Supplemental material for the generic GraphQL resolver function

No Author Given

1 RML (RDF Mapping Language)

RML [1,2] is a declarative mapping language for linking data to ontologies [3]. An RML document has one or more `Triples Maps`, which declare how input data is mapped into triples of the form (subject, predicate, object). An example of RML mappings is shown in Listing 1.1. A `Triples Map` contains the following three components (`Logical Source`, `Subject Map` and a set of `Predicate-Object Maps`). A logical source declares the source of input data to be mapped. It contains definitions of `source` that locate the input data source, `reference formulation` declaring how to refer to the input data, and `logical iterator` declaring the iteration loop used to map the input data. For instance, line 2 to line 6 in Listing 1.1 constitute the definition of a logical source. The definition declares that the data source is a JSON-formatted data source on the Web and also describes the way of iterating the JSON-formatted data (line 5). A subject map declares a rule for generating subjects when transforming underlying data into triples, including how to construct URIs of subjects (e.g., line 8) and specifying the concept to which subjects belong (e.g., line 9). A predicate-object map consists of one or more predicate maps declaring how to generate predicates of triples (e.g., line 12), and one or more object maps or referencing object maps defining how to generate objects of triples. An object map can be a reference-valued term map or a constant-valued term map. The former declares a valid reference to a column (relational data sources), or to an object (JSON data sources). The latter declares the value of the object as constant data. For instance, line 39 to line 41 make up a reference-valued term map. Line 19 to line 25 constitute a definition of a referencing object map including the join condition based on two triples maps. A referencing object map refers to another triples map (called a parent triples map) by using a `rr:joinCondition` property to state the join condition between the current triples map and the parent triples map. A join condition contains two properties, `rr:child` and `rr:parent`, of which the values must be logical references to logical sources of the current triples map and the parent triples map, respectively.

Listing 1.1: An example of RML mappings transforming university domain data.

```
1 <UniversityMapping>
2  rr:logicalSource [
3    rml:source "http://example.com/universities.json";
4    rml:referenceFormulation ql:JSONPath;
5    rml:iterator "$.data.universities[*]";
6  ];
7  rr:subjectMap [
```

```
    rr:template "http://example.com/university/{uid}";
    rr:class schema:University;
  ];
  rr:predicateObjectMap [
    rr:predicate schema:UniversityID;
    rr:objectMap [
      rml:reference "uid";
    ];
  ];
  rr:predicateObjectMap [
    rr:predicate schema:departments;
    rr:objectMap [
      rr:parentTriplesMap <DepartmentMapping>
      rr:joinCondition [
       rr:child "uid";
       rr:parent "university_id";
      ];
    ];
  ].

<DepartmentMapping>
  rr:logicalSource [
    rml:source "http://example.com/departments.csv";
    rml:referenceFormulation ql:CSV;
  ];
  rr:subjectMap [
    rr:template "http://example.com/department/{department_id}";
    rr:class schema:Department;
  ];
  rr:predicateObjectMap [
    rr:predicate schema:DepartmentID;
    rr:objectMap [
      rml:reference "department_id";
    ];
  ];
  rr:predicateObjectMap [
    rr:predicate schema:head;
    rr:objectMap [
     rml:reference "HEAD";
    ];
  ].
```

2 The *Evaluator* algorithm

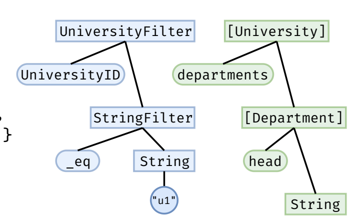
We present the details of *Evaluator* in Algorithm 1 and show an example in Figure 2 of how evaluators work for answering the query in Figure 1a.

```
{
  UniversityList(
    filter:{
      UniversityID:{
       _eq:"u1"}
    }){
    departments{
      head
    }
  }
}
```

(a) Query

```
{
  "data":{
   "UniversityList":[
   {
    "departments":[
     {"head":"Harry,Potter"},
     {"head":"Sheldon,Cooper"}
    ]
   }]
  }
}
```

(b) Query response



(c) Abstract Syntax Trees

Fig. 1: GraphQL query, response, and ASTs for the input argument and query fields.

An AST and a number of triples maps from the semantic mappings are essential inputs to the algorithm. For a given AST, we can obtain the object type and fields that are requested in the query based on the root node and child nodes, respectively (line 2). For instance, taking the ASTs in Figure 1c as examples, the root type and the field for evaluating the filter expression are `University` and `UniversityID`, and the root type and the first level requested field for evaluating query fields are `University` and `departments`, respectively. After getting the relevant triples maps based on the root node type (line 4 in Algorithm 1, e.g., `UniversityMapping` in Listing 1.1) or from the argument (line 28, the parent triples map, `DepartmentMapping`, which is an argument in the recursive call of an evaluator), the algorithm iterates over triples maps and merges the data obtained based on each triples map (line 5 to line 30). Exploring this in more detail, the algorithm parses each triples map to get the logical source and relevant predicate-object maps (line 8 and line 9). As described in Section 1, there are three different types of predicate-object map depending on the different maps of object, which are a reference-valued term map, a constant-valued term map or a referencing-object map. The algorithm iterates over the predicate-object maps and parses each one (line 10 to line 16). For a reference-valued term map, the mapping between the predicate and the reference column or attribute is stored (line 12, e.g., {`UniversityID: uid`} is stored in *pred_attr*), which will be used for rewriting a filter expression according to the underlying data source (line 18, e.g., `uid = 'u1'`), annotating the obtained underlying data (line 21, e.g., `HEAD` is annotated as `head` for *Department* data). For a constant-valued term map, the mapping between the predicate and the constant data value and type is stored (line 14). Both *pred_attr* and *pred_const* will be used to annotate the data from underlying sources (line 21).

In the phase of evaluating a filter expression, *local_filter*, which represents the rewritten filter expression, is a necessary argument when sending requests to underlying data sources (line 19). While in the phase of evaluating query fields, *filter_ids*, being a *NULL* value or having at least one element, is a necessary argument (line 19, arrow `(a)` in Figure 2). A *NULL* value represents the fact that the GraphQL query does not include an input argument. After obtaining the data from the underlying data sources, the data is serialized into JSON format (key/value pairs) in which the keys are predicates stated in the predicate-object map (line 21), where each predicate corresponds to a field in the GraphQL schema. In the next step, the algorithm iterates over predicate-object maps in which the object map refers to another triples map (called a parent triples map) (line 22 to line 29). An evaluator is called again to fetch data based on this parent triples map (line 28, arrow `(4)` in Figure 2). For the query example, the parent triples map refers to the `DepartmentMapping`. Since such a referencing-object map definition states the join condition between the current triples map (`UniversityMapping`) based on $child\_field$ (`uid`) and the parent triples map (`DepartmentMapping`) based on $parent\_field$ (`university_id`) (line 21 to line 23 of the mappings in Listing 1.1), we can pass referencing data (ref), which contains the data obtained according to the current triples map and $parent\_field$, to the call of an evaluator when we fetch data according to the parent triples map (line 28). Such referencing data is taken into account, in the recursive call to an evaluator, when the request is sent to the underlying data sources (line 19,

Algorithm 1: *Evaluator*

```
   Input  : an Abstract Syntax Tree: ast;
            the semantic mappings: triples_maps;
            the referencing data: ref;
            the identifiers for filtered out result: filtered_ids
   Output: result of evaluating a filter expression or query fields
1  Initialize an empty list: result
2  get the root type and query fields from ast: root_type, query_fields
3  if triples_maps is Empty then
4      get relevant triples maps based on the root_type: triples_maps
5  for tm in triples_maps do
6      Initialize an empty list: referencing_poms
7      Initialize two empty lists: pred_attr, pred_const
8      get the logical source from tm: source
9      get all the predicate-object maps from tm based on query_fields: poms
10     for pom in poms do
11         if object_map in pom is a reference-valued term map then
12             extend pred_attr with a map between the predicate and
                 column/attribute
13         if object_map in pom is a constant-valued term map then
14             extend pred_const with a map between the predicate and data
                 value, type
15         if object_map is a referencing-object map term map then
16             extend referencing_poms with pom
17     parse ast and get the filter expression: filter_expr
18     localize filter_expr based on pred_attr: local_filter
19     access the data source based on source, local_filter, ref, filtered_ids:
         temp_result
20     if temp_result is not Empty then
21         annotate temp_result based on pred_attr, pred_const
22         for (pred, object_map) in referencing_poms do
23             get the sub tree from ast based on pred: sub_ast
24             parse object_map: parent_triples_map, join_condition
25             parse join_condition: child_field, parent_field
26             get the referencing data from temp_result on child_field:
                 child_data
27             ref = (child_data, parent_field)
28             call Evaluator based on sub_ast, parent_triples_map, ref:
                 parent_data
29             join temp_result and parent_data based on join_condition, pred:
                 temp_result
30     merge result and temp_result: result
31 return result
```

arrow (b) in Figure 2). After the data is obtained according to the parent triples map (arrow (c) in Figure 2), it is joined with data obtained according to the current triples map (line 29, frame (A) in Figure 2).

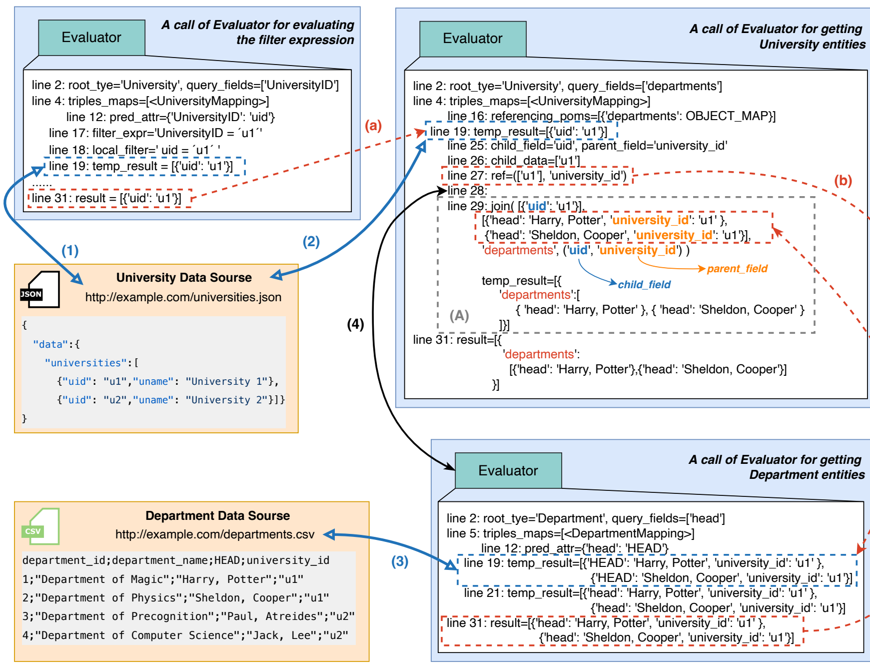


Fig. 2: An example for answering the query in Figure 1a, (1)-(3) indicate the requests to and responses from the data sources; (a)-(c) indicate the parameter passing between the calls to *Evaluators*; (4) indicates a recursive call to *Evaluator* for getting the data of *Departments*; frame (A) indicates a join operation.

References

1. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In: Proceedings of the Workshop on Linked Data on the Web co-located with the 23rd International World Wide Web Conference (WWW 2014). vol. 1184 (2014), `http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf`
2. Dimou, A., Vander Sande, M., Slepicka, J., Szekely, P., Mannens, E., Knoblock, C., Van de Walle, R.: Mapping Hierarchical Sources into RDF Using the RML Mapping Language. In: 2014 IEEE International Conference on Semantic Computing. pp. 151–158 (2014). https://doi.org/10.1109/ICSC.2014.25
3. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking Data to Ontologies. In: Journal on Data Semantics X, pp. 133–173. Springer (2008). https://doi.org/10.1007/978-3-540-77688-8_5