Geo-spatial Information Science

ISSN: 1009-5020 (Print) 1993-5153 (Online) Journal homepage: http://www.tandfonline.com/loi/tgsi20

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

Gloria Bordogna, Steven Capelli, Daniele E. Ciriello & Giuseppe Psaila

To cite this article: Gloria Bordogna, Steven Capelli, Daniele E. Ciriello & Giuseppe Psaila (2017): A cross-analysis framework for multi-source volunteered, crowdsourced, and authoritative geographic information: the case study of volunteered personal traces analysis against transport network data, Geo-spatial Information Science, DOI: 10.1080/10095020.2017.1374703

To link to this article: https://doi.org/10.1080/10095020.2017.1374703

Published online: 24 Oct 2017.

GEO-SPATIAL INFORMATION SCIENCE, 2017
https://doi.org/10.1080/10095020.2017.1374703

Taylor & Francis
Taylor & Francis Group

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

Gloria Bordogna[a], Steven Capelli[b], Daniele E. Ciriello[c] and Giuseppe Psaila[c]

[a]CNR IREA, Milano, Italy; [b]DISCo, Universitá degli studi di Milano Bicocca, Sesto San Giovanni, Italy; [c]DIGIP, University of Bergamo, Dalmine, Italy

ABSTRACT

The paper discusses the need of a high-level query language to allow analysts, geographers and, in general, non-programmers to easily cross-analyze multi-source *VGI* created by means of apps, crowd-sourced data from social networks and authoritative geo-referenced data, usually represented as *JSON* data sets (nowadays, the de facto standard for data exported by social networks). Since an easy to use high-level language for querying and manipulating collections of possibly geo-tagged *JSON* objects is still unavailable, we propose a truly declarative language, named *J-CO-QL*, that is based on a well-defined execution model. A plug-in for a *GIS* permits to visualize geo-tagged data sets stored in a *NoSQL* database such as *MongoDB*; furthermore, the same plug-in can be used to write and execute *J-CO-QL* queries on those databases. The paper introduces the language by exemplifying its operators within a real study case, the aim of which is to understand the mobility of people in the neighborhood of Bergamo city. Cross-analysis of data about transportation networks and *VGI* from travelers is performed, by means of *J-CO-QL* language, capable to manipulate and transform, combine and join possibly geo-tagged *JSON* objects, in order to produce new possibly geo-tagged *JSON* objects satisfying users' needs.

ARTICLE HISTORY
Received 9 June 2017
Accepted 1 September 2017

KEYWORDS
Cross-analysis framework; comparing VGI; crowd-sourced and authoritative geographical data; *JSON* data-sets; declarative query language; heterogeneous data-sets

1. Introduction

Volunteered Geographic Information (*VGI*), created by means of smart applications installed on mobile devices connected to the Internet, and geo-tagged crowd-sourced information, created within social networks, also named user-generated geo-tagged contents, are attracting more and more the interest of both scientific and business companies. Let's think about the many citizen science projects in which *VGI* is created by amateurs eager to contribute to science, by describing observations of entities of interests in situ. For example, in the projects named *iNaturalist*[1] and *eBird*[2] (see Fitzpatrick et al. 2002), flora and fauna observations are collected to study climate changes in relation to the geographic distribution of species' habitats. Such projects could not be carried out without cross-analyzing the distribution of *VGI* all over the world and information about geographic areas recognized to be the habitats of species.

There exist many other application fields for cross-analysis of *VGI*, crowd-sourced and authoritative data. An example could be understanding how to improve public transportations, based on actual users needs, as well as understanding where to open new restaurants, discovering the areas of a city mostly visited by tourists, etc.

To ease such a kind of cross-analysis, a novel framework that permits to overlay, compare and correlate geo-tagged information from volunteers, crowd-sourced and authoritative information within the same digital environment is needed by analysts. Furthermore, the everyday increasing adoption of non-relational models for representing data coming from social networks and *VGI* apps, as well as for authoritative information published in open data portals, significantly stresses the need for flexible tools to cross-analyze heterogeneous collections. In particular, *JSON* is now the de facto standard for information representation and exchange.

Current *GIS*s, are inadequate for this purpose. They are suitable for spatially visualizing and manipulating classical geographic data layers in standard vectorial and raster formats; now, they are also able to load and export geographic information in *GeoJSON* format, which relies on *JSON* syntax. Nevertheless, they provide limited facilities to manipulate *GeoJSON* objects, and, more important, they do not provide a declarative query language to manipulate *GeoJSON* entities. For example, *QGIS*, the open source *GIS* software, cannot visualize *GeoJSON* objects with geometries of distinct types at once: different geometry types must be

CONTACT Giuseppe Psaila giuseppe.psaila@unibg.it

in different layers (this way, the same entity may appear in different layers). Furthermore, if *GeoJSON* is attractive for its capability of representing entities with very complex geometry (possibly mixing points, polylines and polygons), it is still focused on geometric entities, instead of representing geo-tagged *JSON* objects, i.e. objects enriched with a geographic tag.

Based on such premises, the goal of our research work is to devise *J-CO*, a framework for performing complex transformations and aggregations of collections of pure *JSON* objects possibly geo-tagged with complex geometry, originated by multiple sources. The core of the framework is a declarative query language named *J-CO-QL*, in the style of *SQL* spatial, for creating, modifying, selecting, merging, and cross-analyzing collections of possibly geo-tagged *JSON* objects. We implemented a plug-in for *QGIS* to visualize geo-tagged *JSON* objects together with other geographical layers, as well as to issue *J-CO-QL* queries to the query engine. Collections of *JSON* objects are stored in databases managed by *MongoDB*, one of the most famous *NoSQL DBMS*s.

The paper will rely on a case study (described in Section 1.1) based on our experience in a real project. Then, after the related work is discussed in Section 2 and a preliminary description of the *J-CO* framework is reported in Section 3, the *J-CO-QL* query language will be introduced while describing the cross-analysis tasks conducted on the data-sets of our case study in Sections 4, 5 and 6. Finally, Section 7 draws the conclusions.

1.1. Case study: people mobility in Bergamo Neighborhood

The goal of this paper is to show that, by designing a consistent framework for the definition of declarative operators and their chained evaluation, it is possible to compose queries specifying complex manipulations. To this aim, we make use of a case study originated from a research project whose final goal is to understand how workers, students and tourists move in the city of Bergamo. We are exploiting a collaborative and social approach: volunteers (in our context students and workers) participate to the research supported by an app for mobile devices named *Moves*.[3] This app continuously traces registered people, characterizing the type of movements performed by them (on foot, by car or other transport means). We ask volunteers to register to our web site, so that we can automatically get their traces from *Moves API*s.

In order to understand how many people use trains to get to the city, besides *VGI* of travelers we collected information about rails, provided by official authoritative sources. All these data-sets are represented as collections of *JSON* objects and stored within a *NoSQL MongoDB* database. This choice is motivated by the fact that these data-sets are heterogeneous in terms of structure, with distinct levels of nesting, so traditional relational technology is not suitable for them.

The challenges that we have to face in our case study are manifold.

- We have to cross-analyze traces of people with railway lines, in order to detect the number of volunteers that travel through public transport lines.
- We have to provide tools for performing such an analysis that do not require to write computer programs, but specifying high-level queries to manipulate, select and filter the data-sets.

The reader will see that this case study is an interesting example of how the proposed framework could be effectively applied to study mobility and transportation in cities. In particular, by considering not only trains, but also buses and underground lines, the analysis becomes more complex, but the approach hereafter described will still work.

2. Related work

J-CO-QL moves from our previous work on the problem of querying heterogeneous collections of complex spatial data (see Bordogna, Pagani, and Psaila 2006; Psaila 2011). In that works, we proposed a database model capable to deal with heterogeneous collections of possibly nested spatial objects, based on the composition of more primitive spatial objects; at the same time, a relational algebra to query complex spatial data is provided. With regard to those works, *J-CO-QL* relies on the *JSON* standard and does not require an *ad-hoc* data model; furthermore, *J-CO-QL* abandons the relational algebra syntax, in favor on a more flexible and intuitive execution model.

The adoption of *NoSQL* databases is motivated by the need of flexibility, as far as data structures are concerned. An interesting survey about *NoSQL* databases can be found in Han et al. (2011), where several systems are cataloged and classified. In particular, a DBMS like *MongoDB* falls into the category of *document databases*, because collections of *JSON* objects are generically considered as *documents*. Consequently, the query language provided by such a system does not allow complex and multi-collection transformations like those provided by *J-CO-QL* (see the web sites reported in the footnote[4] for details). Readers interested in *NoSQL DBMS*s evaluation can refer to Robin and Jablonski (2011) and to Cattell (2011).

As far as query languages for *JSON* objects are concerned, several proposals were made. However, none of them is explicitly designed to provide geographical data analysis capabilities, natively integrated in a high-level query language, as for *J-CO-QL*. Hereafter, we shortly refer to them.

Jaql (see Nayak, Poriya, and Poojary 2013) was designed to help *Hadoop* (see White 2012) programmers writing complex transformations, avoiding low-level programming, to perform in a cloud and parallel environment. Flexibility and physical independence are the main goals of *Jaql*: in particular, its execution model is similar to our execution model, since it explicitly relies on the concept of pipe; in fact, the pipe operator is explicitly used in *Jaql* queries. However, it is still oriented to programmers, while *J-CO-QL* constructs are at a higher level and truly declarative.

An interesting proposal is *SQL++* that has been defined to query both *JSON* native stores and *SQL* databases. The *SQL++* semi-structured data model is a superset of both *JSON* and the relational data model. Yet, *SQL++* is *SQL* backwards compatible and is generalized towards *JSON* by introducing only a small number of query language extensions (Ong, Papakonstantinou, and Vernoux 2014). In *SQL++*, the classical SELECT statement of *SQL* is adapted and extended to perform queries on collections of *JSON* objects. In our opinion, this is a clean proposal, if compared with others, that tries to work at a higher abstraction level. However, it does not deal explicitly with heterogeneity of objects. Furthermore, complex transformations that require several queries sequentially executed would need to explicitly save intermediate results into the persistent database. In contrast, the execution model on which *J-CO-QL* relies clearly separate persistent databases and temporary databases.

Also the industry is looking at the extension of *SQL* to query *JSON* objects. An example is *N1QL*[5] that is a declarative language extending *SQL* for *JSON* objects stored in *NoSQL* databases, specifically implemented for *Couchbase 4.0*, in order to handle semi-structured, nested data. It enables querying *JSON* documents without any limitations, sort, filter, transform, group, and combine data with a single query from multiple documents with a **`JOIN`**. Nevertheless it does not provide operators to manipulate *GeoJSON* objects. Finally, other declarative languages for *JSON* objects have been defined as extensions of structured languages for semi-structured documents, such as *JSONiq*[6] that borrowed a large numbers of ideas from *XQuery*, like the functional aspect of the language, the semantics of comparisons in the face of data heterogeneity, the declarative, snapshot-based updates. Nevertheless, like *XQuery* it can be hardly used by unexperienced users.

3. *J-CO* framework

As previously introduced, the *J-CO* framework (which stands for *J*SON *CO*llections) has been devised to enable non programmers to perform complex geo-spatial queries on geo-tagged *JSON* data-sets stored within a *MongoDB* database; furthermore, we wanted users to be able to visualize such data-sets within classical *GIS* environments. Figure 1 depicts the high-level components in the *J-CO* framework, and hereafter, we describe them.

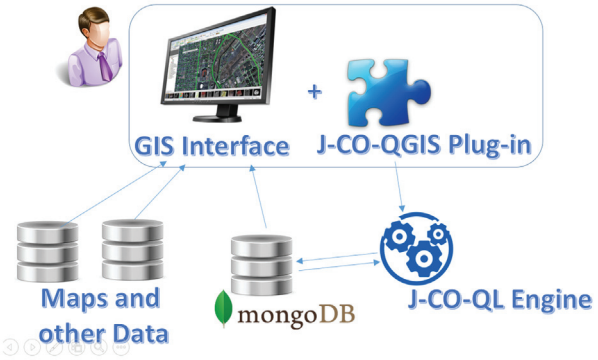


Figure 1. Components of the J-CO Framework.

- The framework is founded on a *Data Model* and an *Execution Model*, that make possible to define a *query language*, that we called *J-CO-QL*. This novel query language is specifically designed to specify complex geo-spatial analysis tasks on heterogeneous collections of *JSON* objects.
- The *J-CO-QL Engine* executes queries on the collections stored within *MongoDB* databases and stores results into (possibly different) *MongoDB* databases.
- The *J-CO-QGIS Plug-in* provides *QGIS* (the free and widely used *GIS* tool) users with the possibility to explore collections in *MongoDB* databases and show geo-tagged *JSON* objects and their geometries as spatial layers, so that they can be overlaid with other information layers.

3.1. Data and execution models

3.1.1. Data model

The data model is founded on the basic concept of *JSON* object. *JSON* (JavaScript Object Notation) is a de facto standard serialized representation for objects. Fields (object properties) can be simple (numbers or strings), complex (i.e. nested objects), arrays (of numbers, strings, objects).

As far as spatial representation is concerned (i.e. geo-tagging of *JSON* objects), we rely on the *GeoJSON* standard (see Butler et al. 2016). In particular, we assume that the geometry is described by a field named `~geometry`, defined as a *GeometryCollection* object type in *GeoJSON* standard. The absence of this top-level field means that the object does not have an explicit geo-tag. *J-CO-QL* provides constructs to add geo-tags to *JSON* objects.

The following definition formalizes the concepts of *Collection* and *Database*.

Definition 1 (Collections and Databases): A *Database* db is a set of collections: $db = \{c_1, \ldots, c_n\}$. Each

collection c has a name $c.name$ (unique in the database) and an instance $Instance(c) = [o_1, \ldots, o_m]$ that is a vector of *JSON* objects o_i.

3.1.2. Execution model

Queries will transform collections stored in databases managed by *MongoDB* and will generate new collections that will be stored again into *MongoDB* databases, for persistence. For simplicity, we call such databases as *Persistent Databases*

Definition 2 (Query Process State): A *state* s of a query process is a pair $s = (tc, IR)$, where tc is a collection named *Temporary Collection*. while IR is a database named *Intermediate Result database*.

Definition 3 (Operator Application): Consider an operator op. Depending on the operator, it is parametric w.r.t. input collections (present in the persistent databases or in IR) and, possibly, w.r.t. an output collection, that can be saved either in the persistent databases or in IR.

The application of an operator op, denoted as $\overline{op}$, is defined as:

$$\overline{op} : s \rightarrow s'$$

where both domain and co-domain are the universe of query process states. The operator application takes a state s as input, possibly works on the temporary collection $s.tc$, possibly takes some intermediate collections stored in $s.IR$; then, it generates a new query process state s', with a possibly new temporary collection $s'.tc$ and a possibly new version of the intermediate result database $s'.IR$.

The idea is that the application of an operator starts from a given query process state and generates a new query process state. The *temporary collection tc* is the result of the operator; alternatively, the operator could save a collection as *intermediate result* into the IR database, that could be taken as input by a subsequent operator application.

Definition 4 (Query): A query q is a non-empty sequence of operator applications, i.e. $q = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle$, with $n \geq 1$.

Each operator application starts from a given query process state and generates a new query process state, as defined by the following definition.

Definition 5 (Query Process): Given a query $q = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle$, a query process QP is a sequence of query process states $QP = \langle s_0, s_1, \ldots, s_n \rangle$, such that $s_0 = (tc : [], IR : \emptyset)$ and, for each $1 \leq i \leq n$, it is $\overline{op}_i : s_{i-1} \rightarrow s_i$.

The query process starts from the empty temporary collection $s_0.tc$ and the empty intermediate result database $s_0.IR$. Thus, *J-CO-QL* provides operators (named *start operators*) able to start the computation, taking collections from the persistent databases, while other operators (named *carry on operators*) carry on the process, continuously transforming the temporary collection and possibly saving it into the persistent databases or, for temporary results, into the intermediate result database IR. This way, new sub-tasks can be started, by taking collections both from persistent databases and from IR.

3.1.3. Innovative features

The intermediate result database IR is motivated by the need to support complex transformation processes that typically proceed through the computation of several intermediate results. In fact, it would be inappropriate to store them into the persistent databases that should store source and target collections. Furthermore, by means of IR, intermediate collections are clearly stated to be intermediate, and disappear from the system at the end of the process. Nevertheless, IR is implicitly related to each single execution process, that is automatically managed in isolation, w.r.t. other execution processes.

The procedural flavor is the result of the applications of operators: they are declarative (see next sections) but their application defines a process. Anyway, queries are not programs, in the sense of procedural programming languages: they are processes.

3.2. J-CO-QL query language and its execution engine

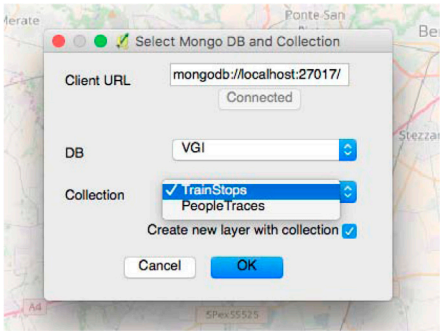
The key components of the *J-CO* framework are the query language, named *J-CO-QL*, and its execution engine.

Typical limitations of other query languages for *JSON* collections are the inability of dealing with heterogeneous objects in the same collection at the same time (typically, several queries must be written and then their results united together). *J-CO-QL* provides operators specifically designed to deal with objects with different structure within the same operator application.

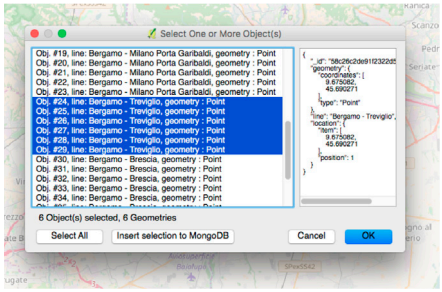
A second typical limitation of other languages for querying *JSON* data is that they are thought for programmers, or at least for people having programming skills. In contrast, operators provided by *J-CO-QL* are high-level operators, based on a declarative approach.

Finally, *J-CO-QL* directly deals with spatial representations possibly contained in *JSON* objects, because its data model explicitly consider them. Other languages completely miss support for managing spatial representation. In the rest of the paper, we will present relevant operators of *J-CO-QL* by explaining their use in transformation tasks. The reader can refer to our internal report (see Capelli et al. 2017) for a detailed description of the language.

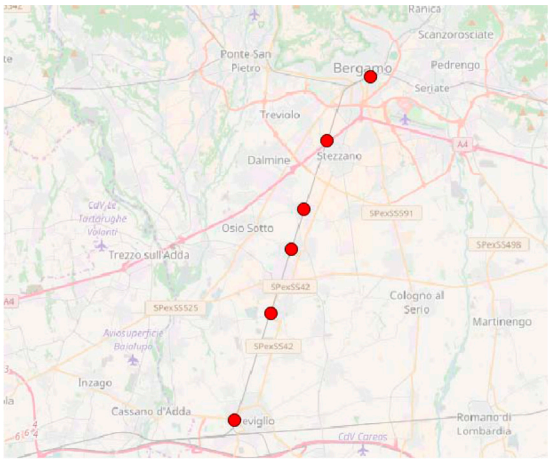
The *J-CO-QL Engine* actually executes *J-CO-QL* queries. It is a Java tool that receives queries from the *J-CO-QGIS Plug-in* , extracts collections from within



(a) DB Browser



(b) Collection Viewer



(c) Collection `TrainStops`

Figure 2. *J-CO-QGIS* plugin's DB Browser and Collection Viewer windows and train stops imported from a *MongoDB* collection.

MongoDB persistent databases, executes queries and saves final results into *MongoDB* persistent databases.

3.3. The J-CO-QGIS plug-in for QGIS

In order to provide analysts with a powerful tool for querying and visualizing data within classical *GIS* software, we developed a plug-in for *QGIS*, the free *GIS* software.

The plug-in, named *J-CO-QGIS*, provides basic, yet useful, functionalities. In particular:

- The *DB Browser* allows users to connect to *MongoDB* persistent databases to select the collections to show.
- The *Collection Viewer* takes collections from *MongoDB* persistent databases and loads those objects with geometrical representation (having the `~geometry` field) into *QGIS* as new information layers.
- The *Query Issuer* is a text editor that allows users to write *J-CO-QL* queries and send them to the *J-CO-QL* engine.

The plug-in adds two buttons to the tool-bar: the first one opens the *DB Browser* window, while the second one opens the *Query Issuer* window.

Figure 2(a) shows the *DB Browser*: first of all, the user must specify the connection string to the *MongoDB* server.[7] By clicking on the *Connect* button, the plug-in actually connects to the specified *MongoDB* server and shows the list of available databases. By browsing this list, shown in the *DB* list-box, the user chooses the database from which to get the desired collection. The content of the *Collection* list-box is updated, with the list of collections available in the chosen database.

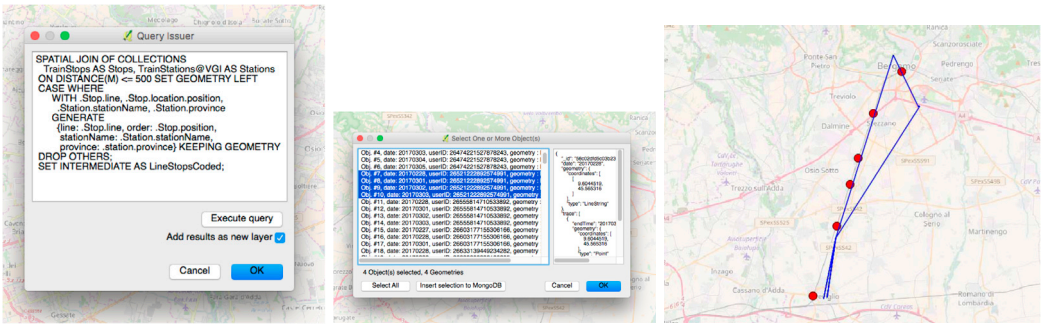
In Figure 2(a) we connected to our *Mongo* server, and selected the **`VGI`** database, that contains two collections: **`TrainStops`** and **`PeopleTraces`**. We selected the first collection.

The selected *Create new layer with collection* flag, causes the immediate generation of a *QGIS* layer with all geo-tagged objects contained in the chosen collection. After pressing the *OK* button, the *Collection Viewer*'s window is open.

Figure 2(b) shows the *Collection Viewer*. In this window, it is possible to select objects of interest to add to the layer; the right-hand side area shows the full structure of selected *JSON* objects. In the example, we selected objects having `"Bergamo-Treviglio"` as value for the `line` field. The *Insert selection to MongoDB* button re-exports the selected objects into a new collection within the persistent database, whose name will be asked in a prompt. The *OK* button shows the selected objects. Figure 2(c) depicts the layer created with the objects selected from collection **`TrainStops`**.

The user can write and execute *J-CO-QL* queries through the *Query Issuer* window, shown in Figure 3(a). Currently, this is a simple text editing window with a button that actually sends the query to the *Query Engine*.

Suppose that a complex process has been performed, obtaining a new collection named **`PeopleTraces`**, that describes traces of people. If we select objects related to traces of some people of interest (Figure 3(b)),



(a) Query Issuer (b) Collection Viewer (c) Layers for collections `TrainStops` and `PeopleTraces`

Figure 3. *Query Issuer*, selection of traces for a single person and the trace added as new layer in *QGIS*.

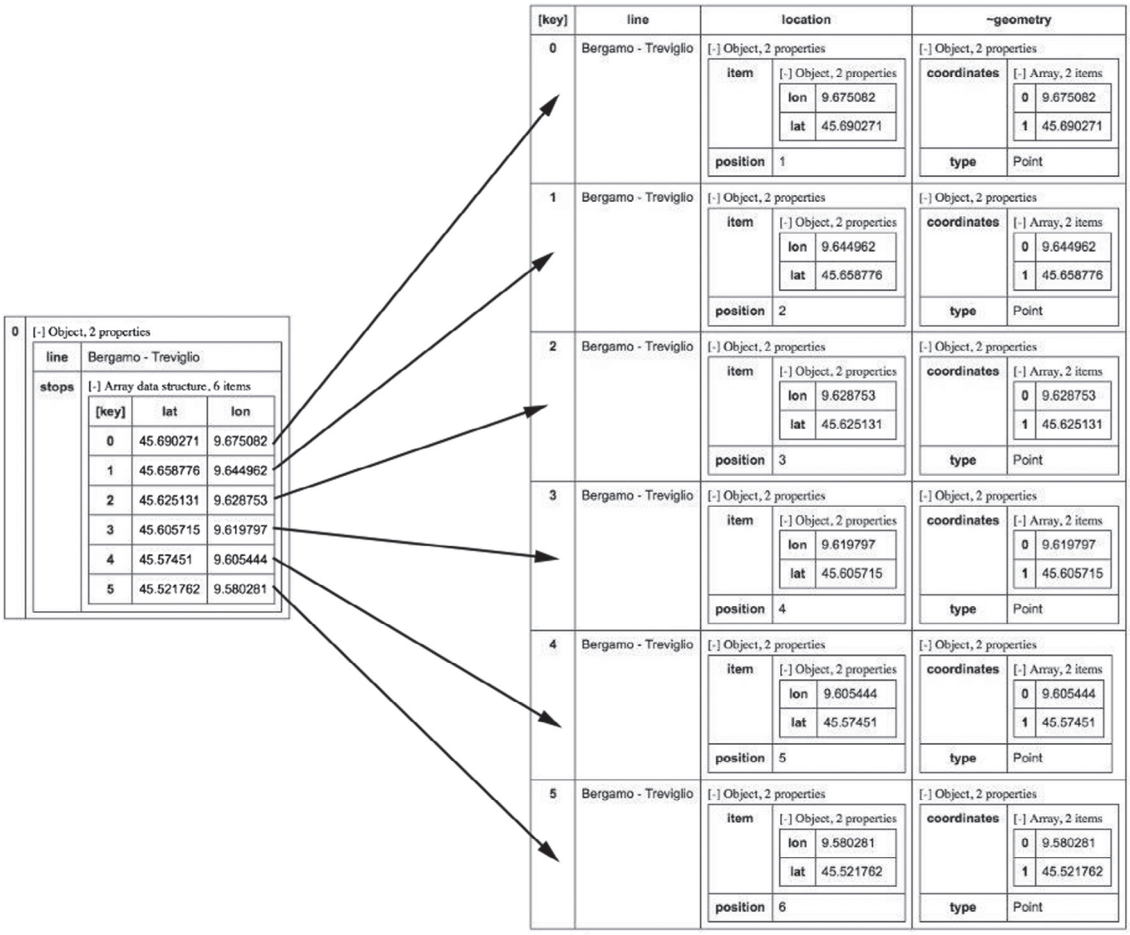


Figure 4. Intermediate collection **`TrainStops`** (stops of train lines).

in *QGIS* we can easily overlay this new layer with the one created for collection **`TrainStops`**, as shown in Figure 3(c). This possibility allows users to perform visual analysis of the query results.

4. Transforming data concerning train lines

We are now ready to illustrate how *J-CO-QL* can be exploited to specify complex transformation processes on our data sets. We will introduce the main *J-CO-QL* operators by exemplifying their behavior when they appear in the queries.

The first process we show transforms data concerning train lines. This process is necessary to build a representation of train lines suitable for next processes. Suppose we have a collection named **`SourceTrainLines`** stored in the persistent database named **`VGI`**. Each object in the collection describes a line, by means of two fields, named **`line`** and **`stops`**. The former field is the name of the line; the latter is an

array of objects, each one describing a stop in terms of longitude (field **lon**) and latitude (field **lat**). A sample is reported hereafter, concerning the line that connects *Bergamo* to *Treviglio* (a small city 20 km far away from Bergamo).

```
[{ "line": "Bergamo - Treviglio",
   "stops": [{"lon": 9.675082, "lat": 45.690271},
             {"lon": 9.644962, "lat": 45.658776},
             {"lon": 9.628753, "lat": 45.625131},
             {"lon": 9.619797, "lat": 45.605715},
             {"lon": 9.605444, "lat": 45.57451},
             {"lon": 9.580281, "lat": 45.521762}]
  }
]
```

This format is not suitable for processing people traces since there is no evidence about station names. Furthermore, there is not explicit geo-tagging (there is not a ~geometry field) compliant with our data model. Consequently, we need to transform it in order to perform other activities such as geo-coding.

The *J-CO-QL* query that performs the transformation is reported hereafter. In the following, we explain the query, by meantime introducing the operators.

```
GET COLLECTION SourceTrainLines@VGI;
EXPAND
   UNPACK WITH ARRAY .stops AND WITH STRING .line
    ARRAY .stops TO .location
    GENERATE SETTING GEOMETRY
             POINT(.location.item.lon, .location.item.lat)
DROP OTHERS;
SET INTERMEDIATE AS TrainStops;
```

The **GET COLLECTION** operator starts the process, retrieving the collection named **SourceTrainLines** from the persistent database named **VGI**; this collection becomes the new temporary collection.

At this point, we want to unnest objects contained within array fields named **stops**, in order to get one object for each stop. The **EXPAND** operator performs this task.

In details, the **UNPACK** clause is a condition that checks for the objects contained in the temporary collection that have all the desired fields; in particular, we want to process objects having an array field named **stops** and a string field named **line**.

For such objects, the array field named **stops** is unpacked and a new temporary object t_i is generated for each item contained within the array. t_i contains all fields in the input objects (except for the unpacked array field), plus an extra field named **location**, as specified after the **TO** keyword. Field **location** is a nested object that contains two fields: **item** is the item extracted from within the array (in this case, the object with fields **lon** and **lat**); **position** is the position of the item in the array (starting from 1). The **GENERATE** action actually generates the output object o_i to insert into the output temporary collection, by deriving it from t_i and by generating the ~geometry field (**SETTING GEOMETRY** specification), that is derived as a **POINT** (notice that coordinates are taken from the coordinates in the **item** object within the new field **location**).[8] The final option **DROP OTHERS** says that objects in the input temporary collection that donot match the **UNPACK** condition are discarded from the output (none, in this case).

The new collection is then saved by the **SET INTERMEDIATE** operator into the intermediate result database *IR* with name **TrainStops**.

The left-hand side of Figure 4 shows again collection **SourceTrainLines** pretty formatted; the right-hand side of figure shows the collection **TrainStops**; arrows show the correspondence between elements in the unpacked array and new objects.

4.1. Geo-coding of stops as stations

In order to associate every stop in the train lines with the corresponding station name, it is necessary to perform a geo-coding activity, based on a collection, named **TrainStations**, that reports information about stations, including their coordinates. Here, we will show how a usually non trivial task like geo-coding can be easily done by means of *J-CO-QL* operators.

Figure 5(a) reports collection **TrainStations** for our case study, describing stations on line *Bergamo-Treviglio*. Notice the presence of the ~geometry field in each object.

The geo-coding task on train lines is defined by the *J-CO-QL* query reported in Figure 6.

The key operator for performing geocoding is the **SPATIAL JOIN** operator (introduced in our language in Bordogna, Capelli, and Psaila (2017)). It takes two collections (either stored in a *MongoDB* persistent database or in the intermediate result database *IR*) and builds pairs of objects coming from the two in-

[key]	province	stationName	~geometry
0	Bergamo	Bergamo	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.674404 1: 45.690966 type: Point
1	Bergamo	Stezzano	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.643825 1: 45.659274 type: Point
2	Bergamo	Levate	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.629568 1: 45.625273 type: Point
3	Bergamo	Verdello	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.619515 1: 45.605834 type: Point
4	Bergamo	Arcene	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.605916 1: 45.574281 type: Point
5	Bergamo	Treviglio	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.580061 1: 45.522083 type: Point

(a)

[key]	line	order	province	stationName	~geometry
0	Bergamo - Treviglio	1	Bergamo	Bergamo	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.675082 1: 45.690271 type: Point
1	Bergamo - Treviglio	2	Bergamo	Stezzano	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.644962 1: 45.658776 type: Point
2	Bergamo - Treviglio	3	Bergamo	Levate	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.628753 1: 45.625131 type: Point
3	Bergamo - Treviglio	4	Bergamo	Verdello	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.619797 1: 45.605715 type: Point
4	Bergamo - Treviglio	5	Bergamo	Arcene	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.605444 1: 45.57451 type: Point
5	Bergamo - Treviglio	6	Bergamo	Treviglio	[-] Object, 2 properties coordinates: [-] Array, 2 items 0: 9.580281 1: 45.521762 type: Point

(b)

Figure 5. (a) Collection **`TrainStations`**, describing coordinates of stations; (b) The intermediate collection **`LineStopsCoded`**.

```
SPATIAL JOIN OF COLLECTIONS
   TrainStops AS Stop, TrainStations@VGI AS Station
 ON DISTANCE(M) <= 500 SET GEOMETRY LEFT
 CASE WHERE
      WITH .Stop.line, .Stop.location.position,
        .Station.stationName, .Station.province
      GENERATE
        {line: .Stop.line, order: .Stop.position,
         stationName: .Station.stationName,
         province: .station.province} KEEPING GEOMETRY
 DROP OTHERS;
 SET INTERMEDIATE AS LineStopsCoded;
```

Figure 6. Process "Geo-coding of Stops as Station".

put collections, on the basis of the relationship existing between their geometries.

In our case, we join objects in the collection **`TrainStops`** (in the intermediate result database *IR*), that describe stops of train lines, with collection **`TrainStations`**, stored in the persistent database **`VGI`**, that describe stations. The two collections are aliased, within the operator, as **`Stop`** and **`Station`**, respectively.

The spatial join is performed on the basis of the spatial join condition that follows the **`ON`** keyword: two objects are joined if their distance is less than 500 m (a reasonable distance, considering the area occupied by train stations). Let's consider two source objects l_i (from the left collection) and r_j (from the right collection). A temporary object $t_{i,j}$ is generated if l_i`.~geometry` and r_j`.~geometry` satisfy the spatial join condition. $t_{i,j}$ has three fields: one has the name (or alias) of the left collection and its value is the entire object l_i; the second field has the name (or alias) of the right collection and its value is the entire object r_j; the third one is `~geometry`, containing the overall geometry. As far as the geometry of $t_{i,j}$ is concerned, in our case we specified **`SET GEOMETRY LEFT`**, meaning that we keep

the geometry of the left object l_i; alternatively, we can specify **RIGHT**, **INTERSECTION**, **ALL** (the last option merges the two collections).

Figure 7 illustrates the semantics of spatial join. In Figure 7(a) and (b) we report, respectively, an object from collection **TrainStops** (aliased as **Stop**), that is the source object l_i, and an object from collection **TrainStations** (aliased as **Station**), that is the source object r_j. When the spatial join between those two objects is performed, the temporary object in Figure 7(c) is generated: notice the **Stop** and **Station** fields, which correspond to the objects shown in 7(a) and (b). In addition, the `~geometry` field equals the `~geometry` property in l_i object, as specified by the **SET GEOMETRY LEFT** construct.

Finally, the **CASE** block transforms all temporary objects $t_{i,j}$, in order to flatten them, removing nesting, as shown in Figure 7(d). Note the **DROP OTHERS** option, that drops all objects resulting from the spatial join that are not selected by the **WHERE** condition (in the example, all resulting objects satisfy the **WHERE** condition). As an example, Figure 7(d) reports the output object derived from the temporary object in Figure 7(c).

The resulting collection, shown in Figure 5(b), is stored into the intermediate result database *IR* with name **LineStopsCoded**.

The reader can notice that geo-coding has been performed in a straightforward way, directly working on *JSON* data and within the database, without the need to import collections into a *GIS* software; results are saved as *JSON* objects, ready for further transformations. Furthermore, notice that no attention must be paid to the actual format of geometries: the language is designed to be independent of the geometry representation.

4.2. Adding directions to lines

In order to get ready to discover volunteers that traveled by train, we need to further process the intermediate collection **LineStopsCoded** we obtained after geocoding. In fact, in this collection, the direction of trains is not represented. However, in order to actually find travel directions, we have to consider this issue.

The *J-CO-QL* query in Figure 8 further transforms collection **LineStopsCoded**, cloning geocoded stops and adding a new field, named **direction**, whose values can be either **"F"** (forward) or **"B"** (backward).

At first, geocoded stops in collection **LineStopsCoded** are grouped together. First of all, the **GROUP** operator builds a partition containing all objects satisfying the condition that follows the keyword **PARTITION**. These objects are grouped together by the field(s) specified after the **BY** keyword. For each group, an object is generated in the output temporary collection, that contains all grouping fields and the new array field specified after **TO** keyword. Later, this array field, that contains all the original objects that have been grouped together, is sorted by the optional **SORTED BY** specification. In our query, the **GROUP** operator groups together train stops that have the same value for field **line**, generating the array field named **list**; this array field is sorted in reverse order w.r.t. values of **position** field (this way, train stops of the same line are in reverse order, w.r.t. the original one).

Then, the subsequent **EXPAND** operator re-expands objects, having field **order** (derived from item position in the array) in reverse order w.r.t. the original sequence of stops in the line. Field **order** indicates the order of stops in the line, moving in backward direction. The resulting temporary collection is saved into the intermediate result database with name **LineStopsBackward**.

The **MERGE** operator is a binary operator that merges objects contained in two collections. In the query, collections **LineStopsCoded** and **LineStopsBackward** are merged together.

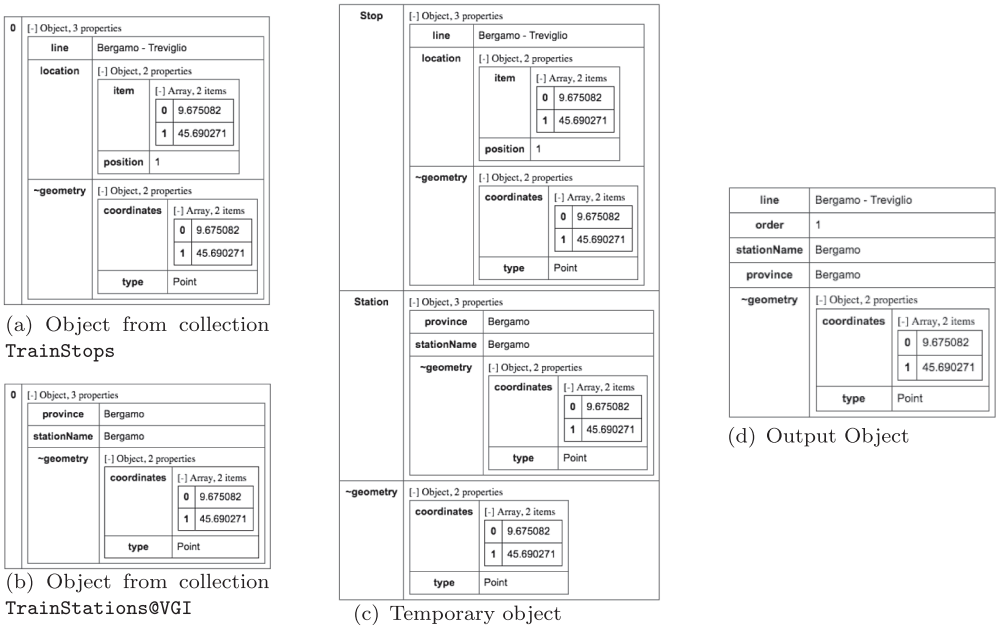
At this point, it is necessary to make objects homogeneous, because some of them have the **direction** field, while others have not (i.e. the ones describing the forward direction). In fact, the temporary collection contains objects with heterogeneous structures. The **FILTER** operator filters and restructures subsets of objects, based on the **WHERE** condition. In our query, the **WHERE** condition looks for objects without field **direction** (note the use of the **WITHOUT** predicate): these objects are regenerated by adding the missing field with value **"F"** and by renaming field **position** as **order**. The other objects in the input temporary collection are kept (option **KEEP OTHERS**), because they are complete already. The resulting collection is finally saved into the persistent database named **VGI**, with name **LineStopsStations**.

5. Transforming data concerning volunteers

In our research project, volunteered data are gathered through the app named *Moves*. Our web site is connected to the *API*s provided by *Moves* servers. Data returned by *Moves API*s are formatted as *JSON* data; however, their structure is very complex, and contains more information than that we need. Therefore, we defined a transformation process that brings the dataset into a format suitable for our analysis. Figure 9 shows the structure (schema) of objects provided by *Moves APIs*.

At the top level, we find only two fields; **userId** reports the volunteer identifier, while **storyline** is an array containing volunteer's activities. Each item of this array is a complex object describing activities of one single day (see field **date**). In particular, we are interested in field **segments**, that is an array of objects.

Each object of array **segments** describes an activity, that can be either a stay in a place (in this case, field



(a) Object from collection `TrainStops`

(b) Object from collection `TrainStations@VGI`

(c) Temporary object

(d) Output Object

Figure 7. Spatial Join semantics: source objects, temporary object, output object for procedure in Figure 6.

```
GET COLLECTION LineStopsCoded;
GROUP PARTITION
  WITH .line, .position, .stationName,
       .province, .~geometry
   BY .line TO .list SORTED BY .position DESC
   DROP OTHERS;
EXPAND UNPACK WITH ARRAY .list
       ARRAY .list TO .temp
   GENERATE { .line, direction: "B", order: .temp.position,
      stationName: .temp.stationName, province: .temp.province}
      SETTING GEOMETRY .temp.~geometry
   DROP OTHERS;
SET INTERMEDIATE AS LineStopsBackward;

MERGE COLLECTIONS LineStopsCoded, LineStopsBackward;
FILTER
  CASE  WHERE WITHOUT .direction
    GENERATE {.line, direction: "F",
       .order: .position, .stationName, .province}
      KEEPING GEOMETRY
   KEEP OTHERS;
SAVE AS LineStopStation@VGI;
```

Figure 8. Process "Adding Directions to Line".

type is **"place"**), or a movement from one place to another (in this case, field **type** has value **"move"**). The objects of the latter type do not describe places, thus we have to focus only on objects of the former type. Objects of type **"place"** have a field named **place**, which contains two more fields describing, respectively, latitude and longitude of the place.

The *J-CO-QL* query reported in Figure 10 performs this transformation and generates collection **PeopleTraces** reported in Figure 11(b). After operator **GET COLLECTION**, a first application of the **EXPAND** operator is necessary, in order to unnest objects within the array field **storyLine**; into the new field **SLItem**. Recall that this new field is an object with two fields, named **item** and **position**: the first one contains the actual item extracted from the array; the second one is the position occupied by the item within the array.

Since **.SLItem.item** is an object that contains an array of segments named **segments**, a second application of the **EXPAND** operator is necessary. The result is a temporary collection of objects, where each object describes, in the **seg** field, each segment originally contained in the **segments** field. Some of them have field **.Seg.item.type** with value **"place"**, other with value **"move"**: we want objects with type **"place"**.

To this purpose, the **FILTER** operator is used: the **WHERE** condition selects the desired objects and

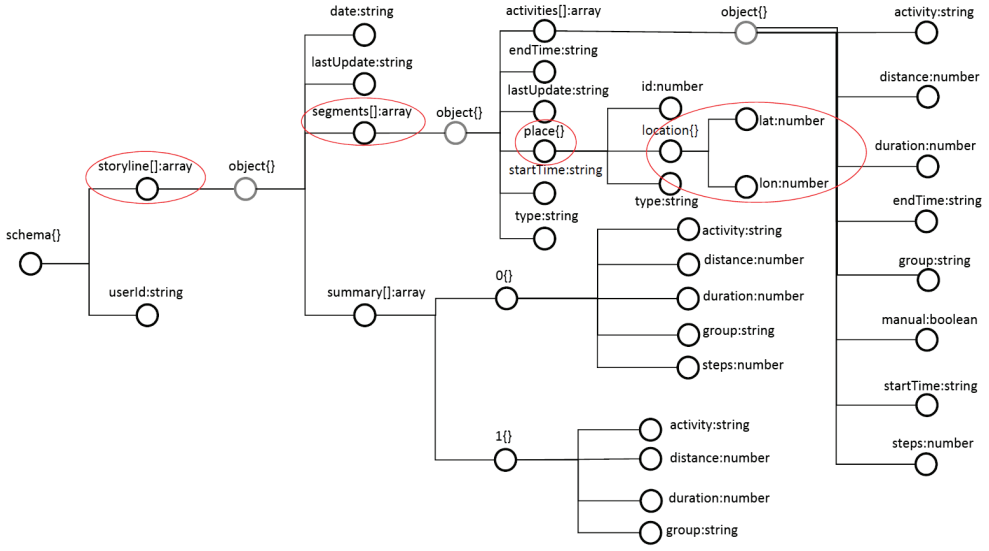


Figure 9. Schema for *Moves* traces.

```
GET COLLECTION MovesTraces@VGI;
EXPAND UNPACK WITH .userId, .storyLine
      ARRAY .StoryLine TO .SLItem
   DROP OTHERS;
EXPAND UNPACK WITH .userId, .SLItem
   ARRAY .SLItem.item.segments TO .seg
  DROP OTHERS;
FILTER
   CASE WHERE WITH .userId, .seg AND .seg.item.type = "place"
     GENERATE {.userId, date: .seg.item.date,
        startTime: .seg.item.startTime,
        endTime: .seg.item.endTime, location: .seg.item.location}
     SETTING GEOMETRY POINT(.seg.item.place.location.lon,
                            .seg.item.place.location.lat)
   DROP OHERS;
SAVE AS PeopleSinglePlaces@VGI;

GROUP PARTITION
 WITH .userId, .date, .startTime,  .endTime,
      .location, .~geometry
 BY .userId, .date INTO .Trace SORTED BY .startTime
     GENERATE SETTING GEOMETRY To_POLYLINE( .Trace )
   DROP OTHERS;
SAVE AS PeopleTraces@VGI;
```

Figure 10. Process "Transforming Data concerning Volunteers".

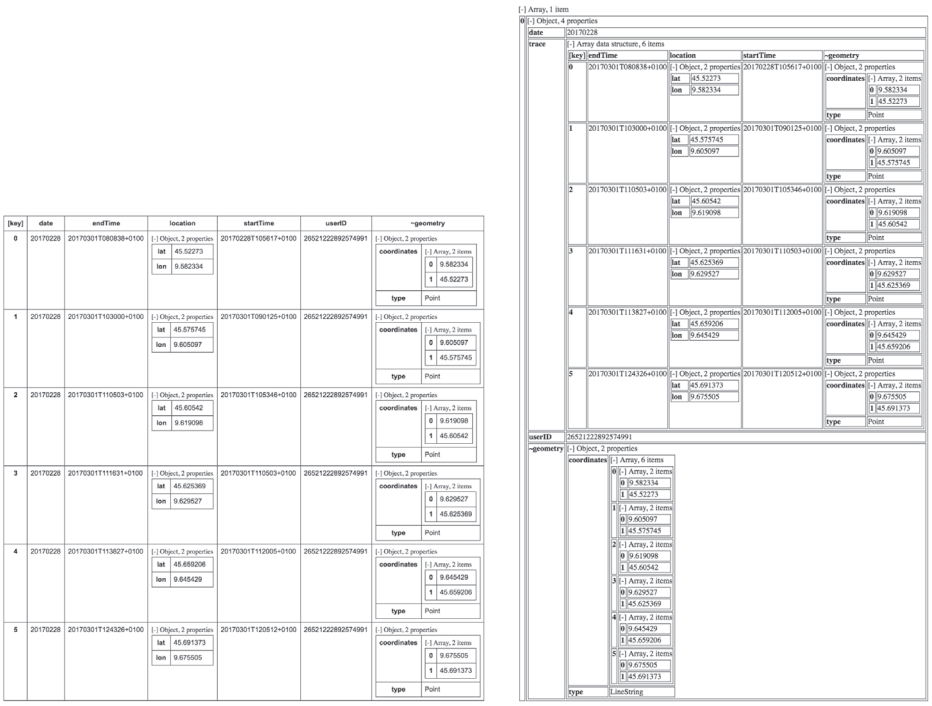
transforms them into simplest objects, deriving the geometry from **lat** and **lon** fields nested within the complex field **place**.

The resulting collection is saved into the persistent database **VGI** with name **PeopleSinglePlaces**, which is shown in Figure 11(a).

To complete the preparation of volunteers' traces, in order to visualize them in *QGIS*, we further derive collection **PeopleTraces** shown in Figure 11(b). This task is performed by means of the **GROUP** operator. Objects are grouped together on the basis of fields **userId** and **date**; the array containing all grouped objects is named **Trace** and it is sorted by field **startTime**; finally, the overall geometry is derived by aggregating geometries of all points in array **Trace** into a polyline.

6. Discovering train voyagers

We are finally ready to discover train voyagers using collections **LineStopsStations** and **PeopleSinglePlaces** obtained by the previous processes. In general terms, a train voyager is a volunteer whose trace is mostly overlapped with a train line. But a question naturally arises: Is it possible to detect volunteers traces overlapped with train lines, by means of *J-CO-QL* current set of operators (that do not encompasses operators for sequence matching)? The answer is *"yes, it is possible"*. The *J-CO-QL* query reported in Figure 12 shows the way in which this can be performed. The key of the process is, again, the spatial join.



(a) Collection `PeopleSinglePlaces` (b) Collection `PeopleTraces`

Figure 11. Intermediate collection (a) **`PeopleSinglePlaces`** (set of all places visited by people) and (b) of collection **`PeopleTraces`** (each user has, for each date, the trace within a field)

The **`SPATIAL JOIN`** operator at the beginning of the query is used to find out points in volunteers' traces that correspond to train stations: the spatial join is performed on objects in collection **`PeopleSinglePlaces`** (depicted in Figure 11(a)) and collection **`LineStopsStations`** (depicted in Figure 5(b)); they are respectively aliased as **`Trace`** and **`Stop`**. The spatial join occurs if the trace point is less than 500 m from the station point. For each pair of spatially joined objects, a new output object is generated, which reports the user id, the date, the time, the train line, the direction, the order in the direction, the station name and the province. The collection is saved into the intermediate result database *IR* with name **`PeopleInStations`**.

Clearly, for the same station, two objects are generated, one with direction **`"F"`** and one with direction **`"B"`**. Consequently, it is necessary to understand if the user actually traveled by train. The idea is simple: we look for pairs of consecutive stops visited by people in the same direction and with increasing time. To do so, the **`JOIN`** operator joins the intermediate collection **`PeopleInStations`** with itself; the two occurrences of the same collection are aliased as **`PS1`** and **`PS2`**. The **`JOIN`** operator behaves similarly to the **`SPATIAL JOIN`** operator, but the pairs of objects are built on the basis of **`WHERE`** conditions. The **`WHERE`** condition we wrote handles with two different situations at the same time: if a pair of stops of the same user in the same date is built for direction **`"F"`**, the order of point **`PS2`** must follow the order of point **`PS1`** and the same is for **`startTime`**; if the direction is **`"B"`**, the order of point **`PS2`** must precedes the order of point **`PS1`**. Users with such pairs actually traveled by train.

To remove duplications, the **`GROUP`** operator groups pairs, in order to get objects that, at the external level, describes user, date, line and direction; the resulting collection is stored into the intermediate result database with name **`PeopleByTrain`**.

The last **`JOIN`** operator in the query is necessary to produce the final collection, then save it into the persistent database **`VGI`** with name **`PeopleByTrain`**, that contains traces of people discovered in the previous steps. It is the subset of traces in collection **`PeopleTraces`** that actually reveal people traveling by train. In our sample data, the only user we considered actually traveled by train, so the content of collection **`PeopleByTrain`** is the same as that of collection **`PeopleTraces`** (see Figure 11(b)).

Once people traveling by train are discovered, the analyst can conduct an exploratory activity through the *J-CO-QGIS* plug-in. For instance, Figure 3(c) shows the train line and the trace of our traveler overlapped on the layer describing roads.

7. Conclusions

In this paper, we presented an innovative framework for performing cross-analysis of heterogeneous data sets in the form of collections of possibly geo-tagged

```
SPATIAL JOIN OF COLLECTIONS
  PeopleSinglePlaces@VGI AS Trace, LineStopStations@VGI AS Stop
 ON DISTANCE(M) <= 500 SET GEOMETRY LEFT
 CASE WHERE WITH .Trace.userId, .Trace.date,  .Trace.startTime
    AND WITH .Stop.line, .Stop.order, .Stop.direction,
             .Stop.stationName, .Stop.province
   GENERATE {userId: .Trace.userId, date: .Trace.date,
       startTime: .Trace.startTime,  line: .Stop.line,
       direction: .Stop.direction, order: .Stop.order,
       stationName: .Stop.stationName, province: .Stop.province}
      KEEPING GEOMETRY
 DROP OTHERS;

SET INTERMEDIATE AS PeopleInStations;

JOIN OF COLLECTIONS PeopleInStations AS PS1,
   PeopleInStations AS PS2
 CASE
  WHERE .PS1.userId = .PS2.userId AND
        .PS1.date = .PS2.date AND
        .PS1.line = .PS2.line AND
        .PS1.direction = .PS2.direction AND
        .PS1.startTime < .PS2.startTime
        AND ((.PS1.direction = "F" AND
              .PS1.order < .PS2.order)
              OR
              (.PS1.direction = "B" AND
              .PS1.order > .PS2.order))
   GENERATE {userId: .PS1.userId,
             date: .PS1.date,
             line: .PS1.line,
             direction: .PS1.direction }
   DROPPING GEOMETRY
 DROP OTHERS;

GROUP
   PARTITION WITH .userId, .Date, .line, .direction
   BY .userId, .date, .line, .direction
   INTO .list
   GENERATE {.userId, .date, .line, .direction}
 DROP OTHERS;

SET INTERMEDIATE AS PeopleByTrain;

JOIN OF COLLECTIONS PeopleByTrain AS PBT,
    PeopleTraces@VGI AS T
  CASE
   WHERE .PBT.userId = .T.userId AND
      .PBT.date = .T.date
   GENERATE {userId: .T.userId, date: .T.date, trace: .T.trace}
        SETTING GEOMETRY .T.~geometry
 DROP OTHERS;
SAVE AS PeopleByTrain@VGI;
```

Figure 12. Process "Discovering Train Voyager".

JSON objects. The idea is to provide powerful tools to analysts and geographers that make possible to perform complex analysis processes without writing programs, but specifying processes in a declarative way. The framework is founded on a high-level declarative query language, named *J-CO-QL*, specifically devised to query heterogeneous collections of (possibly) geotagged *JSON* objects. The execution model on which *J-CO-QL* relies is simple, intuitive, and suitable to express complex queries specifying several subtasks.

The framework is completed by *J-CO-QGIS*, a plugin for *QGIS* that makes it possible to visualize geotagged *JSON* objects as information layers that can be overlapped with other information layers.

A real case study is illustrated that needs to cross-analyze authoritative geo-spatial data relative to

transportations and volunteer's traces collected by the use of a smart app. The case study is introduced to exemplify the possible use of the language operators. So we just presented those operators that were needed to solve the specific tasks of the case study. In fact, the language explicitly supports geographical representation and spatial aggregation operations.

The case study also shows that the *J-CO* framework can be effectively applied to cross-analyze transport data, coming from open data portals, authoritative sources and volunteers' data. Obviously, the process to write depends on the actual problem to address, but the methodology is the same we show in the paper.

We are aware that *J-CO-QL* operators are not "easy" and need a significant effort to be used; however, the number of operators to apply is limited, even for complex processes. Readers could object that a graphical user interface could be better. However, such a user interface must rely on a robust semantic foundation. *J-CO-QL* plays this role: in fact, we plan to extend the *J-CO-QGIS* plug-in with a graphical tool to specify *J-CO-QL* queries without writing *J-CO-QL* code.

In our opinion, the *J-CO* framework could become a day-to-day tool for analysts. For example, if enriched with data mining operators, it could be a valuable tool to support knowledge discovery processes, in a similar way to what previously studied in Meo and Psaila (2002) for XML. Furthermore, based on our previous experience in defining query languages for non-classical contexts (see Bordogna et al., 2002, 2008; Bordogna and Psaila, 2004, 2009; Psaila 2002), novel operators could be added, in order to provide users with powerful tools manipulating collections in a very complex way.

For example, we want to address the data independence principle, i.e. providing the capability of writing queries reducing the degree of awareness about the actual structure of *JSON* objects. This could be addressed in two ways: by exploring the possibility to define new higher level operators; by extending the *J-CO-QGIS* plug-in with a wizard that semi-automatically composes operator applications by exploring in advance the actual content of source collections and of intermediate results.

Notes

1. http://www.gbif.org/dataset/50c9509d-22c7-4a22-a47d-8c48425ef4a7.
2. http://ebird.org/content/ebird/.
3. https://apps.moves-app.com/.
4. MongoDB: https://www.mongodb.com/ *CouchDB*: http://docs.couchdb.org/en/2.0.0/.
5. www.couchbase.com/n1ql.
6. www.jsoniq.org.
7. The connection URL has the URI form mongodb://user:password@example.com/the_database?authMechanism=SCRAM-SHA-1.
8. In *J-CO-QL*, fields are referenced by a dot notation which starts with a dot. For Example, **`.location`** is a field in the root level of the object, **`.location.item.lat`** references a field **`lat`** contained in the upper object field **`item`** that, in turns, is contained in the root object field **`location`**. The "." at the beginning means that we start from the root of the object, i.e. the most external level of the object.

Notes on contributors

Gloria Bordogna, PhD, is a senior researcher of IREA CNR Institute for the Electromagnetic Sensing of the Environment of the Italian National Research Council. Her current research concerns flexible content-based querying of catalogue services, fuzzy ontologies for VGI creation and querying, and fuzzy quality assessment and spatio-temporal analytics of crowdsourced geotagged information. She has published 200 papers in journals and in the proceedings of international and national conferences, among which 90 papers are listed in the Web of Science.

Steven Capelli received his PhD in computer science from the University of Bergamo (Italy) in 2017. Currently, he is a post-doc at Dipartimento di Informatica, Sistemistica e Comunicazione of Università degli Studi di Milano-Bicocca (Italy). His research interests have been focused on information security, service component architecture, NoSQL databases, and geoSpatial data, Information retrieval.

Daniele E Ciriello, PhD, received his MSc degree from the University of Bergamo in 2016. He is now post-graduate fellow at DITESIS Lab at the University of Bergamo.

Giuseppe Psaila is a researcher and professor at the University of Bergamo, Italy. He received his PhD from the Politecnico di Torino (Italy) in 1998. Current research interests cover retrieval and management of open data, big data, and geographical data.

ORCID

Gloria Bordogna http://orcid.org/0000-0002-6775-753X
Giuseppe Psaila http://orcid.org/0000-0002-9228-560X

References

Bordogna, G., A. Campi, G. Psaila, and S. Ronchi. 2002. "A Language for Manipulating Clustered Web Documents Results." Proceedings of CIKM08, Napa Valley, CA, November 2008.

Bordogna, G., S. Capelli, and G. Psaila. 2017. "A Big Geo Data Query Framework to Correlate Open Data with Social Network Geotagged Posts." In *International Conference on Geographic Information Science*, 185–203. Berlin: Springer.

Bordogna, G., M. Pagani, and G. Psaila. 2006. "Database Model and Algebra for Complex and Heterogeneous Spatial Entities." In *Progress in Spatial Data Handling*, edited by A. Riedl, W. Kainz and G. A. Elmes, 79–97. Berlin: Springer.

Bordogna, G., and G. Psaila. 2004. "Fuzzy-spatial SQL." In *International Conference on Flexible Query Answering Systems*, 307–319. Berlin: Springer.

Bordogna, G., and G. Psaila. 2009. "Soft Aggregation in Flexible Databases Querying based on the Vector p-norm." *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 17 (supp01): 25–40.

Bordogna, G., and G.Psaila. 2008. "Customizable Flexible Querying for Classical Relational Databases." In *Handbook*

of Research on Fuzzy Information Processing in Databases, edited by J. Galindo, 191–217. Hersey: Information Science Reference.

Butler, H., M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub. 2016. *The GeoJSON Format*. Technical Report RFC 7946. Fremont, CA: IETF.

Capelli, S., P. Fosci, F. Marini, and G. Psaila. 2017. *J-CO-QL: A Flexible Query Language for Complex Geographical Analysis of Heterogeneous Geo-tagged JSON Data Sets*. Technical Report. http://cs.unibg.it/psaila/j-co-ql/internal-report.pdf.

Cattell, R. 2011. "Scalable SQL and NoSQL data stores." *SIGMOD Record* 39 (4): 12–27.

Fitzpatrick, J. W., F. Gill abd, M. Powers, J. V. Wells, and K. V. Rosenberg. 2002. "Introducing eBird: The Union of Passion and Purpose." *North American Birds* 56 (2): 11–12.

Han, J., E. Haihong, G. Le, and J. Du. 2011. "Survey on NoSQL Database." In *2011 6th International Conference on Pervasive Computing and Applications (ICPCA)*, 363–366. New York: IEEE.

Meo, R., and G. Psaila. 2002. "Toward Xml-based Knowledge Discovery Systems." In *2002 IEEE International Conference on Data Mining, 2002. ICDM 2003. Proceedings*, 665–668. New York: IEEE.

Nayak, A., A. Poriya, and D. Poojary. 2013. "Type of NOSQL Databases and its Comparison with Relational Databases." *International Journal of Applied Information Systems* 5 (4): 16–19.

Ong, K. W., Y. Papakonstantinou, and R. Vernoux. 2014. "The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases." *CoRR*. abs/1405.3631.

Psaila, G. 2002. "Erx-ql: Querying an Entity-relationship db to Obtain Xml Documents." In *The 2002 International Workshop on Database Programming Languages*, Frascati, Italy, September 8–10.

Psaila, G. 2011. "A Database Model for Heterogeneous Spatial Collections: Definition and Algebra." *2011 International Conference on Data and Knowledge Engineering (ICDKE)*, 30-35. New York: IEEE.

Robin, H., and S. Jablonski. 2011. "NoSQL Evaluation: A Use Case Oriented Survey." CSC-2011 International Conference on Cloud and Service Computing, 336–341. China, Hong Kong, December.

White, T. 2012. *Hadoop: The Definitive Guide*, O'Reilly Media.