J-CO-QL: A Flexible Query Language for Complex Geographical Analysis of Heterogeneous Geo-tagged JSON Data Sets

Steven Capelli, Paolo Fosci, Fabio Marini and Giuseppe Psaila

Abstract—Analysts that wish to perform geographical analysis are provided with large volumes of publicly available geo-tagged data sets. Often, these data sets are published by public administrations as *Open Data*, and are formatted as JSON objects. Furthermore, these JSON data sets are also heterogeneous, in terms of format and structure, even though they describe the same territorial entities. So far, analysts need new powerful tool to easily perform complex geographical analysis on large collections of geo-tagged JSON data, possibly stored in NoSQL databases.
In this paper, we introduce the *J-CO-QL*: it provides a set of high level operators able to operate on heterogeneous collections of JSON objects, explicitly dealing with geometry and providing advanced spatial aggregation and comparison capabilities. *J-CO-QL* relies on an intuitive execution model that permits to write complex queries; operators are designed to be high-level operator, based on a clear database vision of the problem and suitable for non programmers.

Index Terms—Colelctions of JSON objects, Geo-tagged data sets, Query Language for geographical analysys, Powerful spatial operators.

1 Introduction

An impressive amount of information concerning territories has (and is) made publicly available on the Internet. Very often, this information is published on *Open Data* portals, where public administrations publish data sets concerning several aspects of territories and citizenship. In spite of the wide use of

JSON ([1]) or CSV as general format, every single data set has a specific structure, with specific field names, even though they describe similar topics. Obviously, these data sets often contain geo-referenced information.

Other sources of (possibly geo-tagged) information could be descriptions of networks (such as water networks, electricity networks, etc.) and environment descriptions (streets, buildings, etc.), that might be cross processed to discover useful information, or integrated with Open Data sets.

NoSQL databases [2], [3], [4] are emerging as a key technology, because traditional relational/SQL databases are unable to flexibly integrate so heterogeneous data sets. Nowadays, the most famous NoSQL DBMS is *MongoDB* [5], [6]: it stores collections of heterogeneous JSON objects, with no limitations to the variety of structure of objects gathered within the same collection.

Beside NoSQL databases, researchers are extending relational technology as well, in order to store and query JSON documents within textual and bloc attributes in tables [1], [7], thus obtaining a kind of hybrid solution, i.e., managing schema-less documents within a totally structured database.

In parallel, query languages are evolving as well, in order to be able to deal with JSON documents. Apart from raw query languages provided by some NoSQL databases (like the one provided by MongoDB), some interesting proposals appeared in literature, such as [8], where the basic and well known syntactic structure of SQL is extended with constructs able to query JSON documents.

However, these languages do not deal at all with geo-tagging of JSON objects. Furthermore, they are not suitable to specify complex analysis tasks, that require to build and aggregate several intermediate results, maintaining a clear database vision of the problem, as well as dealing with high level of heterogeneity within collections.

In our opinion, analysts working on the integration and cross analysis of heterogeneous and geo-tagged data are facing a gap between their needs and currently available processing technology.[1] In particular, they need tools, possibly integrated within a GIS for visualization purposes, that permit to easily perform complex data analysis tasks on highly heterogeneous and geo-tagged data sets.

The goal of our research work is to fill in such a gap, that can be expressed as the inability of GISs to flexibly query

- *S. Capelli is with the DISCo Department of Università degli studi di Milano Bicocca, Viale Sarca 336 - 20126 Sesto San Giovanni (MI) - Italy E-mail: steven.capelli@unimib.it*
- *P. Fosci is with the DIGIP Department of University of Bergamo, Viale Marconi 5 - 24044 Dalmine (BG) - Italy E-mail: paolo.fosci@unibg.it*
- *F. Marini works for GN Informatica, Calusco dAdda (BG) - Italy E-mail: fabio.marini@gninformatica.com*
- *G. Psaila is with the DIGIP Department of University of Bergamo, Viale Marconi 5 - 24044 Dalmine (BG) - Italy E-mail: giuseppe.psaila@unibg.it*

1. As an example, the reader can think about geo-coding activities, such as associating points with zip codes, an activity that with relational database extended with geographical functions (PotgreSQL/PostGIS, for example) could require several ours or days of work.

and transform heterogeneous data sets of geo-tagged JSON objects.

The perspective scenario is depicted in Figure 1: we are developing a framework, named *J-CO-QL Framework* (*J-CO-QL* stands for *J*SON *CO*llections *Q*uery *L*anguage), designed to enrich a GIS as a plug-in and provide an engine to execute queries on top of MongoDB (currently) as well as other NoSQL DBMS (in the future) able to store large volumes of JSON documents. This way, an analyst will be able to operate within the well known user interface of her/his favourite GIS, visually analysing geo-tagged JSON data sets, writing complex queries and transformation processes that will be executed on the data sets, visualizing the results.

In this paper, we present *J-CO-QL*, the query language of our framework.

J-CO-QL provides a pool of operators for various data transformation tasks on collections of JSON objects (possibly geo-tagged with GeoJSON [9], [10]). In particular, we defined specific spatial operations on the geometric fields of JSON objects. The main design goals of J-CO-QL are:

1) dealing in a native way with geometrical features of objects;
2) easily managing highly heterogeneous collections;
3) providing a clear semantics of the query process, on the basis of a database vision;
4) providing high-level operators, so far abstracting from the programmer vision typical of some other proposals.

In this paper, we present the basic and minimal operators we consider necessary in J-CO-QL to meet our design goals, for which the data model and the execution model are devised. We will show the language and its use by means of a running example. In the future, we will introduce other operators, specifically designed to address specific, yet complex, geographical analysis problems.

During the design of the language, we were inspired by early works [11], [12] about a query language for heterogeneous, although structured, collections of geo-referenced data.

The paper is organized as follows. in Section 2, we present the data model and a toy running example that will be exploited along with the paper. Section 3 presents the execution model on which J-CO-QL relies. At this point, Section 4 presents basic notations and the basic building blocks (action and clauses) of J-CO-QL; after the presentation of basic operators in Section 5, the **SPATIAL JOIN** operator is presented in Section 6, the fundamental operators necessary to write complex query processes are presented in Section 7 and some useful set-oriented operators are introduced in Section 8. A complete example is presented in Section 9, to illustrate the potential application of J-CO-QL. Section 10 presents the implementation of the prototype and discuss some experimental results. Finally, Section 11 discusses related works and Section 12 draws the conclusions.

2 Data Model

The basic concept which we rely on is the *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), vectors (of numbers, strings, objects).

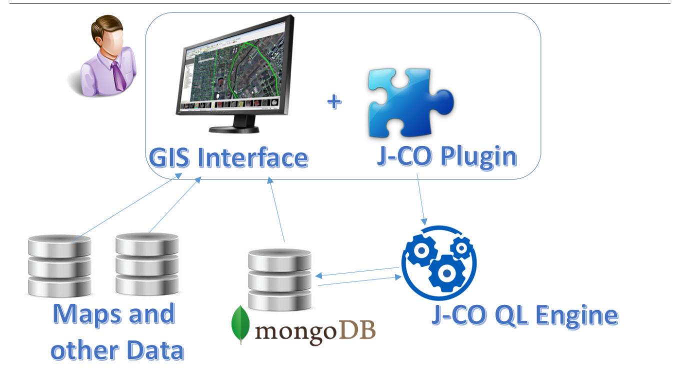

Fig. 1. Perspective Application of the J-CO-QL Framework.

As far as spatial representation is concerned, we rely on the GeoJSON standard [9], [10]). In particular, we assume that the geometry is described by a field named *geometry*, defined as a *GeometryCollection* objects type in GeoJSON standard. The absence of this top-level field means that the object does not have an explicit geometry.
As an example, consider the object with name `"buildingA"` reported in Listing 1. The `geometry` field describes the polygon representing the footprint of the building on the ground.

The following definition defines 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 *InstanceOf(c)*= $[o_1, \ldots, o_m]$ that is a vector of JSON objects o_i.

Thus, we need operators (see Section 3) to transform collections and get new collections. Our language should satisfy the *closure property*.

Example 1 (Running Example). In order to illustrate the data model and, in the next sections, the execution model and the J-CO-QL operators, we provide a running example.
Suppose we have a sample database named **ToyDB**. Within it, We have three toy collections: the first one is named `Buildings` (shown in Listing 1); the second one is named `WaterLines` (see Listing 2). Finally, the third one is named `Restaurants` (see Listing 3).
In particular, collection **Buildings** contains two objects, describing two buildings. Each building is described by its name, the city name, the address and the geometrical description (field **geometry**). Notice that while **buildingA** has an field named **city**, the corresponding one in **buildingB** is named **cityName**. This is an example of heterogeneous objects objects that describe homogeneous real world entities.
Collection **WaterLines** describes lines for water distribution. this is not heterogeneous and each line is described by the city and the geometrical representation (poly-line) of the network.

Finally, collection **Restaurants** is an heterogeneous collection: in fact, the three objects share three descriptive fields (name, city and address), but the first one has no more fields, while the second object has two numerical fields named **lat** and **lng**, the third object has the **geometry** field.

Listing 1. Collection **Buildings**

```
[{ "name":"buildingA",
   "city":"city A",
   "address":"address A",
   "geometry":{"type":"GeometryCollection",
                 "geometries":[
                  {
                   "type": "Polygon",
                   "coordinates": [
                     [[100.0, 0.0],
                      [101.0, 0.0],
                      [101.0, 1.0],
                      [100.0, 1.0],
                      [100.0, 0.0] ]]
                  }

              ]

         }

    },

{ "name":"buildingB",
  "cityName":"city B",
  "address":"address B",
  "geometry":{"type":"GeometryCollection",
                "geometries":[
                  {
                   "type": "Polygon",
                   "coordinates": [
                     [[20.0, 0.0],
                      [21.0, 0.0],
                      [21.0, 1.0],
                      [20.0, 1.0],
                      [20.0, 0.0] ]]

                     }
                 ]
           }
    }]
```

Listing 2. Collection **WaterLines**

```
[{ "name":"WaterLineA",
   "city":"city A",
   "geometry":{"type":"GeometryCollection",
                 "geometries":[
                   {
                    "type": "LineString",
                    "coordinates":
                        [[90.0, 0.0],
                         [103.0, 1.0],
                         [104.0, 0.0],
                         [105.0, 1.0]]
                   }
              ]
           }
},
{ "name":"WaterLineB",
  "city":"city A",
  "geometry":{"type":"GeometryCollection",
                 "geometries":[
                   {
                    "type": "LineString",
                    "coordinates":
                      [[102.0, 10.0],
                       [103.0, 2.0],
                       [104.0, 1.0],
                       [102.0, -1.0]]
                 }
               ]
             }
},
{ "name":"WaterLineC",
  "city":"city C",
  "geometry":{"type":"GeometryCollection",
                 "geometries":[
                   {
                    "type": "LineString",
                    "coordinates":
                      [[104.0, 10.0],
                       [105.0, 2.0],
                       [109.0, 1.0],
                       [110.0, -1.0]]
                   }
                ]
             }
}]
```

Listing 3. Collection **Restaurants**

```
[{ "name":"RestaurantA",
   "city":"city A",
   "address":"address A"
 },

 { "name":"RestaurantB",
   "city":"city B",
   "address":"address C"
   "lat": -10.574228,
   "lng": 21.015217
 },

 { "name":"RestaurantC",
   "city":"city C",
   "address":"address D",
   "geometry":{"type":"GeometryCollection",
                "geometries":[
                  {
                   "type": "Polygon",
                   "coordinates":[
                      [[20.0, -10.0],
                       [21.0, -10.0],
                       [21.0, -11.0],
                       [20.0, -11.0],
                       [20.0, -10.0]]]
          }
       ]
    }
}]
```

3 EXECUTION MODEL

Queries will transform collections stored in databases (for example, managed by MongoDB), and will generate new collections that will be stored again into these 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 Results 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, 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 set 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 collection 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 results database $s_0.IR$. Thus, J-CO-QL must provide operators able to start the computation, taking collections from the persistent databases, while other operators carry on the process, continuously transforming the temporary collection and possibly saving it into the persistent databases. But the query could be complex and composed by several subtasks, thus the temporary collection could be saved into the intermediate results database IR. At this point, a new subtask can be started by the same operators that can start the query, which can take collections either from persistent databases or from the intermediate result database as input, giving rise to a new subtask.

For this reason, we identified two classes of operators (see Table 1): *start operators* and *carry on operators*, that will be described in the next sections.

TABLE 1
Type of operators

Family	Operators
Start	GET COLLECTION INTERSECT COLLECTIONS JOIN OF COLLECTIONS MERGE COLLECITONS SPATIAL JOIN OF COLLECTIONS SUBTRACT COLLECTIONS
Carry-on	EXPAND FILTER GROUP SAVE AS SET INTERMEDIATE AS

3.0.0.1 Rationale.: Two points are innovative in the design of the execution model, w.r.t. standard approaches to query languages: the intermediate result database and the procedural favour.
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. Where to store such intermediate results? It is inappropriate to store them into the same DBs that contain initial and target collections. In contrast, the adoption of the intermediate result database forces analysts to explicitly set intermediate results, without touching regular DBs; furthermore, intermediate collections are clearly stated to be intermediate, and disappear from the system at the end of the process.
Moreover, the fact that the intermediate results database IR is part of query process states implicitly guarantees *isolation*: it exists only during the query process and in case of parallelism, each parallel process has its own intermediate results database.

The procedural flavour is the result of the applications of operators: operators 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.
Observe that this is not a new concept in databases: the classical notion of *Transaction* is, in fact, a query process where operators are applied sequentially, storing results into the database. Differently from other approaches, the J-CO-QL execution model makes it explicit.

4 J-CO-QL Basic Blocks

This section is devoted to introduce the basic blocks that are exploited in many J-CO-QL operators. In some sense, they constitute the skeleton of the language. Before we present them, it is necessary to introduce notation and terminology.

4.1 Notation and Terminology

Hereafter, we introduce terminology and symbols that we will use in the rest of the paper.

4.1.0.1 Syntax.: Regarding the notation for the syntax of operators, we will make use of the $\star$ symbol to denote 0 or more repetitions and of the $+$ symbol to denote 1 or more repetitions; square brackets denote optionality; the vertical bar | separates alternatives.

4.1.0.2 Collections and databases.: With *collectionReference*, we denote a reference to a collection stored in a database. Its syntax is:

collectionName[@*dbName*] [*AS collectionAlias*]

where *dbName* is the persistent database that contains the collection, *collectionName* is the name of the desired collection, *collectionAlias* is an alias name given to the collection (if not specified, the default value for *collectionAlias* is *collectionName*). Notice that *dbName* is optional: if not specified, the collection is looked for within the intermediate result database *IR*.

4.1.0.3 Field Reference.: A *fieldReference* permits to refer to fields within JSON objects. Its syntax is

(.*fieldName*)**+**

i.e., a dotted list of field names. The dot at the beginning means that we start from the root of the object, i.e., the most external level of the object.

For example, **.a**, **.a.a3** and **.a.a3.a31** are three samples of *fieldReference*.

4.2 Object Generation

J-CO-QL operators manipulate JSON objects. For this reason, many operators change the structure of objects when they insert objects into the output temporary collection. This is done by the **GENERATE** action. In the rest of the paper, we will refer to it as *generateAction*.

The syntax of the **GENERATE** action is reported hereafter.

```
GENERATE ( objectStructure [geometricOption]
         | geometricOption )
```

where *objectStructure* specifies the structure of the generated object, that might be followed by *geometryOption*, that specifies how and whether the **geometry** field is generated in the output object. If the *geometryOption* is not specified, this means that the **geometry** field is kept as it is from the input to the output object (if missing in the input, it will not appear in the output object). Notice that the clause cannot be empty: if the *objectStructure* is missing (meaning the the structure of the input object is not changed), a *geometryOption* must be specified.

4.2.0.1 Object Structure Definition.: *objectStructure* is a non-empty list of *outputFieldSpec* within braces, i.e.,

{ *outputFieldSpec* (, *outputFieldSpec*)**⋆** }

In its simplest form, an *outputFieldSpec* can be a *fieldReference*: in this case, the new field is given the most internal name of the referenced field (e.g., if **.a.a3.a31** is specified, the new field name is **a31**).

The second form for *outputFieldSpec* can be a field name followed either by a constant value (e.g., **dn: 'John'**), or by an *objectStructure* (to specify nested objects), or by a *fieldReference* (e.g., **db: .b.b2**).

Notice that a *fieldReference* takes the full value (included nested objects and arrays) of the field. However, if the field is not present, the null value is given to the new field.

4.2.0.2 Geometric Option Specification.: *geometryOption* specifies how and whether the output **geometry** field must be generated into the output object. Three options are possible:

1) **KEEPING GEOMETRY** means that the **geometry** field in the input object is translated into the output object as it is (the same if missing). When the *geometryOption* is not specified, this is the default setting.
2) **DROPPING GEOMETRY** means that the **geometry** field will not appear in the output object (if present in the input object, it is drooped).
3) **SETTING GEOMETRY** *geometryFunction* means that the **geometry** field appears in the output object and is obtained as specified by *geometryFunction*.
   As far as *geometryFunction* is concerned, currently we have identified three alternatives (but we foresee to add other functions in the future).

   - **POINT(*latFieldRef*,*longFieldRef*)** takes two numerical fields denoting, respectively the latitude and the longitude of a point, and generates the corresponding GeoJSON representation. *latFieldRef* and *longFieldRef* are both *fieldReference* to fields in the input object.
   - **AGGREGATE(*arrayFIeldReference*)** looks for **geometry** fields appearing in objects listed within the array field present in the input object (as specified by *arrayFIeldReference*, that is a *fieldReference*), and aggregates them into a unique GeoJSON representation.
   - *fieldRefeference* takes the specified field in the input object. If it is a geometrical, GeoJSON compliant, field, it is taken as geometry of the output object; in the contrary, the output **geometry** field is missing.

Example 2. Consider the following JSON object

```
{
 "a": {
       "a1": 1,
       "a2": 2,
       "a3": {
              "a31": 90.000022,
              "a32": 10.000001
             }
      },
 "b": {
       "b1": 'bb',
       "b2": [1, 2, 3]
      },
 "c": 'CC'
}
```

Consider now the following **GENERATE** clause:

```
GENERATE {.c,
             d: { da: {d1: .a.a1, d2:
  .a.a2},
                  db: {.b.b2, dn: 'John'},
             e: .b, .g}
  SETTING GEOMETRY POINT(.a.a3.a31,
  .a.a3.a32)
```

The **GENERATE** action will generate the output object hereafter reported.

```
{
 "c": 'CC',
 "d": {
       "da": {
             "d1": 1,
             "d2": 2,
             }
        "db": {
              "b2": [1, 2, 3],
              "dn": 'John'
             }
      },
 "e": {
       "b1": 'bb',
       "b2": [1, 2, 3]
      },
 "g": null,
 "geometry":{"type":"GeometryCollection",
             "geometries":[
                {
                 "type": "Point",
                 "coordinates":
                    [10.000001, 90.000022]
                 }]
            }

}
```

Notice that both fields **c** and **g** are generated by simple *fieldReference* specifications (**.c** and **.g**, respectively), while the other fields are generated by full field specifications. Finally, notice that the value of field **g** is **null**, since there is no field named **g** in the upper level of the input object.

As far as the **geometry** field is concerned, notice that it was not present in the input object. It is derived by means of the **POINT** function, that takes values of fields **.a.a3.a31** and **.a.a3.a32** as latitude and longitude, respectively, deriving the corresponding GeoJSON representation associated to field **geometry** in the output object.

4.3 CASE Clause

The heterogeneity of JSON collections asks for a clause that is able to manage separately objects with different structure, possibly applying the **GENERATE** action (see Section 4.2) in different ways. For this reasons, many operators defined in J-CO-QL make use of the **CASE** clause (in the syntax of operators referred to as *caseClause*).

Its syntax is the following:

```
CASE
 (WHERE selectionCondition
   [generateAction])+
 (KEEP OTHERS|DROP OTHERS);
```

i.e., the **CASE** keyword is followed by a non-empty list of **WHERE** branches, followed by the **KEPP OTHERS**-**DROP OTHERS** option.

The role of a **CASE** clause is the following: given a set of objects, they are filtered based on the conditions expressed in the **WHERE** branches and possibly transformed into new objects. When an object satisfies a *selectionCondition*, its structure can be modified by using the *generateAction* (see Section 4.2) or can remain unchanged, if the optional *generateAction* is not specified

The *selectionCondition* is a classical boolean condition, enriched with two predicates:

- **WITH** predicate, that is true when an object presents the specified *fieldReference*s;
- **WITHOUT** predicate, that is true for objects without the specified *fieldReference*s.

4.3.0.1 WITH predicate.: The syntax of this predicate is

WITH [*typeSelector*] *fieldReference* (, *fieldReference*)*

where, in the simplest form (without *typeSelector*), it is true when all the listed *fieldReference*s match a field in the input object.

When present, *typeSelector* specifies the data type all fields specified by the list of *fieldReference*s must have. We identified the following options for *typeSelector*:

- **SIMPLE**, fields must be simple-value fields;
- **COMPLEX**, fields must be complex (object) fields;
- **ARRAY**, fields must be arrays;
- **STRING**, fields must be strings;
- **NUMBER**, fields must be numerical values;
- **INTEGER**, fields must be integer values;
- **FLOAT**, fields must be floating point values;
- **GEOMETRY**, fields must be GeoJSON-compliant geometries.

4.3.0.2 WITHOUT predicate.: The syntax of this predicate is

WITHOUT *fieldReference* (, *fieldReference*)*

which is evaluated as true when all the listed *fieldReference*s do not match a field in the object.[2]

WHERE branches are cascaded each other (remind that there can exist many **WHERE** branches with different *selectionCondition*); when an object satisfies more than one *selectionCondition*, only the first matching condition in the sequence is considered.

The final mandatory option **KEEP OTHERS** and **DROP OTHERS** specifies what to do with objects that do not match any **CASE** branch. If **KEEP OTHERS** is specified, these objects are put into the output temporary collection, keeping their structure as it was in input; if **DROP OTHERS** is specified, these objects are not put into the output collection.

Example 3. Consider the following example which aims at showing a possible use of the **CASE** clause.

```
CASE
 WHERE WITH .geometry
  GENERATE KEEPING GEOMETRY
 WHERE WITHOUT .geometry AND
       WITH .a.a1, .a.a2, .c, .b.b1, .b.b2
  GENERATE
   {.c, d: { da: {d1: .a.a1, d2: .a.a2},
    db: {.b.b2, dn: 'John'}, e: .b, .g} }
   SETTING GEOMETRY
          POINT(.a.a3.a31, .a.a3.a32)
 DROP OTHERS
```

2. In JSON, a missing field is equivalent to a field having null value. In J-CO-QL, we comply with this semantics.

The first **WHERE** branch works on all input objects which have the **geometry** field and outputs them as they are (through the use of the **GENERATE** action).

The second **WHERE** branch, instead, works on all input objects which have not the **geometry** field and have the fields listed in the **WITH** clause. The **GENERATE** action modifies the structure of the input objects as shown in Example 2.

Finally, the **DROP OTHERS** option drops all input objects which do not meet any **WHERE** branch selection condition.

Notice that predicates **WITH** and **WITHOUT** are necessary, in order to query the schema of each single object and apply the proper selection condition or transformation. This idea was argued in the work [1]: authors observe that query languages designed to query JSON objects must provide constructs to query the schema of single objects; this is necessary to make the language able to adapt to the specific structure of single objects in the collections.

5 BASIC OPERATORS

In this section, we introduce the basic operators that allow to start and terminate a query process

5.1 *GET COLLECTION*

The **GET COLLECTION** operator is a *start* operator that gets a collection from a database (persistent or intermediate) and makes it the new temporary collection. The syntax of the operator is:

```
GET COLLECTION collectionReference;
```

collectionRefference specifies the collection name and the database in which the collection is stored. Remind that when the database is not specified, the collection is taken from the intermediate result database *IR*.

5.1.1 SET INTERMEDIATE AS

The **SET INTERMEDIATE AS** operator stores the input temporary collection into the intermediate results database IR. The syntax of operator is:

```
SET INTERMEDIATE AS collectionName;
```

Note that *collectionName* is the name given to the temporary collection when stored into the intermediate results database *IR*.

5.1.2 SAVE AS

The **SAVE AS** operator saves the input temporary collection into a persistent database.

```
SAVE AS collectionName@dbName;
```

6 SPATIAL JOIN OF COLLECITONS

Currently, J-CO-QL provides one operator for spatial processing and analysis, i.e., the **SPATIAL JOIN OF COLLECTIONS** operator. This is the key operator for performing spatial analysis and is so rich that it is possible to perform very complex analysis. Furthermore (see Section 4), geometrical features are supported in the basic blocks of a large number of operators.

The **SPATIAL JOIN OF COLLECTIONS** (or **SPATIAL JOIN** for simplicity) operator is a *start* operator that performs the geo-spatial join between two input collections based on the truth of a metric or topological condition evaluated between the geometries of any pair of two objects from the two input collections. The metric conditions can be defined on the distance between two geometries, for example requiring that their distance is lower than a maximum threshold, or the intersection of their geometries that must be not empty or with an area greater than a specific value. The topological condition can be defined on the orientation of one geometry w.r.t. the other geometry, or that the two geometries share some part of their boundary, i.e., they meet, or that one geometry is covered (is included) or covers (includes) the second geometry.

The syntax of the **SPATIAL JOIN** operator is the following:

```
SPATIAL JOIN OF COLLECTIONS
collectionReference1, collectionReference2
[ON spatialJoinCondition]
SET GEOMETRY (INTERSECTION | RIGHT
             | LEFT | ALL)
[caseClause];
```

The operator makes the join between two input collections: these can be stored either in persistent, possibly distinct, databases or in the intermediate result database *IR*. Furthermore, the two input collections must have different names (or must be aliased with different names, see Section 4.1) to avoid any ambiguity in the output. The spatial join of two input objects is performed if a metric or a topological condition defined on their geometries is satisfied: this is the mandatory *spatialJoinCondition* specified after the **ON** keyword. The **SET GEOMETRY** parameters allows specifying the geometry of the output object.

The resulting objects are possibly selected and modified by *caseClause* (see Section 4.3), that actually generates the new temporary collection.

Let us give a precise semantic description. For each object l_i in the left collection (*collectionReference1*, named or aliased as *lcn*) and an object r_j in the right collection (*collectionReference2*, named or aliased as *rcn*) both having the **geometry** field, an object $o_{i,j}$ is generated if the geometries of l_i and r_j meet the *spatialJoinCondition*. and if the $o_{i,j}$ is approved by *caseClause* (see Section 4.3), when this clause is used. The generated object $o_{i,j}$ has three fields: the first one has the name *lcn*) and contains the left object l_i (i,.e., $o_{i,j}$.*lcn*= l_i); the second field has the name *rcn*) and contains the right object r_j (i,.e., $o_{i,j}$.*rcn*= r_j); the third field is the **geometry** field, resulting from the spatial join, based on the **SET GEOMETRY** parameter. When **SET GEOMETRY INTERSECTION** is specified, $o_{i,j}$.**geometry** is the intersection of the geometries of the joined objects; when **SET GEOMETRY RIGHT** (resp. **SET GEOMETRY LEFT**) is specified, $o_{i,j}$.**geometry** is the geometry of the right (resp., left) input object; when **SET GEOMETRY ALL** is specified $o_{i,j}$.**geometry** is the union of the geometries of the input objects.

If the *caseClause* is not specified, the objects $o_{i,j}$ are put as they are in the output temporary collection. If the *caseClause* is specified, the objects $o_{i,j}$ are then possibly filtered ad restructured (see Section 4.3) before they are put into the output temporary collection. This way, the output can be customized.

Metric and topological relationships are expressed by pre-defined properties, which are as follows:

- **DISTANCE(**$unit$**)** is the distance between the center of the two geometries, expressed based on the specified $unit$, that can be **M** (meters), **KM** (Kilometres), **ML** (miles).
- **AREA(**$unit$**)** is the area of the intersection of the two geometries expressed based on the specified $unit$, that can be **M** (square meters), **KM** (square Kilometres), **ML** (square miles).
- **ORIENTATION(**$from$**)** reports the cardinal orientation of l_i**.geometry** w.r.t. r_j**.geometry** or vice versa; If the $from$ parameter is **LEFT** (resp. **RIGHT**), it is the orientation of the spatial vector from the center of l_i**.geometry** to the center of r_j**.geometry** (resp., from the center of r_j**.geometry** to the center of l_i**.geometry**); orientation values are strings obtained by composing the 4 letters **N** (for North), **E** (for East), **S** (for South) and **W** (for West): a single letter (e.g., `"N"` for North orientation), a pair (e.g., `"NE"` for North-East), a triple (e.g., `"NNE"` for North-North-East).
- **INCLUDED(**$side$**)** is a boolean property that denotes the inclusion of l_i**.geometry** w.r.t. r_j**.geometry**; if the $side$ parameter is **LEFT**, **INCLUDED(LEFT)** is true if l_i**.geometry** is completely included in r_j**.geometry**; if the $side$ parameter is **RIGHT**, **INCLUDED(RIGHT)** is true if r_j**.geometry** is completely included in l_i**.geometry**..
- **MEET** is a boolean property, that is true when the two geometries share a common part of their boundaries.
- **INTERSECT** is a boolean property that is true when the two geometries intersect.

Example 4. Consider collection `Buildings` shown in Listing 1 and collection *WaterLine* shown in Listings 2, stored in database `ToyDB`. Suppose we are a company committed to perform maintenance of water lines. Thus, we want to know which water lines passes below which buildings in city `"City A"`. The query is the following:

```
SPATIAL JOIN OF COLLECTIONS
Buildings@ToyDB, WaterLines@ToyDB
ON INTERSECT
SET GEOMETRY INTERSECTION
CASE
 WHERE WITH STRING .Buildings.City
   AND .Buildings.City = "City A"
 WHERE WITH STRING .Buildings.CityName
   AND .Buildings.CityName = "City A"
DROP OTHERS;
```

The operator performs a geospatial join between each object b_i in collection `Buildings` and each object w_j in collection *WaterLines*, in such a way the geospatial representations of b_i and w_j intersect. In practice, we are interested in discovering which water lines passes below which building. The output collections:

```
[
  {
   Buildings:{
     "name":"buildingA",
     "city":"city A",
     "address":"address A",
     "geometry":{"type":"GeometryCollection",
                     "geometries":[
                       {
                        "type": "Polygon",
                        "coordinates": [
                          [[100.0, 0.0],
                           [101.0, 0.0],
                           [101.0, 1.0],
                           [100.0, 1.0],
                           [100.0, 0.0] ]]
                       }

                 ]

            }

   },
   Waterlines:{
     "name":"WaterLineA",
     "city":"city A",
     "geometry":{"type":"GeometryCollection",
                     "geometries":[
                       {
                        "type": "LineString",
                        "coordinates":
                          [[90.0, 0.0],
                           [103.0, 1.0],
                           [104.0, 0.0],
                           [105.0, 1.0]]
                       }
                     ]
                  }
   },
   "geometry":{
              "type": "LineString",
              "coordinates":
               [[100.0,0.7692307692307692],
               [101.0,0.8461538461538461]]
             }
  }
]
```

Notice that the output has the default structure (**GENERATE** action has not been used). The field *Buildings* contains the matching object b_i coming from the left collection, while field `WaterLines` contains the matching object w_J coming from the right collection; field *geometry* is the GeoJSON representation of the geospatial intersection between geometries of objects b_i and w_j, that in our cases is a line. Notice that we obtain only one pair, related to city `"City A"`, because it is the only one that actually overlays and the city of the building is the required one.
The goal of the example is to know which water lines pass below which building. For this purpose, not all information contained in the output shown, are needed (i.e. the details about geospatial information of the specific building and waterline, may not be needed).

The below listing, for example, shows the same query written using the **GENERATE** action, that permits to obtain a customized output containing only the desired information and structured as we want.

```
SPATIAL JOIN OF COLLECTIONS
Buildings@ToyDB, WaterLines@ToyDB
ON INTERSECT
SET GEOMETRY INTERSECTION
CASE
 WHERE WITH STRING .Buildings.City
   AND .Buildings.City = "City A"
   GENERATE {.City: .Buildings.City,
             .Building: .Buildings.name,
             .Waterline: .WaterLine.name}
             KEEPING GEOMETRY
WHERE WITH STRING .Buildings.CityName
   AND .Buildings.CityName = "City A"
   GENERATE {.City: .Buildings.CityName,
             .Building: .Buildings.name,
             .Waterline: .WaterLine.name}
             KEEPING GEOMETRY
DROP OTHERS;
```

The output is the following:

```
[
 {
  "City":"city A",
  "Building":"buildingA",
  "Waterline":"WaterLineA",
  "geometry":{
              "type": "LineString",
              "coordinates":
                [[100.0,0.7692307692307692],
                 [101.0,0.8461538461538461]]
             }
  }
]
```

The output contains only the objects needed for our purpose and are structured in a more compact way than in the input. The field *City* represents the city of the building, the field *Building* represents the name of the building and the field *WaterLine* represents the name of the waterline. At last, the field *geometry* contains the geometry intersection between building (*buildingA*) and waterline (*waterLineA*) according to GeoJSON standard.
Notice the power of the **GENERATE** action which permits to restructured a JSON object, completely changing its structure.

Example 5. Consider the collection *Buildings* inside *ToyDB* database and shown in Listing 1. Assume, we want to know which building are distant less than 50 meters in the city C.
The syntax of the query is the follow:

```
SPATIAL JOIN OF COLLECTIONS
Buildings@ToyDB AS B1, Buildings@ToyDB AS B2
ON DISTANCE(M) < 50
SET GEOMETRY ALL
CASE
 WHERE WITH STRING .B1.City
   AND .B1.City = "City C"
WHERE WITH STRING .B1.CityName
   AND .B1.CityName = "City C"
DROP OTHERS;
```

The Listing above performs a geospatial join between each object $b1_i$ (in collection named B1) and each object $b2_j$ (in collection named B2), only if the distance between the buildings is less than 50 meters (see **ON** clause, where the **DISTANCE** metric is used). Moreover, the use of **WHERE** clauses permit to keep only the pairs of building located in the city C. The output is not modified by **GENERATE** action and each object oi_j present in output contains: $b1_i$ object in collection named B1; $b2_j$ object in collection named B2; *geometry* object which include both geometry of the object $b1_i$ and $b2_j$ (see **SET GEOMETRY ALL**).
Notice the simplicity of use of the **DISTANCE** metric in the **SPATIAL JOIN** operator.

7 Fundamental Non-Spatial Operators

7.1 *JOIN OF COLLECTIONS*

The **JOIN OF COLLECTIONS** (or **JOIN** for simplicity) operator is a *start* operator that makes the no-geospatial join between two collections. W.r.t. the **SPATIAL JOIN** operator, the **JOIN** operator works on non-geospatial fields. The syntax of operator is hereafter.

```
JOIN OF COLLECTIONS
collectionReference1, collectionReference2
[caseClause];
```

The output of the **JOIN** operator is a JSON collection obtained by pairing objects in both input collections.

Let us denote with *lcn* the name (or alias) of the left collection *collectionReference1*, with *rcn* the name (or alias) of the right collection *collectionReference2*; in order to avoid ambiguity, *lcn* and *rcn* must differ. For each object l_i in the left collection and for each object r_j in the right collection, an object $o_{i,j}$ is generated, with two fields: the first one has the name of the left collection *lcn* and its value is the left object l_i, i.e., $o_{i,j}$.*lcn*$= l_i$; the second field has the name of the right collection *rcn* and its value is the right object r_j, i.e., $o_{i,j}$.*rcn*$= r_j$.

At this point, the *caseClause* selects and possibly restructure the generated objects, in dealing with heterogeneous resulting object all together: the various **WHERE** branches recognize different objects and apply different join condition. After the **WHERE** branches, the user specifies either the **KEEP OTHERS** option or the **DROP OTHERS** option.

In practice, the **JOIN** is designed as a Cartesian product followed by a pool of selection conditions, where the associated **GENERATE** action cab restructure the output object. The result is a very flexible operator, because:

- if the *caseClause* is absent, the operator behaves as a pure Cartesian product, i.e., the output temporary collection contains all possible pairs of objects in the input collections;
- if one or more **WHERE** branches are specified with the **DROP OTHERS** option, this corresponds to a classical join enriched with the possibility of dealing with heterogeneous object combinations;
- if one or more **WHERE** branches are specified with the **KEEP OTHERS** option, this corresponds to a probably uncommon situation, where all objects generated

by the Cartesian product are kept into the output temporary collection, even though no join condition (i.e., no **WHERE** branch) is met by them. Although unusual, we think that it could be useful in practical applications and, nevertheless, it is coherent with the design.

Note that, although the **JOIN** operator does not deal with geometries, the **GENERATE** actions specified in the **WHERE** branches deals with geometry. The following example shows that.

Example 6. Suppose we are the same company of Example 4. We need to associate restaurants to buildings, based on their address, considering restaurants for which an geographical description is given. We can write the following query.

```
JOIN OF COLLECTIONS Buildings@ToyDB AS B,
                    Restaurants@ToyDB AS R
CASE
 WHERE WITH STRING .B.City, .B.Address,
           .R.City, .R.Address AND
       WITH GEOMETRY .R.geometry AND
       .B.City=.R.City AND
       .B.Address=.R.Address
  GENERATE {BuildingName: .B.Name,
           RestaurantName: .R.Name,
           City: .B.City,
           Address: .B.Address}
           SETTING GEOMETRTY .B.geometry
 WHERE WITH STRING .B.CityName, .B.Address,
           .R.City, .R.Address AND
        WITH GEOMETRY .R.geometry AND
        .B.CityName=.R.City AND
        .B.Address=.R.Address
  GENERATE {BuildingName: .B.Name,
           RestaurantName: .R.Name,
           City: .B.CityName,
           Address: .B.Address}
           SETTING GEOMETRTY .B.geometry
DROP OTHERS;
```

The **JOIN** operator creates all pairs between the objects containing in **Buildings@ToyDB** (aliased as **B**) and **Restaurants@ToyDB** (aliased as **R**) collections. Then, the use of *caseClause* permits to keep only pairs of objects in which their cities and their address coincides. The output collection is reported in the listing below.
The two **WHERE** branches makes substantially the same thing, i.e., express a condition that filters out those pairs having the same city name and the same address, provided that the restaurant object (aliased as **R**) has a geometry (predicate **WITH GEOMETRY .R.geometry**). Two branches are necessary to deal with the fact that, within buildings, some objects have field **City** and some others have field **CityName**.
Finally, for both branches, the **GENERATE** action actually generates the output objects. The field *BuildingName* represents the name of the building, the field *RestaurantName* represents the name of the restaurant (the names are different because they are kept by different collections). The *City* and *Address* represent respectively the city and the address of the building or restaurant. At last, the field *geometry* contains the geometry of the Building (*buildingA*).

Notice that, the use of the **GENERATE** action permits to generate a JSON collection with a completely different structure from the structure of the input objects. In the case of the example, the need is to get a uniform structure for the output objects.

```
[{
    "BuildingName":"buildingA",
    "RestaurantName":"RestaurantA",
    "City":"city A",
    "Address":"address A",
    "geometry":{"type":"GeometryCollection",
                "geometries":[
                  {
                   "type": "Polygon",
                   "coordinates": [
                      [[100.0, 0.0],
                       [101.0, 0.0],
                       [101.0, 1.0],
                       [100.0, 1.0],
                       [100.0, 0.0]]]
                   }
                ]
              }

}]
```

7.1.1 FILTER

The **FILTER** operator is a *carry-on* operator that permits to filter objects in the temporary collection, according to a *caseClause* (see Section 4.3).

The syntax is the following:

```
FILTER
 caseClause;
```

The **FILTER** clause takes the input temporary collection, filters its objects according to **WHERE** clause and possibly generates a restructured version of them (when the **GENERATE** action is specified). In practice, the **FILTER** operator applies a *caseClause* (see Section 4.3) to the temporary collection. The following example will show a practical application of the operator.

Example 7. Consider collection *Restaurants* shown in Listing 3 which represents some restaurant. We want to derive the **geometry** field for those restaurants having fields **lat** (latitude) and **lng** (longitude), which are not provided with a GeoJSON-compliant **geometry** field.
The query is hereafter.

```
GET COLLECTION Restaurants@ToyDB;
FILTER
 CASE
   WHERE WITH LOAT .lat, .lng AND
         WITHOUT .geometry
     GENERATE SETTING GEOMETRY
            POINT(.lat, .lng)
 KEEP OTHERS;
```

The **GET COLLECTION** operator retrieves the initial `Restaurants` collection from the persistent database; this collection becomes the new temporary collection. The **FILTER** operator carries on the process.
Notice the simplicity of the *caseClause*. Only one single **WHERE** branch is sufficient for our purpose; the condition

looks for objects without **geometry** field, having two floating point valued fields named **lat** and **lng**. The **GENERATE** action does not specify a structure for the output object, meaning that the structure of the object remains untouched, but the **SETTING GEOMETRY** parameter derives a point geometry from fields **lat** and **lng**. Finally, **KEEP OTERS** says that not selected objects remains as they are in the output temporary collection.
The new temporary collection produced by the operator is hereafter.

```
[
  { "name":"RestaurantA",
   "city":"city A",
   "address":"address A"
  },

  {
   "name":"RestaurantB",
   "city":"city B",
   "address":"address C"
   "lat": -10.574228,
   "lng": 21.015217,
   "geometry":{
             "type": "Point",
             "coordinates": [
                21.015217,
               -10.574228]
            }
    },

    {
     "name":"RestaurantC",
     "city":"city C",
     "address":"address D",
     "geometry":{"type":"GeometryCollection",
               "geometries":[
                 {
                  "type": "Polygon",
                  "coordinates":[
                     [[20.0, -10.0],
                      [21.0, -10.0],
                      [21.0, -11.0],
                      [20.0, -11.0],
                      [20.0, -10.0]]]
                 }
                ]
               }
    }
]
```

7.1.2 GROUP

The **GROUP** operator is a *carry-on* operator that groups objects in the input temporary collection. Although conceptually similar to the classical **GROUP BY** provided by SQL, it is a rich operator designed to deal with the heterogeneous nature of JSON collections. Here is its syntax.

```
GROUP
 ( PARTITION whereCondition
   BY fieldReference (, fieldReference)*
   INTO fieldName
   [SORTED BY
      fieldReference (, fieldReference)*]
   [generateClause] )+
 ( KEEP OTHERS|DROP OTHERS );
```

First of all, the **GROUP** operator partitions objects in the input temporary collection on the bases of the *whereCondition* reported after the **PARTITION** keyword; for each **PARTITION** branch, the P_i set of all objects that satisfy the *whereCondition* is built (in case an object meets more than one condition, it is inserted into the partition associated with the first satisfied condition in the list).

For each partition P_i, objects belonging to P_i are further grouped by the list of fields after the **BY** keyword: the operator groups the objects in such a way a group contains all the objects $o_1, \ldots, o_n$ having the same values for fields specified by the list of *fieldReference* after the **BY** keyword.

For each group, an object g_k appears in the output temporary collection, such that it has all the grouping fields and a array field containing objects $o_1, \ldots, o_n$; the name of this field is specified after the keyword **INTO**.

The optional **SORTED BY** option specifies whether to sort objects into the array fields. If so, the *fieldReference*s listed after the **SORTED BY** keywords are the sort keys.

Finally, if specified, the **GENERATE** clause can further restructure the generated objects.

Notice that the output temporary collection contains as many objects as the number of groups. Furthermore, in case the **KEEP OTERS** option is specified, all objects that do not match any **PARTITION** branch are kept into the output temporary collection.

The following example shows practical application of the **GROUP** operator.

Example 8. Consider the collection *WaterLines* shown in Listing 2. We might be interested in grouping water lines by their city name. The query is hereafter.

```
GET COLLECTION WaterLines@ToyDB;
GROUP
  PARTITION WITH STRING .city
  BY .city
  INTO waterLineCity
 DROP OTHERS;
```

The **GET COLLECTION** operator retrieves the `WaterLines` collection from the persistent database and makes it the new temporary collection, on which the **GROUP** operator works.
Water lines having the string-valued **city** field constitute the partition in which we are interested in (in the case of Listing 2, all water lines). They are grouped by field `City`; the name given to the array field containing grouped objects is `waterLineCity`. Here is the output temporary collection.

```
[
 {"city":"city A",
  "waterLineCity":[
        {"name":"WaterLineA",
     "city":"city A",
     "geometry":{"type":"GeometryCollection",
                 "geometries":[
                    {"type": "LineString",
                     "coordinates":
                       [[90.0, 0.0],
                        [103.0, 1.0],
                        [104.0, 0.0],
                        [105.0, 1.0]]
                    }
```

```
                ]
              }
    }]
  },

  {"name":"WaterLineB",
     "city":"city A",
     "geometry":{"type":"GeometryCollection",
              "geometries":[
                 {"type": "LineString",
                  "coordinates":
                     [[102.0, 10.0],
                      [103.0, 2.0],
                      [104.0, 1.0],
                      [102.0, -1.0]]
                 }
               ]
             }
  },

  {"city":"city C",
   "waterLineCity":[
    {"name":"WaterLineC",
     "city":"city C",
     "geometry":{"type":"GeometryCollection",
                 "geometries":[
                   {
                    "type": "LineString",
                    "coordinates":
                      [[104.0, 10.0],
                       [105.0, 2.0],
                       [109.0, 1.0],
                       [110.0, -1.0]]
                   }
                 ]
               }
    }]
}]
```

The output contains an object which represents the group for *city A* (first object) and another object which represents the grouping for *city C* (second object). The first object contains a field *city* (grouping field) with value *city A* and a field *waterLineCity* which contains all objects of the Listings 2 with the field *city* equals to *"city A"* (grouped objects). The second object has the same structure but the grouping field is *City C*.

If the user would like to derive a global geometry for each resulting object, a **GENERATE** action with could be used, as follows.

```
GET COLLECTION WaterLines@ToyDB;
GROUP
  PARTITION WITH STRING .city
  BY .city
  INTO waterLineCity
  GENERATE SETTING GEOMETRY
      AGGREGATE(.waterLineCity)
 DROP OTHERS;
```

The **GENERATE** action simply adds a **geometry** to each external object, obtained by aggregating all **geometry** fields of objects grouped within the array field **waterLineCity**. For the sake of space, we do not report the new output collection.

The above example is based on one single partition. Multiple partitions are very useful when the collection contains very heterogeneous objects, which corresponds to different real world entities, that must be separately grouped. For instance, suppose we have a temporary collection containing both railways and roads: railways have (among other fields) a **railwayOwne** field, a **type** field and a **maxSpeed** field; roads have (among others) a **countyName** field, a **townName** field and **width** field. As far as railways are concerned, we want to group them by **railwayOwner** and **type**; as far as roads are concerned, we want to group them by **countyName** and **townName**. The solution is to define two different partitions and, then, group objects within them separately. The resulting **GROUP** operator could be the following.

```
GROUP
 PARTITION
   WITH .RailwayOwner, .Type, .maxSpeed
   BY .RailwayOwner, .Type
   INTO groupedRailways
   ORDERED BY .maxSpeed
 PARTITION
   WITH STRING .countyName, .townName AND
        WITH FLOAT .width
   BY .countyName, .townName,
   INTO groupedRoads
 DROP OTHERS;
```

In contrast, if the need is to group together objects that describe homogeneous real world entities but have different field name describing the property, on which fields objects should be grouped, the solution is to further apply the **FILTER** operator and, by means of the **GENERATE** action, rename the fields homogeneously. Then the **GROUP** operator could be applied by with one single partition.

7.1.3 EXPAND

The **EXPAND** operator is a *carry-on* operator that permits to un-nest objects contained in an array field, putting the resulting objects into the output temporary collection. The syntax of operator is reported hereafter.

```
EXPAND
 ( UNPACK whereCondition
   ARRAY fieldReference
   TO fieldName
   [generateClause] )+
 ( KEEP OTHERS|DROP OTHERS ) ;
```

In order to deal with heterogeneity of collections, several **UNPACK** branches are possible. Each **UNPACK** branch has a *whereCondition*, that selects objects to unpack in the branch. For each selected object s_i, the array field specified after the **ARRAY** keyword is unpacked, i.e., in place of s_i, for each item a_j (object or simple value) contained in in the array, a new object $o_{i,j}$ is generated, which has all fields in s_i apart from the array field *fieldReference*, which is replaced by a new field *fieldName* having a_j as value, .e., $o_{i,j}$.*fieldName*= a_j.

After that, within a branch, it is possible to specify a **GENERATE** action, that permits to further restructure the output objects.

As for *WHERE* clauses, if an object meets several *whereConditions*, the first one in the list will be considered, and the corresponding **UNPACK** branch applied on the object.

Finally, as for other operators, **KEEP OTHERS|DROP OTHERS** options denote if not selected objects must be, resp., taken or discarded in the output temporary collection.

Example 9. Suppose we want to backward the **GROUP** operation reported in Example 8. The **EXPAND** operation is the following.

```
EXPAND
 UNPACK WITH STRING .city AND
        WITH ARRAY .waterLineCity
 TO tmp
  GENERATE {.tmp.name, .tmp.city}
      SETTING GEOMETRY .tmp.geometry
DROP OTHERS;
```

One single unpack branch is necessary, which selects for objects having a string-valued **city** field and an array field **waterLineCity**. In place of this array field, expanded object will have field **tmp**, containing the previously grouped objects describing water networks. To obtain again the original structure, the **GENERATE** action is specified.

Again, the reader can notice that non-spatial operators apparently do not support geometrical features of objects. However, it is always possible to apply the **GENERATE** action, overtaking this apparent limitation.

8 Useful Set-Oriented Operators

For the sake of providing users with a large variety of operators, some useful set-oriented operators are provided.

8.1 *MERGE COLLECTIONS*

The **MERGE COLLECTIONS** operator merges the content of two or more collections into the temporary collection. Here is its syntax.

```
[ALL] MERGE COLLECTIONS collectionReference
                    (, collectionReference)+;
```

When the option **ALL** is not specified, the operator removes duplicate objects, possibly coming from different collections. When the **ALL** option is specified, duplicate objects are not removed. The operator exploits the heterogeneous nature of JSON collections: objects with different structure can be stored together without any limitation.

8.2 *INTERSECT COLLECTIONS*

The **INTERSECT COLLECTIONS** operator makes a set intersection between collections and puts the resulting collection into the temporary collection.

```
INTERSECT COLLECTIONS collectionReference1,
                      collectionReference2;
```

The **INTERSECT COLLECTIONS** performs a deep equality matching: only identical objects present in both the input collections matches and only one single occurrence of them is put into the output collection.

8.3 *SUBTRACT COLLECTIONS*

The **SUBTRACT COLLECTIONS** operator makes a set-oriented subtraction between collections and puts the resulting objects into the temporary collection.

```
SUBTRACT COLLECTIONS collectionReference1,
                     collectionReference2;
```

The **SUBTRACT COLLECTIONS** returns a JSON collections containing all objects in *collectionName1@dbName* without an identical object (based on deep equality matching) in *collectionName2@dbName*.

9 Complete Example

In order to show the effectiveness of J-CO-QL for complex tasks, we wrote the query in Listings 4, based on the collections stored in our sample DB named `ToyDB`. The goal is to extract the informations of restaurants located in buildings in *city A* under which some water lines pass. This query is useful for water lines maintenance in *city A*.

Listing 4. Complete Example

```
1  SPATIAL JOIN OF COLLECTIONS
2    Buildings@ToyDB, WaterLines@ToyDB
3    ON INTERSECT
4    SET GEOMETRY INTERSECTION
5   CASE
6    WHERE WITH STRING .Buildings.City
7      AND .Buildings.City = "City A"
8   WHERE WITH STRING .Buildings.CityName
9      AND .Buildings.CityName = "City A"
10  DROP OTHERS;
11 SET INTERMEDIATE AS BWCityA;
12
13 GET COLLECTION Restaurants@ToyDB;
14 FILTER
15  CASE
16    WHERE WITH LOAT .lat, .lng AND
17          WITHOUT .geometry
18     GENERATE SETTING GEOMETRY
19            POINT(.lat, .lng)
20  KEEP OTHERS;
21 SET INTERMEDIATE AS RestaurantsWGeom;
22
23 JOIN OF COLLECTIONS BWCityA AS Bwca,
24                     RestaurantsWGeom AS Rwg
25  CASE
26   WHERE WITH STRING .Bwca.City,
27             .Bwca.Address,
28             .Rwg.City, .Rwg.Address AND
29         WITH GEOMETRY .Rwg.geometry AND
30         .Bwca.City=.Rwg.City AND
31         .Bwca.Address=.Rwg.Address
32    GENERATE {BuildingName: .Bwca.Name,
33              RestaurantName: .Rwg.Name,
34              City: .Bwca.City,
35              Address: .Bwca.Address}
36            SETTING GEOMETRTY .Bwca.geometry
37   WHERE WITH STRING .Bwca.CityName,
38             .Bwca.Address, .Rwg.City,
39             .Rwg.Address AND
40         WITH GEOMETRY .Rwg.geometry AND
41          .Bwca.CityName=.Rwg.City AND
42          .Bwca.Address=.Rwg.Address
43    GENERATE {BuildingName: .Bwca.Name,
44              RestaurantName: .Rwg.Name,
```

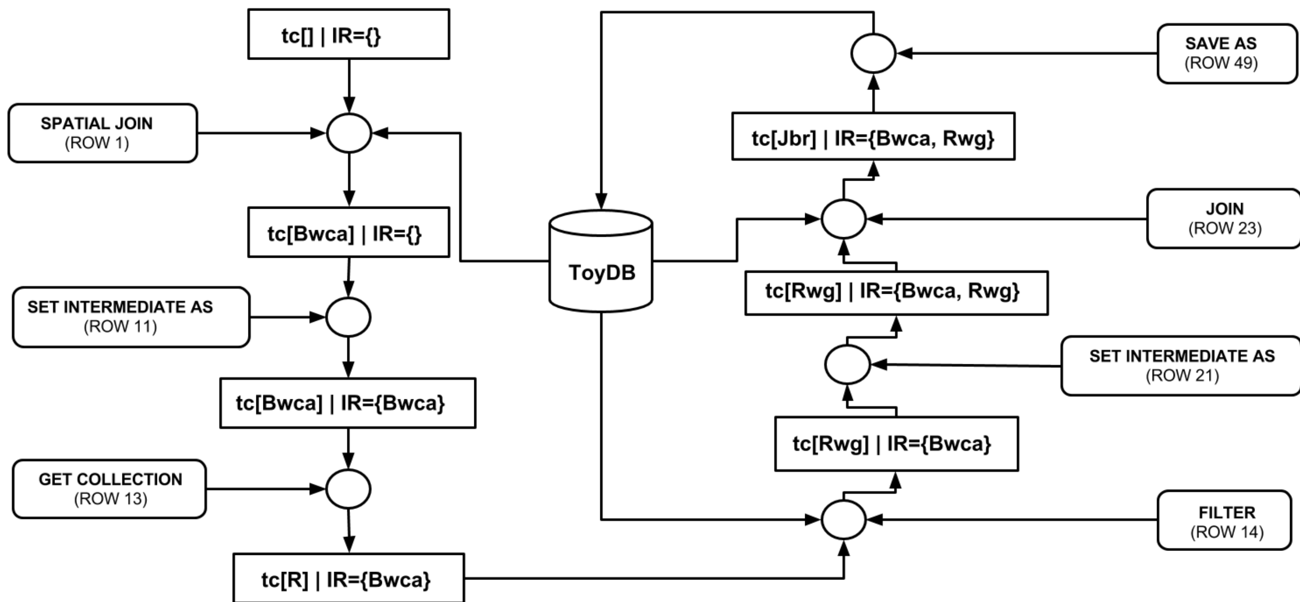

Fig. 2. Execution Model example.

```
45                City: .Bwca.CityName,
46                Address: .Bwca.Address}
47              SETTING GEOMETRTY .Bwca.geometry
48  DROP OTHERS;
49 SAVE AS RestaurantsWL@ToyDB;
```

Figure 2 shows the execution model of the example. It shows how the temporary collection and intermediate database IR change during the process. In Figure 2, for lack of space, we used abbreviated names with the following meaning:

- `Bwca` stands for `BWCityA`,
- `R` stands for `Restaurants@ToyDB`,
- `Rwg` stands for `RestaurantsWGeom`,
- `Jbr` stands for *Output of the join operator*.

Moreover, with term *tc*, we indicate the temporary collection and with term *IR*, we denote the intermediate database. Rectangles represent the query process state and show how the temporary collection and intermediate database IR change during the query process.

The query process works as follows:

9.1 *New Task*

The query starts a new task using the **`SPATIAL JOIN`** operator (which is a *Start operator*). The **`SPATIAL JOIN`** (see row 1) makes the geospatial join considering the geospatial intersection between `Buildings@ToyDB` (shown in Listings 1) and `WaterLines@ToyDB` (shown in Listings 2). The **`SPATIAL JOIN`** outputs, into a new temporary collection, the pairs of buildings and water lines in *cityA* which intersect, with their geo-spatial intersection. Figure 2 shows that the element `Bwca` (`BWCityA`) has been put into *tc*.

The temporary collection generated by **`SPATIAL JOIN`** is then saved into the intermediate result database IR by the **`SET INTERMEDIATE AS`** operator. This operator saves *tc* into IR, naming the collection as `BWCityA` (see row 11). Figure 2 shows that the element `Bwca` (`BWCityA`) has been added to IR.

9.2 *Subtask 1*

The **`GET COLLECTIONS`** operator (which is a *Start operator*) starts a new subtask. The **`GET COLLECTION`** (see row 13) outputs the `Restaurants@ToyDB` (shown in Listings 3) into the temporary collection. Figure 2 shows that the element `R` (`Restaurants@ToyDB`) has become the new *tc*.

The temporary collection is now the input collection for the **`FILTER`** operator. It keeps any restaurants and adds a field **`geometry`** (of point type) to anyone of them which have the fields **`lat`** and **`lng`** and do not have the field **`geometry`**. Any other restaurant in the collection is kept as it is in input (see the **`CASE`** clause). The result becomes the new temporary collection. Figure 2 shows that the element `Rwg` (`RestaurantsWGeom`) has become the new *tc*.

The temporary collection produced by **`FILTER`** is then saved into the intermediate results database IR by the **`SET INTERMEDIATE AS`** operator. This operator saves *tc* into IR, naming the collection as `RestaurantsWGeom` (see row 21). Figure 2 shows that the element `Rwg` (`RestaurantsWGeom`) has been added to IR.

This subtask permits to change the structure of the *Restaurants@ToyDB* collection (shown in Listings 3) according to J-CO-QL data model. The **`GENERATE`** action adds the field **`geometry`** to any restaurants which do not have that field. This way, J-CO-QL can manage any JSON collection, even if some objects are geo-tagged in a way not compliant with GeoJSON standard.

9.3 *Subtask 2*

In row 23 the **`JOIN`** operator starts the final subtask. This operator makes a join between the intermediate collections `BWCityA` and `RestaurantsWGeom` (retrieved from IR and

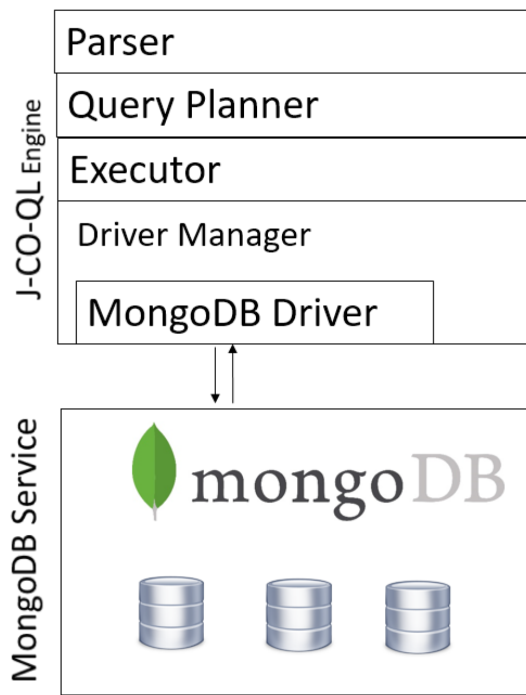

Fig. 3. Architecture of J-CO-QL Engine.

respectively renamed as `Bwca` and `Rwg`). It outputs the join between buildings and restaurants (pre-filtered by the **`FILTER`** operator at row 14) which have the same address and the same city (see the **`CASE`** clause). The output becomes the new temporary collection. Figure 2 shows that the element `Jbr` (output of the join) has been the new *tc*.

Finally, the current temporary collection *tc*, which indeed contains the desired restaurant information, is saved into the persistent database named `ToyDB` by the **`SAVE AS`** operator, with name `RestaurantsWL`. Figure 2 shows that the element `Jbr` (output of the join) is saved into persistent database *ToyDB*.

The reader can observe that the query is easy to read and rather intuitive as far as its execution is concerned.

10 J-CO-QL ENGINE

In order to prove the feasibility of our approach, we developed a prototype version of the *J-CO-QL Engine*. In Subsection 10.1 we present the architecture; in Subsection 10.2 we discuss a preliminary performance evaluation that encourages us to carry on the development.

10.1 Architecture

J-CO-QL Engine is written in Java; it is designed to be an independent tool, external to any DBMS (currently, MongoDB): this choice makes the J-CO-QL engine able to operate on several DBMSs. The architecture of J-CO-QL Engine is reported in Figure 3.

The stack of components includes the *Parser*, that transforms the query text into an internal representation received by the *Planner*. This component generates the execution plan by relying on an internal object-oriented representation. The *Executor* controls the execution of each single instructions in the query plan, manage main memory usage and temporary collections and the intermediate results database. When necessary, it interacts with the *Driver Manager*, that currently includes only the *MongoDB Driver*, but in the future will includes drivers for other DBMSs and data sources. The Driver Manager interacts with MongoDB to retrieve collections, store result collections and, when necessary (in case of low levels of available main memory) transfers intermediate collections to MongoDB.

The query plan is represented as a vector of objects defined on the abstract class *JCO_Executable*, from which specific and non-abstract subclasses are derived, one for each J-CO-QL operator. By exploiting polymorphism, the *Executor* calls the `execute_operator` method of each object, providing references to its internal component to get input collections, store the temporary collection, the intermediate collections and save collections to the persistent database. The `execute_operator` method can call these components when necessary.

The implementations of the operators make use of some libraries, necessary to an efficient implementation. In particular, *LocationTech Spatial4j*[3] and *Tsusiat Software Java Topology Suite* [4] are used to deal with spatial representations, while *David Moten's R-tree/R*-tree in memory indexing*[5] in order to perform all computation related with geometry.
Currently, the implementation is not optimized: it tries to adopt a main memory approach to execute queries, but no optimizations are implemented. Anyway, we are evaluating the introduction of (spatial) main-memory indexes, that will certainly improve performance.

10.2 Performance Evaluation

In order to have a first validation of our approach, we ran some experiments with the prototype of the J-CO-QL Engine. We made the experiments on a MacBook Pro Retina, equipped with an Intel i7 Quad Core processor with 2,5 GHz clock-rate, 16 GB RAM and an SSD drive.

We used two data sets available on the internet. The first one is named *restaurants.json* and contains about 26500 objects with point geometry. The second one is named *countries.geo.json* and describes the polygon geometry of 180 countries.

In Table 2, we report the execution times (in sec) we measured during tests; each experiment was repeated 10 times and we report the average execution times.

We tested the most critical operators, i.e., the **`JOIN`** operator and the **`SPATIAL JOIN`** operator.

In the table, columns *Size of C1* and *Size of C2* report the number of objects in the input collections; column *size of output* reports the number of objects in the output collection; column *Total time* reports the execution time of the operator.

As far as the **`JOIN`** operator is concerned, we tested it by joining the collection *restaurants.json* with itself. The number of potential pairs was 25360×25360 and the execution time was 23 sec. Certainly, it must be improved, but it is acceptable for off-line analysis.

3. https://github.com/locationtech/spatial4j
4. JTS - http://tsusiatsoftware.net/jts/main.html
5. https://github.com/davidmoten/rtree

As far as the **SPATIAL JOIN** operator is concerned, we joined collection *countries.geo.json* (countries) with collection *restaurants.json* (restaurants), in the second experiment with the **SPATIAL JOIN** operator, we joined four times the *restaurants.json* collection to get to 101440 objects.

The execution times show that it is necessary more than one minute in the second case; it is an acceptable time, but we want to improve it.

The results show that the approach is feasible, but the implementation needs the introduction of significant optimization. In fact, the application perspective of the J-CO-QL Framework is in Big Data Analysis.

11 Related Work

J-CO-QL moves from our previous work on the problem of querying heterogeneous collections of complex spatial data [11], [12]. In that work, we proposed a database model able to deal with heterogeneous collections of possibly nested spatial objects, based on the composition of more 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 works, J-CO-QL rely 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.

The adoption of NoSQL databases is motivated by the need of flexibility, as far as data structures are concerned, in interesting survey about NoSQL databases in [2], 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*. Consequently, the query language provided by such systems does not allow complex and multi-collection transformations like those provided by J-CO-QL (see the web sites reported in the footnote[6] for details). Readers interested in NoSQL DBMSs evaluation can refer to [3] and to [4].

Nevertheless, there are attempts to introduce support for JSON objects within traditional relational technology. The idea is to store JSON objects into text or blob attributes; then, SQL is extended with constructs that provide query and transformation capabilities among JSON objects. An example of this approach is in [7], where Oracle DBMS is extended this way.

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. Hereafter, we shortly refer to them.

Jaql [13] was designed to help Hadoop [14] programmer 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, while J-CO-QL constructs are at a higher level and truly declarative.

On the same track of query languages for improving MapReduce/Hadoop programming, we cite *ChuQL* [15], which deals with XML documents (not JSON objects, even though an XML is logically related to JSON). Compared to Jaql, ChuQL is even worst, in the sense that its constructs are still too programmatic; thus, it is not suitable for non programmers.

An interesting language is *Pig Latin* [16], a query language developed by Yahoo for writing complex analysis tasks on nested (1-NF, first normal form) data sets on top of Hadoop; thus, JSON collections are implicitly included. Pig Latin's constructs have names similar to J-CO-QL constructs, however, it strongly relies on the concept of variables: the result of each statement must be explicitly assigned to a variable, that can be later referred to by other statements; Clearly, this solution permits to explicit the computation flow, but it is not based on a database view of the problem. In contrast, J-CO-QL provides the concepts of temporary collection and intermediate results database, because it relies on a database view of the problem. The *DryadLINQ* language, presented in [17], follows a very similar approach to Pig Latin's approach.

Since JSON and XML are both suitable for representing semi-structured documents, it is worth mentioning the mostly known languages for querying XML documents. The first to mention is *XPath* [18], that allows to write path expressions to retrieve elements in a single XML document. On the basis of XPath, a complex language designed to work on collections of XML documents is *XQuery*. Among all features, it provides constructs to generate new documents, as well as the possibility to express complex queries.

The same approach is followed by *JSONiq* [19], a recent query language devised to write complex transformations on JSON documents. In practice, it can be considered the adaptation of *XQuery* to JSON and many features are taken from *XQuery*.

Although these languages are declarative, they are still oriented to a programmer vision. In contrast, J-CO-QL is oriented to data analysts, that need to explicitly manage heterogeneous collections of real world entities.

An interesting proposal, that tries to unify all previous languages, as well as SQL, is SQL++ [8]. 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 [8] nothing is said about data manipulation operators such as **INSERT**). In contrast, the execution model on which J-CO-QL relies clearly separate persistent databases and temporary databases, by means of the temporary collection and the intermediate result database *IR*.

6. MongoDB: https://www.mongodb.com/
CouchDB: http://docs.couchdb.org/en/2.0.0/

Operation	Size of C1	Size of C2	Size of Output	Total time
JOIN	25360	25360	446768	25.7
SPATIAL JOIN	176	25360	21336	16.6
SPATIAL JOIN	176	101440	101440	70.5

TABLE 2
Execution times (in sec) observed during the experiments.

12 Conclusions

In this paper, we proposed a query language, named J-CO-QL, specifically devised to query heterogeneous collections of (possibly) geo-tagged JSON objects, in order to provide a powerful tool to perform complex geographical analysis.

The language is defined in order to provide non programmers with a high level query language, which explicitly supports geographical representation and spatial aggregation operations.

The execution model which J-CO-QL relies on is, simple, intuitive, and suitable to express complex queries based on several subtasks.

The prototype we realized demonstrates the feasibility of the approach; performance will be improved by introducing several optimizations in the current implementation-

At this moment, we do not think J-CO-QL is at the end of development. We are planning to extend current operators with new features able to better deal with nesting and arrays. But above all, we are going to define new powerful operators for spatial analysis, such as sequence and trajectory matching. spatial clustering.

The final goal of the project is to obtain a powerful tool suitable for analysis of big data concerning territorial and geographical data sets, coming from heterogeneous sources of information.

References

[1] Z. H. Liu, B. Hammerschmidt, and D. McMahon, "Json data management: supporting schema-less development in rdbms," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1247–1258.

[2] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE, 2011, pp. 363–366.

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

[4] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Record*, vol. 39 (4), pp. 12–27, 2011.

[5] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.

[6] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing nosql mongodb to an sql db," in *Proceedings of the 51st ACM Southeast Conference*. ACM, 2013, p. 5.

[7] C. Chasseur, Y. Li, and J. M. Patel, "Enabling json document stores in relational systems." in *WebDB*, vol. 13, 2013, pp. 14–15.

[8] 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.

[9] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub, "The geojson format," Tech. Rep., 2016.

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

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

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

[13] 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.

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

[15] S. Khatchadourian, M. P. Consens, and J. Siméon, "Having a chuql at xml on the cloud." in *AMW*. Citeseer, 2011.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1099–1110.

[17] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language." in *OSDI*, vol. 8, 2008, pp. 1–14.

[18] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon, "Xml path language (xpath)," *World Wide Web Consortium (W3C)*, 2003.

[19] J. Robie, G. Fourny, M. Brantner, D. Florescu, T. Westmann, and M. Zaharioudakis, "Jsoniq-the sql of nosql 1.0 jsoniq."

[20] E. Meijer, B. Beckman, and G. Bierman, "Linq: reconciling object, relations and xml in the. net framework," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 706–706.

[21] E. Meijer, "The world according to linq," *Queue*, vol. 9, no. 8, p. 60, 2011.