GeCo: Towards a Flexible Query Language for Heterogeneous Collections of Geo-referenced JSON Objects

Steven Capelli,,
Paolo Fosci and Giuseppe Psaila
University of Bergamo - DIGIP
Viale Marconi 5
24044 Dalmine (BG) - Italy
Email: steven.capelli, paolo.fosci, giuseppe.psaila@unibg.,it

Fabio Marini
GN Informatica
Calusco d'Adda (BG) - Italy
Email:

***Abstract*—In the era of Big Data and Open Data, the heterogeneity of data sets to process and integrate for analysis purposes is a matter of fact. Furthermore, such data sets are often geo-referenced, because they describe territories. For this reason, a NoSQL DBMS like *MongoDB* is widely used, which stores collections of heterogeneous JSON objects and supports spatial representation and querying by means of GeoJSON format. However, the query language provided by MongoDB is not suitable for non-programmer users (typically analysts), that prefer a high level query language with declarative operators.**

In this paper, we introduce the *GeCo* query language: it provides a set of declarative operators able to operate on heterogeneous collections of JSON objects, providing explicit support for spatial queries on geo-referenced JSON objects. The *GeCo* query language relies on an intuitive execution model that permits to write reusable queries and that can easily host new and complex operators defined in the future.

I. Introduction

The buzzword *Big Data* is used with several possible meaning. The obvious one is "volume", but another meanings are related to the adverbs "variety" ([1]–[3]): complex analyses require to integrate several data sets coming from different sources of information.

Very often, these sources of information are *Open Data* portals, where public administrations publish data sets concerning several aspects of territories and citizenship. There is not a standard for those data sets: in spite of the fact that usually they are JSON collections or CSV files or XML documents, every single data set has a specific structure, with specific field names, even though they describe similar topics. Often, these data sets contain geo-referenced information.

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

These considerations motivate a large use of *NoSQL* databases [4]–[6], because traditional relational/SQL databases are unable to flexibly integrate so heterogeneous data sets. Nowadays, the most famous NoSQL DBMS is *MongoDB* [7], [8]: it deals with collections of JSON objects, in such a way within the same collection objects with different structures can be gathered without any limitation.

However, the query language provided by MongoDB it is not easy to use for non programmers: it is strongly based on the object-oriented paradigm, and complex operations on collections are not easy to write. Furthermore, the geographical support is provided by means of *GeoJSON* [9], [10] and several functions that implements spatial operations on such a data format. The probelm is that it is complicated writing complex operations on that involves several colelctions and, possibly, spatial operations.

To overcome this problems and reducing the distance between MongoDB and users (possibly geographers and analysts) we decided to define a novel flexible query language for heterogeneous collections of possibly geo-referenced JSON objects, named *GeCo* (Geographical Collections). Our vision is the following: We want to provide users with a declarative query language, that is able to express complex query processes, that could be executed several times; the operators allow to directly work at the *collection level*, i.e., they express transformations on collections, for example, by filtering objects or aggregating objects is two or more collections; the operators natively deal with spatial representation and provide high level spatial operations. We were inspired by early works [11], [12] about a query language for heterogeneous, altough structured, collections of geo-referenced data.

GeCo is not closed, in the sense that we think about it as a continuously growing language: as far as new needs arise, new operators can be defined and added to the language. The approach is then open to new developments.

In this paper, we present the basic and minimal operators we considered necessary in the GeCo query language, 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. This paper is the first step on this research line, and our goal is to validate the approach, showing the fundamental operators and their applicability. In our future work, we will address various application areas, in order to identify specific needs and define new operators.

The paper is organized as follows:

II. Running Example

Listing 1. GeoCollection_CityA_Polygon

```

{ "name":"building",
  "city":"city 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] ]]
               }
             ]

     }

}
```

Listing 2. GeoCollection_CityA_Line

```

{ "name":"water line",
  "city":"city A",
  "geometry":{"type":"GeometryCollection",
              "geometries":[
               {
                "type": "LineString",
                "coordinates":
                    [[102.0, 0.0],
                     [103.0, 1.0],
                     [104.0, 0.0],
                     [105.0, 1.0]]
               }
              ]
             }
}
```

The Listings 1,2 show JSON files. They have the structure supported by GeCo query language. They can have any number of attributes and the attributes can have any name and value.

In the Listings shown, there are three attributes: *name*, *city* and *geometry*. The attribute *geometry* has to exist in any JSON file used by Geco query language. This attribute contains an object with geo spatial information which follows the standard GeoJSON.

The GeoJSON objects are characterized by an attribute *type* with spicific values (i.e Feature, FeatureCollection, GeometryCollection) and an attribute with geo spatial information.

The GeCo query language uses the GeoJSON object with *type* equals to *GeometryCollection* and an attribute *geometries* with geo spatial information.

The Listings 1,2 contains informations which can be queried by GeCo query language. The querying of informations are executed using operators shown in IV.

III. Execution Model

Before introducing the operators, we specify the execution model.

Definition 1 (Collections and Databases): A *Database* db is a set of collections $db = \{c_1, \dots, c_n\}$. Each collection c has a name *c.name* (unique in the database) and an instance *Instance(c)*$= [o_1, \dots, o_m]$ that is a vector of JSON objects o_i.

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

Definition 2 (Query Process State): A *state* s of a query process is a tuple $s = (tc, IR)$, 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 applications of an operator *op*, denoted as $\overline{op}$, is defined as

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

where both domain and codomain 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., $q = \langle \overline{op}_1, \dots, \overline{op}_n \rangle$, with $n \geq 1$.

Thus, the query is a sequence of operator applications; each of them 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 $q = \langle \overline{op}_1, \dots, \overline{op}_n \rangle$, a query process QP is a sequence of query process states $QP = \langle s_0, s_1, \dots, s_n \rangle$, such that $s_0 = (tc : [], IR : \emptyset)$ and, for each $1 \leq i \leq n$, it is $\overline{op} : s_{i-1} \rightarrow s_i$

The query process starts from the empty temporary collection $s_0.tc$ and the empty intermediate results database $s_0.IR$. Thus, the GeCo query language 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

TABLE I. TYPE OF OPERATORS

	Operators
Start operators	GET COLLECTIONS OVERLAY COLLECTIONS JOIN COLLECTIONS MERGE COLLECITONS INTERSECT COLLECTIONS SUBTRACT COLLECTIONS
Carry on operators	FILTER GROUP BY UNWRAP SET INTERMEDIATE AS SAVE AS DERIVE GEOMETRY
Alias operators	SET ALIAS

TABLE II. MEANING OF TERMS

Terms	Meaning of terms
dbName	name of a persistent database
collectionName	name of a collection
attributeName	name of a aatribute
list_of_collections	list of collections separated by comma (,)
list_of_attributes	list of attributes separated by comma (,)
value	value of a attribute
dbName.collectionName	collection inside a persistent database
collectionName.attributeName	attribute of a collection

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 main classes of operators (see Table I): *start operators* and *carry on operators*, that will be described in the next section. A third class, named *Alias Operators*, encompasses practical operators to define aliases.

As a final consideration, observe that the reason why the intermediate results database IR is part of query process states is *isolation*: it exists only during the query process and in case of parallelism, each parallel process has its own intermediate results database.

IV. GeCo Operators

GeCo operators can be separated in three groups as it shown in Table I. The *Start operators* can be used to start a new task or subtask. Their input can be collections inside the persistent or intermediate databases and temporary collections.

The *Carry on operators* cannot be used to start new task or subtask. They can only use temporary collections or intermediate collections inside *IR* database as input.

The *Alias operators* are used to define alias inside the query. The *aliasName* are valide after *alias operator* execution. For this reason it is suggested to use this operator before to all other operators, inside the query.

Below, in the syntax of operators, we will use the terms in Table II in order to exaplain the operators. The brackets indicates optionality.

A. Start operators

The *GET COLLECTION* operator permits to get a collection and make it the new temporary collection. The collection can be inside a persistent database, a temporary collection or an intermediate collection. The syntax of the operator is:

```
GET COLLECTION [dbName.]collectionName;
```

Consider *dbName* the name of persistent database and *collectionName* the name of a collection *c* inside of *dbName* database. The *GET COLLECTIONS* operators returns a JSON object corrisponding to the collection *dbName.collectionName*. When the *dbname* is missing, we implicitely refer to an intermediate collection in the IR database.

The *OVERLAY COLLECTION* operator makes the geo-spatial join between two collections and it puts the resulting object into a new temporary collection. The input collections can be took by a persistent database, a temporary collection or an intermediate collection. This operator makes available a clause to compare no-geo-spatial attribute and some clauses to decide the geo-spatial and no-geo-spatial output. The syntax of operator is:

```
[LEFT|RIGHT|FULL] OVERLAY COLLECTIONS
                  [dbName.]collectionName1,
                  [dbName.]collectionName2
[ON [dbName.]collectionName.attributeName=
[dbName.]collectionName.attributeName]
KEEP (INTERSECTION|RIGHT|LEFT|ALL);
```

Consider *dbName* the name

The *OVERLAY COLLECTIONS*The *KEEP* clause indicates what geometry must return. *KEEP INTERSECTION* returns the geometry of intersection of collections. *KEEP LEFT* returns the geometry of first collection.*KEEP RIGHT* returns the geometry of second colelction. *KEEP ALL* returns the complete geometry of the collections with their intersection.

The *LEFT OVERLAY*, *RIGHT OVERLAY* and *FUUL OVERLAY* work like LEFT, RIGHT and FULL join in SQL. These clause take into account the no-geometry part of the objects. The result of *LEFT OVERLAY* will be all objects of the left side of the instruction, in case of *KEEP LEFT*. In case of *KEEP RIGHT*, the object of the right side of the instruction will be null if there is no a match with left side, otherwise the object of the right side will be returned with all its content. The *RIGHT OVERLAY* works like a mirror of *LEFT OVERLAY* instruction.

The *FULL OVERLAY* takes all objects of right and left side.

```
[LEFT|RIGHT|FULL]JOIN COLLECTIONS
              [dbName.]collectionName1,
              [dbName.]collectionName2 ON
[dbName.]collectionName.attributeName =
[dbName.]collectionName.attributeName;
```

The *JOIN COLLECTIONS* instruction makes inner join between two collections comparing specific no-geospatial-attributes. The *LEFT JOIN*, *RIGHT JOIN* and *FULL JOIN* behave like in SQL.

These instructions are indipendent of others instructions.

```
[ALL] MERGE COLLECTIONS list_of_collections;
```

The *MERGE COLLECTIONS* instruction makes a merge between collections creating an object with inside the shared objects between the collections. The list_of_collections is separate by comma (,).

The *MERGE COLLECTIONS* with *ALL* clause makes a marge between collections creating an object with inside all objects in the collections including any duplicates.

```
INTERSECT COLLECTIONS [dbName.]collectionName1,
                      [dbName.]collectionName2;
```

The *INTERSECT COLLECTIONS* instruction keep all objects shared between the collections. It works on no-geospatial-attribute.

```
SUBTRACT COLLECTIONS [dbName.]collectionName1,
                     [dbName.]collectionName2;
```

The *SUBTRACT COLLECTIONS* instruction keep all objects of the right collection which are not present in the left collection. It works on no-geospatial-attribute.

(ii) The commands which follows process are:

```
FILTER
CASE: attributeName=Value, [WITH geometry]
  [WHERE attribute=value]
  [PROJECT list_of_attributes]
CASE: attributeName=Value, [WITHOUT
                            attributeName]
  [WHERE attribute=value]
  [PROJECT list_of_attributes]
CASE: attributeName=Value
  [WHERE attribute=value]
  [PROJECT list_of_attributes]
KEEP OTHERS
Or //if exist keep others,
     drop others does not exist
DROP OTHERS;
```

The *FILTER* instruction permits to filter data using the *case* clause. In the *case* clause is possible to use *WITH* and *WITHOUT* clause which checked the presence and no-precsence of attributes respectively. *KEEP OTHERS* and *DROP OTHERS* permit to keep or drop the data which did not pass the filter. This behavior makes horizontal filtering.

The *FILTER* instruction supports vertical filtering. This behavior is equivalent to projection. This type of filter is used when in the *case* clause is not present the *WHERE* clause.

The *PROJECT* clause indicates which attributes have to be projected.

```
GROUP BY list_of_attributes
INTO fieldName
SORTED BY list_of_attributes;
```

The *GROUP BY* instruction permits to group the data with specific criteria and store the result into a field. The result can be shorted with specific criteria.

```
UNWRAP FIELD fielName
INTO value;
```

The *UNWRAP* instruction allows to unwrap a field into a value. It creats a number of objects equals to the number of values in the field *fieldName*. Each object contains a field with name *value* with inside one value of field *fieldName* and all other field of original object.

```
SET INTERMEDIATE AS Name;
```

The *SET INTERMEDIATE AS* instruction creates an intermediate state of process named ”Name” to which you can refer.

```
SAVE AS dbName.collectionName;
```

The *SAVE AS* instruction saves a collection in persistent way on database. The instruction save the collection named ”collectionName” into database named dbName.

```
DERIVE GEOMETRY POINT(latitude,longitude);
```

or

```
DERIVE GEOMETRY attribute;
```

This instruction with *POINT* clause, put all the information about the point in a field named geometry according to gecoGeometry. The command only with attribute convert the attribute from GeoJson standard to gecoGeometry.

These type of instructions (instructions which follows process) takes in input the output of previous command.

(iii) The command to define alias name is:

```
SET ALIAS Name aliasName;
```

The *SET ALIAS* instruction permits to set a new name for a variable.

IEEEtran.cls version 1.7 and later.

mds

January 11, 2007

B. Example

Image to have three collections as shows in Listing 1 where is represents a building in the city A, 2 where is represents the water line of the city A. Image you want to know if the water line of the city A passes under the building of the city A in order to manage the maintenance of the water line of the city A.

The geco query language permits easy to get this information using *OVERLAY* istructions. The Listings 3 shows an example.

Listing 3. Example OVERLAY

```
GET COLLECTION GeoCollection_CityA_Polygon;
FILTER
  CASE: WITH geometry
    WHERE city="city A"
DROP OTHERS;
SET INTERMEDIATE AS Collection1CityA;
GET COLLECTION GeoCollection_CityA_Line;
```

```
 8 FILTER
 9   CASE: WITH geometry
10     WHERE city="city A"
11 DROP OTHERS;
12 SET INTERMEDIATE AS Collection2CityA;
13 OVERLAY COLLECTIONS Collection1CityA,
14                     Collection2CityA;
15 SAVE AS db.OverlayCollections;
```

Showing Listing 3, the first row permits to get a collection named ”GeoCollection_CityA_Polygon”, the FILTER in rows 2 and 8 permits to keep only the collection owns a field geometry and a field city equals to ”’city A’. The row six permits to set an intermediate state named ”Collection1CityA”.

V. Related Work

[13]

VI. Concludesions

VII. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] M. Chen, S. Mao, and Y. Liu, “Big data: a survey,” *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[2] D. Laney, “3-d data management: controlling data volume, velocity and variety,” META Group, Tech. Rep. META Group Research Note, February 2001.

[3] V. Mayer-Schönberger and K. Cukier, *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013.

[4] 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.

[5] 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.

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

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

[8] 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.

[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] 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.