Article

Towards Flexible Retrieval, Integration and Analysis of JSON Data Sets through Fuzzy Sets: A Case Study

Paolo Fosci and **Giuseppe Psaila** *

Department of Management, Information and Production Engineering, University of Bergamo, 24044 Dalmine, BG, Italy; paolo.fosci@unibg.it
* Correspondence: giuseppe.psaila@unibg.it; Tel.: +39-035-205-2355

Abstract: How to exploit the incredible variety of *JSON* data sets currently available on the Internet, for example, on Open Data portals? The traditional approach would require getting them from the portals, then storing them into some *JSON* document store and integrating them within the document store. However, once data are integrated, the lack of a query language that provides flexible querying capabilities could prevent analysts from successfully completing their analysis. In this paper, we show how the *J-CO* Framework, a novel framework that we developed at the University of Bergamo (Italy) to manage large collections of *JSON* documents, is a unique and innovative tool that provides analysts with querying capabilities based on fuzzy sets over *JSON* data sets. Its query language, called *J-CO-QL*, is continuously evolving to increase potential applications; the most recent extensions give analysts the capability to retrieve data sets directly from web portals as well as constructs to apply fuzzy set theory to *JSON* documents and to provide analysts with the capability to perform imprecise queries on documents by means of flexible soft conditions. This paper presents a practical case study in which real data sets are retrieved, integrated and analyzed to effectively show the unique and innovative capabilities of the *J-CO* Framework.

Keywords: retrieving JSON documents from the web; open data portals; fuzzy sets and soft selection conditions; platform-independent framework; case study

Citation: Fosci, P.; Psaila, G. Towards Flexible Retrieval, Integration and Analysis of JSON Data Sets through Fuzzy Sets: A Case Study. *Information* **2021**, *12*, 258. https://doi.org/10.3390/info12070258

Academic Editor: Elaheh Pourabbas

Received: 29 April 2021
Accepted: 7 June 2021
Published: 22 June 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Many people say that we are in the era of Big Data, but what does this mean exactly? Trying to characterize this claim, researchers have proposed various models, by means of which they tried to identify specific features concerning Big Data; typically, these features begin with "V" (such as "Volume", "Variety", "Velocity" and "Value"), so a plethora of *n*-Vs models are available in the literature (see, for example, the 7-Vs model proposed by [1]).

Focusing on "Variety", we can say that this is the first time that so many data sets are publicly available on the Internet. In this regard, Open Data portals have become typical channels through which public administrations publish authoritative data concerning the administered territory. For example, data about air quality and atmospheric measurements are published daily to allow citizens and analysts (in particular) to perform analyses on the quality of air.

JSON (JavaScript Object Notation) [2] has become the de facto standard format to represent data sets over the Internet. In fact, APIs (Application Programming Interfaces) of social media as well as of Open Data portals adopt this format very often to provide data sets. The reason is its simplicity and flexibility, because *JSON* documents are serialized representations of main-memory objects managed by object-oriented programming languages. Often, *JSON* documents are geo-referenced (or geo-tagged), i.e., they describe geographical entities located on the Earth globe, through spatial coordinates and geometries provided in documents.

Consequently, the Internet can be viewed as a giant source of *JSON* data sets, which might be cross-analyzed with each other and integrated with other data sets stored within *JSON* document stores (these are NoSQL databases managed by a novel family of DataBase Management Systems, or DBMSs, that natively store *JSON* documents); the results of analyses could be saved into *JSON* document stores too. However, this integration process could be cumbersome to carry on, possibly requiring computer programming skills: retrieving relevant data could be a tedious and frustrating activity due to the lack of a unique tool that provides the necessary capabilities.

At the University of Bergamo, we worked on a novel approach to integrate and cross analyze *JSON* data sets coming from web portals and document stores, enabled by the *J-CO* Framework. *J-CO* is a novel framework under development at the University of Bergamo with the goal to provide analysts with a unique tool (and related query language) for managing, integrating and querying possibly large *JSON* data sets. Currently, the *J-CO* Framework has become a platform-independent tool, able to manipulate *JSON* data sets independently from the specific computational capabilities provided by *JSON* document stores [3]. *J-CO-QL* (the query language of the *J-CO* Framework) provides high-level declarative instructions, which natively deal with geo-tagging; this way, spatial cross-analyses can be easily performed.

Nevertheless, analysts wishing to analyze *JSON* data sets coming from web sources could encounter significant obstacles caused by their imperfect knowledge of the data sets. In this regard, if they had the capability to perform imprecise and/or vague selections, they could be able to find the desired documents easily. Moving from past work [4] on flexible querying over relational databases, we extend *J-CO-QL* with constructs to evaluate the belonging of documents to fuzzy sets so as to express "customizable soft selection conditions" [5]. Although extension of the language with fuzzy constructs has not been completed, the current state is already effective in retrieving the desired *JSON* documents from data sets over the Internet by expressing imprecise and vague predicates so as to formulate soft selection conditions on *JSON* documents; the output of a soft query is expected to contain documents that are as close as possible to what looked for by analysts, even though there is not a perfect match.

The contribution of the paper is to show how the *J-CO* Framework can be effectively used to "query" web sources through fuzzy sets and soft conditions, customized by user needs. This way, we provide analysts with a powerful and unique tool to easily retrieve, integrate and analyze possibly large *JSON* data sets from the Internet and from *JSON* document stores. To this end, we identified a practical case study, based on real data sets provided by Open Data portals; the *J-CO-QL* script shows how data can be retrieved, integrated and analyzed; all of the scripts are explained so as to introduce the instructions provided by *J-CO-QL*.

For the sake of clarity, the case study is divided into three parts: Part 1 is focused on retrieving data sets from web portals (through a novel yet minimal extension of the language) and on integrating them; Part 2 defines three fuzzy operators, which are the mean to express soft conditions; and Part 3 shows how to actually exploit soft conditions to retrieve possibly relevant *JSON* documents.

The reader will see how the features provided by *J-CO-QL*, all together, make the *J-CO* Framework a unique and innovative tool, with no counterpart at the moment.

The paper is organized as follows. Section 2 introduces the background of our work; in particular, Section 2.1 recalls basic concepts concerning fuzzy sets, Section 2.2 introduces the relevant literature, while Section 2.3 briefly summarizes genesis and evolution of the *J-CO* Framework. Section 3 gives an overview of the *J-CO* Framework. The main contribution of the paper involves many sections: Section 4 provides an overview of the case study, and the specific problem to solve is defined; Section 5 presents the first part of the *J-CO-QL* script, which retrieves *JSON* data sets from an Open Data portal and integrates them; Section 6 shows how to define three fuzzy operators through *J-CO-QL*; Section 7 presents the *J-CO-QL* script that actually analyzes data through fuzzy sets in order to retrieve documents

that are as suitable as possible to the goal of the case study. Section 8 discusses how the case study could be addressed without the *J-CO* Framework. Finally, Section 9 draws the conclusions and sketches future work.

2. Background

We now present the background of the paper. First, in Section 2.1 we briefly introduce basic concepts concerning fuzzy sets. Then, in Section 2.2, we report past work that is relevant to the paper. Finally, in Section 2.3, we summarize the genesis and evolution of the *J-CO* Framework.

2.1. Brief Introduction to Fuzzy Sets

In [6], Zadeh introduced the Fuzzy Set Theory. It was rapidly clear that it had (and still has) an enormous potentiality to be successfully applied to many areas of computer science, such as decision making, control theory, expert systems, artificial intelligence, natural language processing, and so on. Here, we report some basic concepts, which constitute the basis for understanding the main contributions of this paper.

To start, let X denote a non-empty universe, either finite or infinite. Definition 1 presents the concept of "fuzzy set".

Definition 1. ***Fuzzy Set.*** *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*.*

The membership degree $A(x)$ denotes the degree with which x belongs to A. If the membership degree is 1, this means that x fully belongs to A. If it is 0, this means that x does not belong at all to A. Finally, an intermediate value denotes that x partially belongs to A; of course, 0.6 denotes a stronger membership than 0.3.

From a practical point of view, 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.

We can consider some typical situations. First, a fuzzy set is "empty" if and only if its membership function is identically zero for each $x \in X$.

Second, two fuzzy sets A and B are "equal", denoted as $A = B$, if and only if $A(x) = B(x)$ for all $x \in X$.

Classical set operators such as union and intersection are defined for fuzzy sets as well.

Definition 2. ***Union and Intersection.*** *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.,* $C = A \cap B$*), in which the 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`) operator. The logical `NOT` operator corresponds to the complement set: given a fuzzy set A, its complement is denoted as $\overline{A}$; its membership function is defined as $\overline{A}(x) = 1 - A(x)$.

Fuzzy sets are useful to represent vague concepts, which characterize many real-life application contexts. For example, if the universe is the set of people, we could think to divide them into "young" and "old". However, is a person whose age is 40 actually young or actually old? He/she is a little bit young and a little bit old, neither fully young nor fully old.

We can consider another partition of the universe of people: "poor" and "rich". Is a person that earns EUR 3000 a month poor or rich? He/she is a little bit poor and a little bit rich, certainly neither fully poor nor fully rich.

Observe how we use "linguistic concepts", such as "young", "old", "poor" and "rich", to partition the universe of people and to denote fuzzy sets. Thus, "Jane is rich" is a "linguistic predicate" (see [6]) that is modeled by means of a fuzzy set. The consequence of this consideration is that linguistic predicates can be easily managed by means of fuzzy

sets and complex linguistic conditions can be managed by combining the membership degrees to fuzzy sets on the basis of union, intersection and complement. As an example, if we want to look for a person p such that "p is rich and p is young", in terms of fuzzy sets, it can be expressed as "rich(p) `AND` young(p)": given a set P, for each $p \in P$, the resulting membership degree denotes how close p is to the desired person (a rich and young person); consequently, this is a "soft query", where the membership degree is used to rank results.

2.2. Related Work

Two main topics related to the work presented in this paper can be considered: languages for querying and manipulating *JSON* documents and languages for flexibly querying databases.

Languages for Querying *JSON* documents. The advent of *JSON* as the *de-facto* standard for representing and sharing data over the Internet is a matter of fact. For this reason, the research community working on data management started working on NoSQL databases specifically designed to natively store *JSON* documents; these systems are called "*JSON* document stores". The most famous of these systems is *MongoDB* [7,8], which is a true DBMS for storing and querying large collections of small *JSON* documents. Another important DBMS is *CouchDB* [9,10], which has been chosen as the underlying DBMS for *HyperLedger Fabric*, the blockchain platform for realizing distributed and shared databases among information systems [11,12].

Consequently, many query languages for *JSON* data sets have been proposed. We only mention the most popular ones (the interested reader can refer to [3]): *Jaql* [13], *SQL++* [14], *JSONiq* [15] and the query language provided by *MongoDB* [16]. Specifically, *Jaql* [13] was introduced into *Hadoop*, the popular Map-Reduce platform [17], to help programmers develop complex transformations on Big Data represented as *JSON* data sets; however, it is strongly oriented toward programmers and does not support spatial operations. *SQL++* [14] is designed as an extension of the classical `SELECT` statement of SQL. To achieve this result, it relies on a data model that is an extension of the classical relational model that includes *JSON* documents; it is a quite interesting proposal, as confirmed by the fact that it has been chosen as the query language for analytics in *CouchDB*, with the name *N1QL for Analytics* [18]. *JSONiq* [15] is a query language for *JSON* documents derived from *XQuery* [19], the reference language for querying XML documents. Although it relies on a functional style, the very complex semantics of its constructs make this language difficult to use; furthermore, no support for spatial operations is provided. Finally, the query language provided by *MongoDB* [20] is thought to be mainly for operational purposes; its object-oriented programming style is good for programmers, while it is not so easy to use for complex analytical transformations. Furthermore, it provides a limited support to spatial operations, provided that documents are previously spatially indexed (this limits the flexibility provided by the language if compared to *J-CO-QL*).

Nevertheless, these languages are not at all suitable to becoming really unifying languages able to be independent of the specific platform and able to query heterogeneous data sets natively supporting spatial operations. These considerations motivated the *J-CO* project [21,22] and the definition of its query language (*J-CO-QL*). In Section 3, we briefly introduce the main features of our framework and of its query language.

Query languages for soft queries on relational databases. In the area of data management, many researchers have addressed the problem of performing soft queries on data sets by allowing users to express vague or imprecise concepts in the query. When selection conditions can rely on vague predicates, queries become "soft" because (i) they are tolerant to thresholds (given a non-fuzzy predicate `age <=30` to select young people, a 31-year-old person would be discarded, while in a fuzzy predicate `young(age)`, the same person would be selected with a membership degree less than 1) and (ii) selected items can be ranked on the basis of their membership degree to the selection condition. For these reasons, vague concepts and relationships can be expressed as fuzzy sets to specify soft selection conditions [23].

Based on the above-reported considerations, it is a quite straightforward idea to apply fuzzy sets to query relational databases. In effect, the literature reports two different approaches that can be adopted. The first approach extends the relational data model towards a fuzzy database model; in this case, imprecision that is intrinsically concerned with relational data can be effectively modeled [24,25]. The most interesting proposal for a query language over fuzzy-relational database is *FSQL* [26,27].

The second approach does not change the basic relational data model but extends the SQL query language with the capability to fuzzy query a conventional relational database (as in [28]). The advantage of this latter approach is evident in the *SoftSQL* proposal [4,29,30]: users are provided with a statement to define non-trivial linguistic predicates, which can be later used in the extended `SELECT` statement to select table rows on the basis of linguistic predicates; membership degrees can be exploited to rank selected rows in reverse order of importance. Other extensions of SQL have been proposed, such as *SQLf* [31,32] (for which we can mention an attempt to implement it [33]) and its extension named *SQLf3*, in which latest constructs introduced in *SQL3* are encompassed. Another interesting proposal of a fuzzy query language for classical relational databases is *FQUERY for Access* [28,34]; as the name suggests, it was designed to operate on databases managed by Microsoft Access. As the reader can see, the topic of extending SQL towards flexible querying on classical relational databases received significant attention; in fact, comparisons of the proposals are available too, such as in [35,36], and strategies to translate flexible queries to classical relational databases have been investigated [37]. However, work on this topic substantially ended with [38], a very large handbook that summarizes all research on defining fuzzy-relational database models and fuzzy query languages for relational databases.

The fuzzy extension we define for the *J-CO-QL* language moves from these previous works on fuzzy querying applied to classical relational databases; in particular, some ideas like providing practical instructions to define customized fuzzy operators has been kept (see Section 6).

From a general point of view, fuzzy querying in databases is a specific application of more general Fuzzy Information Retrieval models [39]. These models adopt the following approach to query sets of documents: the query is interpreted as the specification of soft constraints that the representation of a document can satisfy to a certain degree. During query evaluation, the degree with which the query is satisfied by the representation of each document is computed to rank documents. Refer to [40] for a very good survey regarding the adoption of fuzzy sets in information retrieval.

Another interesting survey about the adoption of an information-retrieval technique and fuzzy logic in databases is [41], where a discussion about the possibility to extend relational databases was provided. Although the paper is not recent (it appeared in 1997), it is still able to provide a clear view of issues concerned with this area. In particular, the attempts to extend the relational model towards a fuzzy-relational model have not succeeded in commercial systems; we agree with [41] that what users expect from a DBMS is simplicity and performance. Unfortunately, fuzzy-relational models are not simple (modeling uncertainty may be very difficult) and performances deteriorate if compared to classical relational databases. In this regard, the *JSON* world shows similar dynamics: although a degree of uncertainty is implicit, *JSON* documents come as crisp documents. Therefore, the real need is for a tool able to flexibly query and manipulate large collections of crisp *JSON* documents; our work follows this research direction.

Soft querying on graph data. The advent of the Semantic Web has been accompanied by the definition of the Resource Description Framework (RDF) [42]: it is a graph-based formalization for meta-data and relationships among web resources. RDF can be also used to build "Geo Linked Data" [43], i.e., the concept of the Semantic Web applied to geographical data and entities. The next step to make RDF actually exploitable has been the definition of a standard query language named *SPARQL* [44], which allows users wishing to query RDF data to express possibly complex queries.

The need for making *SPARQL* flexible by extending it with fuzzy constructs has been addressed in various papers. In [42], *fSPARQL* (the fuzzy version of *SPARQL*) was introduced for the first time; however, the paper does not address how to define fuzzy terms and fuzzy operators. Reference [45] proposed a slightly different version of *fSPARQL*, suitable for clustering data sets; although the problem of defining fuzzy terms and fuzzy operators is addressed, it is solved by asking users to write Java classes to incorporate them into the query engine and, thus, they do not adopt a declarative approach, as we will show in Section 6. Finally, Reference [46] considered an extension of the RDF model in order to obtain fuzzy RDF data; the paper proposed another version of *fSPARQL*that uses fuzzy constructs for navigational purposes in the RDF graph. Consequently, we can say that the various attempts to propose a fuzzy extension of *SPARQL* have not converged yet.

We conclude this section by citing [47]. It discusses preliminary ideas on the possibility to extend *Cypher*, the declarative query language of "Neo4j", a novel NoSQL graph database. The paper is interesting because it considered a different type of NoSQL databases, i.e., graph databases. However, many practical issues were not considered at all, as the way to define fuzzy sets and fuzzy operators. In contrast, the fuzzy extension of the *J-CO-QL* language moved from the definition of fuzzy operators, which are the basis for evaluating fuzzy sets on *JSON* documents.

Fuzzy Querying on *JSON* document stores. In the literature, there are few works that propose fuzzy extensions to query languages on *JSON* document stores.

In [48], an extension of the *MongoDB* query language (*MQL*) is proposed, which allows users to use "fuzzy labels" to query *JSON* documents; the extension is called *fMQL*. A fuzzy label is equivalent to a linguistic predicate: users can use such labels instead of values in comparison predicates. Unfortunately, the authors of [48] do not consider how to define fuzzy labels; furthermore, for each single *JSON* document, only one membership degree is implicitly evaluated. In this regard, *J-CO-QL* provides a more powerful approach because, for each single document, it is possible to evaluate its belonging to more than one fuzzy set at the same time.

The paper [49] continued the work on *fMQL* by proposing a translation of *fMQL* instructions into *fXML*, a fuzzy language for querying XML documents. In [49], *fMQL* is presented in a more extensive way, even though it is not clear how membership degrees are dealt with. Furthermore, a serious concern arises: if *fMQL* queries are translated into *fXML* queries, *JSON* documents should be previously translated into XML documents and stored into an XML Database. Therefore, in our opinion, the usefulness of this proposal is not clear at all.

Finally, we cite the work [50]. It proposed an approach for soft querying *JSON* documents. However, the proposed approach preliminarily translates *JSON* documents into fuzzy RDF triples so that *FSA-SPARQL*, another fuzzy extension to *SPARQL*, is used to query fuzzy RDF graphs that represent *JSON* documents. This approach is really far away from the idea behind *J-CO-QL* and its fuzzy extension.

To the best of our knowledge, there are no other proposals for query languages over *JSON* data sets that provide soft querying capabilities.

2.3. Genesis of the J-CO Framework

The *J-CO* Framework sprouted from continuous research on the problem of gathering and managing data sets coming from the Internet in general and from Open Data portals and Social Media in particular, carried out at the University of Bergamo (Italy).

Originally, we started working on web search results by developing a technique for clustering and organizing web search results [51,52]. Then, the advent of Open Data and the massive diffusion of Social Media have addressed our research towards different yet related directions.

Before the advent of Open Data, reviews about products and services were significantly popular; the idea that users provided their own judgment to share non-official information about products and their quality was innovative. We launched the *Hints from the Crowd*

project [53–56]: product reviews were collected and indexed through inverted indexes; then, natural-language queries formulated by users were addressed to the indexes in order to retrieve reviews that possibly matched user needs. Meanwhile, Open Data portals have become more important than product reviews in the global context, so we moved our research activity towards them. In fact, they have been used more and more as sources of authoritative information, where public administrations publish data concerning the administered territory. In [57,58], we proposed a technique for blindly querying these portals: in fact, the number of published data sets is so large that it is not possible to know their structure in advance; the blind approach exploits information-retrieval techniques to retrieve possibly interesting data sets and single documents of interest on the basis of a modern implementation on Map-Reduce platforms [59,60]. Since the developed technique works on data sets in the form of *JSON* documents, we then applied it to data stored in a *JSON* document store managed by the *MongoDB* NoSQL DBMS [61].

Social Media APIs (Application Programming Interfaces) provide data in the form of *JSON* documents as well; among them, Twitter follows this approach too, and we decided to exploit it. In 2015, we started the *FollowMe* project [62,63], in which the goal was to gather traces of Twitter users by detecting their geo-located tweets. We focused on users possibly interested in visiting EXPO 2015, held in Milan (Italy) in 2015. To exploit the gathered data set, we developed various tools, such as a framework [64] for managing and analyzing traces of Twitter users by means of a clustering technique [65].

The *FollowMe* project and the collected data became the fundamental bricks of the *Urban Nexus* project [66]. Computer scientists and geographers worked together to exploit Big Data and Social Media to develop novel methodologies suitable for studying how people move and live within their territory [67]. At the beginning of this project, it was clear that a framework able to easily manipulate possibly geo-tagged *JSON* data sets without writing procedural code were missing, and in [21], we proposed the first version of the *J-CO-QL* language. We treasured the previous experience in defining data models for complex geographical entities [68,69]. After that, we started to continuously develop and refine the *J-CO* Framework (and *J-CO-QL* [22,70]), which has reached the current shape of platform-independent framework [3].

Meanwhile, we have thought about introducing constructs to manage fuzzy sets in the language: we presented this in [5,71], treasuring the experience from our previous work [30,72] on soft querying relational databases and uncertain location-based queries.

3. The *J-CO* Framework

J-CO is a platform-independent framework specifically designed to provide analysts with a powerful tool able to retrieve, integrate and transform possibly geo-tagged *JSON* documents from possibly large *JSON* data sets.

The framework is built around a powerful query language named *J-CO-QL*. Many software tools constitute the framework; however, together with external sources and tools, it is possible to imagine a *J-CO* environment that analysts can exploit to perform analyses. Hereafter, we provide an overview.

- *Web Data Sources.* Web sites, Open Data portals and Social Media can be accessed to obtain *JSON* data sets. In particular, end points of Open Data portals and Social Media can be easily contacted through an HTTP call. All together, these sources constitute an immense mine of potentially useful data.
- *JSON Document Stores.* Several NoSQL DBMSs have been built in order to natively store and manage *JSON* documents. For this reason, they are called "*JSON* Document Stores". The most famous one is *MongoDB*, which is currently integrated within the *J-CO* environment. This category encompasses also tools that are not properly DBMSs but are able to store and provide *JSON* data sets. One of them is *ElasticSearch*, which is an information-retrieval tool; it is not a DBMS because it provides neither a query language nor the usual capabilities expected for a DBMS.

- *J-CO-DS*. *JSON* Document Stores can suffer for significant limitations in managing *JSON* documents. For example, *MongoDB* is unable to store very large documents because it has been designed to manage a very large number of small documents (thinking about operational processing). On the opposite side, *ElasticSearch* has to index documents, so it cannot be used as a pure storage system because it would be highly inefficient.
  For this reason, we designed *J-CO-DS* [73]: it is a component of the *J-CO* Framework that can be used to build *JSON* databases able to store very large *JSON* documents. It does not provide a query language because computational capabilities are provided by the framework. However, it can be a valuable tool to easily build flexible *JSON* stores.
- *J-CO-QL Engine*. This is the component of the framework that actually provides the computational capabilities. It executes *J-CO-QL* queries by accessing external data sources, *JSON* document stores and *J-CO-DS* databases to retrieve source data and to store output data.
- *J-CO-UI*. This is the current user interface provide by the *J-CO* Framework. It allows analysts to write *J-CO-QL* complex queries (or scripts, for simplicity) in a step-by-step way. In fact, the user can write one single instruction at a time, execute it and inspect temporary results. At this point, the user can decide either to submit another instruction or to roll back the last executed one (this way, a *trial-and-error* approach typical of investigation tasks can be adopted).

Figure 1 illustrates the vision. In particular, the idea is that the *J-CO* Framework becomes the unifying tool to overcome issues concerned with the integration of different technologies; in particular, *J-CO-QL* is meant to be the unique and high-level query language for this enlarged environment so as to let analysts be unaware of specific query languages (if any) provided by *JSON* stores.

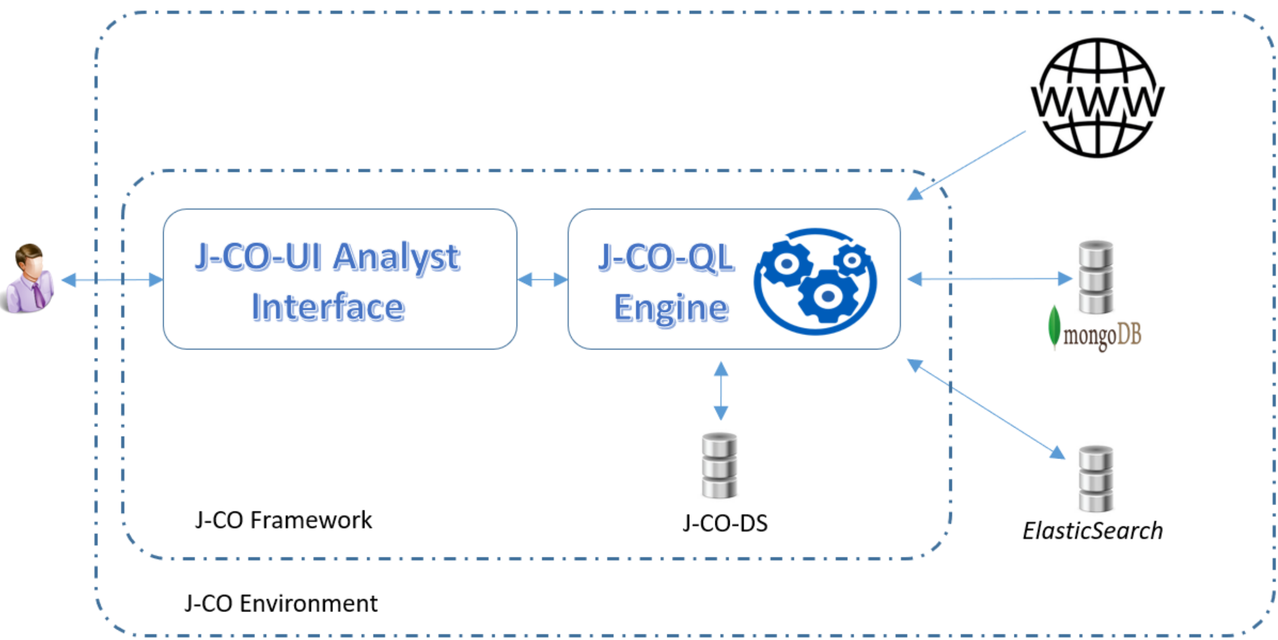


Figure 1. The *J-CO* Framework within the *J-CO* Environment.

3.1. The J-CO-QL Language

The *J-CO* Framework is built around *J-CO-QL*, the query language designed to query *JSON* data sets. Here, we present the main features of the language. The instructions are presented while discussing the case study.

3.1.1. Data Model

First, we present the data model upon which the language relies. We start by defining the concept of "document".

Definition 3. ***Document.*** *A JSON document d is a map $field_name \rightarrow value$, where the value can be a number (integer or real), a string, a Boolean value, a nested document or an array of values. Obviously, since d is a map, there cannot be two fields with the same name.*

When the document d is serialized, it is depicted with braces; each field is depicted as a pair $field_name$`:`*$value$, where $field_name$ is enclosed within a pair of* `"` *(e.g.,* `"a":1`*) and fields are separated by a comma; string values are either within a pair of* `"` *or within a pair of* `'`*; and arrays are enclosed within square brackets.*

Very often, *JSON* documents are "geo-tagged" or "geo-referenced", i.e., they describe information related to spatial entities, described with some kind of "geometry", i.e., points (denoted by latitude and longitude), polylines and areas, as well as collections of them.

Definition 4. ***Geometry.*** *Given a document d, a special field named* `~geometry` *at the root level of the document describes spatial properties of the real-world entity described by d.*
This field is a nested JSON document that relies on the `GeometryCollection` *type defined in the GeoJSON standard [74].*

The data model must support the evaluation of membership degrees to fuzzy sets. A single document d can belong to many fuzzy sets, with specific membership degrees. The following definition addresses this point.

Definition 5. ***Fuzzy Sets.*** *Given a document d, it could be known if d belongs to one or more fuzzy sets $S_1, S_2, \dots, S_n$, for each one with a given membership degree $S_i(d) \in [0,1]$ (with $1 \leq i \leq n$). A special field named* `~fuzzysets` *at the root level of document d represents the membership degrees to fuzzy sets $S_1, S_2, \dots, S_n$: each field in* `~fuzzysets` *corresponds to a fuzzy set S_i, where the field name is the name S_i of the fuzzy set, while its value is the membership degree $S_i(d)$ of d to the S_i fuzzy set (thus, the value is in the range $[0,1]$).*

Notice that the names `~geometry` and `~fuzzysets` are compliant with the *JSON* standard [2]. This choice gives the advantage that documents processed by *J-CO-QL* can be natively stored within standard *JSON* document stores, such as *MongoDB*, as well as that they can be easily shared as regular *JSON* documents.

A document is a basic item upon which *J-CO-QL* operates. It is not designed to work with one single document at a time but with many documents at the same time.

Definition 6. ***Collection.*** *A collection $c = \{d_1, d_2, \dots, d_n\}$ is a possibly-empty finite set of documents $d_1, d_2, \dots, d_n$.*

The concept of "collection" is borrowed directly from *MongoDB*, where the database is a set of collections.

3.1.2. Execution Model

We now present the execution model of the language. The goal is to let the reader fully understand scripts that are presented in the rest of the paper.

Definition 7. ***Query-Process State.*** *The state of the query process is a tuple*

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

where tc is the temporary collection*, i.e., the input collection for the next instruction and the output collection for the previous instruction. IR is the* Intermediate Results *database, i.e., a volatile database associated to the query process, used to temporarily store intermediate results.*

DBS is the set of databases managed by JSON stores to which the process is connected. FO is the set of fuzzy operators defined within the query, used to evaluate fuzzy sets on JSON documents.

The rationale behind the *IR* database is the following: the query could be a long transformation process that could generate many intermediate results to be used later. Since the *J-CO* Framework is designed to be platform-independent and autonomous from computational capabilities possibly provided by *JSON* document stores and since it is not the case to make persistent databases dirty with intermediate results, each query process operates in isolation with its own volatile *IR* database, which is lost at the end of the query process.

Definition 8. ***Query.*** *In J-CO-QL, a query is a sequence of instructions. Formally, it is*

$$q = (i_1,\ i_2,\ \ldots,\ i_n)$$

where i_j, with $1 \leq j \leq n$, is the jth instruction in the sequence.

We can now complete the definition of the execution model by defining the concept of "query process".

Definition 9. ***Query Process.*** *Given a query q, its execution passes through $n+1$ query-process states, where $n = |q|$ is the length of the query. The initial query-process state is $s_0 = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ (respectively, the temporary collection is an empty collection, the IR database is empty, and the set of connected databases and the set of fuzzy operators are empty).*

Given an instruction $i_j \in q$, with $1 \leq j \leq |q|$, it takes the previous $s_{(j-1)}$ query-process state as input and generates the novel s_j query-process state as output. Depending on the instruction, one or more of the elements in the query-process state could change.

The reader can notice that the query is a kind of "pipe" of instructions: the query-process state passes through instructions and evolves. In the rest of this paper, the reader will see that, this way, the order in which instructions are written follows the natural way in which instructions are thought out by the analyst.

3.1.3. Features of *J-CO-QL* Instructions

J-CO-QL provides many instructions to manipulate collections of *JSON* documents. We present many of them in the next sections when we show how the case study could be addressed. Here, we want to introduce the features that characterize them and make *J-CO-QL* unique in the current panorama of query languages for *JSON* documents.

- **Adaptability to heterogeneity.** Collections of documents could contain documents with heterogeneous structures. Instructions are designed to allow users to deal with documents with different structures in one single instruction.
- **Spatial operations.** The data model natively encompasses geometries within documents. Consequently, the language provides spatial operations, which rely on the `~geometry` field, if present. Furthermore, these operations can be made "on the fly", with no need to build persistent spatial indexes in advance.
- **Orientation to analysts.** Instructions are not oriented to programmers, while they are oriented to analysts. This means that the language is not imperative and does not provide any instruction to control the execution flow, such as conditional instruction or cycles. The execution semantics presented in Definition 9 are clear and intuitive; instructions are thought to be declarative, since each of them allows for specifying a specific transformation in a high-level way, i.e., a way that does not require strong skills in computer programming.

4. Case Study: Overview

We now address the main contribution of the paper: we show how *J-CO-QL* can be effectively used for flexibly querying data sets directly acquired from an Open Data portal.

To this end, we define a case study that is inspired by real data sets available on a real Open Data portal. In fact, in this case study, we show how to exploit the *J-CO* Framework in order to gather data from a publicly available Open Data portal, handle, transform and integrate the acquired data, and then perform a soft query (based on fuzzy concepts) on them.

Currently, many public administrations continuously publish data concerning the environment, i.e., measurements performed by sensors on pollution, meteorological events, and so on. We retrieved data on the fly from the Open Data portal of *Regione Lombardia* [75], the administrative body of the region named Lombardia in northern Italy. The Open Data portal of Regione Lombardia is based on *Socrata* [76], one of the de facto standard platforms for building Open Data portals, which provides data by means of web APIs.

The portal publishes new data every day, measured by a plethora of sensors distributed on the territory of the region, which describe levels of pollutants, rain, solar radiation, air temperature and pressure, and so on. Analysts could be interested in cross-analyzing them, for example, to discover potential abnormal situations, characterized by strange combinations of pollutants and environmental factors. For example, the city of Milan (the capital of Lombardia) suffers from high levels of pollutants in specific periods, in particular, high levels of *nitric oxide*; an analyst could be interested in discovering if high levels of *nitric oxide* occur either with low *solar radiation* or with *rain precipitation* (notice that high levels of *nitric oxide* typically occur with high *solar radiation* and/or without *rain*).

Hereafter, the problem that the case study addresses is defined in a more precise way.

Problem 1. *Consider a time window of interest, denoted as TW, and a city of interest, denoted as C. From the Open Data portal, get geo-located data about sensors for nitric oxide pollution, rain and solar radiation located in C.*

Then, from the Open Data portal, get all measures concerning nitric oxide, solar radiation and rain related to TW made by the sensors in city C.

Finally, cross analyze the acquired data in order to find areas in C where a high level of nitric oxide occurred when either the solar radiation was low or there was persistent rain.

The considered time window *TW* is from 08 April 2021 to 15 April 2021; the city *C* of interest is Milan. To address Problem 1, we wrote a complex *J-CO-QL* script that is divided in three different parts, each one devoted to performing a specific sub-task. For the sake of clarity, a specific section of the paper is dedicated to each part.

- **Part 1**. This part of the script, reported in Listings 1–6, is in charge of (i) obtaining the data from Regione Lombardia Open Data portal in the time window of interest and (ii) selecting only data related to the city of Milan describing the sensors that provide the desired measures. The final goal of this sub-script is to create documents that represent *measure triplets*, i.e., associations of three measurements of *nitric oxide*, *solar radiation* and *rain precipitation*, occurring at the same time and that are spatially close each other. Section 5 extensively discusses this part of the script by introducing non-fuzzy *J-CO-QL* instructions while presenting the script.
- **Part 2**. This part of the script, reported in Listing 7, contains the definitions of specific fuzzy operators to be used to analyze *measure triplets* that are obtained at the end of Part 1. Fuzzy operators allow for expressing linguistic predicates that allow for imprecise matching. Defining fuzzy operators is the preliminary step to actually use them to query *measure triplets*. Section 6 explains how it is possible to create fuzzy operators in *J-CO-QL*.
- **Part 3**. This part of the script, reported in Listing 8, applies the fuzzy operators defined in Part 2 to query the *measure triplets* produced by Part 1 on the basis of source data gathered from the Open Data portal. In order to solve Problem 1, we exploit linguistic predicates specified by means of the fuzzy operators defined in Part 3 so as to capture triplets that only partially match the condition as well; the membership degree is used to rank triplets, if necessary. Remember that we want to discover those areas in the

city of Milan where high levels of *nitric oxide* are registered when there is either low *solar radiation* or *persistent rain*. Section 7 extensively explains the third part of the script in order to show how fuzzy sets can be successfully applied to analyze data sets represented as *JSON* documents.

The reader will see that the case study is not trivial and that the overall *J-CO-QL* script is not short. The features provided by the *J-CO-QL* instructions are able to deal with such a complex activity (i) to obtain the source data from the Open Data portal, (ii) to harmonize and integrate data sets and (iii) to perform a soft query on the data on the basis of soft linguistic conditions; all of this is performed without writing procedural computer programs but by specifying transformations in a declarative way. This way, through the case study, we unveil the potential capabilities of the *J-CO* Framework. Nevertheless, note that we are not environmental analysts; thus, we do not analyze the outcomes of the script because that is responsibility of domain experts and is outside the scope of the paper.

5. Case Study Part 1: Retrieving Data about Sensors and Measurements

In this section, we present the first part of the *J-CO-QL* script we wrote to address the case study. The goal of Part 1 is to gather all data about the wanted sensors and their measurements in the selected time window and then to create *JSON* documents that describe *measure triplets*, which reports the measurements made at the same time of *nitric oxide*, *solar radiation* and *rain precipitation* by sensors that are spatially close each other. Due to the large number of activities to perform in order to obtain the desired data set, the *J-CO-QL* script is long; thus, it is divided in six separate fragments, reported in Listings 1–6. Hereafter, we give an overview of the work performed by each fragment.

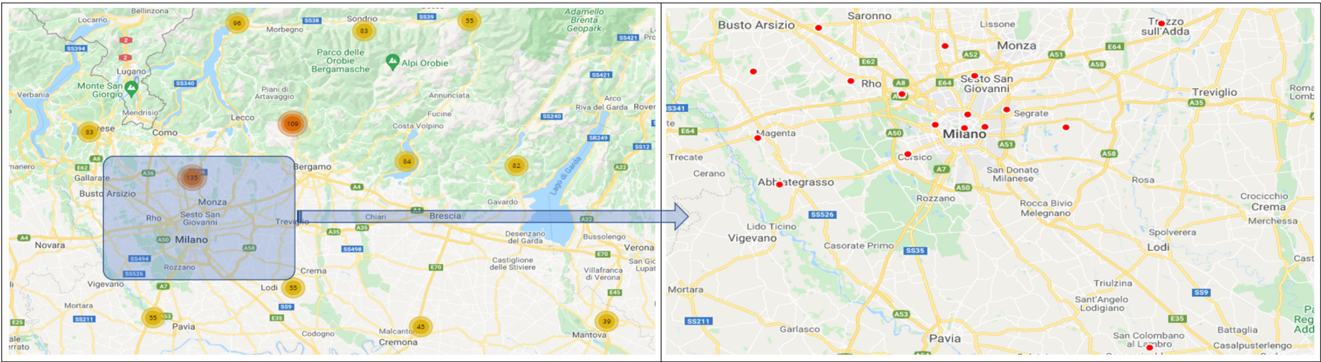


Figure 2. Locations of the 1026 active weather sensors on a regional scale (**left-hand side**) and of the 23 weather (*rain precipitation* and *solar radiation*) sensors located in 17 stations (red-dots) in the province of Milan (**right-hand side**).

- Listing 1 retrieves all weather sensors located in Lombardia from Regione Lombardia Open Data portal [77]. Then, it selects sensors located in the area of Milan that measure either levels of *solar radiation* (expressed in W/m^2) or *rain precipitation* (expressed in mm). The left-hand side of Figure 2 shows the map [78] of weather sensors in Lombardia, while the right-hand side shows their precise positions within the city of Milan.
- Listing 2 retrieves data about air quality sensors located in Lombardia from Regione Lombardia Open Data portal [79]. Then, it selects sensors located in the area of Milan that measure levels of *nitric oxide* (expressed in mg/m^3). The left-hand side of Figure 3 shows the map [80] of air quality sensors in Lombardia, while the right-hand side shows their position within the city of Milan.
- Listing 3 builds *virtual sensor stations*, i.e., groups of *nitric oxide*, *solar radiation* and *rain precipitation* sensors that are close each other.

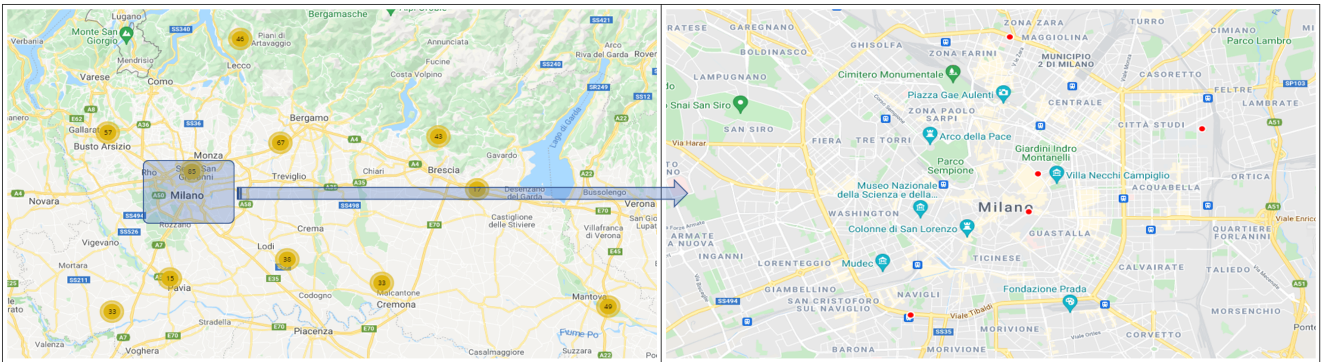


Figure 3. Locations of the 506 active air-quality sensors on a regional scale (**left-hand side**) and of the 5 *nitric oxide* sensors (red-dots) within the city of Milan (**right-hand side**).

- Listing 4 retrieves all of the measurements produced by weather sensors during the time window from 08 April 2021 to 15 April 2021 from Regione Lombardia Open Data portal. Each measurement is described by a specific *JSON* document that reports the sensor identifier and the date and time in which the measurement was performed. The value of the measurement is provided by the portal as a pure number, with no information about the measure unit.
- Listing 5 retrieves all of the measurements produced by air quality sensors during the time window from 08 April 2021 to 15 April 2021 from Regione Lombardia Open Data portal. Again, each measurement is described by a *JSON* document that reports the sensor identifier and the date and time in which the measurement was performed. The value of the measurement is reported as a pure number, as it is for weather measurements.
- Listing 6 exploits the set of *virtual sensor stations* built by Listing 3 to create a collection of *JSON* documents describing *measure triplets*, where each document contains the levels of *nitric oxide*, *solar radiation* and *rain precipitation* measured at the same time by sensors grouped in the same virtual station.

In the rest of the section, we analyze each single instruction of Part 1 of the script in detail.

5.1. Retrieving Data about Sensors for Solar Radiation and Rain Precipitation

Listing 1 retrieves data about active sensors for measuring *solar radiation* and *rain precipitation*. We present each single instruction in details by explaining its semantics.

- On line 1, `USE DB` instructs the system to connect to the database managed by the *MongoDB* server installed on the computer where the *J-CO-QL Engine* runs. The database is called `INFORMATION_2021` and will contain the final *JSON* documents obtained by the script. Notice that this instruction alters the *DBS* component of the query-process state (see Definition 7), containing descriptors of databases to which the process is connected.

Listing 1. *J-CO-QL* script retrieving data about weather sensors.

```
1:  USE DB INFORMATION_2021 ON SERVER MongoDB 'http://127.0.0.1:27017';

2:  GET COLLECTION FROM WEB
"https://www.dati.lombardia.it/resource/nf78-nj6b.json?
$limit=1000000&$where=storico<>'S'";

3:  EXPAND
UNPACK
WITH ARRAY .data
ARRAY .data
TO .sensor
DROP OTHERS;

4:  FILTER
CASE
WHERE   WITH  .sensor.item.idsensore, .sensor.item.provincia,
.sensor.item.tipologia
AND   .sensor.item.provincia  = "MI"
AND   .sensor.item.tipologia  = "Radiazione Globale"
GENERATE {
.stationName   :  .sensor.item.nomestazione,
.province      :  .sensor.item.provincia,
.latitude      :  TO_FLOAT (.sensor.item.lat),
.longitude     :  TO_FLOAT (.sensor.item.lng),
.altitude      :  TO_FLOAT (.sensor.item.quota),
.sensorId      :  .sensor.item.idsensore,
.sensorType    :  "Global Radiation",
.sensorUnit    :  .sensor.item.unit_dimisura
}
WHERE   WITH   .sensor.item.idsensore, .sensor.item.provincia,
.sensor.item.tipologia
AND    .sensor.item.provincia = "MI"
AND    .sensor.item.tipologia = "Precipitazione"
GENERATE {
.stationName   :  .sensor.item.nomestazione,
.province      :  .sensor.item.provincia,
.latitude      :  TO_FLOAT (.sensor.item.lat),
.longitude     :  TO_FLOAT (.sensor.item.lng),
.altitude      :  TO_FLOAT (.sensor.item.quota),
.sensorId      :  .sensor.item.idsensore,
.sensorType    :  "Rain Precipitation",
.sensorUnit    :  .sensor.item.unit_dimisura
}
DROP OTHERS;

5:  FILTER
CASE
WHERE    WITH .latitude, .longitude
GENERATE
SETTING GEOMETRY POINT (.latitude, .longitude)
DROP OTHERS;

6:  SET INTERMEDIATE AS Weather_Sensors;
```

- The `GET COLLECTION` instruction on line 2 retrieves data about the weather sensors from Regione Lombardia Open Data portal by sending the request to the dedicated end point [77]. This is performed by specifying the URL which to send the HTTP request. In the URL, notice the parameters after the question mark: the first parameter

(`$limit=1000000`) specifies the maximum number of documents to return; the second parameter (`$where=storico<>"S"`) is written according to *Socrata* query language [76] (remember that Regione Lombardia Open Data portal is powered by the *Socrata* platform) selects only active weather sensors. The data are provided in *JSON* format as a unique array of documents, one for each sensor; the connector converts this array into a unique document containing an array field named `data`, which contains all documents provided by the portal. There are different kinds of sensor, each one providing a different measurement type. As an example, measurement types could be levels of *solar radiation* or *rain precipitation*, *temperature*, *wind speed*, and so on.
Figure 4a shows an excerpt of the beginning of the document that constitutes the initial temporary collection (see Definitions 7 and 9). Notice that most of the field names are in Italian and that all of their values are reported as string values.
We just introduced the capability of the `GET COLLECTION` instruction to send HTTP requests to web portals; in our previous publication [3], the instruction was able to retrieve collections from *JSON* databases only. With this minor extension, the entire World-Wide Web can be seen as a giant *JSON* store by the *J-CO* Framework.


```
{
  data = [
    {
      ":@computed_region_6hky_swhk" : "3",
      ":@computed_region_ttgh_9sm5" : "3",
      "cgb_est" : "595908",
      "cgb_nord" : "5131141",
      "datastart":"2002-01-01T00:00:00.000",
      "idsensore" : "100",
      "idstazione" : "52",
      "lat" : "46.327055775545546",
      "lng" : "10.245970197827939",
      "location" : {
        "human_address" : "{\"address\":
  \"\", \"city\": \"\", \"state\": \"\",
  \"zip\": \"\"}",
        "latitude" : "46.327055775545546",
        "longitude" : "10.245970197827939"
      },
      "nomestazione" : "Grosio Diga Fusino",
      "provincia" : "SO",
      "quota" : "1220",
      "storico" : "N",
      "tipologia" : "Precipitazione",
      "unit_dimisura" : "mm"
    },
    {
      ...
    },
    ...
  ]
}
```


(**a**)


```
{
  "sensor" : {
    "item" : {
      ":@computed_region_6hky_swhk" : "3",
      ":@computed_region_ttgh_9sm5" : "3",
      "cgb_est" : "595908",
      "cgb_nord" : "5131141",
      "datastart":"2002-01-01T00:00:00.000",
      "idsensore":"100",
      "idstazione" : "52",
      "lat" : "46.327055775545546",
      "lng" : "10.245970197827939",
      "location" : {
        "human_address" : "{\"address\":
\"\", \"city\": \"\", \"state\": \"\",
\"zip\": \"\"}",
        "latitude" : "46.327055775545546",
        "longitude" : "10.245970197827939"
      },
      "nomestazione" : "Grosio Diga Fusino",
      "provincia" : "SO",
      "quota" : "1220",
      "storico" : "N",
      "tipologia" : "Precipitazione",
      "unit_dimisura" : "mm"
    },
    "position" : 0
  }
}
```


(**b**)

Figure 4. Example of documents generated by line 2 (**a**) and by line 3 (**b**). (**a**) Excerpt of the beginning of the document obtained from the portal with data about weather sensors. (**b**) Example of the document obtained after unnesting documents received from the portal about weather sensors.

- The `EXPAND` instruction on line 3 generates a new temporary collection by unnesting documents within array fields. Specifically, we want to generate one single document for each weather sensor obtained from the portal. The `UNPACK` clause selects all documents that satisfy the condition: in this case, by means of the `WITH` predicate, the `UNPACK` clause selects the lonely document in the input temporary collection because it has the `data` field. The `ARRAY` sub-clause specifies the field to expand (the `data` field) and the `TO` keyword precedes the field to generate into the new documents by

unnesting an item in the source array. In fact, the instruction behaves this way: given a document to expand, for each item in the array, a novel document is generated; all fields in the source document are kept, except for the expanded array, while a new field is added. In this case, since the input document has only the `data` array field (to expand), the output documents have only the `sensor` field. Figure 4b shows an example of document generated by the instruction: notice that the `sensor` field is a sub-document with the `item` field, which contains the unnested document, and the `position` field, which denotes the position of the unnested item in the source array. Finally, the `DROP OTHERS` option discards all of those documents in the input temporary collection that is not selected by the `UNPACK` clause; clearly, in this case, it is ineffective.

- The `FILTER` instruction on line 4 filters and transforms documents in the temporary collection. It is an example of the feature we call *adaptability to heterogeneity* in Section 3.1.3.
  In the `CASE` clause, there are two distinct `WHERE` conditions that select documents from the input temporary collection. The evaluation policy is the following: for each document d in the input temporary collection, (i) if d matches the condition expressed in the first `WHERE` condition, then (ii) the following `GENERATE` action is applied to d and the subsequent `WHERE` condition(s) is ignored; (iii) if d does not match the first `WHERE` condition, then (iv) the second `WHERE` condition is evaluated on d and, if d matches it, the following `GENERATE` action is applied; otherwise, (v) d is discarded according to the final `DROP OTHERS` option.
  Notice that a `FILTER` instruction can have multiple `WHERE` conditions, each one with its own `GENERATE` action; this way, it can manage heterogeneous collections.
  - In this case, the first `WHERE` condition selects documents that have the fields that are listed in the `WITH` predicate and for which the values identify *solar radiation* sensors (`.sensor.item.tipologia = "Radiazione Globale"`) in the province of Milan (`.sensor.item.provincia = "MI"`). Notice that a ”province” is an administrative body that administers a territory smaller than a region). The following `GENERATE` action generates a new output document in place of the selected one; the structure described in the `GENERATE` action can be quite complex, but in this case, a simple flat document is generated (for ease of use). A field value can be: (i) a constant value (e.g., `.sensorType : "Global Radiation"`); (ii) a value taken as is from a source field (e.g., `.sensorId : .sensor.item.idsensore`); or (iii) a value obtained by converting the value of a source field from one data type to another (e.g., `.latitude:TO_FLOAT (.sensor.item.lat)`, which converts a string into a floating-point number). The remaining source fields are discarded.
  - The second `WHERE` condition and the following `GENERATE` action work similarly to the previous ones; they select and rebuild only documents that describe *rain precipitation* sensors (`.sensor.item.tipologia = "Precipitazione"`) in the province of Milan.
    Notice that documents that describe other types of sensor are discarded because they are not relevant to the case study.

  Figure 5a shows an example of an output document. Notice that now all of the fields have an English name and their values are numerical if they represent a numerical property (e.g., the `latitude` field).
- The `FILTER` instruction on line 5 selects those documents with the `latitude` and `longitude` fields (in this case, all of the documents) and the following `GENERATE` action specifies the `SETTING GEOMETRY` option, which creates and adds a geometry to the document by deriving the `~geometry` field (according to Definition 4) from the `latitude` and `longitude` fields, which denote the position of the sensor. This new special field enables spatial operations on documents.
  Figure 5b shows the document obtained by adding the `~geometry` field to the document reported in Figure 5a.

```
{
  "altitude"    : 120.0,
  "latitude"    : 45.4967798240395,
  "longitude"   : 9.25751539968107,
  "province"    : "MI",
  "sensorId"    : "2008",
  "sensorType"  : "Global Radiation",
  "sensorUnit"  : "W/m²",
  "stationName" : "Milano Lambrate"
}
```

(**a**)

```
{
  "altitude"    : 122.0,
  "latitude     : 45.471656293243,
  "longitude"   : 9.18911038370884,
  "province"    : "MI",
  "sensorId"    : "19374",
  "sensorType"  : "Global Radiation",
  "sensorUnit"  : "W/m²",
  "stationName" : "Milano v.Brera",
  "~geometry"   : {
    "type"        : "Point",
    "coordinates" : [
        9.18911038370884,
        45.471656293243
    ]
  }
}
```

(**b**)

Figure 5. Example of a document in the temporary collection generated by line 4 (**a**) and by line 5 (**b**). (**a**) Example of a document describing a weather sensor, without geometry. (**b**) Example of a document describing a weather sensor with geometry.

- The `SET INTERMEDIATE AS` instruction on line 6 saves the current temporary collection into the Intermediate Results database *IR* (as defined in Definition 7), naming it `Weather_Sensors`. This way, this key intermediate result is saved for subsequent use.

The script reported in Listing 1 retrieved 1026 documents describing weather sensors; among them, 7 *solar radiation* sensors and 16 *rain precipitation* sensors were selected. Consequently, the `Weather_Sensors` collection contains 23 documents. Notice the documents describing sensors of solar radiation, where the `sensorType` field was assigned the constant `"Global Radiation"` value: this is the English translation of the label used in the portal to denote this type of sensors.

As a final remark, observe the way fields are referred to in the instructions. For example, `.sensor.item.idsensore` is a path from the root of the document (denoted by the initial dot) to the `idsensore` field nested within the `item` field, which is in turn nested within the root level `sensor` field.

5.2. Retrieving Data about Sensors for Level of Nitric Oxide

Listing 2 reports the sub-part of the *J-CO-QL* script that retrieves data about sensors for *nitric oxide*; these data are provided by the portal through an end point that is not the same one that provides data about weather sensors. Apart from this detail, the sub-script in Listing 2 is structured in the same way as the sub-script in Listing 1. Nevertheless, we provide a brief description.

- The `GET COLLECTION` instruction on line 7 obtains the data about active sensors for air quality from Regione Lombardia Open Data portal by sending the request to the dedicated end point [79]. As for the `GET COLLECTION` instruction on line 2 (in Listing 1), the data are received in *JSON* format as a unique document containing an array field (named `data`) where each item represents a single air-quality sensor. There are different kinds of sensor, each one performing a different measurement type; they can be (i) level of *nitric oxide*, (ii) level of *ozone*, (iii) level of *carbon monoxide*, and so on. Figure 6a shows an excerpt of the beginning of the document provided by the instruction, which constitutes the new temporary collection. Notice that the structure of the items is very similar to the structure of the items shown in Figure 4a, apart from a few extra fields.
- The `EXPAND` instruction on line 8 generates a new temporary collection where each document represents a single air-quality sensor; similar to the `EXPAND` instruction on line 3; these documents are obtained by unnesting items in the `data` array field of the

lonely and giant document provided by the `GET COLLECTION` instruction on line 7. Figure 6b shows an example of one of the documents inserted into the new temporary collection. Notice that the document is similar to the document shown in Figure 4b.

Listing 2. *J-CO-QL* script retrieving data about nitric-oxide sensors.

```
7: GET COLLECTION FROM WEB
"https://www.dati.lombardia.it/resource/ib47-atvt.json?
$limit=1000000&$where=storico<>'S'";

8: EXPAND
UNPACK
WITH ARRAY .data
ARRAY .data
TO .sensor
DROP OTHERS;

9: FILTER
CASE
WHERE    WITH  .sensor.item.idsensore, .sensor.item.comune,
.sensor.item.nometiposensore
AND   .sensor.item.comune           = "Milano"
AND   .sensor.item.nometiposensore  = "Ossidi di Azoto"
GENERATE {
.stationName   : .sensor.item.nomestazione,
.province      : .sensor.item.provincia,
.city          : .sensor.item.comune,
.latitude      : TO_FLOAT (.sensor.item.lat),
.longitude     : TO_FLOAT (.sensor.item.lng),
.altitude      : TO_FLOAT (.sensor.item.quota),
.sensorId      : .sensor.item.idsensore,
.sensorType    : "Nitric Oxide",
.sensorUnit    : .sensor.item.unitamisura
}
DROP OTHERS;

10:   FILTER
CASE
WHERE WITH .latitude, .longitude
GENERATE
SETTING GEOMETRY POINT (.latitude, .longitude)
DROP OTHERS;

11:   SET INTERMEDIATE AS NitricOxide_Sensors;
```

- The `FILTER` instruction on line 9 works in a simpler way to the `FILTER` instruction on line 4. Its `WHERE` condition selects documents that describe sensors for *nitric oxide* (predicate `.sensor.item.tipologia = "Ossidi di Azoto"`) located in Milan (predicate `.sensor.item.comune = "Milano"`). We decided to select sensors in Milan instead of in the province of Milan because the documents in this collection have an extra field called `comune`, which identifies the municipality in which the sensor is located.
The following `GENERATE` action operates in the same way as in line 4; in addition, it adds the extra `city` field to output documents.
Figure 7a shows an example of one document in the new temporary collection. Notice that the document structure is identical to the document shown in Figure 5a apart from the extra `city` field. Original string values are converted into numerical floating-point values, and field names are translated into English from Italian.

```
{
  data = [
    {
      ":@computed_region_6hky_swhk" : "4",
      ":@computed_region_ttgh_9sm5" : "4",
      "comune" : "Truccazzano",
      "datastart":"2006-02-17T00:00:00.000",
      "datastop":"2018-01-01T00:00:00.000",
      "idsensore" : "10006",
      "idstazione" : "684",
      "lat" : "45.48455813667995",
      "lng" : "9.471348388974729",
      "location" : {
        "human_address" : "{\"address\":
  \"\", \"city\": \"\", \"state\": \"\",
  \"zip\": \"\"}",
        "latitude" : "45.48455813667995",
        "longitude" : "9.471348388974729"
      },
      "nomestazione" : "Truccazzano",
      "nometiposensore":"Biossido di Zolfo",
      "provincia" : "MI",
      "quota" : "105",
      "storico" : "S",
      "unitamisura" : "µg/m³",
      "utm_est" : "536835",
      "utm_nord" : "5036889"
    },
    {
      ...
    },
    ...
  ]
}
```


(a)


```
{
  "sensor" : {
    "item" : {
      ":@computed_region_6hky_swhk" : "5",
      ":@computed_region_ttgh_9sm5" : "5",
      "comune" : "Calusco d'Adda",
      "datastart":"2006-06-30T00:00:00.000",
      "idsensore" : "10017",
      "idstazione" : "685",
      "lat" : "45.69042907854514",
      "lng" : "9.484260927647789",
      "location" : {
        "human_address" : "{\"address\":
\"\", \"city\": \"\", \"state\": \"\",
\"zip\": \"\"}",
        "latitude" : "45.69042907854514",
        "longitude" : "9.484260927647789"
      },
      "nomestazione" : "Calusco d' Adda",
      "nometiposensore" : "Ossidi di Azoto",
      "provincia" : "BG",
      "quota" : "273",
      "storico" : "N",
      "unitamisura" : "µg/m³",
      "utm_est" : "537706",
      "utm_nord" : "5059767"
    },
    "position" : 8
    }
}
```


(b)

Figure 6. Example of a document in the temporary collection generated by line 7 (**a**) and by line 8 (**b**). (**a**) Excerpt of the beginning of the document obtained from the portal with data about air-quality sensors. (**b**) Example of a document obtained after unnesting documents received from the portal about air-quality sensors.

- The `FILTER` instruction on line 10, similar to the `FILTER` instruction on line 5, creates the `~geometry` field from the `latitude` and `longitude` fields.
  As an example, the document reported in Figure 7b is derived from the document reported in Figure 7a by adding the `~geometry` field.
- Finally, the `SET INTERMEDIATE AS` instruction on line 11 saves the current temporary collection into the Intermediate Results database *IR* with name `NitricOxide_Sensors`.

The script reported in Listing 2 retrieved 506 documents describing air-quality sensors; among them, five documents that describe sensors for *nitric oxide* located in the city of Milan were selected.

5.3. Building Virtual Sensor Stations

Once the data about the desired sensors were obtained, it was possible to group them based on their closeness. Listing 3 exploits the two intermediate collections, named `Weather_Sensors` and `NitricOxide_Sensors`, temporarily saved into the Intermediate Results database *IR* at the end of Listings 1 and 2, respectively; the goal is building a new collection of *virtual sensor stations*, where each virtual station aggregates one sensor for *nitric oxide* with one sensor for *solar radiation* and one sensor for *rain precipitation* that are close each other. We decided to use the position of sensors for *nitric oxide* as geo-location for *virtual sensor stations* because these are the only sensors located for sure in the city of Milan. Then, having the position of sensors for *nitric oxide* as a central point, we decided to combine them with the *solar radiation* and *rain precipitation* sensors in a radius of 2 *Km*, considering the low number of sensors that are consequently widespread all around the city of Milan.

```
{
  "altitude"    : 118.0,
  "city"        : "Milano",
  "latitude"    : 45.470499014097,
  "longitude"   : 9.19746036011253,
  "province"    : "MI",
  "sensorId"    : "6354",
  "sensorType"  : "Nitric Oxide",
  "sensorUnit"  : "µg/m³",
  "stationName" : "Milano - via Senato"
}
```

(**a**)

```
{
  "altitude"    : 116.0,
  "city"        : "Milano",
  "latitude"    : 45.4633467406665,
  "longitude"   : 9.19532480766886,
  "province"    : "MI",
  "sensorId"    : "6366",
  "sensorType"  : "Nitric Oxide",
  "sensorUnit"  : "µg/m³",
  "stationName" : "Milano - Verziere",
  "~geometry"   : {
    "type"         : "Point",
    "coordinates"  : [
        9.19532480766886,
        45.4633467406665
    ]
  }
}
```

(**b**)

Figure 7. Example of a document in the temporary collection generated by line 9 (**a**) and by line 10 (**b**). (**a**) Example of an air-quality sensor document without geo-reference. (**b**) Example of an air-quality sensor document with geo-reference.

Hereafter, we explain Listing 3, which relies on geometries built by Listings 1 and 2; remember that geometries are represented by the special `~geometry` field.

- The `SPATIAL JOIN` instruction on line 12 shows how *J-CO-QL* performs *spatial operations*; remember that this capability is one of the distinctive features of *J-CO-QL* that we mentioned in Section 3.1.3.
  - The `SPATIAL JOIN` instruction retrieves the intermediate collections produced and stored into the Intermediate Results database *IR* by Listings 1 and 2; they are named `NitricOxide_Sensors` and `Weather_Sensors`, but are respectively aliased as `NO` and as `WS`.
  - Each document *l* in the left `NO` collection is coupled to each document *r* in the right `WS` collection so as to create a document *d* that describes a sensor for *nitric oxide* and a generic weather sensor.
    By exploiting the geometries that are present in the *l* and *r* documents, the spatial-join condition specified by the `ON` clause is evaluated; specifically, the `DISTANCE` function provides the distance between geometries in Km. If the distance is less than 2 Km, the condition is satisfied and the new *d* document is kept for further processing.
  - The *d* document contains three fields: (i) a field named `NO`, which contains the source *l* document; (ii) a field named `WS`, which contains the source *r* document; and (iii) the `~geometry` field, which contains the geometry of the source left *l* document, as specified by the `SET GEOMETRY LEFT` specification.
  - Each *d* document is filtered by the `WHERE` condition in order to select only those documents in which the `WS` field describes a sensor for *solar radiation*. If the *d* document satisfies the `WHERE` condition, the following `GENERATE` action builds a new document *o* that is included in the output temporary collection; the *o* document has only three fields: (i) the `NOsensorId` field identifies the sensor for *nitric oxide*; (ii) the `GRsensorId` field identifies the sensor for *solar radiation*; and (iii) the `NOsensorName` field is a human readable field that denotes the address of the station containing the sensor for *nitric oxide* (it makes it easy for a user to locate the sensor in the city).
  - The `KEEPING GEOMETRY` option maintains the geometry in *d* (that we set to be the one provided by the *l* source document describing the sensor for *nitric oxide* before) for *o*, while the `DROP OTHERS` option discards all *d* documents that do

not satisfy the WHERE condition (i.e., documents that do not describe a sensor for *solar radiation*).

Figure 8a shows an example of documents in the new temporary collection produced by line 12. Notice the few fields and the presence of the ~geometry field.

Listing 3. *J-CO-QL* script building virtual sensor stations.

```
12:   SPATIAL JOIN OF COLLECTIONS
NitricOxide_Sensors AS NO, Weather_Sensors AS WS
ON DISTANCE (KM) < 2
SET GEOMETRY LEFT
CASE
WHERE .WS.sensorType = "Global Radiation"
GENERATE {
.GRsensorId       :  .WS.sensorId,
.NOsensorId       :  .NO.sensorId,
.NOstationName    :  .NO.stationName,
}
KEEPING GEOMETRY
DROP OTHERS;

13:   SET INTERMEDIATE AS SensorPairs;

14:   SPATIAL JOIN OF COLLECTIONS
SensorPairs AS SP, Weather_Sensors AS WS
ON DISTANCE (KM) < 2
SET GEOMETRY LEFT
CASE
WHERE .WS.sensorType = "Rain Precipitation"
GENERATE {
.RPsensorId       :  .WS.sensorId,
.GRsensorId       :  .SP.GRsensorId,
.NOsensorId       :  .SP.NOsensorId,
.NOstationName    :  .SP.NOstationName,
}
KEEPING GEOMETRY
DROP OTHERS;

15:      SET INTERMEDIATE AS SensorTriplets;
```

- The SET INTERMEDIATE AS instruction on line 13 saves the current temporary collection into the Intermediate Results database IR, with name SensorPairs.
- The SPATIAL JOIN instruction on line 14 works very similarly to the SPATIAL JOIN instruction on line 12.
  In this case, the instruction couples each document in the intermediate collection generated by the previous SPATIAL JOIN, named SensorPairs and aliased as SP, to each document in the intermediate collection Weather_Sensors (the documents describe weather sensors), aliased as WS.
  Again, the spatial-join condition is satisfied by geometries for which the distance is less than 2 Km from the *nitric oxide* sensor; the WHERE condition selects only the d documents for which the new WS field describes a sensor for *rain precipitation*. The output o documents generated by the GENERATE action have all of the fields already present in the source left documents (i.e., sensor pairs) plus the RPsensorId field that denotes the identifier of the sensor for *rain precipitation*. Again, the KEEPING GEOMETRY option maintains the geometry in d in o, which represents the position of the sensor for *nitric oxide*, while the DROP OTHERS option discards all d documents that do not satisfy the WHERE condition (i.e., documents that do not describe the desired triplets

of sensors).
Figure 8b shows an example of documents in the new temporary collection. Notice that the structure is identical to the document in Figure 8a apart from the extra `RPsensorId` field.

- The `SET INTERMEDIATE AS` instruction on line 15 saves the current temporary collection into the Intermediate Results database *IR*, with name `SensorTriplets`. The name is motivated by the fact that each document describes a triplet of sensors.

The `SensorTriplets` collection contains six documents, each one describing a virtual station of the three desired sensors, all located in the city of Milan.

```
{
  "GRsensorId"   : "6458",
  "NOsensorId"   : "6312",
  "NOstationName":"Milano - via Juvara",
  "~geometry" : {
    "type" : "Point",
    "coordinates" : [
        9.22231513873747,
        45.4732257085772
    ]
  }
}
```

(a)

```
{
  "GRsensorId"   : "6458",
  "NOsensorId"   : "6312",
  "NOstationName":"Milano - via Juvara",
  "RPsensorId"   : "5908",
  "~geometry" : {
    "type" : "Point",
    "coordinates" : [
        9.22231513873747,
        45.4732257085772
    ]
  }
}
```

(b)

Figure 8. Examples of documents in the collections generated by line 12 (**a**) and by line 14 (**b**). (**a**) Example of documents in the `SensorPairs` collection. (**b**) Example of documents in the `SensorTriplets` collection.

5.4. Retrieving Measurements Made by Weather Sensors

Once *virtual sensor stations* are obtained, it is time to obtain measurements about weather. Listing 4 retrieves measurements from Regione Lombardia Open Data portal in the time window from 08 April 2021 to 15 April 2021. The sub-script is similar to sub-scripts in Listings 1 and 2, although it is a little bit simpler. Hereafter, we present the sub-script.

- The `GET COLLECTION` instruction on line 16 obtains all of the measurements made by weather sensors from Regione Lombardia Open Data portalby sending the request to the dedicated end point [81]. This time, the *Socrata* query (included in the URL after the question mark) is more complex than the ones reported on line 2 and line 7, since we have to retrieve only measurements taken during the time window from 08 April 2021 to 15 April 2021.
  As usual, the data are obtained in *JSON* format as a unique document containing the `data` array, where each item is a document that represents a single measurement.
  Figure 9a shows an excerpt of the obtained document that constitutes the new temporary collection.
- As for line 3 and line 8, the `EXPAND` instruction on line 17 generates a new temporary collection where each document represents a single weather measurement; documents are obtained by unnesting them from the `data` field in the unique document obtained from the portal.
  Figure 9b shows an example of documents in the new temporary collection. Notice that all field names are in Italian and that the numerical values are reported as strings.
- As for line 4 and line 9, the `FILTER` instruction on line 18 transforms the documents in the input temporary collection in order to translate field names into English and convert string values into numerical floating-point values, where necessary. The final structure of documents contains fields that denote (i) the identifier of the sensor (the `sensorId` field), (ii) the `dateTime` of the acquisition and (iii) the `value` of the

measurement (recall that the unit of measure is within sensor descriptions). Figure 9c shows an example of documents in the new temporary collection.

- The `SET INTERMEDIATE AS` instruction on line 19 saves the current temporary collection into the Intermediate Results database *IR*, with name `WeatherMeasures`.

Listing 4. *J-CO-QL* script retrieving data about weather measurements.

```
16:   GET COLLECTION FROM WEB
"https://www.dati.lombardia.it/resource/647i-nhxk.json?$limit=100000000&
$where=data >= '2021-04-01' AND data < '2021-04-15'
AND (stato='VV' OR stato='VA')";

17:   EXPAND
UNPACK
WITH ARRAY .data
ARRAY .data
TO .sensorData
DROP OTHERS;

18:   FILTER
CASE
WHERE WITH .sensorData.item.idsensore
GENERATE {
.sensorId      : .sensorData.item.idsensore,
.dateTime      : .sensorData.item.data,
.value         : TO_FLOAT (.sensorData.item.valore)
}
DROP OTHERS;

19:   SET INTERMEDIATE AS WeatherMeasures;
```

Notice that the *Socrata* query retrieves all weather measurements; in fact, the *Socrata* query does not provide a way to select only measurements detected by a sensor either of a given typology or located in a given area. The reader should not think that this is a limitation provided by *Socrata*: it is due to the fact that the typology of sensors is in the descriptions of the sensors, which are provided by a different end point. Thus, measurements of the desired sensor types must be selected by joining measurements with sensors; the sub-script in Listing 6 shows that the sensors and measurements are joined.

Consequently, since it is not possible to select measurements made by the desired sensors only, it is necessary to download them all. The number of retrieved documents, one for each measurement, retrieved per day can be easily larger than 150,000 items. Since they are represented within a unique giant document; this document cannot be stored into a *MongoDB* database because its binary representation certainly exceeds the upper threshold of 16 MB (for the time window of interest, the size of this document would be around 130 MB). Paradoxically, the best way to manage such a document is in the main memory, as the *J-CO-QL Engine* does. Notice that this is a positive side effect of introducing the *J-CO* Framework, that is, overcoming the limitations of *JSON* document stores.
For the sake of completeness, we downloaded 1,157,188 weather measurements for the time interval from 08 April 2021 to 15 April 2021.

```
{ data = [
    { "idsensore"    : "14062",
      "data":"2021-04-01T03:20:00.000",
      "valore"       : "217",
      "idoperatore"  : "1",
      "stato"        : "VA"
    },
    { "idsensore"    : "8385",
      "data":"2021-04-01T04:50:00.000",
      "valore"       : "-6",
      "idoperatore"  : "1",
      "stato"        : "VA"
    },
    ...
  ]
}
```


(a)


```
{
  "sensorData" : {
    "item" : {
      "data" :"2021-04-01T04:20:00.000",
      "idoperatore"  : "1",
      "idsensore"    : "203",
      "stato"        : "VA",
      "valore"       : "30.5"
    },
    "position" : 2
  }
}
```


(b)


```
{
  "dateTime": "2021-04-01T02:30:00.000",
  "sensorId": "14634",
  "value"   : 22.0
}
```


(c)

Figure 9. Examples of the temporary collections generated by line 16 (**a**), line 17 (**b**) and line 18 (**c**). (**a**) Excerpt of the beginning of the document obtained from the portal with data about weather measurements. (**b**) Example of a document obtained by unnesting data from the portal about weather measurements. (**c**) Example of a restructured document describing weather measurement.

5.5. Retrieving Measurements Made by Air-Quality Sensors

Listing 5 is the counterpart of Listing 4 for downloading data about measurements. Specifically, Listing 5 retrieves measurements about air quality made in the time window from 08 April 2021 to 15 April 2021 from Regione Lombardia Open Data portal.

If we compare the sub-script reported in Listing 5 and the sub-script in Listing 4, we see that they are identical. The only difference is the end point contacted to obtain the measurements. Nevertheless, hereafter we provide a brief description of the sub-script in Listing 5.

- The `GET COLLECTION` instruction on line 20 retrieves all air-quality measurements made in the time window from 08 April 2021 to 15 April 2021 from Regione Lombardia Open Data portal, by sending the request to the dedicated end point [82]. Again, a unique document with an array field named `data` that contains all retrieved measurements is obtained.
  Figure 10a reports the excerpt of this document. Notice that the document structure is identical to the structure of the document shown if Figure 9a.
- The `EXPAND` instruction on line 21 unnests items in the `data` field of the unique document provided by the `GET COLLECTION` instruction. This way, the new temporary collection contains one document for each single retrieved measurement. Figure 10b shows an example document. Notice that the document structure is identical to the structure of the document shown in Figure 9b.
- The `FILTER` instruction on line 22 translates field names into English and converts string values into numerical floating-point values, where necessary.
  Figure 10c shows an example document in the new temporary collection. Notice that the document structure is identical to the structure of the document shown if Figure 9c: this way, documents describing measurements are homogeneous.

- The `SET INTERMEDIATE AS` instruction on line 23 saves the current temporary collection into the Intermediate Results database IR, with name `AirQualityMeasures`.

Listing 5. *J-CO-QL* script retrieving data about air quality measurements.

```
20:   GET COLLECTION FROM WEB
"https://www.dati.lombardia.it/resource/nicp-bhqi.json?$limit=1000000&
$where=data >= '2021-03-15' AND data < '2021-03-23'
AND (stato='VV' OR stato='VA')";

21:   EXPAND
UNPACK
WITH ARRAY .data
ARRAY .data
TO .sensorData
DROP OTHERS;

22:   FILTER
CASE
WHERE WITH .sensorData.item.idsensore
GENERATE {
.sensorId        : .sensorData.item.idsensore,
.dateTime        : .sensorData.item.data,
.value           : TO_FLOAT (.sensorData.item.valore)
}
DROP OTHERS;

23:   SET INTERMEDIATE AS AirQualityMeasures;
```

For the sake of completeness, the sub-script downloaded 50,066 air-quality measurements. As for weather measurements, these cover all of the Lombardia region and all sensor types.

5.6. Aggregating Measurements

The last step to complete Part 1 is to build *measure triplets*, moving from *virtual sensor stations* built by the sub-script in Listing 3. The goal is to obtain documents that describe measurements made at the same time by a sensor of *nitric oxide*, a sensor of *solar radiation* and a sensor of *rain precipitation*, when the three sensors are close to each other.

Listing 6 exploits many of the intermediate collections retrieved in previous parts of the script. In order to achieve its goal, the sub-script extends each measurement of *nitric oxide* with the data about the *virtual sensor station* that contains the sensor for *nitric oxide* that performed the measurement.

Then, for each new document, we build a *measure pair* by adding the measurement detected at the same date and time by the sensor for *solar radiation* included in the same *virtual sensor station*.

Finally, with the same process, a *measure triplet* is built by adding the measurement of *rain precipitation* detected at the same date and time by the sensor for *rain precipitation* in the same *virtual sensor station*.

As the reader can see, the sub-script in Listing 6 applies the `JOIN` instruction three times. To help the reader understand the sub-script, it is worth introducing the `JOIN` instruction before presenting the sub-script.

The `JOIN` instruction works very similarly to the `SPATIAL JOIN` instruction shown in Listing 3. Its goal is to couple documents coming from two collections. Suppose the joined collections are either named or aliased as `L` (standing for "left") and `R` (standing for "right").

Listing 6. *J-CO-QL* script building measure triplets.

```
24:   JOIN OF COLLECTIONS
SensorTriplets AS S, AirQualityMeasures AS M
CASE
WHERE .S.NOsensorId = .M.sensorId
GENERATE {
.dateTime          :  .M.dateTime,
.NOmeasure         :  .M.value,
.NOsensorId        :  .S.NOsensorId,
.GRsensorId        :  .S.GRsensorId,
.RPsensorId        :  .S.RPsensorId,
.NOstationName     :  .S.NOstationName
}
DROP OTHERS;

25:   SET INTERMEDIATE AS AirQualityMeasures_Enriched;

26:   JOIN OF COLLECTIONS
AirQualityMeasures_Enriched AS A, WeatherMeasures AS M
CASE
WHERE .A.GRsensorId  = .M.sensorId  AND
.A.dateTime    = .M.dateTime
GENERATE {
.dateTime          :  .A.dateTime,
.NOstationName     :  .A.NOstationName,
.NOsensorId        :  .A.NOsensorId,
.NOmeasure         :  .A.NOmeasure,
.RPsensorId        :  .A.RPsensorId,
.GRsensorId        :  .A.GRsensorId,
.GRmeasure         :  .M.value
}
DROP OTHERS;

27:   SET INTERMEDIATE AS MeasurePairs;

28:   JOIN OF COLLECTIONS
MeasurePairs AS P, WeatherMeasures AS M
CASE
WHERE .P.RPsensorId  = .M.sensorId  AND
.P.dateTime    = .M.dateTime
GENERATE {
.dateTime          :  .P.dateTime,
.NOstationName     :  .P.NOstationName,
.NOsensorId        :  .P.NOsensorId,
.NOmeasure         :  .P.NOmeasure,
.RPsensorId        :  .P.RPsensorId,
.GRsensorId        :  .P.GRsensorId,
.GRmeasure         :  .P.GRmeasure,
.RPmeasure         :  .M.value
}
DROP OTHERS;
```

```
{ data = [
    { "idsensore"  : "10070",
      "data":"2021-04-01T00:00:00.000",
      "valore"     : "1.4",
      "stato"      : "VA",
      "idoperatore": "1"
    },
    { "idsensore"  : "6685",
      "data":"2021-04-01T00:00:00.000",
      "valore"     : "17.6",
      "stato"      :"VA",
      "idoperatore":"1"
    },
    ...
  ]
}
```

(**a**)

```
{
  "sensorData" : {
    "item" : {
      "data":"2021-04-01T00:00:00.000",
      "idoperatore"  : "1",
      "idsensore"    : "5542",
      "stato"        : "VA",
      "valore"       : "81.1"
    },
    "position" : 12
  }
}
```

(**b**)

```
{
  "dateTime" :"2021-04-01T11:00:00.000",
  "sensorId" : "5828",
  "value"    : 0.9
}
```

(**c**)

Figure 10. Examples of the temporary collections generated by line 20 (**a**), line 21 (**b**) and line 22 (**c**). (**a**) Excerpt of the beginning of the document obtained from the portal with data about air-quality measurements. (**b**) Example of a document obtained by unnesting data about air-quality measurements. (**c**) Example of a restructured document describing air-quality measurement.

Given a document l coming from the left `L` input collection and a document r coming from the right `R` input collection, l and r are coupled by obtaining a new d document. The d document contains two fields: the first one is named `L` and its value is the l document; the second one is named `R` and its value is the r document (the actual field names depends on the actual collection names/aliases).

The `CASE` clause operates on all of the d documents generated by pairing documents in the input collections; as for the `FILTER` and `SPATIAL JOIN` instructions, the inner `WHERE` conditions select and restructure documents, determining which documents to put into the output temporary collection.

Hereafter, we present the sub-script in details.

- The goal of the `JOIN` instruction on line 24 is to join documents in the `SensorTriplets` collection, saved on line 15, with documents in the `AirQualityMeasurements` collection, saved on line 23; the two collections are aliased, respectively, as `S` and as `M`. Consequently, the intermediate d documents contain two fields, named `S` and `M`.
  The documents produced by line 24 extends documents describing *virtual sensor stations* with measurements of *nitric oxide*. In fact, as the reader can see, the `GENERATE` action generates documents with the identifiers of the three grouped sensors, the date and time of measurement, and the *nitric oxide* measurement (the `NOmeasure` field).
  Figure 11a shows an example of documents generated by the first `JOIN` instruction.
- The `SET INTERMEDIATE AS` instruction on line 25 saves the current temporary collection with name `AirQualityMeasures_Enriched` into the Intermediate Results database *IR*.
- The goal of the `JOIN` instruction on line 26 is to further extend documents produced by line 24 with measurements of *solar radiation*.
  In fact, the reader can see that documents generated by the `GENERATE` action have the novel `GRmeasure` field in comparison with documents generated by line 24.
  Figure 11b shows an example of documents generated by the second `JOIN` instruction.

- The `SET INTERMEDIATE AS` instruction on line 27 saves the current temporary collection into the Intermediate Results database IR, with name `MeasurePairs`.
- The last `JOIN` instruction on line 28 further extends documents produced by line 26 with the measurement of *rain precipitation*.
  This way, as specified by the `GENERATE` action, we obtain the final documents to be analyzed with fuzzy sets. Specifically, we have (i) the `dateTime` field, denoting the date and time the measurements were acquired; (ii) the `NOstationName` field, which denotes the name of the station containing the sensor for *nitric oxide*; (iii) the `NOmeasure` and `NOsensorId` fields, denoting the measurement of *nitric oxide* and the identifier of the sensor; (iv) the `GRmeasure` and `GRsensorId` fields, which denote the measurement of *solar radiation* and the identifier of the sensor; and (v) the `RPmeasure` and `RPsensorId` fields, denoting the measurement of *rain precipitation* and the identifier of the sensor. We call these documents *measure triplets*.
  Figure 12a shows an example document.

For the sake of completeness, the script produced 968 *measure triplets*, on which Listing 8 (described in Section 7) performs a soft query.

```
{
  "GRsensorId"  : "6458",
  "NOmeasure"   : 129.1,
  "NOsensorId"  : "6354",
  "NOstationName":"Milano - via Senato",
  "RPsensorId"  : "5908",
  "dateTime": "2021-04-12T12:00:00.000"
}
```

(**a**)

```
{
  "GRmeasure"   : 172.4,
  "GRsensorId"  : "6458",
  "NOmeasure"   : 129.1,
  "NOsensorId"  : "6354",
  "NOstationName":"Milano - via Senato",
  "RPsensorId"  : "5908",
  "dateTime": "2021-04-12T12:00:00.000"
}
```

(**b**)

Figure 11. Example of temporary collection generated after line 24 (**a**) and line 26 (**b**). (**a**) Example of a document describing a *virtual sensor station* extended with the measurement of *nitric oxide*. (**b**) Example of a document describing a *virtual sensor station* extended with the measurements of and *solar radiation*.

```
{
  "GRmeasure"   : 172.4,
  "GRsensorId"  : "6458",
  "NOmeasure"   : 129.1,
  "NOsensorId"  : "6354",
  "NOstationName":"Milano - via Senato",
  "RPmeasure"   : 0.2,
  "RPsensorId"  : "5908",
  "dateTime": "2021-04-12T12:00:00.000"
}
```

(**a**)

```
{
  "GRmeasure"   : 172.4,
  "GRsensorId"  : "6458",
  "NOmeasure"   : 129.1,
  "NOsensorId"  : "6354",
  "NOstationName":"Milano - via Senato",
  "RPmeasure"   : 0.2,
  "RPsensorId"  : "5908",
  "dateTime": "2021-04-12T12:00:00.000",
  "~fuzzysets" : {
    "HighPollution"  : 0.785000011920929,
    "LowRadiation"   : 0.755199984788895,
    "PersistentRain" : 0.799999982118605,
    "Wanted"         : 0.785000011920929
  }
}
```

(**b**)

Figure 12. Examples of documents in the temporary collections generated by line 28 (**a**) and line 32 (**b**). (**a**) Example of a document describing a *measure triplet*. (**b**) Example of a document describing a *measure triplet* after an evaluation of the fuzzy sets.

6. Case Study Part 2: Defining Fuzzy Operators

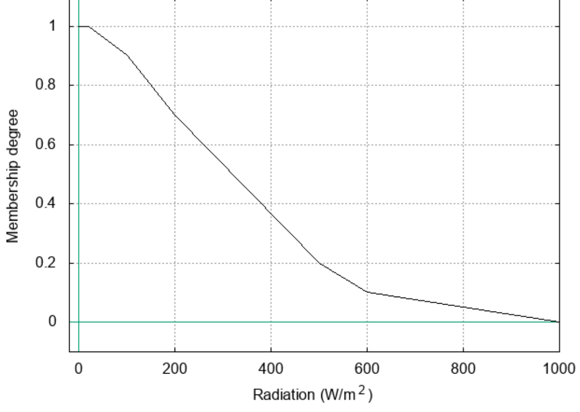
Once Part 1 has produced the temporary collection containing *measure triplets*, it is possible to start defining the fuzzy operators to use in Part 3 to perform the soft selection of *measure triplets*.

J-CO-QL provides a specific construct (called `CREATE FUZZY OPERATOR`) to create novel fuzzy operators so as to use them within queries to evaluate the belonging of *JSON* documents to fuzzy sets. The `CREATE FUZZY OPERATOR` instruction changes the *FO* field in the query-process state (see Definition 7), leaving the other fields untouched; this way, the temporary collection *tc* remains ready for further processing.

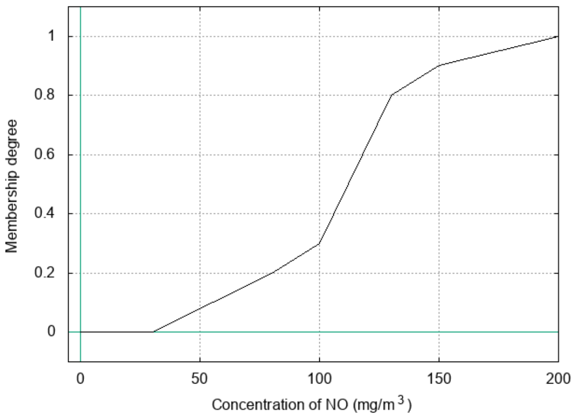
Listing 7 reports the instructions that create three fuzzy operators, named `Has_Low_Radiation`, `Has_High_Pollution` and `Has_Persistent_Rain`. Hereafter, we present them in details.

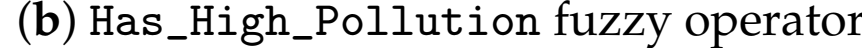
Listing 7. *J-CO-QL* script defining fuzzy operators.

```
29:   CREATE FUZZY OPERATOR Has_Low_Radiation
PARAMETERS
radiation TYPE Float
PRECONDITION
radiation >= 0
EVALUATE radiation
POLYLINE
[  (    0,    1.0),
(   20,    1.0),
(  100,    0.9),
(  200,    0.7),
(  500,    0.2),
(  600,    0.1),
(1000,    0.0)  ];

30:   CREATE FUZZY OPERATOR Has_High_Pollution
PARAMETERS
NO_Concentration TYPE Integer
PRECONDITION
NO_Concentration >= 0
EVALUATE NO_Concentration
POLYLINE
[  (   0, 0.0),
(  30, 0.0),
(  80, 0.2),
(100, 0.3),
(130, 0.8),
(150, 0.9),
(200, 1.0)  ];

31:   CREATE FUZZY OPERATOR Has_Persistent_Rain
PARAMETERS
Precipitation TYPE Float
PRECONDITION
Precipitation >= 0
EVALUATE Precipitation
POLYLINE
[  (0.00,    0.0),
(0.10,    0.1),
(0.15,    0.3),
(0.20,    0.8),
(0.30,    0.9),
(1.00,    1.0)  ];
```

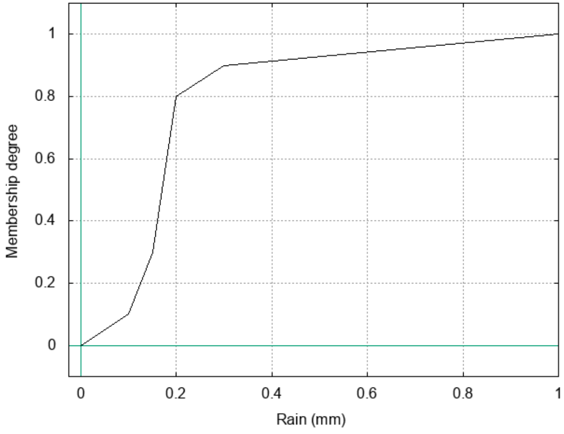


(**a**) `Has_Low_Radiation` fuzzy operator.





(**b**) `Has_High_Pollution` fuzzy operator.



(**c**) `Persistent_Rain` fuzzy operator.

Figure 13. Membership functions of fuzzy operators in Listing 7.

- Let us consider the instruction that defines the `Has_Low_Radiation` operator on line 29.
  - The `PARAMETERS` clause defines the list of formal parameters needed by the fuzzy operator to provide a membership degree. Specifically, the `Has_Low_Radiation` operator receives one formal parameter, named `radiation`, which denotes a measured level of *solar radiation*.
  - The `PRECONDITION` clause specifies a precondition that must be met in order to correctly evaluate the membership degree. In line 29, the `radiation` parameter is considered valid if its value is no less than 0 (negative values of radiation are not meaningful).
  - The `EVALUATE` clause defines the mathematical function whose resulting value is used to get the actual membership degree. In line 29, the evaluation function is the `radiation` parameter itself.
  - The value returned by the function specified in the `EVALUATE` clause is used as x-value for the *membership function*, which provides the corresponding y-value. The membership function is defined as a polyline function, declared by means of the `POLYLINE` clause. The polyline is described by a list of $n+1$ points (x_i, y_i), with $0 \leq i \leq n$, $x_i < x_{i+1}$ and $0 \leq y_i \leq 1$. When $x < x_0$, the corresponding y-value is assumed to be equal to y_0; when $x > x_n$, the corresponding y-value is assumed to be equal to y_n.

The polyline function defined in line 29 is depicted in Figure 13a: if the level of radiation is 100 W/m^2, the membership degree is 0.95; if the radiation is 800 W/m^2, the membership degree is 0.05; if the radiation is 1200 W/m^2, the membership degree is 0 because the radiation is grater than the maximum x-axis value specified in the polyline.

- The second fuzzy operator, defined at line 30, is named `Has_High_Pollution`. Similarly to the definition of the `Has_Low_Radiation` operator, it receives one single parameter, named `NO_Concentration`, which is considered valid by the precondition if its value is no less than 0. The membership polyline function is defined as depicted in Figure 13b.
- The third fuzzy operator, defined on line 31, is called `Has_Persistent_Rain`. It receives one single parameter, named `Precipitation`, which is considered valid by the precondition if its value is no less than 0. The membership polyline function is defined as depicted in Figure 13c.

Notice that the polyline allows users to specify complex membership functions, which is a novel solution if compared to trapezoidal functions proposed in earlier works. In fact, we want to provide users with a powerful tool, by means of which operators with complex semantics could be defined.

This is exactly the same rationale that led us to introduce the `EVALUATE` clause: in fact, fuzzy operators could be used to evaluate complex fuzzy relationships, such as the distance between two points; a further situation could be the need to correct raw input values by means of a correction factor (for example, modifying a temperature on the basis of the altitude).

What happens if the precondition is not satisfied? The evaluation of the fuzzy operator is stopped and the `USING` clause (where fuzzy operators are allowed; see Section 7) is stopped too.

In comparison with classical "crisp" predicates, a fuzzy operator allows for expressing the closeness of compared values to the desired concept. Let consider the `Has_Low_Radiation` operator: a radiation of 100 W/m^2 is not fully low; however, its membership degree of 0.9 says that it is almost low, even though it is not completely low. As a result, soft conditions that rely on that fuzzy operator still consider documents with such a value of radiation, but at the same time, a satisfaction degree that is less than 1 is provided, which denotes how close the selected document is to the desired situation.

7. Case Study Part 3: Detecting Highly Polluted Areas with Low Solar Radiation or Persistent Rain

The final part of the *J-CO-QL* script applies the fuzzy operators defined in Part 2 to the collection of *measure triplets* generated at the end of Part 1, which is still the current temporary collection. As stated by Problem 1, the goal is to discover areas in the city of Milan where we can detect high levels of pollution (in terms of *nitric oxide*) in the presence of either low levels of *solar radiation* or *persistent rain*. Notice the basic bricks of the request: "high pollution", "low radiation" and "persistent rain"; these are "linguistic predicates" that could be partially satisfied by measurements in the *measure triplets*.

The *J-CO-QL* sub-script for Part 3 is reported in Listing 8. To pursue this goal, for each *measure triplet*, the degrees of membership to three different fuzzy sets are evaluated based on the measurements for *nitric oxide*, *solar radiation* and *rain precipitation*by means, respectively, of the `Has_High_Pollution`, `Has_Low_Radiation` and `Has_Persistent_Rain` fuzzy operators. At this point, a complex soft condition can be expressed by means of the fuzzy set names so far evaluated: the resulting membership degree ranks documents in terms of closeness to the linguistic situation expressed by Problem 1. In particular, only relevant documents are kept, i.e., those *measure triplets* for which the membership degree is no less than a given minimum threshold.

Listing 8. *J-CO-QL* script performing the fuzzy analysis.

```
32:   FILTER
CASE
WHERE WITH .NOmeasure, .GRmeasure, .RPmeasure
CHECK FOR FUZZY SET HighPollution
USING Has_High_Pollution (.NOmeasure)
CHECK FOR FUZZY SET LowRadiation
USING Has_Low_Radiation (.GRmeasure)
CHECK FOR FUZZY SET PersistentRain
USING Has_Persistent_Rain (.RPmeasure)
CHECK FOR FUZZY SET Wanted
USING   HighPollution AND (LowRadiation OR PersistentRain)
ALPHA-CUT 0.75 ON Wanted
DROP OTHERS;

33:   FILTER
CASE
WHERE KNOWN FUZZY SETS Wanted
GENERATE {
.stationName   : .NOstatoinName,
.dateTime      : .dateTime,
.rank          : MEMBERSHIP_OF (Wanted)
}
DROPPING ALL FUZZY SETS
DROP OTHERS;

34:   SAVE AS DetectedEvents@INFORMATION_2021;
```

Hereafter, we present the sub-script reported in Listing 8. Notice that no specific instruction for managing fuzzy sets has been introduced; in contrast, only novel clauses in the `FILTER` instruction are sufficient. In particular, a `WHERE` clause (and its `GENERATE` action) can be followed by an optional list of `CHECK FOR FUZZY SET` clauses, which are applied only to those documents selected by the `WHERE` condition and possibly restructured by the `GENERATE` action.

A `CHECK FOR FUZZY SET` clause specifies the name of the fuzzy set to evaluate, and after the `USING` keyword, the soft condition by means of which the membership degree of the document is computed.

The optional `ALPHA-CUT` action specifies a minimum threshold for a fuzzy set: only documents with a membership degree no less than the specified threshold for the specified fuzzy set are selected. Remember from Definition 5 that, in the *J-CO-QL* model, fuzzy-set membership is represented through the special `~fuzzysets` field.
With this premise, we can more easily explain how fuzzy sets are managed by *J-CO-QL*.

- The behavior of the `FILTER` instruction on line 32 is a bit different from what has been shown previously. It receives the temporary collection with *measure triplets* generated on line 28 in Listing 6, which has remained unchanged by the instructions in Part 2 (Listing 7). The `WHERE` condition selects those documents with the fields `NOmeasure`, `GRmeasure` and `RPmeasure`.
  Since there is no `GENERATE` action, the selected documents remain unchanged. The following `CHECK FOR FUZZY SET` clauses allow for adding membership degrees to fuzzy sets to each document that is selected by the `WHERE` condition. All membership degrees are grouped together inside the special root-level field named `~fuzzysets`, as reported in Definition 5.
  Let us discuss the instruction in details. First, remember that the input documents are structured as the sample document reported in Figure 12a.

- Given a document *d* selected by the WHERE condition, the first CHECK FOR FUZZY SET clause evaluates its membership degree to a fuzzy set named HighPollution. The evaluation is performed by means of the fuzzy condition specified after the USING keyword: the fuzzy operator Has_High_Pollution is used to evaluate if the level of *nitric oxide* (the NOmeasure field) is actually high. Since this is a fuzzy condition, it is not true or false but it has a membership degree: in this case, this is the value returned by the fuzzy operator. Since this is the first fuzzy set for which a membership degree is evaluated for *d*, the ~fuzzysets field is not present yet in *d*; thus, the ~fuzzysets field is added to *d* with the HighPollution nested field within, for which the value is the membership degree that results from the evaluation of the USING soft condition. Even in the case the membership degree is 0, the field is created, meaning that the degree of membership to the fuzzy set is known.
- Similarly, the second CHECK FOR FUZZY SET clause evaluates the degree of membership to the fuzzy set named LowRadiation; the fuzzy operator Has_Low_Radiation is applied on the level of *solar radiation* (denoted by the GRmeasure field) in the USING soft condition.
  Since now *d* already has the ~fuzzysets field, within it, a novel field named LowRadiation is added, for which the value is the membership degree resulting from the evaluation of the USING condition.
- The third CHECK FOR FUZZY SET clause evaluates the degree of membership to the fuzzy set named PersistentRain; the USING soft condition applies the fuzzy operator named Has_Persistent_Rain to the level of *rain precipitation* (denoted by the RPmeasure field).
  A new nested field is added to the ~fuzzysets field at the root level of *d*: this field is named PersistentRain (as the name of the evaluated fuzzy set) and its value is the result of the evaluation of the USING soft condition.
- The fourth CHECK FOR FUZZY SET clause evaluates the membership degree of *d* to the fuzzy set named Wanted; the name was chosen since it expresses whether the documents are actually the documents wanted by the user. In fact, the higher the membership degree, the closer they are to what the user wanted.
  What actually does the user want? Recalling Problem 1, the user wants the *measure triplets* that denote a high-level of pollution in conjunction with either low *solar radiation* or *persistent rain*. Since the three membership degrees for the fuzzy sets named HighPollution, LowRadiation and PersistentRain have already been evaluated, the USING soft condition is expressed by composing these three fuzzy sets:
  ```
  HighPollution AND (LowRadiation OR PersistentRain)
  ```
  The AND and OR operators are redefined according to their semantics for fuzzy sets (see Section 2.1): the AND operator returns the minimum membership degree of operands, while the OR operator returns the maximum.
  Notice how the fuzzy-set names play the role of "linguistic predicates", i.e., they express the desired documents in a soft way; the underlying membership degrees allow for imprecise matching, tolerant to incomplete satisfaction.
  The new Wanted field is added to the root-level ~fuzzysets field in *d*; its value is the membership degree resulting from the evaluation of the USING soft condition.
- The user is interested only in documents that are close enough to the wanted ones, thus documents that belong to the Wanted fuzzy set with a too low membership degree are not of interest. The ALPHA-CUT action discards all documents for which the degree of membership to the Wanted fuzzy set is less than the threshold of 0.75. This way, only documents that represent *measure triplets* that are sufficiently close to the desired ones are put into the output temporary collection.

What happens if the precondition of the fuzzy operators is not satisfied? The evaluation of the USING soft condition is stopped and the document is no longer modified,

i.e., it remains either as the WHERE condition selected it or as the GENERATE action (if present) restructured it.
Figure 12b shows the same document reported in Figure 12a after the evaluation of all the four fuzzy sets: notice the presence of the ~fuzzysets field and, within it, the four fields corresponding to each single fuzzy set; their value is the membership degree of the document. Notice that the document does not completely match the soft condition; however, its membership degree for the Wanted fuzzy set is 0.785, meaning that it could be still of interest for the user, in that it can be considered sufficiently close to the user needs.

- The FILTER instruction on line 33 slightly restructures the documents coming from the previous FILTER instruction in order to generate the final data set.
  - The WHERE condition makes use of the KNOWN FUZZY SETS predicate: this is a Boolean predicate that is true if, for the evaluated document, the degree of membership to the specified fuzzy set(s) is known (i.e., a field with the same name appears in the root-level ~fuzzysets field). In this case, documents for which the membership to the Wanted fuzzy sets has been evaluated are selected. Recall that the WHERE condition is still a Boolean condition; thus, the KNOWN FUZZY SETS predicate is a kind of bridge between the Boolean world and the fuzzy world.
  - For each selected document, the GENERATE action builds a new document reporting the station name (the stationName field) and the date and time of the measurements (the dateTime field); furthermore, the membership degree to the Wanted fuzzy set becomes the value of the new rank field. In this regard, notice the expression rank : MEMBERSHIP_OF(Wanted), in which the MEMBERSHIP_OF function is used to obtain the membership degree of the specified fuzzy set. The goal of the rank field is to provide a rank for each single document in relation to the soft selection condition.
  - Finally, the DROPPING ALL FUZZY SETS option removes the special ~fuzzysets field. This way, documents are completely ”de-fuzzified”, i.e., they become traditional crisp documents again; the only trace of the fuzzy process is the rank field.

  Figure 14 shows the same document reported in Figure 12b after it has been restructured by the GENERATE action and after the DROPPING ALL FUZZY SETS option.
- Finally, the SAVE AS instruction on line 34 persistently saves the current temporary collection into the *MongoDB* database named INFORMATION_2021 (that was declared for use on line 1 of Listing 1); the collection is saved and named DetectedEvents.

```
{
  "dateTime":"2021-04-12T12:00:00.000",
  "rank"        : 0.785000011920929,
  "stationName" : "Milano - via Senato"
}
```

Figure 14. Example of a document in the final collection generated by Listing 8.

Considering the time window from 08 April 2021 to 15 April 2021, the final collection contains 35 documents, each representing a single event; the membership degree of all documents varies in the range 0.755 to 0.949, meaning that no document fully belongs to the Wanted fuzzy set; out of the 35 detected events, none reported a membership degree of 1 to at least one of the intermediate fuzzy sets HighPollution, LowRadiation and PersistentRain. On the contrary, for 33 events, the membership degree to the PersistentRain fuzzy set is 0, while they have a high membership degree (greater than 0.75) to the LowRadiation fuzzy set (this is why they appear in the final result, even though they do not fully satisfy the condition). This means that the soft approach was effective in retrieving possibly interesting documents: definitely, the document with 0.949 (very

close to 1) as the membership degree is of interest to the user. Consequently, the reader can notice the effectiveness of the fuzzy approach; a crisp approach able to obtain the same results would be cumbersome and much less intuitive.

The rationale behind the fuzzy extension of the `FILTER` instruction can be explained by the following discussion.

- Instead of considering a Boolean condition as a particular case of a soft condition, we propose a solution where the Boolean world (the `WHERE` Boolean condition) and the fuzzy world (the `USING` soft condition) are clearly separated.
- To be more precise, the Boolean world has a higher priority than the fuzzy world, i.e., (i) the Boolean condition selects documents to work on; then, (ii) the fuzzy world evaluates their belonging to fuzzy sets.
- The key point is exactly our view that a document can belong to many fuzzy sets. Necessarily, evaluating the belonging to fuzzy sets moves from fields in the documents.
- The *adaptability to heterogeneity* feature that characterizes *J-CO-QL* instructions is the reason why the `CASE` clause admits more than one `WHERE` condition: the language must be able to deal with many different structures of documents at the same time.
- However, documents structured in different ways should be treated differently as far as the evaluation of their belonging to fuzzy sets is concerned. Consequently, the list of `CHECK FOR FUZZY SET` clauses possibly followed by the `ALPHA-CUT` action must be necessarily considered conditioned by the `WHERE` condition: first, documents of interest are selected, in a traditional way, by the `WHERE` Boolean condition; then, on these documents, their belonging to as many fuzzy sets as necessary is evaluated; in practice, we further extend the documents by adding or updating the special `~fuzzysets` field.
- The `KNOWN FUZZY SETS` predicate we added to the `WHERE` condition is, in some sense, the bridge between the two worlds because it allows for selecting documents (on which to possibly further evaluate their belonging to other fuzzy sets) by accessing the fuzzy side of documents in a Boolean way.

The explanation of the physical meaningfulness of the detected results as well as the conceptualization of more meaningful experiments is beyond the scope of this paper, and it is demanded to domain experts in this specific subject.

Nevertheless, as stated in Section 2.2, the discussion above explains the novelty of the approach followed in *J-CO-QL* in comparison with proposals for flexible query languages on relational databases and on NoSQL data sets. In this sense, *J-CO-QL* provides a unique and innovative view to the adoption of fuzzy sets in query languages.

8. What Would Happen without *J-CO*?

Are *J-CO* and *J-CO-QL* really necessary? Would it be possible to perform the same task with traditional technology? These questions can naturally arise. Specifically, we can consider the following issues.

- We treated the documents provided by the Open Data portal as if they were flat documents. In this sense, it could be the correct thinking that a data set provided in the CSV (*Comma Separated Values*) format, instead of *JSON* format, would be easier to manage. However, if the reader pays attention to the example documents provided by the Open Data portal in Figures 4a and 6a, he/she can notice that documents are not actually flat, since they both have a structured field named `location`, holding geographic information. In these cases, we avoided access to its field called `longitude` and `latitude` (denoted as `.location.longitude` and `.location.latitude`), since there were the `lng` and `lat` fields at the root level, reporting the same values for longitude and latitude. Thus, the issue is as follows: what if the data sets are a set of structured documents? Would a CSV format be (more) suitable for a complex manipulation?

- Could a relational DBMS be used to integrate data about sensors and to build *measure triplets*, once data are provided as CSV files?
- Are fuzzy sets and soft querying really necessary? At the end, data are provided as crisp values, without uncertainty. Why should they be treated by means of fuzzy sets?

The answer is that, for the case study, the same process could be performed without *J-CO*. Hereafter, we sketch how the process could be performed with CSV and SQL.

1. Open Data provided by the portal as CSV files are saved into files on the analyst's PC.
2. By means of wizards provided by the specific relational DBMS, CSV files are uploaded into tables within the relational database.
3. Cleaning and pre-processing activities are performed by writing SQL queries that create other tables. The alternative option could be to create "views" in place of novel tables.
4. Triplets should be built by performing `INNER JOIN` operations in which the join conditions are based on the geodetic distance between points. If the DBMS used is not equipped with extensions to deal with spatial data, it is necessary to write a "User Defined Function" (UDF) with the procedural code to compute the distance.
5. Once a table containing the *measure triplets* is built (and made persistent in the database), it is necessary to write another SQL query that somehow ranks them. In fact, simply cutting values of attributes based on a crisp comparison is not equivalent to the *J-CO-QL* script so far presented because this approach does not give a relevance to tuples based on attribute values.
   For example, if we write `NOmeasure >= 30`, we give equal importance to values 30 and 220; however, 30 denotes a low level of pollution while 220 denotes a very high level of pollution.
6. In order to replicate the same semantics of the fuzzy operators reported in Listing 7 and discussed in Section 6, it is necessary to write a specific User Defined Function for each of them: a procedural code can easily perform the same computation.
7. Finally, an SQL query adds a novel attribute by computing an expression that exploits the User Defined Functions; the resulting value should be the rank of each single row with respect to the wishes of the user; a final selection filters only relevant rows.

Consequently, we can make the following considerations, which illustrate the advantages provided by the *J-CO* Framework.

- The process based on CSV and SQL is quite cumbersome. In particular, the need to create persistent tables within the database to store temporary results creates confusion. Part 1 of the *J-CO-QL* script (presented in Section 5) performs exactly the same activities but in a homogeneous way and working directly on the data sets received from the portal, with no need to save them on disk and importing them in another system. Therefore, activities are performed in a more natural way.
- In the relational setting, the geodetic distance can be computed by writing a specific User Defined Function; this fact asks for skills in procedural programming.
  The `SPATIAL JOIN` instruction provided by *J-CO-QL* performs this operation in a native way; no skills in procedural programming are needed.
- In the relational setting, User Defined Functions can be written in place of fuzzy operators. Apart from the need for skills in procedural programming, this approach makes what the analyst is called to do semantically unclean.
  In fact, the declarative approach provided by the `CREATE FUZZY OPERATOR` instruction in *J-CO-QL* lets the analyst focus on the actual semantics to provide single linguistic predicates.
- The adoption of fuzzy sets and soft conditions to rank data in *J-CO-QL* allows analysts to focus on specifying what they actually wish to look for.
- Finally, if portals provide non-flat data possibly containing arrays of documents, as often occurs with Social Media APIs, the relational setting becomes no longer practicable (see, for example, [22]).

We can conclude this discussion by claiming that: (i) when *JSON* documents provided by the portal represent flat data, *J-CO* provides a more flexible and intuitive way of processing data than the relational setting; (ii) the soft approach based on fuzzy operators and fuzzy sets provided by *J-CO-QL* constitutes a mean to rank data in a clear and intuitive way because viewing crisp data as items in fuzzy sets enables expressing complex wishes and ranking items on the basis of how close they are to the wished ones; and (iii) if data are not flat, *J-CO-QL* is natively able to deal with them, while the relational setting becomes not practicable. Consequently, the *J-CO* Framework and *J-CO-QL* provides a significant improvement with regard to traditional settings.

9. Conclusions and Future Work

We are at the end of the paper. In this section, we summarize the contribution of the paper and we sketch future developments.

9.1. Conclusions

The best way to illustrate the benefits of adopting a novel technology to manage data sets is to show its application to a practical case study. This is exactly what we did in this paper.

Specifically, we started with real-life data sets actually published by Regione Lombardia Open Data portal; these data sets describe environmental sensors and measurements. Based on these data sets, we identified a case study, summarized by Problem 1.

We then presented a non-trivial *J-CO-QL* script, the goal of whcih was to show the capabilities of the current stage of the *J-CO-QL* language. Specifically, the contributions of the paper are resumed thereafter.

- We showed the capability of *J-CO-QL* in directly retrieving data from Open Data portals and in processing them, performing complex cross-analysis and data integration tasks, in a totally integrated way. Part 1 demonstrated that there is no need to rely on a *JSON* document store such as *MongoDB* to acquire data and to perform complex transformations on them.
- The fuzzy extension of *J-CO-QL* provides a specific construct to define "fuzzy operators", i.e., operators to be used to evaluate the degree of membership to fuzzy sets of *JSON* documents. Part 2 showed that fuzzy operators can be defined by users; they are provided with a powerful tool that allows them to define complex ways to compute membership degrees. Our proposal significantly improves previous proposals, usually limited to simpler trapezoidal functions.
- We presented the extension made to the `FILTER` instruction, specifically the fuzzy clauses that follow a `WHERE` clause, to evaluate the degrees of membership to fuzzy sets of documents. Part 3 showed the flexibility of the language in dealing with multiple fuzzy sets at the same time so as to express soft selection conditions on documents; furthermore, the membership degree can be used to rank documents, where the rank represents how close a document is in relation to the situation desired by the user.

Altogether, these features make *J-CO-QL* (and the *J-CO* Framework) a unique tool in the panorama of *JSON* query languages and data management tools. Demonstrating these unique and innovative capabilities is the main contribution of this paper.

This paper does not evaluate execution performance. In [3], we performed an extensive evaluation, so the interested reader can refer to that paper. Nevertheless, we want to report some synthetic data about performance, as far as the *J-CO-QL* script presented in this paper is concerned. By executing the script on a laptop PC powered by a Processor Intel quad-Core i7-8550-U, running at 1.80 GHz, equipped with 16 GB RAM and 250 GB Solid State Drive and running the Java Virtual Machine version 1.8.0_251 (the *J-CO-QL Engine* is written in the Java programming language), we observed the following execution times: Part 1 takes around 40 min, Part 2 takes 1 ms and Part 3 takes about 300 ms. In fact, Part 1 contains instructions, such as `EXPAND`, `SPATIAL JOIN` and `JOIN`, that either operate on a large number of documents (such as the `EXPAND` instruction on line 17 that manages more

than 1.1 Million documents) or are intrinsically computationally expensive (such as `JOIN` and `SPATIAL JOIN`). Creating fuzzy operators is almost immediate (Part 2 takes 1 ms in total) because the temporary collection remains unchanged. Finally, the time needed by Part 3 is negligible (only 300 ms, which is nothing in comparison with 40 min taken by Part 1) because the `FILTER` instructions operate linearly on the data set.

Consequently, we can say that the execution times are not affected by the fuzzy extension to the `FILTER` instruction; this is another element that confirms the feasibility of the idea of extending *J-CO-QL* with fuzzy constructs. Nevertheless, the overall execution time of 40 min is absolutely acceptable for analysts, if we consider the very complex task performed by the proposed *J-CO-QL* script.

9.2. Future Work

J-CO is an ongoing research project where we try to address known and novel problems concerned with *JSON* data management. We follow a step-by-step approach, i.e., the *J-CO-QL* language is continuously extended and improved by means of minor and major changes, and the overall framework is continuously extended with novel tools in order to improve the practical applicability of the framework.

Consequently, we have in mind many directions to work on in order to extend the *J-CO* Framework.

- As far as the *J-CO-QL* language is concerned, we are going to further improve the support for fuzzy concepts. First, we will consider the `JOIN` and `SPATIAL JOIN` instructions; this latter one, in particular, has to apply fuzzy concepts to deal with geometrical properties.
- As far as the *J-CO-QL Engine* is concerned, we will explore the adoption of indexes and spatial indexes computed on the fly to improve computationally heavy operations such as `JOIN` and `SPATIAL JOIN`, towards the possible application of the *J-CO* Framework to manage Big Data.
- The need to continuously gather data from Open Data portals, which are data sets that could disappear from the portals, inspired us the notion of "virtual database", i.e., a tool that provides a database view of data sets provided by Open Data portals. We only have an idea, but we are going to develop that idea.
- Another family of tools that can be added to the framework encompasses domain-specific languages and processors for managing specific JSON formats, such as *GeoJSON*; we performed the first step in this direction in [71], but we plan to complete that proposal and to extend the approach to other formats. The goal is to exploit *J-CO-QL* as the basis for implementing high-level tools.

As the reader can see, the *J-CO* project is only at the beginning of its (hopefully) long-term development. The *J-CO* Framework will be shortly available on a public GitHub repository [83].

Author Contributions: Conceptualization and methodology, G.P.; software, P.F.; writing—original draft preparation, P.F. and G.P.; writing—review and editing, P.F. and G.P. Both authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data are available from 8 June 2021 at the URL. https://github.com/zunstraal/J-Co-Project/ (accessed on 8 June 2021) in folder `paperData→MPDI Information 2021`.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Khan, M.A.; Uddin, M.F.; Gupta, N. Seven V's of Big Data understanding Big Data to extract value. In Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education, Bridgeport, CT, USA, 3–5 April 2014; pp. 1–5.
2. Bray, T. The Javascript Object Notation (JSON) Data InterchangeFormat. 2014. Available online: https://www.rfc-editor.org/rfc/rfc7159.txt (accessed on 22 April 2021).
3. Psaila, G.; Fosci, P. J-CO: A Platform-Independent Framework for Managing Geo-Referenced JSON Data Sets. *Electronics* **2021**, *10*, 621. [CrossRef]
4. Bordogna, G.; Psaila, G. Customizable flexible querying in classical relational databases. In *Handbook of Research on Fuzzy Information Processing in Databases*; IGI Global: Hershey, Pennsylvania, USA, 2008; pp. 191–217.
5. Psaila, G.; Marrara, S. A First Step Towards a Fuzzy Framework for Analyzing Collections of JSON Documents. In Proceedings of the 16th IADIS International Conference on Applied Computing 2019, Cagliari, Italy, 7–9 November 2019; pp. 19–28.
6. Zadeh, L.A. The concept of a linguistic variable and its application to approximate reasoning—I. *Inform. Sci.* **1975**, *8*, 199–249. [CrossRef]
7. MongoDb. 2021. Available online: https://www.mongodb.com/ (accessed on 22 April 2021).
8. Chodorow, K. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*; O'Reilly Media, Inc.: Newton, MA, USA, 2013.
9. CouchDb. 2021. Available online: https://couchdb.apache.org/ (accessed on 22 April 2021).
10. Anderson, J.C.; Lehnardt, J.; Slater, N. *CouchDB: The Definitive Guide: Time to Relax*; O'Reilly Media: Newton, MA, USA, 2010.
11. Bringas, P.G.; Pastor, I.; Psaila, G. Can BlockChain technology provide information systems with trusted database? The case of HyperLedger Fabric. In Proceedings of the International Conference on Flexible Query Answering Systems, Amantea, Italy, 2–5 July 2019; Springer: Cham, Switzerland, 2019; pp. 265–277.
12. Garcia Bringas, P.; Pastor-López, I.; Psaila, G. BlockChain Platforms in Financial Services: Current Perspective. *Bus. Syst. Res. Int. J. Soc. Adv. Innov. Res. Econ.* **2020**, *11*, 110–126.
13. Nayak, A.; Poriya, A.; Poojary, D. Type of NOSQL databases and its comparison with relational databases. *Int. J. Appl. Inf. Syst.* **2013**, *5*, 16–19.
14. Ong, K.W.; Papakonstantinou, Y.; Vernoux, R. The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases. *arXiv* **2014**, arXiv:1405.3631.
15. Florescu, D.; Fourny, G. JSONiq: The history of a query language. *IEEE Internet Comput.* **2013**, *17*, 86–90. [CrossRef]
16. Cattell, R. Scalable SQL and NoSQL data stores. *ACM Sigmod Rec.* **2011**, *39*, 12–27. [CrossRef]
17. Beyer, K.S.; Ercegovac, V.; Gemulla, R.; Balmin, A.; Eltabakh, M.; Kanne, C.C.; Ozcan, F.; Shekita, E.J. Jaql: A scripting language for large scale semistructured data analysis. *Proc. VLDB Endow.* **2011**, *4*, 1272–1283. [CrossRef]
18. Chamberlin, D. SQL++ For SQL Users: A Tutorial. 2018. Available online: http://asterixdb.apache.org/files/SQL_Book.pdf (accessed on 22 April 2021).
19. Chamberlin, D. XQuery: An XML query language. *IBM Syst. J.* **2002**, *41*, 597–615. [CrossRef]
20. Arora, R.; Aggarwal, R.R. Modeling and querying data in mongodb. *Int. J. Sci. Eng. Res.* **2013**, *4*, 141–144.
21. Bordogna, G.; Capelli, S.; Psaila, G. A big geo data query framework to correlate open data with social network geotagged posts. In Proceedings of the The 20th AGILE International Conference on Geographic Information Science, Wageningen, The Netherlands, 9–12 May 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 185–203.
22. Bordogna, G.; Capelli, S.; Ciriello, D.E.; Psaila, G. 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-Spat. Inf. Sci.* **2018**, *21*, 257–271. [CrossRef]
23. Blair, D.C. Information Retrieval, 2nd ed. C.J. Van Rijsbergen. London: Butterworths; 1979: 208 pp. Price: $32.50. *J. Am. Soc. Inf. Sci.* **1979**, *30*, 374–375. [CrossRef]
24. Bosc, P.; Prade, H. 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*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 285–324.
25. Medina, J.M.; Pons, O.; Vila, M.A. Gefred: A generalized model of Fuzzy Relational Databases. *Inform. Sci.* **1994**, *76*, 87–109. [CrossRef]
26. Galindo, J.; Urrutia, A.; Piattini, M. *Fuzzy Databases: Modeling, Design, and Implementation*; IGI Global: Hershey, PA, USA, 2006.
27. Galindo, J. New characteristics in FSQL, a fuzzy SQL for fuzzy databases. *WSEAS Trans. Inf. Sci. Appl.* **2005**, *2*, 161–169.
28. Kacprzyk, J.; Zadrożny, S. FQUERY for Access: Fuzzy Querying for a Windows-Based DBMS. In *Fuzziness in Database Management Systems*;Springer: Berlin/Heidelberg, Germany, 1995; Volume 5, pp. 415–433.
29. Bordogna, G.; Psaila, G. Modeling soft conditions with unequal importance in fuzzy databases based on the vector p-norm. In Proceedings of the International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU), Malaga, Spain, 22–27 June 2008.
30. Bordogna, G.; Psaila, G. Soft Aggregation in Flexible Databases Querying based on the Vector p-norm. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.* **2009**, *17*, 25–40. [CrossRef]
31. Bosc, P.; Pivert, O. SQLf: A relational database language for fuzzy querying. *IEEE Trans. Fuzzy Syst.* **1995**, *3*, 1–17. [CrossRef]
32. Bosc, P.; Pivert, O. SQLf query functionality on top of a regular relational database management system. In *Knowledge Management in Fuzzy Databases*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 171–190.

33. Galindo, J.; Medina, J.M.; Pons, O.; Cubero, J.C. A server for fuzzy SQL queries. In Proceedings of the International Conference on Flexible Query Answering Systems, Roskilde, Denmark, 13–15 May 1998; Springer: Berlin/Heidelberg, Germany, 1998; pp. 164–174.
34. Zadrozny, S.; Kacprzyk, J. Fquery for access: Towards human consistent querying user interface. In Proceedings of the 1996 ACM symposium on Applied Computing, Philadelphia, PA, USA, 17–19 February 1996; pp. 532–536.
35. Kacprzyk, J.; Zadrozny, S. SQLf and FQUERY for Access. In Proceedings of the Joint 9th IFSA World Congress and 20th NAFIPS International Conference (Cat. No. 01TH8569), Vancouver, BC, Canada, 25–28 July 2001; Volume 4, pp. 2464–2469.
36. Urrutia, A.; Tineo, L.; Gonzalez, C. FSQL and SQLf: Towards a standard in fuzzy databases. In *Handbook of Research on Fuzzy Information Processing in Databases*; IGI Global: Hershey, PA, USA, 2008; pp. 270–298.
37. Ma, Z.M.; Yan, L. Generalization of strategies for fuzzy query translation in classical relational databases. *Inform. Softw. Technol.* **2007**, *49*, 172–180. [CrossRef]
38. Galindo, J. *Handbook of Research on Fuzzy Information Processing in Databases*; IGI Global: Hershey, PA, USA, 2008.
39. Bordogna, G.; Pasi, G. Linguistic aggregation operators of selection criteria in fuzzy information retrieval. *Int. J. Intell. Syst.* **1995**, *10*, 233–248. [CrossRef]
40. Kraft, D.H.; Colvin, E.; Bordogna, G.; Pasi, G. Fuzzy information retrieval systems: A historical perspective. In *Fifty Years of Fuzzy Logic and Its Applications*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 267–296.
41. Kraft, D.H.; Petry, F.E. Fuzzy information systems: Managing uncertainty in databases and information retrieval systems. *Fuzzy Sets Syst.* **1997**, *90*, 183–191. [CrossRef]
42. Cheng, J.; Ma, Z.M.; Yan, L. f-SPARQL: A flexible extension of SPARQL. In Proceedings of the International Conference on Database and Expert Systems Applications (DEXA), Bilbao, Spain, 30 August–3 September 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 487–494.
43. Lopez-Pellicer, F.J.; Silva, M.J.; Chaves, M.; Zarazaga-Soria, F.J.; Muro-Medrano, P.R. Geo linked data. In Proceedings of the International Conference on Database and Expert Systems Applications (DEXA), Bilbao, Spain, 30 August–3 September 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 495–502.
44. Pérez, J.; Arenas, M.; Gutierrez, C. Semantics and complexity of SPARQL. *ACM Trans. Database Syst. (TODS)* **2009**, *34*, 1–45. [CrossRef]
45. De Maio, C.; Fenza, G.; Furno, D.; Loia, V. f-SPARQL extension and application to support context recognition. In Proceedings of the 2012 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), Brisbane, Australia, 10–15 June 2012; pp. 1–8.
46. Pivert, O.; Slama, O.; Thion, V. An extension of SPARQL with fuzzy navigational capabilities for querying fuzzy RDF data. In Proceedings of the 2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), Vancouver, BC, Canada, 24–29 July 2016; pp. 2409–2416.
47. Castelltort, A.; Laurent, A. Fuzzy queries over NoSQL graph databases: Perspectives for extending the cypher language. In Proceedings of the International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU), Montpellier, France, 15–19 July 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 384–395.
48. Abir, B.K.; Amel, G.T. Towards fuzzy querying of NoSQL document-oriented databases. In Proceedings of the DBKDA 2015: The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications, Rome, Italy, 24–29 May 2015; pp. 153–158.
49. Mehrab, F.; Harounabadi, A. Apply Uncertainty in Document-Oriented Database (MongoDB) Using F-XML. *J. Adv. Comput. Res.* **2018**, *9*, 87–101.
50. Almendros-Jimenez, J.M.; Becerra-Teron, A.; Moreno, G. Fuzzy queries of social networks with FSA-SPARQL. *Expert Syst. Appl.* **2018**, *113*, 128–146. [CrossRef]
51. Bordogna, G.; Campi, A.; Psaila, G.; Ronchi, S. A language for manipulating clustered web documents results. In Proceedings of the 17th ACM on Conference on Information and Knowledge Management, Napa Valley, CA, USA, 26–30 October 2008; pp. 23–32.
52. Bordogna, G.; Campi, A.; Psaila, G.; Ronchi, S. An interaction framework for mobile web search. In Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia, Linz, Austria, 24–26 November 2008; pp. 183–191.
53. Fosci, P.; Psaila, G. Toward a Product Search Engine based on User Reviews. In Proceedings of the International Conference on Data Technologies and Applications (DATA-2012), Rome, Italy, 25–27 July 2012; pp. 223–228.
54. Fosci, P.; Psaila, G. Finding the best source of information by means of a socially-enabled search engine. In Proceedings of the KES 2012-COnference on in Knowledge-Based and Intelligent Information and Engineering Systems, San Sebastian Spain, 10–12 September 2012; IOS Press: Amsterdam, The Netherlands, 2012; Volume 243, pp. 1253–1262.
55. Fosci, P.; Psaila, G.; Di Stefano, M. Hints from the Crowd: A Novel NoSQL Database. In Proceedings of the International Conference on Model and Data Engineering, Amantea, Italy, 25–27 September 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 118–131.
56. Fosci, P.; Psaila, G.; Di Stefano, M. The hints from the crowd project. In Proceedings of the International Conference on Database and expert Systems Applications, Prague, Czech Republic, 26–29 August 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 443–453.
57. Pelucchi, M.; Psaila, G.; Toccu, M. Building a Query Engine for a Corpus of Open Data. In Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017), Porto, Portugal, 25–27 April 2017; pp. 126–136.

58. Pelucchi, M.; Psaila, G.; Toccu, M. Enhanced Querying of Open Data Portals. In Proceedings of the International Conference on Web Information Systems and Technologies, Barcelona, Spain, 4–6 October 2017; Springer: Cham, Switzerland, 2017; pp. 179–201.
59. Pelucchi, M.; Psaila, G.; Toccu, M. The Challenge of using Map-reduce to Query Open Data. In Proceedings of the DATA-2017 6th International Conference on Data Science, Technology and Applications, Madrid, Spain, 24–26 July 2017; pp. 331–342.
60. Pelucchi, M.; Psaila, G.; Toccu, M. Hadoop vs. Spark: Impact on Performance of the Hammer Query Engine for Open Data Corpora. *Algorithms* **2018**, *11*, 209. [CrossRef]
61. Marrara, S.; Pelucchi, M.; Psaila, G. Blind Queries Applied to JSON Document Stores. *Information* **2019**, *10*, 291. [CrossRef]
62. Cuzzocrea, A.; Psaila, G.; Toccu, M. Knowledge discovery from geo-located tweets for supporting advanced big data analytics: A real-life experience. In *Model and Data Engineering*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 285–294.
63. Cuzzocrea, A.; Psaila, G.; Toccu, M. An innovative framework for effectively and efficiently supporting big data analytics over geo-located mobile social media. In Proceedings of the 20th International Database Engineering & Applications Symposium, Montreal, QC, Canada, 11–13 July 2016; pp. 62–69.
64. Bordogna, G.; Cuzzocrea, A.; Frigerio, L.; Psaila, G.; Toccu, M. An interoperable open data framework for discovering popular tours based on geo-tagged tweets. *Int. J. Intell. Inf. Database Syst.* **2017**, *10*, 246–268. [CrossRef]
65. Bordogna, G.; Frigerio, L.; Cuzzocrea, A.; Psaila, G. Clustering geo-tagged tweets for advanced big data analytics. In Proceedings of the 2016 IEEE International Congress on Big Data (BigData Congress), San Francisco, CA, USA, 27 June–2 July 2016; pp. 42–51.
66. Burini, F.; Cortesi, N.; Gotti, K.; Psaila, G. The Urban Nexus Approach for Analyzing Mobility in the Smart City: Towards the Identification of City Users Networking. *Mobile Inform. Syst.* **2018**, *2018*. [CrossRef]
67. Burini, F.; Cortesi, N.; Psaila, G. *From Data to Rhizomes: Applying a Geographical Concept to Understand the Mobility of Tourists from Geo-Located Tweets*; Multidisciplinary Digital Publishing Institute: Basel, Switzerland, 2021; Volume 8, p. 1.
68. Bordogna, G.; Pagani, M.; Psaila, G. Database model and algebra for complex and heterogeneous spatial entities. In *Progress in Spatial Data Handling*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 79–97.
69. Psaila, G. A database model for heterogeneous spatial collections: Definition and algebra. In Proceedings of the 2011 International Conference on Data and Knowledge Engineering (ICDKE), Milan, Italy, 6 September 2011; pp. 30–35.
70. Bordogna, G.; Ciriello, D.E.; Psaila, G. 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, Leipzig, Germany, 23–26 August 2017; pp. 499–508.
71. Fosci, P.; Marrara, S.; Psaila, G. Soft Querying GeoJSON Documents within the J-CO Framework. In Proceedings of the 16th International Conference on Web Information Systems and Technologies (WEBIST 2020), Online Streaming, 3–5 November 2020; pp. 253–265.
72. Bordogna, G.; Pagani, M.; Pasi, G.; Psaila, G. Evaluating uncertain location-based spatial queries. In Proceedings of the 2008 ACM Symposium on Applied Computing, Fortaleza, Ceara, Brazil, 16–20 March 2008; pp. 1095–1100.
73. Psaila, G.; Fosci, P. Toward an Anayist-Oriented Polystore Framework for Processing JSON Geo-Data. In Proceedings of the International Conferences on WWW/Internet, ICWI 2018 and Applied Computing 2018, Budapest, Hungary, 21–23 October 2018; pp. 213–222.
74. Butler, H.; Daly, M.; Doyle, A.; Gillies, S.; Hagen, S.; Schaub, T. The GeoJSON format. *Internet Engineering Task Force (IETF)*. 2016. Available online: https://datatracker.ietf.org/doc/html/rfc7946 (accessed on 22 April 2021)
75. Regione Lombardia. Open Data Portal. 2021. Available online: https://www.dati.lombardia.it/ (accessed on 22 April 2021).
76. Socrata Platform. 2021. Available online: https://dev.socrata.com/ (accessed on 22 April 2021).
77. Regione Lombardia. Open Data Portal-Weather Stations. 2021. Available online: https://www.dati.lombardia.it/Ambiente/Stazioni-Meteorologiche/nf78-nj6b (accessed on 22 April 2021).
78. Regione Lombardia. Open Data Portal-Weather Sensor Map. 2021. Available online: https://www.dati.lombardia.it/Ambiente/Mappa-Stazioni-Meteorologiche/8ux9-ue3c (accessed on 22 April 2021).
79. Regione Lombardia. Open Data Portal-Air Quality Stations. 2021. Available online: https://www.dati.lombardia.it/Ambiente/Stazioni-qualit-dell-aria/ib47-atvt (accessed on 22 April 2021).
80. Regione Lombardia. Open Data Portal-Air Quality Sensor Map. 2021. Available online: https://www.dati.lombardia.it/Ambiente/Mappa-stazioni-qualit-dell-aria/npva-smv6 (accessed on 22 April 2021).
81. Regione Lombardia. Open Data Portal-Weather Measures. 2021. Available online: https://www.dati.lombardia.it/Ambiente/Dati-sensori-meteo/647i-nhxk (accessed on 22 April 2021).
82. Regione Lombardia. Open Data Portal-Air Quality Measures. 2021. Available online: https://www.dati.lombardia.it/Ambiente/Dati-sensori-aria/nicp-bhqi (accessed on 22 April 2021).
83. GitHub Repository of the JCO-Project. 2021. Available online: https://github.com/zunstraal/J-Co-Project (accessed on 22 April 2021).