International Journal of Approximate Reasoning 170 (2024) 109209

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

journal homepage: www.elsevier.com/locate/ijar

Tuning fuzzy SPARQL queries ☆

Jesús M. Almendros-Jiménez [a,*], Antonio Becerra-Terón [a], Ginés Moreno [b], José A. Riaza [b]

[a] Dept. of Informatics, University of Almería, Almería, 04120, Spain
[b] Dept. of Computing Systems, University of Castilla-La Mancha, Albacete, 02006, Spain

ARTICLE INFO

Keywords:
SPARQL
Debugging
Semantic Web
Fuzzy logic

ABSTRACT

During the last years, the study of fuzzy database query languages has attracted the attention of many researchers. In this line of research, our group has proposed and developed FSA-SPARQL (*Fuzzy Sets and Aggregators based SPARQL*), which is a fuzzy extension of the Semantic Web query language SPARQL. FSA-SPARQL works with fuzzy RDF datasets and allows the definition of fuzzy queries involving fuzzy conditions through fuzzy connectives and aggregators. However, there are two main challenges to be solved for the practical applicability of FSA-SPARQL. The first problem is the lack of fuzzy RDF data sources. The second is how to customize fuzzy queries on fuzzy RDF data sources. Our research group has also recently proposed a fuzzy logic programming language called $\mathcal{FASILL}$ that offers powerful tuning capabilities that can accept applications in many fields. The purpose of this paper is to show how the $\mathcal{FASILL}$ tuning capabilities serve to accomplish in a unified framework both challenges in FSA-SPARQL: data fuzzification and query customization. More concretely, from a FSA-SPARQL to $\mathcal{FASILL}$ transformation, data fuzzification and query customization in FSA-SPARQL become $\mathcal{FASILL}$ tuning problems. We have validated the approach with queries against datasets from online communities.

1. Introduction

Semantic Web data are usually structured in a triple format, that is, (subject, predicate, object), called Resource Description Framework (RDF), defining an RDF graph. SPARQL[1] [30], the W3C standard for RDF query languages, has gained significant popularity in recent years, and an increasing amount of effort is focused on improving the functionality and usability of SPARQL-based search engines. Recent studies focus on solving well-known query problems for users of SPARQL: a certain query returns too many answers, the user expects a particular answer which is not included –in particular, the set of answers is empty–, or a combination of these problems.

A feasible solution to the above problems, mainly for missing answers, is to adopt fuzzy semantics for SPARQL. It is well known that fuzzy logic can be used to relax restrictions in query systems. Typically, replacing a Boolean condition by a soft condition

☆ This work was supported by the State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant PID2019-104735RB-C42 (SAFER).
* Corresponding author.
E-mail addresses: jalmen@ual.es (J.M. Almendros-Jiménez), abecerra@ual.es (A. Becerra-Terón), Gines.Moreno@uclm.es (G. Moreno), joseantonio.riaza@uclm.es (J.A. Riaza).
[1] https://www.w3.org/TR/sparql11-query/.

https://doi.org/10.1016/j.ijar.2024.109209
Received 25 September 2023; Received in revised form 24 April 2024; Accepted 24 April 2024

Available online 30 April 2024

$\&_{\texttt{prod}}(x,y) \triangleq x * y$	$\|_{\texttt{prod}}(x,y) \triangleq x + y - xy$	*Product*
$\&_{\texttt{godel}}(x,y) \triangleq \min(x,y)$	$\|_{\texttt{godel}}(x,y) \triangleq max(x,y)$	*Gödel*
$\&_{\texttt{luka}}(x,y) \triangleq \max(0, x+y-1)$	$\|_{\texttt{luka}}(x,y) \triangleq \min(x+y, 1)$	*Łukasiewicz*

Fig. 1. Conjunctions and disjunctions of three different fuzzy logics over $([0,1], \leq)$.

enables one to capture finer solutions for a certain request. Such soft conditions can be defined, for instance, in terms of fuzzy sets. For example, a Boolean condition about the height of a person can be relaxed in terms of the *Tall* fuzzy set in such a way that instead of requiring a height above 1.90 meters, one can request a tall person, which is a fuzzy concept. According to fuzzy logic principles, every person is tall to a certain degree, typically a real number of the interval $[0,1]$, which is called the degree of membership to the *Tall* fuzzy set. People with a height around 1.90 would have a membership degree close to one, but people of around 1.70 would also be tall with a lower membership degree. Such membership degrees to fuzzy sets can be explicitly defined (by enumerating all possible cases), or alternatively, they can be defined in terms of a fuzzy membership function.

Fuzzy logic typically also replaces Boolean logic combinators (also called *operators* or *connectives*) by fuzzy versions that act on truth degrees instead of true and false values. A wide range of connectives have traditionally been proposed to manipulate truth degrees or, equivalently, to manipulate membership degrees to fuzzy sets. Fig. 1 shows the well-known *Product*, *Łukasiewicz* and *Gödel* fuzzy conjunction/disjunction connectives which replace *and/or* Boolean logic operators by their respective binary fuzzy versions. Other fuzzy operators are used to modulate a truth degree, as the $@_{\texttt{very}}(x) \triangleq x^2$ or $@_{\texttt{more\_or\_less}}(x) \triangleq \sqrt{x}$ linguistic modifiers do. And others, which are generally called *aggregators* to differentiate themselves from connectives, describe other ways to combine the membership degrees of fuzzy sets. For example, $@_{\texttt{mean}}(x,y) \triangleq (x+y)/2$, and $@_{\texttt{wmean}}(w,x,y) \triangleq (w * x + (1-w) * y)$. The operator $@_{\texttt{wmean}}$ has an additional parameter (w), called the *weight*, and serves to prioritize one of the membership degrees.

However, there is still a problem to be solved. Queries involving fuzzy sets and fuzzy connectives/operators may suffer the same problems as the crisp ones: missing, scarce, and overabundant answers. This is due to two factors.

(1) On the one hand, the selection of the membership function is difficult when the domain of the function is not familiar or dynamic. Let us suppose that our goal now is to define a membership function for the fuzzy set *High Salary*. Here, to establish what is supposed to be a high salary, one should know the dataset to be classified: the country, the range of age, etc. Otherwise, people might not be uniformly distributed by the membership function. Inspection of the data set to be classified, or at least some sample of the data set, might help design the membership function.

(2) On the other hand, the choice of fuzzy operators is also not easy. Let us suppose that we are looking for good movies on the *TMDB*[2] website, in which the rates and the number of votes are provided, and also suppose that the notions of highly rated and highly voted movies are modeled as fuzzy sets, according to the previous explanations. Under a fixed threshold (i.e., lowest score), a bad choice of fuzzy operator to combine both membership degrees will discard some good movies. In fact, it could happen that none of the movies is below the threshold. It could happen that this threshold works well for Oscar winner movies, but not, in general, for comedies. One can try *Product*, *Łukasiewicz*, *Gödel* and $@_{\texttt{mean}}$ connectives, or even prioritizing rates or number of votes using the operator $@_{\texttt{wmean}}$ to give different values to priorities. This process of *customization* can be tedious and time consuming until a satisfactory result is reached. However, as in the previous case, inspection of the dataset to be ranked, or at least some sample of a good movie, might help to select adequate fuzzy operators. Thus, in general terms, in a fuzzy context, customization of queries involves both customization of fuzzy sets and fuzzy connectives/operators as well as their parameters.

1.1. Contributions

In the research line of improving flexible SPARQL-based search engines, this paper contributes by studying a *tuning technique* of fuzzy SPARQL queries whose goal is to debug fuzzy queries by customizing queries to the desired answers by selecting appropriate fuzzy sets, as well as fuzzy connectives/operators and their parameters.

Basically, the fuzzy set tuning process –which is called *fuzzification*– takes a sample, in which a membership degree is assigned to each element in the sample and guesses a *trapezoidal membership function* fitting the sample.

For tuning fuzzy queries –*query customization*–, our technique proceeds in a similar way. A sample of answers–together with their scores– is provided to which to accommodate the query, guessing *fuzzy connectives/operators and their parameters*.

In both cases, the tuning processes try to *optimize* the distance to the sample, returning a *deviation*, which quantifies how the tuned membership functions and the tuned queries are accomplished or are deviated from the respective sample. Moreover, in the particular case of fuzzy sets (and something similar can be said for queries), the deviation measures the consistency of the membership degrees assigned to the sample.

To adjust fuzzy SPARQL queries, we have used the tuning tool proposed by our group for the rule-based language $\mathcal{FASILL}$ [18]. $\mathcal{FASILL}$ is a fuzzy logic programming language in which, similar to PROLOG language, the programs consist of a set of facts and logic rules, together with a goal. A tuning tool for $\mathcal{FASILL}$ has been developed whose aim is to customize the facts and rules of $\mathcal{FASILL}$. The basis of the tuning process is the introduction of *symbolic constants* in facts and rules and *symbolic execution* of the so-called symbolic $\mathcal{FASILL}$ programs. Test cases are then used to find values for symbolic constants (see Moreno and Riaza [25], Moreno et al. [24] for more details). The tuning tool is, in general terms, a method that is in charge of customizing fuzzy parameters involved in software systems dealing with uncertainty.

[2] https://www.themoviedb.org/.

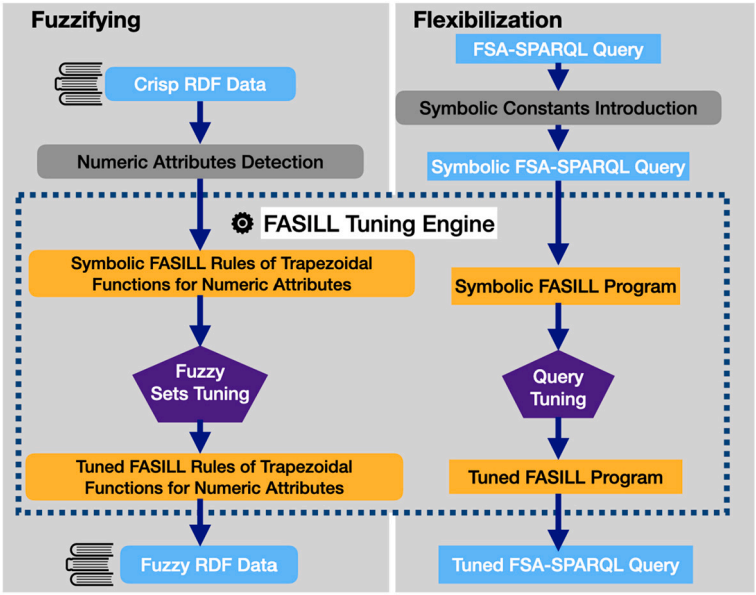

Fig. 2. Tuning Process.

Both fuzzification and query customization in our case can be seen as tuning problems (see Fig. 2), where in the first case (that is, fuzzification) the parameters of the trapezoidal membership functions, that is, *upper and lower limits* are tuned. In the second case (i.e., customization of queries), fuzzy connectives and aggregators, as well as their parameters, are tuned. $\mathcal{FASILL}$ symbolic constants are used to represent the upper and lower limits of the trapezoidal membership to be tuned, as well as the fuzzy connectives and aggregators (and parameters) to be tuned.

Furthermore, the degree of membership in the fuzzy sets and the expected response scores of the fuzzy queries become $\mathcal{FASILL}$ *test cases*, respectively. The computed substitution from the resulting symbolic $\mathcal{FASILL}$ program codes the parameters of the trapezoidal membership functions and the parameters of the fuzzy queries, respectively, to which the membership degrees to fuzzy sets and the expected answers scores are adapted. As we shall see later, our approach is based on the introduction –similarly to $\mathcal{FASILL}$ – of symbolic constants in SPARQL queries and the guessing of values for such symbolic constants. Such symbolic constants will be instantiated by the $\mathcal{FASILL}$ tuning process with fuzzy connectives/operators and their parameters. In other words, the symbolic constants represent the parts of the query to be customized, and the tuning process permits one to look for suitable values making it possible to obtain the desired answers.

We validated the method by fuzzifying datasets from online communities (TMDB and Youtube) and customizing fuzzy queries against them. A Web tool developed by our group (available at https://dectau.uclm.es/floper/tuning) has facilitated the work, allowing the loading of datasets from online communities, the selection of test cases, the visualization of the results of the tuning processes, and the execution of fuzzy queries on them.

The structure of the paper is as follows. Section 2 analyses some related work by comparing our approach with existing proposals. Section 3 presents the FSA-SPARQL language. The transformation FSA-SPARQL to $\mathcal{FASILL}$ and the $\mathcal{FASILL}$ tuning engine are introduced in Section 4. Section 5 shows the tuning process in FSA-SPARQL. Section 6 describes the experiments with datasets from online communities. Finally, future work lines are sketched in Section 7.

2. Related work

The proposed solutions for empty, missing, and overabundant answers range from relaxation of queries (for missing answers) [16,15], query rewriting [17,36,23] and debugging [28,4,37,12,39]. In Frosini et al. [16], an extension of SPARQL is proposed equipped with two operators, APPROX and RELAX, to allow flexible querying over property paths. In Fokou et al. [15], class relaxation, property relaxation, and constant-to-variable relaxation are proposed. Six types of modifier are proposed in Jian et al. [17], which are used to modify a query to obtain a new query, allowing one to insert/modify/delete a triple pattern or a filter. A similar solution is given in Song et al. [36]. Fuzzy techniques have been used in Moises and Pereira [23], using semantic proximity to implement a flexible query answering tool and allowing automatic reformulation of queries with empty or overabundant answers. In Parkin et al. [28], authors classify the various types of unsatisfactory answers and propose algorithms to compute the causes of failure. In Almendros-Jiménez and Becerra-Terón [4], constraint and ontology reasoners are used to detect ill-typed and unsatisfiable queries. There are also proposals [37] to report which part of a graph query is responsible for unexpected results. Empty answers in fuzzy knowledge databases have also been the subject of study in Dellal et al. [12], explaining the reasons for the failure and computing alternative relaxed queries. The authors of Wang et al. [39] adopt a graph-based and operator-based approach to generate logical explanations that help users redefine queries.

The fuzzy nature of FSA-SPARQL already provides more flexible queries than the standard SPARQL. But the proposed query tuning technique also facilitates to *relax* or *strengthen* the imposed (fuzzy) constraints in order to discard some answers or incorporate new ones, as well as to improve scores. From test cases (i.e. samples), the user can better specify the type of answer/score he/she expects. Our work shows how to solve two important challenges of fuzzy (query) systems in a *unified framework*.

Our work also contributes to the research line of fuzzy extensions for the Semantic Web and the design of fuzzy database query languages for this area. FSA-SPARQL is a fuzzy extension of SPARQL, which can be considered the most prominent query language on the Semantic Web. Similar fuzzy extensions of SPARQL have been proposed during the last years. This is the case for *FURQL*[3] [35,32,33], and *f-SPARQL* language (later extended to *fp-SPARQL*) [22,38,11]. To our knowledge, the other approaches do not provide tuning (or similar) techniques for automatically customizing fuzzy sets and queries. In *FURQL* the clause `DEFINE` is used to explicitly declare trapezoidal-based fuzzy sets. In our approach, instead of that, trapezoidal-based fuzzy sets are automatically obtained from test cases and the tuning process. The same can be said for *fp-SPARQL*. In Li et al. [20], the authors introduced an extension of the RDF model to represent vagueness on a fuzzy RDF graph and defined a new SPARQL graph pattern, making it possible to query a fuzzy RDF knowledge graph at the element level with a high degree of truth.

The idea of fuzzy set tuning is not really new. L.A. Zadeh already posed this question [41] about fuzzy sets, which can be formulated as follows [21]: Suppose that we are given $n+1$ points $x_0, \ldots, x_n$ in $\mathbb{R}$, and for each of these points we have a fuzzy value in $\mathbb{R}$, rather than a crisp one. Is it then possible to construct some function on $\mathbb{R}$ with range also a collection of fuzzy values that coincide, on the given $n+a$ points, with the given fuzzy values and that fulfills some natural smoothness condition? This question has found answers in classical mathematical interpolation techniques [21,19,13,31]. Some numerical approximation methods for fuzzy functions using fuzzy polynomials [1] have also been used to the same end. Other approaches perform linear regression analysis [14] or use genetic algorithms [10].

Our fuzzy sets tuning technique has the same goal, adjusting a sample (i.e. a set of test cases) to a trapezoidal membership function. However, rather than approximating the shape of a trapezoidal function, the tuning technique properly returns a trapezoidal function obtained from discrete partitions of the domain, which minimizes the distance to test cases.

Regarding query tuning, the well-known Yager *Ordered Weighted Average (OWA)* operator [40] has been widely studied and extended, and numerous weight-generating methods have appeared in the literature [42,2,9,34,3]. In our case, FSA-SPARQL handles instances of the *OWA* operator, and the proposed tuning method also uses a discrete partition of the domain to generate weights that minimize the distance from the test cases.

Let us remark that in our previous work [6], we already proposed "fuzzification mechanisms" for social network data, in such a way that the user was responsible for the (manual) definition of fuzzy sets. In Almendros-Jiménez et al. [7] we adapted for the first time our tuning technique to the FSA-SPARQL language, but focused only on tuning the query `FILTER` condition and the weight of the `WMEAN` aggregator. Although in Almendros-Jiménez et al. [7] a transformation from FSA-SPARQL to $\mathcal{FASILL}$ was defined, the $\mathcal{FASILL}$ tuner has been modified later, as Section 4.3 shows, to handle the tuning of FSA-SPARQL. The computation of the deviation of the test cases from the solution and a more efficient and optimized search of solutions including a tolerance level, among others, have been required. Last but not least important, the tuning technique of $\mathcal{FASILL}$ has now been adapted to tune the trapezoidal functions of fuzzy sets. In Almendros-Jiménez et al. [7] RDF datasets were manually fuzzified, and now the tuning technique is capable of automatically tuning fuzzy sets. Finally, in the current work, datasets from online communities have been used to validate the highly improved power of the tuning engine.

3. The language FSA-SPARQL

In this section, we will review the main elements of our query language FSA-SPARQL. For more details and examples, please see Almendros-Jiménez et al. [5] and Almendros-Jimenez et al. [6].

Basically, FSA-SPARQL is an extension of SPARQL that allows querying fuzzy RDF datasets. Our fuzzy RDF extension handles truth degrees on triples about RDF properties like *movie:OceansEleven movie:leading_role (movie:GeorgeClooney 0.7)* in which the truth degree *0.7* is associated with the RDF triple *movie: OceansEleven movie:leading_role movie:GeorgeClooney*.

Such fuzzy RDF tuples can be combined in datasets with standard RDF triples, making it possible to work with crisp and fuzzy information. The fuzzy sets in our fuzzy RDF extension are represented by RDF classes, and the membership level of an element to a fuzzy set is annotated with an RDF property and a membership degree. FSA-SPARQL extends the standard vocabulary of RDF with three elements with the namespace *f* in such a way that *f:type*, *f:subClassOf* and *f:subPropertyOf* are introduced. *f:type* represents a fuzzy concept membership relation, while *f:subClassOf* (and *f:subPropertyOf*) refers to fuzzy subconcept (and subrole) relationships.

For instance, *movie:OceansEleven f:type* (*movie:quality movie:Good 0.5*), is an RDF tuple that represents the degree of membership (i.e. *0.5*) of *Ocean's Eleven* to the *Good* fuzzy set with regard to the RDF *quality* property. In general, a given element can be a member of the same fuzzy set with different degrees of membership with regard to different properties. Fuzzy RDF tuples can be mapped to standard RDF triples following the transformations in Fig. 3. Blank nodes (*_:u*) are used to interconnect the triples obtained with the transformation. For instance, *_:u* is created for *movie:OceansEleven f:type* (*movie:quality movie:Good 0.5*) and two object properties *f:type* and *f:onProperty* and a data property called *f:truth* are used for the encoding as follows: (*movie:OceansEleven f:type _:u*), (*_:u f:onProperty movie:quality*) (*_:u rdf:type movie:Good*) and (*_:u f:truth 0.4*).

[3] https://www-shaman.irisa.fr/surf/.

(1) (s p (o t))	→ *(s p _:u) (_:u f:item o) (_:u f:truth t)*
(2) (s f:type (p o t))	→ *(s f:type _:u) (_:u f:onProperty p) (_:u rdf:type o) (_:u f:truth t)*
(3) (s f:subClassOf (o t))	→ *(s f:subClassOf _:u) (_:u rdfs:subClassOf o) (_:u f:truth t)*
(4) (s f:subPropertyOf (o t))	→ *(s f:subPropertyOf _:u) (_:u rdfs:subPropertyOf o) (_:u f:truth t)*

Fig. 3. Translation of fuzzy RDF triples into crisp RDF triples.

$$\text{AND\_PROD}(x,y) = x * y \qquad \text{OR\_PROD}(x,y) = x + y - x * y$$
$$\text{AND\_GOD}(x,y) = \min(x,y) \qquad \text{OR\_GOD}(x,y) = \max(x,y)$$
$$\text{AND\_LUK}(x,y) = \max(x+y-1,0) \qquad \text{OR\_LUK}(x,y) = \min(x+y,1)$$
$$\text{MEAN}(x,y) = \tfrac{x+y}{2} \qquad \text{WMEAN}(w,x,y) = w * x + (1-w) * y$$
$$\text{WSUM}(w,x,u,y) = w * x + u * y$$
$$\text{WMAX}(w,x,u,y) = \max(\min(w,x),\min(u,y))$$
$$\text{WMIN}(w,x,u,y) = \min(\max(1-w,x),\max(1-u,y))$$
$$\text{VERY}(x) = x^2 \qquad \text{MORE\_OR\_LESS}(x) = \sqrt{x}$$
$$\text{CLOSE\_TO}(x,l,\alpha) = \frac{1}{1+(\frac{x-l}{\alpha})^2} \qquad \text{AT\_LEAST}(x,l,\alpha) = \begin{cases} 0 & \text{if } x \le \alpha \\ \frac{x-\alpha}{l-\alpha} & \text{if } \alpha < x < l \\ 1 & \text{if } x \ge l \end{cases}$$
$$\text{AT\_MOST}(x,l,\alpha) = \begin{cases} 1 & \text{if } x \le l \\ \frac{\alpha-x}{\alpha-l} & \text{if } l < x < \alpha \\ 0 & \text{if } x \ge \alpha \end{cases}$$

Fig. 4. FSA-SPARQL Semantics: Connectives, Operators and Aggregators.

FSA-SPARQL queries are similar to SPARQL queries, but the triple patterns can be fuzzy or standard triple patterns. Additionally, the following fuzzy operators (see Fig. 4) can be used: *fuzzy connectives*[4]: inspired by the standard fuzzy logics of *Product* (`AND_PROD/OR_PROD`), *Łukasiewicz* (`AND_LUK/OR_LUK`) and *Gödel* (`AND_GOD/OR_GOD`); *linguistic modifiers*: `VERY`, `MORE_OR_LESS`; *fuzzy equalities/inequalities*: `CLOSE_TO`, `AT_MOST` and `AT_LEAST`; as well as *fuzzy aggregators*: `MEAN`, `WMEAN`, where $0 \le w \le 1$, and `WSUM`, `WMIN` and `WMAX`, where $0 \le w, 0 \le u, 0 \le w + u \le 1$.

For example, assuming a fuzzy RDF dataset of movies, where movies are described by the crisp attribute *name*, as well as by fuzzy attributes *leading_role*, *category* and *quality*, and actors described by a crisp attribute *name*, the following query requests (1) thrillers or (2) good movies in which *George Clooney* is the leading role:

```
SELECT ?Name ?Rank
WHERE {
  ?Movie movie:name ?Name .
  ?Movie movie:leading_role (?Actor ?l) .
  ?Actor movie:name "George Clooney".
  ?Movie f:type (movie:category movie:Thriller ?t) .
  ?Movie f:type (movie:quality movie:Good ?g) .
  BIND(f:OR_GOD(f:AND_LUK(?g,?l),?t) as ?Rank) .
  FILTER(?Rank >= 0.75)
}
```

Here condition (2) is expressed by the `AND_LUK` connective, which combines two truth degrees, the membership degree in the fuzzy set *Good* and the degree in which *George Clooney* is the *leading_role* of the movie. Condition (1) is expressed by `OR_GOD` which combines the result of (2) with the degree of membership in the fuzzy set *Thriller*.

FSA-SPARQL enables expressing more complex preferences, such as in the following example, the retrieval of information from a fuzzy RDF data set of hotels described by crisp attributes *name* and *price*, as well as fuzzy attributes *quality* and *style*.

```
SELECT ?Name ?p ?d
WHERE {
  ?Hotel hotel:name ?Name .
  ?Hotel hotel:price ?p .
  ?Hotel f:type (hotel:quality hotel:Good ?g) .
  ?Hotel f:type (hotel:style hotel:Elegant ?e) .
  BIND(f:WSUM(0.1,f:MEAN(f:MORE_OR_LESS(?e),f:VERY(?g)),0.9,
               f:CLOSE_TO(?p,100,50)) as ?d) .
  FILTER(?d >= 0.75)
}
```

Here, the query retrieves hotels that are *very* good and *more or less* elegant, with a price *close to* 100 euros (from 50 to 150 euros). But the query gives much more importance to the price. To express this problem, the fuzzy aggregator `WSUM` is used, so elegance and

[4] Here the term connective covers t-norms (conjunctions), t-conorms (disjunctions) as well as any other operator.

quality are weighted with *0.1*, while price is weighted with *0.9*. Additionally, `VERY` is used to qualify the membership degree for the fuzzy set *Good*, while `MORE_OR_LESS` is used to qualify the membership degree for the fuzzy set *Elegant*. The operator `MEAN` is used to combine both truth degrees. Finally, `CLOSE_TO` is used to approximate the prices to 100 euros.

4. Encoding FSA-SPARQL in $\mathcal{FASILL}$

As commented in the Introduction, to tune fuzzy sets and queries, a FSA-SPARQL transformation is carried out into the fuzzy logic programming language $\mathcal{FASILL}$ in order to use $\mathcal{FASILL}$ as a tuning engine. In this section, after introducing the language $\mathcal{FASILL}$ we illustrate the transformation of FSA-SPARQL into $\mathcal{FASILL}$ followed by the presentation of the tuning engine of $\mathcal{FASILL}$.

4.1. The language $\mathcal{FASILL}$

$\mathcal{FASILL}$ is a fuzzy logic programming language that associates a complete lattice with each program to model notions of fuzzy truth degrees and fuzzy connectives beyond the simpler boolean lattice typically used in crisp logic languages.[5] Similarly to the PROLOG language, programs consist of a set of facts and logic rules, together with a goal. However, in $\mathcal{FASILL}$, the calculated *fuzzy computed answers* (fca's, in brief) for the goals are given by pairs of truth degrees and substitutions (for instance, "`<0.695, {X/hydropolis}>`"). Additionally, facts define the truth degrees associated with atoms, and logic rules use fuzzy connectives and aggregators to combine atoms (and possibly truth degrees) in their bodies.

More exactly, a *complete lattice* is a partially ordered set $(L, \leq)$ such that every subset S of L has infimum and supremum elements. Then, it is a bounded lattice, that is, it has bottom and top elements, denoted by $\bot$ and $\top$, respectively. In this paper we use the lattice $([0,1], \leq)$, where $\leq$ is the usual ordering relation on real numbers, and three sets of conjunctions/disjunctions corresponding to the fuzzy logics of Gödel, Łukasiewicz and Product (with different capabilities for modeling *pessimistic*, *optimistic* and *realistic scenarios*), whose truth functions are defined in Fig. 1.

It is also possible to include other fuzzy connectives (and aggregators), such as the arithmetical average $@_{\texttt{mean}}(x,y) \triangleq (x+y)/2$ or the linguistic modifier $@_{\texttt{very}}(x) \triangleq x^2$.

Given a complete lattice L, we consider a first-order language $\mathcal{L}_L$ built upon a signature Σ_L, that contains the elements of a countably infinite set of variables $\mathcal{V}$, function and predicate symbols (denoted by $\mathcal{F}$ and Π, respectively) with an associated arity—usually expressed as pairs f/n or p/n, respectively, where n represents its arity—, and the truth degree literals Σ_L^T and connectives Σ_L^C from L. Therefore, a well-formed formula in $\mathcal{L}_L$ can be either:

- A *value* $v \in \Sigma_L^T$, which will be interpreted as itself, i.e., as the truth degree $v \in L$.
- $p(t_1, \ldots, t_n)$, if $t_1, \ldots, t_n$ are terms over $\mathcal{V} \cup \mathcal{F}$ and p/n is an n-ary predicate. This formula is called *atomic* (atom, for short).
- $\varsigma(e_1, \ldots, e_n)$, if $e_1, \ldots, e_n$ are well-formed formulas and ς is a n-ary connective (or aggregator) with the truth function $[\![\varsigma]\!] : L^n \mapsto L$.

A L-expression is a goal without atoms, that is, a well-formed formula only containing connectives (and aggregators) and truth degrees of L, and a $\mathcal{FASILL}$ program over a complete lattice L is a set of program rules, where each rule is a formula $A \leftarrow \mathcal{B}$, where the following conditions hold:

- A is an atomic formula of $\mathcal{L}_L$ (the head of the rule);
- $\mathcal{B}$ (the body of the rule) is a goal, i.e., a well-formed formula of $\mathcal{L}_L$.
- $\leftarrow$ is an implication symbol that connects the head and the body in the rule.

As an example, the following code represents a $\mathcal{FASILL}$ program:

```
elegant(hydropolis) <- 0.9.
elegant(ritz) <- 0.8.
close(hydropolis, metro) <- 0.7.
good_hotel(X) <- @mean(elegant(X), @very(close(X, metro))).

<- good_hotel(X).
```

Here, the *good_hotel* predicate assigns a degree of truth to a hotel in terms of the average (*@mean*) of its elegance and the strong (*@very*) proximity to a metro station.

As previously said, each $\mathcal{FASILL}$ program includes the definition of a *lattice* of truth degrees, whose elements and ordering relation are defined by means of predicates *member*, *members* and *leq*, respectively. The predicate *members* enumerate a finite subset of truth degrees to be used at tuning time. The bottom and top elements of the lattice are identified by the predicates *bot* and *top*, while the supremum and distance between two given truth degrees are expressed by the predicates *supremum* and *distance*,

[5] $\mathcal{FASILL}$ also admits the presence of similarity relations between symbols and terms, but this extra fuzzy feature is out of the scope of this paper since it is not used in the tool we have implemented with $\mathcal{FASILL}$ for evaluating and tuning FSA-SPARQL queries.

```
% Elements
    member(X) :- number(X), 0 =< X, X =< 1.
    members([0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]).
% Distance
    distance(X,Y,Z) :- Z is abs(Y-X).
% Ordering relation
    leq(X,Y) :- X =< Y.
% Bottom, top and supremum
    bot(0.0). top(1.0).
    supremum(X, Y, Z) :- Z is max(X, Y).
% Binary conjunctions (t-norms) and disjunctions (t-conorms)
    and_prod(X,Y,Z) :- Z is X*Y.
    and_godel(X,Y,Z) :- Z is min(X,Y).
    and_luka(X,Y,Z) :- Z is max(X+Y-1.0,0.0).
    or_prod(X,Y,Z) :- U1 is X*Y, U2 is X+Y, Z is U2-U1.
    or_godel(X,Y,Z) :- Z is max(X,Y).
    or_luka(X,Y,Z) :- Z is min(X+Y,1).
% Aggregators
    agr_mean(X,Y,Z) :- Z is (X+Y)/2.
    agr_very(X,Y) :- Y is X*X.
% Default connectives
    tnorm(godel). tconorm(godel).
```

Fig. 5. PROLOG clauses modeling the $([0,1],\leq)$ lattice used in $\mathcal{FASILL}$ programs.

respectively. Furthermore, fuzzy operators in such lattice (including the by default conjunction and disjunction connectives selected in the predicates *tnorm* and *tconorm*) are defined in a PROLOG style as any other predicate. For example, the PROLOG code in Fig. 5 models the lattice used in the previous $\mathcal{FASILL}$ program.

Note that apart from Fig. 1, the behavior of $\mathcal{FASILL}$ operators `&luk`, `&god`, `&prod`, `|luk`, `|god`, `|prod`, `@mean` and `@very` can be also easily induced from the PROLOG-based lattice definitions in Fig. 5. $\mathcal{FASILL}$ permits PROLOG style facts and rules in its syntax, according to the encoding $\mathcal{FASILL}$ into PROLOG, in which some arguments of PROLOG facts and rules can work as truth degrees. For example, the following (which simulates the previous one) is accepted by $\mathcal{FASILL}$:

```
elegant(hydropolis,0.9).
elegant(ritz,0.8).
close(hydropolis,metro,0.7).
good_hotel(X,TH):- elegant(X,TE),
     close(X,metro,CM),
     agr_very(CM,VCM),
     agr_mean(TE,VCM,TH).

<- good_hotel(X,T).
```

In order to contrast the crisp and fuzzy behaviors of the PROLOG and $\mathcal{FASILL}$ languages, respectively, please observe that the execution of the goal "`good_hotel(X,T)`" in any PROLOG interpreter produces as output the computed answer "`X = hydropolis, T = 0.695`", which is strongly related to the fuzzy computed answer "`<1.0,{T/0.695,X/hydropolis}>`" obtained after executing the same goal with the $\mathcal{FASILL}$ system. On the other hand, the execution of the goal"`good_hotel(X,T) &prod T`" with $\mathcal{FASILL}$ would result the fuzzy computed answer "`<0.695,{T/0.695,X/hydropolis}>`", where the first component of the fuzzy computed answer is the truth degree (also reported in the binding `T/0.695`) assigned to the hotel `hydropolis` (see the binding `X/hydropolis`). To conclude, note that this fuzzy computed answer is strongly related again to the one "`<0.695, {X/hydropolis}>`" that $\mathcal{FASILL}$ would obtain for the simpler goal "`good_hotel(X)`" with respect to the initial $\mathcal{FASILL}$ program we introduced at the beginning of this sub-section.

4.2. *Transforming* FSA-SPARQL *into* $\mathcal{FASILL}$

FSA-SPARQL queries are transformed into a $\mathcal{FASILL}$ program by translating FSA-SPARQL queries into $\mathcal{FASILL}$ (PROLOG style) rules, and fuzzy RDF triples are transformed into $\mathcal{FASILL}$ (PROLOG style) facts. For example, the following query requesting *high* rate and *medium* duration movies, weighting both properties with *0.5*:

```
SELECT ?Movie ?Rank
WHERE {
  ?Movie f:type (movie:rate movie:high ?r).
  ?Movie f:type (movie:duration movie:medium ?d) .
  BIND(l:WMEAN(0.5,?r,?d) as ?Rank)
  FILTER (?Rank > 0.8)
}
```

is translated into $\mathcal{FASILL}$ as:

```
q(Movie,Rank):- rdf(Movie,f:type,X0),
   rdf(X0,f:onProperty,movie:rate),
   rdf(X0,rdf:type,movie:high),
   rdf(X0, f:truth,R),
   rdf(Movie, f:type,X1),
   rdf(X1, f:onProperty,movie:duration),
   rdf(X1, rdf:type,movie:medium),
   rdf(X1, f:truth,D),
   l:WMEAN(0.5,R,D,Rank),
   {Rank > 0.8^^xsd:decimal}.
```

With this encoding, the predicate *q* has the same behavior as the FSA-SPARQL query. RDF triples are represented by predicates *rdf(s,p,o)* in which *s* is the subject, *p* is the property and *o* is the object. So, in particular, according to the proposed crisp representation of fuzzy RDF triples, *movie:OceansEleven f:type* (*movie:rate movie:high 0.5*) becomes:

```
rdf(movie:OceansEleven,f:type,_:OceansElevenrh).
rdf(_:OceansElevenrh,f:onProperty,movie:rate).
rdf(_:OceansElevenrh,rdf:type,movie:high).
rdf(_:OceansElevenrh,f:truth,0.5^^xsd:decimal).
```

where `_:OceansElevenrh` is the blank node associated with *Ocean Eleven* and *high rate*. Additionally, the FSA-SPARQL operators (including arithmetical comparators and fuzzy connectives) rely on the $\mathcal{FASILL}$ lattice, whose definitions can be easily reused by means of the *truth_degree* predicate defined as follows:

```
'{}'(X > Y) <- @gt(X, Y).

l:AND_LUK(X,Y,Z):- truth_degree(&luk(X,Y), Z).
l:AND_GOD(X,Y,Z):-truth_degree(&god(X,Y), Z).
l:AND_PROD(X,Y,Z):-truth_degree(&prod(X,Y), Z).

l:OR_LUK(X,Y,Z):-truth_degree(|luk(X,Y), Z).
l:OR_GOD(X,Y,Z):-truth_degree(|god(X,Y), Z).
l:OR_PROD(X,Y,Z):-truth_degree(|prod(X,Y), Z).

l:MEAN(X,Y,Z) :- truth_degree(@mean(X,Y), Z).
l:WMEAN(W,X,Y,Z) :- truth_degree(@wmean(W,X,Y), Z).
...
```

FSA-SPARQL is a super language of SPARQL. All SPARQL constructors are FSA-SPARQL constructors. FSA-SPARQL adds fuzzy patterns, connectives, and aggregators to SPARQL. However, our tuning method is currently restricted to the case of so-called (fuzzy) *Constrained Graph Pattern (CGP)* queries [17], which are queries with fuzzy triple patterns and filters. Such (fuzzy) CGP queries are easily transformed to $\mathcal{FASILL}$ in which fuzzy triple patterns are translated to $\mathcal{FASILL}$ atoms and filters are transformed to Boolean conditions [7]. Fuzzy connectives and aggregators are defined in $\mathcal{FASILL}$ by rules.

The conversion of FSA-SPARQL to $\mathcal{FASILL}$ does not add extra computation cost: an FSA-SPARQL query becomes a $\mathcal{FASILL}$ rule. However, querying in $\mathcal{FASILL}$ is, in general terms, more costly than in FSA-SPARQL. The implementation of FSA-SPARQL has been developed on top of an SPARQL machine (more concretely, on top of JENA ARQ[6]). Unfortunately, dedicated querying machines are usually more efficient than general purpose machines. However, the tuning process normally works with a small dataset that is used as a test case suite. In any case, we are aware that for larger test case suites, the tuning process answer time can be deteriorated.

4.3. Tuning $\mathcal{FASILL}$

Let us start this section by introducing a few formal details on the symbolic extension of $\mathcal{FASILL}$, called $\mathcal{SFASILL}$, collected from Moreno et al. [24], Moreno and Riaza [25]. Given a complete lattice L, we consider an augmented signature $\Sigma_L^{\#}$ producing an augmented language $\mathcal{L}_L^{\#} \supseteq \mathcal{L}_L$ which may also include a number of symbolic values and symbolic connectives that do not belong to L. Symbolic objects are usually denoted $o^{\#}$ by a superscript # and, in our tool, their identifiers always start with #. An $L^{\#}$-*expression* is now a well-formed formula of $\mathcal{L}_L^{\#}$ that is composed of values and connectives from L as well as symbolic values and connectives.

A *symbolic substitution* is a mapping $\Theta : \Sigma_L^{T^{\#}} \cup \Sigma_L^{C^{\#}} \rightarrow \Sigma_L^T \cup \Sigma_L^C$ that maps each symbolic value of $\Sigma_L^{T^{\#}}$ with an element Σ_L^T and each symbolic connective of $\Sigma_L^{C^{\#}}$ with a concrete connective of Σ_L^C. We let $\mathrm{sym}(o^{\#})$ denote the symbolic values and connectives in $o^{\#}$. In this setting, the symbolic execution of a goal w.r.t. a symbolic program reports what we call *symbolic computed answers*, sfca's in brief, which are pairs of $L^{\#}$-*expression* and substitutions (instead of truth degrees and substitutions, as occurs with fca's) like, for instance, "`<#@op(&godel(0.6,#s0),0.16), X/hydropolis>`".

[6] https://jena.apache.org/documentation/query/.

Tuning Algorithm

Input: A $\mathcal{SFASILL}$ program $\mathcal{P}^{\#}$ and a number of test cases $v_i \rightarrow Q_i$, $i = 1, \ldots, k$. *(*)*
Output: *A symbolic substitution* Θ_τ and its *deviation* τ.

1. For each test case $v_i \rightarrow Q_i$, compute the corresponding sfca $\langle Q'_i, \theta_i \rangle$ for each $\langle Q_i, id \rangle$ in $\mathcal{P}^{\#}$.
2. Generate a finite number of symbolic substitutions, say $\Theta_1, \ldots, \Theta_n$, $n > 0$, for $\bigcup_{i=1}^{k} \mathrm{sym}(Q'_i)$.
3. $\tau = \infty$; For each symbolic substitution $j \in \{1, \ldots, n\}$ and $\tau \neq 0$
   $z = 0$; For each test case $i = \{1, \ldots, k\}$ and $\tau > z$
   evaluate $\langle Q'_i \Theta_j, \theta_i \rangle$ to obtain $\langle v_{i,j}; \theta_i \rangle$
   let $z = z + distance(v_{i,j}, v_i)$.
   if $z < \tau$ then { $\tau = z$; $\Theta_\tau = \Theta_j$ }. *(**)*

   Finally, return symbolic substitution Θ_τ and its deviation τ.

(*) Include ϕ representing a *tolerance level* for permitted deviations.
(**) if $\tau \leq \phi$ then break.

Fig. 6. Algorithm for thresholded tuning of $\mathcal{SFASILL}$ programs.

Now, we are ready to explain how the tuning engine of $\mathcal{FASILL}$ works. As we have just commented, it is possible to transform a $\mathcal{FASILL}$ program into a symbolic $\mathcal{SFASILL}$ one, simply by replacing truth degrees and fuzzy connectives, as well as the parameters of the aggregators, in the program rules by *symbolic constants* with shape *#identifier*. Such symbolic constants serve to specify some fuzzy connectives and aggregators and their parameters that should be synthesized (that is, guess the most appropriate for them). For example, in the following symbolic $\mathcal{FASILL}$ program:

```
elegant(hydropolis) <- #s.
elegant(ritz) <- 0.8.
close(hydropolis, metro) <- 0.7.
good_hotel(X) <- @mean(elegant(X), @very(close(X, metro))).
```

we have introduced a symbolic constant *#s* in order to compute the membership degree of the hotel *hydropolis* to the fuzzy set *elegant*. In other words, the underlined question of the symbolic $\mathcal{FASILL}$ program is how elegant the hotel *hydropolis* should be for being considered a good hotel. The search space for instantiating the symbolic constant *#s* will be the ten concrete truth degrees listed by predicate *members* in the lattice of Fig. 5.

Now, the tuning process needs a sample (a *test case*) to synthesize *#s*. For example, let us assume that we have some evidence on the degree of membership of *hydropolis* in the fuzzy set *good_hotel*. This test case can be specified in $\mathcal{FASILL}$ as follows:

```
0.8 -> good_hotel(hydropolis)
```

Intuitively, this test case imposes that hotel *hydropolis* should have a membership degree of *0.8* to *good_hotel*. It could happen that the imposed test case uses a low membership degree, for instance:

```
0.2 -> good_hotel(hydropolis)
```

which works similarly to the former case, but here the hotel *hydropolis* is not considered a good hotel.

As we have said, $\mathcal{FASILL}$ is also able to guess fuzzy aggregators for symbolic constants appearing in the bodies of program rules. For example, the following symbolic $\mathcal{FASILL}$ program can be considered:

```
elegant(hydropolis) <- 0.9.
elegant(ritz) <- 0.8.
close(hydropolis, metro) <- 0.7.
good_hotel(X) <- #@op(elegant(X), @very(close(X, metro))).
```

in which a fuzzy aggregator becomes a symbolic constant *#@op*. Here, the search space for *#@op* is the set of fuzzy (binary) connectives defined in the $\mathcal{FASILL}$ lattice. The underlying question of such a program is: Which is the best combination of elegance and strong closeness to accomplish with the test cases? Such a question arises when the test cases consider good (or bad) hotels and one wants to know how their characteristics can be better combined.

The tuning engine looks for an optimal solution for symbolic constants. Optimality means that all answers should reduce their (average) deviations as much as possible to the truth degrees expressed in the test cases. In the above situation, with only one test case, the synthesized value for *#s* would surely lead to a low deviation, but in the general case, when the set of symbolic constants and the set of test cases are greater, the deviation could be higher. The search space for values for symbolic constants is specifically defined by the tuner user.

After informally explaining with the previous example the goal pursued by the $\mathcal{FASILL}$ tuner, let us now describe in a more detailed way the original tuning algorithm we first presented in Moreno et al. [24], Moreno and Riaza [25][7] The tuning engine proceeds in three stages, as shown in the algorithm in Fig. 6 (visit https://dectau.uclm.es/fasill/sandbox and see Julián-Iranzo et al. [18] for more details).

Step 1. Symbolic execution of test cases

[7] In Moreno and Riaza [26,27] we introduce a more recent tuning technique that copes with similarity relations but apart from being more complex, it is useless in this paper since FSA-SPARQL language has not yet been equipped with fuzzy commands based on similarities.

To proceed efficiently, our algorithm avoids an enormous amount of redundant computational steps by performing the symbolic execution of each goal in test cases with respect to the original symbolic program (i.e. before being instantiated with a symbolic substitution). Note that the algorithm computes only once the sfca's (i.e. the symbolic fuzzy computed answers possibly containing symbolic constants) for the goals in the set of test cases. Observe also that later, in the third step, each symbolic substitution is applied to each sfca, thus obtaining a L-expression (that is, a pure arithmetical expression only containing truth degrees and connectives, but neither symbolic constants nor atoms) whose final evaluation produces the corresponding fca.

Step 2: Generation of symbolic substitutions

$\mathcal{FASILL}$ creates a set of symbolic substitutions for mapping symbolic constants with concrete truth degrees and concrete connectives as follows:

- Symbolic constants of type #*label*, are mapped to the concrete truth degrees listed in the only parameter of the predicate `members`.
- Symbolic constants of type #&*label*, #|*label* and #@*label*, are linked to concrete conjunction, disjunction, and aggregation connectives, respectively, defined by means of predicates with names `and_label`, `or_label` or `agr_label`. Moreover, a symbolic constant of type #?*label* can be matched with any type (conjunction, disjunction, or aggregation) of connective.

Step 3. Threshold check of symbolic substitution

In this step, it is mandatory to evaluate the distance between pairs of truth degrees when checking the deviations associated with symbolic substitutions.[8] In our tool, this operation is defined in the lattice box by means of a set of PROLOG clauses associated with the predicate `distance/3`, which returns on its third parameter a real number representing the distance between the truth degrees in the first and second arguments. In our case, we simply use the following clause (see again Fig. 5):

```
distance(X,Y,Z) :- Z is abs(Y-X).
```

In order to improve the efficiency of the tuning algorithm once again, when computing deviations, we use *thresholding* techniques, which are quite standard in the fuzzy logic area for prematurely disregarding useless computations leading to non-significant answers. The idea is to make use of a *deviation threshold* τ to determine when a *partial* solution is acceptable. The value of τ is initialized to ∞ (in practice, a very large number) in Step 3. Then, this deviation threshold is dynamically decreased whenever we find a symbolic substitution with an associated deviation lower than the actual value of τ. Moreover, a partial solution is discarded as soon as the cumulative deviation computed so far is greater than τ. In general, the number of solutions discarded increases as the value of τ decreases, thus improving the pruning power of the thresholding.

Once the original algorithm of the $\mathcal{FASILL}$ tuner presented in Moreno et al. [24], Moreno and Riaza [25] has been described, we wish to include a final improvement on it, represented by the lines marked with (*) and (**) in Fig. 6, and which is especially suitable for the application we focus on in this article. The idea is to relax the initial objective of a tuning problem of finding the *best solution* (that is, the symbolic substitution with the lowest deviation) simply by looking for the *first one* whose deviation falls below a given value ϕ, representing a *tolerance level*. Note that this simplification is easy to implement in the tuning algorithm of Fig. 6 simply by adding the tolerance level ϕ as input (*) that is used to prematurely break the loop in Step 3 as soon as $\tau \leq \phi$ (**). This new stopping condition drastically reduces the search space and improves the performance of the $\mathcal{FASILL}$ tuner when users invoke it with greater tolerance.

5. Tuning FSA-SPARQL

Now, we will explain how $\mathcal{FASILL}$ is able to tune both FSA-SPARQL fuzzy sets and queries. Fig. 7 sketches the sequences of tasks that cooperate at tuning time. From an initial query definition, numeric attributes are extracted from the query. Numerical attributes become fuzzy attributes. For simplicity, we consider three fuzzy sets (*low*, *medium*, and *high*) for each numerical attribute. Next, a selection of test cases is made for fuzzy attributes, and the tuning process looks for trapezoidal membership functions. If the deviation is low, the query could be executed; otherwise, the tuning process should be repeated again. After the query execution, the answers are inspected. If the answers are the expected ones, the tuning process is finished. Otherwise, the expression `FILTER` (if any) is removed, and symbolic constants are introduced in the query. A selection of test cases for the query –expected answers– is made. Next, the tuning process looks for fuzzy operators and parameters for each symbolic constant. If the deviation is low, the query is again executed after adding the `FILTER` condition. The process is repeated until the expected answers are returned.

We remark that the tuning process is directly made on fuzzy queries by introducing symbolic constants in them. The translation to $\mathcal{FASILL}$ is transparent to the user, since the user is not required to know what the $\mathcal{FASILL}$ programs and the symbolic $\mathcal{FASILL}$ programs are. The developed Web tool (https://dectau.uclm.es/floper/tuning) facilitates the introduction of test cases in both fuzzy sets and query tuning processes and makes the tuning strategy more intuitive. The tool shows a graphical representation of the trapezoidal membership functions and reports the deviation. Although the tuning process involves trial and error phases, a few

[8] The final deviation also is very often reported in a normalized way, that is, as the real number τ/k in the unit interval, which represents the average deviation of the k test cases.

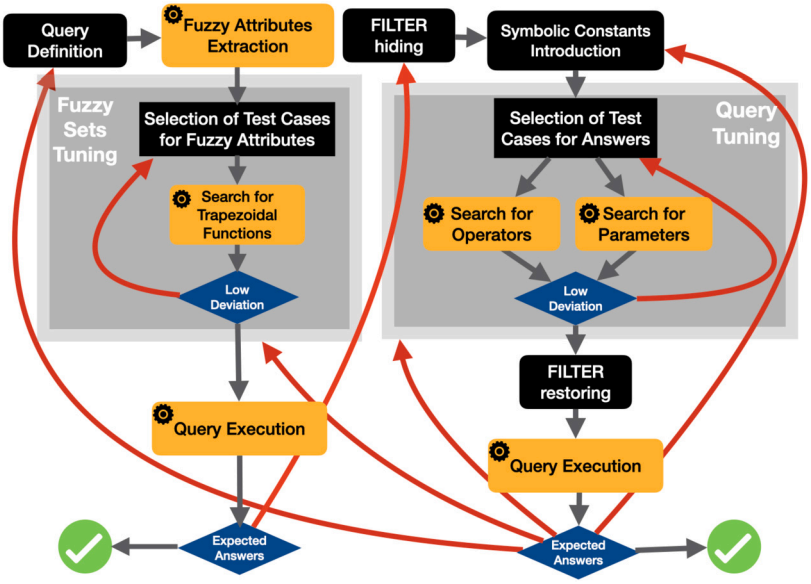

Fig. 7. Tuning Processes.

iteration steps suffice to find the desired answers, since we can always relax as much as we want the tolerance level of the achieved solutions w.r.t. the preferences expressed on the test cases.

5.1. *Tuning fuzzy sets in* FSA-SPARQL

Due to their high generality and simplicity, trapezoidal membership functions have become a fairly natural and standard way to define fuzzy sets [41,29]. As the leftmost image in Fig. 8 shows, each fuzzy set A is defined as a trapezoid based on four parameters (a, b, c and d) verifying the following mathematical formula:

$$\mu_A(x) = max(min((x-a)/(b-a), 1, (d-x)/(d-c)), 0)$$

which is equivalent to:

$$\mu_A(x) = \begin{cases} 0 & \text{if } x \leq a \\ (x-a)/(b-a) & \text{if } a \leq x \leq b \\ 1 & \text{if } b \leq x \leq c \\ (d-x)/(d-c) & \text{if } c \leq x \leq d \\ 0 & \text{if } d \leq x \end{cases}$$

This function has been coded inside the lattice of truth degrees and aggregators of $\mathcal{FASILL}$ as follows:

```
agr_trapezoidal(A^^T, _B^^T, _C^^T, D^^T, X^^T, 0^^T) :-
        X < A ; X > D.
agr_trapezoidal(A^^T, B^^T, _C^^T, _D^^T, X^^T, Y^^T) :-
        A =< X, X =< B, (B-A =:= 0 -> Y = 1 ; Y is (X-A)/(B-A)).
agr_trapezoidal(_A^^T, B^^T, C^^T, _D^^T, X^^T, 1^^T) :-
        B =< X, X =< C.
agr_trapezoidal(_A^^T, _B^^T, C^^T, D^^T, X^^T, Y^^T) :-
        C =< X, X =< D, (D-C =:= 0 -> Y = 1 ; Y is (D-X)/(D-C)).
```

As was said before, we will consider three fuzzy sets (*low*, *medium*, and *high*) for each numerical attribute and tuning consists of searching the four parameters of their corresponding trapezoidal membership functions. These parameters, representing the lower and upper limits of each trapezoid, will be represented by symbolic constants labeled $\#ll, \#ls, \#us$ and $\#ul$. The tuning process works at the $\mathcal{FASILL}$ level by directly manipulating the representation of fuzzy RDF triples. More concretely, for each numeric fuzzy attribute and each fuzzy set (*low*, *medium*, and *high*) a symbolic $\mathcal{FASILL}$ rule is built. For instance, for *rate* and *high* in Ocean's Eleven:

```
rdf(_:OceansElevenrh,f:truth,TV^^xsd:decimal):-
   truth_degree(@trapezoidal(#ll,#ls,#us,#ul,TV^^xsd:decimal))
```

where `_:OceansElevenrh` is the blank node associated with *Ocean Eleven* and *high rate*. A test case *Ocean's Eleven* | *0.7* has the following form:

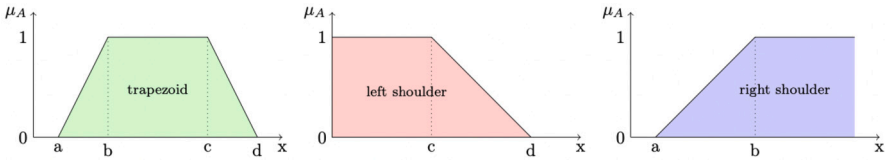

Fig. 8. Graphical view of fuzzy sets based on trapezoidal membership functions.

```
rdf(_:OceansElevenrh,f:truth,0.7^^xsd:decimal)
```

As usual, for modeling a fuzzy set of type *low*, the first pair of symbolic constants are fixed to the bottom element of the attribute domain, which simplifies the shape of the trapezoid and produces what is known as a *left shoulder* membership function. Symmetrically, the functions *right shoulder* are associated with any *high* fuzzy set simply by fixing $\#us$ and $\#ul$ to the top element of the domain. Fig. 8 displays these three different shapes, that is, trapezoidal, left shoulder, and right shoulder, representing, respectively, fuzzy sets *low*, *medium* and *high*.

Here, the search space is no longer the entire [0,1] real interval, since *members* must now be defined as a discrete set of numeric values representing, for example, values of the *rate* and *duration*. So, the fuzzy sets tuning technique tries to adjust a sample (i.e. a set of test cases) to a trapezoidal membership function such that, instead of optimally approximating its shape, the goal is to return the trapezoidal function obtained from a discrete partition of the domain, which minimizes its deviation to the sample.

After tuning the fuzzy sets, new $\mathcal{FASILL}$ rules are generated for each fuzzy set (*high*, *low* or *medium*) associated with a given fuzzy property and the given values $\#ll, \#ls, \#us$ and $\#ul$. For example, if $\#ll = 1.05, \#ls = 9.5, \#us = 9.5$ and $\#ul = 9.5$ are the values of the *high rate*, the $\mathcal{FASILL}$ rule is as follows (where the conjunction symbol & refers by default to $\&_{godel}$ – that is, the minimum –, as the predicate *norm* indicates in Fig. 5):

```
rdf(X,f:truth,Y) <-
atom_concat(Z,rh,X) &
rdf(Z,movie:rate,V^^_) &
truth_degree(@trapezoidal(1.05^^_,9.5^^_,9.5^^_,9.5^^_,V^^xsd:decimal),Y).
```

This $\mathcal{FASILL}$ rule generates the membership degrees of any movie to the fuzzy set *high* with respect to the *rate*.

5.2. *Tuning queries in* FSA-SPARQL

Normally, the query tuning technique is used when the user does not find the expected results of the query. For example, let us consider the following query:

```
SELECT ?Movie ?Rank
WHERE {
  ?Movie f:type (movie:rate movie:high ?r).
  ?Movie f:type (movie:duration movie:medium ?d) .
  BIND(l:AND_PROD(?r,?d) as ?Rank)
}
```

where the conditions imposed are *medium* for *duration* and *high* for *rate*. Let us remark that even though a `FILTER` can be imposed on the query, in the tuning stage, we remove the filtering to get all the (potential) answers. In this example, the user can suspect that certain movies have a *medium duration* and *high rate* but the reported truth degrees are low. The same happens, for example, when certain movies that the user expects to have *high duration* and *low rate* are reported with a high score. In both cases, the user can figure out that the query fails to correctly express the condition and that probably the fuzzy connectives/aggregators or their parameters have not been correctly selected. The tuning mechanism will improve the truth degrees associated with the answers and, to this end, requires a sample (test cases) to be submitted by the user. Queries to be tuned in FSA-SPARQL use symbolic constants as follows:

```
SELECT ?Movie ?Rank
WHERE {
  ?Movie f:type (movie:rate movie:high ?r).
  ?Movie f:type (movie:duration movie:medium ?d) .
  BIND(l:WMEAN('#s',?r,?d) as ?Rank)
}
```

where the symbolic constant *#s* is used for representing the guessing of the parameter of the fuzzy operator `WMEAN`. The underlying request is the estimation from the test cases of the relevance of the properties of *rate* and *duration*.

According to the FSA-SPARQL transformation into $\mathcal{FASILL}$, symbolic FSA-SPARQL queries become symbolic $\mathcal{FASILL}$ programs, such as the following:

```
p(Movie,Rank):- rdf(Movie,f:type,X0),
      rdf(X0,f:onProperty,movie:rate),
      rdf(X0,rdf:type,movie:high),
      rdf(X0, f:truth,R),
```

Table 1
Tuning of the fuzzy sets: popularity and number of votes.

Attribute	Fuzzy Set	Test Cases	Deviation
Popularity	*High*	John Hurt, 0.9 | John Cusack, 0.2 | John Schneider, 0	0.05
Popularity	*Medium*	John Hurt, 0.2 | John Cusack, 0.9 | John Schneider, 0.1	0.05
Popularity	*Low*	John Hurt, 0.1 | John Cusack, 0.1 | John Schneider, 0.9	0.05
Vote Count	*High*	Captain America, 0.8 | Star Wars: The Rise.. , 0.1 | The Twilight Zone, 0	0.03
Vote Count	*Medium*	Captain America, 0.1 | Star Wars: The Rise.. , 0.9 | The Twilight Zone, 0.2	0.03
Vote Count	*Low*	Captain America, 0 | Star Wars: The Rise.. , 0.1 | The Twilight Zone, 0.8	0.03

```
rdf(Movie, f:type,X1),
rdf(X1, f:onProperty,movie:duration),
rdf(X1, rdf:type,movie:medium),
rdf(X1, f:truth,D),
l:WMEAN(#s,R,D,Rank).
```

and the search space for RDF decimals can be specified for *#s*, as usual in $\mathcal{FASILL}$.

```
members([0.0^^xsd:decimal,0.1^^xsd:decimal,...,1.0^^xsd:decimal]).
```

Test cases can be specified by $\mathcal{FASILL}$ with this shape:

```
0.85^^_ -> p(movie:OceansEleven, TD) & TD
```

Here, depending on the degree of membership of *Ocean's Eleven* to fuzzy sets *high* and *medium* in *rate* and *duration*, respectively, the symbolic constant *#s* becomes greater or smaller.

Similarly to the $\mathcal{FASILL}$ case, symbolic constants can also be used to guess a fuzzy aggregator as follows:

```
SELECT ?Movie ?Rank
WHERE {
  ?Movie f:type (movie:rate movie:high ?r).
  ?Movie f:type (movie:duration movie:medium ?d) .
  BIND(l:APP('#?op',?r,?d) as ?Rank)
}
```

where *#?op* values are selected by the tuning engine from the set of binary fuzzy connectives and aggregators, that is, `AND_PROD`, `OR_PROD`, `AND_LUKA`, `OR_LUKA`, `MEAN`, etc. In general, the operator `APP` is used to encapsulate the search for fuzzy connectives and aggregators of a given arity.

6. Experiments

To validate our approach, we used datasets from online communities. The purpose of the experiments is to test whether we can tune queries (and the fuzzy sets involved in the query) to retrieve certain answers from online communities. More concretely, given a dataset D, a list of expected answers $A \subseteq D$, an FSA-SPARQL query Q, and a number of test cases T, the goal is to test whether it is possible to tune Q to Q' according to T so that the answers of A are returned by Q'. Whether an answer $a \in D$ is such that $a \in A$ is checked from an external resource E different from D. Q can be tuned to Q' by synthesizing some symbolic constants introduced in Q and / or selecting different fuzzy sets. Among the test cases T, a number of test cases can be *positive* (that is, with a high fuzzy score) or *negative* (that is, with a low fuzzy score). The deviation τ indicates how the score of the test cases T deviates from the score of Q'.

We have analyzed for each experiment the *precision*, that is, the fraction of expected answers retrieved by the query Q' among all answers retrieved by Q', the *recall*, that is, the fraction of expected answers that are retrieved by the query Q' among all expected answers retrieved by Q', as well as the *accuracy*, that is, the proportion of correct scores (both true positives and true negatives) among the total number of answers.

6.1. TMDB oldest actors whose name is John

The first experiment uses a dataset D retrieved from *The Movie Database (TMDB)*.[9] The list of expected answers A is the subset of D with old actors whose name is John. Whether an answer is expected or not is decided by the actor's age defined in the external resource E of Wikipedia.

Table 1 shows the results of the tuning of fuzzy sets. Attributes, fuzzy Sets, test Cases, and deviations are reported. A positive test case and two negative test cases are sufficient to tune all trapezoidal functions (*high*, *medium*, and *low*) for popularity and number of votes. The deviation τ is 0.05 and 0.03, respectively.

Now, the query Q is as follows[10]:

[9] https://www.themoviedb.org.

[10] The datasets of online communities are retrieved in JSON format with their APIs. A JSON-to-RDF transformation has been implemented to query them with FSA-SPARQL.

Table 2
Tuning of the TMDB query. In Tuned Query (n, m) means *WSUM(m,?p,n,?nv)*, where *?p* is the membership degree to the fuzzy sets of popularity rate and *?nv* is the membership degree to the fuzzy sets of the number of votes.

Actor	P	V1	V2	V3	B	E	HHP	HHW	HMW	HLW	MLW
Travolta	10.51	19748	5115	4312	1954	no	**0.98**	**0.97**	**0.52**	0.42	0
Hurt	8.39	18451	14527	13513	1940	no	**0.9**	0.31	0	0	0
Lithgow	8.22	23964	11346	8678	1945	no	**1**	**0.68**	0	0	0
Cusack	6.46	9105	3962	3740	1966	no	0.24	**0.72**	**0.53**	**0.52**	0
Cena	5.77	4003	2181	2122	1977	no	0	0.18	0.2	**0.8**	0.47
Malkovich	6.60	6958	5954	4928	1953	no	0.15	**0.91**	0.37	0.31	0
Litel	5.50	285	149	127	1892	**yes**	0	0	0	**0.8**	**1**
Leguizamo	5.79	13351	9387	8353	1964	no	0.12	0.15	0	0	0
Cho	6.66	8519	7445	7119	1972	no	0.11	0.36	0	0	0
Mills	5.40	1622	1500	288	1908	**yes**	0	0	0	**0.8**	**1**
Boyega	5.44	15029	11137	5779	1992	no	0	0	0	0.15	0
Cunningham	5.20	3030	1181	784	1932	**yes**	0	0	0	**0.8**	**0.81**
Carpenter	5.11	4126	3173	1866	1948	no	0	0	0	**0.8**	**0.52**
Dehner	4.76	641	516	418	1915	**yes**	0	0	0	**0.8**	**1**
Wayne	5.71	799	661	632	1907	**yes**	0	0.02	0.14	**0.8**	**0.85**
Hughes	4.73	7102	6380	5867	1950	no	0	0	0	0.14	0
Slattery	4.52	16956	15410	15057	1962	no	0	0	0	0	0
Rhys-Davies	4.74	17857	16389	15422	1944	no	0	0	0	0	0
Schneider	4.21	1631	806	761	1960	no	0	0	0	**0.8**	0
Pickard	4.11	455	418	244	1972	no	0	0	0	**0.8**	0
Deviation								0.8	0.66	0	0.09
Test Cases								JL | 0.8	JW | 0.8	JW | 0.8	JW | 0.9 and JD | 0.6
Tuned Query								(0,0)	(0,1)	(0,0.8)	(0.3,0.6)
Precision							0	0	0	0.5	0.83
Recall							0	0	0	1	1
Accuracy							0.6	0.55	0.65	0.75	0.95

```
SELECT ?Name ?Rank
WHERE {
  sn:root sn:list ?pp .
  ?pp sn:name ?Name .
  ?pp f:type (sn:popularity f:high ?Pop) .
  ?pp sn:known_for ?list .
  ?list sn:list ?mv .
  ?mv f:type (sn:vote_count f:high ?Vc)
  BIND(l:AND_PROD(?Pop,?Vc) AS ?Rank)
    FILTER(?Rank >= 0.5)
}
```

in which the popularity of the actors and the number of votes of the movies in which the actors play are used. Initially, *high* popularity and *high* number of votes are considered. The `AND_PROD` has been selected to combine popularity and votes. A `FILTER` of 0.5 is required. Presumably, the run of this query does not return the oldest actors of TMDB. It could be arguable that younger actors would have more popularity and a higher number of votes. This is what the tuning method should elucidate. That is, when tuning is applied with old actors as positive test cases T, the deviation τ should presumably be high. In such a case, the tuning mechanism enables finding a more appropriate connective and aggregator (i.e., `AND_PROD`, `OR_PROD`, `AND_LUKA`, `OR_LUKA`, `MEAN`, `WSUM`, etc.) as well as their parameters to combine a *high* popularity score and a *high* number of votes. In addition, tuning would allow us to test other combinations of fuzzy sets: *high-medium*, *high-low*, *medium-high*, etc.

Table 2 shows the results of the tuning process for only 20 actors. The column *Actor* shows the name of the actor, the column *P* shows popularity, the columns *V1*, *V2*, *V3* report the number of votes for the three most recent movies,[11] column *B* displays the birth date, column *E* shows whether the actor is old or not according to the Wikipedia birth date (a threshold of before 1940 has been imposed). Finally, columns *HHP*, *HHW*, *HMW*, *HLW* and *MLW* report the scores for each tuned query Q'. *HHP* is *high* popularity, *high* number of votes and `AND_PROD`; *HHW* is *high* popularity, *high* number of votes and `WSUM`; *HMW* is *high* popularity, *medium* number of votes and `WSUM`; *HLW* is *high* popularity, *low* number of votes and `WSUM`; and *MLW* is *medium* popularity, *low* number of votes and `WSUM`.

HHP shows the scores with `AND_PROD` and, thus, without tuning. Precision and recall are 0 and accuracy is 0.6. By tuning *high-high* and `WSUM` (HHW) with the test case *John Lite (JL)* | *0.8* the deviation τ is very high: *0.8*, and neither precision nor recall is improved. When tuning *high-medium*, `WSUM` (HMW) with the test case *John Wayne (JW)* | *0.8*, again a high deviation τ of *0.66* is obtained, and again precision and recall are not improved. For *high-low*, `WSUM` (HLW) and the test case *John Wayne (JW)* | *0.8* the tuning method reports a deviation τ of *0* and the tuned query `WSUM(0,?Pop,0.8,?Vc)`. This tuned query gets a recall of 1, but the precision is 0.5. Finally, with *medium-low*, `WSUM` (MLW) and the test cases *John Wayne (JW)* | *0.9* and *John Denver (JD)* | *0.6*,

[11] Let us remark that the showed scores are the maximum of the scores of the pairs actor-movie.

Table 3
Tuning of the fuzzy sets: subscribers, videos and video views.

Attribute	Fuzzy Set	Test Cases	Deviation
Subscribers	*High*	Beyonce, 0.9 | DJKhaled, 0.1 | Grabielle Aplin, 0	0.04
Subscribers	*Medium*	Beyonce, 0.2 | DJKhaled, 0.9 | Grabielle Aplin, 0.1	0.04
Subscribers	*Low*	Beyonce, 0.1 | DJKhaled, 0 | Grabielle Aplin, 0.9	0.04
Views	*High*	Beyonce, 0.9 | DJKhaled, 0.2 | Grabielle Aplin, 0	0.02
Views	*Medium*	Beyonce, 0.1 | Meghan Trainor, 0.9 | Mary JBlige, 0.1	0.02
Views	*Low*	Beyonce, 0 | Meghan Trainor, 0.1 | Mary JBlige, 0.9	0.02
Videos	*High*	KatyPerry, 0.1 | AmyWinehouse, 0 | Disney, 0.9	0.03
Videos	*Medium*	KatyPerry, 0.9 | AmyWinehouse, 0.1 | Disney, 0.1	0.03
Videos	*Low*	KatyPerry, 0.1 | AmyWinehouse, 0.9 | Disney, 0	0.03

a deviation τ of 0.09 is obtained and the tuned query `WSUM(0.3,?Pop,0.6,?Vc)` gets a precision of 0.83 and a recall of 1. All expected answers are reported, but *John Carpenter* is reported and is not expected.

6.2. YouTube most streamed artists

The second experiment uses a larger dataset D of YouTube VEVO channels.[12] In this case, the expected answers A is the subset of YouTube VEVO channels of the most streamed artists according to the external resource E of Spotify (accessed in late 2022).

Table 3 shows the results of the tuning of fuzzy sets. Attributes, fuzzy sets, test cases, and deviations are reported. Again, a positive test case and two negative test cases are sufficient to tune all trapezoidal functions (*high*, *medium*, and *low*) for subscribers, videos, and video views. The deviation is 0.04, 0.02 and 0.03, respectively.

Now, the query Q is as follows:

```
SELECT ?Title ?Rank
WHERE {
  sn:root sn:list ?vd .
  ?vd sn:snippet ?sn .
  ?sn sn:title ?Title .
  ?vd sn:statistics ?ss .
  ?ss f:type (sn:subscriberCount f:high ?Sc) .
  ?ss f:type (sn:viewCount f:high ?Wc) .
  ?ss f:type (sn:videoCount f:high ?Vc) .
  BIND(l:AND_PROD(l:AND_PROD(?Sc,?Wc),?Vc) AS ?Rank) .
    FILTER(?Rank>=0.5)
}
```

in which the number of subscribers, the number of videos, and the number of views are used and `AND_PROD` is used to combine their fuzzy scores. Again, `FILTER` is 0.5 and all fuzzy sets are *high*.

Table 4 shows the results of the tuning session. The column *Channel* shows the name of the channel, columns *Subs*, *Views*, *Videos* report the number of subscribers, views and videos, respectively, column *E* indicates whether or not the artist is one of the most streamed artists according to Spotify. The columns *HHHP*, *HHHW1*, *HHHW2* and *HHMP* show the scores reported from the tuned queries. Here we have again considered several combinations of fuzzy sets: (1) *high* number of subscribers, views, and videos and *high* number of subscribers and (2) *high* number of subscribers and views and *medium* number of videos.

Without tuning (column *HHHP*) the results are not satisfactory, getting a precision and recall of 0, and an accuracy of 0.68. With the fuzzy set *high* in the three attributes, we tuned `WSUM` twice (columns HHHW1 and HHHW2). With the test case *MalumaVEVO | 0.8*, we obtained a deviation τ of *0.09*, and the tuned query `WSUM(1,WSUM(0.7, ?Sc, 0.1, ?Wc),0,?Vc)`. The results are shown in column *HHHW1*. Precision and recall are 0.86, while the accuracy is 0.92. While all expected scores are retrieved, except *LadyGagaVEVO*, some unexpected ones also appear (*VEVO* and *DisneyMusicVEVO*). With the test cases *MalumaVEVO | 0.8* and *ShawnMendesVEVO | 0.7*, we get a deviation τ of *0.06*, and the tuned query `WSUM(1, WSUM(0.8, ?Sc, 0, ?Wc),0,?Vc)` gets a precision of 0.87, recall of 0.93 and accuracy of 0.94. The new results are shown in column *HHHW2*. The scores of *LadyGagaVEVO* and *SelenaGomezVEVO* have increased, now slightly below *0.5*. However, *VEVO* and *DisneyMusicVEVO* are still above *0.5*. Finally, with a *high* number of subscribers and views and a *medium* number of videos, and `AND_PROD` (column HHMP) we only got a precision of 1 and an accuracy of 0.92 but a recall of 0.73. However, by decreasing `FILTER` to 0.24, precision, recall, and accuracy are 1.

7. Conclusions and future work

This paper proposes a solution for fuzzy query customization according to a sample. Guided by test cases provided by users for expressing their preferences, automatic tuning is used here in a twofold way: 1) for building fuzzy sets and 2) for customizing connectives/operators and parameters involved in soft query conditions.

At the conclusion of the experiments, the following elements must be taken into account to ensure the success of the tuning method. The selection of good test cases is crucial for getting good trapezoidal membership functions, and thus well-modeled fuzzy

[12] https://www.youtube.com/user/VEVO.

Table 4
Tuning of the YouTube VEVO query. In Tuned Query (n, m, l) means *WSUM(1,WSUM(n, ?Sc, m, ?Wc),l,?Vc)*, where *?Sc*, *?Wc* and *?Vc* is the membership degree to the fuzzy sets of the number of subscribers, number of views and number of videos, respectively.

Channel	Subs.	Views	Videos	E	HHHP	HHHW1	HHHW2	HHMP
Vevo	19,800,000	568,759,886	1,534	no	0	**0.79**	**0.8**	0
Vevo Playlists	83,800	0	0	no	0	0	0	0
VEVOEspana	145,000	2,390,191	56	no	0	0.003	0	0
Vevo Italia	169,000	10,288,696	57	no	0	0.003	0	0
TheProducerTrapVEVO	175,000	6,435,193	110	no	0	0.007	0	0
Vevo France	226,000	31,421,416	79	no	0	0.005	0	0
SiaVEVO	9,990,000	8,955,834,601	105	no	0.02	0.007	0.34	0.11
shakiraVEVO	18,700,000	18,743,166,430	170	**yes**	0.11	**0.71**	**0.8**	**0.69**
DisneyMusicVEVO	21,800,000	15,083,858,983	1,354	no	**0.9**	**0.79**	**0.8**	0.1
BritneySpearsVEVO	5,000,000	5,657,953,670	108	no	0.0005	0.05	0.05	0.005
TaylorSwiftVEVO	26,500,000	19,649,231,267	147	**yes**	0.09	**0.71**	**0.8**	**0.89**
Vevo Polska	68,100	2,176,830	70	no	0	0.004	0	0
CamilaCabelloVEVO	5,220,000	4,517,726,647	56	no	0	0.06	0.006	0
KatyPerryVEVO	25,300,000	20,057,231,715	156	**yes**	0.1	**0.71**	**0.8**	**0.93**
GabrielleAplinVEVO	559,000	161,554,062	18	no	0	0.001	0	0
RihannaVEVO	25,600,000	15,788,306,682	106	**yes**	0.07	**0.7**	**0.8**	**0.64**
ShawnMendesVEVO	10,800,000	8,361,869,459	123	**yes**	0.03	**0.5**	**0.6**	0.29
CNCOVEVO	5,880,000	5,596,660,489	78	no	0.001	0.11	0.12	0.008
DisneyChannelLAVEVO	6,950,000	3,637,159,732	298	no	0	0.21	0.22	0
JustinBieberVEVO	32,400,000	21,309,756,207	159	**yes**	0.1	**0.71**	**0.8**	**0.95**
AbbaVEVO	974000	2,008,156,395	36	no	0	0.002	0	0
poisonVEVO	57700	169,894,762	18	no	0	0.002	0	0
SelenaGomezVEVO	14,800,000	8,815,197,211	164	**yes**	0.06	**0.71**	0.49	**0.58**
LadyGagaVEVO	9,130,000	9,693,366,635	169	**yes**	0.04	0.39	**0.5**	0.4
ArianaGrandeVevo	18,200,000	14,911,284,136	114	**yes**	0.08	**0.7**	**0.8**	**0.78**
TheBeatlesVEVO	1,290,000	1,873,199,624	53	no	0	0.003	0	0
KrystofVEVO	124,000	215,700,281	17	no	0	0.001	0	0
BeyoncéVEVO	13,100,000	11,329,635,588	142	**yes**	0.05	**0.68**	**0.78**	**0.79**
AdeleVEVO	15,500,000	9,172,451,849	31	**yes**	0.01	**0.7**	**0.8**	0.24
AmyWinehouseVEVO	1,570,000	1,550,804,777	29	no	0	0.001	0	0
EminemVEVO	25,400,000	15,459,281,102	86	**yes**	0.05	**0.7**	**0.8**	**0.53**
Flamingo Vevo	42,200	5,300,027	7	no	0	0	0	0
ChrisBrownVEVO	12,200,000	10,325,165,554	248	**yes**	0.11	**0.62**	**0.7**	**0.65**
MalumaVEVO	14,400,000	14,657,336,145	143	**yes**	0.08	**0.7**	**0.8**	**0.88**
NataliaKillsVEVO	153,000	71,374,912	24	no	0	0.001	0	0
DJKhaledVEVO	5,080,000	4,363,898,141	72	no	0	0.05	0.05	0
MabelVEVO	185,000	493,187,807	46	no	0	0.003	0	0
TheWeekndVEVO	10,800,000	10,112,955,199	87	**yes**	0.03	0.49	**0.62**	0.29
justintimberlakeVEVO	6,410,000	4,794,596,838	69	no	0	0.15	0.17	0
ZuccheroVEVO	115,000	183,309,092	56	no	0	0.003	0	0
JheneAikoVEVO	1,040,000	859,364,085	81	no	0	0.005	0	0
MaryJBligeVEVO	888,000	1,138,328,616	123	no	0	0.008	0	0
marcanthonyVEVO	2,560,000	3,686,725,113	93	no	0	0.006	0	0
DanielVEVO	104000	69,981,835	45	no	0	0.003	0	0
TinasheOfficialVEVO	1,070,000	630,581,461	90	no	0	0.006	0	0
NasVEVO	510000	318,638,964	98	no	0	0.006	0	0
TheDreamVEVO	251,000	245,771,088	56	no	0	0.003	0	0
KaraokeOnVEVO	820,000	134,215,476	8,127	no	0	0.1	0	0
KevinSol Vevo	420	41,014	54	no	0	0.003	0	0
MeghanTrainorVEVO	8,300,000	6,229,179,123	123	no	0.008	0.31	0.34	0.07
Deviation						0.09	0.06	
Test Cases						Maluma | 0.8	Maluma | 0.8 and ShawMendes | 0.7	
Tuned Query						(0.7,0.1,0)	(0.8,0,0)	
Precision					0	0.86	0.87	1
Recall					0	0.86	0.93	0.73
Accuracy					0.68	0.92	0.94	0.92

sets. The same can be said for query tuning, providing positive and negative cases (high and low scores). In general, several connectives and operators could be necessarily tested and combined in order to get a good query customization. When the deviation is high for the selected query test cases, a good strategy is to switch between the fuzzy sets *high*, *medium*, and *low*.

In future work, we plan to work on the following topics. We believe that the search for the most appropriate fuzzy sets (i.e. *high*, *medium*, and *low*) for a given query can be more automated. Furthermore, the tuning engine could automate the search for solutions adjusted to a *validation set* and a certain *precision threshold*. Tuning can also be more flexible, allowing the selection of fuzzy sets other than *high*, *medium* and *low*. We also found it useful to extend the tuning technique with the customization of `CLOSE_TO`, `AT_LEAST` and `AT_MOST` operators which combine $[0, 1]$ interval and integer values. Finally, it could be interesting to cope with other FSA-SPARQL statements. More specifically, fuzzy aggregators and quantifiers [8].

CRediT authorship contribution statement

Jesús M. Almendros-Jiménez: Writing – review & editing, Writing – original draft, Methodology, Conceptualization. **Antonio Becerra-Terón:** Writing – review & editing, Writing – original draft, Validation, Methodology, Conceptualization. **Ginés Moreno:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **José A. Riaza:** Validation, Software, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data are available on links quoted in the paper

References

[1] S. Abbasbandy, M. Amirfakhrian, Numerical approximation of fuzzy functions by fuzzy polynomials, Appl. Math. Comput. 174 (2006) 1001–1006.
[2] B.S. Ahn, Parameterized OWA operator weights: an extreme point approach, Int. J. Approx. Reason. 51 (2010) 820–831.
[3] B.S. Ahn, A new approach to solve the constrained OWA aggregation problem, IEEE Trans. Fuzzy Syst. 25 (2016) 1231–1238.
[4] J.M. Almendros-Jiménez, A. Becerra-Terón, Discovery and diagnosis of wrong SPARQL queries with ontology and constraint reasoning, Expert Syst. Appl. 165 (2021) 113772.
[5] J.M. Almendros-Jiménez, A. Becerra-Terón, G. Moreno, A fuzzy extension of SPARQL based on fuzzy sets and aggregators, in: 2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), IEEE, 2017, pp. 1–6.
[6] J.M. Almendros-Jimenez, A. Becerra-Terón, G. Moreno, Fuzzy queries of social networks with FSA-SPARQL, Expert Syst. Appl. 113 (2018) 128–146.
[7] J.M. Almendros-Jiménez, A. Becerra-Terón, G. Moreno, J.A. Riaza, Tuning fuzzy SPARQL queries in a fuzzy logic programming environment, in: 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), IEEE, 2019, pp. 1–7.
[8] J.M. Almendros-Jiménez, A. Becerra-Terón, G. Moreno, J.A. Riaza, Flexible aggregation in FSA-SPARQL, in: 30th IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2021, Luxembourg, July 11-14, 2021, IEEE, 2021, pp. 1–7.
[9] G.R. Amin, A. Emrouznejad, Parametric aggregation in ordered weighted averaging, Int. J. Approx. Reason. 52 (2011) 819–827.
[10] A. Arslan, M. Kaya, Determination of fuzzy logic membership functions using genetic algorithms, Fuzzy Sets Syst. 118 (2001) 297–306.
[11] J. Cheng, Z. Ma, L. Yan, f-SPARQL: a flexible extension of SPARQL, in: International Conference on Database and Expert Systems Applications, Springer, 2010, pp. 487–494.
[12] I. Dellal, S. Jean, A. Hadjali, B. Chardin, M. Baron, On addressing the empty answer problem in uncertain knowledge bases, in: International Conference on Database and Expert Systems Applications, Springer, 2017, pp. 120–129.
[13] D. Dubois, H. Prade, On fuzzy interpolation, Int. J. Gen. Syst. 28 (1999) 103–114.
[14] P. D'Urso, Linear regression analysis for fuzzy/crisp input and fuzzy/crisp output data, Comput. Stat. Data Anal. 42 (2003) 47–72.
[15] G. Fokou, S. Jean, A. Hadjali, M. Baron, RDF query relaxation strategies based on failure causes, in: European Semantic Web Conference, Springer, 2016, pp. 439–454.
[16] R. Frosini, A. Poulovassilis, P.T. Wood, A. Calí, Optimisation techniques for flexible sparql queries, ACM Trans. Web 16 (2022) 1–44.
[17] X. Jian, Y. Wang, X. Lei, L. Zheng, L. Chen, Sparql rewriting: towards desired results, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 1979–1993.
[18] P. Julián-Iranzo, G. Moreno, J.A. Riaza, The fuzzy logic programming language FASILL: design and implementation, Int. J. Approx. Reason. 125 (2020) 139–168.
[19] O. Kaleva, Interpolation of fuzzy data, Fuzzy Sets Syst. 61 (1994) 63–70.
[20] G. Li, W. Li, H. Wang, Querying fuzzy RDF knowledge graphs data, in: 2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), IEEE, 2020, pp. 1–8.
[21] R. Lowen, A fuzzy Lagrange interpolation theorem, Fuzzy Sets Syst. 34 (1990) 33–38.
[22] R. Ma, X. Jia, J. Cheng, R.A. Angryk, SPARQL queries on RDF with fuzzy constraints and preferences, J. Intell. Fuzzy Syst. 30 (2016) 183–195.
[23] S.A. Moises, S.d.L. Pereira, Dealing with empty and overabundant answers to flexible queries, J. Data Anal. Inf. Process. 2 (2014) 12–18.
[24] G. Moreno, J. Penabad, J.A. Riaza, G. Vidal, Symbolic execution and thresholding for efficiently tuning fuzzy logic programs, in: Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Revised Selected Papers, 2016, pp. 131–147.
[25] G. Moreno, J.A. Riaza, An online tool for tuning fuzzy logic programs, in: Rules and Reasoning - International Joint Conference, RuleML+RR 2017, Proceedings, Springer, 2017, pp. 184–198.
[26] G. Moreno, J.A. Riaza, Symbolic similarity relations for tuning fully integrated fuzzy logic programs, in: Rules and Reasoning +- 4th International Joint Conference, Proceedings, RuleML+RR 2020, Oslo, Norway, June 29 - July 1, 2020, Springer, 2020, pp. 150–158.
[27] G. Moreno, J.A. Riaza, A safe and effective tuning technique for similarity-based fuzzy logic programs, in: I. Rojas, G. Joya, A. Català (Eds.), Advances in Computational Intelligence - 16th International Work-Conference on Artificial Neural Networks, Proceedings, Part I, IWANN 2021, Virtual Event, June 16-18, 2021, Springer, 2021, pp. 190–201.
[28] L. Parkin, B. Chardin, S. Jean, A. Hadjali, Explaining unexpected answers of sparql queries, in: International Conference on Web Information Systems Engineering, Springer, 2022, pp. 136–151.
[29] A.V. Patel, Transformation functions for trapezoidal membership functions, Int. J. Comput. Cogn. 2 (2004) 115–135.
[30] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Trans. Database Syst. 34 (2009) 16.
[31] I. Perfilieva, D. Dubois, H. Prade, F. Esteva, L. Godo, P. Hoďáková, Interpolation of fuzzy data: analytical approach and overview, in: Fuzzy Set Theory — Where do We Stand and Where do We Go?, Fuzzy Sets Syst. 192 (2012) 134–158.
[32] O. Pivert, O. Slama, V. Thion, An extension of SPARQL with fuzzy navigational capabilities for querying fuzzy RDF data, in: 2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), IEEE, 2016, pp. 2409–2416.
[33] O. Pivert, O. Slama, V. Thion, Fuzzy quantified queries to fuzzy RDF databases, in: 2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), 2017, pp. 1–7.
[34] X. Sang, X. Liu, An analytic approach to obtain the least square deviation OWA operator weights, Fuzzy Sets Syst. 240 (2014) 103–116.
[35] O. Slama, Personalized queries under a generalized user profile model based on fuzzy SPARQL preferences, in: 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), IEEE, 2019, pp. 1–6.

[36] Q. Song, M.H. Namaki, P. Lin, Y. Wu, Answering why-questions for subgraph queries, IEEE Trans. Knowl. Data Eng. 34 (2020) 4636–4649.
[37] E. Vasilyeva, M. Thiele, C. Bornhövd, W. Lehner, Answering "why empty?" and "why so many?" queries in graph databases, in: Special Issue on Query Answering on Graph-Structured Data, J. Comput. Syst. Sci. 82 (2016) 3–22.
[38] H. Wang, Z. Ma, J. Cheng, fp-SPARQL: an RDF fuzzy retrieval mechanism supporting user preference, in: 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), IEEE, 2012, pp. 443–447.
[39] M. Wang, J. Liu, B. Wei, S. Yao, H. Zeng, L. Shi, Answering why-not questions on sparql queries, Knowl. Inf. Syst. 58 (2019) 169–208.
[40] R.R. Yager, Weighted maximum entropy OWA aggregation with applications to decision making under risk, IEEE Trans. Syst. Man Cybern., Part A, Syst. Hum. 39 (2009) 555–564.
[41] L. Zadeh, Fuzzy sets, Inf. Control 8 (1965) 338–353.
[42] S. Zadrożny, J. Kacprzyk, Issues in the practical use of the OWA operators in fuzzy querying, J. Intell. Inf. Syst. 33 (2009) 307.