


# Extending the Doctrine ORM Framework Towards Fuzzy Processing of Data Exemplified by Ambulatory Data Analysis

Bożena Małyśiak-Mrozek, Hanna Mazurkiewicz, and Dariusz Mrozek 

Institute of Informatics, Silesian University of Technology,  
ul. Akademicka 16, 44-100 Gliwice, Poland  
{bozena.malysiak,dariusz.mrozek}@polsl.pl

**Abstract.** Extending standard data analysis with the possibility to formulate fuzzy search criteria and benefit from linguistic terms that are frequently used in real life, like *small*, *high*, *normal*, *around*, has many advantages. In some situations, it allows to extend the set of results by similar cases that would not be possible or difficult with precise search criteria. This is especially beneficial when analyzing biomedical data, where sets of important measurements or biomedical markers describing particular state of a patient or person have similar, but not the same values. In other situations, it allows to generalize the data and aggregate it, and thus, quickly reduce the volume of data from Big to small. Extensions that allow the fuzzy data analysis can be implemented in various layers of the database client-server architecture. In this paper, on the basis of the ambulatory data analysis, we show extensions to the Doctrine object-relational mapping (ORM) layer that allow for fuzzy querying and grouping of crisp data.

**Keywords:** Databases · Fuzzy sets · Fuzzy logic · Querying · Information retrieval · Biomedical data analysis · Object-relational mapping · ORM

## 1 Introduction

Nowadays people live faster and more dynamically. Stress affects almost everyone and sleep deprivation and neurosis occur even in children. Unfortunately, this way of life is directly related to the appearance of various civilization diseases, such as: heart disease, diabetes, nervousness, cancer, allergies. Fortunately, the human conscience in this regard and attention to health are growing. People more often report to doctors and often are sent for laboratory testing. Moreover, they often are in control of their health, performing basic laboratory tests, such as: morphology, erythrocyte sedimentation rate (ESR), blood sugar, lipid profile, or by measuring independently blood pressure at home. The majority of these tests are performed in medical laboratories, which services range from routine testing, such as basic blood counts and cholesterol tests, to highly complex methods that

assist in diagnosing genetic disorders, cancers, and other rare diseases. Almost 70% of health care decisions are guided by lab test results.

Laboratories must keep information about the examined people – their personal data and the results of laboratory tests together with ranges for particular test types. These results more often have numeric character.

Some examples of laboratory test are as follows:

– Blood sugar

Normal blood sugar levels are as follows:

- Between 4.0 to 6.0 mmol/L (72 to 108 mg/dL) when fasting
- Up to 7.8 mmol/L (140 mg/dL) 2 h after eating

But for people with diabetes, blood sugar level targets are as follows:

- Before meals: 4 to 7 mmol/L for people with type 1 or type 2 diabetes
- After meals: under 9 mmol/L for people with type 1 diabetes and under 8.5 mmol/L for people with type 2 diabetes

– Lipid profile

The lipid profile is used as part of a cardiac risk assessment to help determine an individual's risk of heart disease and to help make decisions about what treatment may be best, if there is a borderline or high risk. A lipid profile typically includes the following tests:

- LDL Cholesterol
  - \* Optimal: Less than 100 mg/dL (2.59 mmol/L)
  - \* Near/above optimal: 100–129 mg/dL (2.59–3.34 mmol/L)
  - \* Borderline high: 130–159 mg/dL (3.37–4.12 mmol/L)
  - \* High: 160–189 mg/dL (4.15–4.90 mmol/L)
  - \* Very high: Greater than 190 mg/dL (4.90 mmol/L)
- Total Cholesterol
  - \* Desirable: Less than 200 mg/dL (5.18 mmol/L)
  - \* Borderline high: 200–239 mg/dL (5.18 to 6.18 mmol/L)
  - \* High: 240 mg/dL (6.22 mmol/L) or higher
- HDL Cholesterol
  - \* Low level, increased risk: Less than 40 mg/dL (1.0 mmol/L) for men and less than 50 mg/dL (1.3 mmol/L) for women
  - \* Average level, average risk: 40–50 mg/dL (1.0–1.3 mmol/L) for men and between 50–59 mg/dl (1.3–1.5 mmol/L) for women
  - \* High level, less than average risk: 60 mg/dL (1.55 mmol/L) or higher for both men and women
- Fasting Triglycerides
  - \* Desirable: Less than 150 mg/dL (1.70 mmol/L)
  - \* Borderline high: 150–199 mg/dL (1.7–2.2 mmol/L)
  - \* High: 200–499 mg/dL (2.3–5.6 mmol/L)
  - \* Very high: Greater than 500 mg/dL (5.6 mmol/L)

The volume of such ambulatory data obtained in laboratory tests is large, and the data must be stored in a repository. In biomedical laboratories this is usually a relational database that holds all the data. Similarly, when people

make measurements in their homes, for example blood pressure, the amount of data related to measurements can be huge, especially, when the data are collected remotely by an external system. Frequently it happens that a doctor recommends measuring the patient blood pressure three times a day, and the patient must record all measurements and report them to the doctor, or those measurements are automatically sent to doctors through a telemedicine system to monitor patient and provide clinical health care from a distance. No matter where the data is stored, access to it by searching and retrieving appropriate patients' records should be quick.

Since results of various laboratory tests should fall into certain ranges or may exceed them, people, including medical doctors, usually use common terms, like *normal*, *above* or *below* to describe levels of particular biochemical markers. This provides a good motivation to apply such a logic in computer processing of the data, which would be able to appropriately assign particular values to a certain range, or mark them as going beyond the range: above or below. These conditions can be met by using fuzzy logic [20, 21], which becomes particularly handy when dealing with Big Data sets containing results of various ambulatory tests. The fuzzy logic allows to generalize the data, which is of great importance in large medical screenings.

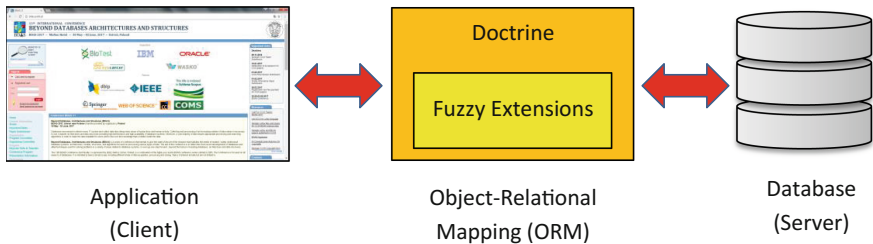
## 2 Fuzzy Logic in Data Processing and Analysis

Extending standard data analysis with the possibility to formulate fuzzy search criteria and benefit from imprecise and proximity-based linguistic terms that are frequently used in real life, like *small*, *high*, *normal*, *around*, *near* has many advantages. In some situations, it allows to extend the set of results by similar cases, which would not be possible, or at least, difficult with precise search criteria. This is especially beneficial when analyzing biomedical data [16], where sets of important measurements or biomedical markers, e.g., blood pressure, BMI, cholesterol, age, describing particular state of a patient or person may have similar, but not the same values. Enriching the set of results with similar cases can be then very helpful in drawing appropriate conclusions, reporting on important lesions, suggesting certain clinical actions, and preparing similar treatment scenarios for patients with similar symptoms. On the other hand, incorporating routines for fuzzy processing in the data analysis pipeline allows to generalize the data, group it and aggregate, or classify and assign to clusters or subgroups, and thus, change the granularity of information that we have to deal with. This provides a way to quickly reduce the volume of data from big to small, which is highly required in the era of Big Data.

### 2.1 Related Works

Extensions that allow fuzzy processing, querying and data analysis can be implemented in various layers of the database client-server architecture (Fig. 1). On the client side, various software tools and applications may incorporate fuzzy

extensions as procedures and functions that are parts of the software, bound to particular controls of the Graphical User Interface (GUI) or invoked internally from other functions. This has several advantages, including adaptation of the fuzzy procedures to the object-oriented environment (which is frequently used in development of such applications), and adjustment to the specificity of the application and processed data. The fuzzy extensions seem to be tailored to data being processed. The huge disadvantage of such a solution is a tight coupling to a particular application and negligible re-usability of procedures for fuzzy processing for other applications and the analysis of other data. Examples of the implementation of fuzzy data processing on the client side include: risk assessment based on human factors [2], database flexible querying [3], modeling and control [6], historical monuments searching [7], damage assessment [8], decision support systems [11], searching candidates for a date, human profile analysis for missing and unidentified people, automatic news generation for stock exchange [13], decision making in business [19], and others.



**Fig. 1.** Client-server architecture with object-relational mapping layer and location of fuzzy extensions proposed in the paper within the architecture.

On the other hand, procedures and functions that allow fuzzy processing of data can be implemented on the server side. This involves implementation of the procedures and functions in the programming language native for the particular database management system (DBMS). Examples of such implementations are: SQLf [5], FQUERY [10], Soft-SQL [4], fuzzy Generalised Logical Condition [9], FuzzyQ [13], fuzzy SQL extensions for relational databases [12, 14, 18], possibilistic databases [17], and for data warehouses [1, 15]. Such an approach usually delivers universal routines that can be used for fuzzy processing of various data coming from different domains. This versatility is a great asset, but it binds users and software developers to particular database management system and its native language, which may also have some limitations.

**2.2 Problem Formulation and Scope of the Work**

In both mentioned approaches, the prevalent problem is mapping between classes of the client software application and database tables, which is necessary for applications that manipulate and persist data. In the past, each application

created its own mapping layer in which the client application's specific classes were mapped to specific tables in the relational database using dedicated SQL statements. Object-relational mapping (ORM) brought evolution in software development by delivering programming technique that allows for automatic conversions of data between relational databases and object-oriented programming languages. ORM was introduced to programming practice in recent decade in response to the incompatibility between relational model of database systems and object-oriented model of client applications. This incompatibility, which is often referred to as the object-relational impedance mismatch, covers various difficulties while mapping objects or class definitions in the developed software application to database tables defined by relational schema. Object-relational mapping tools mitigate the problem of OR impedance mismatch and simplify building software applications that access data in relational database management systems (RDBMSs). Figure 1 shows the role and place of the ORM tools in a typical client-server architecture. However, ORM tools do not provide solutions for people involved in the development of applications that make use of fuzzy data processing techniques.

In the paper, we show extensions to the Doctrine object-relational mapping tool that allow fuzzy processing of crisp data stored in relational database. Doctrine ORM framework is one of several PHP libraries developed as a part of the Doctrine Project, which is primarily focused on database storage and object mapping. Doctrine has greatly benefited from concepts of the Hibernate ORM and has adapted these concepts to fit the PHP language. One of Doctrine's key features is the option to write database queries in Doctrine Query Language (DQL), an object-oriented dialect of SQL, which we extended with the capability of fuzzy data processing.

### 3 Extensions to Relational Algebra

For fuzzy exploration of crisp data stored in a relational database we have extended a collection of standard operations of the relational algebra by *fuzzy selection* operation.

Given a relation  $R$  with  $n$  attributes:

$$R = \{A_1 A_2 A_3 \dots A_n\}, \quad (1)$$

a *fuzzy selection*  $\tilde{\sigma}$  is a unary operation that denotes a subset  $\tilde{\sigma}$  of a relation  $R$  on the basis of fuzzy search condition:

$$\tilde{\sigma}_{A_i \overset{\lambda}{\approx} v}(R) = \{t : t \in R, t(A_i) \overset{\lambda}{\approx} v\}, \quad (2)$$

where:  $A_i \overset{\lambda}{\approx} v$  is a fuzzy search condition,  $A_i$  is one of attributes of the relation  $R$  for  $i = 1..n$ ,  $n$  is the number of attributes of the relation  $R$ ,  $v$  is a fuzzy set (e.g., *young* person, *tall* man, *normal* blood pressure, *age near 30*),  $\approx$  is a comparison operator used to compare crisp value of attribute  $A_i$  for each tuple

$t$  from database with fuzzy set  $v$ ,  $\lambda$  is a minimum membership degree for which the search condition is satisfied.

The selection  $\tilde{\sigma}_{A_i \approx_v^\lambda}(R)$  denotes all tuples in  $R$  for which  $\approx$  holds between the attribute  $A_i$  and the fuzzy set  $v$  with the membership degree greater or equal to  $\lambda$ . Therefore,

$$\tilde{\sigma}_{A_i \approx_v^\lambda}(R) = \{t : t \in R, \mu_v(t(A_i)) \geq \lambda\}, \tag{3}$$

where  $\mu_v$  is a membership function of a fuzzy set  $v$ .

The fuzzy set  $v$  can be defined by various types of membership functions, including:

- triangular

$$\mu_v(t(A_i); l, m, n) = \begin{cases} 0, & \text{if } t(A_i) \leq l \\ \frac{t(A_i)-l}{m-l}, & \text{if } l < t(A_i) \leq m \\ \frac{n-t(A_i)}{n-m}, & \text{if } m < t(A_i) \leq n \\ 0, & \text{if } t(A_i) > n \end{cases}, \tag{4}$$

where  $m$  is the core of the fuzzy set  $v$ ,  $[l, m]$  determines the left spread (boundary) of the fuzzy set, and  $[m, n]$  determines the right spread (boundary) of the fuzzy set.

- trapezoidal

$$\mu_v(t(A_i); l, m, n, p) = \begin{cases} 0, & \text{if } t(A_i) \leq l \\ \frac{t(A_i)-l}{m-l}, & \text{if } l \leq t(A_i) \leq m \\ 1, & \text{if } m \leq t(A_i) \leq n \\ \frac{p-t(A_i)}{p-n}, & \text{if } n \leq t(A_i) \leq p \\ 0, & \text{if } t(A_i) > p \end{cases}, \tag{5}$$

where  $[m, n]$  is the core of the fuzzy set  $v$ ,  $[l, m]$  determines the left spread (boundary) of the fuzzy set, and  $[n, p]$  determines the right spread (boundary) of the fuzzy set.

- Gaussian

$$\mu_v(t(A_i); c, s, m) = e^{-\left(\frac{t(A_i)-c}{2s}\right)^m}, \tag{6}$$

where  $c$  is called a centre,  $s$  is a width, and  $m$  is a fuzzification factor (e.g.,  $m = 2$ ).

Figure 2 shows how filtering (selection) with fuzzy search conditions works for sample data stored in tables *Measurement* and *Measure* of a relational database (Tables 1 and 2).

Fuzzy selection can be performed on the basis of multiple fuzzy search conditions (or mixed with crisp search conditions), e.g.:

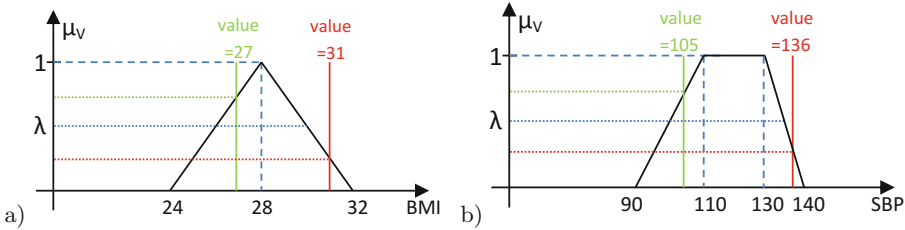
$$\tilde{\sigma}_{A_i \approx_{v_i}^{\lambda_i} \Theta \dots \Theta A_j \approx_{v_j}^{\lambda_j}}(R) = \{t : t \in R, t(A_i) \approx_{v_i}^{\lambda_i} \Theta \dots \Theta t(A_j) \approx_{v_j}^{\lambda_j}\}, \tag{7}$$

**Table 1.** Simplified structure of the *Measure* table

ID	Name
1	BMI
2	Systolic blood pressure
3	Diastolic blood pressure
4	PLT

**Table 2.** Simplified structure of the *Measure* table

MEASURE_ID	VALUE	USER_ID
1	27	1
2	105	1
1	31	2
2	136	2



**Fig. 2.** Filtering crisp data (selection) with fuzzy search conditions for fuzzy sets: (a) body mass index (BMI) around 28 (b) normal systolic blood pressure (SBM is normal, In both cases  $\lambda = 0.5$ ).

where:  $A_i, A_j$  are attributes of relation  $R$ ,  $i, j = 1 \dots n, i \neq j$ ,  $v_i, v_j$  are fuzzy sets,  $\lambda_i, \lambda_j$  are minimum membership degrees for particular fuzzy search conditions, and  $\Theta$  can be any of logical operators of conjunction or disjunction  $\Theta = \{\wedge, \vee\}$ . Therefore:

$$\tilde{\sigma}_{A_i \approx_{\lambda_i} v_i \Theta \dots \Theta A_j \approx_{\lambda_j} v_j} (R) = \{t : t \in R, \mu_{v_i}(t(A_i)) \geq \lambda_i \Theta \dots \Theta \mu_{v_j}(t(A_j)) \geq \lambda_j\}, \tag{8}$$

where:  $\mu_{v_i}, \mu_{v_j}$  are membership functions of fuzzy sets  $v_i, v_j$ .

## 4 Extensions to Doctrine ORM

We have extended the Doctrine ORM library with a *Fuzzy* module that enables fuzzy processing of crisp data stored in a relational database. In this section, we describe the most important classes extending standard functionality of the

Doctrine ORM library with the possibility of fuzzy data processing. Additionally, we present a sample usage of the classes in a real application, while performing a simple fuzzy analysis of ambulatory data.

#### 4.1 Class Model of the *Fuzzy* Module

Extensions to Doctrine object-relational mapping tool are gathered in a dedicated programming module, called *Fuzzy*. The module is available at <https://gitlab.com/aurorae/fuzzy>. The *Fuzzy* module is universal, i.e., independent of the domain of developed application and data stored in a database. Software developers can use the implemented functionality for fuzzy processing of any values stored in the database, e.g., atmospheric pressure, body temperature, person's age, or the number of hours spent watching television. It is necessary to select the type of membership function defining a fuzzy set and provide the relevant parameters.

The *Fuzzy* module provides implementations of functions extending Doctrine Query Language (DQL), which is query language of the Doctrine ORM library. Classes delivered by the *Fuzzy* module are presented in Fig. 3. They are marked in green, in contrast to native PHP classes and classes of the Doctrine ORM/DBAL library that are marked in white. In order to enable fuzzy data processing in the ORM layer we had to implement a set of classes and methods that lead to proper generation of SQL queries for particular functions of fuzzy data processing. To this purpose, we have extended the `Doctrine\ORM\Query\AST\Functions\FunctionNode` class provided by the Doctrine ORM library (Fig. 3). Classes that inherit from the *FunctionNode* class are divided into two groups placed in separate namespaces: membership functions (e.g., *InRange*, *Near*, *RangeUp*) and general-purpose functions (e.g., *Floor*, *Date*). They all implement two important methods: *parse* and *getSql*. The *parse* method detects function parameters in the DQL expression and stores them for future use, then the *getSql* method generates an expression representing a particular mathematical function in the native SQL for a database.

The *InRange* class represents classical (*LR*) trapezoidal membership function and is suitable to describe values of a domain, e.g., a fuzzy set of *normal* blood pressure, but also *near optimal* LDL Cholesterol (which are in certain ranges of values). The *RangeUp* and *RangeDown* classes represent special cases of trapezoidal membership functions - L-functions (with parameters  $n = p = +\infty$ ) and R-functions (with parameters  $l = m = -\infty$ ), respectively. They are both defined automatically with respect to the fuzzy sets of chosen values of a domain and are suitable to represent selection conditions, such as *HDL below the norm* or *slow heart rate* (R-functions) and *LDL above the norm* or *high blood pressure* (L-functions). The *Near* class represents triangular membership functions and is suitable, e.g., in formulating fuzzy search conditions, like *age about 35*. The *NearGaussian* class represents Gaussian membership function and has similar purpose to triangular membership function.

The *Fuzzy* module also provides a function factory (*FuzzyFunctionFactory* class), which creates instances of classes for the selected membership functions,



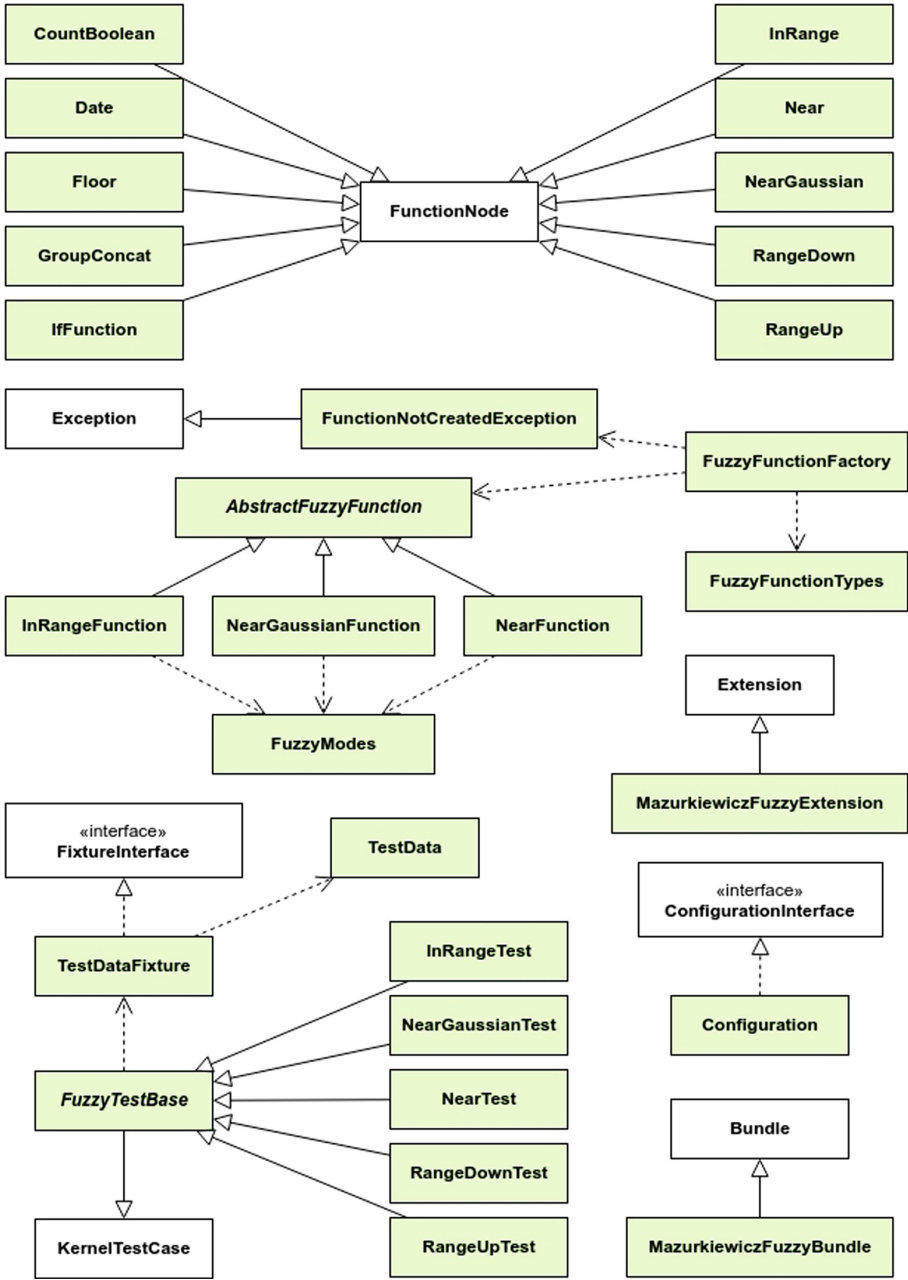


Fig. 3. Overview of classes provided by the *Fuzzy* module extending the Doctrine ORM library.

based on the specified type of the function (one of the values of *FuzzyFunctionTypes*), e.g., instance of the *NearFunction* class for the *Near* characteristic function. The function factory class generates appropriate DQL expression together with fuzzy selection condition (as formally defined in Sect. 3) on the basis of declared query type (accepted types are constants of the class *FuzzyModes*). The *Fuzzy* module also contains classes for various tests, e.g., for testing SQL statements generated by the extended ORM library for particular membership functions (e.g., *InRangeTest*, *NearTest* that inherit from *FuzzyTestBase* class).

## 4.2 Sample Usage of the Doctrine ORM Library with Fuzzy Extensions

In this section, we present a sample usage of the Doctrine ORM library with developed *Fuzzy* module in the analysis of ambulatory data. We show how the fuzzy extensions are utilized in the PHP code of our software application that allows reporting on measurements stored in MySQL relational data repository by calling appropriate DQL queries of the Doctrine ORM library. Finally, we present the form of the SQL query executed in the relational database that corresponds to the DQL query.

Presented sample of the code refers to relational table *measurement* containing ambulatory measurements in the *value* attribute for particular measures identified by *measure\_id* attribute (like in Table 2 in Sect. 3). In the ORM layer, this table is mapped to a class called *Measurement*, which attributes correspond to fields (columns) of the *measurement* table. Fuzzy search conditions, created by means of appropriate classes of the *Fuzzy* module, will be imposed on the *value* attribute - Fig. 4, Sect. 2 - for selected measures of systolic blood pressure and diastolic blood pressure (1).

Part of the PHP code was skipped for the sake of clarity of the presentation. In the presented example we assume that domains of both measures are divided into three fuzzy sets: *normal*, *low*, and *high* blood pressure, according to applicable standards for systolic and diastolic blood pressure. The starting point in this case is to define fuzzy sets for *normal* systolic and diastolic blood pressure with respect to which we define *low* and *high* fuzzy sets for both measures. To represent *normal* blood pressure we use trapezoidal membership functions (specified by *IN\_RANGE* function type in the *Fuzzy* module) with 90, 110, 130, 135 parameters for systolic blood pressure (3) and with 50, 65, 80, 90 parameters for diastolic blood pressure (4). We define both membership functions by invocation of the *create* function of the *FuzzyFunctionFactory* class. We are interested in selecting patients, whose blood pressure (both types) is elevated (*high*), i.e., above the *normal* value, with the minimum membership degree equal to 0.5 (8). Therefore, we have to define fuzzy search conditions by using *getDql* method of the *InRangeFunction* class instance returned by the *FuzzyFunctionFactory*. Then, we have to use *ABOVE\_SET* fuzzy mode in the *getDql* method in order to get values above the *normal*. In such a way, we obtain two fuzzy search conditions for DQL query that will be used in the *where* clause. To build the whole analytical report we formulate a query by using Doctrine Query Builder with

```

/** @var MeasurementRepository $repository */
$repository = ...;
$sysMeasureId = ...; // (1)
$diaMeasureId = ...; // (1)

$valueColumnName = 'mm.value'; // (2)

// (3)
$sysFunction = FuzzyFunctionFactory::create(
    FuzzyFunctionTypes::IN_RANGE,
    [90, 110, 130, 135]
);
// (8)
$sysDqlCondition = $sysFunction->getDql(
    FuzzyModes::ABOVE_SET,
    $valueColumnName,
    0.5
);

// (4)
$diaFunction = FuzzyFunctionFactory::create(
    FuzzyFunctionTypes::IN_RANGE,
    [50, 65, 80, 90]
);
// (8)
$diaDqlCondition = $diaFunction->getDql(
    FuzzyModes::ABOVE_SET,
    $valueColumnName,
    0.5
);

// (9)
$query = $repository->createQueryBuilder('mm')
    ->select('u.id, m.name, mm.value')
    ->join('mm.user', 'u') // (5)
    ->join('mm.measure', 'm') // (6)
    ->where("(mm.measure = {$sysMeasureId} AND {$sysDqlCondition})
        OR (mm.measure = {$diaMeasureId} AND {$diaDqlCondition})") // (7)
    ->getQuery();

$result = $query->getResult();

```

**Fig. 4.** Sample usage of the *Fuzzy* module in PHP code.

appropriate clauses of the query statement (9). We join *User* (5) and *Measure* (6) entities/classes to add data about patients and measure types. Finally, we add fuzzy search conditions in (7). The query will return only those rows for

which values of both measures belong to *high* fuzzy set (i.e., *above the norm*) defined for the particular measure.

The PHP code presented in Fig. 4 produces the DQL query shown in Fig. 5, which will be translated to SQL query for relational database (Fig. 6) by the Doctrine library. Translation of the query built up with the PHP Query Builder (Sect. (9) in Fig. 4) to DQL query produces WHERE clause containing two invocations of the RANGE\_UP functions with appropriate parameters of L-type membership functions representing *above the norm* fuzzy sets for particular measures.

```
SELECT u.id, m.name, mm.value
FROM Mazurkiewicz\TrackerBundle\Entity\Measurement mm
INNER JOIN mm.user u
INNER JOIN mm.measure m
WHERE (mm.measure = 2 AND RANGE_UP(mm.value, 130, 135) >= 0.5)
      OR (mm.measure = 3 AND RANGE_UP(mm.value, 80, 90) >= 0.5)
```

**Fig. 5.** DQL query with two fuzzy search conditions for PHP code presented in Fig. 4

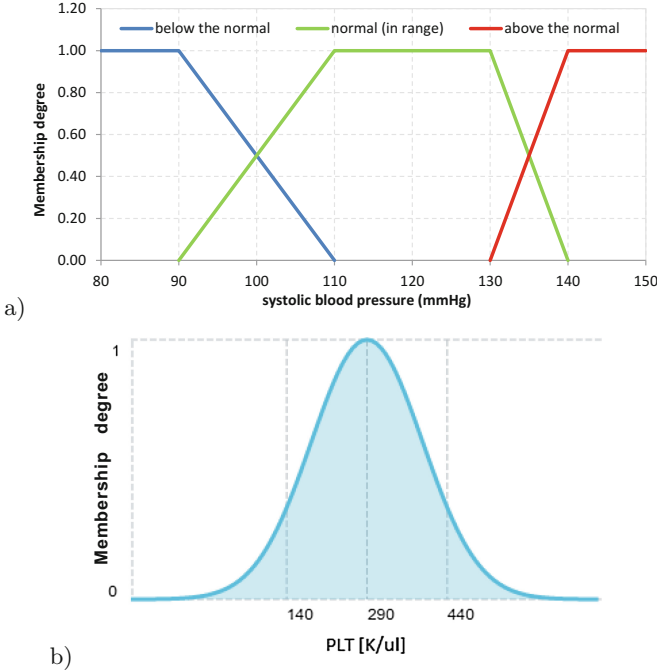
These invocations are then translated to the CASE ... WHEN ... THEN statements in the WHERE clause of the SQL query command (Fig. 6).

```
SELECT u0_.id AS id_0, m1_.name AS name_1, m2_.value AS value_2
FROM measurement m2_
INNER JOIN user u0_ ON m2_.user_id = u0_.id
INNER JOIN measure m1_ ON m2_.measure_id = m1_.id
WHERE
  (m2_.measure_id = 2 AND CASE
    WHEN m2_.value <= 130 THEN 0
    WHEN m2_.value <= 135 THEN (m2_.value-130)/(135-130)
    ELSE 1 END >= 0.5
  )
OR (m2_.measure_id = 3 AND CASE
  WHEN m2_.value <= 80 THEN 0
  WHEN m2_.value <= 90 THEN (m2_.value-80)/(90-80)
  ELSE 1 END >= 0.5
)
```

**Fig. 6.** SQL query translated by fuzzy extension of the Doctrine library from DQL query presented in Fig. 5

## 5 Experimental Results

We tested performance of the fuzzy extension for the Doctrine ORM library in several series of tests. We were primarily interested in verification of how the



**Fig. 7.** Membership functions for *systolic blood pressure* (a). Membership function for the fuzzy set *normal PLT* (b).

necessity of calculation of value of a membership function influences the execution time of particular fuzzy queries with respect to classical queries that operate on given ranges of values (appropriately chosen intervals). For this purpose, we used a database containing 2,500,000 records in the *Measurement* table coming from laboratory tests.

Results of performance tests are presented in Table 3 for three chosen sample fuzzy queries (Q1–Q3) retrieving measurement data for patients having:

- Q1 - *normal* systolic blood pressure (Fig. 7a),
- Q2 - systolic blood pressure *above the normal* (Fig. 7a),
- Q3 - *normal* platelet count (PLT) (Fig. 7b),

with a minimum membership degree  $\lambda = 0.5$ . In a real implementation, we tested many more queries, but they all shown the same execution time tendency.

Queries Q1–Q3 contain fuzzy search conditions. Definitions of fuzzy sets used in these search conditions are presented in Fig. 7. Particular parameters of the membership functions were set on the basis of arbitrary expert’s knowledge, and include some tolerance. These parameters can be changed in specific implementations, which leads to different results. Therefore, they must be assumed carefully, while consulting the shape of membership functions with domain experts.

Additionally, for queries Q1–Q3 we created their classical counterparts with precise search criteria based on intervals, where left and right boundaries of the intervals were calculated for the membership degree  $\lambda = 0.5$ . Particular fuzzy queries and their precise counterparts returned the same sets of results, but were parametrized in a different way - precise queries need exact values of left and right boundaries of intervals, while fuzzy queries need only the minimum membership degree  $\lambda$ , above which the search condition is satisfied.

**Table 3.** Results of performance tests for fuzzy queries Q1–Q3 and their precise counterparts for the minimum membership degree  $\lambda = 0.5$ .

Query	Average execution time (s)		Difference (s)	Relative difference (%)
	Precise query	Fuzzy query		
Q1	0.491274	0.496286	0.005012	1.02
Q2	0.485716	0.497575	0.011859	2.44
Q3	0.610142	0.627314	0.017172	2.81

Results of performance tests presented in Table 3 proved that execution times of fuzzy queries were only slightly worse than execution times of precise queries that returned the same sets of results. Fuzzy queries were executed relatively 1–3% longer than their precise counterparts. This means that for users of the ORM library with fuzzy extensions the difference in execution time is almost imperceptible.

## 6 Discussion and Concluding Remarks

Our research on extending the Doctrine object-relational mapping framework toward fuzzy data processing show that it is possible to incorporate fuzzy logic in the ORM layer and enhance standard database querying with new capabilities of imprecise, proximity-based or similarity-based searching. The enhancement brings new power to the analysis of crisp, numerical data stored in databases, which is important when processing large volumes of biomedical or ambulatory data, and can be now performed in the ORM layer, which is important for software developers. As proved by our experiments, performance costs of such an enhancement are negligible compared to the additional analytic possibilities that are obtained by developers of database applications.

Fuzzy querying with fuzzy search conditions provides several benefits compared to precise queries. Synthetic comparison of precise and fuzzy queries in terms of flexibility of queries and corresponding requirements is presented in Table 4. First of all, fuzzy queries give the possibility to easily filter out uninteresting data on the basis of soft search conditions, while still keeping similar data in the final result set. Therefore, they narrow the result set to similar cases, which is very important while performing large-scale medical screenings based on the

**Table 4.** Comparison of fuzzy and precise queries in terms of flexibility, requirements, and performance.

Fuzzy queries	Precise queries
+ Soft filtering - including similar cases	– Hard filtering - similar data filtered out
+ Require value of the minimum membership degree $\lambda$ in a fuzzy search condition	– Require crisp values in search conditions, e.g., left and right boundaries of an interval
$\pm$ Require expert that arbitrary defines fuzzy sets and corresponding membership functions	– Require users to have knowledge of data domain
– Require additional calculations of membership degrees, which may affect performance	+ No additional calculations are required

+ advantage, – disadvantage

analysis of various types of biomedical data, including ambulatory data. Hard filtering with precise queries may cause that important cases leading to the same medical conclusions and therapeutic recommendations will be just skipped, as they not satisfy precise search conditions. Secondly, precise queries require specification of crisp values for their filtering conditions. These crisp values may not be known for the final users or may require further investigations to know them, especially in medical domain. On the other hand, fuzzy queries require specifying the minimum membership degrees  $\lambda$  for fuzzy search conditions, which may also be a problem, but we must remember that they decide about the similarity degree for data that is returned in the result set. Therefore, they can be chosen in several trials narrowing the final result set in several following steps. Consequently, precise queries require that users of the developed system have a specialized knowledge of the domain of analyzed data, which is sometimes very difficult to gain unless they are experts. For example, when analyzing results of laboratory tests users have to find out what are the normal ranges for specific ambulatory tests, if they are not doctors or laboratory staff, which is a weakness. This can be also a weakness of fuzzy queries, since for many domains the requirement for dividing the analyzed domain into proper ranges and defining proper membership functions for identified fuzzy sets is prevalent and can be a spark for discussion. However, for ambulatory data and many other types of biomedical data these values are usually arbitrary defined by experts and are indisputable for, at least, some period of time. Therefore, for such domains this does not constitute a problem and causes the use of fuzzy search conditions with their large flexibility a more natural solution. Finally, a weak point of fuzzy queries is the necessity to calculate membership degrees for each tuple from the database processed by the fuzzy query, which may negatively affect performance of the query. However, the fuzzy extensions to the ORM layer that we have developed

proved to be only 1–3% slower for tested queries, which allows us to ignore this slight decrease in performance in the face of much better querying capabilities.

Our fuzzy extensions to the Doctrine library mitigate the problem of object-relational impedance mismatch for those software developers that want to perform fuzzy searches while working in the object-oriented model, regardless of the data domain being analyzed. It is limited to the PHP technology of building client software tools, but universal in terms of the type of analyzed data and built application. The *Fuzzy* module for Doctrine framework presented in the paper enables re-usability of procedures for fuzzy data processing for any client application that is developed and any data that is analyzed, which was a limitation of client-based solutions mentioned in Sect. 2.1. On the other hand, software developers are not bound to a particular database management system and its native query language, which was a weakness of server-side solutions presented in Sect. 2.1. This ensures broader portability of our fuzzy extension. In such a way, our solution complements a collection of existing solutions and, to the best of our knowledge, is first such an extension for the ORM layer.

## References

1. Appelgren Lara, G., Delgado, M., Marín, N.: Fuzzy multidimensional modelling for flexible querying of learning object repositories. In: Larsen, H.L., Martin-Bautista, M.J., Vila, M.A., Andreassen, T., Christiansen, H. (eds.) FQAS 2013. LNCS (LNAI), vol. 8132, pp. 112–123. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-40769-7\\_10](https://doi.org/10.1007/978-3-642-40769-7_10)
2. Aras, F., Karaka, Y.: Fuzzy logic-based user interface design for risk assessment considering human factor: a case study for high-voltage cell. *Saf. Sci.* **70**, 387–396 (2014). <http://www.sciencedirect.com/science/article/pii/S0925753514001726>
3. Ben Hassine, M.A., Ounelli, H.: IDFQ: an interface for database flexible querying. In: Atzeni, P., Caplinskas, A., Jaakkola, H. (eds.) ADBIS 2008. LNCS, vol. 5207, pp. 112–126. Springer, Heidelberg (2008). doi:[10.1007/978-3-540-85713-6\\_9](https://doi.org/10.1007/978-3-540-85713-6_9)
4. Bordogna, G., Psaila, G.: Customizable flexible querying in classical relational databases. In: Handbook of Research on Fuzzy Information Processing in Databases, pp. 191–217 (2008)
5. Bosc, P., Pivert, O.: SQLf query functionality on top of a regular relational database management system. In: Pons, O., Vila, A.M., Kacprzyk, J. (eds.) Knowledge Management in Fuzzy Databases, vol. 39, pp. 171–190. Physica-Verlag HD, Heidelberg (2000). doi:[10.1007/978-3-7908-1865-9\\_11](https://doi.org/10.1007/978-3-7908-1865-9_11)
6. Cheng, S., Dong, R., Pedrycz, W.: A framework of fuzzy hybrid systems for modelling and control. *Int. J. Gen Syst* **39**(2), 165–176 (2010). <http://dx.doi.org/10.1080/03081070903427358>
7. Czajkowski, K., Olczyk, P.: Fuzzy interface for historical monuments databases. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małyśiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2014. CCIS, vol. 424, pp. 271–279. Springer, Cham (2014). doi:[10.1007/978-3-319-06932-6\\_26](https://doi.org/10.1007/978-3-319-06932-6_26)
8. Furuta, H., Shiraishi, N.: Fuzzy data processing in damage assessment. In: Natke, H.G., Yao, J.T.P. (eds.) Structural Safety Evaluation Based on System Identification Approaches, pp. 381–392. Vieweg+Teubner Verlag, Wiesbaden (1988). doi:[10.1007/978-3-663-05657-7\\_18](https://doi.org/10.1007/978-3-663-05657-7_18)



9. Hudec, M.: An approach to fuzzy database querying, analysis and realisation. *Comput. Sci. Inf. Syst.* **12**, 127–140 (2009)
10. Kacprzyk, J., Zadrozny, S.: Data mining via fuzzy querying over the internet. In: Pons, O., Vila, A.M., Kacprzyk, J. (eds.) *Knowledge Management in Fuzzy Databases*, vol. 39, pp. 211–233. Physica-Verlag HD, Heidelberg (2000). doi:[10.1007/978-3-7908-1865-9\\_13](https://doi.org/10.1007/978-3-7908-1865-9_13)
11. Macwan, N., Sajja, P.S.: Fuzzy logic: an effective user interface tool for decision support system. *Int. J. Eng. Sci. Innov. Technol.* **3**(3), 278–283 (2014). <http://www.ijesit.com/Volume%203/Issue%203/IJESIT201403.35.pdf>
12. Małysiak, B., Mrozek, D., Kozielski, S.: Processing fuzzy SQL queries with flat, context-dependent and multidimensional membership functions. In: *IATED International Conference on Computational Intelligence*, Calgary, Alberta, Canada, 4–6 July 2005, pp. 36–41 (2005)
13. Małysiak-Mrozek, B., Kozielski, S., Mrozek, D.: Modern software tools for researching and teaching fuzzy logic incorporated into database systems. In: *Proceedings of the iNEER International Conference on Engineering Education*, Gliwice, Poland, pp. 1–8. iNEER, July 2010. [http://www.ineer.org/Events/ICEE2010/papers/T11D/Paper\\_954\\_1141.pdf](http://www.ineer.org/Events/ICEE2010/papers/T11D/Paper_954_1141.pdf)
14. Małysiak-Mrozek, B., Mrozek, D., Kozielski, S.: Data grouping process in extended SQL language containing fuzzy elements. In: Cyran, K.A., Kozielski, S., Peters, J.F., Stańczyk, U., Wakulicz-Deja, A. (eds.) *Man-Machine Interactions*, vol. 59, pp. 247–256. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-00563-3\\_25](https://doi.org/10.1007/978-3-642-00563-3_25)
15. Małysiak-Mrozek, B., Mrozek, D., Kozielski, S.: Processing of crisp and fuzzy measures in the fuzzy data warehouse for global natural resources. In: García-Pedrajas, N., Herrera, F., Fyfe, C., Benítez, J.M., Ali, M. (eds.) *IEA/AIE 2010. LNCS (LNAI)*, vol. 6098, pp. 616–625. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-13033-5\\_63](https://doi.org/10.1007/978-3-642-13033-5_63)
16. Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kozielski, S.: Life sciences data analysis. *Inf. Sci.* **384**, 86–89 (2017)
17. Myszkorowski, K.: Inference rules for fuzzy functional dependencies in possibilistic databases. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) *BDAS 2015-2016. CCIS*, vol. 613, pp. 181–191. Springer, Cham (2016). doi:[10.1007/978-3-319-34099-9\\_13](https://doi.org/10.1007/978-3-319-34099-9_13)
18. Portinale, L., Montani, S.: A fuzzy logic approach to case matching and retrieval suitable to SQL implementation. In: *Proceedings of the 2008 20th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2008*, vol. 02, pp. 241–245. IEEE Computer Society, Washington, DC (2008). <http://dx.doi.org/10.1109/ICTAI.2008.88>
19. Ribeiro, R.A., Moreira, A.M.: Fuzzy query interface for a business database. *Int. J. Hum.-Comput. Stud.* **58**(4), 363–391 (2003)
20. Zadeh, L.: Fuzzy sets. *Inf. Control* **8**, 338–353 (1965)
21. Zadeh, L.: Fuzzy logic. *Computer* **21**(4), 83–93 (1988)