International Journal of Computational Intelligence Systems (2023) 16:163
https://doi.org/10.1007/s44196-023-00325-3

RESEARCH ARTICLE

Soft Querying Features in *GeoJSON* Documents: The *GeoSoft* Proposal

Paolo Fosci[1] · Giuseppe Psaila[1]

Received: 11 November 2022 / Accepted: 22 August 2023

Abstract
Since the advent of *JSON* as a popular format for exchanging large amounts of data, a novel category of *NoSQL* database systems, named *JSON*document stores, has emerged for storing *JSON* data sets; in fact, these novel databases are able to natively manage collections of *JSON* documents. To help analysts and data engineers query and integrate *JSON* data sets persistently saved in *JSON* document stores, the *J-CO* Framework has been developed (at the University of Bergamo, Italy): it is built around a novel query language, named *J-CO-QL*$^+$, that provides sophisticated features, including soft-querying capabilities. However, *J-CO-QL*$^+$ (as the other languages for querying *JSON* data sets) is designed to be general purpose; consequently, it can be cumbersome for users to apply it on specific data formats. This is the case of *GeoJSON*, a specific and popular *JSON* data format that is designed to represent geographical information layers. This paper presents the latest evolution of *GeoSoft*, a novel high-level "domain-specific language" that is specifically designed to express complex queries on the *GeoJSON* documents, including soft-queries. *GeoSoft* is inspired to the classical *SQL* language, so as to reduce the learning curve of potential users. *GeoSoft* queries are translated into *J-CO-QL*$^+$ scripts, to be actually executed.

Keywords J-CO framework · GeoSoft · JSON document stores · GeoJSON · Soft querying

1 Introduction

JSON (JavaScript Object Notation [1]) has become the standard format for representing any kind of data published and shared on the Internet. Indeed, in spite of the original aim to rely on XML (eXtensible Mark-up Language [2]) as the universal format for the Internet, *JSON* has become very popular; probably, *JSON* is now more popular than XML.

However, *JSON* (as XML) is a generic format. To represent specific data or documents, it is necessary to define specific document structures: *GeoJSON* [3] is one of the most famous examples of a well structured format that relies on the *JSON* syntax. *GeoJSON* was designed by the GIS (Geographical Information Systems) community to represent "information layers", i.e., a pool of spatial entities that, all together, provide homogeneous spatial information. Examples are roads, green areas, municipalities, and so on. The key aspect of *GeoJSON* information layers is that all the contained entities are "geo-tagged", i.e., they are provided with their spatial description, also said "geometry", which relies on a coordinate system such as WGS-84 [4].

After this brief presentation, the reader could think that collecting and managing *GeoJSON* documents is easy, since the advent of *NoSQL* (Not-only SQL) database technology [5] to store *JSON* data sets should provide a significant help. Specifically, we are thinking about the category of "*JSON* document stores" (i.e., databases whose data model is not the classical relational data model), which are able to store and query "collections of *JSON* documents". Among them, *MongoDB* [6] is nowadays widely popular.

Unfortunately, things are not so easy, for various reasons. (i) A *GeoJSON* information layer is a single, possibly giant, *JSON* document; consequently, querying it is not immediate, since spatial entities must be previously unnested from within this unique giant document. (ii) *JSON* document stores are usually designed to deal with a large number of small *JSON* documents. The result is that, often, large *GeoJSON* documents cannot be even stored within the *JSON* document store. (iii) Spatial querying in *JSON* document stores, when

Giuseppe Psaila and Paolo Fosci have equally contribuited to this work.

✉ Giuseppe Psaila
giuseppe.psaila@unibg.it

Paolo Fosci
paolo.fosci@unibg.it

[1] Department of Management, Information and Production Engineering, University of Bergamo, Viale Marconi 5, 24044 Dalmine, BG, Italy

supported, can be performed only if *JSON*documents are spatially indexed in advance, thus strongly limiting the flexibility of querying documents on the fly.

The previous considerations are, more or less, the same considerations that inspired the idea of developing a novel framework to easily manage collections of *JSON* documents, possibly stored within *JSON* document stores. The *J-CO* Framework (see [7]) relies on a query language (named *J-CO-QL*$^+$) that provides high-level and declarative statements, which are specifically designed to perform complex transformations on collections of *JSON* documents. The language is undergoing the extension with soft-querying capabilities (see [8]), by means of the evaluation of membership degrees of documents to fuzzy sets. So, the *J-CO-QL*$^+$ language, when applied to *GeoJSON* documents, would provide analysts with the capability of soft querying entities in *GeoJSON* information layers, but due to their intrinsically-complex structure, queries would result long and tedious to write.

In [9, 10], we argued that a Domain-Specific Language (DSL), specifically designed to query *GeoJSON* documents as a whole, based on a SQL-like syntax and natively dealing with soft querying, could significantly help analysts to perform complex soft queries on *GeoJSON* documents. In [9], we showed that a first version of this language could be easily translated into *J-CO-QL*$^+$ queries, but it was limited to work on one single *GeoJSON* document. After that preliminary investigation, we decided to name the language as *GeoSoft* [10], we extended it to support the join operation on *GeoJSON* documents and carefully redesigned it; we also implemented a translator of *GeoSoft* queries into *J-CO-QL*$^+$ scripts.

The contribution of this paper is to provide a comprehensive presentation of *GeoSoft*. The first contribution is the semantic model on which *GeoSoft* is built. The second contribution is the *GeoSoft* language itself: specifically, we present its syntax and clauses, through a couple of information layers downloaded from a real Open-Data portal. The third contribution is to show that executing *GeoSoft* queries is possible; in particular, we show how to translate them into *J-CO-QL*$^+$ scripts, that can be actually executed. The final contribution is evaluating the *GeoSoft* proposal under three different points of view such as “flexibility”, “accessibility” and “efficiency”; in particular, as far as this last point is concerned, we present a comparison with the classical *PostgreSQL/PostGIS* solution.

The paper is organized as follows. Section 2 reports relevant related work. Section 3 presents the background of this work, i.e., *NoSQL* databases for storing *JSON* documents (Sect. 3.1) and the *J-CO* Framework (Sect. 3.2). Section 4 introduces the *GeoJSON* format, by showing its adoption for representing two Geographical Information Layers that will be exploited throughout the paper. Section 5 presents the *GeoSoft* language, by discussing syntax and semantics through several examples. Section 6 addresses the problem of executing *GeoSoft* queries, by translating them into *J-CO-QL*$^+$ scripts; thus, Sect. 6.1 briefly introduces *J-CO-QL*$^+$ scripts, while Sect. 6.2 shows the translation strategy and the translation algorithm from *GeoSoft* to *J-CO-QL*$^+$. Section 7 evaluates the *GeoSoft* proposal from several points of view; i particular, Sect. 7.1 evaluates “flexibility”, Sect. 7.2 evaluates “accessibility” and Sect. 7.3 evaluates “Efficiency”. Finally, Sect. 8 draws the conclusions and highlights possible future works.

The Appendix completes the paper, for the interested readers, with detailed presentations of specific topics. In particular, Appendix A presents the *JSON* format, Appendix B presents the *GeoJSON* format and Appendix C briefly reports basic concepts about fuzzy sets; Appendix D reports details about the `JOIN` operator in the *GeoSoft* language; Appendix E extensively discusses the *J-CO-QL*$^+$ script obtained by translating the more complex *GeoSoft* query presented in this paper.

2 Related Work

In Sect. 4, we will show that a *GeoJSON* document is a particular case of *JSON* document, with a specific structure: indeed, apart from the *JSON* syntax, a *GeoJSON* document actually represents a set of spatial features. Consequently, it is reasonable to figure out a query language specifically designed to query *GeoJSON* features; however, to the best of our knowledge, in the literature there are no other proposals, apart from the preliminary version of *GeoSoft* (see [9]). Consequently, we had to refer to proposals designed for working on standard *JSON* documents. A distinctive characteristic of *GeoSoft* is soft querying, as well as another distinctive characteristic is geographical querying. The above considerations motivate the three different sub-topics addressed in this section: (i) languages for expressing soft queries; (ii) languages for querying *JSON* data; (iii) languages for querying geographical data.

This paper assumes that readers are familiar with basic concepts concerning fuzzy sets. For those that are not familiar, Appendix C reports a brief introduction to fuzzy sets.

2.1 Languages for Expressing Soft Queries

Soft querying relies on Fuzzy-Set Theory, an extension of the classical Set Theory that was introduced by Zadeh in [11]: the idea is that items can belong to a set only partially (see Appendix C). As an effect, vague concepts can be expressed as fuzzy sets; thus, “soft selection conditions” can be expressed as fuzzy selection conditions [12], because the membership degree to a fuzzy set quantifies the closeness of the information carried by the x item with respect to the condition. Possibility Theory [11, 13], together with the concept of linguistic variable defined within Fuzzy-Set Theory [14], provides a valuable formal framework for managing imprecise, vague and uncertain information [15].

Historically, the first attempts to develop languages for soft querying in databases were developed in the context of relational databases. Specifically, two main approaches were followed: (i) preserving the classical relational data model; (ii) extending the data model towards "fuzzy-relational data models".

The rationale behind the first approach was to provide soft-querying capabilities on top of already-existing relational databases, which are still widely used within information systems. Thus, many extensions of SQL towards soft querying were proposed, which provide capabilities of soft querying table rows through Fuzzy-Set Theory. Here, we mention only a few of them, in particular *SQLf* (see [16, 17]) and the attempt to implement it (in [18]); the second proposal we mention is *FQUERY for Access* (see [19, 20]), which was designed to work within Microsoft Access; finally, we mention the proposal named *SoftSQL* (see [21–23]), which also covers the definition of user-defined "linguistic predicates" through a dedicated statement, to be used in soft selection conditions within the SELECT statement.

The second approach was to define a "fuzzy-relational data model" (refer to [24, 25]), i.e., an extension of the classical relational data model that was able to natively represent uncertain values and data within the database. Here, we mention *FSQL*, presented in [26, 27], which is the most remarkable proposal, in our opinion.

To the best of our knowledge, the most recent paper on this topic is [28]. In this work, the authors briefly present a library for PostgreSQL, named *PostgreSQLf*, which provides capabilities for dealing with fuzzy values within PostgreSQL. Although the description is synthetic, the library is still available for download. In our opinion, its point of strength is also its weakness, i.e., it does not extend the query language, but provide functions to call within the classical SQL queries.

2.2 Languages for Querying *JSON* Data

The advent of many systems specifically designed to store *JSON* documents is a matter of fact; they are database systems that does not rely on the classical relational data model, but manage collections of *JSON* documents, independently of their structure. Among them, the most popular is *MongoDB*,[1] which is used for storing and querying large collections of small *JSON* documents. Another important representative is *CouchDB*[2] [29].

The advent of these kind of database systems has caused the definition of many query languages for *JSON* data sets. Popular representatives are *Jaql* [30], *SQL++* [31] (which is, the query language of *CouchDB*), *JSONiq* [32] and the query language provided by *MongoDB* [5], which is generally called *MQL*. Certainly, we can say that the most used of them is *MQL*, due to the popularity of *MongoDB*.

In recent works, we are observing the attempt to resume the ideas explored in the area of soft querying on top of relational databases (topic (i) in 2.1), by adapting them to the new context of *JSON* document stores.

For example, the work [33] proposed *fMQL*, an extension of *MQL* (the *MongoDB* Query Language). The authors worked under the hypothesis that *JSON* documents are previously labeled with "fuzzy labels".

Another recent work [34] proposed an extension of the *MongoDB* data model towards a fuzzy *JSON* document store, supporting fuzzy values in single fields.

In our extension of the *J-CO-QL*$^+$ language towards soft querying [35, 36], we are substantially following the first approach, i.e., the query language is built on top of the standard *JSON* data model (the reader will see more in the remainder of the paper).

2.3 Languages for Querying Geographical Data

GeoJSON is a standard format proposed by the community working on "Geographical Information Systems" (GIS); consequently, it is necessary to give a look at this research area as well. In this domain, many research works have been done, in particular addressing issues concerned with data storage, indexing and query optimization [37]; indeed, a GIS is supposed to store huge amounts of complex data, so as to graphically represent them. As a consequence, many extensions of the relational data model and of the SQL language have been proposed [38, 39]: the goal is to make Data Base Management Systems (DBMSs) able to store and retrieve spatial information, i.e., geo-referenced geometries that represent the spatial shape of entities described by data. Clearly, extensions of SQL are appreciated by users that are familiar with writing SQL queries, while they are not good for analysts who do not have such a skill. Consequently, other proposals were made: tabular approaches [40] (that extend QBE, Query By Example), graphical languages [41, 42], visual languages [43], and "hypermaps" [44] (integrated into hypermedia techniques [45]) provides users with powerful graphical approaches to specify queries by directly operating on maps. An interesting and recent survey the reader can

[1] *MongoDB*. Available online: https://www.mongodb.com/, accessed on 01/07/2023.

[2] CouchDB. Available online: https://couchdb.apache.org/, accessed on 01/07/2023.

refer to is [46], in which it is possible to find a comprehensive literature review about the approaches concerned with spatial-data management in non-relational databases.

However, *JSON* and *GeoJSON* in particular are changing the panorama. In fact, if the geographical information is stored as *GeoJSON* documents, most systems provide low (or none) capability to query them in a geographical way. Indeed, very few query languages are specifically targeted to *GeoJSON*. We can cite *GeoPQLJ* [47]: it is a pictorial query language, which provides drawing facilities (which rely on an underlying algebra) to formulate complex queries; this way, it should be possible to remove semantic ambiguity. However, although it works on *GeoJSON* documents, it does not explicitly rely on peculiarities of *GeoJSON* documents, as *GeoSoft* does.

An early work that somehow anticipates *GeoSoft* is [48]. The authors proposed an extension of SQL for soft querying geographical data; the idea inspired the *SoftSQL* proposal [22] and, many years later, the *J-CO-QL*$^+$ language [7, 8, 49] and, finally, the *GeoSoft* proposal presented in this paper.

3 Background

The goal of this section is to present the background on which *GeoSoft* relies. First of all, we present basic notions about *NoSQL* databases for natively storing *JSON* data sets; then, we will introduce the *J-CO* Framework.

3.1 *JSON* and *NoSQL* Databases

JSON is the acronym for JavaScript Object Notation [1]. It was born in the context of the JavaScript object-oriented programming language, so as to define constant objects. However, its simplicity and immediacy has made it autonomous from the JavaScript language, as a powerful format to represent data with complex structures in a simple and easy-to-process way. Currently, *JSON* is used in a multitude of applications and has become very popular.

In this paper, we assume that the reader is familiar with the *JSON* format and the related terminology. The reader that is not familiar, can find a brief introduction to the *JSON* format in Appendix A.

The term *NoSQL* stands for "Not only SQL" [50]. It denotes a variety of approaches to database systems that do not rely on the classical relational model (thus, they do not provide the SQL query language).

Among all *NoSQL* approaches, the category of "document database" is relevant for our work. A database system that falls into this category is able to store and retrieve "structured documents" (i.e., not plain text) whose structure is not fixed but can freely vary. Typically, such systems adopt *JSON* as the format to represent documents, so they are named "*JSON* document stores" (or simply "*JSON* stores").

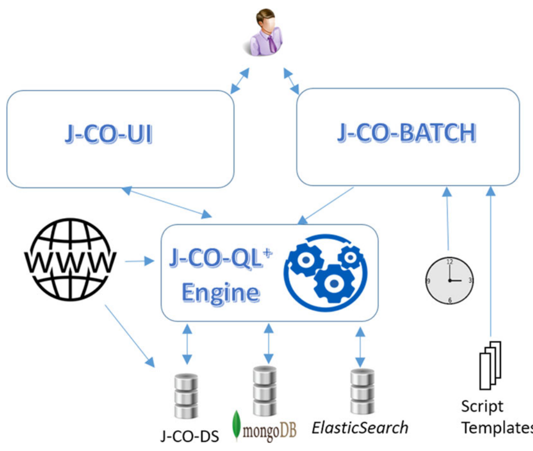


Fig. 1 Organization of the *J-CO* Framework

The most popular *JSON* store in the world is *MongoDB* [51],[3] but other *JSON* stores have become popular too, such as *CouchDB* [29] (which has been adopted as the database system for the *HyperLedger Fabric* permissioned blockchain platform [52]). Similarly, a tool that is not exactly a database system but an information-retrieval system is *Elasticsearch* [53]; however, it receives and provides data as *JSON* documents, so it can be somehow considered as a *JSON* store.

Since *MongoDB* is very popular, its data model is popular too and is considered the reference data model for *JSON* stores.

- A "collection" is an unordered multi-set of heterogeneous documents, i.e., it can contain multiple copies of the same document. No limitations about the structure of contained documents is posed: heterogeneous documents can be contained within the same collection.
- A "database" is a "set of collections". Each collection has a unique name within the database.

3.2 The *J-CO* Framework

Currently there is not an official standard for *JSON* stores; as a results, data models are not perfectly compatible and, primarily, query languages are very different each other (when provided). Consequently, analysts and data engineers have to face a very difficult scenario, when they have to deal with multiple *JSON* stores based on different technologies and query languages, to integrate data sets.

These considerations inspired us to develop the *J-CO* Framework [49, 54, 55]. It is a pool of software tools designed

[3] See https://db-engines.com/en/ranking for a ranking updated in 2022.

to provide analysts with a powerful support for gathering, integrating, transforming and querying collections of *JSON* data sets. The core of the framework is its query language, named *J-CO-QL*$^{+}$, which has been extensively presented in [8, 56–58]. The organization of the framework is depicted in Fig. 1; hereafter, we illustrate it.

- *J-CO-QL*$^{+}$ *Engine*. This component executes *J-CO-QL*$^{+}$ queries (or scripts). It retrieves data from *JSON* stores (for example, managed by *MongoDB* or *Elasticsearch*) and saves results into them; it also retrieves data sets from Web sources.
- *J-CO-UI*. This is the user interface, by means of which *J-CO-QL*$^{+}$ scripts are interactively written and submitted to the *J-CO-QL*$^{+}$ *Engine*.
- *J-CO-DS*. This component is a *JSON* document store specifically designed to store large single documents [59], so as to overcome limitations of other *JSON* stores (such as *MongoDB*, which does not accept documents that are larger than 16 MB in its *BSON* internal binary format). *J-CO-DS* does not provide computational capabilities such as a query language, because it is a component of the *J-CO* Framework, whose query language is *J-CO-QL*$^{+}$. *J-CO-DS* has been recently upgraded [60] with new functionalities, to manage three different types of collections. (i) "Static collections" are persistently stored in the database. (ii) "Virtual Collections" are not materialized within the database; they are defined by means of a pool of URLs that provide the data sets; when the collection is accessed, its content is dynamically built by calling the associated URLs. (iii) "Dynamic collections" are persistently stored in the database, but their content is dynamically acquired; a pool of URLs is associated with the collection, so as the engine periodically acquires new versions of the data (so, data sets are accessible for a user, even when the framework is temporarily off-line). As a major result, *J-CO-DS* provides a database view of Web sources, which is a unique feature in comparison with other *JSON* stores.
- *J-CO-BATCH* has been recently added to the *J-CO* Framework [60]. Its goal is to provide "batch execution" of *J-CO-QL*$^{+}$ scripts. By means of this tool, it is possible to create parametric templates of *J-CO-QL*$^{+}$ scripts, whose execution cam be scheduled, either one-shot or repeatedly in time.

The *J-CO* Framework has originated from our participation to the Urban Nexus project [61], whose goal was to integrate Big Data coming from various sources, such as Open-Data portals and social media, so as to study citizens mobility. In that project, we developed several tools for integrating and analyzing data [62–65]. During that project, it was clear that the support provided by *JSON* stores for managing *JSON* data sets was too raw for analysts; indeed, we felt that higher-level tools were necessary. These are the origins of the *J-CO* Framework.

4 Geographical Information Layers as *GeoJSON* Documents

GeoJSON,[4] is an interchange format for spatial data. It is designed to represent geographical spatial entities (called "features") and their non-spatial properties. *GeoJSON* is based on *JSON* as host syntactic format: this characteristic allows for processing *GeoJSON* documents as any other *JSON* document. Specifically, it is suitable for encoding "geographical information layers", i.e., aggregations of spatial entities that are somehow homogeneous or strongly correlated (e.g., roads, buildings, rivers, and so on). *GeoJSON* is independent of any geographical Coordinate Reference System (CRS); however, most of *GeoJSON* documents implicitly adopt the World Geodetic System 1984 (WGS-84) and decimal-degree units.

Another advantage is that *GeoJSON* is a human-readable format, since it is a plain-text format; however, this characteristic makes it verbose. As an effect, *GeoJSON* documents can become much larger than other formats for representing spatial data, such as *Shapefile*[5] or *GeoPackage*.[6]

For readers that are not familiar with the *GeoJSON* format, Appendix B provides a brief introduction to it. Essentially, a *GeoJSON* document, in its complete form, can be seen as a "collection of features", where a "feature" has "registry properties" and a "geometry" (that is the spatial footprint of the feature). Figure 2 reports an excerpt of a *GeoJSON* document: the `features` field is an array that contains *JSON* documents, one for each feature; each nested document contains the `properties` field and the `geometry` field.

In order to provide a case study for presenting *GeoSoft*, we now present two information layers described by two *GeoJSON* documents, that will be exploited in the remainder of the paper.

Information Layer 1 The *GeoJSON* document reported in Fig. 2 was downloaded from *Regione Lombardia Open Data portal*.[7] It represents the information layer of the 1506 municipalities (or towns) in Lombardy, the Italian region that includes Milan and Bergamo. Consequently, each feature

[4] GeoJson: https://geojson.org/, accessed on 01/07/2023.

[5] Shapefile: https://www.statsilk.com/maps/convert-esri-shapefile-map-geojson-format, accessed on 01/07/2023.

[6] GeoPackage: https://www.geopackage.org/guidance/modeling.html, accessed on 01/07/2023.

[7] End point of the `towns` data set at *Regione Lombardia Open Data portal*: https://www.dati.lombardia.it/api/geospatial/wtqz-z7j6?method=export&format=GeoJSON, accessed on 01/07/2023.

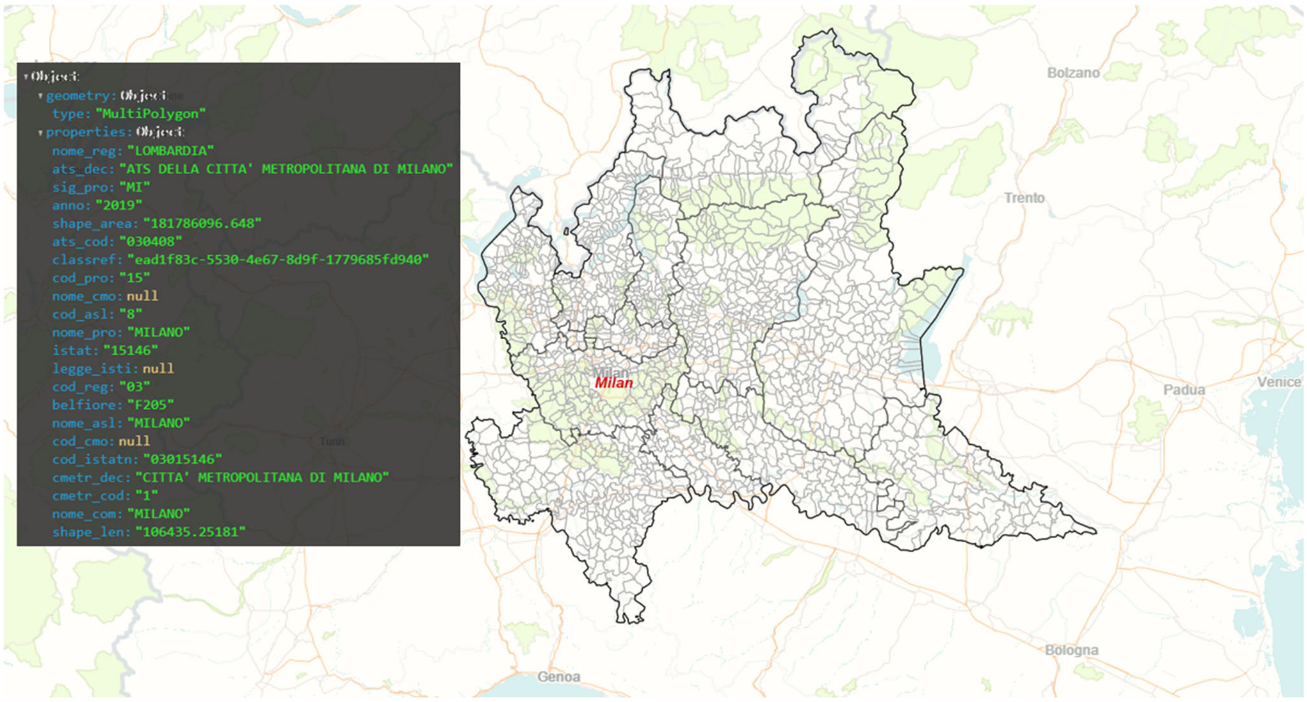
Fig. 2 Excerpt of *GeoJSON* document representing municipalities (towns)

```
{
  "type"      : "FeatureCollection",
  "features"  : [
   {
     "type"          : "Feature",
     "properties"  : {
       "nome_reg"     : "LOMBARDIA",
       "nome_pro"     : "CREMONA",
       "nome_com"     : "PESCAROLO ED UNITI",
       "shape_len"    : "25170.490096",
       "shape_area"   : "16564616.4342",
       … uninteresting other fields …
     },
     "geometry"    : {
       "type"         : "MultiPolygon",
       "coordinates" : [
         [ [ [10.170797474219485, 45.2155294513193],
             [10.171142604232308, 45.215537092353145],
             …,
             [10.170797474219485, 45.2155294513193]  ] ] ]
     }
   },
   …,
   {
     "type"          : "Feature",
     "properties"  : {
       "nome_reg"     : "LOMBARDIA",
       "nome_pro"     : "MILANO",
       "nome_com"     : "INZAGO",
       "shape_area"   : "12143682.7638",
       "shape_len"    : "21159.11567",
       … uninteresting other fields …
     },
     "geometry"    :{
       "type"         : "MultiPolygon",
       "coordinates" : [
         [ [ [9.453252599602981, 45.55888895381179],
             [9.453294885700517, 45.55575733465349],
             …,
             [9.453252599602981, 45.55888895381179] ]  ] ]
     }
   }
  ]
}
```

represents a municipality. Notice the root-level `features` field, which is an array of documents describing features; each feature contains properties (the `properties` field) and the geometry (the `geometry` field). To facilitate the reader in recognizing the structure of the *GeoJSON* document, we enclosed each feature into blue-border boxes, as well as, inside each feature, we enclosed the `properties` and `geometry` fields into red-dashed-border boxes. An analogue highlighting strategy is used in the rest of the paper for the subsequent examples.

The reader can notice that each `geometry` field represents a `MultiPolygon` (see Appendix B), since, in the general case, the territory of a municipality may be non-continuous; implicitly, the WGS-84 Coordinate Reference System is adopted in *GeoJSON*.

Each `properties` field contains a wide variety of fields, including the name of the municipality (the `nome_com` field), the name of the region (the `nome_reg` field) and the name of the province (the `nome_pro` field) to which the municipality belongs; other fields report the length of the borders (the `shape_len` field, expressed in meters) and the area of the municipality (the `shape_area` field, expressed in square meters). For the sake of simplicity, we do not describe the other fields.



```
▾Object
  ▾geometry: Object
    type: "MultiPolygon"
  ▾properties: Object
    nome_reg: "LOMBARDIA"
    ats_dec: "ATS DELLA CITTA' METROPOLITANA DI MILANO"
    sig_pro: "MI"
    anno: "2019"
    shape_area: "181786096.648"
    ats_cod: "030408"
    classref: "ead1f83c-5530-4e67-8d9f-1779685fd940"
    cod_pro: "15"
    nome_cmo: null
    cod_asl: "8"
    nome_pro: "MILANO"
    istat: "15146"
    legge_isti: null
    cod_reg: "03"
    belfiore: "F205"
    nome_asl: "MILANO"
    cod_cmo: null
    cod_istatn: "03015146"
    cmetr_dec: "CITTA' METROPOLITANA DI MILANO"
    cmetr_cod: "1"
    nome_com: "MILANO"
    shape_len: "106435.25181"
```

Fig. 3 Towns in Lombardy Region. The thick lines denote the region borders. The medium-thick lines denote borders of provinces. The thin lines denote borders of cities. The black box, on the left, reports the properties contained in the *GeoJSON* feature representing the city of Milan (whose name is in red)

Finally, notice how the names of the properties are quite unclear: parts in Italian and parts in English; often, contracted forms are used. Notice also that all numerical field values are expressed as strings: this fact denotes poor data design and makes it hard to exploit them.

Figure 3 shows the content of the *GeoJSON* document drawn on a map. The reader can notice the multitude of polygons that actually constitute municipalities.

Information Layer 2 The *GeoJSON* document reported in Fig. 4 describes highways in Lombardy (Italy); it was downloaded from *Regione Lombardia Open Data portal*[8] as a *Shapefile*, and then converted into a *GeoJSON* document by means of the *QGIS* tool.[9] The document describes 94 highway sections, each for one single direction, including also highway junctions. Differently from the *GeoJSON* document that describes municipalities (in Fig. 2), at root-level we find the `name` and the `crs` fields as well (see Appendix B): the goal is to give a name to the information layer and to explicitly set the Coordinate Reference System to WGS-84.

Each feature describes a highway; more precisely, it describes the section of the highway that traverses Lombardy. Notice that the `geometry` field represents a `MultiLineString` geometry (see Appendix B), since, in the general case, the path of a highway may be non-continuous (the same highway could exit and enter the same region multiple times). The `properties` field provides registry data about highways, such as identifier (the `COD_PE` field), name (the `NOME_PERCO` field) and total length in Lombardy (the `SHAPE_LEN` field, expressed in meters) of the highway section. Again, for the sake of simplicity, we do not describe the other fields.

From Fig. 4, the reader can notice that field names are badly designed again: parts of them are in Italian and parts of them are in English, as well as contracted forms are used; we also remark that, this time, the `SHAPE_LEN` field has a numerical value.

Figure 5 depicts the content of the presented *GeoJSON* document by drawing, in green color, all the highway segments on a map. The reader can notice the purple-highlighted highway in center-bottom of the map, whose properties are reported in the black-box in the left-upper corner of the figure: this is the section of the highway named `"A1"` that traverses Lombardy, whose final ending is Naples; notice that the reported total length, in the `SHAPE_LEN` field, is about

[8] End point of the `highways` data set at *Regione Lombardia Open Data portal* (for the sake of brevity we report a short tiny URL): https://www.geoportale.regione.lombardia.it/metadati?p_p_id=detailSheetMetadata_WAR_gptmetadataportlet&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&_detailSheetMetadata_WAR_gptmetadataportlet_identifier=r_lombar%3A3417cfea-5192-467e-a388-947bdd7a85f2&_jsfBridgeRedirect=true, accessed on 01/07/2023.

[9] *QGis* web site: https://www.qgis.org/en/site/, accessed on 01/07/2023.

Fig. 4 Excerpt of *GeoJSON* document representing highways

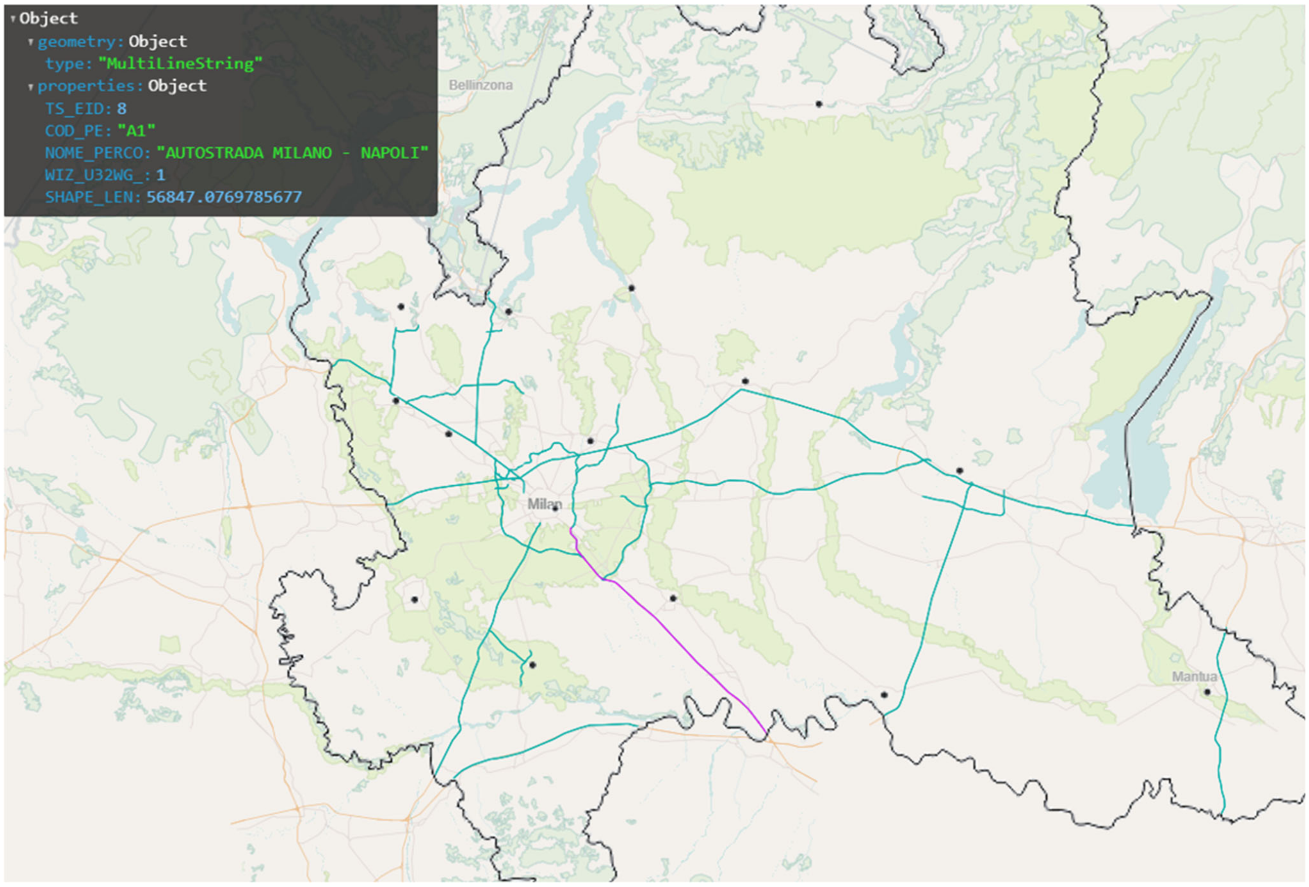
```
{
  "type"     : "FeatureCollection",
  "name"     : "Highways",
  "crs"      : {
    "type"         : "name",
    "properties"  : {
      "name"         : "urn:ogc:def:crs:OGC:1.3:CRS84"
    }
  },
  "features" : [
   {
     "type"          : "Feature",
     "properties"  : {
       "COD_PE"       : "A1",
       "NOME_PERCO"  : "AUTOSTRADA MILANO - NAPOLI",
       "SHAPE_LEN"   : 56847.076978567697,
       "WIZ_U32WG_"  : 1,
       "TS_EID"       : 8.0
     },
     "geometry"     : {
       "type"          : "MultiLineString",
       "coordinates" : [
         [ [ 9.247836038774052, 45.428966206259929 ],
           [ 9.247745878260337, 45.428902492978118 ],
           …,
           [ 9.732565595662319, 45.067241140579682 ] ] ]
     }
   },
   …,
   {
     "type"          : "Feature",
     "properties"  : {
       "COD_PE"       : "A4racc",
       "NOME_PERCO"  : "RACCORDO A4",
       "SHAPE_LEN"   : 3870.81458546453,
       "WIZ_U32WG_"  : 83,
       "TS_EID"       : 21201.0
     },
     "geometry"     : {
       "type"          : "MultiLineString",
       "coordinates" : [
         [ [ 10.31893954983593,  45.460087581660595 ],
           [ 10.318575629192758, 45.461908299136475 ],
            …,
           [ 10.319255506857733, 45.494681495333104 ] ] ]
     }
   }
  ]
}
```

56 km, while the actual total length of the `"A1"` highway is about 760 km.[10]

5 *GeoSoft*

The language we propose is named *GeoSoft*. To make it easy for analysts to use it, we exploited the well known syntax of the SQL `SELECT` statement, but it works on features within *GeoJSON* documents. In Sect. 5.1, we first present the semantic model which the language relies on. Then, in the

[10] `"A1"` highway: https://en.wikipedia.org/wiki/Autostrada_A1_(Italy) accessed on 01/07/2023.



```
▾Object
  ▾geometry: Object
     type: "MultiLineString"
  ▾properties: Object
     TS_EID: 8
     COD_PE: "A1"
     NOME_PERCO: "AUTOSTRADA MILANO - NAPOLI"
     WIZ_U32WG_: 1
     SHAPE_LEN: 56847.0769785677
```

Fig. 5 Highways in Lombardy Region. The black-box in the left-upper corner reports the properties contained in the *GeoJSON* feature representing the highway highlighted in purple in the center-bottom of the map

remainder of this section, we present the *GeoSoft* language by exploiting the information layers introduced in Sect. 4.

5.1 Semantic Model

We present the first contribution of the paper, i.e., the semantic model which the *GeoSoft* language relies on is presented. Specifically, we distinguish between "external semantic model" (i.e., how a query behaves on *GeoJSON* documents, independently of their storage or provenance) and "internal semantic model" (i.e., how the query behaves internally); then, we have to consider the application scope of the language, so we complete the semantic model by defining its relation with databases managed by *JSON* stores.

5.1.1 External Semantic Model

We start by considering the external semantic model, i.e., how a *GeoSoft* query gsq is expected to behave, independently of how it is internally structured.

Definition 1 ***GeoJSON* Feature.** A (crisp) *GeoJSON* feature is a tuple

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

where `properties` and `geometry` (see Sect. 4 and Appendix B) are *JSON* documents describing, respectively, the properties and the geometry of the spatial entity described by the feature.

A feature is the elementary item a *GeoSoft* query is expected to work on.

Definition 2 ***GeoJSON* Document.** A (crisp) *GeoJSON* document gd is modeled as a possibly-empty set of features, i.e., $gd = \{f_1, \ldots, f_n\}$. The domain of *GeoJSON* documents is denoted as GJ.

Figure 6a shows an example of (crisp) *GeoJSON* document, as it is intended in our semantic model. Notice that features are represented as tuples; the overall document is represented as a set of tuples.

Based on these definitions, it is possible to define the "external semantics" of a *GeoSoft* query, i.e., how the query is expected to behave outside the query itself.

properties	**geometry**
...	...
`{ "highwayId" : ..., "highwayTotalLength" : ... }`	`{ "type" : "MultiLineString", "coordinates" : [ [ [..., ...], ..., [..., ...] ] ] }`
...	...

(a) Crisp *GeoJSON* document.

properties	**geometry**	**fuzzysets**
...	...	...
`{ "highwayId" : ..., "highwayTotalLength" : ... }`	`{ "type" : "MultiLineString", "coordinates" : [ [ [..., ...], ..., [..., ...] ] ] }`	$\emptyset$
...	...	...

(b) Soft *GeoJSON* document with features having no membership degree.

properties	**geometry**	**fuzzysets**
...	...	...
`{ "highwayId" : ..., "highwayTotalLength" : ... }`	`{ "type" : "MultiLineString", "coordinates" : [ [ [..., ...], ..., [..., ...] ] ] }`	`MediumLengthHighway` $\longrightarrow$ ...
...	...	...

(c) Soft *GeoJSON* document with features having one membership degree.

properties	**geometry**	**fuzzysets**
...	...	...
`{ "highwayId" : ..., "highwayTotalLength" : ... }`	`{ "type" : "MultiLineString", "coordinates" : [ [ [..., ...], ..., [..., ...] ] ] }`	`MediumLengthHighway` $\longrightarrow$ ... `NotMediumLengthHighway` $\longrightarrow$ ...
...	...	...

(d) Soft *GeoJSON* document with features having two membership degrees.

Fig. 6 Internal model of crisp and soft *GeoJSON* documents, with generic features, as they are managed by *GeoSoft* queries

Definition 3 External Query. Consider the domain GJ of *GeoJSON* documents. An *external GeoSoft* query gsq is a function

$$gsq : GJ^n \rightarrow GJ$$

where $GJ^n = GJ \times \cdots \times GJ$ is the n-ary Cartesian product (with $n > 0$) on the GJ domain.

In the simplest form, i.e., with $n = 1$, the gsq query takes one single *GeoJSON* document as input; in the complex form, i.e., $n > 1$, the gsq query works on multiple input *GeoJSON* documents. The gsq query always generates one single *GeoJSON* document as output.

5.1.2 Internal Semantic Model

We now define the semantic model that is internally followed by *GeoSoft* queries. Clearly, this semantic model takes soft querying into account.

Definition 4 Soft Feature. A *soft feature* $\overline{f}$ is a feature for which membership degrees to some fuzzy sets are considered. A soft feature is defined as a tuple

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

where `properties` and `geometry` are defined as in Definition 1. Each feature can belong to several fuzzy sets with a specific membership degree (the membership degree to each fuzzy set denotes the degree with which the feature belongs to the fuzzy set). In the $\overline{f}$ tuple, the `fuzzysets` member is a key/value map, that associates a fuzzy set name *fsn* (the key) to the membership degree (the value) of the $\overline{f}$ feature to the fuzzy set *fsn*.

The reader can notice that the universe of fuzzy sets is the domain of spatial features; in other words, we consider the membership of a spatial feature to a fuzzy set as a whole.

Introducing the concept of soft feature allows us to evolve the concept of *GeoJSON* document into the concept of "soft *GeoJSON* document".

Definition 5 Soft *GeoJSON* Document. A soft *GeoJSON* document $\overline{gd}$ is modeled as a possibly-empty set of soft features, i.e., $\overline{gd} = \{\overline{f}_1, \dots, \overline{f}_n\}$. The domain of soft *GeoJSON* documents is denoted as $\overline{GJ}$.

As an example, Fig. 6b–d show different instances of soft *GeoJSON* document, as intended in the semantic model. Notice that the generic feature in Fig. 6b has no membership degree (the `fuzzysets` map is empty); the generic feature in Fig. 6c has one membership degree (one entry in the `fuzzysets` map); finally, the generic feature in Fig. 6d has two membership degrees (two entries in the `fuzzysets` map)-

Consequently, we argue that a *GeoSoft* query should work on soft *GeoJSON* documents internally, while externally it must work on crisp *GeoJSON* documents (as defined by Definition 3). Hereafter, the formal semantic framework is extended.

Definition 6 Internal Query. Consider the domain $\overline{GJ}$ of soft *GeoJSON* documents. An *internal GeoSoft* query $\overline{gsq}$ is a function

$$\overline{gsq} : \overline{GJ}^n \to \overline{GJ}$$

where $\overline{GJ}^n = \overline{GJ} \times \cdots \times \overline{GJ}$ is the n-ary Cartesian product (with $n > 0$) of the $\overline{GJ}$ domain.

Thus, it is necessary to create a bridge between the external and the internal semantics, in that crisp input *GeoJSON* documents gd must be transformed into input soft *GeoJSON* documents $\overline{gd}$ for the internal query, while the output soft *GeoJSON* document from the internal query must be translated into a crisp *GeoJSON* document. This is done by the following definition.

Definition 7 Consider the domain GJ of crisp *GeoJSON* documents $gd \in GJ$ and the domain $\overline{GJ}$ of soft *GeoJSON* documents $\overline{gd} \in \overline{GJ}$. An *external GeoSoft* query $gsq : GJ^n \to GJ$ (with $n > 0$) is defined as

$$gsq\,(gd_1, \dots, gd_n) = toCrisp(\overline{gsq}(toSoft\,(gd_1)\,, \dots, toSoft\,(gd_n)).$$

We make use of $toCrips : \overline{GJ} \to GJ$, which is a function that removes the `fuzzysets` member from soft features, thus obtaining crisp features.

We also make use of $toSoft : GJ \to \overline{GJ}$, which is a function that extends features with an empty `fuzzysets` member.

The reader can notice that input *GeoJSON* documents are automatically "fuzzified", i.e., translated to the domain $\overline{GJ}$ of soft *GeoJSON* documents; in contrast, output soft *GeoJSON* documents are automatically "de-fuzzified", i.e., translated to the domain GJ of crisp *GeoJSON* documents.

Figure 6b shows an example of soft *GeoJSON* document immediately after "fuzzification": the generic feature has the empty `fuzzysets` map.

5.1.3 Queries and Database

To complete the semantic model, it is necessary to consider the database. Indeed, we consider a scope in which input *GeoJSON* documents are stored within a *JSON* document store and output *GeoJSON* documents are saved into a *JSON* document store. Thus, first of all we define the notion of database.

Definition 8 Consider a *JSON* document store. A "collection" c is a multi-set $c = \{d_1, \dots, d_n\}$, where multiple instances of the same document are possible. A collection c has a property $c.name$.

A "database" db is a set of collections $db = \{c_1, \dots, c_h\}$, such that the name $c_i.name$ (for a collection $c_i \in db$) uniquely identifies c_i (in other words, there cannot be two collections $c_i, c_j \in db$ such that $c_i.name = c_j.name$).

The external *GeoSoft* query gsq was defined (see Definition 3) to operate on the domain GJ of (crisp) *GeoJSON* documents. However, the *GeoSoft* language must be defined in such a way it takes *GeoJSON* documents from one or more databases and saves the output *GeoJSON* document into a database. The following definition defines the final concept of "database query" dbq.

Definition 9 Database Query. Consider a "collection descriptor" $cd = \langle cname, dbname, docname \rangle$, where $cname$ is the name of a collection and $dbname$ is the name of a database; if $docname$ has a non-null value, it denotes the name of a *GeoJSON* document. A "database query" is defined as

$$dbq\,(to, from_1, \ldots, from_n) = \\ = save\,(to, gsq(get\,(from_1)\,, \ldots, (from_n)))$$

where to and $from_1, \ldots, from_n$ are collection descriptors. dbq is a procedure that receives $n+1$ (with $n > 0$) collection descriptors: to denotes the collection which the output *GeoJSON* document must be saved into, while $from_1$ to $from_n$ denote the collections from which the input *GeoJSON* documents must be acquired.

We make use of $get(from_i)$ (with $0 < i \le n$), which is a function that actually accesses the $from_i.dbname$ database and gets all *GeoJSON* documents from the $from_i.cname$ collection. Specifically, non-*GeoJSON* documents (possibly contained in the collection) are not considered, while all *GeoJSON* documents actually present in $from_i.cname$ are fused into one single *GeoJSON* document. Furthermore, if the $from_i.docname$ has a non-null value, only *GeoJSON* documents whose `name` field (see Appendix B) equals $from_i.docname$ are considered.

We also make use of $save(to, gd)$, which is a procedure that actually saves the gd *GeoJSON* document into the $to.cname$ collection in the $to.dbname$ database (by dropping the previous instance of the collection, if present). Furthermore, if $to.docname$ has a non-null value, the `name` field of the saved *GeoJSON* document (see Appendix B) is set to $to.docname$.

Through the above definition of database query dbq, we fully define the application scope of the *GeoSoft* language. Notice that the fact that source *GeoJSON* documents are contained in collections of *JSON* documents is solved by applying Definition 2. The set-oriented view of a *GeoJSON* document provides the way to solve the issue of multiple *GeoJSON* documents stored within the same input collection: features within them are united into one single *GeoJSON* document.

5.2 Simple Queries

We can now introduce the second main contribution of the paper, i.e., the *GeoSoft* language. Here, we present simple queries, i.e., queries that take one single *GeoJSON* document as input and produce a new *GeoJSON* document, without nested queries or joins.

5.2.1 Selection and Projection

Suppose that a user wants to query the *GeoJSON* document that describes highways (see Information Layer 2), which is depicted in Fig. 4. The goal of the user is: generating a new *GeoJSON* document that contains only highways whose length is greater than 3 *km*. Listing 1 reports the (simple) *GeoSoft* query. We present it hereafter.

Listing 1 Simple Selection-Projection query in *GeoSoft*.

```
GET CONTEXT "jcoContextDb.jco";

SELECT .COD_PE      AS .highwayId,
       .SHAPE_LEN   AS .highwayTotalLength
   FROM highways@geosoftDb
       WITH NAME "highways"
   WHERE .SHAPE_LEN > 3000
   SAVE AS ProperHighways@geosoftDb
       SETTING NAME "Proper Highways";
```

Listing 2 Content of the `jcoContextDb.jco` file.

```
USE DB geosoftDb
  ON SERVER jcods 'http://127.0.0.1:17017';
```

First of all, notice the `GET CONTEXT` directive, by means of which it is possible to load the specified file, whose content is the "execution context". This is a preamble written in *J-CO-QL*$^+$; this is due to the fact that *GeoSoft* query is translated into a *J-CO-QL*$^+$ script and then executed by the *J-CO-QL*$^+$ *Engine*; the notion of execution context allows us to exploit fragments of scripts written in *J-CO-QL*$^+$, without replicating constructs in *GeoSoft*.

Specifically, the `GET CONTEXT` directive in Listing 1 loads the `jcoContextDb.jco` file, whose content is reported in Listing 2: it declares the connection to the database (notice that the database is managed by *J-CO-DS*, the *JSON* document store provided by the *J-CO* Framework, remember from Sect. 3.2).

The `SELECT` statement actually performs the query, as explained hereafter. Remember that the goal of the query is "generating a novel *GeoJSON* document that contains only features describing highways whose length is greater than 3 *km*".

- The `FROM` clause specifies the collection in the database where the document is stored. Specifically, in the query we refer to the `highways` collection, which is contained in the `geosoftDb` database. Notice the `WITH NAME` option: when specified, it allows us to focus the acquisition only on *GeoJSON* documents whose root-level `name` field (see Appendix B) has the specified value (discarding the other documents that are possibly present in the same collection).
- The `WHERE` clause selects features in the *GeoJSON* document, by means of a classical Boolean condition. Specifically, features having the value for the `SHAPE_LEN` property greater than 3 *km* (the length is in meters) are selected.
- The list that follows the `SELECT` keyword (in the following, we will refer to it as "the `SELECT` clause")

specifies the list of properties of interest; we can say that "features are projected on a subset of properties". Specifically, the list of properties is projected on the `COD_PE` property (which is renamed as `highwayId` by the subsequent alias) and on the `SHAPE_LEN` property (which is renamed as `highwayTotalLength`).
- Finally, the `SAVE AS` clause saves the resulting *GeoJSON* document into a database collection. Specifically, the document is saved into the `ProperHighways` collection in the `geosoftDb` database. Notice the optional `SETTING NAME` specification: if present, it adds the root-level `name` field to the output *GeoJSON* document (see Appendix B), with the specified value.

The reader can notice that properties are referred through the dot notation, e.g., "`.SAMPLE_LEN`". Remember that properties are within the nested `properties` sub-document; properties can be, in turn, nested sub-documents. Consequently, the first dot is rooted within the `properties` sub-document (member of the feature, see Definition 4).

As a final remark, the choice for relying on the basic SQL syntax is towards users: indeed, they do not have to learn a completely new language.

5.2.2 Soft Querying

The query reported in Listing 1 does not exploit soft-querying, while this is done by the query in Listing 3. As the reader can see, the basic structure of the query is the same; clearly, novel clauses are used, to deal with fuzzy sets.

Suppose that a user wants to query the *GeoJSON* document describing highways (Information Layer 2), with the following goal: generating a novel *GeoJSON* document that contains features describing highways that are more or less *medium-length highways*; both the degree with which they satisfy the requirement and the degree with which they do not satisfy the requirement are of interest. Hereafter, we show how the *GeoSoft* query reported in Listing 3 expresses this soft query.

- The `FROM` clause specifies again the `highways` collection in the `geosoftDb` database. Remember from Definition 7 that features in the source *GeoJSON* document are converted into soft features (Definitions 4 and 5) by adding the `fuzzysets` key-value map.
- The `WHERE` clause selects those features whose value of the `SHAPE_LEN` property is greater than 3000 meters.
- The subsequent `USING` clause works on the set of features selected by the `WHERE` clause and evaluates a "soft condition", so as to evaluate the membership degree of features to a specific fuzzy set. The general syntax of this clause is as follows:`USING` *softCondition* `FOR FUZZY SET` *fuzzySetName* (`,` *softCondition* `FOR FUZZY SET` *fuzzySetName*)* i.e., the clause can specify several branches. In each branch, *softCondition* is a condition whose terms are fuzzy-set names and "fuzzy operators" (a user-defined tool that evaluates membership degrees from properties; they will be presented hereafter); the resulting membership degree is referred to the *fuzzySetName* specified after the `FOR FUZZY SET` keywords.
Specifically, two soft conditions are specified in Listing 3: the first one evaluates the membership degree to the `MediumLengthHighways` fuzzy set, to obtain the positive satisfaction degree to the request; the second one evaluates the membership degree to the `NotMedium LengthHighway`, to obtain the degree of non satisfaction; notice that both the soft conditions exploit the *J-CO-QL*$^+$ `MediumLengthHighwayOp` fuzzy operator, that is defined in the `jcoContextOp1.jco` file (reported in Listing 4 and discussed hereafter).
Referring to Definition 4, the key *fuzzySetName* is added to the `fuzzysets` map; the value is the one obtained by evaluating the *softCondition*.
Figure 6b, c show such an evolution of the `fuzzysets` map.
- The `ALPHACUT` clause specifies a minimum threshold for the given fuzzy-set name(s), so as to select only those features whose membership degree is no less than the specified threshold(s). Specifically, only features whose membership degree to the `MediumLengthHighways` fuzzy set is no less than 0.8 are selected. This way, highways that are evaluated as not-completely medium-length highways, but at a sufficiently high degree, are selected.
- The `SELECT` clause projects properties of features only on properties of interest, by renaming them. Notice the `wanted` property, whose value is the membership degree to the `MediumLengthHighways` fuzzy set; the value is obtained through the `MEMBERSHIP_TO` built-in function; in the same way, the `notWanted` property is derived by taking the membership degree to the `NotMediumLengthHighway` fuzzy sets, to denote the degree of non satisfaction to the request. In fact, when the output *GeoJSON* document is generated, features are "de-fuzzified"; however, the membership degree is useful to know the relevance of the feature with respect to the query, thus the only way is to add a property with that value.
- Finally (remember Definition 7), the *GeoJSON* document is "de-fuzzified" (i.e., the `fuzzysets` map is removed from features) and the resulting crisp *GeoJSON* document is saved into the `MediumLengthHighways` collection in the `geosoftDb` database. Again, the `SETTING NAME` option defines the value for the root-level `name` field in the output *GeoJSON* document.

Listing 3 Soft Selection-Projection query in *GeoSoft*.

```
    GET CONTEXT "jcoContextDb.jco";
    GET CONTEXT "jcoContextOp1.jco";

    SELECT .COD_PE       AS .highwayId,
           .SHAPE_LEN    AS .highwayTotalLength,
           (MEMBERSHIP_TO (MediumLengthHighways))    AS .wanted,
           (MEMBERSHIP_TO (NotMediumLengthHighways)) AS .notWanted,
        FROM highways@geosoftDb
           WITH NAME "Highways"
        WHERE .SHAPE_LEN > 3000
        USING
           MediumLengthHighwayOp (.SHAPE_LEN)
               FOR FUZZY SET MediumLengthHighways,
           NOT MediumLengthHighways
               FOR FUZZY SET NotMediumLengthHighways,
        ALPHACUT MediumLengthHighways : 0.80
        SAVE AS MediumLengthHighways@geosoftDb
           SETTING NAME "Medium Highways";
```

The novel `USING` and `ALPHACUT` clauses are optional, as the `WHERE` clause: if they are present, they operate on the soft *GeoJSON* document generated by the previous clause (if present, it is the `WHERE` clause, otherwise it is the `FROM` clause). In particular, notice the clear distinction between crisp conditions (in the `WHERE` clause) and soft conditions (in the `USING` clause): this way, there is no semantic ambiguity, because crisp conditions keeps their well known crisp behavior; furthermore, soft conditions are not extensions of crisp conditions but are specified by a novel construct, clearly distinct from crisp conditions.

Thus, through basic queries it is possible to specify complex soft queries on *GeoJSON* documents, even based on the evaluation of multiple membership degrees to fuzzy sets.

Defining Fuzzy Operators. In *J-CO-QL*$^+$, the key to evaluate membership degrees of *JSON* documents from their fields is the concept of "fuzzy operator": it is a user-defined operator whose goal is to compute a membership degree, on the basis of the needs of the users. In *GeoSoft*, we decided to rely on fuzzy operators as they are provided by *J-CO-QL*$^+$: this way, we avoid to introduce a construct that would replicate the *J-CO-QL*$^+$ construct to define fuzzy operators in an exact way. Clearly, this decision is justified by the fact that a *GeoSoft* query is translated into a *J-CO-QL*$^+$ script, to be executed.

As far as the *GeoSoft* query is concerned, the definition of a fuzzy operator constitutes its execution context, as for connections to databases. Consequently, a file specified in the `GET CONTEXT` directive can contain *J-CO-QL*$^+$ definitions of fuzzy operators. In Listing 3, the second `GET CONTEXT` directive loads the `jcoContextOp1.jco` file, which contains the definition of the `MediumLengthHighwayOp` fuzzy operator, as reported in Listing 4. Hereafter, we present it in details.

The operator is named `MediumLengthHighwayOp`, because it will be used to evaluate if and how much a document (feature) describes a medium-length highway.

- The `PARAMETER` clause defines input formal parameters; their actual value will be provided by expressions on document fields.
- The `PRECONDITION` clause expresses a condition to be met before evaluating the membership degree; if the condition is not met, the evaluation is stopped. Specifically, the precondition is met if the `length` parameter is greater than 0.
- The `EVALUATE` clause evaluates a mathematical expression on parameters; the resulting value will constitute the basis for obtaining the actual membership degree. In line 2, the expression is very simple: it just contains the `length` parameter as it is.
- The last clause is the `POLYLINE` clause: it defines a membership function whose co-domain is [0, 1]. Figure 7 depicts the polyline, for the sake of clarity: provided the x-axis value by the `EVALUATE` clause, the corresponding y-axis value will be returned as membership degree. In case of x-axis value less than (respectively, greater than) the minimum (respectively, maximum) value, the y-axis value corresponding to the minimum (respectively, maximum) x-axis value is returned.

GeoSoft has inherited fuzzy operators from *J-CO-QL*$^+$. In our previous works [8, 36, 56, 60], we presented several examples of fuzzy operators that rely on complex poly-

Listing 4 Content of the `jcoContextOp1.jco` file.

```
CREATE FUZZY OPERATOR MediumLengthHighwayOp
  PARAMETERS      length TYPE Float          // in meters
  PRECONDITION    length > 0.0
  EVALUATE        length
  POLYLINE
    [ (    0, 0.0),  (10000, 0.0),  (15000, 0.3),
      (20000, 1.0),  (40000, 1.0),  (50000, 0.3),
      (60000, 0.0) ];
```

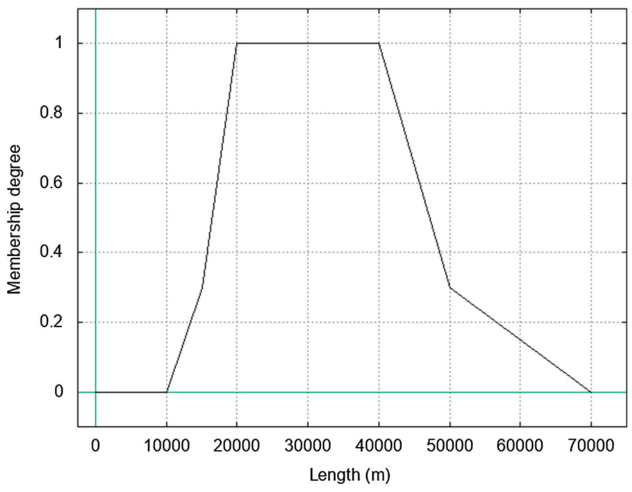


Fig. 7 Membership function of the `list:ContextOp1` fuzzy operator (in Listing 4)

lines. Traditionally, in the literature triangular and trapezoidal functions are considered, because they are intuitive to use; however, we argued that providing users with the possibility to define more complex (and not necessarily convex) shapes could increase the degree of flexibility for power users. As an example, in [56], we exploited this feature to compensate anomalous behaviours of the Jaro–Winkler string-similarity metric. Clearly, users that are familiar with trapezoidal functions can define them.

5.3 Nested Queries

When queries become more and more complex, “nested queries” could significantly help either keep complexity under control or specify sophisticated transformations (as in the classical SQL). *GeoSoft* allows nested queries to be specified only in the `FROM` clause.

The query reported in Listing 5 derives from the one presented in Listing 3; consequently, the addressed problem is the same.

The query relies on two nested queries. The inner-most query is a crisp query, since no membership degree is evaluated for features; in contrast, the intermediate one is a soft query. This difference is only apparent, since the semantic model is the same: when loaded, features become soft features, so they have the `fuzzysets` map; however, this map remains empty when no membership degree is evaluated. The `FROM` clause of the outer query receives a soft *GeoJSON* document from the nested query, independently of the fact that the nested query is crisp or not.

Hereafter, we explain the query reported in Listing 5.

- The innermost query retrieves the `highways` *GeoJSON* document from the database; features are projected on the `highwayId` field and on the `highwayTotalLength` (obtained by renaming the `SHAPE_LEN` property). The `fuzzysets` map of features in the output *GeoJSON* document is empty.
- The intermediate query evaluates the membership degrees to the `MediumLengthHighways` and to the `Not MediumLengthHighways` fuzzy sets; the resulting soft *GeoJSON* document contains all the features selected by the `WHERE` clause; as a result, the `fuzzysets` map has two entries, for each feature.
- The outermost query receives the soft features produced by the intermediate query and selects those of interest, i.e., those having a membership value no less than 0.8, and generates the output crisp *GeoJSON* document.

Again, the semantic model is still the same. We can understand that the internal semantic $\overline{gsq}$ corresponds to the *GeoSoft* clauses from `SELECT` to `ALPHACUT`. When a collection name is specified in the `FROM` clause, the input crisp *GeoJSON* document is implicitly transformed into a soft *GeoJSON* document. Finally, the `SAVE AS` clause is allowed only in the *dbq* database query, because (i) it transforms the output soft *GeoJSON* document into a crisp *GeoJSON* document and (ii) it saves the output crisp *GeoJSON* document into the database.

We can show that composition is encompassed by the semantic model. Indeed, if we denote the innermost, intermediate and outermost internal queries as $\overline{gsq}_i$, $\overline{gsq}_m$ and $\overline{gsq}_o$, respectively, the external query can be written as (moving from Definition 7):

Listing 5 Nested soft query in *GeoSoft*.

```
GET CONTEXT "jcoContextDb.jco";
GET CONTEXT "jcoContextOp1.jco";

SELECT .highwayId,  .highwayTotalLength,
       (MEMBERSHIP_TO (MediumLengthHighways))    AS .wanted,
       (MEMBERSHIP_TO (NotMediumLengthHighways)) AS .notWanted
   FROM ( SELECT .highwayId, .highwayTotalLength
            FROM ( SELECT .COD_PE AS .highwayId,
                          .SHAPE_LEN AS .highwayTotalLength
                     FROM highways@geosoftDb
                          WITH NAME "Highways" )
            WHERE .SHAPE_LEN > 3000
            USING
              MediumLengthHighwayOp (.highwayTotalLength)
                FOR FUZZY SET MediumLengthHighways,
              NOT MediumLengthHighwayOp
                FOR NotMediumLengthHighways )
   ALPHACUT MediumLengthHighways : 0.80
   SAVE AS MediumLengthHighways@geosoftDb
     SETTING NAME "Medium Highways";
```

$$gsq(gd) = toCrisp(\overline{gsq}_o(\overline{gsq}_m(\overline{gsq}_i(toSoft(gd))))).$$

Considering the overall database query (Definition 9), the full semantic expression is the following one:

$$dbq(to, from) = \\ = save(to, toCrisp(\overline{gsq}_o(\overline{gsq}_m(\overline{gsq}_i \\ (toSoft(get(from))))))).$$

5.4 JOIN Queries

A key success factor for a query language on *GeoJSON* documents is certainly the capability to integrate them. The construct we introduced to this end is the `JOIN` operator. This operator is a completely novel contribution, in comparison to our previous works on *GeoSoft* (see [9, 10]).

Syntax The `JOIN` operator is allowed in the `FROM` clause (as in the classical SQL). The syntax is as follows:

source$_1$ `AS` n_1`JOIN` *source*$_2$ `AS` n_2
  `ON` *[*`GEOMETRY` *spatialCond]*
    [`PROPERTIES` *propertyCond]*
  `SET GEOMETRY`
    [`SET FUZZY SETS` *fuzzySetSpec]*

that we explain hereafter.

- With *source*$_1$ and *source*$_2$ we denote either a collection name or a nested query. A collection name can be aliased by specifying a name after the `AS` keyword; a nested query must be aliased, necessarily.
- The `ON` keyword is followed by two distinct join conditions: the first one is introduced by the `GEOMETRY` keyword and specifies a join condition on geometries of features (it is denoted as *spatialCond*); the second one is introduced by the `PROPERTIES` keyword and specifies the join condition on properties of features (it is denoted as *propertyCond*). One of the two conditions can be omitted, but at least one of them must be specified.
- The `SET GEOMETRY` clause specifies how to determine the geometry of the output features (denoted as *geometrySpec*). We will say more in Appendix D.2.
- The `SET FUZZY SETS` clause is optional; if present, it provides a way to choose what membership degrees to choose from the ones of the source features. Since it is quite complex, it is not fully described hereafter; Appendix D.2 presents it in details.

The semantics of crisp `JOIN` operator is presented in details in Appendix D. The interested reader can refer to it for a fully understanding it.

Example Listing 6 reports a simple *GeoSoft* query that illustrates how to exploit soft spatial functions. The query looks for pairs of municipalities that share a significant part of their border. The query is explained hereafter.

- The `FROM` clause joins the *GeoJSON* document stored within the `towns` collection (see Information Layer 1) with itself. The resulting features pair two municipalities if their borders touch. The condition on properties that is specified after the `PROPERTIES` keyword is necessary to avoid coupling of a municipality with itself.
- The membership degrees to two fuzzy sets are evaluated by the `SET FUZZY SETS` clause. They make use of the `HOW_MEET` function, to obtain the degrees with

Listing 6 *GeoSoft* query with soft `JOIN`.

```
GET CONTEXT "jcoContextDb.jco";

SELECT .t1.nome_com AS .name1, .t2.nome_com AS .name2,
       (MEMBERSHIP_TO (BorderingLeft)) AS .SharedBorder1,
       (MEMBERSHIP_TO (BorderingRight)) AS .SharedBorder2
   FROM towns@geosoftDb AS t1
       JOIN towns@geosoftDb AS t2
           ON
               GEOMETRY MEET
               PROPERTIES .t1.nome_com != .t2.nome_com
           SET GEOMETRY INTERSECTION
           SET FUZZY SETS
               HOW_MEET (LEFT) AS BorderingLeft,
               HOW_MEET (RIGHT) AS BorderingRight
   ALPHACUT BorderingLeft : 0.4
   SAVE AS BorderingTowns@geosoftDb;
```

which the left (respectively, right) border meets the right (respectively, left) border. These fuzzy sets cannot be evaluated later, because the `HOW_MEET` function works on the source geometries, which will be lost after the `FROM` clause. Notice that `HOW_MEET` is an example of fuzzy spatial relationship between geometries.

- Neither the `WHERE` clause nor the `USING` clause are present in the query; indeed, the `ALPHACUT` clause suffices to select the features of interest, i.e., those features whose membership degree to the `BorderingLeft` fuzzy set is no less than 0.4.
- The `SELECT` clause projects on the names of the two municipalities, as well as on the membership degrees to the two evaluated fuzzy sets, renamed as `SharedBorder1` and `SharedBorder2`. Finally, the output crisp *GeoJSON* document is saved into the `BorderingTowns` collection.

Semantics The semantics of the `JOIN` operator is as follows.

- Consider two soft *GeoJSON* documents $\overline{gd}_1$ and $\overline{gd}_2$. The `JOIN` operator generates a new soft *GeoJSON* document, here denoted as $\overline{gd}_3$. Then, consider two features $\overline{f}_1 \in \overline{gd}_1$ and $\overline{f}_2 \in \overline{gd}_2$. A new $\overline{f}_3$ feature is generated.
- $\overline{f}_3$.`properties` contains two properties: the n_1 (alias of the first source *GeoJSON* document) property has the value $\overline{f}_1$.`properties`; the n_2 (alias of the second source *GeoJSON* document) property has the value $\overline{f}_2$.`properties`.
- If *spatialCond* is specified (after the `GEOMETRY` keyword), the spatial condition is evaluated on the two source geometries $\overline{f}_1$.`geometry` (i.e., left geometry) and $\overline{f}_2$.`geometry` (i.e., right geometry). Several predicates are admitted, such as `INTERSECT`, `MEET`, and so on. A detailed introduction is reported in Appendix D.
- If *propertyCond* is specified (after the `PROPERTIES` keyword), it is evaluated on the properties of the novel $\overline{f}_3$ feature, i.e., on $\overline{f}_3$.`properties`. Classical comparison predicates and mathematical expressions are admitted.
- $\overline{f}_3$.`geometry` assumes the value specified by *geometrySpec* after the `SET GEOMETRY` keywords, such as `INTERSECTION`, `UNION`, and so on (see Appendix D for a detailed introduction).
- $\overline{f}_3$.`fuzzysets` is populated as specified by the `SET FUZZY SETS` clause. The presence of membership degrees associated with features in the input soft *GeoJSON* documents to the `JOIN` operator is explicitly addressed by *GeoSoft*. Specifically, the `JOIN` operator has to deal with the following issues.
  - *Dealing with joined features both belonging to the same fuzzy set.* When the joined features belong to the same fuzzy set (possibly with different membership degree), it is necessary to manage this situation, so as to assign the output feature with the proper membership degree.
  - *Selecting the fuzzy sets of interest.* Not necessarily all the fuzzy sets for which membership degrees of the joined features are known are of interest for the output features. In other words, only a subset of them should be selected.
  - *Deriving membership degree to novel fuzzy sets.* Fuzzy spatial relationships that concern coupling of two features must be evaluated when they are paired. For example, the degree of inclusion of a geometry within the other geometry cannot be evaluated later, because the source geometries are lost.

Appendix D.2 presents details about the soft part of the `JOIN` operator; the interested reader can refer to it for a full understanding.

- The novel $\overline{f}_3$ feature is inserted into the output $\overline{gd}_3$ soft *GeoJSON* document if all the specified conditions (i.e., either *spatialCond* or *propertyCond* or both) are satisfied. If at least one of *spatialCond* and *propertyCond* is false, $\overline{f}_3$ is not inserted into $\overline{gd}_3$.

Notice that the semantics of the `JOIN` operator is perfectly integrated with the semantic model introduced in Sect. 5.1. Indeed, independently of the fact that the `FROM` clause contains one single source or a `JOIN` of sources, its output is always a soft *GeoJSON* document, on which subsequent clauses are evaluated.

5.5 A Complete Example

To conclude the introduction to *GeoSoft*, we show a complete example that exploits the main characteristics of the language.

Suppose that the user wants to find out those municipalities in the province of Milan that can be considered "medium towns", whose territory is crossed by "medium-length highways", such that the segment that traverses the municipality is a significant part of the overall highway. The *GeoSoft* query that solves this problem is reported in Listing 7.

As usual, the beginning of Listing 7 includes the `GET CONTEXT` directives. Notice that now all previous context files are specified, together with two novel files, named `jcoContextOp2.jco` (reported in Listing 8) and `jcoContextOp3.jco` (reported in Listing 9).

Listing 8 defines the `RelevantPortionOp` fuzzy operator. It receives two formal parameters, which are named `lengthInTown` and `totalLength`: the former is the length of the highway fragment that traverses a given municipality, the latter is the total length of the highway in the region. Once the precodition is satisfied, the `EVALUATE` clause computes the percentage of `lengthInTown` on `totalLength`; this value is the x-axis value used to get the final membership degree from the membership function defined by the `POLYLINE` clause. The membership function is depicted in Fig. 8: notice that it is 0 up to 1%, then it starts increasing, reaching 1 for 5% (meaning that we consider a highway fragment as fully interesting if its length in a municipality is at least the 5% of the overall length in the region).

Listing 9 reports the content of the `jcoContext Op3.jco` file. It contains the definition of a third fuzzy operator, named `MediumTownOp`. It is possible to see that it is a simple fuzzy operator, which receives the area of a municipality to evaluate if and how much a municipality is a

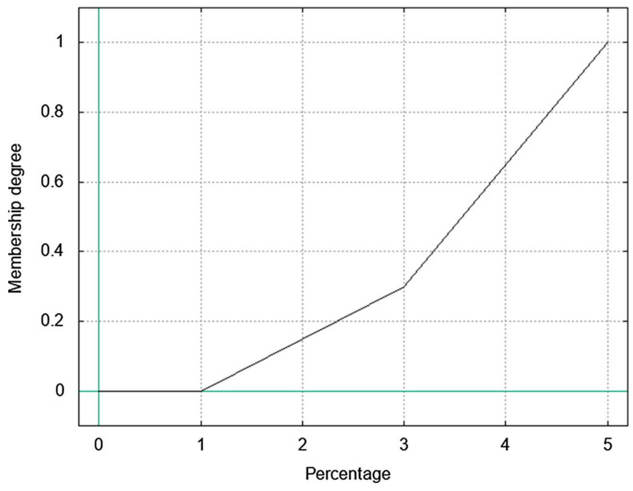


Fig. 8 Membership function for the fuzzy operator named `RelevantPortionOp` (in Listing 8)

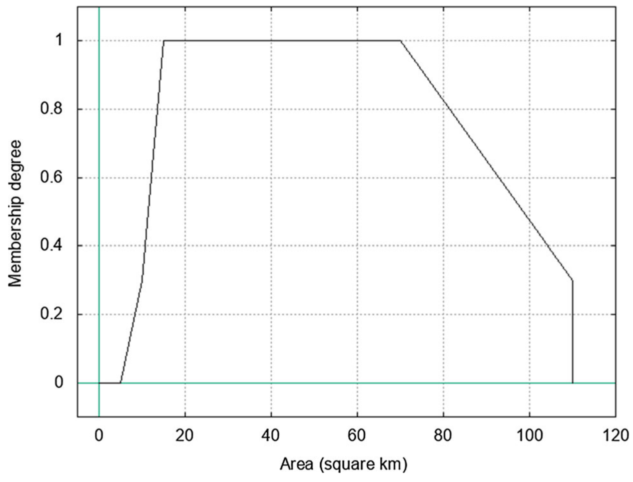


Fig. 9 Membership function for the fuzzy operator named `MediumTownOp` (in Listing 9)

"medium town"; Fig. 9 depicts the polyline defined as membership function. Again, remember that the `GET CONTEXT` directives exploit *J-CO-QL*$^+$ scripts as "execution context', in particular database connections and declarations of fuzzy operators; clearly, the content of this files is not part of the *GeoSoft* language, but will be inserted into the script obtained by translating the query into a *J-CO-QL*$^+$ script (see Sect. 6 and Appendix E.1).

Now, we are ready to explain the *GeoSoft* query.

- The blue box highlights the first nested query. Its goal is to work on features in the *GeoJSON* document describing towns (presented in Information Layer 1 and depicted in Fig. 2), stored within the `town` collection in the `geosoftDb` database.

Listing 7 Complex *GeoSoft* query based on soft `JOIN`.

```
GET CONTEXT "jcoContextDb.jco";
GET CONTEXT "jcoContextOp1.jco";
GET CONTEXT "jcoContextOp2.jco";
GET CONTEXT "jcoContextOp3.jco";

SELECT .t.town                  AS .town,
       .t.province              AS .province,
       .h.highwayId             AS .highwayId,
       .h.highwayName           AS .highwayName,
       .h.highwayTotalLength    AS .highwayTotalLength,
       (GEOMETRY_LENGTH ("M"))  AS .highwayLengthInTown,
       (MEMBERSHIP_TO (Wanted)) AS .wanted
   FROM   (SELECT  .nome_com     AS .town,
                   .nome_pro     AS .province
             FROM towns@geosoftDb
             WHERE .nome_pro = "MILANO"
             USING MediumTownOp (TO_FLOAT(.shape_area))
               FOR FUZZY SET MediumTowns
          ) AS t
     JOIN (SELECT  .COD_PE       AS .highwayId,
                   .NOME_PERCO   AS .highwayName,
                   .SHAPE_LEN    AS .highwayTotalLength
             FROM highways@geosoftDb
               WITH NAME "Highways"
             USING
               MediumLengthHighwayOp(.SHAPE_LEN)
                 FOR FUZZY SET MediumLengthHighways,
               NOT (MediumLengthHighways)
                 FOR FUZZY SET NotMediumLengthHighways
          ) AS h
       ON GEOMETRY INTERSECT
       SET GEOMETRY INTERSECTION
       SET FUZZY SETS
         RIGHT MediumLengthHighways,
         LEFT MediumTowns,
         RIGHT NotMediumLengthHighways
   WHERE .h.highwayTotalLength > 3000
   USING
     RelevantPortionOp(GEOMETRY_LENGTH ("M"), .h.highwayTotalLength)
       FOR FUZZY SET RelevantSegments,
     MediumTowns AND MediumLengthHighways AND RelevantSegments
       FOR FUZZY SET Wanted
   ALPHACUT Wanted : 0.80
   SAVE AS highwayTowns@geosoftDb
     SETTING NAME "Highway Towns";
```

The fuzzy operator named `MediumTownOp` is called in the `USING` clause, to evaluate the membership degree to the fuzzy set named `MediumTowns`.
Definitely, the goal of the nested query is to establish if municipalities, in the province of Milan, are likely to be considered medium towns.

Thus, the output soft features have the `fuzzysets` map with one single entry, i.e., the membership degree to the `MediumTowns` fuzzy set. The output soft *GeoJSON* document is aliased as `t`.

- The green box highlights the second nested query. It moves from the *GeoJSON* document describing high-

Listing 8 Content of the `jcoContextOp2.jco` file.

```
CREATE FUZZY OPERATOR RelevantPortionOp
  PARAMETERS
    lengthInTown TYPE Float,                // in meters
    totalLength  TYPE Float                 // in meters
  PRECONDITION   lengthInTown > 0.0
             AND totalLength  > 0.0
  EVALUATE       100 * lengthInTown / totalLength
  POLYLINE
    [ (0, 0.0),   (1, 0.0),
      (3, 0.3),   (5, 1.0)  ];
```

Listing 9 Content of the `jcoContextOp3.jco` file.

```
CREATE FUZZY OPERATOR MediumTownOp
   PARAMETERS     area TYPE Float    // in square meters
   PRECONDITION   area > 0
   EVALUATE       area / 1000000     // to square kilometers
   POLYLINE
     [ (  0, 0.0), ( 5, 0.0),  ( 10, 0.3),
       ( 15, 1.0), (70, 1.0),  (110, 0.3),
       (111, 0.0)  ];
```

ways (presented in Information Layer 2 and depicted in Fig. 4), which is stored within the `highways` collection in the `geosoftDb` database. The nested query evaluates if and how much a feature describes a medium-length highway or not (see the two evaluated fuzzy sets, i.e., `MediumLengthHighways` and `NotMediumLengthHighways`). Consequently, soft features in the output soft *GeoJSON* document have two entries in the `fuzzysets` member.
The output soft *GeoJSON* document is aliased as `h`.

- The red dashed box highlights the `JOIN` expression.
First of all (see the `ON GEOMETRY` clause), features in the two soft *GeoJSON* documents (aliased as `t` and `h`) are joined based on geometries: features are paired if their geometries intersect.
If pairing succeeds, the intersection of source geometries becomes the geometry of the novel feature (as specified by the `SET GEOMETRY` clause). Remember that properties in the novel feature are `t` (containing all properties coming from the left feature) and `h` (containing all properties coming from the right feature).
Finally, the `SET FUZZY SETS` clause specifies which membership degrees to fuzzy sets the novel feature must have. Specifically, the membership degree to the `MediumLengthHighways` is taken from the right feature (the `h` soft *GeoJSON* document), the membership degree to the `MediumTowns` fuzzy set is taken from the left feature (the `t` soft *GeoJSON* document), while the membership degree to the `NotMediumLengthHighways` fuzzy set is taken from the right feature (see Appendix D for details about the `SET FUZZY SETS` clause).
This way, the pool of soft features to evaluate in the following root-level clauses has been computed.
- The `WHERE` clause selects those composite features that describe municipalities (from the first nested query, in the province of Milan) that are crossed by a true highway, i.e., whose length is greater than 3000 m (to discard connections, whose length is a few hundred meters). This selection can be performed only in a crisp mode, thus it is specified in the `WHERE` clause.
- The `USING` clause actually performs the soft query. Specifically, we have two branches, because the membership degrees to two distinct fuzzy sets are evaluated.
The first fuzzy set that is evaluated by the `USING` clause is named `RelevantSegments`. Evaluated through the `RelevantPortionOp` fuzzy operator (reported in Listing 8), the resulting membership degree denotes if a significant portion of the highway crosses the municipality.
The second branch evaluates the membership degree to the `Wanted` fuzzy set: given a feature, its membership degree denotes the degree with which it satisfies the request. The soft condition is a fuzzy `AND` among three previously computed fuzzy sets, i.e., `MediumTowns`, `MediumLengthHighways` and `RelevantSegments`: this way, the soft condition linguistically expresses what the user is looking for, i.e., features that

Fig. 10 Excerpt of the *GeoJSON* document produced by the *GeoSoft* query reported in Listing 7

```
{
  "name"     : "Highway Towns",
  "type"     : "FeatureCollection",
  "features" : [
   {
    "type" : "Feature",
    "properties" : {
      "town"                  : "RHO",
      "province"              : "MILANO",
      "highwayId"             : "A50",
      "highwayName"           : "TANGENZIALE OVEST DI MILANO",
      "highwayLengthInTown"   : 4009.72781105714,
      "highwayTotalLength"    : 31997.4580651079,
      "wanted"                : 1
    },
    "geometry"   : {
      "type"                  : "MultiLineString",
      "coordinates"           : [
        [ [ 9.05717895599136, 45.5495986799443 ],
          [ 9.05724324188698, 45.5494843334202 ],
          …
          [ 9.0654992929525,  45.5037312658056 ]  ] ]
    },
    …,
    {…}
  ]
}
```

represent a medium town crossed by a medium-length highway, such that a significant portion of the highway crosses the town territory (the resulting membership degree is the minimum among the three mentioned fuzzy sets).

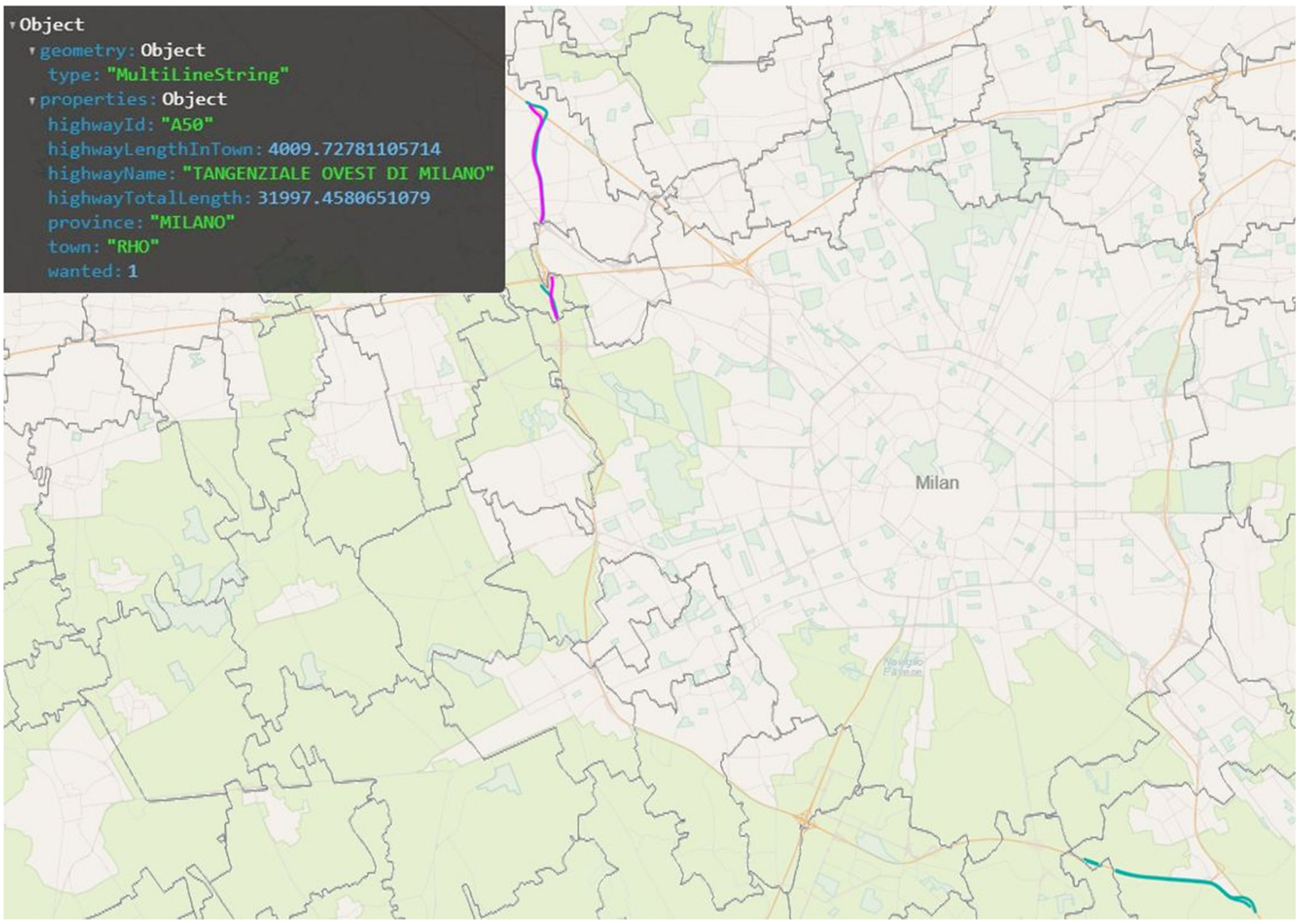
- The `ALPHACUT` clause selects only those features whose satisfaction degree is no less than 0.8, so as to keep only those features with a very-high satisfaction degree.
- Finally, the `SELECT` clause flattens the structure of properties. Furthermore, it adds two extra properties: the first one is named `highwayLengthInTown`, whose value is provided by the `GEOMETRY_LENGTH` built-in function (it provides the length, in meters, of the geometry); the second property is named `wanted`, whose value is the membership degree to the `Wanted` fuzzy set, by exploiting the `MEMBERSHIP_TO` built-n function.
- The `SAVE AS` clause transforms the final soft *GeoJSON* document into a crisp one and saves it into the collection named `highwayTowns`, within the `geosoftDb` database.

The reader can notice that, although the query is not trivial, it is not so complicated: in particular, the choice to rely on the well known syntax of the SQL `SELECT` statement, as well as the key semantic choice of viewing a *GeoJSON* document as a set of features, greatly simplify thinking and writing the query for analysts that are used to write SQL queries.

Figure 10 reports an excerpt of the *GeoJSON* document produced by the *GeoSoft* query in Listing 7, when applied on the same data sets described in Information Layer 1 and in Information Layer 2, while Fig. 11 shows the same *GeoJSON* document drawn on a map. In Fig. 11, the reader can notice that the highlighted (in purple) feature is a non-continuous line. This is the effect of the intersection of a line representing a highway with the area of a town, which can have a convex shape. The *GeoJSON* format supports these cases by means of the `MultiLineString` geometry type (see Appendix B).

6 From *GeoSoft* to *J-CO-QL*$^+$

GeoSoft works on a specific type of *JSON* documents, i.e., *GeoJSON* documents. *J-CO-QL*$^+$ (the query language of the *J-CO* Framework) is actually able to manage *GeoJSON* documents, since it is designed for manipulating any kind of *JSON* document; however, it is quite complex to do and we conceived the idea of developing *GeoSoft*. Indeed, the advantage provided by *GeoSoft* is that it is explicitly designed to query features within *GeoJSON* documents; consequently, queries are much easier to write than the corresponding *J-CO-QL*$^+$ scripts. Consequently, the *J-CO* Framework becomes the engine that performs *GeoSoft* queries, provided that these are translated into *J-CO-QL*$^+$ scripts.



```
▾Object
  ▾geometry: Object
    type: "MultiLineString"
  ▾properties: Object
    highwayId: "A50"
    highwayLengthInTown: 4009.72781105714
    highwayName: "TANGENZIALE OVEST DI MILANO"
    highwayTotalLength: 31997.4580651079
    province: "MILANO"
    town: "RHO"
    wanted: 1
```

Fig. 11 *GeoJSON* document produced by the *GeoSoft* query reported in Listing 7 drawn on a map. The black-box in the left-upper corner reports the properties contained in the *GeoJSON* feature representing the highway segment that is highlighted in purple in the left-upper corner of the map

In this section, we present the translation technique we implement into the *GeoSoft* compiler, to translate *GeoSoft* queries into *J-CO-QL*$^+$ scripts. To this end, it is necessary to introduce the data and execution models of *J-CO-QL*$^+$.

6.1 Brief Introduction to *J-CO-QL*$^+$

In order to fully understand how a *J-CO-QL*$^+$ script works, it is worth briefly introducing the underlying data and execution models.

6.1.1 Data Model

J-CO-QL$^+$ works on collections of standard *JSON* documents (see Appendix A). However, a special meaning is given to fields whose name begins with the "~" character: such field names are fully compliant with *JSON*naming rules, so documents with such fields can be saved into *JSON*stores without any problem. Currently, two special root-level fields are managed by *J-CO-QL*$^+$: they are named `~fuzzysets` and `~geometry`.

- The `~fuzzysets` field works as a map *fsn*→ *md*, where *fsn* is a fuzzy-set name and *md* is the corresponding membership degree; this way, the degrees of membership to multiple fuzzy sets of a document can be simultaneously represented.
- The `~geometry` field represents geometries of spatial entities possibly represented by *JSON* documents [7, 55]. We chose to rely on the same format for geometries of *GeoJSON*, which was described in Sect. 4. The advantage of this choice is straightforward: it is a standard format, which is world-wide adopted and can be easily managed. As far as this paper is concerned, it is the same format to deal with when querying *GeoJSON* features and their `geometry` fields.

6.1.2 Execution Model

We now briefly introduce the execution model of *J-CO-QL*$^+$ scripts.

A query $q = (i_1, \ldots, i_n)$ is a sequence of n (with $n > 0$) instructions i_j (with $1 \leq j \leq n$). Each instruction i_j takes a *query-process state* $s_{(j-1)}$ as input and generates a query-process state s_j as output.

A query-process state is a tuple

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

Hereafter, we explain each member.

- The tc member is named *temporary collection* and contains the current collection of *JSON* documents to process.
- The IR member is a local and volatile database, where the query can temporarily save *Intermediate Results*.
- The DBS member is the set of *database descriptors*, so as to connect to databases to retrieve/store collections of *JSON* documents.
- The FO member is the pool of *fuzzy operators* defined throughout the query, which are used to evaluate membership degrees of documents to fuzzy sets (see Sect. 5.2.2).
- The UDF member is the set of user-defined user-defined functions (written either in JavaScript or Java), defined to empower the query with additional computational capabilities (see [58]).

This execution model allows for writing complex and long queries, in a way that preserves the natural order with which transformations on collections are thought by human beings.

6.1.3 Brief Description of the *J-CO-QL*$^+$ Script

The *J-CO-QL*$^+$ script reported in Listings from 10 to 14 are obtained by translating the *GeoSoft* query reported in Listing 7. In principle, it could be written by hand directly as a *J-CO-QL*$^+$ script, but clearly using *GeoSoft* is much simpler. Hereafter, we provide a brief description, by shortly introducing the statements. This brief description is necessary to allow the reader to understand the translation strategy and algorithm; the reader that is interested in understanding in depth how the *J-CO-QL*$^+$ script works can find a complete description in Appendix E.

Databases and Fuzzy Operators Listing 4 reports the preliminary part of the script, i.e., the definition of connections to databases and the definitions of the fuzzy operators used later in the script. They are the same previously shown in the execution contexts of *GeoSoft* queries.

Loading a GeoJSON Document Listing 11 corresponds to the first nested query in Listing 7.

- The first three instructions (lines from 5 to 7) are necessary to acquire the *GeoJSON* document and adapt it to the *J-CO-QL*$^+$ data model. Specifically, the `GET COLLECTION` instruction retrieves the content of the specified database collection and makes it the current temporary collection.
- Then, the `EXPAND` instruction unnests *JSON* documents from within the `features` array, so as to obtain a single *JSON* document for each feature.
- Finally, the `FILTER` instruction adds the special `~geometry` field, using the former `geometry` field from features. Notice how the `CASE WHERE` crisp selection condition is used to specify which documents to work on, while the `GENERATE` clause contains all transformations performed on the selected documents, to generate the output ones (`SETTING GEOMETRY` specifies the geometry of the document, while `BUILD` restructure the output documents).

This is the general sequence to perform to load *GeoJSON* documents; indeed, the first three lines of Listing 9 (which corresponds to the second nested query in Listing 7) are identical.

Crisp and Soft conditions We now consider how *J-CO-QL*$^+$ deals with crisp and soft conditions on *JSON* documents.

- The `FILTER` instruction on line 8 of Listing 8 actually selects the features of interest and evaluates (through the `CHECK FOR` clause) the membership degrees to fuzzy sets (by adding the special `~fuzzysets`
- The `SAVE` instruction on line 9 saves the temporary collection into the database of Intermediate Results, to exploit it later.

Clearly, in Listing 9, which corresponds to the second nested query in Listing 7, lines 13 and 14 are identical, apart from the fact that the membership degrees to two fuzzy sets are evaluated.

Joining Documents In Listing 7, line 15 actually joins documents generated by Listings 8 and 9.

- The `ON GEOMETRY` clause specifies the spatial condition, while the `SET GEOMETRY` clause specifies how to derive the geometries of resulting documents. Specifically, given a document l in the left source collection and a document r in the source right collection, the output o document contains two fields named as the source aliases, whose value is the source l (respectively, r) document.
- The `ADD` clause adds extra fields to the o document: in this case, the `properties` field is added, so as to be coherent with the semantic model of features in *GeoSoft*.
- The `SET FUZZY SETS` clause evaluates membership degrees to fuzzy sets by exploiting both membership degrees already evaluated for source documents and spatial fuzzy relationships.
- Finally, the `CASE WHERE` clause selects documents of interest and the `CHECK FOR` clause evaluates member-

Listing 10 *J-CO-QL*$^+$ translation of the *GeoSoft* query in Listing 7: Execution context.

jcoContextDb.jco

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

JcoContextOp1.jco

```
2. CREATE FUZZY OPERATOR MediumLengthHighwayOp
      PARAMETERS      length TYPE Float          // in meters
      PRECONDITION    length > 0.0
      EVALUATE        length
      POLYLINE
        [ (    0, 0.0),     (10000, 0.0),   (15000, 0.3),
          (20000, 1.0),     (40000, 1.0),   (50000, 0.3),
          (60000, 0.0) ];
```

jcoContextOp2.jco

```
3. CREATE FUZZY OPERATOR RelevantPortionOp
      PARAMETERS
        lengthInTown TYPE Float,                 // in meters
        totalLength  TYPE Float                  // in meters
      PRECONDITION    lengthInTown > 0.0
                  AND totalLength  > 0.0
      EVALUATE        100 * lengthInTown / totalLength
      POLYLINE
        [ (0, 0.0),     (1, 0.0),
          (3, 0.3),     (5, 1.0)  ];
```

jcoContextOp3.jco

```
4. CREATE FUZZY OPERATOR MediumTownOp
      PARAMETERS      area TYPE Float    // in square meters
      PRECONDITION    area > 0
      EVALUATE        area / 1000000     // to square kilometers
      POLYLINE
        [ (  0, 0.0), ( 5, 0.0),  ( 10, 0.3),
          ( 15, 1.0), (70, 1.0),  (110, 0.3),
          (111, 0.0)  ];
```

ship degrees to fuzzy sets, as in the previous `FILTER` instructions.
The `ALPHACUT` clause selects documents having a membership degree for the specified fuzzy set no less than the specified threshold.

- Finally, output documents are restructured (by the `BUILD` action).

The temporary collection generated by Listing 7 contains documents that correspond to the features to put into the final *GeoJSON* document. This is generated by the tail of the script.

Generating the Output GeoJSONDocument

The final task to perform is to build he output *GeoJSON* document. This is done by the tail of the script, reported in Listing 14.

- Line 16 in Listing 7 prepares the documents to comply with the structure of features in the *GeoJSON* standard (see Appendix B). Documents are also "de-fuzzified", i.e., the `~fuzzysets` field is dropped, thus removing any reference to fuzzy sets.
- The `GROUP` instruction on line 17 groups together all documents, so as to obtain a unique document with the `features` array field.
- The `SAVE` instruction on line 18 actually saves the output *GeoJSON* document into a database.

6.2 Translation Strategy and Algorithm

We can now present how a *GeoSoft* query can be translated into a *J-CO-QL*$^+$ script. Hereafter, we first present the

Listing 11 *J-CO-QL*$^+$ translation of the *GeoSoft* query in Listing 7: First nested query.

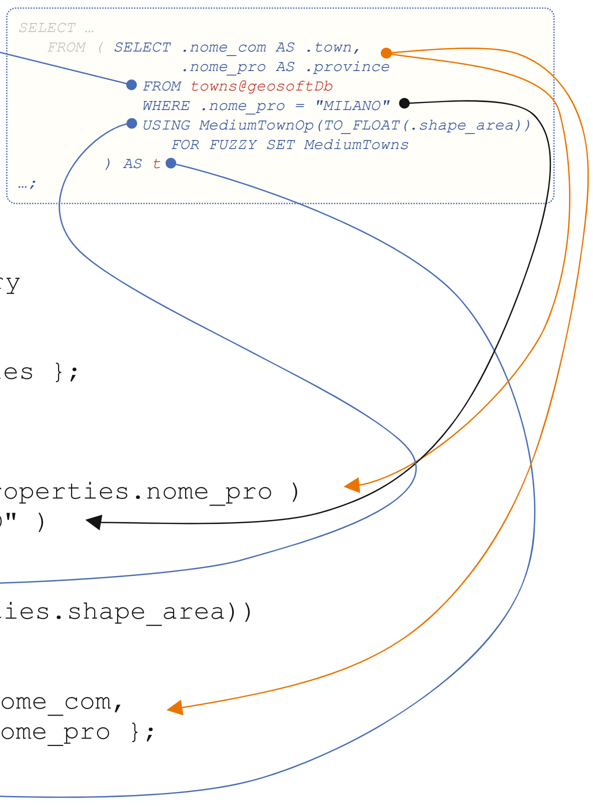
```
5.  GET COLLECTION towns@geosoftDb;

6.  EXPAND
    UNPACK WITH ARRAY .features
    ARRAY .features TO .feature;

7.  FILTER
    CASE WHERE WITH .feature.item
     GENERATE
       SETTING GEOMETRY .feature.item.geometry
       BUILD {
         .type       : .feature.item.type,
         .properties : .feature.item.properties };

8.  FILTER
    CASE WHERE WITH .type, .properties
         AND ( WITH .properties.nome_com, .properties.nome_pro )
         AND ( .properties.nome_pro = "MILANO" )
     GENERATE
       CHECK FOR FUZZY SET MediumTowns
         USING MediumTownOp(TO_FLOAT(.properties.shape_area))
       BUILD {
         .type                : "Feature",
         .properties.town     : .properties.nome_com,
         .properties.province : .properties.nome_pro };

9.  SAVE AS GeoSoftIntermediate_0_t;
```



```
SELECT …
   FROM ( SELECT .nome_com AS .town,
                 .nome_pro AS .province
          FROM towns@geosoftDb
          WHERE .nome_pro = "MILANO"
          USING MediumTownOp(TO_FLOAT(.shape_area))
             FOR FUZZY SET MediumTowns
        ) AS t
…;
```

translation strategy, that provides the rational behind the algorithm.

6.2.1 Translation Strategy

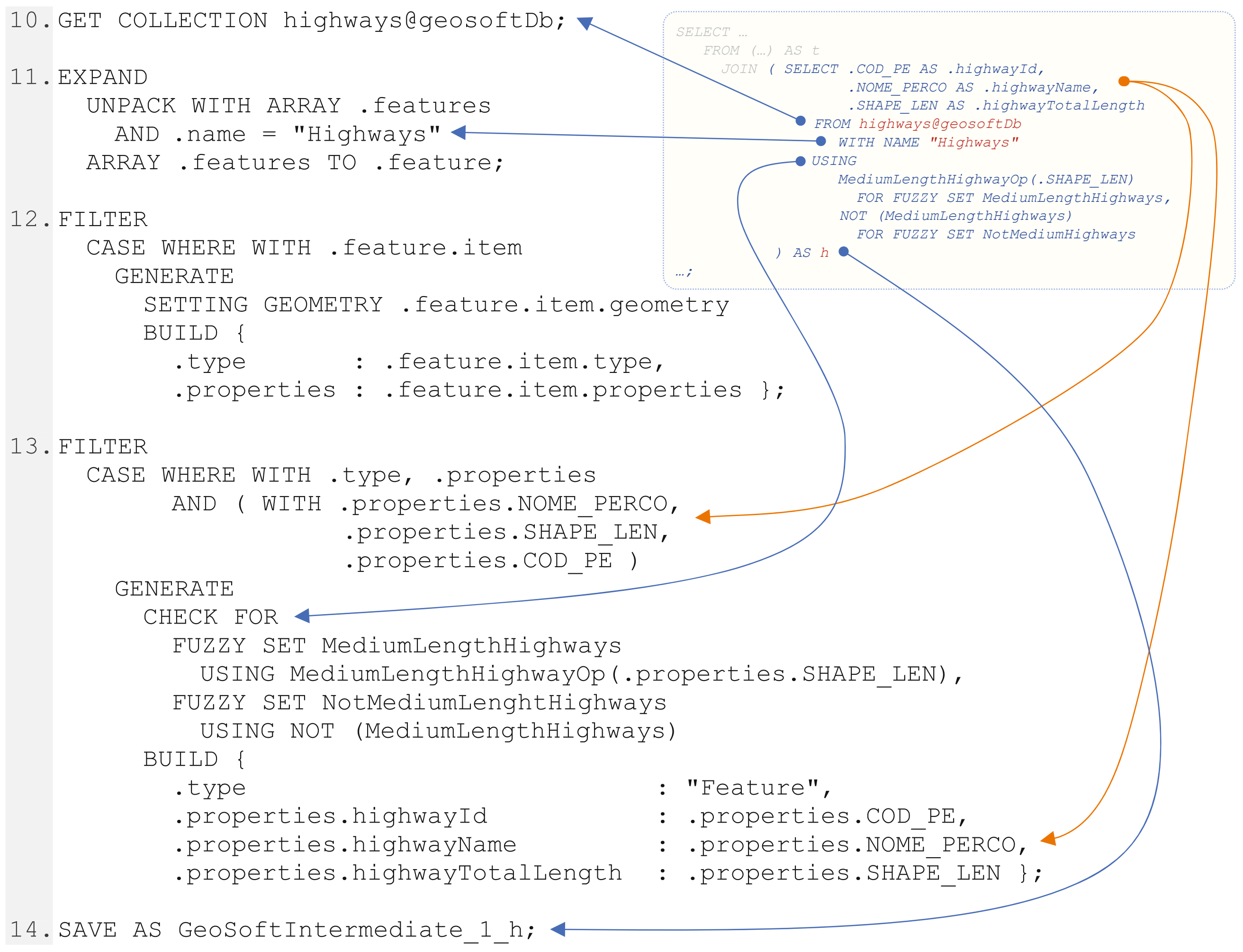
The translation algorithm is presented in Sect. 6.2.2. Before presenting it, we illustrate the rationale behind the translation strategy.

1. Source collections stored within the database are retrieved by *J-CO-QL*$^+$ through a `GET COLLECTION` instruction; then, features within the `features` array field must be unnested, so as to transform the source *GeoJSON* document into a collection of *JSON* documents, one for each feature. This is the general pattern to follow to acquire *GeoJSON* documents from the database.
2. Soft features are natively represented by *JSON* documents, enriched by means of the `~fuzzysets` field (see Sect. 6.1.1); furthermore, the original `geometry` member in features is translated into the `~geometry` field (see Sect. 6.1.1), which represents geometries in the data model of *J-CO-QL*$^+$.
3. A nested query is translated into a sequence of *J-CO-QL*$^+$ instructions. The output temporary collection becomes the input of the instructions that correspond to the clauses that follow the `FROM` clause in the outer query.
4. In the case of a `FROM` clause that contains the `JOIN` operator, the operands are preliminarily processed; since there is only one temporary collection in the query-process state of a *J-CO-QL*$^+$ script (see Sect. 6.1.2), it is necessary to save the temporary collection that contains the soft features provided by an operand into the *Intermediate Results* (*IR*) database. The *J-CO-QL*$^+$ instruction that actually performs the join (named `JOIN OF COLLECTIONS`, see Sect. 1) will later refer to this collection saved into the *IR* database.

Consequently, the general translation model is the following.

1. First of all, the "execution context" contained in the files specified by the `GET CONTEXT` directives is load as a general preamble. Remember that the execution context defines database connections and fuzzy operators, through *J-CO-QL*$^+$ instructions.
2. Input *GeoJSON* documents are acquired and features are unnested; indeed, a *GeoJSON* document is a collection of features; this collocation is represented as a collec-

Listing 12 *J-CO-QL*$^+$ translation of the *GeoSoft* query in Listing 7: Second nested query.



```
10. GET COLLECTION highways@geosoftDb;

11. EXPAND
      UNPACK WITH ARRAY .features
        AND .name = "Highways"
      ARRAY .features TO .feature;

12. FILTER
      CASE WHERE WITH .feature.item
        GENERATE
          SETTING GEOMETRY .feature.item.geometry
          BUILD {
            .type       : .feature.item.type,
            .properties : .feature.item.properties };

13. FILTER
      CASE WHERE WITH .type, .properties
            AND ( WITH .properties.NOME_PERCO,
                       .properties.SHAPE_LEN,
                       .properties.COD_PE )
        GENERATE
          CHECK FOR
            FUZZY SET MediumLengthHighways
              USING MediumLengthHighwayOp(.properties.SHAPE_LEN),
            FUZZY SET NotMediumLenghtHighways
              USING NOT (MediumLengthHighways)
          BUILD {
            .type                          : "Feature",
            .properties.highwayId          : .properties.COD_PE,
            .properties.highwayName        : .properties.NOME_PERCO,
            .properties.highwayTotalLength : .properties.SHAPE_LEN };

14. SAVE AS GeoSoftIntermediate_1_h;
```

```
SELECT …
   FROM (…) AS t
     JOIN ( SELECT .COD_PE AS .highwayId,
                   .NOME_PERCO AS .highwayName,
                   .SHAPE_LEN AS .highwayTotalLength
            FROM highways@geosoftDb
              WITH NAME "Highways"
            USING
              MediumLengthHighwayOp(.SHAPE_LEN)
                FOR FUZZY SET MediumLengthHighways,
              NOT (MediumLengthHighways)
                FOR FUZZY SET NotMediumHighways
          ) AS h
…;
```

tion of *JSON* documents that have the `properties`, `~geometry` and `~fuzzysets` fields; in particular, `~fuzzysets` is automatically added when the first membership degree to a fuzzy set is evaluated.

3. Nested *GeoSoft* queries are translated into a sequence of *J-CO-QL*$^+$ instructions, whose output temporary collection represents a collection of soft features.
4. The outermost *GeoSoft* query has to generate again a *GeoJSON* document; consequently, documents in the temporary collection that represent the output soft features are de-fuzzified and aggregated into one single *GeoJSON* document (this is the tail of the *J-CO-QL*$^+$ script).

6.2.2 Translation Algorithm

The translation algorithm is reported in Algorithm 1. We adopted a pseudo-code inspired to the Pascal programming language as far as the general syntax is concerned. Hereafter, we present it.

- The **RewriteDBQuery** function is the entry point of the algorithm. As denoted by its name, it actually deals with the full database query dbq. It is organized as explained hereafter.
  - The function receives one single parameter named dbq; it is a structured object that represents the database query (see Definition 9) to translate. The function returns the string containing the output *J-CO-QL*$^+$ script.
  - Line $D.1$ calls the **RewriteInternalQuery** function, which actually translates the outermost internal query (from the `SELECT` clause to the `ALPHACUT` clause), by generating a string with the corresponding *J-CO-QL*$^+$ script. The string is assigned to the T variable.
  - Line $D.2$ calls the **GenTail** function, whose goal is to generate the last part of the *J-CO-QL*$^+$ script. An example of tail is reported in Listing 14 (we discussed it in Section E.4). The string containing the

Listing 13 *J-CO-QL*$^+$ translation of the *GeoSoft* query in Listing 7: Outermost query.

```
15. JOIN OF COLLECTIONS
      GeoSoftIntermediate_0_t AS t,
      GeoSoftIntermediate_1_h AS h
    ON GEOMETRY INTERSECT
    SET GEOMETRY INTERSECTION
    ADD FIELDS {
      .properties.t : .t.properties,
      .properties.h : .h.properties    }
    SET FUZZY SETS
      RIGHT MediumLengthHighways,
      LEFT MediumTowns,
      RIGHT NotMediumLenghtHighways
    CASE WHERE  WITH  .properties
          AND ( WITH  .properties.h.highwayTotalLength,
                      .properties.t.province,
                      .properties.t.town,
                      .properties.h.highwayName,
                      .properties.h.highwayId )
          AND ( .properties.h.highwayTotalLength > 3000 )
      GENERATE
        CHECK FOR
          FUZZY SET RelevantSegments
            USING RelevantPortionOp( GEOMETRY_LENGTH ("M"),
                                  .properties.h.highwayTotalLength ),
          FUZZY SET Wanted
            USING MediumTowns AND MediumLengthHighways AND RelevantSegments
        ALPHACUT 0.80 ON Wanted
        BUILD {
          .type                              : "Feature",
          .properties.town                   : .properties.t.town,
          .properties.province               : .properties.t.province,
          .properties.highwayId              : .properties.h.highwayId,
          .properties.highwayName            : .properties.h.highwayName,
          .properties.highwayTotalLength     : .properties.h.highwayTotalLength,
          .properties.highwayLengthInTown    : (GEOMETRY_LENGTH ("M")),
          .properties.wanted                 : (MEMBERSHIP_TO(Wanted))    };
```

```
SELECT .t.town AS .town, …, …AS .wanted
 FROM   (…) AS t
  JOIN (…) AS h
   ON GEOMETRY INTERSECT
  SET GEOMETRY INTERSECTION
  SET FUZZY SETS
     RIGHT MediumLengthHighways,
     LEFT MediumTowns,
     RIGHT NotMediumHighways
 WHERE .h.highwayTotalLength > 3000
 USING
    RelevantPortionOp (GEOMETRY_LENGTH ("M"),
                  .h.highwayTotalLength)
      FOR FUZZY SET RelevantSegments,
    MediumTowns AND MediumLengthHighways
                AND RelevantSegments
      FOR FUZZY SET Wanted
  ALPHACUT Wanted : 0.80
```

tail is concatenated with the content of the T variable and assigned to the new R variable, whose value is returned by line $D.3$.
Since the work performed by the **GenTail** function is straightforward, we do not report it in Algorithm 1.

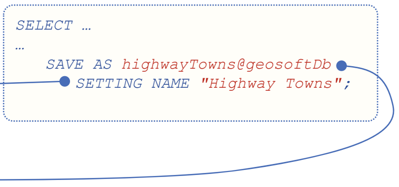
- The **RewriteInternalQuery** function is responsible for translating an internal query, independently of its nesting level. We explain it hereafter.
  - The function receives one single parameter, named iq: this is a structured object that describes the structure of the internal query. The function returns a string with the corresponding *J-CO-QL*$^+$ script.
  - The **If** instruction on line $I.1$ discriminates whether the FROM clause of the iq internal query contains a JOIN expression, because the translation strategy changes.
    - If the FROM clause of the iq internal query does not contain a JOIN expression, then *J-CO-QL*$^+$ instructions corresponding to the translation of the FROM clause generate a temporary collection that can be processed by a *J-CO-QL*$^+$ FILTER instruction (see, for example, Sect. 1 and Section E.2, with Listing 11 and Listing 12). Consequently, line $I.2$ calls the **Rewrite-Source** function to obtain the translation of the source (either collection or nested internal query), to which the "FILTER" constant string is appended; its CASE WHERE block will be generated by line $I.4$.

Listing 14 *J-CO-QL*$^+$ translation of the *GeoSoft* query in Listing 7: Tail.

```
16.FILTER
     CASE WHERE WITH .type, .properties
       GENERATE
         BUILD {
           .key        : 1,
           .type       : "Feature",
           .properties : .properties,
           .geometry   : GEOMETRY_FIELD() }
         DEFUZZIFY
         DROPPING GEOMETRY;

17.GROUP
     PARTITION WITH .key
       BY .key INTO .features
         DROP GROUPING FIELDS
       GENERATE
       BUILD {
         .type     : "FeatureCollection",
         .name     : "Highway Towns",
         .features : .features };

18.SAVE AS highwayTowns@geosoftDb;
```



```
SELECT …
…
    SAVE AS highwayTowns@geosoftDb
        SETTING NAME "Highway Towns";
```

- If the FROM clause contains a JOIN expression, its translation must save temporary collections into the *IR* database and must exploit the *J-CO-QL*$^+$ JOIN OF COLLECTIONS statement (see Sect. 1 and Listing 13). The corresponding translation is generated by calling the **RewriteJoinExpression** on line $I.3$.

  The **RewriteSource** function and the **RewriteJoinExpression** function will be explained later.
  - Independently of the specific case dealt with by lines from $I.1$ to $I.3$, the translation of clauses from WHERE to ALPHHACUT is the same, as illustrated in Sect. 1 and Sect. 1, with Listing 11 and Listing 13: a CASE WHERE block must be generated and appended to the translation so far obtained. This is obtained by calling the **GenCaseWhere** function on line $I.4$.
  - Finally, line $I.5$ returns the translation.

  Since the work performed by the **GenCaseWhere** function is straightforward (although it is not trivial), we do not report it in Algorithm 1.
- The **RewriteSource** function generates the translation of an s source specification in a FROM clause. It returns the string containing the generated translation. The function is explained hereafter.
  - If the source is not a nested internal query, i.e., it is a collection name, line $S.2$ calls the **GenAcquisition** function. It generates a sequence of *J-CO-QL*$^+$ instructions that acquire the content of the collection and expand nested documents (see Sect. 1 and Listing 11).
  - If the s source is a nested internal query, the **RewriteInternalQuery** function is recursively called by line $S.3$ to generate the translation of the nested internal query.

  Since the work performed by the **GenAcquisition** function is straightforward (although it is not trivial), we do not report it in Algorithm 1.
- The **RewriteJoinExpression** function translates a je JOIN expression and returns the corresponding string. It is explained hereafter.
  - Line $J.1$ generates two temporary names, which are assigned to the $Name_1$ and the $Name_2$ variables. Remember from Section E.2 that, in case of JOIN expression, it is necessary to save temporary collections describing soft *GeoJSON* documents into the *IR* database; the two temporary names will be used for this purpose.
  - In line $J.2$, by calling the **RewriteSource** function, the left operand of the JOIN expression is translated; the resulting *J-CO-QL*$^+$ script is concatenated with the SAVE AS instruction that saves the temporary collection into the *IR* database.

    Line $J.3$ performs the same task on the right operand of the je JOIN expression.

Algorithm 1 Translation of *GeoSoft* queries into *J-CO-QL*$^+$ scripts.

```
      Function RewriteSource(s: SourceSpecification): String
      Begin
S.1   If s Is Not a Nested Query Then
S.2       T := GenAcquisition(s)
      Else
S.3       T := RewriteInternalQuery(s);
      End If
S.4   Return T;
      End Function

      Function RewriteJoinExpression(je: JoinExpression): String
      Begin
J.1   $Name_1$ :=GenTempName(); $Name_2$ :=GenTempName();
J.2   $T_1$ :=RewriteSource(je..leftSource)•"SAVE AS "•$Name_1$ •";";
J.3   $T_2$ :=RewriteSource(je.rightSource)•"SAVE AS "•$Name_2$ •";";
J.4   $T_J$ :=GenJoinOfCollections(je, $Name_1$, $Name_2$);
J.5   T := $T_1$ • $T_2$ • $T_J$;
J.6   Return T;
      End Function

      Function RewriteInternalQuery(iq: InternalQuery): String
      Begin
I.1   If iq.FROM Is Not a JOIN Expression Then
I.2       T := RewriteSource(iq.FROM)•"FILTER";
      Else
I.3       T :=RewriteJoinExpression(iq.FROM);
      End If
I.4   R := T•GenCaseWhere(iq);
I.5   Return R;
      End Function

      Function RewriteDBQuery(dbq: DBQuery): String
      Begin
D.1   T := RewriteInternalQuery(dbq.InternalQuery);
D.2   R := T• GenTail(dbq.SaveAs);
D.3   Return R;
      End
```

Notice that the translations are assigned to the T_1 and T_2 variables.

- Line $J.4$ calls the **GenJoinOfCollections** function, whose goal is to generate the first part of the *J-CO-QL*$^+$ JOIN OF COLLECTIONS instruction that actually performs the soft join (see Sect. 1 and the part of Listing 13 that precedes the CASE WHERE clause). The translation is assigned to the T_J variable. Since the work performed by this function is straightforward (although it is not trivial), its code is not reported in Algorithm 1.
- Finally, the three translations are concatenated by line $J.5$ and returned by line $J.6$.

Algorithm 1 shows that translating a (possibly complex) *GeoSoft* query is possible; the translation process is linear and well structured.

In order to avoid boring the reader, we do not explain in details how the *GeoSoft* query reported in Listing 7 is translated. The interested reader can find an extensive description in Appendix E.

7 Evaluation

Before concluding the paper, we summarize the results by evaluating three critical aspects, i.e., “flexibility”, “accessibility” and “efficiency”.

7.1 Flexibility

First of all, we consider “flexibility”. What kind of flexibility is provided by *GeoSoft*?

- Typically, query languages that rely on fuzzy sets and linguistic predicates are considered as “flexible query languages”, in the sense that queries expressed by means of which are tolerant to imprecision and vagueness. From this point of view, *GeoSoft* is certainly a flexible query language.
- The extent of flexibility provided by *GeoSoft* is focused at the level of feature: it does not consider single fuzzy values; in contrast, it considers fuzzy sets defined in the universe of spatial features. As an effect, a spatial feature can belong to multiple fuzzy sets (of spatial features).
- Fuzzy operators are used to evaluate membership degrees of spatial features to fuzzy sets, on the basis of values of feature properties. This approach has been inherited from *J-CO-QL*$^+$: in particular, a distinctive characteristic is the adoption of a polyline as membership function, in place of more traditional triangular and trapezoidal functions (that can be easily defined by means of polylines). This choice further increases the level of flexibility, because complex shapes can be exploited by analysts to deal with complex situations (as we did in [56]).
- Currently, *GeoSoft* does not provide “quantifiers”: a quantifier is an aggregator of several membership degrees, whose goal is to express quantification of linguistic predicates. The reason why quantifiers are not considered in *GeoSoft* is that they are not provided by *J-CO-QL*$^+$ either; we are aware of this lack, but we decided to delay addressing this problem, waiting for the right time in the development of the language. Indeed, we are currently addressing the issue of defining user-defined fuzzy aggregators (and quantifiers can be seen as specific fuzzy aggregators). When this concept is available in *J-CO-QL*$^+$ we will promptly introduce it in *GeoSoft*, together with the GROUP BY clause.
- *GeoSoft* accompanies spatial join with specific functions to evaluate fuzzy spatial relationships among geometries.

This way, through a spatial join, the membership degrees to fuzzy sets of resulting features on the basis of spatial fuzzy relationships can be evaluated at once.

- In our opinion, another issue towards flexibility is the clear separation between crisp conditions and soft conditions. Indeed, previous proposals usually modify the semantics of the classical `WHERE` clause from crisp conditions to soft conditions. In our opinion, this approach is not the right one, because it is true that crisp predicates can be seen as soft predicates whose membership degree is either 0 or 1, but often the crisp and soft world should not be mixed. This is why *GeoSoft* provides both the `WHERE` clause (crisp selection condition) and the `USING` clause (soft conditions that evaluate membership degrees to fuzzy sets).

Nonetheless, we can consider another meaning of flexibility, i.e., the ability to simplify data management and analysis tasks. In this sense, *GeoSoft* is the most flexible solution for analyzing *GeoJSON* information layers stored within a *JSON* document store, for several reasons.

- It works directly on the source *GeoJSON* documents and generates *GeoJSON* documents.
- It does not require to perform tedious transfers of *GeoJSON* document into a different DBMS, such as PostgreSQL/PostGIS, fighting against import/export tools.
- It is completely independent of the specific database technology and of its specific query language.

7.2 Accessibility

The second aspect we consider to evaluate *GeoSoft* is "Accessibility", which we can be intended as the capability of the proposed tool to remove obstacle to manage *GeoJSON* information layers.

- Section 6 presented the translation method we implemented to translate *GeoSoft* queries into *J-CO-QL*$^+$ scripts. The length of *J-CO-QL*$^+$ scripts obtained from (possibly short) *GeoSoft* queries clearly show how *GeoSoft* greatly improve the accessibility of users to *GeoJSON* querying; the choice for adopting a SQL-like syntax for *GeoSoft* further improve accessibility.
- In this paper, we did not consider other query languages for *JSON* documents, such as *MQL* (the *MongoDB* Query Language), but we can say a few words. Syntactically, *MQL* is a JavaScript method calls, where a query is specified as a *JSON* document provided as parameter: this choice makes quite hard to directly work on document nested within array fields, so a *GeoJSON* document should be preliminarily split into single *JSON* documents that must be saved into the database itself, with the effect of making the database dirty. Furthermore, users coming from SQL are not familiar with this approach, as well as *MQL* does not support soft querying. Consequently, we can state that *GeoSoft* provides users with a significantly higher degree of accessibility for querying of *GeoJSON* documents.
- Would it be possible to perform the same kind of analysis with *MongoDB*? The answer is "no", because *MongoDB* supports only "range queries" on previously-indexed *JSON* documents (not on *GeoJSON* information layers) on the basis of their geo-tagging.
- Would it be possible to perform similar queries with PostgreSQL/PostGIS? the answer is "yes", because PostGIS is quite complete. Nonetheless, queries cannot be performed directly on *GeoJSON* documents, but on tables; this means that features within *GeoSoft* documents must be preliminarily imported into tables, processed and exported again as *GeoJSON* documents. *GeoSoft* overtakes all these passages, enabling users to directly work on *GeoJSON* information layers stored within *JSON* data stores.

7.3 Efficiency

The last aspect we consider is "efficiency". To evaluate this aspect, we performed experiments with the real data within the *GeoJSON* documents presented in Information Layer 2 and information Layer 1. Experiments were conducted on a common laptop powered by an Intel quad-Core i7-8550-U Processor, running at 1.80 GHz, equipped with 16 GB RAM and 250 GB Solid-State Drive. The *J-CO-QL*$^+$ *Engine* is implemented with the Java language.

GeoSoft executed via J-CO-QL$^+$ ***script*** The *GeoJSON* query reported in Listing 7 is translated into the *J-CO-QL*$^+$ script reported in Listings from 10 to 14, which was summarized in Sect. 6.1.3 and is extensively presented in Appendix E.

Specifically, the `Highways` information layer describes 94 highways; the `Towns` information layer describes 1506 municipalities. This data set is named *Full Data Set*, to distinguish it from the other two data sets that will be discussed later.

Table 1 reports the result of our experiments. For each data set, the execution times measured for each single *J-CO-QL*$^+$ instruction are reported (left column, labeled as *Partial*) and the sum of execution times is reported on the right column (labeled as *Total*).

The total execution time for the *Full Data Set* is 26 *sec*; looking at the single instruction, it is possible to see that the `JOIN OF COLLECTIONS` instruction takes 1.5 *sec*, so, there are other instructions that determine the overall execution time. These instructions are those from 5 to 7: they

Table 1 Execution times for each *J-CO-QL*$^+$instruction reported in Listings from 10 to 14

#	Instruction	Full data set		Milan data set		Medium data set	
		Partial (ms)	Total (ms)	Partial (ms)	Total (ms)	Partial (ms)	Total (ms)
1	USE DB	179	179	167	167	164	164
2	CREATE FUZZY OPERATOR	0	182	0	171	0	168
3	CREATE FUZZY OPERATOR	0	182	0	171	0	168
4	CREATE FUZZY OPERATOR	0	182	0	171	0	168
5	GET COLLECTION	5,165	5,348	525	696	270	439
6	EXPAND	12,138	17,486	935	1,631	398	837
7	FILTER	5,094	22,582	533	2,167	308	1,147
8	FILTER	370	22,953	231	2,398	112	1,259
9	SAVE AS	970	23,924	446	2,845	246	1,506
10	GET COLLECTION	171	24,095	103	2,949	94	1,601
11	EXPAND	217	24,313	139	3,088	204	1,806
12	FILTER	106	24,420	63	3,151	111	1,917
13	FILTER	63	24,484	35	3,187	56	1,973
14	SAVE AS	97	24,582	66	3,253	69	2,043
15	JOIN OF COLLECTIONS	1,562	26,147	1,046	4,301	395	2,440
16	FILTER	6	26,154	6	4,308	3	2,444
17	GROUP	3	26,157	2	4,311	1	2,446
18	SAVE AS	72	26,229	61	4,372	48	2,494

correspond to the acquisition of the `Towns` *GeoJSON* document and its transformation into single *JSON* documents: in particular, the `EXPAND` instruction is the slowest, because it has to unnests documents with very complex geometries.

In order to better evaluate this behavior, we built a second data set, named `Milan Data Set`; in this data set, we selected only the 133 municipalities in the province of Milan; so the input `Towns` *GeoJSON* document now contains about one-tenth of the initial features.

Looking at Table 1possible to see that now the overall query is executed in 4.3 s; in particular, the critical instructions (from 5 to 7) now take 2 s, i.e., ten times faster (as expected).

Finally, we further restricted the municipalities in the `Towns` document, considering only the 22, in the province of Milan, whose area is between 15 km^2 and 70 km^2 (which correspond to the area, for a municipality, to have a full membership to the `MediumTowns` fuzzy set, according to the `MediumTownOp` fuzzy operator defined in Listing 9). The `Highways` document still contains all the initial features. The data set is named *Medium Data Set*.

The overall time is now 2.5 *sec*; in particular, notice that the `JOIN OF COLLECTIONS` instruction now takes only 0.4 *sec* Considering that the implementation of the `JOIN OF COLLECTIONS` statement is not optimized by spatial indexing, it behaves satisfactorily, not being the bottleneck of the process.

Thus, the bottleneck is the loading of large *GeoJSON* documents in which features have very complex geometries, such as borders of municipalities.

Comparison with PostgreSQL/PostGIS In order to compare *GeoSoft* (and its execution through *J-CO-QL*$^+$) with traditional solutions, we created an Object-Relational database managed by *PostgreSQL*[11] and its extension for spatial data, named *PostGIS*.[12] The two information layers were uploaded, both in the *Full Data Set* version and in the *Medium Data Set* version; two *PostGIS* tables for each version were created, having one row for each feature. A join query based on the spatial intersection of geometries was performed, so as to spatially intersect towns and highways; this query is comparable with the work performed by the `JOIN OF COLLECTIONS` instruction in the *J-CO-QL*$^+$ script, even though the *PostGIS* join does not evaluate soft conditions (as in the *J-CO-QL*$^+$ `JOIN OF COLLECTIONS`).

Processing the *Full Data Set*, *PostGIS* takes 1.2 *sec*, against 1.6 *sec* taken by the *J-CO-QL*$^+$ `JOIN OF COLLECTIONS`.

Processing the *Medium Data Set*, *PostGIS* takes 0.2 *sec*, against 0.4 *sec* taken by the *J-CO-QL*$^+$ `JOIN OF COLLECTIONS`.

[11] PostgreSQL: https://www.postgresql.org/download/, accessed on 01/07/2023.

[12] PostGIS: https://postgis.net/documentation/getting_started/, accessed on 01/07/2023.

It is possible to observe that in absolute terms *PostGIS* is faster than *J-CO-QL*$^+$, in particular if the overall script is considered. However, this experiment does not consider how to upload information layers to *PostGIS* tables and how to export *PostGIS* tables to *GeoJSON* documents. To this end, several tools can be used, but they require the user to execute them manually, thus resulting in tedious activities that more or less can take (when things go fine at the first attempt) at least the same time needed by the overall *J-CO-QL*$^+$ script on the *Full Data Set*, i.e., 26 *sec*. In contrast, the *GeoSoft* query works directly on the information layers stored within *JSON* document stores: users do not have to waste time in other activities, they can concentrate on the analysis they want to perform.

8 Conclusions and Future Works

The contribution of the paper is the *GeoSoft* proposal: it is conceived as a domain-specific language for writing sophisticated queries on features described by *GeoJSON* documents. Hereafter, we summarize the major points that characterize this work.

- The choice for the *GeoJSON* format is motivated by its popularity. Furthermore, moving from the high-level view of a *GeoJSON* document as a "set of features", it is straightforward (for us) to conceive to adopt the same approach on which SQL relies for querying features of *GeoJSON* documents, so as to obtain other *GeoJSON* documents.
- The idea of defining *GeoSoft* originated from the work we are performing on the *J-CO* Framework. It is a pool of software tools built around the *J-CO-QL*$^+$ language, designed to operate on *JSON* data sets at a high level of abstraction, if compared with other languages for querying *JSON* documents (see [7] for a detailed comparison). In particular, native capabilities of managing geo-tagging and soft querying *JSON* documents makes *J-CO-QL*$^+$ a unique proposal. Furthermore, the independence of any specific *JSON* store makes the *J-CO* Framework particularly indicated to integrate data sets coming from multiple *JSON* document stores.
- However, any general-purpose language (even if it is very powerful) may fall into situations that can be solved only by writing long and complex sequences of instructions. This happens with *J-CO-QL*$^+$ as well, in particular when dealing with formats based on many nesting levels, such as *GeoJSON*: writing *J-CO-QL*$^+$ scripts for querying features in *GeoJSON* documents is usually cumbersome, even though such *J-CO-QL*$^+$ scripts are effective. The approach followed for *GeoSoft*, i.e., devising a query language specifically designed for features in *GeoJSON* documents, which exploits the syntax of the classical `SELECT` statement in SQL, dramatically reduces the effort to write complex queries.
- Nevertheless, the distinctive features provided by *J-CO-QL*$^+$ are still precious, in particular native management of geo-tagging and soft querying capabilities: *GeoSoft* exploits them. Consequently, the *J-CO* Framework plays the role of execution engine of *GeoSoft* queries, because they are translated into *J-CO-QL*$^+$ scripts.

Both the *J-CO* Framework and the *GeoSoft* compiler are publicly available on a GitHub repository.[13]

As a future work, several research lines could be followed, which concern the *J-CO* Framework in general and possibly affect *GeoSoft*.

- The *J-CO-QL*$^+$ language is undergoing a continuous evolution, in particular as far as support for soft querying is concerned.
  Specifically, we are going to address the problem of defining complex fuzzy aggregators, so as to enable soft queries that rely on complex soft aggregations. Once defined, we will address, in *GeoSoft*, both the definition of quantifiers and the introduction of the `GROUP BY` clause.
- *Soft Web Intelligence* is the concept we are exploring in a parallel research work [66]. It can be considered as a modern interpretation of the 20-year-old notion of *Web Intelligence*. Currently, we demonstrated that the *J-CO* Framework can play a key role in envisioning the World-Wide Web as a giant data store, supporting its exploitation by means of soft computing and soft querying. From *GeoSoft* a novel language could derive, that could provide analysts with the abstraction to easily write complex (fuzzy) aggregations on data set coming from the Web.
- From the point of view of applications, the line of developing domain-specific languages for particular formats of *JSON* documents can be replicated for other formats. For example, a possible application context could be "linked data": for example, the *JSON-LD* format [67] is meant to substitute the *Resource Description Framework (RDF)* [68] in the context of semantic web.
- As we said, the *J-CO* Framework and *J-CO-QL*$^+$ are general purpose, thus they could be applied to many different application contexts. We already experimented its use in the context of geographical-data integration [56]. The result we obtained encourages us to explore novel application contexts, such as building a product search-engine based on queries written in natural language over

[13] *J-CO* Project GitHub landing page: https://github.com/JcoProjectTeam/JcoProjectPage, accessed on 01/07/2023.

product reviews posted by users and customers (moving from our past experience [69, 70]).

Author Contributions All authors whose names appear on the submission declare that: (1) made substantial contributions to the conception or design of the work; or the acquisition, analysis, or interpretation of data; or the creation of new software used in the work; (2) drafted the work or revised it critically for important intellectual content; (3) approved the version to be published; and (4) agree to be accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved.

Funding The authors did not receive support from any organization for the submitted work.

Data Availability Statement The authors declare that the data sets presented in this paper are available on the GitHub repository (URL: https://github.com/JcoProjectTeam/JcoProjectPage). The authors declare that the tools presented in this paper are available on GitHub repository (URL: https://github.com/JcoProjectTeam/JcoProjectPage).

Declarations

Conflict of interest The authors declare they have no conflict of interest.

Ethical standard The authors declare that the presented research has no ethical concerns.

Informed consent All the authors give their consent for publication.

A. Brief Introduction to the *JSON* Format

To illustrate the *JSON* format, Fig. 12 reports a *JSON* document.

- A document is represented within a pair of braces "`{`" and "`}`"; it is a sequence of fields separated by commas. A field is a "name: value" pair, where "name" is the field name.
  The document reported in Fig. 12 has five fields named `Name`, `FoundationYear`, `FounderName`,`MainAddress` and `Products`.
- Elementary values for fields can be either Boolean values, or numbers (either integer or real numbers) or strings.

```
{
    "Name"             : 'ACME',
    "FoundationYear"   : 2000,
    "Founder"          : "J.C. Ross",
    "MainAddress"      : {
        "Street"   : "Street 1",
        "City"     : 'City 1',
        "Country"  : "Country 1"
    },
    "Products"         : [
        {
            "Code"        : "111",
            "Description" : "Pencil",
            "Price"       : 1.2
        },
        {
            "Code"        : "112",
            "Description" : "Black Pen",
            "Price"       : 0.8
        }
    ]
}
```

Fig. 12 Example of *JSON* document

  Considering strings, while the field name is always enclosed within double quotes, strings can be enclosed either within double quotes or within single quotes.
  In the sample document in Fig. 12, both the `Name` and the `Founder` fields have a string as a value, but the first string is enclosed within single quotes, while the second within double quotes.
- A nested sub-document can be the value of a field too. A sub-document is enclosed within a pair of braces "`{`" and "`}`", as for the root-level document.
  In Fig. 12, the `MainAddress` field has a sub-document as a value; specifically, three fields are encompassed in this nested sub-document, i.e., `Street`, `City` and `Country`.
- Finally, the value of a field can be an array (enclosed within square brackets "`[`" and "`]`"), whose items can be any kind of *JSON* value, separated by commas.
  For example, the `Products` field in the sample document in Fig. 12 is an array of documents. Specifically, two documents describe products that are sold by the company described by the document.

B. Details of the *GeoJSON* Format

According to its official specification,[14] a *GeoJSON* document can represent three types of documents (which, at the end, are *JSON* documents based on a precise structure).

- **Geometry.** This is the simplest type of spatial information that it is possible to represent. Such a document

[14] *GeoJSON* specification: https://www.rfc-editor.org/rfc/rfc7946, accessed on 01/07/2023.

contains a mandatory `type` field, which denotes the type of described geometry. The `type` field can assume one of the following values: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLine String"` and `"MultiPolygon"`; in these cases, the `coordinates` field is an array of two coordinates (for `"Point"`), an array of points (for `"LineString"`, `"Polygon"` and `"MultiPoint"`), an array of line strings (for `"MultiLineString"`) and an array of polygons (for `"MultiPolygon"`).
Finally, the `type` field can also have the `"Geometry Collection"` value; in this case, the `geometries` field (that replaces the `coordinates` field) is an array of documents describing geometries (so as to describe very complex and heterogeneous geometries).

- **Feature.** This type of document represents a spatial entity, in terms of geometry and *properties* (i.e., registry data about the entity).
  It contains three mandatory fields. The first one is `type`, whose value is set to `"Feature"` to denote that the document describes a feature. The second field is `geometry`, whose value is a *GeoJSON Geometry* document (as specified before); the third field is `properties`, whose value is a common *JSON* document that describes properties (i.e., registry data) of the described spatial entity.
- **Feature Collection.** This type of document represents a set of *Feature* documents. In its minimal form, it encompasses two mandatory fields: the `type` field has the fixed `"FeatureCollection"` value, to denote that the document describes a feature collection; the `features` field is an array of *Feature* documents (as specified above).
  Additionally, a *Feature Collection* document can hold an optional `name` field, whose value gives a name to the collection; furthermore, an optional `crs` field is a nested *JSON* document representing the Coordinate Reference System used for geometries in the features that are described within the main document.

C. Brief Introduction to Fuzzy Sets

Fuzzy-Set Theory was introduced in 1965 by Zadeh in [11]; here, we sketch basic concepts.

Let us suppose that U denotes a non-empty universe, either finite or infinite.

Definition 10 A fuzzy set A in U is defined through a mapping $\mu_A : U \rightarrow [0, 1]$. The value $\mu_A(x)$ is referred to as the *membership degree* of the item $x \in U$ to the A fuzzy set.

We can say that a fuzzy set A in U is characterized by a membership function $\mu_A(x)$ that associates each item $x \in U$ with a real number in the range [0, 1]; this value denotes the degree with which x is a member of A, also called "membership degree". Specifically, if $\mu_A(x) = 0$, then x does not belong at all to A; if $0 < \mu_A(x) < 1$, then x belongs to A only partially; if $\mu_A(x) = 1$, then x fully belongs to A.

As an example, consider the universe U of places in a given neighborhood. The fuzzy set named $NicePlaces$ denotes those places that are nice for a citizen named Julia. She judges a place x, based on her preferences, by means of membership degrees; clearly, $\mu_{NicePlaces}(x) = 1$ means that x is one of Julia's favourite places; given two places x_1 and x_2, if $\mu_{NicePlaces}(x_1) = 0.8$ and $\mu_{NicePlaces}(x_1) > \mu_{NicePlaces}(x_2)$, this means that x_1 has more favour than x_2 by Julia, even though x_1 is not fully favorite (its membership degree is 0.8).

Definition 11 Consider two fuzzy sets A and B in U.
Considering an item $x \in U$ and the intersection $I = A \cap B$, the membership degree of x to I is $\mu_I(x) = min(\mu_A(x), \mu_B(x))$.
Considering an item $x \in U$ and the union $S = A \cup B$, the membership degree of x to S is $\mu_S(x) = max(\mu_A(x), \mu_B(x))$.
Considering an item $x \in U$ and the complement of A, i.e., $C_A = U - A$, the membership degree of x to C_A is $\mu_{C_A}(x) = 1 - \mu_A(x)$.

A "Soft Condition" linguistically expresses the fact that an item x belongs to a specific fuzzy set. Consequently, the `AND` operator is mapped to the fuzzy intersection, the `OR` operator is mapped to the fuzzy union, the `NOT` operator is mapped to the fuzzy complement.

For example, considering again Julia's interests, we could think about a second fuzzy set named $CrowdedPlaces$, which denotes if a place is usually crowded; so, Julia could look for $InterestingPlaces$, such that
$InterestingPlaces = CrowdedPlaces$ `AND` $NicePlaces$, which is an example of soft condition. The resulting membership degree, that is $\mu_{InterestingPlaces} = min(\mu_{CrowdedPlaces}(x), \mu_{NicePlaces}(x))$ denotes the relevance of a place x with respect to the soft condition; places could be ranked in reverse order of membership degree.

In *J-CO-QL*$^+$, the representational units are *JSON* documents, so the universe U contains *JSON* documents: consequently, *J-CO-QL*$^+$ provides constructs to evaluate the membership degrees of *JSON* documents to fuzzy sets, so as to express "soft selection conditions" on *JSON* documents.

D. Details about JOIN in *GeoSoft*

Referring to Sect. 5.4, in this appendix we report details about spatial-join conditions and the `SET FUZZY SETS` clause.

D.1 Spatial JOIN

The GEOMETRY keyword in the ON clause of the JOIN operator is followed by the *spatialCond* spatial-join condition. Hereafter, we describe

- The spatial condition is evaluated on the two source geometries $\overline{f}_1$.geometry (i.e., left geometry) and $\overline{f}_2$.geometry (i.e., right geometry). The admitted predicates are: INTERSECT, which is true when the two geometries intersect; MEET, which is true when the two geometries share part of their boundaries; INCLUDE(LEFT) (respectively, INCLUDE(RIGHT)), which is true when the right (respectively, left) geometry contains the left (respectively, right) geometry. Another admitted predicate is "DISTANCE (*unit*) *comp number*"; it is true when the condition about the distance between the center of gravity of the two geometries is satisfied; *unit* can be M, KM or ML in order to evaluate the distance in meters, kilometers or miles, respectively); *comp* can be one of the usual comparison operators, i.e., $=$, $!=$, $<$, $<=$, $>$, $>=$; finally, *number* can be any real number.
- The SET GEOMETRY keyword is followed by *geometrySpec*, which specifies how to generate the geometry $\overline{f}_3$.geometry of the output $\overline{f}_3$. feature.
  Specifically, *geometrySpec* can be: INTERSECTION, to obtain the intersection of the source geometries $\overline{f}_1$.geometry and $\overline{f}_2$.geometry; UNION, to obtain the union of the source geometries; LEFT (respectively, RIGHT), to obtain the geometry $\overline{f}_1$.geometry of the left feature (respectively, the geometry $\overline{f}_2$.geometry of the right feature).

D.2. Setting Fuzzy Sets

The SET FUZZY SETS keywords are followed by the *fuzzySetSpec* specification, by which it is possible to specify the fuzzy sets of interest or derive new ones.

The *fuzzySetSpec* specification can be expressed in two different forms.

- The *synthetic form* is introduced by the KEEP keyword followed by three alternative keywords: they are LEFT, RIGHT or ALL. The semantics is the following:
  - with KEEP LEFT, the fuzzysets map of the left feature provides all membership degrees for $\overline{f}_3$, i.e., $\overline{f}_3$.fuzzysets = $\overline{f}_1$.fuzzysets;
  - with KEEP RIGHT, the fuzzysets map of the right feature provides all membership degrees for $\overline{f}_3$, i.e., $\overline{f}_3$.fuzzysets = $\overline{f}_2$.fuzzysets;
  - with KEEP ALL, both the fuzzysets maps of the left and right features provide the membership degrees for $\overline{f}_3$, i.e., $\overline{f}_3$.fuzzysets = $\overline{f}_1$.fuzzysets $\cup$ $\overline{f}_2$.fuzzysets.

  In the KEEP ALL case, the default behavior in the case the same fuzzy-set name *fsn* appears as a key in both $\overline{f}_1$.fuzzysets and $\overline{f}_2$.fuzzysets maps is to take the minimum membership degree, i.e., $\overline{f}_3$.fuzzysets(*fsn*) $=$ $min(\overline{f}_1$.fuzzysets(*fsn*), $\overline{f}_2$.fuzzysets(*fsn*)). However, it is possible to add the RESOLVING WITH option, to specify the behavior:

  - KEEP ALL RESOLVING WITH AND takes the minimum membership degree (this is the default behavior);
  - KEEP ALL RESOLVING WITH OR takes the maximum membership degree, i.e., $\overline{f}_3$.fuzzysets(*fsn*) $= max(\overline{f}_1$.fuzzysets(*fsn*), $\overline{f}_2$.fuzzysets(*fsn*));
  - KEEP ALL RESOLVING WITH FIRST takes the membership degree from the left feature, i.e., $\overline{f}_3$.fuzzysets(*fsn*) $=$ $\overline{f}_1$.fuzzysets(*fsn*);
  - KEEP ALL RESOLVING WITH LAST takes the membership degree from the right features, i.e., $\overline{f}_3$.fuzzysets(*fsn*) $=$ $\overline{f}_2$.fuzzysets(*fsn*).
- By means of the *detailed form*, it is possible to provide a comma-separated list that specifies, one by one, which membership degrees to hold in the $\overline{f}_3$.fuzzysets map. Each item in the list can be expressed in three different versions.
  - The first version of an item in the detailed form is *side* ALL, where *side* can be either LEFT or RIGHT. In this case, all membership degrees in $\overline{f}_1$.fuzzysets (respectively, in $\overline{f}_2$.fuzzysets) are added to $\overline{f}_3$.fuzzysets.
  - The second version of an item in the detailed form is *side fsn*, where *side* is either LEFT or RIGHT and *fsn* is a fuzzy-set name. If *side* is LEFT (respectively, RIGHT) the membership degree of *fsn* is taken from $\overline{f}_1$.fuzzysets (respectively, $\overline{f}_2$.fuzzysets) and inserted into $\overline{f}_3$.fuzzysets. Optionally, it is possible to specify a novel name for the fuzzy set to insert into $\overline{f}_3$.fuzzysets; in this case, the shape becomes *side fsn* AS *newFsn*, where *newFsn* is the name of the final fuzzy set.
  - The third version of an item in the detailed form is *spatialFunc* AS *newFsn*, where *spatialFunc* denotes a fuzzy spatial function that operates on the source geometries, while *newFsn* is the name of the new fuzzy set inserted into $\overline{f}_3$.fuzzysets. Fuzzy spatial functions provide a membership degree in the

range [0, 1] as a result, which becomes the membership degree of the novel *newFsn* fuzzy set.
Currently, the language provides the following fuzzy spatial functions:

- `HOW_INTERSECT()` provides the intersection degree (in the [0, 1] range) of the two input geometries;
- `HOW_INCLUDE(LEFT)` (respectively, `HOW_INCLUDE(RIGHT)`) provides the inclusion degree of $\overline{f}_1$`.geometry` (respectively, $\overline{f}_2$`.geometry`) within $\overline{f}_2$`.geometry` (respectively, $\overline{f}_1$`.geometry`);
- `HOW_MEET(LEFT)` (respectively, `HOW_MEET(RIGHT)`), provides the degree with which the border of the $\overline{f}_2$`.geometry` (respectively, $\overline{f}_1$`.geometry`) is shared with the border of the $\overline{f}_1$`.geometry` (respectively, $\overline{f}_2$`.geometry`).

In case of ambiguity (i.e., multiple membership degrees for the same fuzzy set) the minimum membership degree is considered; however, it is possible to exploit the `RESOLVING` specification at the end of the detailed form:

- `RESOLVING WITH AND` considers the minimum membership degree;
- `RESOLVING WITH OR` considers the maximum membership degree;
- `RESOLVING WITH FIRST` considers the first membership degree inserted into $\overline{f}_3$`.fuzzysets`;
- `RESOLVING WITH LAST` considers the last membership degree inserted into $\overline{f}_3$`.fuzzysets`.

Notice that both the *synthetic form* and the *detailed form* allows for dealing with complex situations, as far as membership degrees of joined features are concerned.

E. Example of Translation

To explain the translation strategy, it is worth presenting the translation of the *GeoSoft* query reported in Listing 7. For the sake of clarity, it is divided in several listings, each one corresponding to one specific section of the source query.

E.1. Execution Context

Listing 10 reports the execution context of the query. It is obtained by reading the content of the files specified in the `GET CONTEXT` directives.

Specifically, Listing 10 highlights the provenance of each single piece of the script.

- Line 1 comes from the `"jcoContextDb.jco"` file, whose content was shown in Listing 2; it defines the database to connect with.
- Line 2 comes from the `"jcoContextOp1.jco"` file, whose content was shown in Listing 4; it defines the `MediumLengthHighwayOp` fuzzy operator.
- Line 3 comes from the `"jcoContextOp2.jco"` file, whose content was shown in Listing 8; it defines the `RelevantPortionOp` fuzzy operator.
- Finally, line 3 comes from the `"jcoContextOp3.jco"` file, whose content was shown in Listing 9; it defines the `MediumTownOp` fuzzy operator.

First Nested Query

Listing 11 reports the fragment of the *J-CO-QL*$^+$ script corresponding to the translation of the first nested query in Listing 7 (enclosed in the blue box). In the right-hand side, Listing 11 reports the nested query from which it derives; colored arrows show the correspondence between *GeoSoft* clauses and *J-CO-QL*$^+$ instructions. Hereafter, we describe the script.

- The `FROM` clause in the nested query refers to the `towns` collection stored within the `geosoftDb` database. Consequently, the *J-CO-QL*$^+$ script must retrieve documents from this collection and extract features from within them, so as to obtain a collection of documents that represent soft features. Line 5 actually gets the content of the collection from the database.
- Line 6 actually extracts features from within the `features` array field in the *GeoJSON* document previously obtained from the database. The novel collection contains one *JSON* document for each feature, such that all root-level fields (except the `features` array) are preserved.
  In place of the `features` array field, the `feature` field is present: it is a nested document with two fields, i.e., the `item` field (which contains the original item) and the `position` field (which denotes the position occupied by the item in the source array).
- Line 7 makes geo-tagging compliant with the *J-CO-QL*$^+$ data model (see Sect. 6.1.1) and simplifies the structure of documents.
  Specifically, the `SETTING GEOMETRY` option says that the former geometry becomes the novel `~geometry` field. Then, the `BUILD` block simplifies the structure.
- While lines from 5 to 7 are necessary to acquire and prepare data, Line 8 actually corresponds to the remainder of the first nested query. Specifically, the `WHERE` clause of the *GeoSoft* query is added to the `WHERE` clause in the *J-CO-QL*$^+$ `FILTER` instruction. Similarly, the `USING` clause of the *GeoSoft* query becomes the (similar but not identical) `USING` clause in the *J-CO-QL*$^+$ `FILTER`

instruction. The result is that *JSON* documents now have the `~fuzzysets` field with the membership degree to the `MediumTown` fuzzy set.
The `SELECT` clause in the first nested *GeoSoft* query has a twofold effect: firstly, it generates an extra condition in the `WHERE` clause in the *J-CO-QL*$^+$ `FILTER` instruction (to select documents having the properties that are listed in the `SELECT` clause), and then is translated into the `BUILD` block in the *J-CO-QL*$^+$ `FILTER` instruction, so as to project on properties of interest (notice the path expression with the "`.properties`" prefix).
- Finally, since the nested query is involved in a `JOIN` operation, line 9 saves the temporary collection into the Intermediate Results *IR* database, with name (automatically generated) `GeoSoftIntermediate_0_t`.

E.2 Second Nested Query

Listing 12 reports the translation of the second nested query (depicted within the green box) in Listing 7. Again, in the right-hand side the nested query from which it derives is reported, with arrows that show the correspondence between *GeoSoft* clauses and *J-CO-QL*$^+$ instructions. In the following, we shortly explain it.

- Since the second nested *GeoSoft* query still operates on a database collection, its translation is similar to what we showed for the first nested *GeoSoft* query in Sect. 1.
Specifically, lines 10–12 acquire the content of the `highways` database collection (line 10), unnest features into single *JSON* documents (line 11) and align geometries to the *J-CO-QL*$^+$ data model (line 12).
- The `FILTER` instruction on line 13 performs the same work made by line 8 (Listing 11), i.e., translates the nested *GeoSoft* query.
Specifically, the `USING` clause in the second nested *GeoSoft* query corresponds to the `USING` clause in line 13. Notice that the membership degrees to two fuzzy sets are evaluated, i.e., `MediumLengthHighways` and `NotMediumLengthHighways`.
Finally, the `SELECT` clause in the second nested *GeoSoft* query affects the *J-CO-QL*$^+$ `WHERE` clause and then it is translated into the `BUILD` block.
- Line 14 saves the temporary collection (that contains features selected by the second nested *GeoSoft* query) into the *IR* database, because the query is nested within a `JOIN` expression.

Main Query

Listing 13 reports the translation of the main (outermost) *GeoSoft* query reported in Listing 7. Again, the right-hand side reports the source *GeoSoft* query and arrows denotes the correspondence between *GeoSoft* clauses and *J-CO-QL*$^+$ instructions. Hereafter, we explain the script.

- Soft features to be joined are stored within the *IR* database: two collections named `GeoSoftIntermediate_0_t` and `GeoSoftIntermediate_1_h` were saved by the two scripts in Listing 11 and in Listing 12, respectively; they contain documents corresponding to features processed by the two nested *GeoSoft* queries.
- The `JOIN OF COLLECTIONS` instruction on line 15 joins documents in the two source collections, respectively aliased as `t` and `h` (these are the same aliases used in the source *GeoSoft* query in Listing 7).
The instruction makes a pair *d* of two documents *t* (from the `t` collection) and *h* (from the `h` collection). The novel *d* *JSON* document is generated, with two fields: the `t` field contains the source *t* document, while the `h` field contains the source *h* document.
- The `ON GEOMETRY` clause in the source *GeoSoft* query (Listing 7) is reported in the corresponding `ON GEOMETRY` clause in line 15. Here, it specifies that a pair of documents is considered if their geometries satisfy the spatial condition it defines; specifically, the spatial condition asks for a pair of documents whose geometries intersect.
The `SET GEOMETRY` clause from the *GeoSoft* query in Listing 7 is reported in line 15 too. Here, it specifies how to generate the `~geometry` field of the output *d* document. Specifically, the clause specifies that the intersection of the source `~geometry` fields becomes the value of the `~geometry` field in *d*.
- The `ADD FIELDS` clause adds novel fields to the *d* document, so as they can be used later.
Specifically, a novel `properties` field is generated, which in turn contains the `t` and the `h` fields: they contain the nested `properties` field coming from the source documents. This way, path expressions that are present in the *GeoSoft* query can be directly used, without any translation, by simply adding the "`.properties`" prefix.
- The `SET FUZZY SETS` clause in the *GeoSoft* `JOIN` expression (see Listing 7) exploits the same syntax as in *J-CO-QL*$^+$. This way, it can be copied as it is into the `JOIN OF COLLECTIONS` instruction. In line 15 (Listing 13), the `SET FUZZY SETS` sub-clause is located before the `CASE WHERE` clause, because it ends the generation of *d* documents after paring source documents coming from the source collections.
Specifically, the clause adds the membership degrees to the `MediumLengthHighways` fuzzy set (from the *h*, i.e., right, source document), to the `MediumTowns` fuzzy set (from the *t*, i.e., left, source document) and

to the `NotMediumLengthHighways` fuzzy set (from the *h*, i.e., right, source document).

Documents *d* that are internally generated by the `JOIN OF COLLECTIONS` instruction before the `CASE WHERE` clause represent joined features, as specified by the *GeoSoft* `JOIN` expression. The remaining clauses in the outermost *GeoSoft* query are translated into the corresponding `CASE WHERE` clause of the *J-CO-QL*$^+$ `JOIN OF COLLECTIONS` instruction, as described hereafter.

- The *GeoSoft* `WHERE` clause is translated into the *J-CO-QL*$^+$ `WHERE` condition. The approach is identical to what is done for nested *GeoSoft* queries.
- The *GeoSoft* `USING` clause is translated into the *J-CO-QL*$^+$ `CHECK FOR` clause, as for nested *GeoSoft* queries. The same holds for the `APHACUT` clause.
- The *GeoSoft* `SELECT` clause affects the `WHERE` clause and, then, it is translated into the `BUILD` block.

The reader can see that the translation schema is the same as the one adopted for nested *GeoSoft* queries without `JOIN` expression in the `FROM` clause. Indeed, both the `FILTER` and the `JOIN OF COLLECTIONS` statements relies on the `CASE WHERE` clause, with identical semantics. This way, a uniform translation pattern can be adopted.

E.4. Tail

Finally, it is necessary to rebuild the crisp *GeoJSON* document to generate and save it into a database collection, as specified by the *GeoSoft* `SAVE AS` clause. The *J-CO-QL*$^+$ script reported in Listing 14 performs this task. It is shortly illustrated hereafter.

- The goal of line 16 is to generate crisp features which comply with the *GeoJSON* specification. To achieve this, the `BUILD` block adds the `geometry` field, whose value is extracted from within the `~geometry` field by means of the `GEOMETRY_FIELD` built-in function. Notice the `key` field, whose goal is to provide a grouping key to the next instruction.
  Then, the `DEFUZZIFY` option drops the `~fuzzysets` field, while the `DROPPING GEOMETRY` option drops the `~geometry` field.
- Line 17 groups documents into one single *GeoJSON* document, by exploiting the `key` field (which is automatically dropped by the instruction). The `BUILD` block provides the final structure to the document.
- Finally, line 18 saves the *GeoJSON* document into the `highwayTowns` collection in the `geosoftDb` database.

Notice that the tail is substantially constant: the only thing that could change is the presence of the `name` field within the `BUILD` block in line 17, in the case the `SETTING NAME` option is specified in the *GeoSoft* `SAVE AS` clause (as in Listing 7).

References

1. Bray, T.: The javascript object notation (json) data interchange format. URL https://www.rfc-editor.org/rfc/rfc7159.txt (2014). Accessed 11 Sep 2023
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (XML) 1.0. W3C recommendation October (2000)
3. Butler, H., Daly, M., Doyle, A., Gillies, S., Hagen, S., Schaub, T., et al.: The geojson format. In: Internet Engineering Task Force (IETF) (2016)
4. Slater, J.A., Malys, S.: Wgs 84-past, present and future. In: Brunner, F.K. (eds.) Advances in Positioning and Reference Frames, pp. 1–7. Springer, Berlin, Germany (1998)
5. Cattell, R.: Scalable sql and nosql data stores. ACM SIGMOD Rec. **39**(4), 12–27 (2011)
6. Arora, R., Aggarwal, R.R.: Modeling and querying data in mongodb. Int. J. Sci. Eng. Res. **4**(7), 141–144 (2013)
7. Psaila, G., Fosci, P.: J-CO: a platform-independent framework for managing geo-referenced json data sets. Electronics **10**(5), 621 (2021)
8. Fosci, P., Psaila, G.: Towards flexible retrieval, integration and analysis of json data sets through fuzzy sets: a case study. Information **12**(7), 258 (2021)
9. Fosci, P., Marrara, S., Psaila, G.: Soft querying geojson documents within the J-CO Framework. In: 16th International Conference on Web Information Systems and Technologies (WEBIST 2020), pp. 253–265 (2020). SCITEPRESS–Science and Technology Publications, Lda
10. Fosci, P., Marrara, S., Psaila, G.: Geosoft: A language for soft querying features within geojson information layers. In: International Conference on Web Information Systems and Technologies. Springer International Publishing Cham, pp. 196–219 (2020)
11. Zadeh, L.A.: Fuzzy sets. Inf. Control **8**(3), 338–353 (1965)
12. Blair, D.C.: Information retrieval, 2nd ed. c.j. van rijsbergen. London: Butterworths; 1979: 208 pp. price: $ 3250. J. Am. Soc. Inform. Sci. 30(6), 374–375 (1979). https://doi.org/10.1002/asi.4630300621
13. Fuhr, N.: Models for retrieval with probabilistic indexing. Inform. Process. Manag. **25**(1), 55–72 (1989)
14. Zadeh, L.A.: The concept of a linguistic variable and its application to approximate reasoning-i. Inf. Sci. **8**(3), 199–249 (1975)
15. Buell, D.A.: A problem in information retrieval with fuzzy sets. J. Am. Soc. Inform. Sci. (pre-1986) **36**(6), 398 (1985)
16. Bosc, P., Pivert, O.: Sqlf: a relational database language for fuzzy querying. IEEE Trans. Fuzzy Syst. **3**(1), 1–17 (1995)
17. Bosc, P., Pivert, O.: Sqlf query functionality on top of a regular relational database management system. In: Pons, Olga., Vila, M.A., Kacprzyk, J. (eds.) Knowledge Management in Fuzzy Databases. Springer, Berlin, Germany, pp. 171–190 (2000)
18. Galindo, J., Medina, J.M., Pons, O., Cubero, J.C.: A server for fuzzy sql queries. In: International Conference on Flexible Query Answering Systems, pp. 164–174 (1998). Springer
19. Kacprzyk, J., Zadrożny, S.: Fquery for access: fuzzy querying for a windows-based dbms. Bosc P., Kacprzyk J. (eds), Fuzziness in database management systems. Stud. Fuzz. **5** (1995)

20. Zadrozny, S., Kacprzyk, J.: Fquery for access: towards human consistent querying user interface. In: Proceedings of the 1996 ACM Symposium on Applied Computing, pp. 532–536 (1996)
21. Bordogna, G., Psaila, G.: Modeling soft conditions with unequal importance in fuzzy databases based on the vector p-norm. In: Proceedings of the IPMU, Malaga (2008)
22. Bordogna, G., Psaila, G.: Soft aggregation in flexible databases querying based on the vector p-norm. Internat. J. Uncertain. Fuzz. Knowl.-Based Syst. **17**(supp01), 25–40 (2009)
23. Bordogna, G., Psaila, G.: Chap. 8. Customizable flexible querying in classical relational databases. In: Galindo, J. (eds.) Handbook of Research on Fuzzy Information Processing in Databases. IGI Global, Hershey, pp. 191–217 (2008)
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: Motro, A. (eds.) Uncertainty Management in Information Systems. Springer, Berlin, pp. 285–324 (1997)
25. Medina, J.M., Pons, O., Vila, M.A.: Gefred: a generalized model of fuzzy relational databases. Inf. Sci. **76**(1), 87–109 (1994). https://doi.org/10.1016/0020-0255(94)90069-8
26. Galindo, J., Urrutia, A., Piattini, M.: Fuzzy Databases: Modeling, Design, and Implementation. IGI Global, Hershey (2006)
27. Galindo, J.: New characteristics in fsql, a fuzzy sql for fuzzy databases. WSEAS Trans. Inf. Sci. Appl. **2**(2), 161–169 (2005)
28. Smits, G., Pivert, O., Girault, T.: Reqflex: fuzzy queries for everyone. Proc. VLDB Endow. **6**(12), 1206–1209 (2013)
29. Chris Anderson, J., Jan Lehnardt, N.S.: CouchDB: The Definitive Guide. O'Reilly Media, Inc., Sebastopol (2010)
30. Nayak, A., Poriya, A., Poojary, D.: Article: Type of nosql databases and its comparison with relational databases. Int. J. Appl. Inform. Syst. **5**(4), 16–19 (2013)
31. 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:abs/1405.3631 [CoRR] (2014)
32. Florescu, D., Fourny, G.: Jsoniq: the history of a query language. IEEE Internet Comput. **17**(5), 86–90 (2013)
33. Abir, B.K., Amel, G.T.: Towards fuzzy querying of nosql document-oriented databases. DBKDA **2015**, 163 (2015)
34. Medina, J.M., Blanco, I.J., Pons, O.: A fuzzy database engine for mongodb. Int. J. Intell. Syst. 37, 5691–5724 (2022)
35. Fosci, P., Psaila, G.: Powering soft querying in J-CO-QL with javascript functions. In: International Workshop on Soft Computing Models in Industrial and Environmental Applications. Springer, Cham, pp. 207–221 (2021)
36. Fosci, P., Psaila, G.: J-CO, a framework for fuzzy querying collections of json documents. In: International Conference on Flexible Query Answering Systems. Springer, Cham, pp. 142–153 (2021)
37. Aufaure, M.-A., Trépied, C.: Workshops in Computing. In: Kennedy, J. B., Barclay, P. J. (eds.) A Survey of Query Languages for Geographic Information Systems, 3. Springer (1996). https://dblp.org/rec/conf/ids/Aufaure-PortierT96.bib
38. Costagliola, G., Tortora, G., Tucci, M., Busillo, M.: Querying by Content. In: Spaccapietra, S., Jain, R. (eds.) Visual Database Systems 3: Visual information management. Springer, Boston, pp. 275–286 (1995)
39. Egenhofer, M.J.: Spatial sql: a query and presentation language. IEEE Trans. Knowl. Data Eng. **6**(1), 86–95 (1994)
40. Jacobs, B.E., Walczak, C.A.: A generalized query-by-example data manipulation language based on database logic. IEEE Trans. Softw. Eng. **1**, 40–57 (1983)
41. Staes, F., Tarantino, L., Tiems, A.: A graphical query language for object oriented databases. In: Proceedings 1991 IEEE Workshop on Visual Languages, pp. 205–210 (1991)
42. Kim, H.-J., Korth, H.F., Silberschatz, A.: Picasso: a graphical query language. Softw. Pract. Exp. **18**(3), 169–203 (1988)
43. Mayer, B.: Beyond icons : Towards new metaphors for visual query languages for spatial information systems. In: Proceedings of the First International Workshop on Interfaces to Database Systems. R. Cooper, Springer-Verlag, Glasgow, UK, Workshops in Computing, pp. 113–135 (1992). ISBN 978-3-540-19802-4
44. Cai, G.: Geovsm: An integrated retrieval model for geographic information. In: International Conference on Geographic Information Science. Springer, pp. 65–79 (2002)
45. Bieber, M., Kacmar, C.: Designing hypertext support for computational applications. Commun. ACM **38**(8), 99–107 (1995)
46. Guo, D., Onstein, E.: State-of-the-art geospatial information processing in nosql databases. ISPRS Int. J. Geo Inf. **9**(5), 331 (2020)
47. Formica, A., Mazzei, M., Pourabbas, E., Rafanelli, M.: Querying Distributed GIS with GeoPQLJ based on GeoJSON. In: Proceedings of the 5th International Conference on Geographical Information Systems Theory, Applications and Management, GISTAM 2019, Heraklion, Crete, Greece, May 3–5, 2019. SciTePress, pp. 175–182 (2019). https://doi.org/10.5220/0007657201750182
48. Bordogna, G., Psaila, G.: Fuzzy-spatial sql. In: International Conference on Flexible Query Answering Systems. Springer, Berlin, Heidelberg, pp. 307–319 (2004)
49. 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-spatial Inform. Sci. **21**(3), 257–271 (2018)
50. George, S.: Nosql–not only sql. Int. J. Enterp. Comput. Bus. Syst. 2(2) (2013). https://ijecbs.com/manuscripts/index.php/vol-3-issue-2-july-2013
51. Chodorow, K.: MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly Media, Inc., Sebastopol (2013)
52. Bringas, P.G., Pastor, I., Psaila, G.: Can blockchain technology provide information systems with trusted database? the case of hyperledger fabric. In: International Conference on Flexible Query Answering Systems. Springer, Cham, pp. 265–277 (2019)
53. Gormley, C., Tong, Z.: Elasticsearch: The Definitive Guide: a Distributed Real-time Search and Analytics Engine. O'Reilly Media, Inc., Sebastopol (2015)
54. Bordogna, G., Capelli, S., Psaila, G.: A big geo data query framework to correlate open data with social network geotagged posts. In: The Annual International Conference on Geographic Information Science. Springer, pp. 185–203 (2017)
55. 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. ACM, pp. 499–508 (2017)
56. Fosci, P., Psaila, G.: Soft integration of geo-tagged data sets in J-CO-QL+. ISPRS Int. J. Geo Inf. **11**(9), 484 (2022)
57. Fosci, P., Psaila, G.: Soft spatial querying on json data sets. In: European Conference on Advances in Databases and Information Systems. Springer International Publishing Cham, pp. 223–237 (2022)
58. Fosci, P., Psaila, G.: Soft querying powered by user-defined functions in j-co-ql+. Neurocomputing **546**, 126311 (2023)
59. Psaila, G., Fosci, P.: Toward an anayist-oriented polystore framework for processing json geo-data. In: International Conferences on WWW/Internet, ICWI 2018 and Applied Computing 2018, Budapest; Hungary, 21-23 October 2018. IADIS (International Association for Development of the Information Society), pp. 213–222 (2018)
60. Fosci, P., Marrara, S., Psaila, G.: Towards soft web intelligence by collecting and. In: 18th International Conference on Web Information Systems and Technologies (WEBIST 2022), pp. 0–0 (2022). SCITEPRESS–Science and Technology Publications, Lda

61. 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. Mob. Inform. Syst. (2018)
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, Lecture Notes in Computer Science, 9344, pp. 285–294 (2015)
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. ACM, pp. 62–69 (2016)
64. Bordogna, G., Frigerio, L., Cuzzocrea, A., Psaila, G.: Clustering geo-tagged tweets for advanced big data analytics. In: 2016 IEEE International Congress on Big Data (BigData Congress). IEEE, pp. 42–51 (2016)
65. 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. **10**(3–4), 246–268 (2017)
66. Fosci, P., Psaila, G.: Towards soft web intelligence by collecting and processing json data sets from web sources. In: Proceedings of the 18th International Conference on Web Information Systems and Technologies (2022)
67. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: Json-ld 1.0. W3C recommendation 16, 41 (2014). https://www.w3.org/TR/2014/REC-json-ld-20140116/. Accessed 11 Sep 2023
68. Manola, F., Miller, E., McBride, B.: Rdf primer. w3c recommendation (2004). http://www.w3.org/TR/rdf-primer (2004). Accessed 11 Sep 2023
69. Fosci, P., Psaila, G.: Finding the best source of information by means of a socially-enabled search engine. In: KES 2012-Conference on in Knowledge-Based and Intelligent Information and Engineering Systems, vol. 243. IOS Press, pp. 1253–1262 (2012)
70. Fosci, P., Psaila, G.: Toward a product search engine based on user reviews. In: DATA, pp. 223–228 (2012)