

[13] Takači, A., Škrbić, S., Short review of fuzzy relational databases. In: Proc. 51nd ETRAN Conf. (Herceg Novi, Montenegro, 2007), p. VI1.4(CD).

[14] Takači, A., Škrbić, S., Data Model of FRDB with Different Data Types and PFSQL, vol. 1. USA: Information Science Reference, Hershey, PA, 2008, pp. 407–434.

[15] Takači, A., Škrbić, S., Priority, weight and threshold in fuzzy SQL systems. Acta Polytech. Hung. 5, 1 (2008), 59–68.

[16] Takači, A., Škrbić, S., Perović, A., Generalised prioritised constraint satisfaction problem. In: Proc. 7th Serbian-Hungarian Joint Symposium Intelligent Systems and Informatics (Subotica, Serbia, 2009), pp. 145–148.

[17] Škrbić, S., Application of fuzzy logic in database systems. `http://www.is.pmf.uns.ac.rs/fuzzydb/`, 2007.

[18] Škrbić, S., Using fuzzy logic with relational databases. PhD dissertation, Faculty of Science, Novi Sad, Univerity of Novi Sad, 2009.

[19] Škrbić, S., Racković, M., PFSQL: a fuzzy SQL language with priorities. In: Proc. PSU-UNS Inter. Conf. on Engineering Technolgies, ICET 2009 (Novi Sad, Serbia, 2009), pp. 58–63.

[20] Škrbić, S., Racković, M., Takači, A., Towards the methodology for development of fuzzy relational database applications. Comp. Sci. and Inf. Sys. 8, 1 (2011), 27–40.

[21] Škrbić, S., Takači, A., On development of fuzzy relational database applications. In: Proc. 12th Intl. Conf. Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU 2008) (Malaga, Spain, 2008), pp. 268–273.

[22] Škrbić, S., Takači, A., An interpreter for priority fuzzy logic enriched SQL. In: Proc. 4th Balkan Conf. Informatics (Thessaloniki, Greece, 2009), pp. 96–100.

[23] Zvieli, A., Chen, P., ER modelling and fuzzy databases. In: Proc. 2nd Intl. Conf. Data Engineering (Los Angeles, CA, 1986), pp. 320–327.

Received by the editors December 22, 2010