Soft Querying GeoJSON Documents within the J-CO Framework

Giuseppe Psaila[1][a], Stefania Marrara[2][b] and Paolo Fosci[1]

[1] *University of Bergamo, DIGIP, Viale Marconi 5, 24044 Dalmine (BG), Italy*

[2] *Cefriel, viale Sarca 226, 20126 Milano, Italy*

Keywords: GeoJSON Documents, Soft Querying of GeoJSON Documents, Fuzzy J-CO-QL Queries.

Abstract: GeoJSON documents have become important sources of information over the Web, because they describe geographical information layers. Supposing to have such documents stored in some JSON store, the problem of querying them in a flexible and easy way arises.

In this paper, we propose a soft-querying model to easily express queries on features (i.e., data items) within GeoJSON documents, based on linguistic predicates. These are fuzzy predicates that evaluate the membership degree to fuzzy sets; this way, imprecise conditions can be expressed and features can be ranked, accordingly. The paper presents a rewriting technique that translates soft queries on GeoJSON documents into fuzzy *J-CO-QL* queries: this is the query language of the *J-CO* Framework, an Internet-based framework able to get, manipulate and save collections of JSON documents in a way totally independent of the source JSON store.

1 INTRODUCTION

GeoJSON[1] is the standard format to represent geographical layers; today, it is widely used to share geographical information. Indeed, GeoJSON relies on the JSON[2] (JavaScript Object Notation) syntax, a simple and widely used standard that can be easily managed by any object-oriented programming language. As a consequence, GeoJSON documents are often found in open data portals, as well as GIS tools import and export export GeoJSON documents.

Based on this premise, the reader may think that managing and querying GeoJSON documents is easy, especially due to the availability of JSON document stores, i.e., database systems that are able to store JSON documents without any limitation. Unfortunately, this assumption is not true for several reasons: first of all, JSON document stores are designed to manage a multitude of small JSON documents, while GeoJSON documents may be very large; second, at the moment a standard query language for JSON documents does not exist, and each document store implements its own query language, forcing a user interested in several GeoJSON documents stored in different repositories to rewrite her/his queries in every specific query language; third, the query languages provided by all JSON document stores may be inadequate to effectively manage the typical nested structure of GeoJSON documents; finally, all the available languages are based on Boolean conditions that do not help to rank single features described within GeoJSON documents.

The problem of querying and manipulating JSON documents stored within different JSON document stores has been addressed in our previous work: the *J-CO* Framework (Bordogna et al., 2017; Bordogna et al., 2018) is the tentative answer. The framework is designed to be an *internet-based polystore framework*, able to connect to a multitude of heterogeneous JSON stores distributed over the Internet; it provides a query language, named *J-CO-QL*, that is independent of the specific JSON store and provides high-level constructs to manipulate heterogeneous collections of JSON documents (instructions can deal with many different document structures contained in the same collection at the same time).

With this framework, the reader may think that the problem of querying GeoJSON documents has been already solved. The truth is that this is "only partially" true: in fact, non-trivial *J-CO-QL* queries must be written to query GeoJSON documents. Furthermore, as far as soft querying GeoJSON documents is concerned, in principle the last extension to *J-CO-QL* proposed in (Psaila and Marrara, 2019), which

[a] https://orcid.org/0000-0002-9228-560X

[b] https://orcid.org/0000-0003-2745-3539

[1] https://geojson.org/

[2] https://www.w3schools.com/js/js_json_intro.asp

Psaila, G., Marrara, S. and Fosci, P.
Soft Querying GeoJSON Documents within the J-CO Framework.
DOI: 10.5220/0010155702530265
In *Proceedings of the 16th International Conference on Web Information Systems and Technologies (WEBIST 2020)*, pages 253-265
ISBN: 978-989-758-478-7

has introduced constructs to evaluate fuzzy sets over JSON documents, promises to provide this capability, but we will see that queries become harder to write. So, a higher-level approach is necessary, based on a simpler soft-querying model that is specific for GeoJSON documents, but indeed based on fuzzy *J-CO-QL* queries, which becomes the underlying engine.

In this paper, we present a high-level soft querying model for selecting features in GeoJSON documents, in such a way that, given an input GeoJSON document, the query generates a new GeoJSON document containing only the selected features (i.e., data items contained in the document). The adoption of linguistic predicates in the soft query provides the capability to rank features, on the basis of the membership degree to fuzzy sets. The paper will show how such "simple" soft queries can be automatically translated into complex fuzzy *J-CO-QL* queries, which actually disassemble the GeoJSON documents, apply fuzzy querying to single features and re-assemble them into a unique GeoJSON document, such that features are ordered in reverse order of importance with respect to the linguistic condition expressed in the soft query.

The remainder of the paper is organized as follows. Section 2 briefly discusses relevant related work. Section 3 presents the background of our work, i.e., basic notions on fuzzy sets and the GeoJSON format. Section 4 introduces the main research idea of the paper, i.e., applying soft querying to GeoJSON documents. Section 5 shows how the *J-CO-QL* language can manipulate GeoJSON documents to select features. Then, Section 6 shows how fuzzy concepts previously added to *J-CO-QL* in (Psaila and Marrara, 2019) can perform soft querying on GeoJSON documents (Sections 6.1 and 6.2), while a rewriting technique to derive fuzzy *J-CO-QL* queries on GeoJSON features is presented in Section 6.3. Finally, Section 7 concludes the paper.

2 RELATED WORK

Most relational database management systems adopt Boolean logic to formalize queries. This means that a query condition can be either satisfied or not satisfied. By using Boolean logic, it is not possible to have a flexible semantics of relational operations, to express preferences and to rank query results. In many real-world situations, queries are expressed by humans by means of imprecise words. User's intention is not merely to find the items that satisfy a given query; in contrast, the user may wish to estimate how much each item satisfies the conditions in the query (its satisfaction degree), in order to rank items (if possible).

There are several approaches to represent imprecise and vague concepts in Information Retrieval (IR). A first approach defines similarity or proximity relations between pairs of imprecise and vague items. In the Vector Space Model, for instance, documents and queries are represented as points in a space of terms and the distances between the points representing the query and the documents are used to quantify their similarity (Salton et al., 1994). Another category of approaches adopts the notion of Fuzzy Set. Fuzzy Set Theory is an extension to classical set theory (Zadeh, 1975). The notion of fuzzy set has been used to represent vague concepts expressed in a flexible query for specifying soft selection conditions (Blair, 1979). The objective here is to quantify the closeness of the information carried by the proposition with the considered reality. Possibility Theory (Fuhr, 1989; Zadeh, 1965) together with the concept of linguistic variable defined within fuzzy set theory (Zadeh, 1975), provides a complete formal framework to manage imprecise, vague and uncertain information (Buell, 1985).

There are two alternative ways to model the retrieval activity. (i) One possibility is to model the query evaluation mechanism as an uncertain decision process. The concept of relevance is defined as a binary (crisp) condition, since the query evaluation mechanism computes relevance probability of a document d with respect to a query q. Such an approach, which does model the uncertainty of the retrieval process, has been introduced and developed by using probabilistic IR models (Ramer, 1989; Herrera and Herrera-Viedma, 1997; Waller and Kraft, 1979). (ii) Another way is to interpret the query as the specification of soft constraints that the representation of a document can satisfy to a certain degree, and to consider term relevance as a gradual (vague) concept. This is the approach adopted in fuzzy IR models (Bordogna and Pasi, 1995). In this latter case, the decision process performed by the query evaluation mechanism computes the degree with which the query is satisfied by the representation of each document. A very good survey regarding the adoption of Fuzzy Sets in IR can be found in (Kraft et al., 2015).

A well defined context, in which dealing with uncertainty and vagueness is quite common, is XML Retrieval. When defining query languages for XML documents, the problem of querying data collections without a well-defined structure, or with a heterogeneous structure, was soon evident.

To tackle this problem, Fuzzy Set Theory was a fairly immediate choice. In (Damiani and Tanca, 2000) the authors presented an approach in which XML documents are modeled as labeled graphs; their structure is selectively extended by computing the importance of the information carried by tags as fuzzy estimates. The results of such an extension are fuzzy labeled graphs. Query results are sub-graphs of these fuzzy labeled graphs, presented as a ranked list according to their matching degree with respect to the query.

In a different perspective, the approaches presented in (Campi et al., 2009; Marrara and Pasi, 2016) propose flexible XML selection languages, which allow for formulating flexible constraints on both structure and content of XML documents. This way, users can formulate queries that are able to retrieve XML documents that provide information close to users' needs.

According to many research papers (Kacprzyk and Zadrożny, 1995; Medina et al., 1994) there are two main ways to apply Fuzzy Set Theory in the context of database management systems. The first way consists in developing a fuzzy database model to manage imprecise relational data (Bosc and Prade, 1997)

The second way develops a fuzzy querying interface on top of a conventional relational database: the advantage of this approach is evident in the SoftSQL proposal (Bordogna and Psaila, 2009). Other extensions of SQL have been proposed: FSQL (Galindo et al., 2006) and SQLf (Bosc and Pivert, 1995).

3 BACKGROUND

This section presents the relevant background on which the paper relies.

3.1 A Brief Introduction on Fuzzy Sets

Fuzzy Set Theory was introduced by Zadeh (Zadeh, 1975), and soon proved its potentiality, since it was successfully applied to different areas, such as artificial intelligence, natural language processing, decision making, expert systems, neural networks, control theory, and so on. Let us denote by X a non-empty universe, either finite or infinite.

Definition 1. *A fuzzy set (or type-1 fuzzy set) $A \subseteq X$ is a mapping $A : X \rightarrow [0,1]$. The value $A(x)$ is referred to as the* membership degree *of the element x to the fuzzy set A.*

Therefore, a fuzzy set A in X is characterized by a membership function $A(x)$ that associates each element $x \in X$ with a real number in the interval $[0,1]$; in other words, given x, the value $A(x)$ represents the degree of membership of x to A.
A fuzzy set is empty if and only if its membership function is identically zero for each $x \in X$. Two fuzzy sets A and B are equal, denoted as $A = B$, if and only if $A(x) = B(x)$ for all $x \in X$.
The complement of a fuzzy set A is A' and is defined as $A'(x) = 1 - A(x)$.
As in the case of ordinary sets, the notion of containment can be defined as well.

Definition 2. *A is contained in B (or, equivalently, A is a subset of B, or A is smaller than or equal to B) if and only if $A(x) \leq B(x)$, for each $x \in X$.*

Definition 3. *The union (resp. intersection) of two fuzzy sets A and B with respective membership functions $A(x)$ and $B(x)$ is a fuzzy set C, written as $C = A \cup B$ (resp., $C = A \cap B$), whose membership function is related to those of A and B by $C(x) = Max(A(x), B(x))$ (resp., $C(x) = Min(A(x), B(x))$), for all $x \in X$.*

The union (resp., intersection) is used to express the logical `OR` (resp. `AND`) operators, while the complement set represents the logical `NOT` operator.

3.2 The GeoJSON Format

GeoJSON (Butler et al., 2016) is a format for encoding geographic information layers, defined by using the JSON format. With a GeoJSON document, we can represent a region of space (i.e., a *Geometry*), a spatially bounded entity (i.e., a *Feature*), or a list of Features (i.e., a *FeatureCollection*).

GeoJSON supports several geometry types: *Point*, *LineString*, *Polygon*, *MultiPoint*, *MultiLineString*, *MultiPolygon*, and *GeometryCollection*. Features in GeoJSON are composed by a `geometry` field plus additional properties described by the `properties` field. The overall GeoJSON document is of type `FeatureCollection`, and the array field named `features` contains a list of features.

An example of GeoJSON document is reported in Listing 1. Features, that are contained in the `features` array field, describe weather measures, made by weather stations located in Regione Lombardia (Northern Italy). Each feature describes the station identifier, the date of measure, the temperature (in degree Celsius), the percentage of humidity and the atmospheric pressure in millibars.

4 SOFT QUERYING ON GeoJSON

Consider the GeoJSON document reported in Listing 1. With *soft querying of a GeoJSON document* we mean searching for features that meet a qualitative condition, in order to get a new GeoJSON document, Listing 1: Sample GeoJSON Document.

```
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "properties": {
        "Station_Id": 109,
        "Date": "2020-05-01",
        "temperature": 26,
        "humidity": 83,
        "pressure": 1000
      },
      "geometry": {
        "type": "Point",
        "coordinates":
           [9.379659683,45.26066732]
      }
    },
    { "type": "Feature",
      "properties": {
        "Station_Id": 114,
        "Date": "2020-05-01",
        "temperature": 24,
        "humidity": 99,
        "pressure": 980
      },
      "geometry": {
        "type": "Point",
        "coordinates":
           [9.267153115,45.32059361]
      }
    },
    { "type": "Feature",
      "properties": {
        "Station_Id": 117,
        "Date": "2020-05-01",
        "temperature": 17,
        "humidity": 81,
        "pressure": 874
      },
      "geometry": {
        "type": "Point",
        "coordinates":
           [10.33339294, 46.17572375]
      }
    }
  ]
}
```

in which features are sorted in reverse order of importance (as far as the qualitative selection condition is concerned).

A sample qualitative query may be: *extract those meteorological measures (features) that registered a high value of humidity with either a low value of temperature or a medium value of pressure or both.*

To explain the idea, suppose we can express three linguistic predicates, i.e., `High_Humidity`, `Low_Temperature` and `Medium_Pressure`.

By using a SQL-like syntax, the soft query may be expressed as reported in Listing 2.

Listing 2: Soft query in SQL-like syntax.

```
SELECT *
 FROM Meteo_Measures
 WHERE High_Humidity AND
 (Low_Temperature OR Medium_Pressure)
   FOR FUZZY SET Wanted
 ALPHA-CUT Wanted: 0.5
 ADD MEMBERSHIP OF Wanted AS rank
```

The idea is the following: if we are able to evaluate the three linguistic predicates on each feature, we can evaluate a compound `Wanted` linguistic predicate; the membership value associated to this linguistic predicate denotes the degree with which the single feature matches the compound linguistic condition. Let us describe our idea in details.

- The `FROM` clause specifies the source GeoJSON document.
- The `WHERE` clause specifies the compound linguistic condition, by evaluating the membership degree for each single feature.
  The last part of the clause, i.e., `FOR FUZZY SET Wanted`, gives a name to the fuzzy set obtained by evaluating the compound linguistic condition.
- The `SELECT` clause specifies the properties of the feature to project on.
- The `ALPHA-CUT` clause selects only those features that belong to the `Wanted` fuzzy set with membership degree greater than or equal to `0.5`.
- `ADD MEMBERSHIP OF Wanted AS rank` specifies that the membership value of the `Wanted` fuzzy set becomes a new property in features, whose name is `rank`; implicitly, we assume that the features are sorted by the `rank` properties in descending order.

We now present our semantic model.

- Each feature can have a pool of associated fuzzy sets; the memberships degree for each fuzzy set denotes the degree with which the feature belongs to the fuzzy set. Formally, the data model for a feature is:

  $f = \langle \texttt{properties}, \texttt{geometry}, \texttt{fuzzysets} \rangle$

  where `properties` and `geometry` are defined exactly as in the GeoJSON standard; The `fuzzysets` field is a key/value map, that associates a fuzzy set name fsn to the membership degree of f to the fuzzy set fsn.
- The `WHERE` clause evaluates a soft condition, computes the membership degree and adds the new fuzzy set name and the computed membership degree to the map $f.\texttt{fuzzysets}$.

Clearly, linguistic predicates can be interpreted in terms of fuzzy sets, i.e., the satisfaction degree of a linguistic predicate corresponds to the membership degree to a fuzzy set. So, the previous soft query is correct only if we guess that the features in the source GeoJSON document have membership degrees to the fuzzy sets High_Humidity, LowTemperature and Medium_Pressure already computed. However, this is not true in GeoJSON documents: so, we can assume that their membership degree is 0 (as a standard behavior, we can assume that when a fuzzy set name $fsn \notin f$.fuzzysets, its membership degree is 0). Thus, the previous soft query is correct, but would not select any feature.

Consequently, we have to extend our proposal and consider that:

1. fuzzy operators for evaluating membership degrees are available;
2. the FROM clause must accept nested soft queries.

Based on the above-mentioned considerations, and supposing that three fuzzy operators named Has_High_Humidity, Has_Low_Temperature and Has_Medium_Pressure are defined, the soft query can be written as reported in Listing 3.

Listing 3: Nested soft query in SQL-like syntax.

```
SELECT *
FROM (SELECT *
 FROM (SELECT *
  FROM (SELECT *
   FROM Meteo_Measures
   WHERE Has_High_Humidity(humidity)
    FOR FUZZY SET High_Humidity
  )
  WHERE Has_Low_Temperature(temperature)
   FOR FUZZY SET Low_Temperature
 )
 WHERE Has_Medium_Pressure(pressure)
  FOR FUZZY SET Medium_Pressure
)
WHERE High_Humidity AND
(Low_Temperature OR Medium_Pressure)
  FOR FUZZY SET Wanted
ALPHA-CUT Wanted: 0.5
ADD MEMBERSHIP OF Wanted AS rank
```

Consequently, the problem we address in this paper can be formulated as follows

Problem 1. *Given a soft query sq on a GeoJSON document, rewrite the query into a lower-level fuzzy query language for JSON data sets, that generates a new GeoJSON document containing only the features that satisfy the soft condition, sorted by the membership degree in reverse order.*

5 QUERYING GeoJSON FEATURES WITH J-CO-QL

The J-CO-QL language is a rich language to query and transform collections of complex JSON documents. Here, we show how it can be used to select features in GeoJSON documents. This way, we will briefly introduce the language (the reader can refer to (Bordogna et al., 2017; Bordogna et al., 2018; Psaila and Fosci, 2018)) and what it is possible to do without using fuzzy constructs, that will be presented in Section 6.2.

Suppose we have a GeoJSON document that describes measures performed by weather stations (see Listing 1), stored within a database named MyDB as the lonely document of a collection (container of documents) named Meteo_Measures. In this paper, we do not consider how to connect the *J-CO* framework to JSON stores (such as *MongoDB*); the interested reader can refer to (Psaila and Fosci, 2018).

Listing 4: *J-CO-QL* query that selects features from a GeoJSON document.

```
GET COLLECTION Meteo_Measures@MyDB;

EXPAND
 UNPACK WITH ARRAY .features
 ARRAY .features TO feature
 DROP OTHERS;

FILTER
 CASE WHERE WITH
   .feature.item.properties.temperature
   AND
   .feature.item.properties.temperature
   > 30
  GENERATE { .type: .feature.item.type,
   .properties:
            .feature.item.properties,
   .geometry: .feature.item.geometry,
   .key: "K" }
  DROP OTHERS;

GROUP
 PARTITION WITH .key
  BY .key INTO .features
   DROP GROUPING FIELDS
   GENERATE { .features: .features,
        .type: "FeatureCollection" }
 DROP OTHERS;

SAVE AS Filtered@MyDB;
```

The query in Listing 4 generates a new GeoJSON document, and stores it into a database collection named Filtered. Let us discuss it.

The query contains five instructions, performing either simple tasks or complex tasks. The underlying

Listing 5: Temporary collection after the `GROUP` instruction in Listing 4.

```
{"type" : "FeatureCollection",
 "features" : [
  { "type" : "Feature",
    "geometry" : {
      "coordinates" :
        [9.379659683, 45.26066732],
      "type" : "Point"
    },
    "properties" : {
      "Date" : "2020-08-01",
      "Station_Id" : 109,
      "humidity" : 97,
      "pressure" : 750,
      "temperature" : 36.7 }    },
  { "type" : "Feature",
    "geometry" : {
      "coordinates" :
          [9.267153115, 45.32059361],
      "type" : "Point"
    },
    "properties" : {
      "Date" : "2020-08-01",
      "Station_Id" : 114,
      "humidity" : 99,
      "pressure" : 745,
      "temperature" : 38.2 }  }
 ]
}
```

semantics is the following: each instruction starts from an *Execution State* s_i, and produces a new execution state $s_{(i+1)}$. Each execution state contains several information, including also the *Temporary Collection*, i.e., a collection of JSON documents that constitutes the input of the instruction. Thus, each instruction may use this collection and may generate a new temporary collection.

We can now describe the query in Listing 4.

- The `GET COLLECTION` instruction retrieves the collection named `Meteo_Measures` from the database named `MyDB`. This collection becomes the starting *temporary collection* of the process. Notice that in this case the collection is composed by just one document.

Listing 6: Temporary collection after the `EXPAND` instruction in Listing 4.

```
{ "type" : "FeatureCollection",
  "feature" : {
   "item" : {
    "geometry" : {
     "coordinates" :
       [9.379659683, 45.26066732],
     "type" : "Point"},
    "properties" : {
     "Date" : "2020-08-01",
     "Station_Id" : 109,
     "humidity" : 97,
     "pressure" : 750,
     "temperature" : 36.7 },
    "type" : "Feature" },
   "position" : 0 }   }
```

```
{ "type" : "FeatureCollection",
  "feature" : {
   "item" : {
    "geometry" : {
     "coordinates" :
       [9.267153115, 45.32059361],
     "type" : "Point" },
    "properties" : {
     "Date" : "2020-08-01",
     "Station_Id" : 114,
     "humidity" : 99,
     "pressure" : 745,
     "temperature" : 38.2 },
    "type" : "Feature" },
   "position" : 1 }   }
```

```
{ "type" : "FeatureCollection",
  "feature" : {
   "item" : {
    "geometry" : {
     "coordinates" :
       [10.33339294, 46.17572375],
     "type" : "Point" },
    "properties" : {
     "Date" : "2020-08-01",
     "Station_Id" : 117,
     "humidity" : 81,
     "pressure" : 751,
     "temperature" : 17.3 },
    "type" : "Feature" },
   "position" : 2 }   }
```

- The `EXPAND` instruction selects and transforms documents in the temporary collection generating a new temporary collection. In this case, the `UNPACK` clause selects those documents containing an array structure in a field named `features` and then generates new documents, each one related to each element in the `features` array. Each new document has a `feature` field containing one of the value in the source `features` array, while the other fields in the source document remain unchanged. The final `DROP OTHERS` clause, which in this case is ineffective, discards all those documents in the temporary collection that do not match the selecting condition.
  Listing 6 reports the temporary collection obtained by expanding the GeoJSON document reported in Listing 1. Notice that each document contains the `feature` field, as specified; this field, in turns, contains the `item` field, that actually is the previously nested document, and the

Listing 7: Temporary collection after the FILTER instruction in Listing 4.

```
{ "type" : "Feature"
  "key" : "K",
  "geometry" : {
     "coordinates" :
        [9.379659683, 45.26066732],
     "type" : "Point"
  },
  "properties" : {
     "Date" : "2020-08-01",
     "Station_Id" : 109,
     "humidity" : 97,
     "pressure" : 750,
     "temperature" : 36.7
  }   }
```

```
{ "type" : "Feature"
  "key" : "K",
  "geometry" : {
    "coordinates" :
       [9.267153115, 45.32059361],
    "type" : "Point"
  },
  "properties" : {
     "Date" : "2020-08-01",
     "Station_Id" : 114,
     "humidity" : 99,
     "pressure" : 745,
     "temperature" : 38.2
  }   }
```

position field, that denotes the position formerly occupied by the document in the array.

- The FILTER instruction selects and transforms the documents in the temporary collection, generating a new temporary collection. In this case, since documents describe weather measures, the CASE WHERE clause selects documents having the temperature field (by means of the WITH predicate) whose value is above 30 degrees.
  The selected documents are restructured by the GENERATE action, in order to simplify them. Specifically, we keep out all fields inside the .feature.item structure, generating fields at root level as type, property and geometry. Moreover, an extra key field with a constant value is added. The final DROP OTHERS clause discards documents that are not selected by the CASE WHERE clause.
  Notice that the new temporary collection contains two documents, as reported in Listing 7.
- The GROUP instruction creates a partition in the temporary collection and groups the documents in the partition into a new single document.
  In this case, by means of the WITH predicate, all documents that hold a key field are selected to be partitioned. By means of the BY clause, documents in the partition are grouped based on the value of the key field: since we have only one constant value, all documents in the partition are grouped into one single document.
  The INTO clause specifies the name of the new array field that has to contain grouped documents: in this case, the name is features. The DROP GROUPING FIELDS option discards the key field. The GENERATE action maintains unchanged the features field and adds a new type field with "FeatureCollection" as value.
  Notice that the new temporary collection contains one GeoJSON document, as reported in Listing 5.
- Finally, the temporary collection produced by the GROUP instruction is saved into the MyDB database with name Filtered.

The dot notation .features to refer to fields is motivated by the fact that documents can contain nested documents and fields; the dot notation allows for referring to nested fields, for example .A.B (the initial dots means that the A field is in the root level).

6 SOFT QUERYING IN J-CO-QL

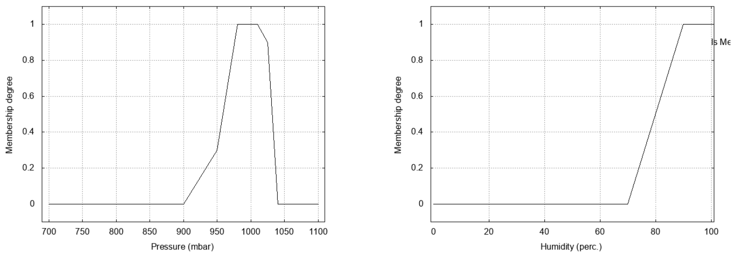
We can now present fuzzy *J-CO-QL* queries (proposed in (Psaila and Marrara, 2019)), which provide the tools to solve Problem 1.

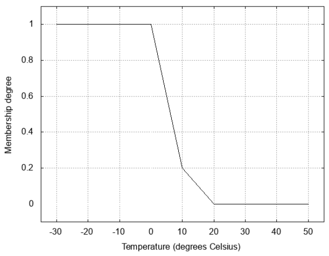
6.1 Creating Fuzzy Operators

In order to evaluate if a document belongs to a fuzzy set, it is necessary to define some operators able to compute the membership degree of a document with respect to a fuzzy set.

The fuzzy extension of the *J-CO-QL* language offers a specific construct to create fuzzy operators. We exploit it (see Listing 8) to define the fuzzy operators used in the soft query presented in Listing 3.

Let us consider the definition of the Has_Medium_Pressure operator. First of all, the PARAMETERS clause specifies the formal parameters received by the operator Has_Medium_Pressure. Specifically, the Has_Medium_Pressure operator receives one single parameter, named pressure, defined as an integer value. When the operator is invoked, all parameters are assembled into one JSON document, having the formal parameters as fields and the actual parameters as values. In our case, this JSON document will have only the pressure field. In order to obtain the membership degree, we have to evaluate an expression based on this parameter. The





(a) `Has_Medium_Pressure`. (b) `Has_High_Humidity`. (c) `Has_Low_Temperature`.

Figure 1: Membership Functions of Fuzzy Operators in Listing 8.

EVALUATE clause specifies this expression. In our case, the value of the pressure parameter is kept untouched.

However, not all values of the input parameters are valid: the PRECONDITION clause specifies a condition to be verified before performing the evaluation. If the condition is false, no membership value is generated and the operator generates an evaluation error. In our case, an atmospheric pressure below zero has no physical meaning.

The RANGE clause specifies the range within which the result of the EVALUATE expression is mapped to the corresponding membership degree, by means of the polyline function defined in the subsequent POLYLINE clause. In this case, we consider a range between 700 and 1100 millibar.

Finally, the POLYLINE clause defines the polyline used as membership function. It is defined as a sequence of pairs (x,y), where x is a coordinate in the range specified by the RANGE clause, while y is a value specified in the range $[0,1]$. The first pair has the value of the x coordinate that coincides with the left bound of the range, while the last pair has a value for the x coordinate that is equal to the right bound of the range. This way, it is possible to define the membership function as a polyline, where the y coordinate denotes the final membership value.

For example, the membership function for the Has_Medium_Pressure operator is depicted in Figure1a: notice the points of the polyline that correspond to the sequence of pairs. If we suppose that the actual value of the pressure parameter is 1003, the corresponding y coordinate is 1 (the membership degree), i.e., the pressure belongs to the set of medium pressure; if the value for the pressure parameter is 1045, the corresponding membership degree is 0, meaning that the pressure does not belong at all to the set of medium pressure. And so on. We assume that in case of values of the expression specified in the EVALUATE clause that are lower than the left bound (greater than the right bound, resp.), the y value of the left bound (right bound, resp.) is the membership value. For example, if the pressure parameter is 1140 mb, the membership value is 0.

Similarly, Listing 8 defines the Has_High_Humidity operator, which considers values over 0% as valid humidity and whose membership function is in the $[0\%,100\%]$ range as depicted in Figure 1b. It also defines the Has_Low_Temperature operator, which considers values over the absolute zero $(-273°)$ as valid temperature and whose membership function is in the $[-30°,50°]$ range, as depicted in Figure 1c.

Listing 8: Definition of the Fuzzy Operators.

```
CREATE FUZZY OPERATOR
    Has_Medium_Pressure
  PARAMETERS pressure TYPE Integer
  PRECONDITION .pressure >=0
  EVALUATE .pressure
  RANGE (700, 1100)
  POLYLINE (700,0), (900,0),
           (950,0.3), (980,1),
           (1010,1), (1025, 0.9),
           (1040,0), (1100,0);

CREATE FUZZY OPERATOR
    Has_High_Humidity
  PARAMETERS humidity TYPE Integer
  PRECONDITION .humidity >=0
  EVALUATE .humidity
  RANGE (0, 100)
  POLYLINE (0,0), (70,0),
           (90,1), (100,1);

CREATE FUZZY OPERATOR
    Has_Low_Temperature
  PARAMETERS temperature TYPE Integer
  PRECONDITION .temperature >=-273
  EVALUATE .temperature
  RANGE (-30, 50)
  POLYLINE (-30,1), (0,1),(10,0.2),
           (20,0), (50,0);
```

6.2 Fuzzy Sets in J-CO-QL Queries

The fuzzy extension to *J-CO-QL* can be actually used to perform the soft query presented in Listing 3. The fuzzy *J-CO-QL* query is reported in Listings 9, 10 and 11. The first thing to do is to introduce how the data model has been extended, in order to deal with membership to fuzzy sets.

A JSON document d can belong to one or more fuzzy sets $A_1, A_2, \dots, A_n$. For each fuzzy set A_i, d belongs to A_i with a membership degree $A_i(d) \in [0,1]$. To represent the multiple membership degrees of d, a special root-level field named ~fuzzysets is introduced: it is a nested structure, containing only numerical fields; each field denotes the membership degree of the JSON document to the fuzzy set named as the field. Notice that the name ~fuzzysets is fully compliant with JSON naming rules. Furthermore, notice that it represents the map introduced in Section 4 that associates fuzzy sets to membership degrees.

As an example, consider the two JSON documents reported in Listing 12. For each document, the membership to four fuzzy sets has been evaluated: in fact, notice the presence of the ~fuzzysets field. The reader can notice that the names of the fuzzy sets are the same introduced in the soft query in Listing 3: hereafter, we will see how to compute them.
However, we want to point out that the documents do not say how the membership degrees to fuzzy sets were evaluated, they only say the degree with which they belong to some fuzzy sets. The advantage of this solution is that the documents are still standard JSON documents: they can be stored into database collections and retrieved from them.

In Section 5, we presented the FILTER instruction without fuzzy sets. From now on, we will make use of the extended version of the FILTER instruction, that provides specific clauses to deal with fuzzy sets. Remember that the general goal is to perform the soft query in Listing 3: starting from the Meteo_Measures GeoJSON document (reported in Listing 1), we want to generate a new GeoJSON document, containing only those features that have high humidity and either low temperature or medium pressure or both, sorted in reverse order of relevance.

Let us start with the preamble of the *J-CO-QL* query, reported in Listing 9. This part of the query coincides with the first three instructions in the classical *J-CO-QL* query reported in Listing 4: this is not surprising, because these three instructions retrieve the collection containing the desired GeoJSON layer from the database, expand the features into one single document for each feature and remove nested structures that are not necessary.

Listing 10 reports the core part of the query. In this part, it is necessary to actually evaluate fuzzy sets, both those that express basic linguistic predicates, such as Medium_Pressure, High_Humidity and Low_Temperature, and the one (Wanted) derived by combining the former ones.

The first FILTER instruction evaluates the membership degree of documents in the temporary collection (generated by the last instruction in Listing 9) to the High_Humidity fuzzy set.
Specifically, after the WHERE condition, the CHECK FOR FUZZY SET sub-clause denotes that the membership degree to this fuzzy set is evaluated.
In particular, the subsequent USING sub-clause reports the fuzzy condition to be used in order to evaluate the membership degree. In this case, the fuzzy condition evaluates the Has_High_Humidity operator on each single document; the membership degree returned by the fuzzy operator becomes the degree with which each single document belongs to the High_Humidity fuzzy set.
Notice that the GENERATE action is absent: the structure of documents that are put into the output temporary collection does not change, apart from the novel ~fuzzysets field that is automatically added.

The second and the third FILTER instructions perform the same activity, i.e., they evaluate the degree of membership to the Medium_Pressure fuzzy set and to the Low_Temperature fuzzy set. After the third FILTER instruction, the ~fuzzysets field contains three fields, one for each of the three evaluated fuzzy sets.

The last FILTER instruction in the core part of the query (Listing 10) evaluates the membership to the Wanted fuzzy set. The fuzzy condition (USING) uses the previously computed membership degrees to fuzzy sets High_Humidity, Medium_Pressure and Low_Temperature as linguistic predicates, in

Listing 9: Fuzzy *J-CO-QL* Query (preamble).

```
GET COLLECTION Meteo_Measures@MyDB;

EXPAND
 UNPACK WITH ARRAY .features
  ARRAY .features TO feature
 DROP OTHERS;

FILTER
 CASE WHERE WITH .feature.item
  GENERATE
   { .type: .feature.item.type,
     .properties:
       .feature.item.properties,
     .geometry:
       .feature.item.geometry }
 DROP OTHERS;
```

Listing 10: Fuzzy *J-CO-QL* Query (core).

```
FILTER
 CASE WHERE
  WITH .properties.humidity
  CHECK FOR FUZZY SET High_Humidity
  USING Has_High_Humidity(
    .properties.humidity )
 DROP OTHERS;

FILTER
 CASE WHERE
  WITH .properties.pressure
  CHECK FOR FUZZY SET Medium_Pressure
  USING Has_Medium_Pressure(
     .properties.pressure )
 DROP OTHERS;

FILTER
 CASE WHERE
  WITH .properties.temperature
  CHECK FOR FUZZY SET Low_Temperature
  USING Has_Low_Temperature(
       .properties.temperature )
 DROP OTHERS;

FILTER
 CASE WHERE KNOWN FUZZY SETS
        High_Humidity,
        Low_Temperature,
        Medium_Pressure
  CHECK FOR FUZZY SET Wanted
  USING High_Humidity AND
  (Low_Temperature OR Medium_Pressure)
  ALPHA-CUT 0.5 ON Wanted
 DROP OTHERS;
```

order to evaluate the degree with which documents belong to the wished `Wanted` fuzzy set.

The structure of the instruction is the same, but with some small differences. First of all, in the `WHERE` clause, we use the `KNOWN FUZZY SETS` predicate, that is evaluated as *true* if the membership to all the listed fuzzy sets has been evaluated in a document.

Second of all, notice that the fuzzy condition in the `USING` sub-clause now simply relies on fuzzy sets that are used as linguistic predicates.

Finally, the `ALPHA-CUT` clause specifies that only documents having a membership degree greater than or equal to `0.5` to the `Wanted` fuzzy set are kept in the output temporary collection.

In Listing 12, we report the documents that are present in the temporary collection at the end of the core part of the query (Listing 10). The reader can notice that, of the three features formerly present in the source GeoJSON document (reported in Listing 1), only two of them have been selected by the `ALPHA-CUT` clause. Notice, in each document, the presence of the special `~fuzzysets` field, with the four fields corresponding to the four fuzzy sets and the corresponding membership degrees.

Listing 11: Fuzzy *J-CO-QL* Query (tail).

```
FILTER
 CASE WHERE
    WITH ."~fuzzysets".Wanted
  GENERATE { .type: .type,
    .properties: .properties,
    .geometry: geometry, .key: 1,
    .properties.rank:
       ."~fuzzysets".Wanted }
  DROPPING ALL FUZZY SETS
 DROP OTHERS;

GROUP
 PARTITION WITH .key
  BY .key INTO .features
  DROP GROUPING FIELDS
  ORDER BY .properties.rank DESC
  GENERATE
   { .type: "FeatureCollection",
     .features: .features }
 DROP OTHERS;

SAVE AS Filtered@MyDB;
```

Finally, the tail of the query, reported in Listing 11, can be executed, to rebuild a GeoJSON document containing the wished features.

The first `FILTER` instruction in Listing 11 prepares documents to be grouped into the output GeoJSON document. In particular, the `GENERATE` action inserts a `rank` field into the `properties` field of each document, whose value is the membership degree to the `Wanted` fuzzy set: this value denotes how much the document satisfies the linguistic selection condition. Notice that, apart from the need to refer to the `~fuzzysets` field as `"~fuzzysets"`, it is a standard JSON field and can be referred to accordingly. As in Listing 6, a field named `key` with a constant value is added, that will play the role of grouping field in the next `GROUP` instruction.

The `DROPPING ALL FUZZY SETS` option asks for removing the `~fuzzysets` special field from the output documents.

The `GROUP` instruction rebuilds a unique GeoJSON document. With respect to the `GROUP` instruction in Listing 6, now we have the `ORDER BY` clause that sorts documents that have been grouped into the new `features` array in descending order (`DESC` option) of `rank`.

Finally, the temporary collection, that contains only the wished GeoJSON document (reported in Listing 13), is saved into the database.

The fuzzy *J-CO-QL* query presented in this section, demonstrates that *J-CO-QL* is effective in performing the soft query reported in Listing 3. However, although *J-CO-QL* is effective, it is a rich and complex language, that asks for the development of

Listing 12 Document belonging to some fuzzy sets.

```
{ "geometry" : {
    "coordinates" :
       [ 9.379659683, 45.26066732 ],
    "type" : "Point"    },
  "properties" : {
    "Date" : "2020-05-01",
    "Station_Id" : 109,
    "humidity" : 83,
    "pressure" : 1000,
    "temperature" : 26  },
  "type" : "Feature",
  "~fuzzysets" : {
    "High_Humidity" : 0.65,
    "Low_Temperature" : 0.0,
    "Medium_Pressure" : 1.0,
    "Wanted" : 0.65     }
}
```

```
{ "geometry" : {
    "coordinates" :
       [ 9.267153115, 45.32059361 ],
    "type" : "Point"    },
  "properties" : {
    "Date" : "2020-05-01",
    "Station_Id" : 114,
    "humidity" : 99,
    "pressure" : 980,
    "temperature" : 24     },
  "type" : "Feature",
  "~fuzzysets" : {
    "High_Humidity" : 1.0,
    "Low_Temperature" : 0.0,
    "Medium_Pressure" : 1.0,
    "Wanted" : 1.0     }
}
```

higher-level tools, such as the soft query language sketched in Section 4. In Section 6.3 we show that it is possible to devise a query-rewriting technique, that generates the fuzzy *J-CO-QL* query reported in Listings 9, 10 and 11. Consequently, based on this consciousness, Problem 1 can be refined as follows.

Problem 2. *Given a soft query sq on a GeoJSON document, rewrite the query into a* J-CO-QL *query that generates a new GeoJSON document containing only the features that satisfy the soft condition, sorted vs the membership degree in reverse order.*

6.3 Rewriting Soft Queries

We can finally provide the final contribution of the paper, i.e., addressing Problem 2: we present an algorithm for rewriting soft queries into fuzzy *J-CO-QL* queries, that proceeds as follows.

1. The concept of temporary collection will be exploited to save intermediate collections generated by nested soft queries.

Listing 13: GeoJSON document saved to database.

```
{ "type" : "FeatureCollection",
  "features" : [
    { "type" : "Feature",
      "geometry" : {
        "coordinates" :
          [ 9.267153115, 45.32059361 ],
        "type" : "Point" },
      "properties" : {
        "Date" : "2020-05-01",
        "Station_Id" : 114,
        "humidity" : 99,
        "pressure" : 980,
        "temperature" : 24,
        "rank" : 1.0 }  },
    { "type" : "Feature",
      "geometry" : {
        "coordinates" :
          [ 9.379659683, 45.26066732 ],
        "type" : "Point" },
      "properties" : {
        "Date" : "2020-05-01",
        "Station_Id" : 109,
        "humidity" : 83,
        "pressure" : 1000,
        "temperature" : 26,
        "rank" : 0.65 }  }
    ]
}
```

2. The data model of *J-CO-QL* naturally deals with fuzzy sets. So, given a feature f, the map f.fuzzysets corresponds to the special field ~fuzzysets.
3. Given a nested soft query sq_i in the FROM clause of another soft query $sq_{(i-1)}$, the translation of sq_i generates the temporary collection that will be the input to the translation of $sq_{(i-1)}$.
4. The preamble of the *J-CO-QL* query is the code that gets the collection from the database and extracts features from within the GeoJSON document; the last part of the *J-CO-QL* query is the code that groups features to generate one single GeoJSON document.

Algorithm 1 reports the pseudo-code of the rewriting process. Hereafter, we present it.

- The rewriting process is managed by the **RewriteSoftQuery** function, that receives a query sq as input parameter; it returns a string with the corresponding *J-CO-QL* query.
- The **RewriteSoftQuery** function takes the string generated by the recursive **RewriteSingleQuery** function and appends the tail of the query, generated by the **GenTail** function.
- The **RewriteSingleQuery** function recursively deals with nesting of soft queries. If the FROM clause of the current query is a nested query, it

Algorithm 1 : Rewrites Soft Queries on GeoJSON documents to fuzzy *J-CO-QL* queries.

```
Function RewriteSingleQuery(sq: SoftQuery)
    : String
Begin
  $T$ := "";
  If sq.FROM Is a Soft Query Then
    $T$ := RewriteSingleQuery(sq.FROM);
  Else
    $T$ := GenPreamble(sq.FROM)
  End If
  $R := T \bullet$GenFilter(sq.FROM);
  Return $R$;
End Function

Function RewriteSoftQuery(sq: SoftQuery)
    : String
Begin
  $T$ := RewriteSingleQuery(sq);
  $R := T \bullet$ GenTail(sq);
  Return $R$;
End
```

recursively recalls itself; otherwise, it is a layer name and it calls the **GenPreamble** function. The **GenPreamble** function prepares the preamble of the *J-CO-QL* query, that gets the collection containing the GeoJSON document from the database and expands it, to obtain as many JSON documents describing features as the features that are grouped within the `features` array at the root level of the GeoJSON document.

- Finally, the **RewriteSingleQuery** function appends the output of the **GenFilter** function, that actually generates a *J-CO-QL* `FILTER` instruction that corresponds to the current soft query.

For the sake of space, we do not report the pseudo code for the **GenFilter** and the **GenTail** functions. However, having in mind Listing 10, the **GenFilter** function generates the `FILTER` instruction corresponding to each single soft query. Having in mind Listing 11, the **GenTail** function generates the whole tail of the query; specifically, it uses the outer soft query to generate the first `FILTER` instruction in Listing 11, that generates the `rank` field by taking the membership value to the `Wanted` fuzzy set, as well as the `ORDER BY` clause in the `GROUP` instruction.

The presented algorithm is thus able to rewrite a high-level and generic soft query on GeoJSON documents, as formulated in Section 4, into a fuzzy *J-CO-QL* query. Thus, we solve Problems 1 and 2, as well as we have shown that it is possible to effectively use the *J-CO-QL* language and, more in general, the *J-CO* framework, as the underlying engine to develop high-level soft query languages on specific classes of JSON documents, such as GeoJSON documents.

7 CONCLUSIONS

In this paper, we introduced the idea of writing soft queries, i.e., queries that rely on linguistic predicates, on a specific class of JSON documents, i.e., GeoJSON documents. The usefulness of this approach is motivated by the fact that GeoJSON has become a widely used format for disseminating geographical information over the Web, but a standard query language able to natively query GeoJSON documents is not available yet.

We explored the adoption of the fuzzy constructs introduced in the *J-CO-QL* language, that is part of the *J-CO* framework, specifically designed to provide a powerful tool to manipulate and query collections of JSON documents in a seamless way with respect to the specific JSON storage system. The query-rewriting technique presented in Section 6.3 shows that it is possible to generate a complex fuzzy *J-CO-QL* query from a simple and high-level soft query. We showed that the high-level soft query language can be actually translated into a concrete query language (such as *J-CO-QL*). The effectiveness of the idea will be evaluated in future validation campaigns.

As a future work, we plan to further investigate the definition of high-level and soft query languages for JSON documents, by considering various JSON standards. In fact, we think that the development of effective query languages for JSON data sets is an important research line; and it is a concrete and useful application of the *J-CO* framework as underlying execution engine. Geo-tagging within JSOn (as GeoJSON) documents will be consider, to perform uncertain location-based queries (Bordogna et al., 2008).

REFERENCES

Blair, D. C. (1979). Information retrieval, 2nd ed. c.j. van rijsbergen. london: Butterworths; 1979: 208 pp. price: $32.50. *Journal of the American Society for Information Science*, 30(6):374–375.

Bordogna, G., Capelli, S., Ciriello, D. E., and Psaila, G. (2018). A cross-analysis framework for multi-source volunteered, crowdsourced, and authoritative geographic information: The case study of volunteered personal traces analysis against transport network data. *Geo-spatial Information Science*, 21(3):257–271.

Bordogna, G., Capelli, S., and Psaila, G. (2017). A big geo data query framework to correlate open data with social network geotagged posts. In *The Annual International Conference on Geographic Information Science*, pages 185–203. Springer.

Bordogna, G., Pagani, M., Pasi, G., and Psaila, G. (2008). Evaluating uncertain location-based spatial queries. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1095–1100.

Bordogna, G. and Pasi, G. (1995). Linguistic aggregation operators of selection criteria in fuzzy information retrieval. *International journal of intelligent systems*, 10(2):233–248.

Bordogna, G. and Psaila, G. (2009). Soft aggregation in flexible databases querying based on the vector p-norm. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 17(supp01):25–40.

Bosc, P. and Pivert, O. (1995). Sqlf: a relational database language for fuzzy querying. *IEEE transactions on Fuzzy Systems*, 3(1):1–17.

Bosc, P. and Prade, H. (1997). An introduction to the fuzzy set and possibility theory-based treatment of flexible queries and uncertain or imprecise databases. In *Uncertainty management in information systems*, pages 285–324. Springer.

Buell, D. A. (1985). A problem in information retrieval with fuzzy sets. *Journal of the American Society for Information Science (pre-1986)*, 36(6):398.

Butler, H., Daly, M., Doyle, A., Gillies, S., Hagen, S., Schaub, T., et al. (2016). The geojson format. *Internet Engineering Task Force (IETF)*.

Campi, A., Damiani, E., Guinea, S., Marrara, S., Pasi, G., and Spoletini, P. (2009). A fuzzy extension of the xpath query language. *Journal of Intelligent Information Systems*, 33(3):285.

Damiani, E. and Tanca, L. (2000). Blind queries to xml data. In *International Conference on Database and Expert Systems Applications*, pages 345–356. Springer.

Fuhr, N. (1989). Models for retrieval with probabilistic indexing. *Information Processing & Management*, 25(1):55–72.

Galindo, J., Urrutia, A., and Piattini, M. (2006). *Fuzzy databases: Modeling, design, and implementation*. IGI Global.

Herrera, F. and Herrera-Viedma, E. (1997). Aggregation operators for linguistic weighted information. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 27(5):646–656.

Kacprzyk, J. and Zadrożny, S. (1995). Fquery for access: Fuzzy querying for a windows-based dbms. *Bosc P., Kacprzyk J. (eds), Fuzziness in Database Management Systems. Studies in Fuzziness*, 5.

Kraft, D. H., Colvin, E., Bordogna, G., and Pasi, G. (2015). Fuzzy information retrieval systems: A historical perspective. In *Fifty Years of Fuzzy Logic and its Applications*, pages 267–296. Springer.

Marrara, S. and Pasi, G. (2016). Fuzzy approaches to flexible querying in xml retrieval. *International Journal of Computational Intelligence Systems*, 9(sup1):95–103.

Medina, J. M., Pons, O., and Vila, M. A. (1994). Gefred: A generalized model of fuzzy relational databases. *Information Sciences*, 76(1):87 – 109.

Psaila, G. and Fosci, P. (2018). Toward an anayist-oriented polystore framework for processing json geo-data. In *International Conferences on Applied Computing 2018, Budapest; Hungary, 21-23 October 2018*, pages 213–222. IADIS.

Psaila, G. and Marrara, S. (2019). A first step towards a fuzzy framework for analyzing collections of json documents. In *IADIS AC 2019*, pages 19–28.

Ramer, A. (1989). Possibility theory: An approach to computerized processing of uncertainty, by didier dubois and henri prade, plenum press, new york 1988. xvi+ 263 pages. *International Journal of General Systems*, 15(2):168–170.

Salton, G., Allan, J., Buckley, C., and Singhal, A. (1994). Automatic analysis, theme generation, and summarization of machine-readable texts. *Science*, 264(5164):1421–1426.

Waller, W. and Kraft, D. H. (1979). A mathematical model of a weighted boolean retrieval system. *Information Processing & Management*, 15(5):235–245.

Zadeh, L. A. (1965). Fuzzy sets. *Information and control*, 8(3):338–353.

Zadeh, L. A. (1975). The concept of a linguistic variable and its application to approximate reasoning—i. *Information sciences*, 8(3):199–249.