WHY WE NEED A NOVEL FRAMEWORK TO INTEGRATE AND TRANSFORM HETEROGENEOUS MULTI-SOURCE GEO-REFERENCED INFORMATION: THE J-CO PROPOSAL

Gloria Bordogna[1] and Giuseppe Psaila[2]

[1]CNR IREA - Via Bassini 15 - 20133 Milano - Italy
bordogna.g@irea.cnr.it
[2]University of Bergamo - Viale Marconi 5 – 24044 Dalmine (BG) - Italy
giuseppe.psaila@unibg.it

ABSTRACT

The large number of geo referenced data sets provided by Open Data portals, social media networks and created by volunteers within citizen science projects (Volunteered Geographical Information) is pushing analysts to define and develop novel frameworks for analysing these multisource heterogeneous data sets in order to derive new data sets that generate social value. For analysts, such an activity is becoming a common practice for studying, predicting and planning social dynamics. The convergence of various technologies related with data representation formats, database management and GIS (Geographical Information Systems) can enable analysts to perform such complex integration and transformation processes. JSON has become the de-facto standard for representing (possibly geo-referenced) data sets to share; NoSQL databases (and MongoDB in particular) are able to natively deal with collections of JSON objects; the GIS community has defined the GeoJSON standard, a JSON format for representing georeferenced information layers, and has extended GIS software to support it.

However, all these technologies have been separately developed, consequently, there is actually a gap that shall be filled to easily manipulate GeoJSON objects by performing spatial operations. In this paper, we pursue the objective of defining both a unifying view of several NoSQL databases and a query language that is independent of specific database platforms to easily integrate and transform collections of GeoJSON objects. In the paper, we motivate the need for such a framework, named J-CO, able to execute novel high-level queries, written in the J-CO-QL language, for JSON objects and will show its possible use for generating open data sets by integrating various collections of geo-referenced JSON objects stored in different databases.

KEYWORDS

Collections of JSON objects, Geo-tagged data sets, Query Language for geographical analysis, Powerful spatial operators

1. INTRODUCTION

Geo-referenced information from Open Data portals, Volunteered Geographic Information (VGI), and crowdsourced information from social networks are recognized as a potential driver for social changes: companies are relying on such information to enhance their existing services or to derive knowledge from its analysis to create social value [3]. As described in the European Commission report on the reuse of open data, the European Data Portal has more than doubled the amount of data it references. In general, a large number of sources are now available to get information about territories, including corpora managed by private companies like Google and Facebook, that collect and integrate official information, VGI and crowd-sourced information about any kind of place.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

In order to turn such geo-referenced information into social value, the so-called data-value chain process must be carried out: once geo-referenced data (geo-data for short) are created, they have to be validated, for example through filtering, normalization and quality assessment, and shared by means of a Web geo-portal, after which they can be analysed. From integrating different geo-data sets, new data can be created, which can lead to new data services or products. It can be seen that, in order for a geo-data analyst to perform such tasks in an easy way, a framework is needed so as he/she can perform several manipulation operations on geo-data, which are heterogeneous, as far as their source, structure, format and semantics are concerned.

In effect, performing integration and transformation processes asks for the convergence of various technologies, originally developed separately, that now all together contribute to this ambitious goal. In particular, we consider data representation formats, database management and GIS (Geographical Information System) technology.

Let us start with the area of *data representation formats*. After the introduction of XML (eXtensible Mark-up Language) at the end of the 1990s, that had to become "the language" for information interchange on the Internet, currently we are observing the rapid diffusion of JSON (JavaScript Object Notation) as a de-facto standard for data interchange, in particular through API interfaces and Web Services. JSON is a flexible format to encode semi-structured compact information. Often, data sets provided by Open Data portals as well as by Web Service APIs contain geo-referenced information, i.e., data are tagged with positions on the Earth Globe (in terms of longitude and latitude), since they describe data concerning territories.

As far as the area of *Database Management* is concerned, the last decade is characterized by the development of so-called *NoSQL databases*, i.e., DBMSs (Data Base Management Systems) which are not based on the relational data model and, consequently, abandon SQL as query language. In particular, value-store, column-store and document-store are different models of NoSQL DBMS, where document stores are able to manage collections of JSON objects in a native way. The most famous representative of this category is *MongoDB* designed to manage large amounts of (relatively small) JSON objects, even though it provides spatial indexes that enable to efficiently perform some types of spatial queries.

As far as *GIS Technology* is concerned, GIS tools are now even more important than in the recent past: in fact, the availability of geo-data sets asks for visualizing such data on maps in the form of information layers, possibly integrating data from distinct data sets. In this respect, it was essential to introduce a standard format for describing information layers: the GIS community has defined the *GeoJSON* format, i.e., a standard format for describing geographical information layers that relies on JSON as syntactic framework.

Apparently, the above-mentioned convergence should be mature, but this is not true. In fact, the different perspectives that have driven the diverse developments make actually difficult to easily and effectively transform and integrate JSON data sets and/or GeoJSON layers, in particular when they are collected within databases managed by MongoDB (or, worse, in simple files). A unifying framework that provides analysists with the capability of effectively integrating and transforming JSON data sets and GeoJSON layers is essential.

These are the main reasons that motivated us to conceive a new framework, named J-CO. The goal of the framework is twofold: it has to provide the capability of working on different MongoDB databases at the same time, allowing analysts to easily integrate data sets stored in different databases; it has to provide a query language, named J-CO-QL, for manipulating collections of JSON and GeoJSON objects, natively supporting spatial operations and representations. The paper will describe, through a study case example, how the defined query language can be used to integrate and transform JSON data sets to create GeoJSON layers:

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

specifically, we consider quartiers and pharmacies of Milan (city in the northern Italy) and we will generate two GeoJSON layers describing quartiers with less than two pharmacies and quartiers with at least two pharmacies. We hope this way the paper will clarify how the J-CO framework can implement the convergence of the above-mentioned technology.

The paper is organized as follows. Section 2 presents the background of our project and related work. Section 3 introduces the J-CO framework, detailing the main features of the J-CO-QL language. Section 4 shows the example and how the J-CO-QL language can be used to perform complex transformations on collections of JSON data. Finally, Section 5 draws the conclusions.

2. RELATED WORKS

2.1. Motivation of the Proposal

Our seminal idea of developing a framework for integrating and transforming collections of JSON objects originated in the *Urban Nexus Project* [12]. The goal of this project is to gather information from several distinct open data repositories on the Web, authoritative and statistical sources, social media and so on, to study how city users live their city and territory. The idea is that geographical studies should take advantage of Big Data, in the sense of large variety of data sets coming from diverse data sources. In such a context, analysts are not programmers and need an integrated framework for performing their analyses. Nevertheless, since data come from many sources as JSON or GeoJSON data sets, it was necessary to develop a novel query language for this purpose. These considerations motivated the development of the J-CO framework.

In [5, 18, 19], we tackled the objective of exploiting social media to trace movements of social media users. We named this project *FollowMe*, because we traced (and we are still tracing) travellers that post geo-located messages on *Twitter*, detecting them in a pool of 30 airports potentially connected with the airport of Bergamo (northern Italy). Next [7], we integrated the *FollowMe* project within a framework for analysing trips of *Twitter* users, furthermore [8], we experimented a clustering technique for identifying common paths followed by users during their trips. That was a preliminary work of the *Urban Nexus* project, in which huge numbers of trips of *Twitter* users represented as JSON objects have to be analysed on the basis of multi-paradigmatic approach (see [12]). While facing this analysis task, we experienced the limitations of current query languages for heterogeneous geo-data in the form of JSON objects.

2.2. Manipulating Heterogeneous Big Geo-Data

The first attempt to abandon the relational data model in favour of more flexibility on the structure of data dates back to the late 1990s with the advent of XML (eXtendible Mark-up Language) as the universal data format for exchanging data over the Internet that stimulated the idea of developing database technology for storing and querying XML documents. Many proposals for XML databases and related query languages were proposed. The reader can refer to [22, 24,27] for some surveys. Obviously, speaking about convergence of technologies, the research area of data mining met the research area of XML-native databases, in order to perform data mining and knowledge discovery directly on XML documents stored within XML databases [30,33]. The idea is that the ability of XML to represent semi-structured and complex data enables to store, within the same XML database, both the mined data and the mined patterns.
However, the potentiality of XML to become “the representation format” met several practical obstacles that have limited its diffusion and favoured the emergence of JSON, namely its

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

extreme verbosity and difficulty of importing data described by XML documents within programs and information systems.

The adoption of JSON and NoSQL databases are motivated by the need for both flexibility and compactness as far as data structures are concerned; an interesting survey on NoSQL databases can be found in [12], where several systems are catalogued 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 [26]. The query language provided by such systems does not allow complex and multi-collection transformations. Readers interested in NoSQL DBMSs evaluation can refer to [37] and to [15].

As far as query language for JSON objects are concerned, several proposal 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 [5]. Anyway, it is worth mentioning them.

Jaql (see [34]) was designed to help Hadoop (see [40]) 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; its constructs are difficult to understand for non programmer users.

An interesting proposal is SQL++,defined to query both JSON native stores and SQL databases. The SQL++ semi-structured data model is a superset of both JSON and the SQL data model [35]. Yet, SQL ++ is SQL backwards compatible and is generalized towards JSON by introducing only a small number of query language extensions to [35]. 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, i.e., it does not provide constructs similar to the WHERE branches provided by J-CO-QL. Furthermore, complex transformations that require several queries sequentially would executed need to explicitly save intermediate results into the persistent database (although in [35] nothing is said about data manipulation operators such as INSERT). In contrast, the execution model on which J-COQL relies clearly separate persistent databases and temporary databases, by means of the temporary collection and the intermediate result database IR.

The industry is looking at the extension of SQL to query JSON objects. An example is N1QLyy 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, 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. However, unlike XQuery, JSONiq is not concerned with the peculiarities of XML, like mixed content, ordered children, or the complexities of XML Schema, and so on. Nevertheless, like XQuery it can be hardly used by unexperienced users.

Although these languages are declarative, they are still oriented to a programmer vision.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

Other approaches to manipulate heterogeneous big data recognize the importance of a declarative query language to guarantee the data independence principle [20, 26]. For example, SparkSQL [1], was developed with an SQL interface to query heterogeneous big data sets managed within the Spark distributed processing infrastructure. It introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data. Nevertheless, it does not support spatial operators.

GeoSPARQL [2] is a Geographic Query Language for RDF Data proposed as standard by the OGC consortium for querying geospatial data on the Semantic Web. GeoSPARQL is designed to accommodate systems based on qualitative spatial reasoning and systems based on quantitative spatial computations to ease data analysis.

Also our proposal is oriented to data analysts, which need to manage heterogeneous collections of real world entities, namely collection in both JSON and GeoJSON.

It is somehow related to the world of *PolyStore DBMS*, i.e., database management systems that deal with several DBMS at the same time, each of them possibly providing a different logical model, such as relational, graph, JSON, pure-text, images, videos. An interesting work on this topic is *BigDAWG*.

Our proposal moves from our previous work on the problem of querying heterogeneous collections of complex spatial data (see [11, 36]). In that works, we proposed a database model capable to deal with heterogeneous collections of possibly nested spatial objects, based on the composition of primitive spatial objects; at the same time, an algebra to query complex spatial data is provided, inspired by classical relational algebra. W.r.t. those previous works, J-CO-QL relies on the JSON standard, thus we do not define an ad-hoc data model; furthermore, J-CO-QL abandons the typical relational algebra syntax, because it relies on a more flexible and intuitive execution model. Nevertheless, the experience made in [28] helped us, where we defined a language for manipulating clusters of web searches performed through a mobile device.

3. A FRAMEWORK FOR GEOREFERENCED DATA TRANSFORMATIONS

The framework we conceived for integrating and transforming multisource geo-referenced JSON data is named J-CO (*J*SON *CO*llections) and comprehends several components depicted in Figure 1.

- One or more NoSQL databases managed by MongoDB (and, in the future, by other systems like *ElastichSearch*). This feature resembles a distributed federated database architecture [28].

- The *J-CO-QL Engine* that executes queries directly operating on data stored in MongoDB databases. It receives queries through a Web Service interface. This feature is typical of distributed databases querying [31].

- A GIS application, like, e.g., QGIS [23], the open source GIS software, that can be used as environment to query and to show GeoJSON layers.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

- A (future) User Interface. Through this interface, the analysis will be able to carry on the transformation process dynamically. This interface can be developed as a plug-in of a GIS application.

- Any kind of publishing tool for geo-data stored within MongoDB-managed databases, such as Open Data geo portals, Web Map services and Web Feature Services [32], etc.

Note that the ability of the J-O-QL Engine of connecting with several databases during the same transformation process is a key feature: in fact, this feature allows analysts to easily integrate data sets, by taking them from the servers that store them. This way, it is possible to avoid a large amount of efforts for transferring data from one server to another, in accordance with optimization techniques in loosely coupled federated databases [13]. In them, a unique schema for queries does not exists but a uniform query language is made available ,which abstracts from the query languages of the components, and hides technical and language heterogeneity. Thus, every user is responsible for handling logical heterogeneity in the components.
Nevertheless, the possibility of visualising data and results of the analysis through a GIS software can greatly help analysts formulate queries and perform a visual analysis of results. This can be done by generating GeoJSON layers during the transformation process performed by means of the J-CO-QL Engine.

On the same line, it is important to be able to publish results anywhere, for example in Open Data Portals. These portals often provide geo-referenced data as GeoJSON layers, but this is not mandatory. Often, when data are not geo-referenced, simple JSON collections are published. In this scenario, the J-CO framework is designed to play a central role, towards the simplification of tasks that, without it, could be very tedious and much more time consuming than necessary.

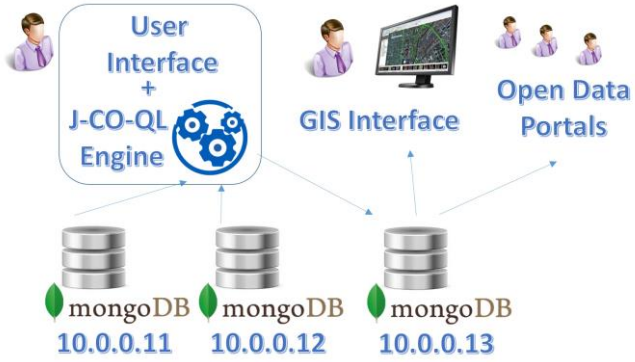

Figure 1. Application Scenario for the J-CO Framework

3.1. J-CO-QL Main Features

The query language named J-COQL is defined to work with *collections of JSON objects*. JSON is a serialized representation for objects. Fields (object properties) can be simple (numbers or strings), complex (i.e., nested objects) and vectors (of numbers, strings and objects).

JSON does not consider geo-references. An official proposal in this sense is the GeoJSON standard [14, 17]. Defined by the GIS community, it provides an excellent format for defining geometries of geo-referenced data. Fields describing geometries are named, in the GeoJSON standard, `geometry`. In J-CO-QL, we rely on the same standard for representing geometries,

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

but the name of the field considered by the J-CO-QL language to handle geometries is `~geometry`: this way, J-CO-QL is able to handle GeoJSON layers in a seamless way.

When the `~geometry` field is absent in an object processed by J-CO-QL, this means that no geo-reference is present in the object and no spatial operations can be performed on it. Specifically, the `~geometry` field (when present) is based on the `GeometryCollection` type for the GeoJSON standard.

Figure 2, reports sample objects of JSON objects with `~geometry` field. The reader can see objects describing quartiers of Milan, as well as objects describing pharmacies in Milan. Coordinates (longitude and latitude) are expressed based on the (World Geodetic System) 84, our default CRS (Coordinate Reference System).

In a NoSQL environment such as MongoDB, a *Database* is viewed as a *set of collections*, while a *Collection* has a *name* and its instance is viewed as a vector of JSON objects. To manipulate JSON collections and to store their results into new collections, in a transparent way w.r.t. to the databases from which to get collections and to which to store collections, we need operators that meet the *closure property*, that is, they get collections and generate collections. This is a first design requirement for the J-CO-QL language [38]. Other key features of the language are reported hereafter.

- J-CO-QL provides operators specifically designed to deal with objects with different structure within the same operation.
- Operators provided by J-CO-QL are high-level operators, which allow analysts to think directly to objects structure; they do not have to write low-level procedures.
- Finally, but not less important, J-CO-QL directly deals with geo-reference possibly contained in JSON objects, because the data model explicitly deals with them through the `~geometry` field.

Queries are sequences of operators applied to collections [16, 29]. The execution process of queries is based on the concept of *state of the query process*, that is a pair *s* = (*tc*; *IR*), where *tc* is a collection named *Temporary Collection*, while I*R* is a database named *Intermediate Results database*.

Each operator starts from a given query process state and generates a new query process state. During the process, the J-CO-QL Engine, can be asked (by a suitable operator) to store *tc* (the *Temporary Collection*) into IR (the *Intermediate Results database*), that could be taken as input by a subsequent operation. Obviously, J-CO-QL provides an operator to store the temporary collection into a persistent database, (a database managed by MongoDB) as well as an operator to get a collection from IR or from a persistent database as new temporary collection. In fact, the idea is that an operator takes the temporary collection as input and generates a new instance of the temporary collection as output. The reader can see an execution trace in Figure 6, depicting the execution process of the query presented in Section 4.1.

The J-CO-QL Engine executes each query process in isolation: several users can use the engine at the same time. Thus, the goal of the IR database, one for each query process, is twofold. First of all, it permits to temporarily store intermediate results of the process, that do not have to be stored in persistent database (since they are intermediate). Second, it ensures isolation of query process execution as far as intermediate results are concerned.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

Collection Quartiers: [

```
{"ID ": 74, "Name": "SACCO",
 "~geometry": {"type": "MultiPolygon",
               "coordinates":  [  [  [  [  9.121949242919204,
45.516020899111012     ],     ,…,     [     9.121949242919204,
45.516020899111012 ] ] ] ] } },
{"ID ": 82, "Name": "COMASINA",
 "~geometry": {"type": "MultiPolygon",
               "coordinates":  [  [  [  [  9.168870308198338,
45.523965029425476     ],     …,     [     9.168870308198338,
45.523965029425476 ] ] ] ] } },
…]
```

Collection Pharmacies: [

```
{"Address": "Via ANGELONI LUIGI",
 "Name": "COMUNALE N.33",
 "~geometry": {"type": "Point",
               "coordinates": [ 9.17529749573575,
                                45.527028643302899 ] } },
{"Address": "Via CASARSA 130",
  "Name": "CASARSA",
   "~geometry": {"type": "Point",
                 "coordinates": [ 9.174392445342329,
                                  45.524802296955897 ]} },
…]
```

Figure 2. Excerpt of collection Quartiers and collection Pharmacies

4. GENERATING OPEN DATA CONCERNING GEO-REFERENCED CONTENTS

In this section, we practically show the effectiveness of our framework. We consider real data sets coming from the Open Data portal (url: `https://dati.comune.milano.it/`) of Milan (Italy) City Council.

Suppose we own an information system having three MongoDB servers, whose toy IP addresses are 10.0.0.11, 10.0.0.12 and 10.0.0.13. The first one, with address 10.0.0.11, has a database named `Boundaries`, that contains collections concerning some cities; we say that collection `Milan_Quartiers` contains JSON objects describing quartiers in Milan. The upper part of Figure 2 shows a few objects describing quartiers.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

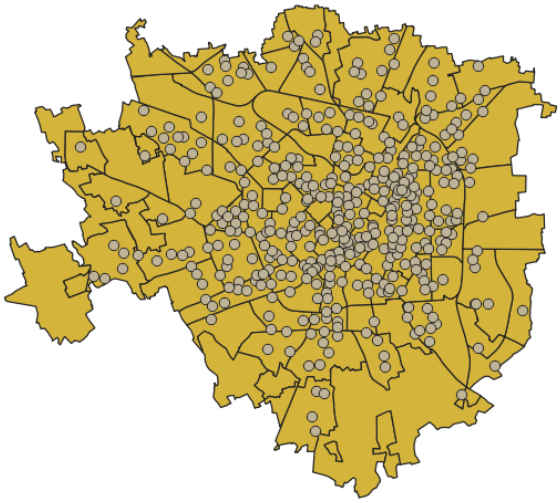

Figure 3. Quartiers and Pharmacies in Milan (Italy)

Server with address 10.0.0.12 manages a database named `MilanInfo`: it contains collections related with territory. As an example, consider collection `Pharmacies`, that describes pharmacies in Milan. We report a few objects in collection `Pharmacies` in the lower part of Figure 2.

Finally, server 10.0.0.13 manages the database named `dbToPublish` that has to store collections containing GeoJSON layers to publish as open data.

The sample process we show in the rest of this section has the following goals:

- Count, for each quartier, the number of pharmacies in the quartier.
- Create two GeoJSON layers to save into a collection named `NilanQuartiersAndPharmacies` and store it in database `dbToPublish`: one layer is named `"Few Pharmacies"` and describes quartiers having less than 2 pharmacies; the other layer is named `"Many Pharmacies"` and describes all other quartiers.

Figures 3 and 7 graphically illustrate the process. We start from the descriptions of quartiers (brown-filled polygons) and pharmacies (grey points). We want to obtain the two layers jointly depicted in Figure 7: red-filled polygons are quartiers with less than 2 pharmacies; green-filled polygons are quartiers with at least 2 pharmacies. The two main steps of the process are described in Sections 4.1 and 4.2, where J-CO-QL queries able to perform the process are presented.

4.1. Counting Pharmacies

The J-CO-QL query for performing the first task, i.e., counting pharmacies in each quartier, is reported hereafter. Afterwards, we will describe single operators and the full process.

```
USE DB Boundaries
ON SERVER MONGODB "http://10.0.0.11:2707";
```

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

```
USE DB MilanInfo
ON SERVER MONGODB "http://10.0.0.12:2707";
USE DB dbToPublish
On SERVER MONGODB "http://10.0.0.13:2707";

SPATIAL JOIN OF COLLECTIONS
   Milan_Quartiers@Boundaries AS Q, Pharmacies@MilanInfo AS P
ON INCLUDED(RIGHT)
SET GEOMETRY LEFT
CASE
WHERE WITH Q.QID, P.Name
GENERATE {.QID: .Q.QID, .Name: .Q.Name, .PharmacyName: .P.Name}
KEEPING GEOMETRY
DROP OTHERS;

GROUP
PARTITION WITH .QID, .PharmacyName
BY .QID, .Name, .~geometry INTO .Pharmacies
DROP OTHERS;

FILTER
CASE WHERE WITH .QID, .Pharmacies
GENERATE {.QID, .Name, .NumOfPharmacies: COUNT(.Pharmacies)}
   KEEPING GEOMETRY;

SET INTERMEDIATE AS QuartiersWithPharmacies;

FILTER
CASE WHERE WITH .QID, .Name
 GENERATE {.ID; .QID,.Name} KEEPING GEOMETRY
DROP OTHERS;

SUBTRACT COLLECTIONS Milan_Quartiers, TEMPORARY;

FILTER
CASE WHERE WITH .QID, .Name
 GENERATE {.ID, .Name, .NumOfPharmacies: 0}
   KEEPING GEOMETRY
DROP OTHERS;
```

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

```
MERGE COLLECTIONS TEMPORARY, QuartiersWithPharmacies;

SET INTERMEDIATE AS QuartiersWithPharmaciesCount;
```

We now describe the query. Notice that the execution trace is depicted in Figure 6.

First of all, it is necessary to specify to which databases to connect. The first three `USE DB` operators do this work. Notice that the `ON` clause specifies the connection string necessary to connect to the desired MongoDB server.

The real procedure starts with the `SPATIAL JOIN` operator. This is the key operator provided by J-CO-QL, in order to perform complex transformations concerned with geo-referenced data. Recall that collection `Milan_Quartiers` describes quartiers in Milan: each object in the collection contains a field named `~geometry`, that describes the boundary of the quartier as a polygon. This collection is aliased as `Q` in the operator. On the other side, collection `Pharmacies` contains objects whose field `~geometry` denotes the point where the pharmacy is located. This collection is aliased as `P` in the operator.
The `SPATIAL JOIN` operator computes pairs of objects in the two collections, such that the spatial join condition specified in the `ON` clause is satisfied. Specifically, a pair of objects is built if the geometry of the right object (in this case, coming from collection `Pharmacies`) is included in the geometry of the left objects (in this case, coming from collections `Milan_Quartiers`). The `SET GEOMTERY` clause specifies the geometry to assign to the object obtained by pairing the two original ones: we specify that we want to maintain the geometry of the left object, i.e., the boundary of the quartier. The upper part of Figure 4 reports an excerpt of the objects resulting from the generation of pairs satisfying the spatial join condition. Notice field `Q` that contains the original object coming from collection aliased ass `Q`, field `P` that contains the original object coming from the collection aliased as `P` and the `~geometry` field resulting from the join (in this case, it coincides with the left geometry, as specified in the operator).
The subsequent `CASE WHERE` clause is necessary to restructure the objects, removing nesting. The `WHERE` selection condition uses the `WITH` predicate, that selects objects having the desired fields; then, the `GENERATE` sub-clause specify how to restructure each object that satisfies the condition; note that we want to maintain the geometry (`KEEPING GEOMTRY` option). The lower part of Figure 4 reports an excerpt of the temporary collection t_1 (as reported in Figure 6) resulting from the `SPATIAL JOIN`; notice how the `CASE WHERE` block restructured the output objects.

At this point, it is necessary to group together objects resulting from the `SPATIAL JOIN` in order to count the number of pharmacies in each quartier.
The `GROUP` operator is, intuitively, similar to the `GROUP BY` clause of SQL. However, it is specifically designed to work with collections of heterogeneous objects. Thus, the goal of the `PARTITION` clause is to select objects (from the t*emporary* collection produced by the previous operator) that have some common fields or characteristics. In the query, we select objects having fields `QID` (quartier identifier) and `PharmacyName` (in other words, we define a partition of the full set of objects); objects in the partition are then grouped on the basis of fields `QID`, `Name` and `~geometry` field, as specified by the `BY` clause. For each identified group of objects, a new object is put into the output collection, such that all common fields are reported and a new field, an array of grouped objects named `Pharmacies` (as specified by the `INTO` clause) is added.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

Within SPATIAL JOIN: and Before CASE WHERE [

```
{"Q": {"QID": 83,
       "Name": "BRUZZANO",
       "~geometry": {"type": "MultiPolygon",
                     "coordinates": […] } },
 "P": {"Address": "Via ANGELONI LUIGI",
       "Name": "COMUNALE N.33",
       "~geometry": {"type": "Point",
                     "coordinates": [ … ] } },
 "~geometry": {"type": "MultiPolygon",
               "coordinates": […]}
},
{"Q": { "QID": 83,
        "Name": "BRUZZANO",
        "~geometry": {"type": "MultiPolygon",
                      "coordinates": […] } },
"P": {"Address": "Via CASARSA 130",
      "Name": "CASARSA",
      "~geometry": {"type": "Point",
                  "coordinates": […]} },
"~geometry": {"type": "MultiPolygon",
                       "coordinates": […] }
…]
```

At the end of SPATIAL JOIN, t_1 : [

```
{"QID": 83, "Name": "BRUZZANO",
 "pharmacyName": "COMUNALE N.33",
"~geometry": {"type": "MultiPolygon",
              "coordinates": […] }},
{"QID": 83, "Name": "BRUZZANO",
 "PharmacyName": "BRUZZANO",
 "~geometry": {"type": "MultiPolygon",
               "coordinates": […] }},
…]
```

Figure 4. Excerpt of objects generated by SPATIAL JOIN

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

At the end of GROUP, t_2: [

```
{"QID": 83, "Name": "BRUZZANO",
 "Pharmacies": [
{"QID": 83, "pharmacyName": "COMUNALE N.33",
 "~geometry": {"type": "MultiPolygon",
                        "coordinates": […] }},
{"QID": 83, "PharmacyName": "BRUZZANO",
 "~geometry": {"type": "MultiPolygon",
                        "coordinates": […] }}],
"~geometry": {"type": "MultiPolygon",
                        "coordinates": […] }}, },
…]
```

Temporary collection t_3 and Intermediate Collection QuartiersWithPharmacies: [

```
{"ID": 83, "Name": "BRUZZANO",
 "NumOfPharmacies": 2},
…]
```

Figure 5. Excerpt of objects generated by the objectys generated by the first GROUP operator and in collection saved into the QuartiersWithPharmacies IR database.

Note the presence of the ~geometry field in the BY clause: this is necessary to avoid the loss of geometry of quartiers during grouping (the geometry is implied by the quartier identifier). An excerpt of the temporary collection t_2, as numbered in Figure 6, is reported in the upper part of Figure 5.

At this point, it is necessary to add a field witch counts how many elements are present in array Pharmacies. The FILTER operator selects the desired objects and restructures them by adding the field named NumOfPharmacies. The lower part of Figure 5 reports the temporary collection t_3, as numbered in Figure 6, resulting from the FILTER operator.

The temporary collection is saved with name QuartiersWithPharmacies into the Intermediate Results Database. It will be used a few operators later. Notice in Figure 6 that the temporary collection in the state produced by SET INTERMEDIATE operator does not change (it is still labelled as t_3). In contrast, the IR database, depicted with empty braces in previous states, now contains the new saved collection.

Some quartiers may have not been produced by the spatial join, i.e., those quartiers without pharmacies. To restore them, we subtract quartiers with pharmacies from the full set of quartiers. Preliminarily, it is necessary to make the structure of objects in the temporary collection homogeneous with that of objects in collection Milan_Quartiers, with a FILTER operator that removes field NumOfPharmacies.
Then, the SUBTRACT operator performs the set-oriented difference between objects in

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

collection `Milan_Quartiers` and the temporary collection (that actually contains quartiers with at least one pharmacy). Since only quartiers without pharmacies survive the difference, the next `FILTER` operator adds the missing `NumOfPharmarcies` field (set to 0). Finally, the `MERGE` operator unites the objects in the temporary collection and in collection `QuartiersWithPharmacies`, previously saved into the *Intermediate Results database*.

The first part of the process end ssaving the temporary collection into the Intermediate Results database with name `QuartiersWithPharmaciesCount`.

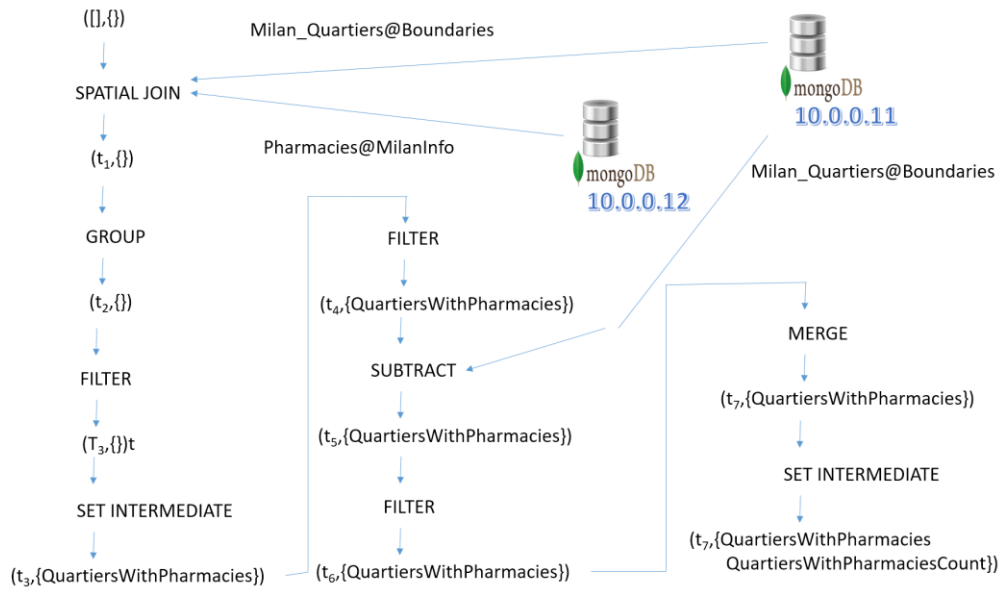

Figure 6. Execution trace for the query performing the first task

4.2. Generating GeoJSON Layers

The J-CO-QL query for performing the second task, i.e., generating the desired GeoJSON layers, is reported hereafter.

```
GET COLLECTION QuartiersWithPharmaciesCount;

FILTER
CASE WHERE WITH ..ID, .Name, .NumOfPharmacies AND
            .NumOfPharmacies < 2
  GENERATE {.ID, .Name, .LayerName: "Few Pharmacies"}
             KEEPING GEOMETRY
 WHERE WITH ..ID, .Name, .NumOfPharmacies AND
       .NumOfPharmacies >= 2
  GENERATE {.ID, .Name, .LayerName: "Many Pharmacies"}
             KEEPING GEOMETRY
DROP OTHERS;
```

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

```
FILTER
CASE WHERE WITH .ID, .Name, ..NumOfPharmacies, LayerName,
  GENERATE {.type: "Feature",
            .properties:{.ID, .Name,
                         .NumOfPharmacies,
                         .LayerName},
                         .geometry: .~geometry }
    DROPPING GEOMETRY
DROP OTHERS;

GROUP
PARTITION WITH .type, .properties, .geometry,
               .Properties.LayerName
BY .Properties.LayerName INTO .features
DROP OTHERS;

FILTER
CASE WHERE WITH .LayerName, .features
  GENERATE {.tye:"FeatureCollection",
            .name: .LayerName,
            .features}
DROP OTHERS;

SAVE AS MilanQuartiersAndPharmacies@dbBToPublish;
```

In order to generate two GeoJSON layers from objects stored in the intermediate collection `QuartiersWithPharmaciesCount`, we have to perform a sequence of transformations.

First of all, the operator `GET COLLECTION` retrieves the desired collection from the *Intermediate Results* database and makes it the temporary collection.

The subsequent `FILTER` operator adds a new field to objects, named `LayerName`. Notice the two `WHERE` conditions: if an object satisfies the first one, the new field has value `"Few Pharmacies"`; if an object satisfies the second condition, the new field has value `"Many Pharmacies"`. These are the names of the two layers we are going to generate, The second `FILTER` operator is necessary to restructure objects, in order to comply with the structure of GeoJSON features. In particular, notice the field specification `.geometry: .~geometry`, that is necessary to rename the `~geometry` field (required specified by the J-CO-Ql data model) into `geometry`, as required by the GeoJSON format.

At this point, it is possible to generate a layer by aggregating all objects having the same value for field `LayerName`. This is easily performed by the `GROUP` operator, which groups the objects based on the value of field `LayerName` nested within field `properties`.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

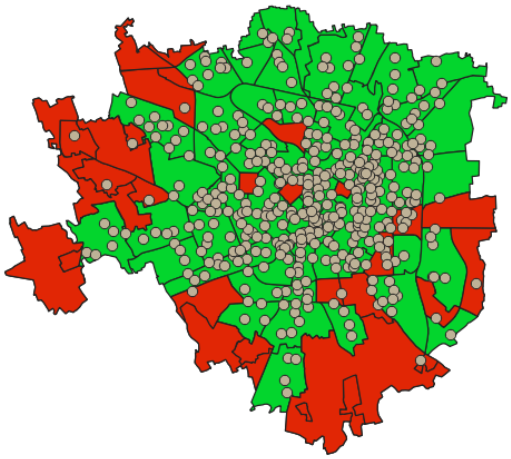

Figure 7. Layers "Few Pharmacies" (in red) and "Many Pharmacies" (in green)

The last FILTER operator is necessary to add the missing field type at the external level and rename field LayerName as name.

The obtained collection is saved into the dbToPublish database, that is managed by server with IP address 10.0.0.13. Hereafter, we report an excerpt of layer "Few Pharmacies", that is depicted in red in Figure 7. Layer "Many Pharmacies" is identical, apart from the described quartiers and the name of the layer.

```
{"type": "FeatureCollection",
 "name": "Few Pharmacies",
 "features": [
   {"type": "Feature",
    "properties": {"ID ": 74, "Name": "SACCO",
                   "LayerName": "Few Pharmacies"},
    "geometry": {"type": "MultiPolygon", "coordinates": [ … ] }
   },
   {"type": "Feature",
    "properties": {"ID": 75, "Name": "STEPHENSON",
                   "LayerName": "Few Pharmacies"},
    "geometry": {"type": "MultiPolygon", "coordinates":  […  ]}
   },
…] }
```

At the end of this section, we want to point out the major results we obtained with J-CO-QL.

- A J-CO-QL query is certainly a procedural specification, but it is not a procedural program, in the sense of classical procedural programming languages.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

- The syntax of operators is English-like, inspired by the same approach adopted for SQL. This way, a certain degree of semantics of operators implicitly is expressed by the syntax.

- We are aware that operators need training to be properly used, but it is more intuitive for non programmers that other languages for JSON objects. For example, the same operations performed with the native query language of MongoDB could result a little bit hard to perform (probably exploiting JavaScript in many case, thus again a programming language).

- JSON collections are heterogeneous, i.e., they can contain objects with different structure. J-CO-QL natively deals with such a situation.

- The language natively deals with spatial operations on geometries. This is an essential feature of the language, very useful for managing data with associated geometries, a more and more frequent situation in the Open Data world.

As far as the J-CO framework is concerned, we think that this section has shown some of the reasons why we decided to devise it.

- The J-CO framework is able to provide a unique environment to integrate data coming from different databases managed by different servers. This feature is essential to integrate data in a seamless way. Given this characteristics it could be a suitable framework towards querying distributed NoSQL databases in a distributed processing infrastructure such as Hadoop or Spark.

- Consequently, the J-CO framework can be considered a milestone toward a flexible framework for building polystore database systems for data science applications.

5. CONCLUSIONS

In this paper, we proposed an innovative framework, named J-CO, for integrating and transforming heterogeneous data sets in the form of collections of possibly geo-tagged JSON objects. The idea is to provide both analysts and geographers with a powerful tool that makes possible to perform complex analysis processes without writing procedural programs, but specifying transformation processes in a high-level way. The framework is founded on a high-level query language, named J-CO-QL, specifically devised to query heterogeneous collections of (possibly) geo-tagged JSON objects. Furthermore, the framework and the query language have been designed to retrieve input collections and to store output collections to several NoSQL databases in a seamless way, allowing analysts to easily integrate collections stored in different databases.

An example is illustrated. We show how it is possible to integrate geo-referenced information concerning quartiers and pharmacies of a city (we considered Milan, Italy) to create two new GeoJSON layers, one reporting quartiers with less than 2 pharmacies and one reporting quartiers with at least two pharmacies. Through the example, we introduced many J-CO-QL operators, briefly describing them. For a more detailed presentation, the reader can refer to [5].

The development of the J-CO framework is ongoing.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

An important issue is to develop a suitable user interface that allows users to write query processes step by step, possibly inspecting the temporary collection and the intermediate results database and, if necessary, backtracking the query. Such a user interface is currently under development.

More ambitiously, we want J-CO-QL to meet the *data independence principle* by defining a shell layer framework of operators for easing the transformation of JSON objects transparently to the user. This is inspired by the concept of mediators, lightweight integration components, deemed to access sources on demand [39]. As defined in [39] "A mediator is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications": sources are encapsulated by wrappers (which access their data sources in a transparent way to mediators) to present data in the form needed by a mediator.
A weakness of the current proposed operators is that users need to be aware of the structure of JSON objects, thus violating the independence of the language from data and forcing to user to be aware of the structure of the data. In our next evolution of the language, we aim at defining two layers of operators: the user-layer consisting of operators directly invoked by users; the hidden layer, consisting of operators automatically invoked by the user–layer operators, whenever it is necessary through the mediators to transform a JSON object to allow its comparison/join with another JSON objects having a different structure.
The final goal of the project is to define a powerful language suitable for integrating and transforming big data concerning territorial and geographical data sets, coming from heterogeneous sources with the less effort as possible to the final user, possibly supporting novel applications such as location-based queries [9, 10].

REFERENCES

[1] M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley and M. Zaharia, "Spark sql: Relational data processing in spark". In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (pp. 1383-1394). ACM. 2015.

[2] R. Battle and D. Kolas, "Geosparql: enabling a geospatial semantic web". Semantic Web Journal, 3(4), 355-370. 2011.

[3] F. Benitez-Paez, A. Degbelo, S. Trilles and J. Huerta, "Roadblocks hindering the reuse of open geodata in Colombia and Spain: a data user's perspective. ISPRS International Journal of Geo-Information, 7(1), 6. 2017.

[4] G. Bordogna, A. Campi, G. Psaila and S. Ronchi, "An interaction framework for mobile web search". In Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia (pp. 183-191). ACM.2008.

[5] G. Bordogna, S. Capelli, D.E. Ciriello, and G. Psaila, , "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, 2017.

[6] G. Bordogna, S. Capelli, and G. Psaila, "A big geo-data query framework to correlate open data with social network geotagged posts," Proceedings in AGILE 2017 international conference, 2017.

[7] G. Bordogna, A. Cuzzocrea, L. Frigerio, G. Psaila and M. Toccu, "An interoperable open data framework for discovering popular tours based on geo-tagged tweets. International", Journal of Intelligent Information and Database Systems, 10(3-4), 246-268, 2017.

[8] G. Bordogna, L. Frigerio, A. Cuzzocrea and G. Psaila, "Clustering geo-tagged tweets for advanced big data analytics". In Big Data (BigData Congress), 2016 IEEE International Congress on (pp. 42-51). IEEE. 2016.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

[9] G. Bordogna, M. Pagani, G. Pasi and G. Psaila, “Evaluating uncertain location-based spatial queries”. In Proceedings of the 2008 ACM symposium on Applied computing (pp. 1095-1100). ACM. 2008.

[10] G. Bordogna, M. Pagani, G. Pasi and G. Psaila, “Managing uncertainty in location-based queries”. Fuzzy sets and systems, 160(15), 2241-2252. 2009.

[11] G. Bordogna, M. Pagani, and G. Psaila, “Database model and algebra for complex and heterogeneous spatial entities,” in Progress in Spatial Data Handling, pp.79–97, Springer, 2006.

[12] F. Burini, N. Cortesi, K. Gotti, and G. Psaila, “The Urban Nexus Approach for Analyzing Mobility in the Smart City: Towards the Identification of City Users Networking”. Mobile Information Systems, 2018.

[13] S. Busse, R.D. Kutsche, U. Leser and H. Weber, “Federated Information Systems: Concepts, Terminology and Architectures”, Forschungsberichte des Fachbereichs Informatik Bericht Nr. 99-9, Technische Universität Berlin, seen at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.8618&rep=rep1&type=pdf the 01-09-2018. 2018.

[14] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub, “The GeoJSON format,” tech. rep., 2016.

[15] R. Cattell, “Scalable SQL and NoSQL data stores,” SIGMOD Record, vol.39 (4), pp.12–27, 2011.

[16] D. Chamberlin, J. Robie, D. Florescu, “Quilt: an XML query language for heterogeneous data sources”, e Int. Workshop on the Web and Data Bases (WebDB), 53-62, 2000.

[17] T.E. Chow, “Geography 2.0: A mashup perspective, “Advances in web-based GIS, mapping services and applications”, pp.15–36, 2011.

[18] A. Cuzzocrea, **G.** Psaila and M. Toccu, “Knowledge discovery from geo-located tweets for supporting advanced big data analytics: A real-life experience”. In Proceedings of MEDI-2015 Int. Conf. on Model and Data Engineering (pp. 285-294). Springer, 2015.

[19] A. Cuzzocrea, G. Psaila, and M. Toccu, “An innovative framework for effectively and efficiently supporting big data analytics over geo-located mobile social media,” Proceedings of the 20th International Database Engineering & Applications Symposium, pp.62–69, ACM, 2016.

[20] C. Doulkeridis and K. Nørvåg, “A survey of large -scale analytical query processing in MapReduce”, The VLDB Journal, 1-26, 2013 .

[21] J. Duggan, A.J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner and S. Zdonik, “The BIGDAWG polystore system”. ACM Sigmod Record, 44(2), 11-16. 2015.

[22] J. H. Feng, Q. Qian, Y.G. Liao, G.L. Li, N. Ta and L.Z. Zhou, “Survey of Research on Native XML Databases”, Application Research of Computers, 6, 1-7. 2006.

[23] U. Gandhi, “QGIS tutorial and Tips”, Last updated on Apr 30, 2018. seen the 01-09-2018 at https://www.qgistutorials.com/en/index.html#. 2018.

[24] G. Gou and R. Chirkova, “Efficiently querying large XML data repositories: A survey”. IEEE Transactions on Knowledge and Data Engineering, 19(10).2007.

[26] V. Goyal and D. Soni, “Survey paper on Big Data Analytics using Hadoop Technologies”, Int. J. of Current Engineering and Scientific Research (IJCESR) ISSN (PRINT): 2393-8374, (ONLINE): 3(7), 2394-0697. 2016.

[26] J. Han, E. Haihong, G. Le, and J. Du, “Survey on NoSQL database”, Pervasive computing and applications (ICPCA), 2011 6th international conference on, pp.363–366, IEEE, 2011.

[27] S.C. Haw and C.S. Lee, “Data storage practices and query processing in XML databases: A survey”. Knowledge-Based Systems, 24(8), 1317-1340. 2011.

[28] D. Heimbigner and D. McLeod, “A federated architecture for information management”. ACM Transactions on Information Systems (TOIS), 3(3), 253-278. (1985).

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304

[29] G. Hubert, G. Cabanac, C. Sallaberry, and D. Palacio, Query Operators Shown Beneficial for Improving Search Results, in TPDL'11: Proceedings of the 1st International Conference on Theory and Practice of Digital Libraries . Sous la dir. de S.Gradmann, F. Borri, C. Meghini and H.Schuldt .T. 6966. LNCS., p. 118-129. Springer, 2011.

[30] L.A. Kurgan and P. Musilek, "A survey of Knowledge Discovery and Data Mining process models". The Knowledge Engineering Review, 21(1), 1-24. 2006.

[31] W. Litwin, L Mark and N. Roussopoulos, "Interoperability of multiple autonomous databases". ACM Computing Surveys (CSUR), 22(3): 267 –293. 1990.

[32] M. Lupp, OGC Web Services, Encyclopedia of GIS 2008 Edition, Editors: Shashi Shekhar, Hui Xiong . 2008.

[33] R. Meo and G. Psaila, "An XML-based database for knowledge discovery". In International Conference on Extending Database Technology (pp. 814-828). Springer, Berlin, Heidelberg. 2006.

[34] A. Nayak, A. Poriya, and D. Poojary, "Type of nosql databases and its comparison with relational databases," International Journal of Applied Information Systems, vol.5, no.4, pp.16–19, 2013.

[35] K.W. Ong, Y. Papakonstantinou, and R. Vernoux, "The sql++ unifying semi-structured query language, and an expressiveness benchmark of sql-on-hadoop, nosql and newsql databases," CoRR, abs/1405.3631, 2014.

[36] G. Psaila, "A database model for heterogeneous spatial collections: Definition and algebra," Data and Knowledge Engineering (ICDKE), 2011 International Conference on, pp.30–35, IEEE, 2011.

[37] H. Robin and S. Jablonski, "Nosql evaluation: A us case oriented survey," CSC-2011 International Conference on Cloud and Service Computing, Hong Kong, China, pp.336–341, December 2011.

[38] M.Y Vardi, M. Y., "A theory of regular queries". In Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (pp. 1-9). ACM. 2016.

[39] G. Wiederhold, "Mediators in the architecture of future information systems". Computer, 25(3), 38-49. 1992.

[40] T. White, "Hadoop: The definitive guide", O'Reilly Media, Inc.", 2012.

Authors

Gloria Bordogna is researcher at CNR-IREA (Italy). Her research activity mainly concerns the representation and management of imprecision and uncertainty within information retrieval systems (IRSs) database management systems (DBMSs) and Geographic Information Systems (GIS), soft computing.

Giuseppe Psaila is researcher and professor at University of Bergamo (Italy). He works on many topics concerning data management, such as data mining, XML processing, query languages and soft computing, information retrieval, Big Data and Open Data.

Natarajan Meghanathan et al. (Eds) : DaKM, SIPP, CCSIT, NCWMC - 2018
pp.183–202, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81304