J-CO: 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: fabio.marini@gninformatica.com

***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 *J-CO* 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 *J-CO* 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 ([4]) 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 publicly 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 [5]–[7], because traditional relational/SQL databases are unable to flexibly integrate so heterogeneous data sets. Nowadays, the most famous NoSQL DBMS is *MongoDB* [8], [9]: 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. Moreover, it is even impossible to write operations that cross-combine two collections (unless an external program is written).
Furthermore, the geographical support is provided by means of *GeoJSON* [10], [11] and several functions that implements spatial operations on such a data format. The problem is that it is complicated writing complex operations on that involves several collections 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 *J-CO* (which stands for JSON 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 in two or more collections; the operators natively deal with spatial representation and provide high level spatial operations. We were inspired by early works [12], [13] about a query language for heterogeneous, altough structured, collections of geo-referenced data.
Ambitiously, we can say that we are trying to repeat what happened with SQL in the 70s, i.e., enabling database technology for non-programmer users with a declarative query language.

J-CO is not complete, 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 J-CO 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. in Section II, we present the data model and a toy running example that will be exploited along with the paper. Section III presents the execution model on which J-CO relies; thus way, Section IV can effectively introduce the operators. A complete example is presented in Section V, to illustrate the potential application of J-CO. Section VI presents the implementation of the prototype and discuss experimental results. Finally, Section VII discusses related works and Section VIII draws the conclusions.

II. Data Model

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

As far as spatial representation is concerned, we rely on the GeoJSON standard. 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 *Instance(c)*$= [o_1, \ldots, o_m]$ that is a vector of JSON objects o_i.

Thus, we need operators (see Section III) 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 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 `Water Networks` (see Listing 2). Finally, the third one is named `Restaurants` (see Listing 3).

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",
  "city":"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"
 },

 { "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]]]
         }
     ]
   }
}]
```

J-CO is able to query the informations inside *ToyDB* database by using the operators shown in Section IV. We will use the *ToyDB* database and its informations, in the next Sections in order to explain J-CO query language.

III. Execution Model

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., $q = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle$, 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 $q = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle$, 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 $\overline{op} : s_{i-1} \rightarrow s_i$

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 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 I): *start operators* and *carry on operators*, that will be described in the next section.

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. J-CO Operators

J-CO operators can be classified into two groups, reported in Table I: *Start operators* and *Carry on operators*.

Start operators can be used to start a new processing (sub)task in the query. Their inputs are collections coming from a persistent database or from the intermediate results databases *IR*. Their output is a new temporary collection computed from input collections.

TABLE I. TYPE OF OPERATORS

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

TABLE II. MEANING OF TERMS

Terms	Meaning of terms
dbName	name of a persistent database
collectionName	name of a collection
fieldName	name of a field
list_of_collections	list of collections separated by comma (,)
list_of_fields	list of fields separated by comma (,)
value	value of a attribute
list_of_field_values	list of fields comparated by values
dbName.collectionName	collection inside a persistent database
collectionName.fieldName	field of a collection

Carry on operators can continue the processing (sub)task. They implicitly take the temporary collection and possibly produce a new version of it. Furthermore, they can store collections into the intermediate results database *IR*, or into a persistent databases.

Hereafter, we will make use of terms reported in Table II to introduce the syntax of operators and explain their behaviours. Regarding the notation for the syntax of operators, we will make use of the $\star$ symbol to denote denote 0 or more repetitions and of the + symbol to denote 1 or more repetitions; square brackets denote optionality; the vertical bar | separates alternatives. For example, *([dbName.]collectionName)+* denotes that a non empty list of collection names is required, where each one could come from a persistent database (when the optional *dBName.* is specified) or from the intermediate results database *IR* (when the optional *dBName.* is not specified).

A. Start Operators

1) GET COLLECTION*:* The **GET COLLECTION** operator permits to get a collection from a database (persistent or intermediate) and make it the new temporary collection. The syntax of the operator is:

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

When the *dbname* is specified, the JSON collection named *collectionName* uis retrieved from the persistent database named *dbname*; otherwise, it is retrieved from the intermediate result database *IR*.

2) OVERLAY COLLECTIONS*:* The **OVERLAY COLLECTIONS** operator makes the geospatial join between two collections; the result becomes the new temporary collection. The input collections can come from a persistent database or the intermediate results database *IR*. The syntax of the operator is hereafter.

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

The output of the **OVERLAY COLLECTIONS** operator is a JSON collection containing the result of the geospatial join between *dbName.collectionName1* (left collection) and *dbName.collectionName2* (right collection). For each object l_i in the left collection and an object r_j in the right collection, an object $o_{i,j}$ appears in the output collection if the geometries of l_i and r_j overlaps (intersect), and the optional *selectionCondition* (expressed on fields of the two objects) is true. The output object $o_{i,j}$ has three fields: one with the name of the left collection, one with the name of the right collection, a `geometry` field.

The **KEEP** clause permits to specify the content of field `geometry`. IF **KEEP INTERSECTION** is specified, field `geometry` in $o_{i,j}$ represents the spatial intersection (for instance, the intersection of two crossing line is a point); if **KEEP LEFT** (resp., **KEEP RIGHT**) is specified, field `geometry` in $o_{i,j}$ represents the full geometry of the left object l_i (resp., of the right object r_j); if **KEEP ALL** is specified, field `geometry` in $o_{i,j}$ represents the union of geometries in both the left object l_i and the right object r_j.

The alternative options **LEFT**, **RIGHT** and **FULL** at the beginning of the operator slightly change the behaviour of the operator when specified. If the **LEFT** (resp., **RIGHT**) option is specified, all non-matching objects $\overline{l}_k$ in the left collection (resp., all non-matching objects $\overline{r}_k$ in the right collection) have a corresponding object $\overline{o}_k$ in the output collection, such that the field in $\overline{o}_k$ corresponding to the right (resp., left) object is not present, and its `geometry` field coincides with the geometry of $\overline{l}_k$ (resp., $\overline{r}_k$). The **FULL** option specifies that all non-matching objects $\overline{l}_k$ and $\overline{r}_k$) must have a corresponding object $\overline{o}_k$ in the output collection.

Example 2: Consider collection `Buildings` shown in Listing 1and collection *WaterLine* shown in Listings 2, stored in database `ToyuDB`. 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:

```
OVERLAY COLLECTIONS ToyDB.Buildings,
                    ToyDB.WaterLines
ON Buildings.City = "City A"
KEEP INTERSECTION;
```

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:{...,...,...}
  }
]
```

Notice that field *Buildings* contains the matching object b_i coming from the left collection, while field `WaterLines` contain s 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.

3) JOIN COLLECTIONS*:* The **JOIN COLLECTIONS** operators makes the no-geospatial join between two collections. W.r.t. operator **OVERLAY COLLECTIONS**, the **JOIN COLLECTIONS** operator works on non-geospatial fields. The syntax of operator is hereafter.

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

The output of the **JOIN COLLECTIONS** operator is a JSON collection obtained by pairing objects in both collections. For each object l_i in *dbName.collectionName1* (left collection) and for each object r_j in *dbName.collectionName2* (right collection), an object $o_{i,j}$ appears in the output collection if the *joinCondition* is tyre for the pair l_i, r_j. The output object $o_{i,j}$ contains two fields: the first one has the name of the left collection and contain object l_i, while the second one has the name of the right collection and contain object r_j.

Similarly to the **OVERLAY COLLECTIONS** operator, the alternative options **LEFT**, **RIGHT** and **FULL** at the beginning of the operator slightly change the behaviour of the operator, when specified. If the **LEFT** (resp., **RIGHT**) option is specified, all non-matching objects $\overline{l}_k$ in the left collection (resp., all non-matching objects $\overline{r}_k$ in the right collection) have a corresponding object $\overline{o}_k$ in the output collection, such that the field in $\overline{o}_k$ corresponding to the right (resp., left) object is not present. The **FULL** option specifies that all non-matching objects $\overline{l}_k$ and $\overline{r}_k$) must have a corresponding object $\overline{o}_k$ in the output collection.

Example 3: Suppose we are the same company of Example 2. We need to associate restaurants to buildings, based on their address. We can write the following query.

```
JOIN COLLECTIONS ToyDB.Buildings,
                 ToyDB.Restaurants
ON Buildings.City=Restaurants.City AND
   Buildings. Address=Restaurants.Address;
```

The join condition specifies that objects must be pairs if their cities and their address coincides. The output collection is reported in the listing below. Notice that no attribute `geometry` is present at the top level.

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

}]
```

The join in example can be useful in order to get restaurant position information from its address.

4) 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
[dbName].collctionMa (, [dbName].collctionMa)+;
```

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.

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

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

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.

6) SUBTRACT COLLECTIONS*:* The *SUBTRACT COLLECTIONS* operator makes a setoriented -subtraction between collections and puts the resulting object into the temporary collection.

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

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

B. Carry on Operators

This group encompasses operators whose input collection is the temporary collection and possibly generate a new version of it. They are suitable to perform continued transformations on one single collection, thus avoiding the need to continuously refer to collections.

1) FILTER*:* The **FILTER** operator permits to filter objects in the temporary collection, according to some selection conditions, and possibly change the structure of selected objects. The operator is designed to deal with the heterogeneous nature of JSON collections; for ths reason, the syntax is more articulated than other operators.

```
FILTER
(CASE:
 (fieldName = value |
   WITH fieldName |
        WITHOUT fieldName)+
 [WHERE selectionCondition]
 [PROJECT (fieldName)+])+
(KEEP OTHERS|DROP OTHERS);
```

The *FILTER* operator contains one or more **CASE** branches. Each **CASE** branch specifies a subset of objects to select by means a list of selectors. Three types of selectors are provided: equal conditions on fields; **WITH** selectors, that asks for objects with the specified field name; **WITHOUT** selectors, that ask for objects without the specified field name. For any objects in the input temporary collection that matches with the specified selectors, the **WHERE** clause, if specified, is evaluated and if it is false, the object is discarded, otherwise the object is kept. Finally, if the **PROJECT** clause is specified, the object is projected on the specified list of fields, in order reduce the number of fields or extract (by means of a dot notation) fields nested in object fields.

If more than one **CASE** branch is specified, they are evaluate in the order: an object is handled by the first branch that matches with it.

The alternative clauses **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 collection; if **DROP OTHERS** is specified, these objects are not put into the output collection.

Example 4: Consider *Restaurants* shown in Listing 3 which represents some restaurant. we are only interested in the names of restaurants in city `"city C"` having the geometry. The query is hereafter.

```
GET COLLECTION ToyDB.Restaurants;
FILTER
 CASE city="city C" WITH geometry
 PROJECT name
DROP OTHERS;
```

The **GET COLLECTION** operator retrieves the initial `Restaurants` collection from the persistent database; this collection becomes the new temporary collection. The, the **FILTER** operator carries on the process.

Only one **CASE** branch is sufficient for our purpose, with two selectors: the first one is an equal condition on field `City`; the second one is a **WITH** selector on field `geometry`. As a result, only one object in our sample `Restaurants` collection will be selected, and then projected on the fiel `name`. Other possibly not matching objects are dropped.

The new temporary collection produced by the operator is hereafter.

```
[{
 "name":"RestaurantC"
}]
```

Notice the very simple structure of the resulting object.

2) GROUP BY*:* The **GROUP BY** operator groups objects in the input temporary collection based on a list of grouping fields. Here is its syntax.

```
GROUP BY (fieldName)+
INTO fieldName
[SORTED BY (fieldName)+];
```

The **GROUP BY** operator groups the objects in such a way a group contains all the objects $o_1, \ldots, o_n$ having the same values for field names specified after the **GROUP BY** jeywords.

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

Finally, the optional clause **SORTED BY** specifies whether to sort objects into the vector fields. If so, the following field names are the sort keys.

Notice that the output temporary collection contains as many objects as the number of groups.

Example 5: Consider collection *WaterLines* shown in Listing 2. We might be interested in grouping water lines by their city. The query is hereafter.

```
GET COLLECTION ToyDB.WaterLines;
GROUP BY city
 INTO waterLineCity;
```

The **GET COLLECTION** operate retrieves the WaterLines collection from the persistent database and makes it the new temporary collection, on which the **GROUP BY** operator works. Water lines are grouped by attribute City; the name given to the vector 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 represent the group for *city A* (first object) and an other object which represents the grouping for *city C* (second object). The first object contains a field *name* (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*.

3) 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 new temporary collection into the intermediate results database.

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

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

The name of the new collection stored into the persistent database *dbName* is *collectionName*.

5) DERIVE GEOMETRY*:* The *DERIVE GEOMETRY* operator is used to manage geospatial fields, when geospatial fields of JSON input are not conform to the J-CO data model. The input collection is the temporary collection created by previous operator. The syntax of operator is:

```
DERIVE GEOMETRY (POINT(latFieldName,lonFieldName)|
                 fieldName);
```

The **DERIVE GEOMETRY** operator adds the geometry field to each object o_i in the input temporary collection. If the **POINT** pair is specified, the values of the two specified fields plays the role, respectively, of latitude and longitude. If a simple *fieldName* is specified, this is intepreted as a GeoJSON specification whose name is not geometry, or, in case of dot notation expression, it can be in some nested objects.

V. Complete Example

In order to show the effectiveness of J-CO 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 names of restaurants located in buildings in *city A* under which some water lines pass. This query is useful for the water lines maintenance in the *city A*.

Listing 4. Complete Example

```
1  GET COLLECTION ToyDB.Buildings;
2  FILTER
3   CASE city="city A" WITH geometry
4   DROP OTHERS;
5  SET INTERMEDIATE AS BuildingsCityA;
6  OVERLAY COLLECTIONS BuildingsCityA,
7                      ToyDB.WaterLines
8    KEEP INTERSECTION;
9  SET INTERMEDIATE AS BWCityA;
10 JOIN COLLECTIONS BWCityA,
11                 ToyDB.Restaurants
12  ON BWCityA.BuildingsCityA.address =
13     Restaurants.address;
14 FILTER
15  CASE WITH name
16   PROJECT Restaurant.name
```

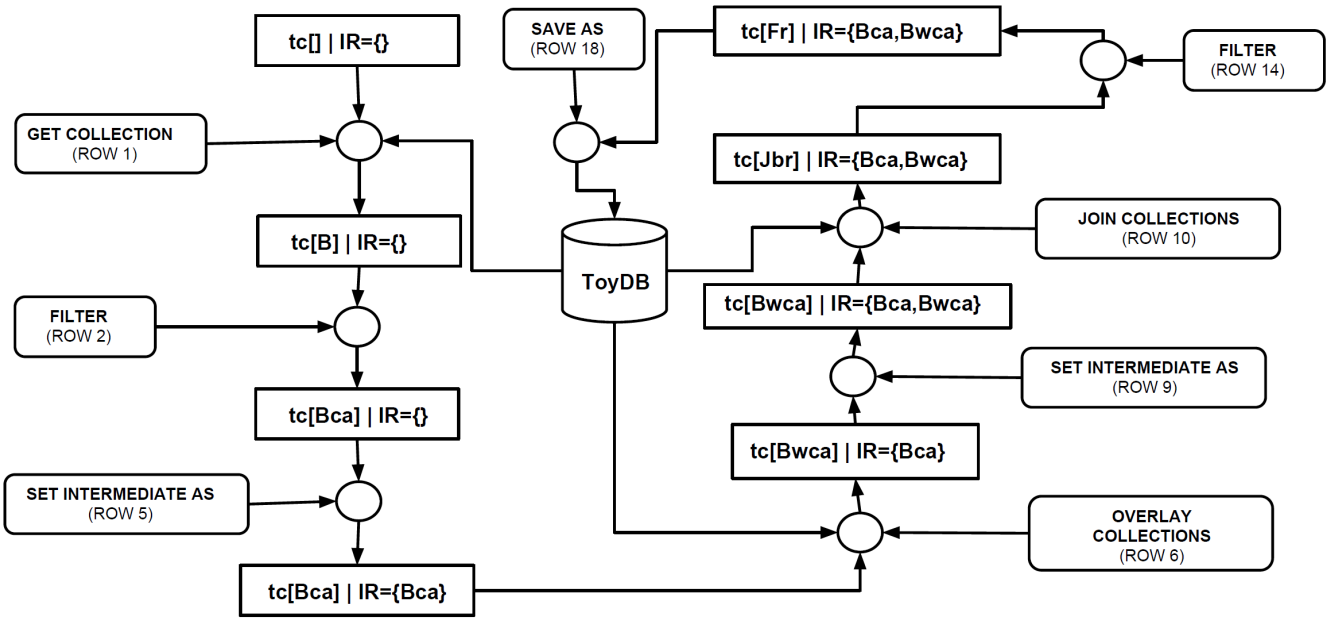

Fig. 1. Execution Model example.

```
17  DROP OTHERS;
18  SAVE AS ToyDB.RestaurantsWL;
```

Figure 1 shows the execution model of the example. It shows how the temporary collection and intermediate database IR change during the process. In Figure 1, for lack of space, we used abbreviated names with the following meaning:

- `B` means `ToyDB.Buildings`,
- `Bca` means `BuildingsCityA`,
- `Bwca` means `BWCityA`,
- `Jbr` means *Output of the join operator* and
- `Fr` means *Output of second filter*.

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

The query process works as follows:

A. New Task

The query starts a new task using the **GET COLLECTIONS** operator (which is a *Start operator*). The **GET COLLECTION** (see row 1) outputs the `ToyDB.Buildings` (shown in Listings 1), into the temporary collection. The Figure 1 shows that the element `B` (`ToyDB.Buildings`) has become the new *tc*.

The temporary collection is now the input collection for the **FILTER** operator. It keeps any buildings located in *"'city A"'* which have the field *geometry* (see its **CASE** clause). The result becomes the new temporary collection. Figure 1 shows that the element `Bca` (`BuildingsCityA`) has become the new *tc*. All others buildings are dropped (see the **DROP OTHERS** option).

The temporary collection produced by *FILTER* is then saved into the IR Fb by the *SET INTERMEDIATE AS* operator. This operator saves *tc* into IR, naming the collection as `BuildingsCityA` (see row 5). Figure 1 shows that the element `Bca` (`BuildingsCityA`) has been added to IR.

B. Subtask 1

In row 6 the **OVERLAY COLLECTIONS** operator starts a new subtask. This operator makes a geospatial join between collection `BuildingsCityA` (previously stored in the intermediate results database IR) and collection `ToyDB.WaterLines` (retrieved from the persistent database `ToyDB`). It outputs the geospatial intersection (see **KEEP INTERSECTION** clause). The output will be saved as the new temporary collection *tc*. Figure 1 shows that the element `Bwca` (`BWCityA`) has become the new *tc*.

The following *SET INTERMEDIATE AS* operator in row 9 saves the temporary collection generated filled by the **OVERLAY COLLECTIONS** operator into IR with thename `BWCityA`. Figure 1 shows that the element `Bwca` (`BWCityA`) has been added to IR.

C. Subtask 2

In row 10 the **JOIN COLLECTIONS** operator starts the final subtask. This operator makes a join between the intermediate collection `BWCityA` (retrieved from IR) and the collection `ToyDB.Restaurants` (retrieved from the persistent database `ToyDB`). It outputs the join between buildings and restaurants which have the same address (see the **ON** clause). The output becomes the new temporary collection. Figure 1 shows that the element `Jbr` (output of the join) has been the new *tc*.

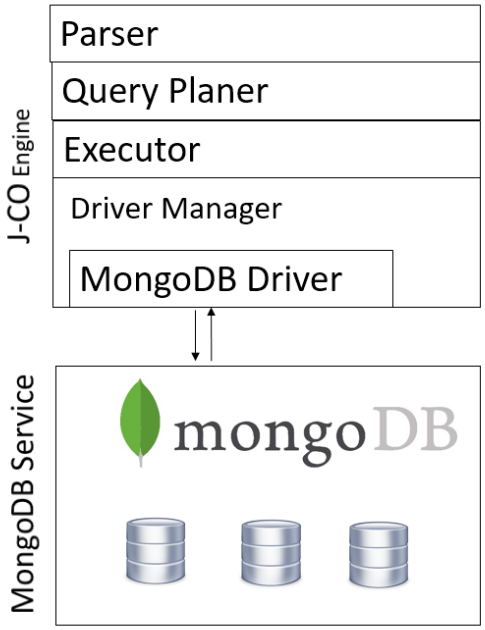

Fig. 2. Architecture of J-CO Engine.

The temporary collection generated by *JOIN COLLECTIONS*, is then filtered by the *FILTER* operator. The operator (see row 14) keeps any restaurants which have field *name* (see, the **CASE** branch) and projects them on the name of this restaurants (see the **PROJECT** clause), generating a new temporary collection. Figure 1 shows that the element `Fr` (output of the **FILTER** operator) has become the new *tc*. All others objects are dropped (see the **DROP OTHERS** clause).

Finally, the current temporary collection *tc*, whichh indeed contains the desired restaurant names, is saved into the persistent database named `ToyDB` by the **SAVE AS** operate, with name `RestaurantsWL`. Figure 1 shows that the element `Fr` (output of the filter) is saved into persistence database *ToyDB*).

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

VI. J-CO Engine

In order to prove the feasibility of our approach, we developed a prototype version of the *J-CO Engine*. In Subsection VI-A we present the architecture; in Subsection VI-B we discuss a performance evaluation that encourages us to carry on the research.

A. Architecture

J-CO Engine is written in Java; it is designed to be an external tool w.r.t. MongoDB: this choice makes the tool substantially independent of the DBMS technology, and opens the way to make J-CO engine able to operate on several others DBMSs. The architecture of J-CO Engine is reported in Figure 2.

The stack of components includes the *Parser*, that t5ransform 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 *Driveer Manager*, that at the moment 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 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 operator (see Figure 3. 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` 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*[1] and *Tsusiat Software Java Topology Suite* [2] are used to deal with spatial representations, while *David Moten's R-tree/R*-tree in memory indexing*[3] is used to build main-memory spatial indexes necessary to implement the `OVERLAT COLLECTIONS` operator.

Main-memory indexes are important to accelerate the execution of complex operators, such as **GROUP BY** and **JOIN COLLECTIONS**. In both case, we create main-memory indexes on the fly, so that we can perform equi-join operations (currently, we support only the equality predicate in join conditions) or grouping with linear complexity, after the indexes are built.

A similar approach is used for the **OVERLAY COLLECTIONS** operator. A spatial index on-the-fly over the geometry field of the left collection using the *R*-Tree* library. For each element of the right collection, a search on the index is done and all the documents of the left collection that intersect with the geometry of the element are retrieved. For each pair, a new JSON object is generated and inserted into the new temporary collection.

B. Performance Evaluation

In order to have a first validation of our approach, we ran some experiments with the prototype of the J-CO 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 e a SS hard disk.

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.

[1] https://github.com/locationtech/spatial4j
[2] JTS - http://tsusiatsoftware.net/jts/main.html
[3] https://github.com/davidmoten/rtree

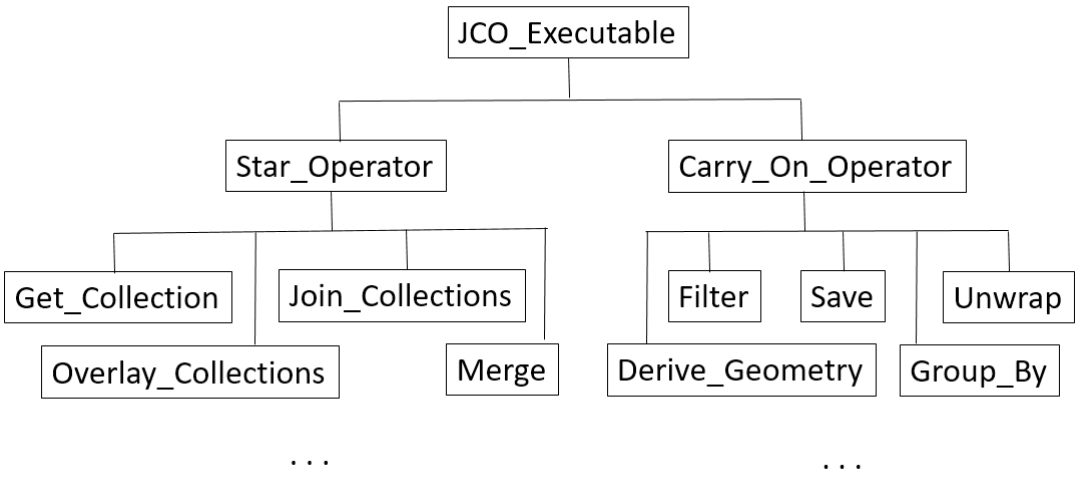

Fig. 3. Class hierarchy for JCO_Executable objects used to build the query plan.

In Table III, we report the execution times (in msec) 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 COLLECTIONS** family, the **OVERLAY COLLECTIONS** and the **GROUP BY** operators. Furthermore, we also tested the **DERIVE GEOMETRY** with the **POINT** option.

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; columns *Indexing time*, *Computation time* and *Total time* reports, respectively, the execution time neded for the indexing phase (when done), the execution time for the computation phase (generation of the output objects) and the total execution time of the operator.

As far as the **JOIN COLLECTIONS** operator is concerned, we tested it by joining the collection *restaurants.json* with itself. Notice that, even though the number of potential pairs was 25360×25360, the total execution time is only a quarter of second.

As far as the **OVERLAY COLLECTIONS** operator is concerned, we overlaid collection *countries.geo.json* (countries) with collection *restaurants.json* (restaurants). IN order to stress the implementation, in the second experiment with the **OVERLAY COLLECTIONS** operator, we duplicated the *restaurants.json* collection to get to 111993 objects. The execution times show that this is the slowest operator, but 3 secs and 15 secs are acceptable times for users (notice that the time needed to build the R*-tree index is negligible).

The **GROUP BY** operator was tested on collection *restaurants.json* (restaurants) with three different settings: 1, 2 and 3 grouping fields. Observe that the execution times are substantially stable and less than half second is needed.

Finally, the **DERIVE GEOMETRY** with the **POINT** option has been tested on collection *restaurants.json* (restaurants). It is able to process about 150000 objects in a quarter of sec.

The results show that the approach is correct. Even though some operators could appear too complex to process, the J-CO engine is able to execute them producing results in seconds or tenth of seconds, depending on the operators. Consequently, we can expect that complex queries can be processed in minutes. Thus, the J-CO query language is effective as far as execution time is considered.

VII. Related Work

J-CO moves from our previous work on the problem of querying heterogeneous collections of complex spatial data [12], [13]. In those 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 rely on the JSON standard, thus we do not define an ad-hoc data model; furthermore, J-CO abandon 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 i [5], where several systems are catalogued ans 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 (see the web sites reported in the footnote[4] for details). Readres interested in NoSQL DBMSs evaluation can refer to [6] and to [7].

As far as query language for JSON objects are concerned, the closest proposal is *Jaql* [14]. It was designed to help Hadoop [15] programmer writing complex transformations,

[4]MongoDB: https://www.mongodb.com/
CouchDB: http://docs.couchdb.org/en/2.0.0/

Operation	Size of C1	Size of C2	Size of Output	Indexing time	computation time	Total time
JOIN COLLECTIONS	25360	25360	446768	194.95	62,6	257.55
OVERLAY COLLECTIONS	176	25360	21336	245.1	3455.3	3700.4
OVERLAY COLLECTIONS	176	111993	111993	870.4	14808.9	15679.3
GROUP BY 1 field	25360		6	201	212.1	413.1
GROUP BY 2 fields	25360		3058	192.9	229	421.9
GROUP BY 3 fields	25360		11854	193.6	248.1	441.7
DERIVE GEOMETRY	25360		25360		220.5	220.5

TABLE III. EXECUTION TIMES (IN MSEC) OBSERVED DURING THE EXPERIMENTS.

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 constructs are at a higher level and truly declarative.

On the same track of query languages for improving MapReduce/Hadoop programming, we cite *ChuQL* [16], 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* [17], 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 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; in contrast, J-CO provides the concepts of temporary collection and intermediate results database. The *DryadLINQ* language, presented in [18], follows a very similar approach to Pig Latin's approach.

J-CO is instead thought as a language oriented to non programmers. In our mind, we would like to replicate what happened with SQL, that enabled database querying to non technicians. Furthermore, J-CO provides operators for transforming geo-referenced objects, that is a common lack of other languages.

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* [19], 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 docuemtns is *XQuery*. Among all features, it provides constructs to generate new documents, as well as the possibility to express complex queries. However, it is still oriented to programmes and not to non programmer users.

Finally, our proposal is somehow related to the world of *PolyStore* DBMS, i.e., database management systems that deal with several DBMS at the same time, each of them possibly providing a different logical model, such as relational, graph, JSON, pure-text, images, videos. An interesting work on this topic is *BigDAWG* [20], [21]. Among all features, the support to querying relies on the querylanguages provided bythe integrated DBMSs,, enriched with a couple of instructions that permit to transform data from one model to another (CAST) and to specify where to store a query results.

Even considering a more classical enterprise environment, where images and video are not considered, the concept of PolyStore DBMS is relevant. An example is the *QUEPA* (QUerying and Exploring a Polystore by Augmentation) system, that provides a solution to give a uniform view of relational DB, data warehouse, JSON data. In this perspective, J-CO could play a significant role in a PolyStore environment, since relational data, CSV files and spatial data managed by spatially augmented object-relational DB such as PostgreSQL/-PostGIS can be viewed as non-nested JSON/GeoJSON objects.

VIII. CONCLUSIONS

In this paper, we proposed a query language, named J-CO, specifically devised to query heterogeneous collections of JSON objects stored in a NoSQL document DBMS such as MongoDB. The idea is to provide non-programmer users with a declarative query language able to handle JSON collections in such a way objects can be filtered, recombined and aggregated in a flexible way. Geo-referenced objects can be queried by means of the `Overlay` operator, allowing the specification of complex spatial queries in a very simple way. The execution model on which the J-CO query language relies is, simple, intuitive, and suitable to express complex queries based on several subtasks. Furthermore, J-CO overcome typical limmitations of query languages provided by document DBMSs, that are usually unable to combine collections in a flexible way.

The prototype we realized demonstrate the feasibility of the approach, and the performance we obtained,, even though the implementation is not particularly optimized, are encouraging. We expect, at the end of development, that J-CO could become a powerful and effective tool for querying databases of JSON collections.

At this moment, we do not think J-CO is complete. We are planning to extend current operators with new features able to better deal with nesting, such as specific aggregation functions possibly based on geo-references. But above all, we are going to define new powerful operators for spatial analysis over big data, since territorial analysis (with the contribution of possibly geo-referenced open data published by institutions) are becoming more and more important in day-by-day life of institutions and citizens.

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] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska *et al.*, "A demonstration of the bigdawg polystore system," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1908–1911, 2015.

[21] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker, "The bigdawg polystore system and architecture," *arXiv preprint arXiv:1609.07548*, 2016.