Towards Soft Web Intelligence by Collecting and Processing JSON Data Sets from Web Sources

Paolo Fosci[a] and Giuseppe Psaila[b]
University of Bergamo, DIGIP, Viale Marconi 5, 24044 Dalmine (BG), Italy
paolo.fosci@unibg.it, giuseppe.psaila@unibg.it

Keywords: Web Intelligence, Continuous Acquisition of JSON Data Sets from Web Sources, Soft Querying on JSON Data Sets.

Abstract: Since the last two decades, Web Intelligence has denoted a plethora of approaches to discover useful knowledge from the vast World-Wide Web; however, dealing with the immense variety of the Web is not easy and the challenge is still open. In this paper, we moved from the previous functionalities provided by the *J-CO* Framework (a research project under development at University of Bergamo Italy), to identify a vision of Web Intelligence scopes in which capabilities of soft computing and soft querying provided by a stand-alone tool can actually create novel possibilities of making useful analysis of *JSON* data sets directly coming from Web sources. The paper identifies some extensions to the *J-CO* Framework, which we implemented; then it shows an example of soft querying enabled by these extensions.

1 INTRODUCTION

Web Intelligence (Yao et al., 2001) is now an established sub-discipline of computer science. Although it is quite difficult to exactly characterize it, we can report the original definition proposed in (Yao et al., 2001): "Web Intelligence (WI) exploits Artificial Intelligence (AI) and advanced Information Technology (IT) on the Web and Internet": in other words, we could say that the aim of Web Intelligence is "discovering knowledge from the vast World-Wide Web".

Two decades later, Web Intelligence is still an open research area: the Web is not suited to easily support Web Intelligence tasks yet. So, building a Web Intelligence environment is not easy and novel tools (even those that were not explicitly designed for that) can contribute, if suitably adapted.

A novel trend of Web technology is the exploitation of *JSON* (JavaScript Object Notation[1]) as the standard format for representing data sets: Web Services, social-media APIs (Application Programming Interfaces) and Open-Data portals adopt *JSON*. Looking at NoSQL databases, *JSON* document stores represent probably the most successful category: these are DBMSs (Database Management Systems) that natively store "collections of *JSON* documents in a schema-less way. These DBMSs, in particular *MongoDB*, are now widely exploited.

In our previous works, we realized that analysts wishing to gather, integrate and analyze *JSON* data sets (possibly in *JSON* document stores) missed the right tools. For this reason, we started the development of the *J-CO* Framework: it is a platform-independent suite that does not rely on any specific *JSON* document store; the framework is built around a high-level language designed to manage heterogeneous *JSON* data sets. Furthermore, since (Fosci and Psaila, 2021a), we have added soft-querying capabilities: it is possible to define fuzzy operators and exploit them to evaluate the membership of documents to fuzzy sets and query documents accordingly (so as to consider vagueness, imprecision and uncertainty).

So, the following question arose: is it possible to exploit the *J-CO* Framework to build a practical system that provides analysts with the capability of soft querying *JSON* data sets directly acquired from Web sources, in a Web-Intelligence scope (i.e., a context in which Web-Intelligence tasks must be performed)?

The paper presents our vision of Web Intelligence and identifies the missing functionalities to let the *J-CO* Framework suitable for a Web-Intelligence scope. The ultimate goal is to show that a stand-alone tool like the *J-CO* Framework can actually provide many useful functionalities, in particular soft querying on

[a] https://orcid.org/0000-0001-9050-7873
[b] https://orcid.org/0000-0002-9228-560X
[1] https://www.rfc-editor.org/rfc/rfc7159

Fosci, P. and Psaila, G.
Towards Soft Web Intelligence by Collecting and Processing JSON Data Sets from Web Sources.
In *Proceedings of the 18th International Conference on Web Information Systems and Technologies (WEBIST 2022)*, pages 302-313
ISBN: 978-989-758-813-2; ISSN: 2184-3252

JSON data sets, to widen the set of tools available to analysts for performing Web-Intelligence tasks. We could call this achievement as "Soft Web Intelligence".

In the remainder, Section 2 introduces relevant related works. Section 3 provides a short presentation of the *J-CO* Framework and shows a case study. Section 4 presents our vision, while Section 5 identifies limitations and presents novel developments to make the *J-CO* Framework suitable for Web-Intelligence scopes. Section 6 shows an example of soft querying *JSON* data sets in a Web-Intelligence scope. Section 7 draws conclusions and future work.

2 RELATED WORK

The concept of Web Intelligence was introduced in (Yao et al., 2001). It originates from the concept of Business Intelligence, which encompasses tools for reporting, making multi-dimensional analysis and, more in general, discovering knowledge from data possibly gathered in data warehouses (Negash and Gray, 2008). In some sense, we can figure out that Web Intelligence is the evolution of Business Intelligence towards the World-Wide Web, which is intrinsically semi-structured and incredibly vast.

Although (Yao et al., 2001) talks about Artificial Intelligence as the key tool for performing Web Intelligence, Artificial Intelligence itself is not clearly defined. Data Mining techniques have been successful (when applied to relational data) and they are accepted as Artificial-Intelligence techniques. Thus, they have been immediately adopted for Web Intelligence (Han and Chang, 2002), with the consciousness that peculiarities of the Web were (and still are) a hard challenge. Web Intelligence has been applied to the education field too, as reported in (Devedžić, 2004). Moreover, (Molnár et al., 2014) exploits Web Intelligence for Customer Relationship Management.

Fuzzy Logic and Soft Computing constitute a side part of Artificial Intelligence. Thus, Zadeh, the creator of Fuzzy-Set Theory (Zadeh, 1965), provided his interpretation of the concept of Web Intelligence in (Zadeh, 2004a; Zadeh, 2004b), in which soft computing could play an important role. Specifically, Zadeh thought about the concept of WebIQ (or WIQ), as the evolution of the idea of "Machine IQ": machines are increasing their capability to answer vague or imprecise questions; fuzzy-set theory provides the formal tools towards WebIQ. This is not the only attempt to give a wide interpretation of the concept of Web Intelligence. We can also cite Computational Web Intelligence (Zhang and Lin, 2002; Zhang and Lin, 2002) as the adoption of "Computational Intelligence in Web-Intelligence" scopes, as well as "Brain Informatics" (Zhong et al., 2006), whose goal is to foster Web Intelligence through techniques that come out from the study of the human brain.

However, focusing on fuzzy logic and soft computing in Web-Intelligence scopes, we are aware of very few works. The paper (Kacprzyk and Zadrożny, 2010) exploits soft computing in a group decision-making system to express preferences, while Web Intelligence is used to gather knowledge to be exploited to make decisions. The paper (Poli, 2015) uses `FUZZYALGOL`, a fuzzy procedural programming language (Reddy, 2010), for soft querying Web sources.

Consequently, we can say that, at the best of our knowledge, the contribution of the paper (i.e., adopting a stand-alone tool for soft querying *JSON* data sets continuously acquired and integrated from Web sources) is novel and not explored yet. We rely on our previous works on soft querying in relational databases (Bordogna and Psaila, 2009), on fuzzy clustering of data coming from moving users of social media (Bordogna et al., 2017b) and on blind querying of Open-Data portals (Pelucchi et al., 2017a; Pelucchi et al., 2017b; Pelucchi et al., 2018).

3 THE J-CO FRAMEWORK

In this section, we provide a synthetic presentation of the *J-CO* Framework and a short script for soft querying meteorological data coming from a Web source.

3.1 Organization of the Framework

The *J-CO* Framework is a pool of software tools whose goal is to provide analysts with a powerful support for gathering, integrating and querying possibly-large collections of *JSON* data sets. The core of the framework is its query language, named *J-CO-QL*$^+$. We will show a sample script for soft querying Web sources in Section 3.2, where the language will be synthetically introduced in action.

The organization of the framework before the evolution presented in this work is depicted in Figure 1.

- *J-CO-QL*$^+$ *Engine*. This component actually executes *J-CO-QL*$^+$ scripts. It is able to retrieve data from external document databases (for example, managed by *MongoDB*) and save results into them; it also can send HTTP requests to Web Services and portals, to get *JSON* data sets directly from Web sources.

- *J-CO-DS*. This component is a novel document store specifically designed to store large single documents

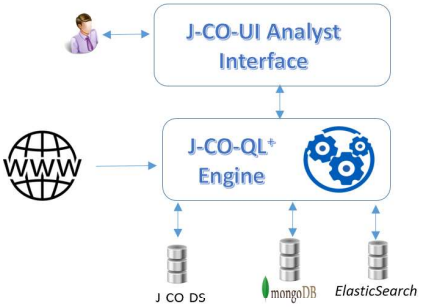

Figure 1: Former organization of the *J-CO* Framework.

(Psaila and Fosci, 2018), so as to overcome limitations of other *JSON* stores (such as *MongoDB*, which does not allow to store documents larger than 16Mb). *J-CO-DS* does not provide computational capabilities such as a query language, because it is an integral part of the *J-CO* Framework, so queries can be performed by means of *J-CO-QL*$^+$ and its engine.

- *J-CO-UI*. This is the user interface, by means of which analysts can write *J-CO-QL*$^+$ scripts, submit them to the *J-CO-QL*$^+$ *Engine* and inspect results.

3.2 Soft Querying Meteorological Data

J-CO-QL$^+$ was originally designed to natively deal with heterogeneous collections of *JSON* documents, which can be "geo-tagged" (i.e., describing spatial properties or entities through their geometries). Originally, its name was *J-CO-QL* (the reader can see various descriptions in (Bordogna et al., 2018; Bordogna et al., 2017a; Psaila and Fosci, 2021)); we decided to rename it as *J-CO-QL*$^+$ when we decided to redesign various statements and systematically introduce capabilities of "soft querying" (Fosci and Psaila, 2021b). Indeed, changes were necessary, so as to improve usability and clarity of semantics.

Listing 1 reports a script whose goal is to retrieve data about meteorological measurements provided by the Open-Data portal of Regione Lombardia (Regione Lombardia is the region which the cities of Milan and Bergamo are located in), so as to find days in which, in the province of Bergamo, the temperature at midday was high. Notice that the concept of "high temperature" is imprecise: for this reason, we exploit soft computing and fuzzy sets.

Hereafter, we explain the script in Listing 1.

Step 1. Defining a Fuzzy Operator
Line 1 of Listing 1 creates a "fuzzy operator", i.e., an operator that can evaluate the degree of membership of a document to a fuzzy set (Zadeh, 1965).

Very shortly, a "fuzzy set" A in U (where U is a universe) is a mapping $A : U \rightarrow [0,1]$: given an item

Listing 1: *J-CO-QL*$^+$: retrieval and soft querying.

```
1. CREATE FUZZY OPERATOR HighTemperatureFO
      PARAMETERS    temperature TYPE Float
      PRECONDITION  temperature >= -273
      EVALUATE      temperature
      POLYLINE      [ (0, 0.00), (20, 0.2), (25, 0.3),
                      (30, 0.5), (35, 0.8), (37, 0.9), (40, 1.0) ];

2. GET COLLECTION FROM WEB
      "https://www.dati.lombardia.it/resource/nf78-nj6b.json?
      $limit=100000";

3. EXPAND UNPACK WITH .data
      ARRAY .data TO .sensor;

4. SAVE AS sensorCollection;

5. GET COLLECTION FROM WEB
      "https://www.dati.lombardia.it/resource/647i-nhxk.json?
      $limit=100000000&$where= (stato='VV' OR stato='VA') AND
      data >= '2022-07-04' AND data < '2022-07-05'";

6. EXPAND UNPACK WITH .data
        ARRAY .data TO .value;

7. JOIN OF COLLECTIONS sensorCollection AS S, temporary AS V
      CASE WHERE  .S.sensor.item.idsensore  = .V.value.item.idsensore
                AND .S.sensor.item.tipologia  = "Temperatura"
                AND .S.sensor.item.provincia  = "BG"
        GENERATE
          CHECK FOR FUZZY SET HighTemperature
            USING HighTemperatureFO (TO FLOAT(.V.value.item.valore))
          ALPHACUT 0.75 ON HighTemperature
          BUILD {
              .province     : "Bergamo",
              .stationName  : .S.sensor.item.nomestazione,
              .sensorType   : "temperature",
              .sensorId     : .S.sensor.item.idsensore,
              .date         : .V.value.item.data,
              .temperature  : TO FLOAT(.V.value.item.valore),
              .latitude     : TO FLOAT (.S.sensor.item.lat),
              .longitude    : TO_FLOAT (.S.sensor.item.lng),
              .rank         : MEMBERSHIP_OF (HighTemperature)
          }
          DEFUZZIFY;

8. USE DB Webist2022  ON SERVER jcods 'http://127.0.0.1:17017';

9. SAVE AS HighTemperaturePlaces@Webist2022;
```

$x \in U$, $A(x) \in [0,1]$ is the membership degree of x to A; in other words, if $A(x) = 1$, then x fully belongs to A; if $A(x) = 0$, then x does not belong at all to A; if $0 < A(x) < 1$, then x partially belongs to A (the greater the value, the greater the way x belongs to A).
For example, given an a apartment, we could see that it is in the set of *Expensive Flats* if its price is greater than $600,000$ Euro, i.e., $Expensive\_Flats(a) = 1$. But what happens if the price is $360,000$ Euro? The membership degree could be $Expensive\_Flats(a) = 0.6$, meaning that the apartment is not too expensive.

The instruction on line 1 of the script in Listing 1 creates a fuzzy operator named `HighTemperatureFO`. It receives a formal parameter named `temperature` (as a floating-point number), whose value must be greater than $-273°C$ (as stated by the `PRECONDITION` clause). If the precondition is true, the `EVALUATE` clause says that the `temperature` parameter is used as x-axis value against the membership function defined by the `POLYLINE` clause; the function is depicted in Figure 2: if the x-axis value is less than 0 degrees, the corresponding 0 value is returned as membership degree; if the temperature is greater than 40, the corresponding 1 value is returned as membership degree.

Step 2. Getting Data about Sensors
In Listing 1, lines from 2 to 4 are devoted to acquire

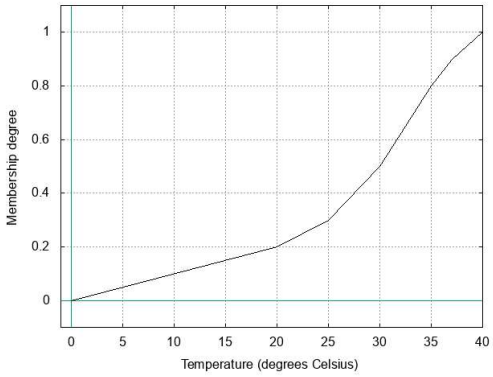

Figure 2: Membership function for the `HighTemperatureFO` fuzzy operator.

data from the Open-Data portal and prepare them for further processing.

Before going on, it is worth introducing the *Execution Model* of *J-CO-QL*$^+$ scripts.
A *J-CO-QL*$^+$ query (or script) is a sequence of instructions $q = (i_1, \ldots, i_n)$. Implicitly, instructions receive an input *query-process state* and generates a new one. A query-process state is defined as a tuple:

$$s = \langle tc, IR, DBS, FO \rangle.$$

The tc member is called "temporary collection": its goal is to store the collection of *JSON* documents processed by instructions. The *IR* member is called the "database of Intermediate Results": it is a database that is local to the execution process, to temporarily save collections, for further processing during the same query. The *DBS* member contains database descriptors, so as to enable connecting to databases. Finally, the FO member contains all fuzzy operators (see Step 1) defined throughout the query. Provided that the initial state is $s_0 = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ (i.e., all members are empty), the i_j instruction takes the $s_{(j-1)}$ query-process state and generates a new s_j query-process state.

Coming back to the script, the `GET COLLECTION FROM WEB` instruction on line 2 sends an HTTP call to the end point that returns a data set about meteorological sensors. It generates a new temporary collection $s_2.tc$, with one single *JSON* document d; this document contains the `data` field, which is either an array of documents (if the Web source returns an array) or a single document (if the Web source returns one single document); in the example, an array of *JSON* documents is returned by the Open-Data portal. Other fields, in this lonely document, are `url` and `timestamp`, which denote the URL that provided the data and the time of the request. Figure 3 reports an example of document produced by the instruction. Notice that sub-documents in the `data` array, provided by the portal, have the field names in Italian, and all their values are reported as string values.

```
{
  "timestamp" : "2022-07-07T03:22:29.138",
  "url"       : "https://www.dati.lombardia.it/resource/
                 nf78-nj6b.json?$limit=100000"
  "data"      : [
    {
      "idsensore"     : "2433",
      "idstazione"    : "132",
      "lat"           : "45.66051928748766",
      "lng"           : "9.658768470289832",
      "nomestazione"  : "Bergamo v.Stezzano",
      "provincia"     : "BG",
      "tipologia"     : "Temperatura",
      … uninteresting other fields …
    },
    … { other sensor documents }, …
    {
      "idsensore"     : "11654",
      "idstazione"    : "132",
      "lat"           : "45.66051928748766",
      "lng"           : "9.658768470289832",
      "nomestazione"  : "Bergamo v.Stezzano",
      "provincia"     : "BG",
      "tipologia"     : "Velocita  Vento",
      … uninteresting other fields …
    }
  ]
}
```

Figure 3: Example of document after Line 2 in Listing 1.

```
{
  "timestamp" : "2022-07-07T03:22:29.138",
  "url"       : "https://www.dati.lombardia.it/resource/
                 nf78-nj6b.json?$limit=100000"
  "sensor"    : {
   "position"  : 662
   "item"      : {
     "idsensore"     : "2433",
     "idstazione"    : "132",
     "lat"           : "45.66051928748766",
     "lng"           : "9.658768470289832",
     "nomestazione"  : "Bergamo v.Stezzano",
     "provincia"     : "BG",
     "tipologia"     : "Temperatura",
     … uninteresting other fields …
   }
  }
}
```

Figure 4: Example of document after Line 3 in Listing 1.

Line 3 has to extract *JSON* documents contained in the `data` field of the single document in the temporary collection. The `EXPAND` instruction does that: it generates a new collection with as many documents as the items in the `data` array. Each document contains all the root-level fields (i.e., `url` and `timestamp`) and the `sensor` field; in turn, this field has the `item` field, with the document unnested from within the array, and the `position` field, which denotes the position occupied by the item in the original array. Figure 4 reports a document produced by the instruction.

Now, one document for each sensor is in the output temporary collection. Line 4 saves it into the *IR* database (named `sensorCollection`) for later use.

Step 3. Getting Meteorological Data

Lines 5 and 6 are responsible to get meteorological data from the Open-Data portal.

The `GET COLLECTION FROM WEB` instruction on line 5 calls the dedicated end point of the Open-Data portal and generates a new temporary collection. Remember that only one single document is generated, with the `data` array field containing all the returned documents. Figure 5 reports an example of document produced by the instruction.

Consequently, this single document is expanded

```
{
  "timestamp" : "2022-07-07T03:22:32.136",
  "url"       : "https://www.dati.lombardia.it/resource/
                 647i-nhxk.json?$limit=100000000&
                 $where=data%20%3E=%20'2022-07-04'%20
                 AND%20data%20%3C%20'2022-07-05'%20
                 AND%20(stato='VV'%20OR%20stato='VA')%20
                 AND%20idoperatore%3C%3E'4'",
  "data" : [
    {
      "data"        : "2022-07-04T00:00:00.000",
      "idoperatore" : "1",
      "idsensore"   : "11827",
      "stato"       : "VA",
      "valore"      : "323"
    },
    … { other measurements documents } …
    {
      "data"        : "2022-07-04T00:00:00.000",
      "idoperatore" : "3",
      "idsensore"   : "14213",
      "stato"       : "VA",
      "valore"      : "4.8"
    }
  ]
}
```


Figure 5: Example of document after Line 5 in Listing 1.


```
{
  "timestamp" : "2022-07-07T03:22:32.136",
  "url"       : "https://www.dati.lombardia.it/resource/
                 647i-nhxk.json?$limit=100000000&
                 $where=data%20%3E=%20'2022-07-04'%20
                 AND%20data%20%3C%20'2022-07-05'%20
                 AND%20(stato='VV'%20OR%20stato='VA')%20
                 AND%20idoperatore%3C%3E'4'""data",
  "value"     : {
   "position"  : 62263
   "item"      : {
     "data"        : "2022-07-04T00:00:00.000",
     "idoperatore" : "1",
     "idsensore"   : "11827",
     "stato"       : "VA",
     "valore"      : "323"
    }
  }
}
```


Figure 6: Example of document after Line 6 in Listing 1.

by the `EXPAND` instruction on line 6, to have one single document for each single measurement. Figure 6 reports an example of resulting document.

Step 4. Soft Querying Temperatures

Line 7 of the script has a twofold goal: (i) associating meteorological measurements with the sensors that performed them, so as to consider only sensors in the wished province, focusing only on temperature measurements; (ii) soft querying measurements to look for high temperatures. Both these tasks are performed by the `JOIN` instruction.

(i) The `sensorCollection`, previously saved by line 4 into the database of Intermediate Results (here aliased as `S`, since it describes sensors) is joined with the temporary collection (aliased as `V`, since it describes measured values).

(ii) For each pair of documents s (from the `S` collection) and v (from the `V` collection), a new d document is generated, with two fields: d.`S` contains the source s document; d.`V` contains the source v document.

(iii) The `WHERE` condition selects only d documents of interest. Specifically, a d document is selected if it associates a v measure with the s sensor that actually performed it; furthermore, only measurements made by temperature sensors are selected, which were made by sensors in the province of Bergamo (strangely, the Open-Data portal provides the province in which sensors are located in but not the city/municipality). Notice that field names and values are in Italian.


```
{
    "date"         : "2022-07-04T14:30:00.000",
    "latitude"     : 45.6605192874877,
    "longitude"    : 9.65876847028983,
    "province"     : "BG",
    "rank"         : 0.830000001192093,
    "sensorId"     : "2433",
    "stationName"  : "Bergamo v.Stezzano",
    "temperature"  : 35.6
}
```


Figure 7: Example of document after Line 7 in Listing 1.

(iv) The `GENERATE` block further processes the selected documents. In particular, the `CHECK FOR FUZZY SET` clause performs the soft query on the selected documents. Specifically, it evaluates the membership degree of the document to the fuzzy set named `HighTemperature`. The evaluation is done by the `USING` soft condition, by calling the `HighTemperatureFO` fuzzy operator.
The membership degree is described by a special field (here added to the d document), called `~fuzzysets`. In it, field names denote fuzzy-set names, and their value is a membership degree (thus, the membership degrees to many fuzzy sets can be represented).
The `ALPHACUT` option discards documents with membership degree to the `HighTemperature` fuzzy set less than 0.7 (so as to deal with imprecise definition of the concept of "high temperature").

(v) The final `BUILD` block restructures output documents. Specifically, field names are converted to English, numerical values reported as string values are converted to floating-point numbers and the novel `rank` field is added, with the membership degree to the `HighTemperature` fuzzy set. This latter field could denote the relevance of the output document with respect to the desired search. The `DEFUZZIFY` keyword removes the special `~fuzzysets` field (documents leave the domain of fuzzy sets).

Figure 7 reports an example of document produced by the `JOIN` instruction.

The script ends by saving the temporary collection into a database. We created a database called `Webist2022` managed by *J-CO-DS* (see Section 3.1). Line 8 connects to the database, while line 9 saves the temporary collection into the database with name `HighTemperaturePlaces`.

4 VISION

As stated in Section 2, a uniform vision of Web Intelligence is missing: different problems ask for different solutions. This paper focuses on scopes where *JSON* data sets are collected from Web sources and queried to possibly discover knowledge. We identified a general vision for this scope (in Figure 8), which is the basis for achieving Soft Web Intelligence.

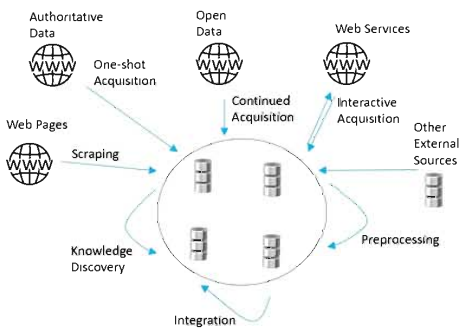

Figure 8: Vision of Web Intelligence.

Data Storage. To perform complex integration and analysis of data coming from the Web, a support for data storage is essential, so as to collect data and results of analysis. In Figure 8, the circle is the *Data-Storage Area*, which encompasses all the data-storage systems involved in the process.

Web Sources. In an environment for Web Intelligence, many Web sources could provide data. We identified four distinct categories.

- **Authoritative Data**. Public Administrations and Authorities often publish data that can be considered "immutable" or slowly mutable, such as city borders. Usually, these data sets are downloaded only once, due to their immutability.
- **Open-Data Portals**. Open-Data portals distribute data produced by public administrations that concern the administered territory. Often, these data sets change continuously (as in the case of meteorological measurements); to cope with this volatility, it is necessary to perform a "continued acquisition" of these data sets, to accumulate all versions that change in time.
- **Web Services**. To perform analyses, often data must be acquired from Web Services, through HTTP calls on the basis of previously-acquired single data items. For example, having a pool of "locations", denoted by latitude and longitude, a "geo-coding" service could provide the corresponding address. One call for each single item is necessary (thus, we used bi-directional arrows and the "interactive acquisition" label in Figure 8).
- **Scraping**. Web pages (formatted as HTML pages) can provide valuable information; in this case, "scraping tools" could extract semi-structured descriptions of Web pages.

External Data. Any kind of data not directly coming from Web sources can give valuable information.

Processing Tasks. Many tasks could be guessed. Figure 8 reports the ones we consider most relevant (without excluding any other processing task).

- **Pre-processing**. Once downloaded, a data set usually needs to be pre-processed, so as to clean it and to make it suitable for further processing.
- **Integration**. Once data sets are collected, they could be integrated, so as to put together all relevant information into one single data set.
- **Knowledge Discovery**. Once data sets to analyze have been built, it is possible to perform complex tasks of "knowledge discovery". Here, a plethora of techniques can be used; anyway, we do not use the terms "data mining" and "machine learning", because we consider a general approach to knowledge discovery, not only the ones coming from the area of Machine Learning.

5 PROPOSED EXTENSIONS

In (Fosci and Psaila, 2021a), we presented how the introduction of constructs able to support the evaluation of fuzzy sets on *JSON* documents could allow *J-CO* users to perform complex knowledge-discovery tasks, by directly accessing an Open-Data portal to get data sets describing levels of pollutants and related sensors networks. In practice, (Fosci and Psaila, 2021a) shows how soft querying *JSON* data sets could help analysts in discovering unexpected situations.

On the basis of the vision (Figure 8) explained in Section 4, we studied how far the *J-CO* Framework was to be able to fully support our vision of Web Intelligence. We identified the following limitations.

No Continued Acquisition. *J-CO-QL*$^+$ can connect to Web Services and portals via the HTTP protocol, to get data. However, this can be done only when the script is executed. A continued acquisition of *JSON* documents from Web sources (i.e., repeated over time to capture changing data sets) is not possible.

Only One-shot Acquisition in *J-CO-QL*$^+$. Even though *J-CO-QL*$^+$ can get collections from Web sources, this can be done only in a one-shot way. In contrast, *J-CO-QL*$^+$ is not suitable if it is necessary to contact a Web Service on the basis of data contained in previously collected collections, so as to collect a pool of *JSON* documents provided by a Web Service through many HTTP calls.

No Script Library. Although the *J-CO-QL*$^+$ *Engine* provides the "batch mode", i.e., it is possible to launch the execution of an entire script, there is no way to manage libraries of scripts in a structured way, making them parametric, so as to reuse scripts.

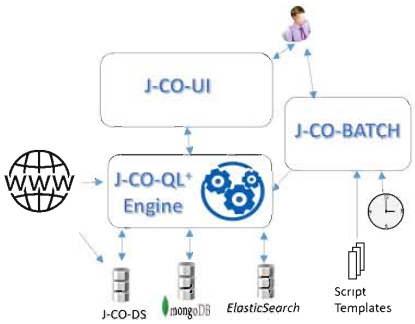

Figure 9: New organization of the *J-CO* Framework.

Based on the previous analysis of limitations, we identified novel features to add, so as to make the framework suitable for Web-Intelligence scopes.

Dynamic and Virtual Collections. For *J-CO-DS*, we conceived three different types of collections within a database. **Static Collections** are the classical collections. The novel **Dynamic Collections** are collections whose content is dynamically and periodically filled in by *J-CO-DS* by calling a pool of Web sources (so, continued gathering of *JSON* documents from Web sources can be realized). The novel **Virtual Collections** are collections whose content is not materialized within the database but it is obtained on the fly by calling Web sources when the collection is accessed.

Parametric Script Templates. In order to create script libraries, we defined a meta-language (with respect to *J-CO-QL*$^+$) that allows for defining parameters within *J-CO-QL*$^+$ scripts; we took inspiration from the classical idea of "macro-substitution".

Batch Execution of Parametric Scripts. A meta-engine able to collect script templates can support the creation of script-template libraries and support the repeated execution of them. Furthermore, this meta-engine could be invoked by external services to execute a template, as well as it could execute scheduled batches that launch script templates.

Repeated HTTP Calls. The final novelty we envisioned is a novel statement to add to *J-CO-QL*$^+$, so as to allow multiple calls of a Web Service on the basis of field values in *JSON* documents contained within a collection (so as to compose URLs). The novel statement has been called `LOOKUP FROM WEB`.

Hereafter, we will show the extensions we made to the *J-CO* Framework.

5.1 Collection Types in *J-CO-DS*

In order to allow for "continued gathering" of *JSON* documents from Web sources, we conceived two novel types of collections to be managed by *J-CO-DS*. After this extension, three different types of collections are available.

Static collections are the classical collections managed by *J-CO-DS*, i.e., created and possibly updated by the user, as well as created by a *J-CO-QL*$^+$ script.

Dynamic collections are thought for continued acquisition; they are created specifying a pool of Web sources; the content of the collection is periodically updated with novel documents.

Let us define a dynamic collection in more details.

- A "dynamic-source descriptor" is a tuple

$$dsd = \langle name, URL, startTime, frequency, mode \rangle$$

in which *name* is the name provided by the user to characterize the source. The URL member is the URL to contact to get documents. The $startTime$ member is the time when to start the acquisition. The $frequency$ member is the frequency with which gathering must be repeated; admitted values can be expressed in hours (e.g., $2h$), days (e.g., $4d$) or weeks (e.g., $5w$). Finally, the *mode* member is the update mode of content; its value can be either `"replace"`, i.e., the full content of the collection is replaced from scratch, or `"append"`, i.e., new documents are appended to those already present in the collection.

- A dynamic collection *dc* is described by the tuple

$$dc = \langle name, sources \rangle$$

where *name* identifies the collection, while *sources* is a non-empty array of source descriptors.

Different sources could be accessed, with different frequency and update mode, as clarified hereafter.

- Data collected from each source are managed in a dedicated storage area, so as to be managed independently of data coming from the other sources.

- A Web source usually provides either a single *JSON* document, or a *JSON* array (usually an array of documents, but not necessarily).

For each single request sent to the Web source that returns data (either a single document or an array), a *JSON* document is created, with the following fields: `source` is the name of the source (the *name* field in the *dsd* source descriptor); `url` is the contacted URL; `timestamp` is the acquisition time; `data` actually contains the returned data (either a single *JSON* document or an array). This new document is inserted into the data space of the web source.

- In the case the `"replace"` update mode is selected, the previous content of the data space is cleaned and the new document is added. Thus, only one document at a time is present in the data space.

In the case the `"append"` update mode is selected, the new document is added to the previous ones possibly present in the data space of the Web source.

The rationale behind the two update modes is the following. For continued acquisition, clearly the best solution is the "append" mode, because it allows for recording the history of data sets returned by the source; in case the different versions of data sets have overlapping content, a pre-processing activity must be performed, through dedicated *J-CO-QL*$^+$ batches, to remove duplicates, if necessary.
The usefulness of the "replace" mode is to decouple *J-CO-QL*$^+$ batches and scripts from the Web source and the intermediate network connection that provide the data to process: this way, data can be easily processed, independently of the availability of the Internet connection or of the Web source.

Virtual collections provide a unified abstraction to many Web sources. They are defined by specifying a pool of URLs, which provide collections of *JSON* documents. When a user or a *J-CO-QL*$^+$ script reads the virtual collection, the actual content is obtained by calling all associated sources on the fly.
In virtual collections, a source descriptor is a tuple

$$vsd = \langle name, URL \rangle$$

which has the *name* and *URL* members only.
A virtual collection is, in turn, defined by the tuple

$$vc = \langle name, sources \rangle$$

where *name* identifies the collection, while *sources* is a non-empty array of virtual-source descriptors *vsd*.

As a general constraint, a *J-CO-QL*$^+$ SAVE AS instruction issued either to a dynamic or a virtual collection is not allowed, because it would destroy collection descriptors in the database, loosing the precious information about Web sources and URLs. Furthermore, in the case of dynamic collections with "append" update mode, the history of collected data would be lost as well.

5.2 J-CO-Batch

J-CO-BATCH is a completely-new component introduced in the *J-CO* Framework. Its goal is to provide a "batch execution" of *J-CO-QL*$^+$ scripts. Figure 9 shows how the organization of the framework has changed with the new component. These are its functionalities.

Macro-Substitution. *J-CO-QL*$^+$ scripts can be parameterized. A "macro-parameter" has the lexical structure ##*name*##, since no lexical element in *J-CO-QL*$^+$ has the same structure.
A *J-CO-QL*$^+$ script that contains macro-parameters is called "*J-CO-QL*$^+$ script template".
The *Template pre-processor* is the internal component that, provided a context C as a key/value map, and a t template, performs a macro-substitution of all macro-parameters $mp \in t$ with the corresponding value $C(mp)$; if all macro-parameters in t have a corresponding value in C, the t template is actually transformed into an executable script.
Only constant values (either numbers or strings), identifiers and path-expressions are valid values for macro-parameters in C.

Batch Execution. Users can launch the execution of templates in batch mode, by providing the context (currently provided as a "property file"). We plan to define a high-level language to specify contexts, possibly through a dedicate user interface.

Scheduled Batch Execution. Another functionality is scheduling batch execution. We considered two types: "one shot", in which the template is executed at a predefined instant; "repeated", when a template is executed multiple times.

5.3 LOOKUP FROM WEB

The novel LOOKUP FROM WEB statement extends *J-CO-QL*$^+$, so as to give scripts the capability to perform multiple HTTP calls to Web Services. The idea is the following: suppose that a Web Service *WS* provides one single *JSON* document as reply to a request, in such a way the content of the returned *JSON* document depends on some parametric data. Given a collection C of *JSON* documents, such that documents $d \in C$ provide the data to send to *WS*, we might want to collect all documents returned by *WS* by calling it for each document $d \in C$. This is the goal of the LOOKUP FROM WEB statement.

Syntax and Semantics. The syntax of the statement is as follows.

LOOKUP FROM WEB
  (FOR EACH *Condition* CALL *stringExpression*)$^+$

After the LOOKUP FROM WEB keywords, a non-empty list of FOR EACH clauses constitutes the statement. The FOR EACH keywords are followed by a condition; after that, the CALL sub-clause is followed by a string expression representing a URL.

The semantics of the statement is as follows.

(i) The statement works on the tc temporary collection provided by the query-process state.

(ii) For each document $d \in tc$, the condition in the first FOR EACH branch is evaluated; if it is true, d is processed accordingly to the CALL sub-clause; otherwise, the condition in the next FOR EACH branch is evaluated, and so on.

(iii) If the d document meets a condition in a FOR EACH branch, an HTTP call is performed to the URL

specified by the associated `CALL` sub-clause. This is followed by an expression that provides the URL to contact, which can refer to fields in the d document, so as to use data in d to compose the URL.

(iv) The *JSON* documents (in principle, more than one document could be returned by the contacted Web Service) are assembled with the source d document. Specifically, if the HTTP call succeeds, a new $\overline{d}$ document is built. Its fields are: `source` contains the source d document; `data` contains the array of documents returned by the Web Service; `url` specifies the URL contacted to obtain the returned documents; `timestamp` contains the time-stamp of the call.

(v) The final output temporary collection $\overline{tc}$ contains all the output $\overline{d}$ documents. If no condition in the `FOR EACH` branches is true for d, the $\overline{d}$ document is generated without the `data` field.

6 AN EXAMPLE

In section 3.2 we showed how the *J-CO-QL*$^+$ script reported in Listing 1 is able to soft query meteorological data to discover high-temperature events in a predefined province. This could be considered a first example of Soft-Web-Intelligence task based on *JSON* data sets. However, the script also shows the limitations of the *J-CO* Framework that inspired the proposed extensions. We discuss these limitations.

(i) The data set that describes sensors reports the name of the province where sensors are located in, but not the city/municipality. Since the data set provides coordinates (latitude and longitude) of sensors, one could think to exploit Web Services, such as the one provided by *GeoNames*[2] that, provided the coordinates, gives the names of the administrative entities that contain a given location. The need to exploit such a kind of service inspired the definition of the `LOOKUP FROM WEB` statement (Section 5.3).

(ii) The end point of the Open-Data portal that provides meteorological data continuously updates (every 6 hours) the data set; but when the current month changes, data about the previous month are removed. This observation inspired the need for continued acquisition, which gave us the idea of introducing "dynamic collectons" in *J-CO-DS* (Section 5.1).
Nevertheless, since dynamic collections save images of data sets returned by the end point, we argued that it was necessary to periodically schedule pre-processing activities on the data sets. This consideration gave us the idea of developing *J-CO-BATCH* (Section 5.2).

[2]GeoNames URL: https://www.geonames.org/export/web-services.html

Listing 2: Script pre-processing meteorological data.

```
1. USE DB Webist2022 ON SERVER jcods 'http://127.0.0.1:17017';

2. GET COLLECTION LastMeteoValue@Webist2022;

3. EXPAND
     UNPACK WITH .data
       ARRAY .data TO .value;

4. FILTER
     CASE WHERE WITH .value.item
       GENERATE
         BUILD {
           .timestamp    : .value.item.data,
           .idSensor     : .value.item.idsensore,
           .sensorState  : .value.item.stato,
           .sensorValue  : TO_FLOAT (.value.item.valore)
         };

5. MERGE COLLECTIONS temporary, valueCollection@Webist2022
     REMOVE DUPLICATES;

6. SAVE AS AllMeteoValues@Webist2022;
```

(iii) Which is the city of interest? What is the time window of interest? Clearly, the code of the script is the same, apart from the specific city name and time window. We understood that there was a problem concerning "reusability" of *J-CO-QL*$^+$ scripts. This consideration inspired "script templates", that are supported by *J-CO-BATCH* (Section 5.2).

We can now present the Soft-Web-Intelligence task, to illustrate the approach.

Step 1. Defining a Dynamic Collection Gathering Meteorological Data
In order to continuously track the latest version of the data set with meteorological data, we created a dynamic collection called `LastMeteoValues` in the `Webist2022` database managed by *J-CO-DS*. We selected the `"replace"` update mode, because we are not interested in tracking the history of versions (the update frequency of 4 updates a day is too high). The Web source has been configured in such a way the acquisition is performed at 1am, 7am, 1pm and 7pm (Italian time), so as to be sure that the novel version of the data set is available on the portal.
Clearly, it is necessary to pre-process this collection, so as to have a consolidated version with all measurements, including those performed before the current month (see Step 2).

In the `Webist2022` database managed by *J-CO-DS*, we created a "virtual collection" named `Sensors`. It is associated with the same Web source exploited in Listing 1 to get data about meteorological sensors.
We defined this virtual collection to keep knowledge about all possible Web sources that provide data about meteorological sensors. The idea is to consider several Open-Data portals (for example, of different regions). The final effect is that, this way, the Web is seen as a kind of "giant dispersed collection of data", in which all data sets that are reasonably homogeneous (i.e., describe meteorological sensors) can be accessed all together by a *J-CO-QL*$^+$ script.

```
{
  "idSensor"   : "12759",
  "state"      : "VA",
  "timestamp"  : "2022-07-04T00:00:00.000",
  "value"      : 26.2
}
```

Figure 10: Example of document after Line 4 in Listing 2.

Step 2. Repeated Scheduled Pre-processing
The script reported in Listing 2 actually pre-processes the last image of the meteorological measurements acquired by *J-CO-DS* and saved into the `LastMeteoValues` dynamic collection. The script merges measurements in the `LastMeteoValues` collection with those in the `AllMeteoValues` collection. This latter one is a static collection, which is both the input and the output of the script. In details:

- Line 1 connects the script to the `Webist2022` database managed by *J-CO-DS*.

- Line 2 gets the `LastMeteoValues` dynamic collection, which gathers the last image of meteorological data. It becomes the current temporary collection.

- The `EXPAND` instruction on line 3 unnests documents from within the `data` array field of the single document contained in the temporary collection (coming from the `LastMeteoValues` dynamic collection).

- The `FILTER` instruction restructures documents in the temporary collection (one for each measurement), so as to convert field names to English, strings containing numbers into numeric values, as well as to flatten the structure. Figure 10 reports an example of document produced by the instruction.

- The `MERGE` instruction on line 5 actually merges the temporary collection with the old version of the `AllMeteoValues` static collection, stored within the `Webist2022` database. Notice the `REMOVE DUPLICATES` option, so as to obtain a new temporary collection without duplicates (every single measurement must appear only once in the collection).

- Finally, the last temporary collection becomes the new version of the `AllMeteoValues` static collection (it is actually saved into the database).

This script is scheduled to be executed every day at 3am, 9am, 3pm and 9pm (so as to be sure that the last version of the `LastMeteoValues` dynamic collection has been acquired and saved).

Step 3. Template for Soft Querying Data
Users (analysts) who wish to find out high-temperature events in a given city and in a given time window can launch a batch that executes the *J-CO-QL*$^+$ script template reported in Listing 3. This template directly derives from the script in Listing 1; however, now it has been adapted to the new general context: it is able to acquire and select cities/municipalities whose name has been parameterized (Listing 4 reports parameter values). We present it in more details.

Listing 3: *J-CO-QL*$^+$ template.

```
1. CREATE FUZZY OPERATOR HighTemperatureFO
      PARAMETERS    temperature TYPE Float
      PRECONDITION  temperature >= -273
      EVALUATE      temperature
      POLYLINE      [ (0, 0.00), (20, 0.2), (25, 0.3),
                      (30, 0.5), (35, 0.8), (37, 0.9), (40, 1.0) ];

2. USE DB Webist2022 ON SERVER jcods 'http://127.0.0.1:17017';

3. GET COLLECTION Sensors@Webist2022;

4. EXPAND UNPACK WITH .data
      ARRAY .data TO .sensor;

5. FILTER
      CASE WHERE .sensor.item.tipologia = "Temperatura"
             AND .sensor.item.provincia = "BG";

6. LOOKUP FROM WEB
      FOR EACH WITH .sensor.item.lat, .sensor.item.lat
        CALL "http://api.geonames.org/findNearbyPlaceNameJSON?"
             + "formatted=false&lat=" + .sensor.item.lat
             + "&lng=" + .sensor.item.lng + "&fclass=P&fcode=PPLA"
             + "&fcode=PPL&fcode=PPLC&username=paolofosci&style=full";

7. EXPAND UNPACK WITH .data.geonames
      ARRAY .data.geonames TO .geoname;

8. JOIN OF COLLECTIONS temporary AS S, AllMeteoValues@Webist2022 AS V
      CASE
        WHERE .S.source.sensor.item.idsensore = .V.idSensor
          AND WITH .S.geoname.item.adminName3
          AND .V.timestamp                > ##dateMin##
          AND .V.timestamp                < ##dateMax##
          AND .S.geoname.item.adminName3 = ##city##
        GENERATE
          CHECK FOR FUZZY SET HighTemperature
            USING HighTemperatureFO (.V.value)
          ALPHACUT ##threshold## ON HighTemperature
          BUILD {
            .sensorId     : .S.source.sensor.item.idsensore,
            .stationName  : .S.source.sensor.item.nomestazione,
            .temperature  : .V.value,
            .latitude     : TO_FLOAT (.S.source.sensor.item.lat),
            .longitude    : TO_FLOAT (.S.source.sensor.item.lng),
            .province     : .S.source.sensor.item.provincia,
            .city         : .S.geoname.item.adminName3,
            .date         : .V.timestamp,
            .rank         : MEMBERSHIP_OF (HighTemperature)
          }
          DEFUZZIFY;

9. SAVE AS HighTemperaturePlaces@webist2022;
```

Listing 4: Property file for parameters in Listing 3.

```
dateMin   = "2022-07-03"
dateMax   = "2022-07-05"
threshold = 0.75
city      = "Stezzano"
```

- Line 1 is identical to the same line in Listing 1: it creates the `HighTemperatureFO` fuzzy operator.

- Line 2 connects to the `Webist2022` database managed by *J-CO-DS*.

- Line 3 retrieves the `Sensors` virtual collection from the database. Remember that this collection returns one single document, because currently only one Web source is configured.

- Line 4 unnests the documents nested within the `data` array field.

- Line 5 selects documents describing temperature sensors in the province of Bergamo.

- Line 6 exploits the novel `LOOKUP FROM WEB` statement: the goal is to associate each sensor with the name of the city/municipality where it is located in. For this reason, a specific Web Service provided by

```
{
  "timestamp" : "2022-07-07T18:19:25.713",
  "url"       : "http://api.geonames.org/findNearbyPlaceNameJSON?
                 formatted=false&lat=45.66051928748766
                 &lng=9.658768470289832&fclass=P&fcode=PPLA
                 &fcode=PPL&fcode=PPLC&username=paolofosci&style=full",
  "source"    : {
    "sensor"    : {
      "timestamp" : "2022-07-07T18:19:23.617",
      "url" : "https://www.dati.lombardia.it/resource/
             nf78-nj6b.json?$limit=100000",
      "item"      : {
        "idsensore"     : "2433",
        "idstazione"    : "132",
        "lat"           : "45.66051928748766",
        "lng"           : "9.658768470289832",
        "nomestazione"  : "Bergamo v.Stezzano",
        "provincia"     : "BG",
        "tipologia"     : "Temperatura",
          … uninteresting other fields …
      },
      "position" : NumberLong(662)
    }
  },
  "data"      : {
    "geonames"  : [
      {
        "adminId1"    : "3174618",
        "adminId2"    : "3182163",
        "adminId3"    : "6541671",
        "adminName1"  : "Lombardy",
        "adminName2"  : "Provincia di Bergamo",
        "adminName3"  : "Stezzano",
          … uninteresting other fields …
      }
    ]
  }
}
```

Figure 11: Example of document after Line 6 in Listing 3.

```
{
  "timestamp" : "2022-07-07T18:19:25.713",
  "url"       : "http://api.geonames.org/findNearbyPlaceNameJSON?
                 formatted=false&lat=45.66051928748766
                 &lng=9.658768470289832&fclass=P&fcode=PPLA
                 &fcode=PPL&fcode=PPLC&username=paolofosci&style=full",
  "source"    : {
    "sensor"    : {
      "timestamp" : "2022-07-07T18:19:23.617",
      "url" : "https://www.dati.lombardia.it/resource/
             nf78-nj6b.json?$limit=100000",
      "item"      : {
        "idsensore"     : "2433",
        "idstazione"    : "132",
        "lat"           : "45.66051928748766",
        "lng"           : "9.658768470289832",
        "nomestazione"  : "Bergamo v.Stezzano",
        "provincia"     : "BG",
        "tipologia"     : "Temperatura",
          … uninteresting other fields …
      },
      "position" : NumberLong(662)
    }
  },
  "data"      : {},
  "geoname" : {
   "position" : 1,
   "item" : {
     "adminId1" : "3174618",
     "adminId2" : "3182163",
     "adminId3" : "6541671",
     "adminName1" : "Lombardy",
     "adminName2" : "Provincia di Bergamo",
     "adminName3" : "Stezzano",
     … uninteresting other fields …
   }
  }
}
```

Figure 12: Example of document after Line 7 in Listing 3.

GeoNames is contacted, by sending an HTTP call. Notice how the URL is composed by exploiting fields in the documents contained in the input temporary collection. Figure 11 reports an example of document produced by the `LOOKUP FROM WEB` instruction.
Unfortunately, GeoNames returns an array of documents with several names: they are the municipality name, the province and the region. Thus, it is necessary to expand this array, so as to be able to refer to the desired name. This unnesting is done by the `EXPAND` instruction on line 7. Figure 12 reports an example of document produced by line 7.

- The `JOIN OF COLLECTIONS` instruction is similar,

```
{
    "city"         : "Stezzano",
    "date"         : "2022-07-04T14:30:00.000",
    "latitude"     : 45.6605192874877,
    "longitude"    : 9.65876847028983,
    "province"     : "BG",
    "rank"         : 0.830000001192093,
    "sensorId"     : "2433",
    "stationName"  : "Bergamo v.Stezzano",
    "temperature"  : 35.6
}
```

Figure 13: Example of document after Line 8 in Script 3.

but not identical, to the same instruction in Listing 1: it pairs documents describing sensors and documents describing measurements; the former are in the input temporary collection (aliased as `S`), the latter are in the `AllMeteoValues` static collection, which is generated by the scheduled script in Listing 2.
Compared with Listing 1, now documents that describe sensors have the `.S.geoname.item.adminName3` field, which denotes the name of the municipality; consequently, it is possible to select only temperature measurements performed in the city of interest.
Notice the three macro-parameters used in the `WHERE` selection condition (depicted in bold face for clarity): this way, the template is independent of the city name and of the time window of interest.
The `GENERATE` block behaves almost as in Listing 1: a macro-parameter for the `ALPHACUT` threshold is used; the output documents now have the `city` field.
Figure 13 reports an example of output document.

7 CONCLUSIONS

The paper proposes our vision towards Soft Web Intelligence, i.e., an application scope of Web Intelligence in which it is necessary to acquire, integrate and analyze *JSON* data sets from Web sources, by exploiting soft querying capabilities provided by a stand-alone tool. The tool is the *J-CO* Framework, a research project of University of Bergamo (Italy), whose goal is to provide analysts with a high-level and platform-independent suite for manipulating *JSON* data sets. The paper shows the capabilities provided by the *J-CO* Framework before this work; then, our vision of Web-Intelligence scopes is presented and extensions necessary to achieve the vision are presented through an example (enabled by the presented extensions). Now the *J-CO* Framework can be actually exploited for Soft Web Intelligence.

As future work, we will pursuit two main evolution lines. First, we identified the problem of uniformly defining soft aggregations and managing multiple types of fuzzy sets (for example, Intuitionistic Fuzzy Sets and Type-2 Fuzzy Sets). Second, we will develop novel software tools, in particular user inter-

faces, to foster usability of the framework.

REFERENCES

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., Ciriello, D. E., and Psaila, G. (2017a). A flexible framework to cross-analyze heterogeneous multi-source geo-referenced information: the j-co-ql proposal and its implementation. In *Proceedings of the International Conference on Web Intelligence*, pages 499–508. ACM.

Bordogna, G., Cuzzocrea, A., Frigerio, L., Psaila, G., and Toccu, M. (2017b). An interoperable open data framework for discovering popular tours based on geotagged tweets. *International Journal of Intelligent Information and Database Systems*, 10(3-4):246–268.

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

Devedžić, V. (2004). Web intelligence and artificial intelligence in education. *Journal of Educational Technology & Society*, 7(4):29–39.

Fosci, P. and Psaila, G. (2021a). Powering soft querying in j-co-ql with javascript functions. In *International Workshop on Soft Computing Models in Industrial and Environmental Applications*, pages 207–221. Springer, Cham.

Fosci, P. and Psaila, G. (2021b). Towards flexible retrieval, integration and analysis of json data sets through fuzzy sets: a case study. *Information*, 12(7):258.

Han, J. and Chang, K.-C. (2002). Data mining for web intelligence. *Computer*, 35(11):64–70.

Kacprzyk, J. and Zadrożny, S. (2010). Soft computing and web intelligence for supporting consensus reaching. *Soft Computing*, 14(8):833–846.

Molnár, E., Molnár, R., Kryvinska, N., and Greguš, M. (2014). Web intelligence in practice. *Journal of Service Science Research*, 6(1):149–172.

Negash, S. and Gray, P. (2008). Business intelligence. In *Handbook on decision support systems 2*, pages 175–193. Springer.

Pelucchi, M., Psaila, G., and Toccu, M. (2017a). Building a query engine for a corpus of open data. In *WEBIST-2022*, pages 126–136.

Pelucchi, M., Psaila, G., and Toccu, M. (2017b). Enhanced querying of open data portals. In *Int. Conf. on Web Information Systems and Technologies*, pages 179–201. Springer, Cham.

Pelucchi, M., Psaila, G., and Toccu, M. (2018). Hadoop vs. spark: Impact on performance of the hammer query engine for open data corpora. *Algorithms*, 11(12):209.

Poli, V. S. R. (2015). Fuzzy data mining and web intelligence. In *I. Conf. on Fuzzy Theory and Its Applications (iFUZZY)*, pages 74–79. IEEE.

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

Psaila, G. and Fosci, P. (2021). J-co: A platform-independent framework for managing geo-referenced json data sets. *Electronics*, 10(5):621.

Reddy, P. V. S. (2010). Fuzzyalgol: Fuzzy algorithmic language for designing fuzzy algorithms. *J. of Computer Science and Engineering*, 2(2):21–24.

Yao, Y., Zhong, N., Liu, J., and Ohsuga, S. (2001). Web intelligence (wi) research challenges and trends in the new information age. In *Asia-Pacific Conference on Web Intelligence*, pages 1–17. Springer.

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

Zadeh, L. A. (2004a). A note on web intelligence, world knowledge and fuzzy logic. *Data & Knowledge Engineering*, 50(3):291–304.

Zadeh, L. A. (2004b). Web intelligence, world knowledge and fuzzy logic–the concept of web iq (wiq). In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 1–5. Springer.

Zhang, Y.-Q. and Lin, T. Y. (2002). Computational web intelligence (cwi): synergy of computational intelligence and web technology. In *Wolrd Congress on Computational Intelligence.*, volume 2, pages 1104–1107. IEEE.

Zhong, N., Liu, J., Yao, Y., Wu, J., Lu, S., Qin, Y., Li, K., and Wah, B. (2006). Web intelligence meets brain informatics. In *I. Ws. on Web Intelligence Meets Brain Informatics*, pages 1–31. Springer.