KGRAM: a Generic SPARQL Interpreter for Linked Data Querying and Mashup

Olivier Corby a, Catherine Faron-Zucker b, *

a Edelweiss, INRIA Sophia Antipolis, 2004 route des Lucioles, BP 93, F-06902 Sophia Antipolis cedex, France
E-mail: olivier.corby@sophia.inria.fr

b I3S, Université de Nice Sophia Antipolis, CNRS, 930 route des Colles, BP 145, F-06930 Sophia Antipolis, France
E-mail: catherine.faron-zucker@unice.fr

Abstract. In this paper we present the KGRAM engine dedicated to the querying and mashup of linked data. KGRAM is an acronym for Knowledge Graph Abstract Machine. It is designed as the interpreter of an abstract language which generalizes SPARQL (its upcoming 1.1. version) to query not only RDF but also any knowledge graph model. KGRAM comes with an application programming interface which corresponds both to the abstract primitives of this query language and to abstract graph data structures and KGRAM’s evaluation function only manipulates those abstract structures. We describe KGRAM’s abstract query language and its interpreter’s evaluation function and KGRAM’s application programming interface, and we explain how it can be used to the querying and mashup of linked data.

Keywords: Semantic Search, Semantic Mashup, SPARQL 1.1. Interpreter, KGRAM, Graph Abstract Machine, Knowledge Graphs, Linked Data

1. Introduction

In this paper we present the KGRAM engine dedicated to the querying and mashup of linked data. It is an open source software that can be downloaded here1. KGRAM is quite a mature software. It benefited from our passed experience on the design and development of the Corese semantic engine. Corese is dedicated to querying Semantic Web data (in RDF, RDFS, part of OWL Lite and RDF Graph Rules) with SPARQL and its internal data structures and operations rely on the Conceptual Graph model [4,5].

KGRAM is an acronym for Knowledge Graph Abstract Machine. It results from an abstraction process we conducted to propose a generic solution to the problem of querying oriented labelled graphs. KGRAM addresses current major challenges related to the multiplication of coexisting knowledge representation languages. It provides unifying reasoning mechanisms for querying various knowledge graph models [6]. For an abstract graph model, we build upon preliminary results of the GRIWES project to which we participated [1].

KGRAM is designed as the interpreter of an abstract language which generalizes the SPARQL query language for RDF in its upcoming 1.1. version: including aggregate functions, subqueries, negation and property paths. It enables to query not only RDF but also any knowledge graph model, and even more any model — provided that it is capable of producing a graph view of its data (e.g. RDFa without intermediate transformation into RDF, XML, relational databases, etc). To achieve this genericity, KGRAM is designed with a strict distinction between the interpreter of the query language and the data graph manager. KGRAM’s application programming interface reflects both the abstract primitives of the query language and the abstract graph data model. KGRAM’s interpreter is connected to one or several graph managers implementing KGRAM’s interface. Its evaluation function only ma-
nipulates the abstract graph structures of KGRAM’s interface.

To sum up, the genericity achieved in the design of KGRAM is twofold. It is first relative to KGRAM’s query language. It is indeed conceived as an extensible set of primitives defining in fact a family of languages. KGRAM is thus able to interpret any query language built upon these primitives — provided that a compiler is written for the chosen language. This task of writing a compiler is facilitated by the genericity in KGRAM whose data structures are interfaces.

The genericity of KGRAM is also relative to the different knowledge graph models it is able to query. Its function for evaluating a query over a graph base only manipulates interfaces — of graph nodes and edges — and calls for functions of interfaces — of an abstract graph manager, an abstract comparator of node labels and of an abstract constraint evaluator. KGRAM thus enables to query graphs of various models — provided the implementation of these interfaces.

This genericity of KGRAM makes it interoperable in the sense that it enables it to exploit graphs coming from different models by connecting different graph managers and constraint evaluators implementing the same interfaces. In the simplest case, KGRAM enables to match oriented labelled graphs — by supplying a basic implementation of a comparator of node and edge labels. As further described later in this paper, we developed two other implementations of KGRAM interfaces which take into account the semantics of graphs; one matches conceptual graphs with constraints and the other queries RDF graph with an extension of SPARQL.

KGRAM’s genericity and interoperability open the path to distribute the storage of knowledge and the treatment of queries over different knowledge graph bases that may be heterogeneous. This can be understood according to several viewpoints. KGRAM can first be viewed as a mean to handle distributed knowledge bases and to unify querying over graph-based knowledge representations or more generally over graph-based data in various models. This is a major issue for the web of data where several models among which RDF/S, XML or DB data coexist. Moreover combining the results of partial results on different bases enables the development of mashup applications. Finally, the call of several graph managers in separate parallel threads enables to tackle the problem of scaling in web querying.

When downloading KGRAM, a default implementation comes with KGRAM’s interface which enables to process a query over several knowledge bases with a default meta graph manager that transparently polls the target bases. The KGRAM interpreter can therefore perform a basic mashup of data coming from several stores. However, the polling of heterogenous knowledge models is not yet handled by its meta graph manager in this default implementation — the target bases must share the same chosen knowledge model. In addition, KGRAM’s language and interpreter provide a way to mashup XML data or relational data while querying knowledge graph bases, through some features of its query language which extend SPARQL.

We first present in section 1 KGRAM’s abstract query language and show how it generalizes and extends SPARQL. Then we describe the interpreter of this language and we show how it enables to query any knowledge graph model. Section 3 is dedicated to the presentation of KGRAM’s application programming interface and the highlightment of the corollary genericity of the KGRAM interpreter. We show in section 4 how to use KGRAM to build web applications to the querying and mashup of distributed and heterogenous data. We summarize in section 5 KGRAM’s capabilities and discuss its limitations.

### 2. A Generic Query Language and its Interpreter

In this section we first describe KGRAM’s generic query language and we identify the outstanding subsets of expressions corresponding to knowledge graph homomorphisms, SPARQL and SPARQL 1.1. Then we describe KGRAM’s interpreter: the operational semantics it implements and its evaluation function.

#### 2.1. A Generic SPARQL-like Language

**2.1.1. Abstract Syntax**

The abstract syntax of KGRAM’s query language is given by the following grammar:

\[
\begin{align*}
\text{QUERY} &::= \text{query(\text{NODE} \,*\,, \, \text{EXP})} \\
\text{EXP} &::= \text{QUERY} \\
& \quad | \, \text{NODE} \ | \ \text{EDGE} \ | \ \text{PATH} \ | \ \text{FILTER} \\
& \quad | \, \text{and}(\text{EXP}, \, \text{EXP}) \\
& \quad | \, \text{union}(\text{EXP}, \, \text{EXP}) \\
& \quad | \, \text{option}(\text{EXP}) \\
& \quad | \, \text{not}(\text{EXP}) \ | \ \text{exist}(\text{EXP}) \\
& \quad | \, \text{graph}(\text{NODE}, \, \text{EXP}) \\
\text{NODE} &::= \text{node}(\text{label}) \\
\text{EDGE} &::= \text{edge}(\text{label}, \, \text{NODE} \, \,*\,) \\
\text{PATH} &::= \text{path}(\text{RegExp}, \, \text{NODE}, \, \text{NODE}) \\
\text{FILTER} &::= \text{filter}(\text{FilterExp})
\end{align*}
\]


2.1.2. Overview of the Language Constructs

A query is defined by an expression to be evaluated and a list of variables for which the list of values is searched when the expression is evaluated on the graph which is queried. An expression QUERY enables to express such a query. Here is a simple example of an expression asking to query for authors and titles of documents. Let us note that it does not depend on the model of the data that will be queried.

query(
  {node('?x'), node('?title')},
  and(edge('hasCreated',
    {node('?x'), node('?doc')}),
    edge('hasTitle',
      {node('?doc'), node('?title')})))

Its parameter EXP represents the expression to evaluate — in this example it is an and () expression — and its parameters NODE the variables for which the bindings are searched — in this example they are ?x and ?title. These variables correspond in a concrete syntax to those of a SELECT clause in an SPARQL-like language or to the parameters of a lambda-expression in the Conceptual Graph model.

A QUERY expression also enables to formulate a query nested into another. In that case the result of its evaluation determines bindings for the rest of the evaluation of the embedding query.

NODE and EDGE expressions enable to query for nodes or n-ary relations (hyperarcs) in a hypergraph. The label parameter of a NODE or EDGE expression represents the label of a node or an edge in a graph; it is a constant (or a variable for NODE).

The FilterExp parameter of a FILTER expression enables to express constraints on the searched nodes in the graph which is queried. It is a boolean expression of the following constraint language (interpreted by a filter evaluator given to KGRAM):

\[
\text{FilterExp ::= Variable | Constant | Term}
\]
\[
\text{Term ::= Oper(FilterExp * )}
\]
\[
\text{Oper ::= '<' | '<=' | '>' | '='}
\]
\[
\text{| '!=' | '&&' | '||' | '!'}
\]
\[
\text{| '+' | '-' | '/'}
\]
\[
\text{| FunctionName}
\]

Let us note that NODE, EDGE and FILTER expressions are primitive and we will show in section 3 that they correspond to interfaces of KGRAM.

A PATH expression is a generalization of an EDGE expression. It enables to query for paths of binary relations between two nodes in a graph. The RegExp parameter is a regular expression describing a set of relation paths. It is for knowledge graphs what XPath paths are for XML trees. A basic path is a binary relation. A path can also be the iteration of one relation or one relation path (represented by the * operator) any times or a sequency of relations or relation paths (represented by the / operator), with possible alternatives (represented by the | operator):

\[
\text{RegExp ::= label | RegExp ' * '}
\]
\[
| RegExp' / 'RegExp
\]
\[
| RegExp' | 'RegExp
\]

Here is an example of a query with a PATH expression enabling to retrieve all the elements in a list:

query({node('?y')},
  path(rdf:rest*/rdf:first,
    node('?x'), node('?y')))

An and (resp. union) expression enables to express a conjunction (resp. disjunction) between two expressions. An option expression makes optional the existence of solutions to some expression in the search of solutions to a query. A not expression expresses negation as failure. An exist expression enables to search for only one solution (the first retrieved). A graph expression enables to specify the knowledge graph upon which the query is evaluated (without such an expression it is a default graph which is considered).

2.1.3. From Graph Homomorphism to SPARQL 1.1.

Depending on the subset of query expressions that we consider, we define a particular (sub) language for KGRAM. This is summarized on figure 1. Worth noticing, the NODE and EDGE expressions define a query language corresponding to the one of the Simple Conceptual Graph model [9,3]. By including FILTER expressions, we consider conceptual graphs with constraints [2].

The expressions NODE, EDGE, FILTER, and(), union(), option() and graph() define a sublanguage which corresponds to the core of SPARQL SELECT-WHERE query pattern extended to n-ary relations. In addition, the exist() expression corresponds to the ASK query pattern of SPARQL.

The notion of nested query captured in the query() expression and that of relation path captured in the
path) expression are under review by SPARQL 1.1 WG. SPARQL 1.1. project expressions (and aggregates) can also be expressed in KGRAM’s language through pairs of a node and a filter but this is not detailed in this paper.

2.2. KGRAM Generic Interpreter

2.2.1. Natural Semantics

Natural Semantics [7] is a semantics specification formalism originally used for programming languages where axioms and inference rules characterize each language construct. An inference rule is applied within an environment and produces one or several new environments. In the case of KGRAM, an environment represents a set of bindings of query variables with values. The rules we established for KGRAM’s query language describe the evolution of the environment (initially empty) during the evaluation of an expression building up a query.

The following rule 1 governs the way to evaluate an expression for searching an EDGE in a graph. It specifies that the evaluation of such an expression in an environment ENV requires to compute the list of environments LENV capturing the possible matching of EDGE in the graph which is queried and to merge ENV and LENV. These two operations are synthesized in the rule bases match and merge which specify the semantics of the comparator of edge labels and the environment manager of KGRAM (see section 2.2.2).

\[
\begin{align*}
\text{match}(ENV \vdash \text{EDGE} \rightarrow LENV) \land \\
\text{merge}(ENV, LENV \rightarrow LENV')
\end{align*}
\]

A similar rule governs the way to evaluate an expression for searching a NODE in a graph.

Rules 2 and 3 define the way to evaluate a FILTER expression. The rule base eval is relative to the evaluation of the boolean expression by which a FILTER expression is parameterized; it exploits the bindings of the query variables embedded in the current environment ENV. Rule 2 specifies that if this boolean expression is evaluated to false then an empty environment list (nil) is produced: there is no solution. Rule 3 specifies that otherwise the list produced contains a single element which is the current environment (this list is created with the list operator).

\[
\begin{align*}
eval(ENV \vdash F : false) & \quad \rightarrow nil \\
eval(ENV \vdash F : true) & \quad \rightarrow \text{list } ENV
\end{align*}
\]

We will see in the next section that the rules associated to these three expressions of the query language — NODE, EDGE and FILTER — are the keystone of the algorithm of KGRAM interpreter. In the same way, we have defined similar rules for each other expression of KGRAM’s query language that we do not present in this paper.

2.2.2. Evaluation Function

The core of KGRAM is its evaluation function which interprets KGRAM’s abstract query language. Its algorithm implements the rules of Natural Semantics specifying the semantics of the language. It specially relies on those associated to the expressions NODE and EDGE. The operationalisation of these rules corresponds to the search of homomorphisms on labelled graphs whose relations may be n-ary. More precisely, the environments produced by these rules represent the (partial) homomorphisms found between the expression of the query language and the data graph. In addition, the operationalization of the rules associated to expression FILTER corresponds to the search of homomorphisms with constraints.

**Principles of the Algorithm** KGRAM’s algorithm is given on the next page. The queryStack argument of the eval function represents the stack of expressions participating to the query that is evaluated. Its argument i represents the current position in this stack. The function is initially called with the whole query in the stack and a value of zero for i. An instance of KGRAM is created with (1) a producer responsi-
ble for the production of candidate nodes and edges of the data graph matching those of the query graph, (2) a matcher responsible for the matching of query and target nodes or edges, (3) an evaluator responsible for the evaluation of constraints (filters), (4) an environment manager env responsible for the storage in a stack structure of the current environment, i.e. a partial homomorphism described as node bindings and (5) a list of complete homomorphisms (representing the results of the evaluated query expression). We will see in section 3 that the producer, the matcher and the evaluator called in this algorithm implement KGRAM’s application programming interfaces. This is what ensures the independance of the interpreter of the query language from the data models which can be queried and therefore the interoperability of KGRAM.

```java
eval(queryStack, i){
    if (queryStack.size() = i){
        store(env); return;
    }
    exp = queryStack(i);
    switch(exp){
        case EDGE:
            for (Edge r :
                producer.getEdges(exp, env)){
                if matcher.match(exp, r){
                    env.push(exp, r));
                    eval(queryStack, i+1);
                env.pop(exp, r);}
            break;
        case NODE:
            for (Node n :
                producer.getNodes(exp, env)){
                if matcher.match(exp, n){
                    env.push(exp, n));
                    eval(queryStack, i+1);
                env.pop(exp, n);}
            break;
        case FILTER:
            if (evaluator.test(exp, env))
                eval(queryStack, i+1);
    }
}
```

In the switch control instruction, the blocks labelled by NODE and EDGE implement the rules associated to the expressions NODE and EDGE of the query language and hence complete the current environment with node and edge bindings between the query and target graphs. The getEdges function of the graph manager producer is called; it takes as argument an EDGE expression from the stack queryStack and the current environment env. It uses the environment to retrieve, if any, the nodes in the exp expression that are already bound. Therefore it returns the only edges compatible with the bindings in the current environment. These candidate edges are then matched again with the query edge by the matcher. This enables to tune the semantics of the matching at KGRAM’s level and therefore to handle possibly primitive producers which would exhibit inconvenient candidate edges or nodes. In the case where the matching succeeds, the queried edges are added alternately in the current environment as new bindings. The search of a homomorphism eventually succeeds and the partial homomorphism is completed when the summit of the stack is reached: env is then added into the result list by calling the function store().

Let us note that the double recursion suggested by the rules of natural semantics associated to the expressions node and edge — on both the expression to evaluate and on the environments to build — is translated by the recursivity of the eval function and the iterative recording in a result list of the complete homomorphisms built in env.

The FILTER block in the switch control instruction implements the rules 2 and 3 specifying the expression FILTER. KGRAM delegates the evaluation of filters (constraints) to an abstract filter evaluator evaluator. The test function of the latter takes as argument a filter to be evaluated and the current environment which acts as a variable binding environment. If the filter evaluates to true, the search for an homomorphism continues with the same environment. Otherwise the partial homomorphism represented by the current environment cannot be completed and a backtrack in the eval function enables to go back to a previous level in the stack of expressions queryStack, to enumerate new candidates and then evaluate the filter in other environments where it may succeed.

All the rules of natural semantics specified for KGRAM’s language have been implemented in the interpreter’s algorithm by specific blocks integrated to the backbone of the algorithm shown above: each expression has its own block. We do not detail them in this paper. Note that these are just some of these blocks of instructions that modify the stack of expressions queryStack to be evaluated.

**Optimizations** When the partial homomorphism represented by the current environment cannot be completed, a recursive call of the eval function may backtrack in the call stack and therefore restore previous states of the stack queryStack and of the corre-
sponding environment stack. The enumeration of candidate nodes or edges then continues in order to find new bindings in this state. In order to optimize the algorithm, we have refined backtrack (i.e. return to level n-1) with a mechanism that enables to backtrack directly at a lower level (e.g. n-2) in case of a local failure. This mechanism, called backjump, enables to return directly to an expression whose evaluation furnishes a new binding for at least one query node from the last expression that just failed. For this purpose, the push function of the env environment records, for each node binding, the position of the first expression in the expression stack that produces it’s first binding. The backjump function of the env environment is then able to compute the position in the expression stack where to backtrack when an expression fails: backjump occurs at the greatest bind index (highest in the stack) that may modify the binding of one of the nodes of current expression. For brevity and clarity reasons, we omitted this backjump mechanism in the code of the algorithm presented above.

KGRAM’s evaluation function is also optimized by using its ability to build homomorphisms alternatively by node search and edge search. This enables to optimize the evaluation of a query in the cases where some query nodes are known statically or some query edges have very few target candidates.

Performance KGRAM runs the SPB2 benchmark\(^2\) of 17 queries on 120,000 triples in 6.93 seconds (0.41 s per query) on a laptop. One query takes 60% of the whole time, some optimizations still need to be done.

We also run KGRAM on an internal benchmark of 524 queries and 25,000 triples in 17.33 seconds.

3. An Application Programming Interface to Develop Query and Mashup Tools

The KGRAM interpreter comes with the application programming interface (API) it uses — and default implementations of it. In this section we describe the overall architecture of KGRAM’s API, its main interfaces and functions and we explain how it makes the KGRAM interpreter generic and interoperable.

\(^2\)http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B
described below and thus ignores the way nodes or edges are matched.

A node and edge matcher implements the KGRAM Matcher interface. It is responsible for comparing node and edge labels. It implements the match semantic rule base occurring in the rules of natural semantics specifying the expressions NODE and EDGE of the query language. Depending on the Matcher implementation, the label comparison consists in testing string label equality or it may take into account class and property subsumption, or compute approximate matching based on semantic similarities, etc.

Constraints (or filters) are abstract entities that implement the Filter interface which specify the filter expression of the query language. Filters are evaluated by an object that implements the Evaluator interface. KGRAM ignores the internal structure of filters, it calls the eval function of Evaluator on Filter objects and passes the Environment as argument. This eval function implements the eval rule base occurring in the rules of natural semantics 2 and 3.

3.2. Description of KGRAM’s Main Interfaces

Here we concretely describe the main methods of the main Java interfaces of KGRAM. The getEdges method of the Producer interface has already been encountered in the description of the algorithm of the interpreter (see section 2.2.2). It iteratively returns the edges of the queried graph that match its parameter qEdge (indicating a query edge) in the current environment env. Its signature is given below. The queried graph either is a default graph or it is specified in the query expression; in that case it is indicated by the gNode parameter and/or by the from parameter (a list of graph identifiers).

```java
Iterable<Entity> getEdges(Node gNode,
List<Node> from, Edge qEdge,
Environment env);
```

The match method of the Matcher interface is responsible for testing the matching of a query edge q and a target edge e in an environment env. The interpreter asks the matcher to call this method for each edge retrieved by the producer (see section 2.2.2). Its signature is as follows:

```java
boolean match(Edge q, Edge e,
Environment env);
```

The test method in the Evaluator interface is responsible for computing the truth value of a filter f in an environment env. Its signature is as follows:

```java
boolean test(Filter f, Environment env);
```

The.getLabel method in the Edge interface is responsible for returning the label of a n-ary relation between nodes in a graph; the nbNode method is responsible for returning the number of nodes involved in this relation and the getNode method for returning the ith node argument of the relation. The signatures of these three methods are as follows:

```java
String getLabel();
int nbNode();
Node getNode(int i);
```

3.3. Genericity of the KGRAM Interpreter

3.3.1. Principles of Genericity

The design of the KGRAM interpreter relies on interfaces. It is both independent of the concrete query language and data model. The access to data is mediated by an abstract producer and an abstract matcher. Moreover the evaluation of filters is delegated to an abstract filter evaluator. It is hence independent of the nature of the filters processed — which depends on the filter language implemented by the filter evaluator.

3.3.2. Porting Experiences

We have tested KGRAM’s portability by implementing its interfaces Node, Edge, Producer, Matcher and Evaluator with both Corese\(^3\) and Jena\(^4\) [8].

KGRAM interfaces are designed in order to minimize the glue code. As a result, Corese’s and Jena’s portings to KGRAM have required a few lines of source code. Corese’s porting was almost immediate because KGRAM was partly designed as an abstraction of the principles of Corese. Jena’s porting has required less than 1000 source lines of code.

For the validation of KGRAM with one implementation or the other, we have used a RDF base comprising 25,000 triples and a base of 524 queries (see

---

\(^3\)http://www-sop.inria.fr/edelweiss/software/corese/

\(^4\)http://jena.sourceforge.net/
In Corese’s porting, KGRAM interprets its whole query language and queries RDF data implemented as conceptual graphs. In Jena’s porting, KGRAM interprets the subset of its language corresponding to SPARQL and queries RDF data.

As a conclusion, these two different implementations testify the genericity of the design of KGRAM and show that the porting of other implementations should be easy. We emphasize that the queried data model does not necessarily need to be a graph model: it is sufficient to provide a graph view on the model through the implementation of the Producer interface (relational databases may be good candidates for such a porting).

4. Querying and Mashup of Linked Data

In this section we explain how the KGRAM interpreter and the implementation of its API enable to build applications to query and mashup linked data.

4.1. Querying and Mashup of Distributed Data

The KGRAM interpreter relies on its Producer interface for the enumeration of data edges and nodes. This enables to seamlessly design a producer that enumerates edges coming from several graph stores. For this purpose, we have designed a metaproducer which implements the Producer interface and is an iterator of producers, each of which implements Producer and can enumerate edges and nodes from one graph store.

This metaproducer is used in the ISICIL project\(^5\) to answer a usage scenario where RDF data is distributed on three servers for performance issues due to the size of the knowledge bases and their heterogeneity. Each RDF server is in charge of inferences on specific types of data:

- Server 1: social network and user profiles, online communities, activity tracking and trust model,
- Server 2: tag model, document metadata, terminologies, thesaurus,
- Server 3: web resource model with low level data such as MIME type, production context, format, duration, etc.

Some of the web applications developed in the ISICIL framework require to answer queries over data distributed on these three servers. We have configured a metaproducer iterating over three RDF producers, one for each server. They all are implementations of KGRAM’s Producer interface based on Corese. The application we developed with KGRAM thus enables to mashup the data of these three RDF stores with SPARQL queries which evaluation produces an environment involving values from the three RDF stores. This is illustrated in Figure 3.

---

\(^5\)http://isicil.inria.fr/
4.2. Querying and Mashup of Heterogenous Data

The implementation of KGRAM’s Producer interface by a metaproducer is also the key to mashup data with heterogenous knowledge models. It suffices that an implementation of the Producer interface is implemented for each knowledge model, with also different implementations of the Node and Edge interface, and that a metaproducer iterates over all of them. In that case, the matcher which is called by the interpreter for each candidate node or edge returned by the metaproducer is here to harmonize the semantics of the preliminary matchings of the producers.

By default, KGRAM is provided with an implementation of its Evaluator interface which handles XML Schema datatypes. Depending on the data to mashup, the evaluator must be able to compare values coming from different implementations of a common datatype or values from different datatypes — with a cast operation. The limits of the mashup capabilities of KGRAM lie precisely in this general problem of handling heterogenous datatype values; this is further discussed in section 5.1.

However, the advantage of KGRAM’s architecture when designing a mashup application, is that the processing of target nodes, edges and values are localized in specific components. This enables to localize and treat separately the difficulties inherent to data heterogeneity.

4.3. Mashup of Knowledge Graphs with XML and Relational Data

In addition to the querying of heterogenous data distributed over several graph stores by combining dedicated graph producers, KGRAM’s language and interpreter also enable to mashup data coming from XML or relational database during the evaluation of a query expression over a data graph. This is the subject of this section.

It is known that RDF can embed XML Literal values by means of the rdf:XMLLiteral datatype. Unfortunately, SPARQL does not allow to query the content of this structured datatype. Moreover URIs of resources can denote XML documents the content of which may be interesting to query. For this purpose we introduced in KGRAM’s language an extension of SPARQL to process XML data using XPath. It consists in:

- an xpath function that enables to apply an XPath expression to an XML Literal or to an XML document at a given URI.
- an unnest function that enables to enumerate a collection of results as variable bindings in a subquery.

These two functions occur in the abstract syntax of KGRAM’s query language as parameters of a FILTER expression. More precisely, in the filter language, they are instances of FunctionName (see section 2.1.2).

Let us consider the example below. The combination of functions xpath and unnest enables to query for book titles in an XML document designated by the ?doc variable and to bind the retrieved values to the ?title variable which is used in a query expression of KGRAM’s language to retrieve both the authors and titles of documents.

```
select * where {
    ?doc c:author ?a
    {select unnest(
xpath(?doc, '/book/title/text()'))
as ?title where {}}
?doc c:title ?title
}
```

Similarly, we also introduced in KGRAM’s language a sql function that computes a SQL query on a relational database. Each row in the SQL query result is translated into a variable binding. Let us consider the example below. The combination of functions sql and unnest enable to query for both the URIs of persons designated by the ?x variable and stored in an RDF base (or any other data graph) and their names and adresses which are designated by variables ?name and ?adr and are stored in a relational database.

```
select * where {
    ?x rdf:type c:Person
    {select unnest(
        sql('select NAME ADDRESS
            from EMP where DEPT = 3.14'))
as (?name, ?adr) where {}}
?x c:name ?name
?x c:name ?adr
}
```

5. Conclusion: Assessment and Outlook

As a conclusion, we summarize the capabilities of KGRAM and discuss its limitations. Then we outline our conceptual and algorithmic outlook on KGRAM.
5.1. Capabilities and Limitations

KGRAM is an abstract machine which interprets an abstract query language to query knowledge graph models. Its query language is a generalization and extension of SPARQL which enables to (1) handle most features of the upcoming SPARQL 1.1 recommendation, (2) query not only RDF but also any knowledge graph model, (3) mashup XML or relational data. It comes with an application programming interface (and a default implementation of it based on Corese) which enables to develop semantic web applications for the querying and mashup of distributed and heterogenous data.

The limits of KGRAM lie in its capabilities to mashup heterogenous data. These are precisely relative to the datatype values of the knowledge models. In the simplest case where the different models share the same datatype (like the XML schema datatype standard) the problem is reduced to handling the different implementations of this datatype adopted in the different producers connected to KGRAM’s metaproducer. Otherwise a cast system between different datatypes must be developed and this may be quite a difficult problem.

However, KGRAM comes with a default evaluator implementation that manipulates values as Java Object. This generic evaluator is provided with a proxy that is able to evaluate basic expressions by casting Object values to target values (e.g. integers, strings, etc.). This two-stage implementation is a first part of the solution to the problem of heterogenous datatypes: KGRAM’s default evaluator interprets a default filter language without freezing the choice of the implementation of the values. This is localized in the proxy.

5.2. Ongoing work

On one hand we are currently integrating optimizations in the interpreter’s algorithm, such as those described in [5], e.g. backjumping in the stack of expressions, heuristically sorting query edges or retrieving several connected edges at once instead of enumerating them each after the other.

On another hand, we are dealing with the problem of the distribution of treatments over the web of data. We envision to make KGRAM’s metaproducer evolve by calling the producers on which it iterates in separate threads. By doing so, we envision KGRAM as a response element to the problem of scaling in querying the web of data.

Acknowledgement

We wish to thank the members of the GRIWES project for our lively discussions. These have been valuable inputs in the design of KGRAM.

We also wish to thank Nicolas Delaforge for his work on KGRAM in the ISICIL framework.

Finally, we would like to pay a tribute to our departed friend and colleague Rose Dieng-Kuntz who had the intuition of the usefulness of knowledge graphs for the research program of the former Acacia team.

References