Neurocomputing 633 (2025) 129621

Contents lists available at ScienceDirect

Neurocomputing

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

Evolving J-CO-QL+ with fuzzy evaluators for flexible querying of JSON data sets

Paolo Fosci, Giuseppe Psaila *

University of Bergamo, Dept. of Management, Information and Production Engineering, Viale Marconi 5, Dalmine, 24044, (BG), Italy

ARTICLE INFO

Keywords:
Soft querying of JSON documents
Fuzzy evaluators for computing membership degrees
Meta-model for fuzzy evaluators
Novel statement
Practical case study

ABSTRACT

How to introduce soft querying based on fuzzy sets in the novel *J-CO-QL*$^+$ query language (specifically designed to query collections of JSON documents from NoSQL databases) has been investigated by the authors in their past work. Specifically, capabilities for defining fuzzy operators and fuzzy aggregators were introduced through two distinct concepts on which two different language constructs were based.

This paper proposes the unified concept of "fuzzy evaluator", by means of which it is possible to define complex methods for evaluating the membership degrees of JSON documents to fuzzy sets, so as to capture complex semantics while analyzing data in a soft way. The paper both provides a formal meta-model for fuzzy evaluators, and proposes a novel statement for the *J-CO-QL*$^+$ language, so as to further foster soft-querying capabilities.

1. Introduction

After their introduction by Zadeh in 1965 [1], Fuzzy Sets have been extensively investigated and exploited to solve many different problems. Indeed, their ability to deal with imprecise concepts and situations, due to the notion of "membership degree" to a (fuzzy) set (i.e., an item can belong to a set with partial extent), has made them an invaluable tool for many applications, such as decision making.

Selection conditions that are based on fuzzy sets are said "soft conditions", in contrast with "crisp conditions" that rely on classical Boolean logic: soft conditions allow for evaluating imprecise concepts, as well as linguistic concepts, over data. When analyzing (possibly large) data sets, they allow for ranking data items, based on the extent with which they satisfy the soft condition (see [2–4], in which fuzzy sets were exploited to outperform classification techniques in tasks of geo-data integration).

Consequently, it should not be surprising that, in the past literature, many researchers investigated how to extend languages for database querying (in particular, SQL) with soft conditions, so as to provide such capabilities directly on top of (relational) databases [5–7]. Unfortunately, those research works have not transitioned into real-life systems, because vendors of relational DBMSs (DataBase Management Systems) have not considered extending their SQL interpreters with soft capabilities. Recently, some revamping of interest in extending query languages with soft capabilities can be observed, in particular on top of NoSQL (Not only SQL) databases, such as *MongoDB*. Specifically, *MongoDB* is probably the most popular representative of a novel (if compared with relational-database technology) family of DBMSs that are capable of natively storing *JSON* documents; for this reason, they are also said "*JSON* Document Stores" or, simply, "*JSON* stores" [8].

The advent of Big Data and the progressive transformation of every scientific field towards "data-driven science", in which the *JSON* format has become the *de facto* standard for representing and sharing data sets, possibly stored in many different NoSQL DBMSs and possibly coming from a plethora of Web sources, has inspired the development of a novel framework at the University of Bergamo (Italy): this is the *J-CO* Framework. Its query language (named *J-CO-QL*$^+$) is not tied to any specific query language provided by a DBMS, because the *J-CO* Framework is designed to be independent of the particular storage (and its computational capabilities) from which data sets are retrieved; indeed, *J-CO-QL*$^+$ is specifically designed to provide analysts with limited skills in computer programming with a high-level query language for collections of *JSON* documents, which is able to natively deal with geographical representation of spatial entities [9–11]; furthermore, *J-CO-QL*$^+$ provides analysts with soft-querying capabilities, in that it is able to evaluate membership degrees of *JSON* documents to many fuzzy sets at the same time, so as to express complex soft conditions and perform non-trivial soft querying [12–14].

Recently, in [11,15], the authors started investigating how to deal with "user-defined fuzzy aggregators" in *J-CO-QL*$^+$. The language already provided the concept of "fuzzy operator", by means of which it

* Corresponding author.
E-mail address: giuseppe.psaila@unibg.it (G. Psaila).

https://doi.org/10.1016/j.neucom.2025.129621
Received 5 July 2024; Received in revised form 8 January 2025; Accepted 31 January 2025
Available online 12 February 2025
0925-2312/© 2025 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

was possible to evaluate the membership degree of a *JSON* document to a fuzzy set either by moving from its field values or by combining, in a complex way, membership degrees previously computed for other fuzzy sets (many works can be considered, in which the application of "user-defined fuzzy operators" is shown, such as [15,16]); the key statement through which *J-CO-QL*$^+$ supports the definition of fuzzy operators was named `CREATE FUZZY OPERATOR`. In contrast, *J-CO-QL*$^+$ lacked the capability of computing the membership degree of a document by aggregating either properties or memberships of documents that are nested within an array field of the processed *JSON* document: specifically, the work [15] proposes a meta-model for aggregating arrays of membership degrees, while the work [11] extends the approach to field values of document arrays; both works propose different versions of the novel statement named `CREATE FUZZY AGGREGATOR`.

The contribution of the present work is to unify the concepts of "fuzzy operator" and "fuzzy aggregator" into the unique and more general concept of "fuzzy evaluator": indeed, both fuzzy operators and fuzzy aggregators shared common features; by observing them from a novel perspective, the authors were inspired to unify fuzzy operators and fuzzy aggregators into one single concept. As a result, this paper presents a unified meta-model for "fuzzy evaluators"; then, the novel *J-CO-QL*$^+$ statement, which is named `CREATE FUZZY EVALUATOR`, is introduced; finally, a case study shows the practical exploitation of fuzzy evaluators on a real data set.

The remainder of the paper is organized as follows. Section 2 presents relevant related works, while Section 3 briefly introduces the *J-CO* Framework and its support to soft querying. Section 4 formally introduces the meta-model on which the concept of fuzzy evaluator relies; consequently, Section 5 is able to present the syntax of the novel statement `CREATE FUZZY EVALUATOR`, as well as its semantics in relation to the meta-model. Section 6 presents a real-life case study, which performs a soft query based on three different fuzzy evaluators within a *J-CO-QL*$^+$ query. Finally, Section 7 draws the conclusions and sketches possible future works.

2. Related work

The ancient origins of this work is the long research work that was made on defining query languages for soft querying on relational databases. For the interested reader, the topic was extensively presented in [12]. The remainder of this section focuses on recent works that are more strictly related to the contribution of the paper.

Languages for Querying *JSON* documents. NoSQL (i.e., Not-only SQL) databases are a reality in the current technological panorama. A very specific, yet widespread, category is the one known as "*JSON* document stores": they are DBMSs that natively store *JSON* documents in the databases. Probably, the most popular *JSON* store is *MongoDB*[1] [17]; its query language, now known as "*MQL*", queries collections of *JSON* documents, by exploiting both JavaScript and *JSON* documents with special fields to express queries, in a very unusual fashion.

Another important DBMS is *CouchDB*[2] [18], which has been chosen as the underlying DBMS for *HyperLedger Fabric*, the blockchain platform for realizing distributed and shared databases among information systems [19,20].

In effect, in the last 15 years many query languages for *JSON* data sets have been proposed. Just to remind the most known ones, they are *Jaql* [21], *JSONiq* [22] and *SQL++* [23]. As evidenced by its name, *SQL++* [23] is designed as an extension of the classical `SELECT` statement of SQL; indeed, it has been chosen as the query language for analytics in *CouchDB*, with the name of *N1QL for Analytics* [24]. However, the choice to keep the SQL syntax somehow forces users to view *JSON* documents as tables; it is the opinion of the authors of the present paper that this choice significantly limits the degree of flexibility that is offered by the language.

The common characteristic of the above-mentioned languages is that they do not support at all fuzzy sets, soft querying and fuzzy aggregations, while *J-CO-QL*$^+$ does; therefore, they are not comparable with *J-CO-QL*$^+$ as far as this topic is concerned. Indeed, when considering commercial systems, such as *MongoDB*, similar capabilities can be achieved by writing procedural JavaScript functions to perform complex aggregations and to incorporate some kind of "fuzziness". In contrast, *J-CO-QL*$^+$ explicitly supports fuzzy sets and provides high-level yet declarative constructs for evaluating membership degrees on *JSON* documents. To the best of the authors' knowledge, no similar approaches have been presented in the literature.

Fuzzy Querying on *JSON* document stores. Few papers that propose fuzzy extensions to query languages on JSON document stores are available. These papers are briefly summarized hereafter.

The first paper of interest is [25], in which the authors propose *fMQL*, an extension of *MQL* (i.e., the *MongoDB* query language) that relies on "fuzzy labels" to query *JSON* documents. Fuzzy labels are equivalent to linguistic predicates, so the query can be expressed on them. However, in [25] the authors do not touch the issue of associating fuzzy labels to *JSON* documents; consequently, their proposal is not complete and it is difficult to envision how to practically exploit it. In contrast, *J-CO-QL*$^+$ is a fully operational proposal, as demonstrated in previous works.

The second paper to mention is [26], in which the authors further carry out the development of *fMQL*: trying to provide the possibility to execute queries written in *fMQL*, they propose a translation technique of *fMQL* into *fXML*, a fuzzy language for querying XML documents. The positive aspect of [26] is that *fMQL* is presented in a more extensive way, even though the paper does not clarify how membership degrees are defined and managed. The negative aspect of [26] is that *fMQL* queries are translated into *fXML* queries, as well as *JSON* documents must be converted into XML documents, i.e., the original environment of *JSON* document stores is actually abandoned.

The third paper to mention is [27], which proposes an approach for soft querying *JSON* documents through "fuzzy RDF triples": *JSON* documents are preliminarily translated into fuzzy RDF triples, so they can be queried by means of *FSA-SPARQL*, a fuzzy extension to *SPARQL*. Clearly, this approach changes again the domain in which queries are executed, abandoning the world of *JSON* stores.

The final paper to mention is [28], in which the authors proposed a more holistic approach: they extend the *MongoDB* data model to support fuzzy values at the level of a single document field. In other words, the paper proposes a model for a "fuzzy *JSON* document store"; the authors further show that they implemented it on top of *MongoDB*, by developing JavaScript functions that are internally executed by *MongoDB*. Indeed, the proposal in [28] looks like the more advanced one among the few competitors of *J-CO-QL*$^+$; nonetheless, *J-CO-QL*$^+$ is quite rich and independent of any specific *JSON* store.

Fuzzy Aggregators. The present paper considers the problem of aggregating *JSON* documents so as to obtain membership degrees, in other words, it considers "fuzzy aggregators". This notion is not new in the literature and a plethora of papers were published in the past; consequently, hereafter it is possible to show only those papers that can be considered the "milestones" of this topic.

Many proposals on fuzzy aggregators, such as "t-norm" and "t-conorm" operators (see [29]), consider the aggregation of several fuzzy sets for one single item; for example, the classical `AND` and `OR` operators in the fuzzy version, actually evaluate if and how an item belongs to the intersection and union (respectively) of two fuzzy sets. In contrast, the orthogonal view is to consider groups (or categories) $G_j = \{x_{j,1}, x_{j,2}, \ldots\}$ of items $x_{j,i}$ that belong to the same group G_j because they share some common properties or are samples of the same category of items. With this vision, the membership degree of a group G_j to a fuzzy set A

[1] *MongoDB* website (on 25/06/2024): https://www.mongodb.com/.
[2] *CouchDB* website (on 25/06/2024): https://couchdb.apache.org/.

```
CREATE FUZZY OPERATOR Has_Increasing_GDP
  PARAMETERS   GDP1 TYPE Float,
               GDP2 TYPE Float
  PRECONDITION GDP1 > 0
           AND GDP2 > 0
  EVALUATE     100*(GDP2-GDP1) / GDP1
  POLYLINE     [ ( 0, 0.0), ( 2, 0.1), ( 4, 0.5),
                 ( 7, 0.8), (10, 0.9), ( 15, 1.0) ];
```

(a) Example definition of fuzzy operator (from [14]).

```
CREATE FUZZY AGGREGATOR owaRain
  PARAMETERS   rainData TYPE ARRAY
  SORT         rd IN rainData
    BY         rd TYPE NUMERIC ASC AS sRainData
  FOR ALL      srd IN sRainData
    LOCALLY    ( POS^2 - (POS-1)^2 )
                 / (COUNT(sRainData)^2) AS w
    AGGREGATE  srd * w AS av
  EVALUATE     av
  POLYLINE     [(0.00, 0.0), (0.10, 0.0), (0.15, 0.7),
                (0.20, 0.8), (0.50, 0.9), (0.80, 1.0)];
```

(b) Example definition of fuzzy aggregator (from [11]).

Fig. 1. Example definitions of fuzzy operator and fuzzy aggregator.

may be derived by somehow aggregating properties of items $x_{j,i} \in G_j$, i.e., either register properties or membership degrees to other fuzzy sets.

Popular fuzzy aggregators of this type are "Weighted aggregation" (see [30]) and "Ordered Weighted Aggregation" (OWA) [31,32]. How to model such aggregators in *J-CO-QL*$^+$ was shown in [11,15]; the present paper considers, as an example, an even more complicated, yet suitable, form of fuzzy aggregation, to show how the novel concept of "fuzzy evaluator" encompasses various interpretations of fuzzy aggregation in a unique semantic model.

3. The *J-CO* Framework

The *J-CO* Framework is a platform-independent framework specifically designed to provide analysts with a powerful tool by means of which it is possible to integrate, transform, and retrieve *JSON* documents from possibly large *JSON* data sets.

The core item of the framework is its query language, named *J-CO-QL*$^+$. It provides high-level constructs for managing complex document structures in a declarative way; since *JSON* documents can represent spatial entities that are provided with a geometrical representation (geo-tagging), the query language natively deals with it. Furthermore, it supports the evaluation of membership degrees of *JSON* documents to fuzzy sets, so as to let users perform soft queries on the basis of linguistic predicates. Both geo-tagging and fuzzy sets can be exploited together, to perform soft-spatial queries [14]. In the present paper, geo-tagging is not addressed.

The *J-CO* Framework encompasses several tools. This paper is not focused on the architecture of the framework (the interested reader can refer to many previous papers by the authors, such as [9,10,13]). Hereafter, a very short description of the main components is provided, for the sake of clarity.

- Data sets are stored as collections of *JSON* documents in NoSQL *JSON* stores managed, e.g., by *MongoDB*. Furthermore, data sets can be acquired from Web sources, such as Open-Data portals [10].
- A *J-CO-QL*$^+$ script (also named *J-CO-QL*$^+$ query) is processed by the *J-CO-QL*$^+$ *Engine*: this tool actually provides the framework with the necessary computational capability; consequently, the *J-CO-QL*$^+$ *Engine* retrieves data sets from Web sources and from *JSON* stores, then it saves results into *JSON* stores.
- A dedicated user interface, named *J-CO-UI*, as well as procedural programs written in Java (but this is only a temporary limitation) can interact with the *J-CO-QL*$^+$ *Engine* and submit *J-CO-QL*$^+$ scripts.

3.1. Fuzzy sets in J-CO-QL$^+$

As introduced before, *J-CO-QL*$^+$ has been designed for analysts with limited programming skills, as well as to help skilled analysts devise complex analysis tasks, by relying on a clear semantic model. The execution model of *J-CO-QL*$^+$ scripts will be sketched in Section 6; the present section shortly presents how fuzzy sets are supported by *J-CO-QL*$^+$.

Before the paper continues, it is worth briefly introducing very basic notions on fuzzy sets. In a fuzzy set S, an item $i \in S$ is provided with a "membership degree" $\mu_S(i) \in [0, 1]$, which denotes the extent to which the item belongs to the fuzzy set. As an effect, the same item belongs to the complement set $\overline{S}$ with membership degree $1 - \mu_S(i)$.

Similarly, given two fuzzy sets A and B of items in the same universe U, an item i belongs to $A \cap B$ with membership degree $min(\mu_A(i), \mu_B(i))$, while i belongs to $A \cup B$ with membership degree $max(\mu_A(i), \mu_B(i))$. The classical logical operators `NOT`, `AND` and `OR` are extended by associating them to fuzzy complement, fuzzy intersection and fuzzy union, respectively.

Now, it is possible to summarize how *J-CO-QL*$^+$ deals with fuzzy sets.

- *J-CO-QL*$^+$ works on "collections of *JSON* documents"; consequently, the basic representational item is a *JSON* document.
- When the membership degrees of a *JSON* document to fuzzy sets are known, the document contains the special field named `~fuzzysets`. It behaves as a key–value map, where the field name denotes the name of the fuzzy set, while the numerical value (in the range $[0, 1]$) denotes the membership degree, i.e., the degree with which the item represented by the document belongs to the fuzzy set. In [33], an extension towards "multi-grade fuzzy sets" is proposed; however, the present paper is limited to the basic model proposed by Zadeh in [1].
- A "soft condition" evaluates the membership degree of a *JSON* document to a specific fuzzy set. In a soft condition, terms are fuzzy-set names (previously evaluated), fuzzy operators and fuzzy aggregators, possibly connected by the classical logical operators (i.e., `AND`, `OR`, `NOT`) in the generalized fuzzy version.
  For example, if documents represent people, for which the extent to the fuzzy sets `RichPeople` and `PopularPeople` are known, a soft condition could be
  `RichPeople AND PopularPeople`
  meaning that the user is interested in evaluating how a person is rich and popular at the same time.

3.2. User-defined fuzzy operators and aggregators

In the previous version of *J-CO-QL*$^+$, two distinct constructs were available to evaluate membership degrees of *JSON* documents on the basis of their properties. These constructs were named "fuzzy operators" and"fuzzy aggregators". Indeed, at the beginning (see [11,15]), fuzzy aggregators were conceived as an independent concept. They are shortly introduced hereafter.

- Fig. 1(a) shows the definition of a fuzzy operator named `Has_Increasing_GDP`, taken from [4].
  In short, the clause `PARAMETERS` defines input formal parameters for the operator (they are the values used to evaluate the membership degree), while the optional clause `PRECONDITION` specifies a crisp condition to be satisfied by actual parameters, to proceed with evaluation. The clause `EVALUATE` evaluates a

value e on the basis of the actual parameter values: the resulting value e is matched against the membership function defined by the clause `POLYLINE`, in such a way e is used as x-axis value, so as the corresponding y-axis value (in the range $[0, 1]$) is returned as the computed membership degree.

- Fig. 1(b) shows the definition of a fuzzy aggregator named `owaRain`, taken from [11].
  In short, the clause `PARAMETERS` defines the input formal parameters, in which the first one must be an array. The optional clause `SORT` sorts the input array parameter, generating a new internal array. This array is scanned by the optional clause `FOR ALL`, which, for each item managed through the `srd` iterator, for each item it computes a local value named `w` and an expression (in the clause `AGGREGATE`) whose values are aggregated by summing them into the novel variable `av`. Finally, the clauses `EVALUATE` and `POLYLINE` behave in the same way as for fuzzy operators: the value e generated by the clause `EVALUATE` is the x-axis value of the membership function defined by the clause `POLYLINE`, such that the corresponding y-axis value is returned as the computed membership degree.

The reader can notice that, when removing the optional clauses `SORT` and `FOR ALL`, a fuzzy aggregator is reduced to a fuzzy operator. Thus, it is possible to argue that a fuzzy operator is a particular case of a fuzzy aggregator; nonetheless, it is possible to argue that a fuzzy aggregator is an extension of a fuzzy operator. Consequently, to improve the language, the two concepts had to be unified.

In the remainder of this paper, the two concepts are fused and extended into the concept of "fuzzy evaluator". Section 4 presents the formal meta-model; Section 5 presents the syntax of the novel *J-CO-QL*$^+$ statement, while Section 6 shows its application to a case study.

4. Meta-model for fuzzy evaluators

This section presents the first contribution of this paper, i.e., a meta-model for fuzzy evaluators. Section 4.1 defines the basic structure of a fuzzy evaluator; Section 4.2 defines a variety of functions that were encompassed to provide fuzzy evaluators with powerful capabilities.

4.1. Basic structure

The first concept to be introduced is the concept of "*context*", hereafter defined.

Definition 1 (*Context*)**.** A "context" C is a key/value map $C[n] \rightarrow v$ where "n" is a name (the key) and "v" is the associated value.

The context is a container of named values, either received as input parameters by the fuzzy evaluator or explicitly defined within it; for simplicity, they can be referred to as "variables". Intuitively, a fuzzy evaluator derives a new context at each internal step (see later).

A second basic concept is that of "Polyline Membership Function".

Definition 2 (*Polyline Membership Function*)**.** A "Polyline Membership Function" is a sequence $[p_1, p_2, \ldots, p_n]$ of points "p_i", with $1 \leq i \leq n$. In turn, $p_i = (x_i, y_i)$ denotes a point on the Cartesian plane, such that $x_i > x_{i-1}$ (for $i > 1$) and $0 \leq y_i \leq 1$.

Thus, a polyline membership function is defined as a sequence of consecutive (left-to-right) points with respect to the x-axis; since it should provide a membership degree, the value of the y-axis coordinates is limited in the range $[0, 1]$.

Definition 3 (*Derivation Sequence*)**.** A "Derivation Sequence" DS is a sequence $DS = (s_1, \ldots, s_n)$ where "s_i" (with $1 \leq i \leq n$ and $n \geq 1$), is named "derivation step" and is based on a "*context-derivation function*" $C_i = cdf(C_{i-1})$ (see Section 4.2).

The rationale is that, through a derivation sequence, it is possible to specify complex computations, in which several intermediate results must be computed.

Definition 4 (*Derivation-Sequence Executor*)**.** Consider a context C_0 and a derivation sequence DS. The "Derivation-Sequence Executor" $DS\mathcal{E}$ is a function $DS\mathcal{E}(C_0, DS) \rightarrow C$ which generates a novel context C moving from the input context C_0.

Specifically, it behaves as follows: $\forall_{i=1}^{n}\ C_i = C_{i-1} \cup s_i(C_{i-1})$ where $n = |DS|$, $s_i \in DS$ and $C = C_n$.

The Derivation-Sequence Executor will be actually implemented in the execution engine.

Definitely, a derivation sequence DS is the basic brick for defining fuzzy evaluators, as defined hereafter.

Definition 5 (*Fuzzy Evaluator*)**.** A "Fuzzy Evaluator" is specified through a "fuzzy-evaluator definition" Φ, defined as a tuple $\Phi = \langle \mathcal{P}, precond, DS, eval, mFunction \rangle$ where "$\mathcal{P}$" is a context containing input parameters; "$precond$" is a Boolean condition expressed on $\mathcal{P}$; "DS" is a derivation sequence (see Definition 3); "$eval$" is a mathematical expression to evaluate on the results of DS; "$mFunction$" is a polyline membership function (see Definition 2).

The semantics of a fuzzy evaluator is explained by defining the concept of "Fuzzy-Evaluator Executor".

Definition 6 (*Fuzzy-Evaluator Executor*)**.** Given a Fuzzy-Evaluator Φ, the "Fuzzy-Evaluator Executor" $\mathcal{FEE}$ is a function $\mathcal{FEE}(C_0, \Phi) \rightarrow \mu$ that, given an input context C_0 of actual input parameters and a Fuzzy Evaluator Φ, returns a membership degree $\mu \in [0, 1]$.

Specifically, it behaves as follows:

$$\mu = polyline(\Phi.mFunction, \Phi.eval(DS\mathcal{E}(\Phi.\mathcal{P}, \Phi.DS))).$$

The rationale is the following. The execution of the derivation sequence $\Phi.DS$ (performed by the Derivation-Sequence Executor $DS\mathcal{E}$, see Definition 4), moving from the input context $\Phi.\mathcal{P}$ (i.e., the actual input parameters), generates a new context that is exploited to evaluate the mathematical expression $\Phi.eval(C) \rightarrow \mathcal{R}$. The real value

$v = \Phi.eval(DS\mathcal{E}(\Phi.\mathcal{P}, \Phi.DS))$ that is returned by this mathematical expression $\Phi.eval$ is evaluated against the polyline membership function $\Phi.mFunction$ (see Definition 2); v is the x-axis value, while μ is the corresponding y-axis value.

4.2. Context-derivation functions

Once the general model is defined, it is necessary to define context-derivation functions that are used within the Derivation Sequence $\Phi.DS$ (see Definition 3).

4.2.1. Derived values

The first derivation function is the one that allows for deriving values to add to the context.

Definition 7 (*Pool of Derived-Value Definitions*)**.** A "Pool of Derived-Value Definitions" is a sequence $\Delta = (dvd_1, \ldots, dvd_r)$ where each dvd_i (with $1 \leq i \leq r$ and $r \geq 0$) is a pair $\langle dvn, df \rangle$. Specifically, "dvn" is said "derived-value name" and is a string that denotes the name of the derived value; "df" is said "derivation function" and is a function $df(C)$ that, given a context C, returns a value for "dvn".

The reader can notice that Δ could be empty ($r \geq 0$).

Through Δ, it is possible to derive values that will be useful for defining complex fuzzy evaluators.

Definition 8 (*Derived-Value Generator*). Given a pool Δ of Derived-Value Definitions, the "Derived-Value Generator" DVG is a function $DVG(C_0, \Delta) \rightarrow C$ that, given a context C_0 and a pool Δ of derived-value definitions, generates a new context C obtained by extending C_0 with derived values defined by Δ.

Specifically, $\forall_{i=1}^{|\Delta|}(C_i = \{\langle dvd_i.dvn, dvd_i.df(C_{i-1})\rangle\} \cup C_{i-1})$ i.e., given $dvd_i \in \Delta$, a derived value with name $dvd_i.dvn$ is derived by means of the derivation function $dvd_i.df$ and is added to the context. Notice that $C = C_{|\Delta|}$ (this way, in the case Δ is empty, $C = C_0$).

A step $s_i \in DS$ (see Definition 3) could be a pool Δ of derived-value definitions. This way, it is possible to compute values to be later used for evaluating the final membership degree, by exploiting the Derived-Value Generator DVG. Indeed, contexts accumulate values step by step.

4.2.2. Sorting multiple arrays

There is no doubt that while working with arrays, it could be necessary to sort them. For this reason, the meta-model for fuzzy evaluators encompasses a function for generating sorted arrays. However, a complex approach to array-sorting is considered, in which multiple arrays could be sorted at once, as they were one unique array.

Definition 9 (*Array-Sorting Definition*). An "Array-Sorting Definition" is a tuple $\Omega = \langle \mathcal{AS}, SC\rangle$ where $\mathcal{AS} = (as_1, \ldots, as_u)$ is a list of "array specifications" as_i (with $1 \leq i \leq u$ and $u \geq 1$), while $SC = (sc_1, \ldots, sc_c)$ is a list of "sorting criteria" sc_j (with $1 \leq j \leq c$ and $c \geq 1$).

An array specification $as \in \mathcal{AS}$ is, in turn, a tuple $as = \langle uName, iName, oName\rangle$, where "$uName$" is the name of an "unsorted array" actually present in the input context, "$iName$" is the name of an "iterator" that will be used to manage the array, "$oName$" is the name of the "output sorted array".

Finally, a "sorting criterion" $sc \in SC$ is a pair $sc = \langle key, direction\rangle$, where "$key$" is the "sorting key" (that is a dot expression rooted in an iterator defined in $\mathcal{AS}$), while "$direction$" can be either "ASC" (ascending) or "DESC" (descending).

As for derived values, the function that actually generates sorted arrays is the "Sorted-Array Generator".

Definition 10 (*Sorted-Array Generator*). Given an "Array-Sorting Definition" Ω, the Sorted-Array Generator" is a function $\mathcal{SAG}(C_0, \Omega) \rightarrow C$ where C_0 is the input context, while C is the output context.

The function accumulates all values in the input arrays specified in $\Omega.\mathcal{AS}$ into an "array of tuples" $AT = [t_1, \ldots, t_k]$, where k is the maximum size of the input arrays. Each tuple $t_i \in AT$ has the fields $t_i = \langle \Omega.\mathcal{AS}[1].iName, \ldots, \Omega.\mathcal{AS}[u].iName\rangle$ so that a tuple collects the values that are in the same position in the input arrays; in the case the position does not exist in the array, the null value is assigned to the corresponding field in the tuple.

Based on the sorting criteria $\Omega.SC$, the sorted version $\overline{AT}$ of the array of tuples AT is generated. The novel output context is generated as follows:

$$C = C_0 \cup (\cup_{j=1}^{|\Omega.\mathcal{AS}|}(\{\langle \Omega.\mathcal{AS}[j].oName, \pi_{\Omega.\mathcal{AS}[j].iName}(\overline{AT})\rangle\}))$$

where the operator π builds an array by projecting the sorted array $\overline{AT}$ on the specified field $\Omega.\mathcal{AS}[].iName$.

The reader can see that the sorted-array generator is able to manage several arrays as they were one single array of tuples. This is an important feature that could be significant in performing complex analysis.

A step $s_i \in DS$ (see Definition 3) could be an array-sorting definition Ω. This way, it is possible to compute novel sorted arrays to be added to the internal context of the fuzzy evaluator; these arrays could be later aggregated to compute the final membership degree.

4.2.3. Aggregated values

The last, but not least, type of derivation functions that is encompassed within fuzzy evaluators allows for defining (and computing) aggregated values by exploiting array input parameters and/or sorted arrays.

Definition 11 (*Pool of Aggregated-Value Definitions*). Given an array $\mathcal{A}$ of "values to aggregate" and a context C, a "Pool of Aggregated-Value Definitions" is defined as a non-empty list $\Theta = (avd_1, \ldots, avd_t)$ where "avd_i" (with $1 \leq i \leq t$ and $t \geq 1$) is said "aggregation definition" and is, in turn, a pair

$avd_i = \langle avn, laf(C), agOp\rangle$.

Specifically, "avn" is the "aggregated-value name", i.e., a string that denotes the resulting aggregated value. Then, "laf" is the "local aggregation function": it is a mathematical expression to be computed for each single item in $\mathcal{A}$ (the array of values to aggregate), whose value is in the input context C; returned values are expected to be later aggregated by means of "$agOp$", i.e., an "aggregation operator" (see Definition 13); the usual aggregation operators can be considered, such as sum, average, minimum and maximum values.

In a general view of aggregation within fuzzy evaluators, it is not strange thinking about the need for computing multiple aggregated values from one single array (see, for example, the paper [11]). This is why the Pool of Aggregated-Value Definitions was conceived. Nonetheless, this concept must be encompassed within the concept of "Array-Aggregator Definition".

Definition 12 (*Array-Aggregator Definition*). An "Array-Aggregator Definition" is a tuple$\Xi = \langle \Theta, \Delta, aName, iName\rangle$ where Θ is a pool of aggregated-value definitions, Δ is a pool of derived-value definitions, "$aName$" and "$iName$" are, respectively, an array name and an iterator name.

The idea is that Ξ specifies an array (by means of its name $aName$) that is in the context of the fuzzy evaluator, whose values must be aggregated through Θ and Δ, by exploiting an iterator (whose name is $iName$) that is exploited in each single aggregated-value definition $avd_i \in \Theta$, i.e., in $avd_i.laf$. In Ξ, Δ specifies how to derive "local values", i.e., derived values that are locally derived for each single item, so as to be exploited in the local aggregation functions $avd_i.laf$ (for each $avd_i \in \Theta$).

A user-defined context variable (with name $iName$, see Definition 12), together with another context variable named POS (that denotes the current position of the processed item), will play the role of iterator and will allow local-aggregation functions and local derivation functions to access the array item to process.

Definition 13 (*Aggregated-Value Generator*). Consider a context C_0 and an Array-Aggregator Definition Ξ. The "Aggregated-Value Generator" $\mathcal{AVG}$ is a function that generates a new context C, i.e.,

$\mathcal{AVG}(C_0, \Xi) \rightarrow C$.

Given a context C_0, it generates a novel context with the aggregated values. Specifically. it is

$$C = C_0 \cup \cup_{i=1}^{|\Xi.\Theta|}\{\langle \Xi.\Theta[i].avn, (\Xi.\Theta[i].agOp)_{pos=1}^{|C_0[aName]|}\Xi.\Theta[i].laf(\overline{C})\rangle\}$$

where

$$\overline{C} = DVG(C_0 \cup \{(\text{"POS"}, pos)\rangle\} \cup \{\langle \Xi.iName, C_0[aName][pos]\rangle\}, \Xi.\Delta)$$

statementCFE →
CREATE FUZZY EVALUATOR *feName*
PARAMETERS *paramList*
[PRECONDITION *precondition*]
(*deriveClause* | *sortClause* | *forAllClause*)∗
EVALUATE *evaluateExpr*
[POLYLINE *pointSequence*]

sortClause →
SORT *iterList* BY *sortCriteria* AS *outList*

deriveClause →
DERIVE *deriveExpr* AS *varName*

forAllClause →
FOR ALL *iName* IN *aName*
[LOCALLY *deriveExpr* AS *varName*
(, *deriveExpr* AS *varName*)∗]
AGGREGATE *deriveExpr* AS *varName*
(, *deriveExpr* AS *varName*)∗

Fig. 2. Syntax of the statement CREATE FUZZY EVLUATOR.

In words, the Aggregated-Value Generator behaves as follows. For each aggregated-value definition $avd_i \in \Xi.\Theta$ (denoted as $\Xi.\Theta[i]$), it generates a new context variable with name $\Xi.\Theta[i].avn$; the value is computed by aggregating (through the aggregation operator $\Xi.\Theta[i].agOp$) the values returned by the local aggregation function $\Xi.\Theta[i].laf$, to which the "local context" $\overline{C}$ is passed.

In turn, for each item in the array to aggregate, a specific version of the local context $\overline{C}$ is generated: first of all, a temporary context is derived from the global context C_0 by adding the variable POS (which denotes the current position in the array to aggregate) and the iterator $\Xi.iName$ (with the value of the current item in the array). This temporary context is the first input parameter for the Derived-Value Generator $\mathcal{DVG}$ (see Definition 8), which possibly adds locally-derived values as defined by $\Xi.\Delta$. This way, values to be aggregated can be computed in a way that is aware of the position and value of the current item, possibly exploiting temporary values that are derived for each single item to aggregate.

The reader can notice how the same definition of "derived values" Δ (through the concept of derivation sequence, see Definition 3) and the corresponding Derived-Value Generator (see Definition 8) can be used both at the global level (see Section 4.2.1) and locally within aggregators.

A step $s_i \in DS$ (see Definition 3) could be an array-aggregator definition Ξ. This way, it is possible to compute novel aggregated values to be added to the internal context of the fuzzy evaluator, to be later used for evaluating the final membership degree, by exploiting the Aggregated-Value Generator $\mathcal{AVG}$.

5. A novel statement for *J-CO-QL*$^+$

The second contribution of the paper is the novel *J-CO-QL*$^+$ statement CREATE FUZZY EVALUATOR. This section shows the statement syntax, in relation to the meta-model that was introduced in Section 4, while the practical use of the statement will be presented in Section 6.3.

Fig. 2 reports the general structure of the syntax of the novel statement; it relies on the EBNF format (Extended BNF).

The main symbol is *statementCFE*. It defines the main structure of the evaluator; consequently, its clauses correspond to the fields in the formal definition Φ of fuzzy evaluator, reported in Definition 5.

The first one is the mandatory clause PARAMETERS, which defines the list of formal parameters (for simplicity, the symbol *paramList* is not further defined). Clearly, the clause corresponds to the field $\Phi.\mathcal{P}$ in the formal definition of fuzzy evaluator (see Definition 5).

The second clause is the optional clause PRECONDITION, which defines the crisp condition to be met (if present) to proceed with the evaluation (for simplicity, the symbol *precondition* is not further defined, but it is a classical Boolean condition). Clearly, the clause corresponds to the field $\Phi.precond$ in the formal definition of fuzzy evaluator (Definition 5).

A possibly-empty sequence of clauses (within the expression $(\dots)*$) follows: these clauses are defined by the symbols *deriveClause*, *sortClause*, and *forAllClause*: indeed, this block corresponds to the "Derivation-Sequence Definition" $\Phi.DS$ in the formal definition of the Fuzzy Evaluator (see Definition 5).

The clause EVALUATE defines a mathematical expression that provides the value e by means of which the membership degree is obtained. This clause corresponds to the field $\Phi.eval$ in the formal definition of fuzzy evaluator (see Definition 5). For simplicity, the symbol *evaluateExpr* is not further defined, but it is a classical mathematical expression.

The final clause in the statement is the optional clause POLYLINE. It defines a sequence of points, such as
`[(0,0),(1,0.1),(2, 0.8)]`
i.e., the membership function $\Phi.mFunction$ in the formal definition of the fuzzy evaluator (see Definition 5).

The strict correspondence between the clauses in the statement and the formal definition Φ enables a clear understanding of how the defined evaluator works. Consequently, executing a fuzzy evaluator within the *J-CO-QL*$^+$ *Engine* means implementing the function Fuzzy-Evaluator Executor $\mathcal{FEE}$ presented in Definition 6.

The three clauses that derive variables (corresponding to the Derivation-Sequence Definition $\Phi.DS$ in Definition 5) can be now considered.

The clause DERIVE is defined by the symbol *deriveClause*. It is quite simple: a mathematical expression (symbol `deriveExpr`) computes a value that is assigned to the novel context variable whose name is specified by the symbol *varName* after the keyword AS.

Multiple consecutive occurrences of the clause can be assimilated into a unique derivation sequence; executing it means implementing the function Derived-Value Generator $\mathcal{DVG}$ presented in Definition 8.

The clause SORT is defined by the symbol *sortClause*. It corresponds to the Array-Sorting Definition Ω presented in Definition 9.

The symbol *iterList* specifies the iterator name *iName* for each array *aName* to sort. The form is `(i1 IN a1, i2 IN a2, i3 IN a3)` where, for example, `i1` is the iterator that works on the array `a1`.

The symbol *criteria* defines the list of sorting criteria. It appears like `i1 TYPE NUMERIC ASC, i2 TYPE NUMERIC DESC` in which the meaning of keywords is straightforward.

Finally, the symbol *outList* specifies the list of sorted-array names to add to the context. It appears like `(o1, o2, o3)` (generic output names are used in this example).

Clearly, executing the SORT clause means implementing the Sorted-Array Generator function $\mathcal{SAG}$, presented in Definition 10.

The last clause to introduce is the clause FOR ALL, which is defined by the symbol *forAllClause*.

After the keyword FOR ALL, an iterator *iName* on an array *aName* is specified.

The optional clause LOCALLY allows for specifying locally-derived variables, for each single item in the array. This corresponds to the field $\Xi.\Delta$ in the Array-Aggregator Definition Ξ, presented in Definition 12. Indeed, the LOCALLY keyword is followed by a non-empty list of variable derivations, whose syntax is the same as for the clause DERIVE (when the clause is missing, this corresponds to the empty sequence).

Listing 1 *J-CO-QL*$^+$ Script: Defining fuzzy evaluator `hasIncreasingGdpFE`.

```
1. CREATE FUZZY EVALUATOR hasIncreasingGdpFE
     PARAMETERS      gdp1   TYPE NUMERIC,
                     gdp0   TYPE NUMERIC
     PRECONDITION    gdp1>0 AND gdp0>0
     EVALUATE        100*(gdp1-gdp0)/gdp0
     POLYLINE        [ (0.0, 0.0), ( 4.0, 0.2),
                       (6.0, 0.8), (10.0, 1.0) ];
```

The mandatory sub-clause `AGGREGATE` specifies a non-empty list of aggregated values to compute. Again, the same syntax used to define derived variables is used, but this time the symbol *varName* denotes a variable, whose value is obtained by aggregating the values obtained for the *deriveExpr* for each single item.

Notice that the sub-clause `AGGREGATE` corresponds to the field $\Xi.\Theta$ in the Array-Aggregator Definition Ξ, presented in Definition 12. Clearly, executing the clause `FOR ALL` means implementing the function Aggregated-Value Generator $\mathcal{AVG}$ presented in Definition 13.

6. Case study

This section shows how to use a *J-CO-QL*$^+$ script with fuzzy evaluators for soft querying a data set that describes the GDP (Gross Domestic Product) of countries in the European Union, which is available on the *Eurostat* Web portal.[3]

First, the premises of the case study are presented. Then, the source data set is shown. Next, the *J-CO-QL*$^+$ fuzzy evaluators that were defined to analyze the source data are presented. Finally, the *J-CO-QL*$^+$ script that actually allows for performing the analysis is explained.

6.1. Problem definition

The case study analyzes the population and GDP of the whole European Union by considering a time window of ten years, in order to determine, for each year, if there have been increases in GDP along with improvements in equality (the paper calls it *fairness*) of the distribution of the level of income among citizens in the European Union. The problem can be formally expressed as a soft query, as clarified by Problem 1.

Problem 1. Given data on population and GDP for a certain year, determine if there was a significant increment of GDP and equality in the distribution of the income among citizens in the European Union.

Notice the soft concept of "significant increment" that can be considered a "linguistic predicate" to evaluate through membership degrees to fuzzy sets.

To solve Problem 1, the *Gini Index*[4] [34] is exploited: by means of data related to population and GDP, the Gini Index measures the inequality of the distribution of the level of income. The Gini Index varies between 0 and 1: when it is 0, it means that the income is perfectly distributed among citizens (full equality); when it is 1, it means that only one person holds all the income and all the other citizens hold nothing (full inequality).

Important disclaimer: the authors of this paper are not economists but computer scientists; consequently, both political and economical considerations are outside the scope of this paper. Moreover, the important simplification that, in each country, the income is perfectly distributed with the GDP per capita is assumed.

[3] Eurostat website (on 25/06/2024): https://ec.europa.eu/eurostat.

[4] Gini Index (on 25/06/2024): https://en.wikipedia.org/wiki/Gini_coefficient.

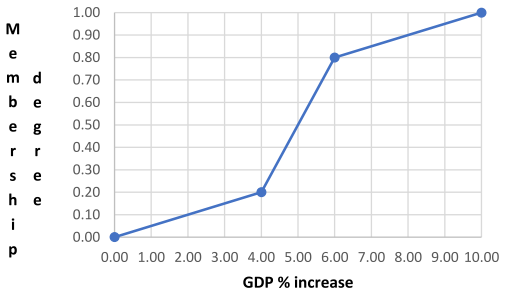


Fig. 3. Polyline of the fuzzy evaluator `hasIncreasingGdpFE`.

6.2. Data set

Data about population,[5] GDP and GDP per capita,[6] in the period from 2014 to 2023, about the 27 countries in the European Union, were downloaded from the Eurostat Web portal. With this data set, a *JSON* collection of 270 documents (like the document shown in Fig. 6) was built and subsequently stored into a *MongoDB* database. As can be seen in Fig. 6, each document reports the reference year (field: `year`), the name of the country (field: `country`), the population of the country (field: `population`), the GDP of the country (in million Euro, field: `gdp`), and the GDP per capita of the country (in Euro, field: `gdpPc`) in the reference year.

6.3. Fuzzy evaluators in J-CO-QL$^+$

Now, the three fuzzy evaluators that were defined to analyze the source data can be presented.

The *J-CO-QL*$^+$ fuzzy evaluator `hasIncreasingGdpFE` is reported in Listing 1. The reader can easily notice that this fuzzy evaluator is formally equivalent to the former fuzzy operator `Has_Increasing_GDP` previously shown in Fig. 1(a). Indeed, the structure of a fuzzy evaluator actually encompasses former fuzzy operators.

Ignoring the naming differences, the fuzzy evaluator, as the former fuzzy operator reported in Fig. 1(a), has two formal numerical input parameters named (in this case) `gdp1` and `gdp0`, which represent, respectively, the GDP of a certain year and of the previous year.

The clause `PRECONDITION` specifies the same crisp condition as in the former fuzzy operator reported in Fig. 1(a), which has to be satisfied by the actual parameters, to proceed with the evaluation.

The clause `EVALUATE` computes a value on the basis of the actual parameter values: this is the percentage variation in GDP. The resulting variation is matched against the membership function that is defined by the clause `POLYLINE` (shown in Fig. 3), so as to return the computed membership degree. In this fuzzy evaluator, the polyline function differs from the one defined in the former fuzzy operator reported in Fig. 1(a), so as to meet the specific needs of the current case study.

Listing 2 reports the fuzzy evaluator `fairnessFE`, which, in short, evaluates the decrement in the Gini Index over two consecutive years.

The fuzzy evaluator has two formal parameters named `gini1` and `gini0`, which, respectively, represent the Gini Index of a certain year and of the previous year. In this case, there is no need to match any precondition to proceed with the evaluation; thus, the clause `PRECONDITION` is missing.

[5] Eurostat - population data (on 25/06/2024): https://ec.europa.eu/eurostat/databrowser/view/demo_pjan/default/table?lang=en&category=demo.demo_pop.

[6] Eurostat - GDP and GDP per capita data (on 25/06/2024): https://ec.europa.eu/eurostat/databrowser/view/tec00001/default/table?lang=en&category=t_na10.t_nama10.t_nama_10_ma.

Listing 2 *J-CO-QL*$^+$ Script: Defining fuzzy evaluator `fairnessFE`.

```
2. CREATE FUZZY EVALUATOR fairnessFE
     PARAMETERS gini1 TYPE NUMERIC,
                gini0 TYPE NUMERIC
     DERIVE     gini1-gini0 AS dg
     EVALUATE   -dg
     POLYLINE   [ (0.00, 0.0), (1.00, 0.0),
                  (1.01, 0.6), (1.10, 0.9),
                  (2.00, 1.0) ];
```

Unlike former fuzzy operators, a fuzzy evaluator can exploit the clause DERIVE to calculate intermediate results to be used subsequently: in this case, the first clause DERIVE calculates the variation in the Gini Index; the resulting value is assigned to the variable dg that, in turn, is added to the context (see Section 4.2).

The subsequent clause EVALUATE computes the decrement of the Gini Index and matches it against the membership function defined by the clause POLYLINE (shown in Fig. 4) to return the computed membership degree.

Listing 3 reports a more sophisticated fuzzy evaluator, named giniIndexFE, that behaves like a former fuzzy aggregator. In short, given population and GDP data, the fuzzy evaluator giniIndexFE computes the Gini Index.

In details, the fuzzy evaluator has five formal parameters named, respectively, popTot (which represents the total population), gdpTot (which represents the total GDP), pop (which is a numeric array containing the population for each considered country), gdp (which is a numeric array containing the GDP for each considered country), and gdpPc (which is a numeric array containing the GDP per capita for each considered country).

The subsequent clause PRECONDITION checks whether the three input arrays have all the same size: indeed, each position in the three input arrays is related to the same country, thus they must have the same size.

The clause SORT considers the three input arrays gdpPc, gdp and pop, as they were one single array: indeed, since a given position refers to a specific country, they must be sorted all together. Three iterators are defined: they are named, respectively, i1, i2 and i3. Then, the keyword BY specifies the sorting criterion by which the three arrays must be ordered: in this case, this is the ascending order of the iterator i1 (in other words, the three arrays are sorted according to the ascending order of the gdpPc array, as they were a single array). Finally, the keyword AS specifies that the three novel sorted arrays named, respectively, sGdpPc, sGdp, and sPop are generated and added to the context, in addition to the source gdpPc, gdp, and pop arrays that remain unchanged.

The subsequent clause DERIVE generates a novel array, named cPop, starting from the array sPop: given an input numerical array, the novel built-in function CUMULATE computes a cumulative array. The second clause DERIVE behaves exactly like the first one: it generates the cGdp array by accumulating the sGdp array. Consequently, two new arrays are added to the context of the fuzzy evaluator. Fig. 5(a) shows a table with the content of the arrays sPop, sGdp, sGdpPc, cPop and cGdp (each one in the homonymous column), with data from the 27 countries of the European Union in 2023. Notice that the column (in light blue) sGdpPc is in ascending order, based on the sorting criterion that was specified in the clause SORT (the other arrays were sorted accordingly). In addition, the table contains two extra yellow columns: they can be considered two extra "virtual" arrays, named cPop% and cGdp%; specifically, cPop% is obtained by normalizing each item of the array cPop with the total population; cGdp% is obtained by normalizing each item of the array cGdp with the total GDP. The virtual arrays cPop% and cGdp% are needed to

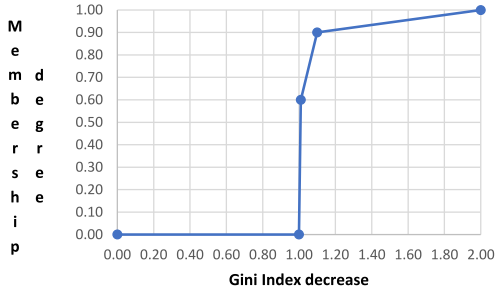


Fig. 4. Polyline of the fuzzy evaluator fairnessFE.

define the "Lorenz curve"[7] [35], which constitutes the basis to compute the Gini Index. The Lorenz curve is defined by plotting the array cPop% on the x-axis and the array cGdp% on the y-axis. Fig. 5(b) shows the Lorenz curve (in blue) calculated with the data that are presented in the table of Fig. 5(a). The Gini Index is the ratio of the area between the so-called "line of equality" (shown with a dashed-green line in Fig. 5(b)) and the Lorenz curve, and the total area under the "line of equality".

Going back to the fuzzy evaluator, the subsequent clause FOR ALL computes the area under the Lorenz curve as the sum of the subtended areas of the trapezoids between two consecutive points of the Lorenz curve and the x-axis. Specifically, an iterator p is defined, to consider each item of the array cPop. The sub-clause LOCALLY calculates, for each value of p, two local values named dp and dg. The value dp is the difference between the current item p and the previous one normalized to the total population popTot: notice the special symbol POS that denotes the actual position of the item p (that could be also accessed with the heavier notation cPop[POS]); moreover, since the first item of the array cPop has no previous item by definition, the POS-1 expression would generate an out-of-range index for the array cPop: thus the built-in function IF_ERROR is needed to return the default value of 0 in case of whatsoever error generated during the evaluation of the expression. Then, the value dg (i.e., the second value computed by the sub-clause LOCALLY) is the normalized mean between two consecutive items in the array cGdp: the corresponding item of the array cGdp, related to the item p in the array cPop, is accessed through the symbol POS . Finally, the sub-clause AGGREGATE computes the value lca, which is the area of the Lorenz curve, by summing the product of the values dp and dg calculated for each item p.

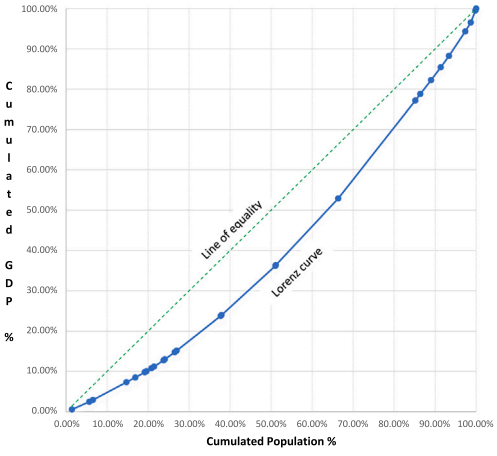
At the end of the fuzzy evaluator, the clause EVALUATE computes the Gini Index, by using the value lca. In this case, the clause POLYLINE is missing, assuming that the default polyline [(0,0), (1,1)] is used to return the membership degree.

The reader can notice how the semantic richness of the novel concept of fuzzy evaluator actually allows users to specify very complex functions to be used as the membership degree to fuzzy sets. In comparison with the former fuzzy operators and fuzzy aggregators, it is possible to highlight that: (i) the former fuzzy operator is based only on the clause EVALUATE and cannot receive arrays, while a fuzzy evaluator can receive array parameters and can derive intermediate results (by means of the clause DERIVE); (ii) the former fuzzy aggregator can receive only one single array parameter, as well as it cannot manage several arrays in the clause SORT and also it does not support many clauses FOR ALL, while a fuzzy evaluator can. Furthermore, the declarative approach is maintained, which is one distinctive feature of *J-CO-QL*$^+$.

[7] Lorenz curve (on 25/06/2024): https://en.wikipedia.org/wiki/Lorenz_curve.

#	sPop	sGdp	sGdpPc	cPop	cGdp	cPop%	cGdp%
1	6447710	93948	14580	6447710	93948	1.44%	0.55%
2	19054548	324578	17030	25502258	418526	5.68%	2.47%
3	3850894	76472	19858	29353152	494998	6.54%	2.92%
4	36753736	750801	19920	66106888	1245799	14.73%	7.34%
5	9599744	196391	20480	75706632	1442190	16.87%	8.50%
6	10413982	220303	20920	86120614	1662493	19.19%	9.80%
7	1883008	40348	21440	88003622	1702841	19.61%	10.04%
8	5428792	122813	22210	93432414	1825653	20.82%	10.76%
9	2857279	71986	25070	96289693	1897640	21.46%	11.19%
10	10467366	265508	25730	106757059	2163148	23.79%	12.75%
11	1365884	37682	27590	108122943	2200830	24.09%	12.98%
12	10827529	305967	28580	118950472	2506796	26.51%	14.78%
13	2116972	63090	29750	121067444	2569886	26.98%	15.15%
14	48085361	1461889	30320	169152805	4031775	37.69%	23.77%
15	920701	29807	32150	170073506	4061582	37.90%	23.95%
16	58997201	2085376	35350	229070707	6146958	51.05%	36.24%
17	542051	19418	35400	229612758	6166376	51.17%	36.35%
18	68172977	2803100	40800	297785735	8969476	66.36%	52.88%
19	84358845	4122210	48770	382144580	13091686	85.16%	77.18%
20	5563970	274880	49280	387708550	13366566	86.40%	78.80%
21	11742796	584699	49720	399451346	13951265	89.01%	82.25%
22	10521556	540652	51000	409972902	14491916	91.36%	85.44%
23	9104772	478190	52370	419077674	14970106	93.39%	88.26%
24	17811291	1034086	57840	436888965	16004192	97.36%	94.35%
25	5932654	373755	62840	442821619	16377947	98.68%	96.56%
26	5271395	504620	95290	448093014	16882567	99.85%	99.53%
27	660809	79310	118770	448753823	16961876	100.00%	100.00%
Tot	448753823	16961876					

(a) `GiniIndexFE` internal data arrays.



(b) Lorenz curve.

Fig. 5. Example of internal data arrays generated by the fuzzy evaluator `GiniIndexFE` with European Union data related to 2023, and the relative Lorenz curve.

Listing 3 *J-CO-QL*$^+$ Script: Defining fuzzy evaluator `giniIndexFE` for calculating Gini Index.

```
3. CREATE FUZZY EVALUATOR giniIndexFE
     PARAMETERS    popTot TYPE NUMERIC,
                   gdpTot TYPE NUMERIC,
                   pop    TYPE ARRAY,
                   gdp    TYPE ARRAY,
                   gdpPc  TYPE ARRAY
     PRECONDITION COUNT(pop) = COUNT(gdp)
              AND COUNT(gdp) = COUNT(gdpPc)
     SORT         (i1 IN gdpPc, i2 IN gdp, i3 IN pop)
              BY  i1 TYPE NUMERIC ASC
              AS  (sGdpPc, sGdp, sPop)
     DERIVE       CUMULATE (sPop) AS cPop
     DERIVE       CUMULATE (sGdp) AS cGdp
     FOR ALL      p IN cPop
       LOCALLY
         (p-IF_ERROR(cPop[POS-1], 0))/popTot  AS dp,
         (cGdp[POS]+IF_ERROR(cGdp[POS-1], 0))
                                  /(2*gdpTot)  AS dg
       AGGREGATE dp*dg AS lca
     EVALUATE 2*(0.5-lca);
```

6.4. Soft querying with J-CO-QL$^+$

This section presents the *J-CO-QL*$^+$ script that was written for soft querying the source data set, on the basis of Problem 1. The three previously-presented fuzzy evaluators, i.e., `hasIncreasingGdpFE`, `fairnessFE` and `giniIndexFE`, are integral parts of the script; for this reason, in Listing 1, Listing 2, and Listing 3, they are labeled as instructions number 1, 2 and 3, respectively. Consequently, the rest of the script starts from the instruction number 4. For editorial purposes, the script is split in two parts, reported in Listing 4 and in Listing 5. The remainder of this section explains the script.

The instruction `USE DB`, on Line 4 in Listing 5, specifies the database to connect with. Specifically, the database is named `Neurocomputing2024` and is managed by a *MongoDB* server.

The instruction `GET COLLECTION`, on Line 5 in Listing 5, actually retrieves the content of the source collection named `EUData` from the database `Neurocomputing2024`; the collection contains the 270 *JSON* documents representing data from the 27 countries of the European Union in the decade 2014-2023.

An example of the *JSON* documents in the collection `EUData` is shown in Fig. 6 and is presented earlier in Section 6.2.

```
{
  "year"       : 2023,
  "country"    : "Austria",
  "gdp"        : 478190.1,
  "gdpPc"      : 52370,
  "population" : 9104772
}
```

Fig. 6. Example of document in the source *JSON* collection.

Listing 4 *J-CO-QL*$^+$ Script: part 1.

```
4. USE DB Neurocomputing2024
    ON SERVER MongoDB 'http://127.0.0.1:27017';

5. GET COLLECTION EUData@Neurocomputing2024;

6. GROUP
    PARTITION WITH .year
      BY     .year
      INTO   .euData
      DROP GROUPING FIELDS
    GENERATE BUILD {
      .year           : .year,
      .euData         : .euData,
      .euPopulation   : SUM_ARRAY (.euData,
                          NUMERIC, .population ),
      .euGdp          : SUM_ARRAY (.euData,
                          NUMERIC, .gdp),
      .euPopulationArray : EXTRACT_ARRAY (
                            .population
                            FROM ARRAY .euData ),
      .euGdpArray     : EXTRACT_ARRAY ( .gdp
                            FROM ARRAY .euData ),
      .euGdpPcArray   : EXTRACT_ARRAY ( .gdpPc
                            FROM ARRAY .euData )
    };
```

The role of the instruction `GROUP`, on Line 6 in Listing 7, is grouping together the documents of the same year that describe data of the 27 countries. Hereafter, it is explained in detail.

The clause `PARTITION` selects all documents that have a value for the field `year`, because the subsequent sub-clause `BY` groups together those documents having the same value for the field `year`: for all documents in the same group, one single novel document is created, with the field `year` at the root-level; furthermore, as specified by the keyword `INTO`, an array field named `euData`, which contains all source documents that belong to the group, is also added at root level. The option `DROP GROUPING FIELDS`, removes the grouping field `year` from all sub-documents in the array field `euData`.

Listing 5 *J-CO-QL*$^+$ Script: part 2.

```
7.  FILTER
     CASE WHERE WITH .year
       GENERATE
         CHECK FOR
           FUZZY SET inequality
             USING giniIndexFE ( .euPopulation,
                  .euGdp, .euPopulationArray,
                  .euGdpArray, .euGdpPcArray );

8.  JOIN OF COLLECTIONS  temporary AS t1,
                         temporary AS t0
     SET FUZZY SETS
       LEFT  inequality   AS newInequality,
       RIGHT inequality   AS oldInequality
     CASE WHERE .t1.year = .t0.year+1
       GENERATE
         CHECK FOR
           FUZZY SET fairness
             USING fairnessFE (
                      EXTENT (newInequality),
                      EXTENT (oldInequality)  ),
           FUZZY SET increasingGDP
             USING hasIncreasingGdpFE (
                      .t1.euGdp, .t0.euGdp ),
           FUZZY SET goodYear
             USING fairness AND increasingGDP
         ALPHA CUT 0.10 ON goodYear
         BUILD {
           .year      : .t1.year,
           .fairness  : EXTENT (fairness),
           .incGdp    : EXTENT (increasingGDP),
           .goodYear  : EXTENT (goodYear)
         }
         DEFUZZIFY;

9.  SAVE AS EUDataAnalyzed@Neurocomputing2024;
```

Going on with the instruction, in the following section GENERATE, the block BUILD further enriches each novel document. Beyond the fields year and euData, which are left unchanged, other fields are added at root level:

- the field euPopulation exploits the built-in function SUM_ARRAY to sum all the fields population in each sub-document contained in the array field euData;
- in a similar way, the value of the novel field euGdp is calculated by applying the built-in function named SUM_ARRAY over the field gdp contained in the sub-documents in the array field euData;
- the field euPopulationArray is actually a plain array field that is built by using the built-in function EXTRACT_ARRAY, that creates a novel array field of numbers by projecting the array field euData on the inner (numerical) field population;
- similarly, the array fields euGdpArray and euGdpPcArray are built by projecting the array field euData on the inner fields gdp and gdpPc, respectively.

Notice that all the three array fields euGdpArray, euGdpPcArray, and euPopulationArray are built according to the order of the sub-documents in the array field euData, thus, the above-mentioned array fields maintain the same order of sub-documents.

Since the source data set covers a ten-year time window, the output collection generated by the instruction GROUP contains exactly 10 documents, one for each year.

Fig. 7 shows an example of the documents that are created by the instruction GROUP: notice the array field euData, which contains an array of 27 sub-documents where, in each of them, the field year is missing. Notice also the content of the array fields euPopulationArray, euGdpArray and euGdpPcArray, which are filled with values taken with the same order of the sub-documents in the array field euData.

```
{
  "year"          : 2023,
  "euGdp"         : 16961876.4,
  "euPopulation"  : 448753823.0,
  "euData"              : [
    { "country"       : "Austria",
      "gdp"           : 478190.1,
      "gdpPc"         : 52370,
      "population"    : 9104772    },
    // … others 26 EU country sub-docs
  ],
  "euGdpArray"        : [
    478190.1,  // … others 26 EU country-GDPs
  ],
  "euGdpPcArray"      : [
    52370,     // … others 26 EU country GDPpCs
  ],
  "euPopulationArray" : [
    9104772, // … others 26 EU country pop
  ],
}
```

Fig. 7. Example of document after instruction GROUP.

```
{
  "year"          : 2023,
  "euGdp"         : 16961876.4,
  "euPopulation"  : 448753823.0,
  "euData"              : [
    { "country"       : "Austria",
      "gdp"           : 478190.1,
      "gdpPc"         : 52370,
      "population"    : 9104772    },
    // … others 26 EU country sub-docs
  ],
  "euGdpArray"        : [
    478190.1,  // … others 26 EU country-GDPs
  ],
  "euGdpPcArray"      : [
    52370,     // … others 26 EU country GDPpCs
  ],
  "euPopulationArray" : [
    9104772, // … others 26 EU country pop
  ],
  "~fuzzysets"    : {
    "inequality"      : 0.200430708685564
  }
}
```

Fig. 8. Example of document after instruction FILTER.

The role of the instruction FILTER, on Line 7 in Listing 7, is to "fuzzify" each document generated by the previous instruction GROUP, by computing the membership to fuzzy sets. Hereafter it is explained in detail.

The condition CASE WHERE selects all those documents that hold the field year at root level.

For each selected document, in the following section GENERATE, the clause CHECK FOR fuzzifies the document by adding, at root level, the special field ~fuzzysets. Then, the branch FUZZY SET evaluates the membership to the fuzzy set named inequality, and the keyword USING specifies that the membership degree is calculated using the fuzzy evaluator giniIndexFE, defined in Listing 2, to which the fields euPopulation, euGdp, euPopulationArray, euGdpArray,euGdpPcArray are passed as actual parameters. The membership to the fuzzy set inequality is stored within the document by adding the homonymous field inequality inside the special field ~fuzzysets.

Fig. 8 shows an example of document in the temporary collection generated by the execution of the instruction FILTER on Line 7. Notice the presence of the special field ~fuzzysets, highlighted in a red box, holding inside the field inequality.

The instruction JOIN OF COLLECTIONS, on Line 8 in Listing 5, generates novel documents, by associating each document in the current temporary collection, generated by the instruction FILTER, with

each other documents in the same temporary collection. Hereafter, the instruction JOIN OF COLLECTIONS is explained in detail.

For each couple $(t1, t0)$ of documents of the temporary collection, a novel document is generated, having at root level two fields, named t1 and t0, which hold, respectively, the $t1$ document and the $t0$ document.

The clause SET FUZZY SETS fuzzifies the novel document by adding, at root level, the special field ~fuzzysets to hold memberships to fuzzy sets. The keyword LEFT sets the membership to the fuzzy set newInequality with the membership to the fuzzy set inequality of the document $t1$. The keyword RIGHT sets the membership to the fuzzy set oldInequality with the membership to the fuzzy set inequality of the document $t0$.

The following condition CASE WHERE, selects only those documents where the field year in the $t1$ document is immediately following the field year in the $t0$ document.

In the next section GENERATE, the clause CHECK FOR evaluates the membership to three novel fuzzy sets:

- the first branch FUZZY SET evaluates the membership to the fuzzy set fairness by using the fuzzy evaluator fairnessFE (defined in Listing 2) to which the memberships (also said "extent") to the fuzzy sets newInequality and oldInequality are passed as actual parameters by means of the built-in function EXTENT;
- the second branch FUZZY SET evaluates the membership to the fuzzy set increasingGDP by using the fuzzy evaluator hasIncreasingGdpFE defined in Listing 1, to which the values of the fields .t1.euGdp and .t0.euGdp are passed as actual parameters;
- finally, the third branch FUZZY SET evaluates the membership to the fuzzy set goodYear, where the membership degree is computed with a combination of the memberships to the fuzzy sets fairness and increasingGDP.

Notice that the expression used for evaluating the membership to the fuzzy set goodYear addresses Problem 1.

The following clause ALPHA CUT discards the document from the current collection if the membership to the fuzzy set goodYear is less than the minimum threshold of 0.1.

Then, the block BUILD restructures and flattens the document; specifically, the final document contains, at root level, the field year with the reference year, the field fairness, with the membership to the fuzzy set fairness, the field incGdp, with the membership to the fuzzy set increasingGDP, and the field goodYear, with the membership to the fuzzy set goodYear. The source fields t1 and t0 are discarded.

The last clause DEFUZZIFY "defuzzifies" the document, by removing the special field ~fuzzysets, since the block BUILD does not affect it.

Fig. 9 shows an example of a document that is generated by the instruction JOIN OF COLLECTIONS before the clause DEFUZZIFY. Fig. 10 shows the same document after the clause DEFUZZIFY.

Finally, the instruction SAVE AS, on Line 9 in Listing 9, saves the resulting collection into the database Neurocomputing2024 with the name EUDataAnalyzed.

Fig. 11 shows a tabular representation of the content of the final collection ordered according to decreasing values of the field goodYear. Notice that only those documents with the value of field goodYear over the alpha-cut value of 0.1 are actually present in the output collection, while, in Fig. 11, the lines in red represent documents that are discarded from the final collection.


```
{
  "year"        : 2023,
  "fairness"    : 0.619153244240333,
  "goodYear"    : 0.619153244240333,
  "incGdp"      : 0.830193611755728,
  "~fuzzysets" : {
    "fairness"        : 0.619153244240333
    "goodYear"        : 0.619153244240333
    "increasingGDP"   : 0.830193611755728
    "newInequality"   : 0.200430708685564
    "oldInequality"   : 0.216176675604292
  }
}
```


Fig. 9. Example of temporary document during the execution of instruction JOIN OF COLLECTIONS before the clause DEFUZZIFY.


```
{
  "year"        : 2023,
  "fairness"    : 0.619153244240333,
  "goodYear"    : 0.619153244240333,
  "incGdp"      : 0.830193611755728,
}
```


Fig. 10. Example of document after instruction JOIN OF COLLECTIONS.

year	goodYear	fairness	incGdp
2023	0.62	0.62	0.83
2022	0.52	0.52	0.93
2021	0.28	0.28	0.93
2017	0.25	0.30	0.25
2019	0.18	0.22	0.18
2018	0.18	0.32	0.18
2016	*0.08*	*0.08*	*0.14*
2015	*0.01*	*0.01*	*0.18*
2020	*0.00*	*0.00*	*0.00*

Fig. 11. Ranking table.

6.5. Discussion

The reader can notice the usefulness of the approach based on fuzzy evaluators. Indeed, even though they could be complex, such as the evaluator giniIndexFE, the computed membership degrees to fuzzy sets constitute the basis for reasoning in linguistic terms on data.

Indeed, looking at the last FUZZY SET branch in the instruction on Line 8 of Listing 8, i.e., FUZZY SET goodyear
USING fairness AND increasingGDP
it relies on the fact that the document belongs to the fuzzy sets named fairness and increasingGDP: this way, it is possible to capture the requirement (from Problem 1) that the analyst is interested in those years (represented by documents) in which there was an improvement of both fairness and GDP.

It is possible to say that, when compared with traditional analytics activities that are performed by writing procedural code written in low-level programming languages, the advantage of using *J-CO-QL*$^+$ scripts is to focus directly on the semantic level of the analysis.

As a practical example, it is possible to consider the case of *MongoDB* and its query language (MQL). If native MQL capabilities are not sufficient for performing queries based on complex analytics, it is always possible to write stored procedures, implemented using the JavaScript procedural programming language. Consequently, the following critical issues emerge:

- JavaScript is a complex procedural programming language and requires good programming skills to be effectively exploited;
- it is necessary to know the interface between a JavaScript function and the *MongoDB* execution engine in order to actually perform complex queries from within a JavaScript function;

- it is quite hard to debug the JavaScript code when it is executed within *MongoDB*;
- JavaScript stored procedures are hidden within the *MongoDB* database.

On the basis of these considerations, it is easy to understand the advantages provided by *J-CO-QL*$^+$ and, in particular, by fuzzy evaluators:

- *J-CO-QL*$^+$ is a declarative query language, which is independent of any specific storage system; as an effect of that, *J-CO-QL*$^+$ scripts can be executed on *JSON* data sets coming from any kind of source;
- fuzzy evaluators provide a clear semantics for how to evaluate complex aggregation functions on *JSON* documents, so no sophisticated programming skills are required; furthermore, being expressed in a declarative way, a fuzzy evaluator allows the analyst to concentrate on the actual computation to perform (while a procedural programming language would require focusing on low-level details of procedural instructions);
- in conclusion, in terms of efficiency (as far as developing the analysis task is concerned), writing fuzzy evaluators and a soft query in *J-CO-QL*$^+$ is faster than writing a corresponding JavaScript procedural code, thus obtaining the desired result in a quicker and easier way for analysts.

7. Conclusions and future work

It is time to conclude the paper and sketch future work.

7.1. Conclusions

The paper presented the most recent evolution of the *J-CO-QL*$^+$, i.e., the introduction of the concept of "fuzzy evaluator".

Before summarizing the contribution of the paper, it is worth recalling that the development of the *J-CO* Framework is part of a long-term research project, whose goal is to overcome limitations of standard/industrial solutions for managing large collections of *JSON* documents. Consequently, it should not be surprising that the syntax and statements in *J-CO-QL*$^+$, the query language of the framework, are continuously modified and extended.

This happened with the introduction of the statement `CREATE FUZZY EVALUATOR`, i.e., one of the contributions of the paper. Indeed, introducing support for fuzzy-set evaluation over *JSON* documents and soft querying them has been a process that has taken several steps.

- First of all, the notion of "fuzzy operator" and the corresponding statement `CREATE FUZZY OPERATOR` were introduced [36]. This concept was the first milestone, and the name "operator" was chosen thinking about traditional "t-norm"/"t-conorm" operators. Fuzzy operators are the first tool to evaluate membership degrees.
- The need for deriving membership degrees from arrays of *JSON* documents, as well as from arrays of simple values and of membership degrees [11,15], inspired the notion of "fuzzy aggregator" and the introduction of the statement `CREATE FUZZY AGGREGATOR`, so as to allow analysts to specify user-defined fuzzy aggregators. This step was the second milestone.
- The practical experience writing *J-CO-QL*$^+$ scripts suggested that the two concepts were strongly integrated and interconnected. Consequently, their fusion into the unified concept of "fuzzy evaluator" was straightforward. The choice of the term "evaluator" has the following motivation: their role is to "evaluate" membership degrees moving from field-values in *JSON* documents. Both the terms "operator" and "aggregator" were focused on a specific role; the term "evaluator" is more general and better focuses on the actual role played in the language. The novel statement `CREATE FUZZY EVALUATOR` in practice replaces the previous ones; however, for backward compatibility, the parser of *J-CO-QL*$^+$ still recognizes the old statements, by mapping them onto the novel one.

Then, it is now possible to briefly summarize the contribution of the paper. First of all, the paper presented the general context and, specifically, the basic features of the *J-CO* Framework and of *J-CO-QL*$^+$, in particular as far as soft querying is concerned. Then, Section 4 presented the meta-model for "fuzzy evaluators". Section 5 introduced the syntax of the novel statement `CREATE FUZZY EVALUATOR`, which lets users define user-defined fuzzy evaluators. Section 6 presented a real-life case study, so as to show the definition of fuzzy evaluators and their practical exploitation; the reader could see that the problem was solved with a *J-CO-QL*$^+$ script that contains a limited number of declarative instructions, in which *JSON* documents are transformed and queried through fuzzy sets. Definitely, the level of support for soft querying that is provided by *J-CO-QL*$^+$ is very high and makes the *J-CO* Framework a tool with unique capabilities.

Indeed, as argued in Section 3.2, the former fuzzy operators and fuzzy aggregators were strictly related to each other. Their unification into the unique and more general concept of fuzzy evaluator has significantly improved the clarity of the language, which has become significantly more synthetic and more expressive.

7.2. Future work

The first future work following this paper, is completing the possibility to define extended models for fuzzy sets. This research line was presented in the paper [33], but an important piece of the vision is missing, i.e., defining complex operators on such complex models. Consequently, the definition of fuzzy-set models will be completed with dedicated fuzzy evaluators.

A second important research line, which is already ongoing, is the identification of specific application contexts and the definition of domain-specific languages that are translated into *J-CO-QL*$^+$ scripts and executed by the *J-CO-QL*$^+$ *Engine*. The first result in this research line is *GeoSoft*, a language for querying *GeoJSON* documents. The interested reader can refer to [16,37].

In general, the authors will explore the use of the *J-CO* Framework in several application contexts, in which information represented as *JSON* documents must be managed in a complex way.

The *J-CO* Framework is available on a public GitHub repository.[8]

CRediT authorship contribution statement

Paolo Fosci: Writing – original draft, Software, Methodology, Data curation, Conceptualization. **Giuseppe Psaila:** Writing – original draft, Methodology, Investigation.

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.

Acknowledgments

This study was funded by the European Union - *NextGenerationEU, in the framework of the GRINS - Growing Resilient, INclusive and Sustainable project* (GRINS PE00000018 – CUP F83C22001720001). The views and opinions expressed are solely those of the authors and do not necessarily reflect those of the European Union, nor can the European Union be held responsible for them.

[8] GitHub repository of the *J-CO* Framework: https://github.com/JcoProjectTeam/JcoProjectPage.

Data availability

Data are publicly available on Eurostat web site, URL provided in the paper.

References

[1] L.A. Zadeh, Fuzzy sets, Inf. Control 8 (3) (1965) 338–353.
[2] M. Toccu, G. Psaila, D. Altomare, On-line aggregation of POIs from google and facebook, in: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, 2019, pp. 1081–1089.
[3] G. Psaila, M. Toccu, A fuzzy technique for on-line aggregation of POIs from social media: Definition and comparison with off-line random-forest classifiers, Information 10 (12) (2019) 388.
[4] P. Fosci, G. Psaila, Soft integration of geo-tagged data sets in J-CO-QL+, ISPRS Int. J. Geo- Inf. 11 (9) (2022) 484.
[5] P. Bosc, O. Pivert, SQLf: a relational database language for fuzzy querying, IEEE Trans. Fuzzy Syst. 3 (1) (1995) 1–17.
[6] A. Urrutia, L. Tineo, C. Gonzalez, FSQL and SQLf: Towards a standard in fuzzy databases, in: Handbook of Research on Fuzzy Information Processing in Databases, IGI Global, 2008, pp. 270–298.
[7] G. Bordogna, G. Psaila, Customizable flexible querying in classical relational databases, in: Handbook of Research on Fuzzy Information Processing in Databases, IGI Global, 2008, pp. 191–217.
[8] J. Han, E. Haihong, G. Le, J. Du, Survey on NoSQL database, in: 2011 6th International Conference on Pervasive Computing and Applications, IEEE, 2011, pp. 363–366.
[9] G. Psaila, P. Fosci, J-CO: A platform-independent framework for managing geo-referenced JSON data sets, Electronics 10 (5) (2021) 621.
[10] P. Fosci, G. Psaila, Towards soft web intelligence by collecting and processing JSON data sets from web sources, in: WEBIST, 2022, pp. 302–313.
[11] P. Fosci, G. Psaila, et al., Enhancing soft web intelligence with user-defined fuzzy aggregators, in: WEBIST, Vol. 1, Science and Technology Publications, Lda, 2023, pp. 258–267.
[12] P. Fosci, G. Psaila, Towards flexible retrieval, integration and analysis of JSON data sets through fuzzy sets: a case study, Information 12 (7) (2021) 258.
[13] P. Fosci, G. Psaila, J-CO, a framework for fuzzy querying collections of json documents, in: Flexible Query Answering Systems: 14th International Conference, FQAS 2021, Bratislava, Slovakia, September 19–24, 2021, Proceedings 14, Springer International Publishing, 2021, pp. 142–153.
[14] P. Fosci, G. Psaila, Soft spatial querying on json data sets, in: European Conference on Advances in Databases and Information Systems, Springer International Publishing Cham, 2022, pp. 223–237.
[15] P. Fosci, G. Psaila, Fuzzy aggregators in practice: Meta-model and implementation, in: International Conference on Soft Computing Models in Industrial and Environmental Applications, Springer Nature Switzerland Cham, 2023, pp. 56–68.
[16] P. Fosci, S. Marrara, G. Psaila, Soft querying geojson documents within the J-CO framework, in: 16th International Conference on Web Information Systems and Technologies, WEBIST 2020, SCITEPRESS–Science and Technology Publications, Lda, 2020, pp. 253–265.
[17] K. Chodorow, MongoDB: The Definitive Guide: Powerful and Scalable Data Storage, "O'Reilly Media, Inc.", 2013.
[18] J.C. Anderson, J. Lehnardt, N. Slater, CouchDB: The Definitive Guide: Time to Relax, "O'Reilly Media, Inc.", 2010.
[19] P.G. Bringas, I. Pastor, G. Psaila, Can BlockChain technology provide information systems with trusted database? The case of HyperLedger fabric, in: International Conference on Flexible Query Answering Systems, Springer, Cham, 2019, pp. 265–277.
[20] P. Garcia Bringas, I. Pastor-López, G. Psaila, BlockChain platforms in financial services: Current perspective, Bus. Syst. Res.: Int. J. Soc. Adv. Innov. Res. Econ. 11 (3) (2020) 110–126.
[21] A. Nayak, A. Poriya, D. Poojary, Type of NOSQL databases and its comparison with relational databases, Int. J. Appl. Inf. Syst. 5 (4) (2013) 16–19.
[22] D. Florescu, G. Fourny, JSONiq: The history of a query language, IEEE Internet Comput. 17 (5) (2013) 86–90.
[23] K.W. Ong, Y. Papakonstantinou, R. Vernoux, The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases, 2014, CoRR, abs/1405.3631.
[24] M.A. Hubail, A. Alsuliman, M. Blow, M. Carey, D. Lychagin, I. Maxon, T. Westmann, Couchbase analytics: NoETL for scalable NoSQL data analysis, Proc. VLDB Endow. 12 (12) (2019) 2275–2286.
[25] B.K. Abir, G.T. Amel, Towards fuzzy querying of NoSQL document-oriented databases, in: DBKDA 2015, 2015, p. 163.
[26] F. Mehrab, A. Harounabadi, Apply uncertainty in document-oriented database (MongoDB) using F-XML, J. Adv. Comput. Res. 9 (3) (2018) 87–101.
[27] J.M. Almendros-Jimenez, A. Becerra-Teron, G. Moreno, Fuzzy queries of social networks with FSA-SPARQL, Expert Syst. Appl. 113 (2018) 128–146.
[28] J.M. Medina, I.J. Blanco, O. Pons, A fuzzy database engine for mongoDB, Int. J. Intell. Syst. Online library (2022).
[29] E. Klement, R. Mesiar, E. Pap, Triangular norms, Tatra Mt. Math. Publ. 13 (1997) 169–193.
[30] J. Dombi, T. Jónás, Weighted aggregation systems and an expectation level-based weighting and scoring procedure, European J. Oper. Res. 299 (2) (2022) 580–588.
[31] R.R. Yager, On ordered weighted averaging aggregation operators in multicriteria decisionmaking, IEEE Trans. Syst. Man Cybern. 18 (1) (1988) 183–190.
[32] H. Li, V.C. Yen, Fuzzy Sets and Fuzzy Decision-Making, CRC Press, 1995.
[33] P. Fosci, G. Psaila, A unified view of multi-grade fuzzy-set models in J-CO-QL+, Neurocomputing 565 (2024) 126968.
[34] C. Gini, Variabilità e mutabilità (variability and mutability), vol. 156, Tipografia di Paolo Cuppini, Bologna, Italy, 1912.
[35] M.O. Lorenz, Methods of measuring the concentration of wealth, Publ. Am. Stat. Assoc. 9 (70) (1905) 209–219.
[36] G. Psaila, S. Marrara, A first step towards a fuzzy framework for analyzing collections of JSON documents, in: IADIS AC 2019, 2019, pp. 19–28.
[37] P. Fosci, G. Psaila, Soft querying features in GeoJSON documents: The GeoSoft proposal, Int. J. Comput. Intell. Syst. 16 (1) (2023) 163.