View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by INRIA a CCSD electronic archive server

KGRAM Versatile Inference and Query Engine for the Web of Linked Data

Olivier Corby, Alban Gaignard, Catherine Faron Zucker, Johan Montagnat

To cite this version:

Olivier Corby, Alban Gaignard, Catherine Faron Zucker, Johan Montagnat. KGRAM Versatile Inference and Query Engine for the Web of Linked Data. IEEE/WIC/ACM International Conference on Web Intelligence, Dec 2012, Macao, China. pp.1-8. hal-00746772

HAL Id: hal-00746772
https://hal.archives-ouvertes.fr/hal-00746772

Submitted on 29 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

KGRAM Versatile Inference and Query Engine for the Web of Linked Data

Olivier Corby*, Alban Gaignard†, Catherine Faron Zucker†, Johan Montagnat†
*INRIA Sophia Antipolis-Méditerranée / University of Nice Sophia Antipolis / CNRS, Sophia Antipolis, France
http://wimmics.inria.fr
Université Nice Sophia Antipolis / CNRS, I3S laboratory, Sophia Antipolis, France
http://www.i3s.unice.fr

***Abstract*—Querying and linking distributed and heterogeneous databases is increasingly needed, as plentiful data resources are published over the Web. This work describes the design of a versatile query system named KGRAM that supports (i) multiple query languages among which the SPARQL 1.1 standard, (ii) federation of multiple heterogeneous and distributed data sources, and (iii) adaptability to various data manipulation use cases. KGRAM provides abstractions for both the query language and the data model, thus delivering unifying reasoning mechanisms. It is implemented as a modular software suite to ease architecting and deploying dedicated data manipulation platforms. Its design integrates optimization concerns to deliver high query performance. Both KGRAM's software versatility and performance are evaluated.**

***Keywords*-Knowledge graphs; Data Mediation; Distributed Query Processing; Modular architecture**

I. INTRODUCTION

An increasing number of data sources becomes available over the Internet, which either adopt graph-based knowledge representation (*e.g.* RDF and RDFS standards), or comply to various inherited representations, such as relational data. Pushed by the need to federate such data sources, the concept of a Web of *Linked Data* has emerged, where data entities are uniquely identified and relations between entities are semantically well defined. Nowadays, a growing number of applications expect to manipulate the Linked Data seamlessly available over the Web. Conversely, Linked Data opens new opportunities to enrich existing applications by exploiting enlarged, joined data and knowledge repositories. This momentum creates the need for new tools able to query, join, and manipulate the heterogeneous and distributed data sources composing the Web of Linked Data.

The objective of this work is the design of a versatile *Query System* that supports (i) multiple Query Languages for the Web of Linked Data (especially, but not limited to, the SPARQL 1.0 and 1.1 standards); (ii) query-based federation of multiple, possibly remote, data sources; (iii) mediation of a wide range of heterogeneous data models encountered on the Web; and (iv) adaptability to various data manipulation use cases. This paper introduces a semantic query distribution algorithm relying on a modular mediation software architecture to implement efficient dynamic querying over distributed heterogeneous data sources

To simultaneously address these objectives, this Query System abstracts both the query language used to select and manipulate data entities, and the data models of the connected data sources. A pivot *Abstract Knowledge Graph* (AKG) model is used internally to represent data entities and the relations between them (see Section II). The core Query Engine, described in Section III, is founded on an *Abstract Query Language* (AQL) that provides an abstract and extended representation of the SPARQL 1.1 language (see Section III-A). The system provides unifying reasoning mechanisms for querying various knowledge graph models [5], hence its name: *Knowledge Graph Abstract Machine* (KGRAM). A novel query distribution (Section III-B) and optimization (III-C) algorithm is introduced in KGRAM. It relies on the KGRAM modular software suite based on well-defined abstract interfaces that is described in Section IV. KGRAM software modules can easily be combined to architect and deploy dedicated platforms tackling specific data query use cases. The system design integrates optimization concerns to deliver high query performance and improve end-users experience. A reference implementation that covers the SPARQL 1.1 standard query language is exemplified and evaluated in Section V.

II. KGRAM PRINCIPLES

KGRAM is an *Abstract Machine* in the sense that its core is a generic query and inference engine abstracted from the data model and query language used. It processes *Knowledge Graphs* through a graph-based knowledge representation model (AKG) and an associated graph-based query language (AQL) designed for abstracting and manipulating data entities extracted from heterogeneous data sources [1]. Indeed, typed graph structures, which nodes and edges are labeled with types that can support inferences, are multiplying on the Web. They not only support logical reasoning but can also be seen as metric spaces to pilot approximate reasoning and querying, as indexes of knowledge in distributed environments, as models to make interfaces more intelligible to end-users, or as new frameworks for social structures analysis. The mapping of knowledge graphs (in particular conceptual hypergraphs and RDF graphs) to the AKG representation is straightforward by design. Other commonly encountered data models such as relational databases or

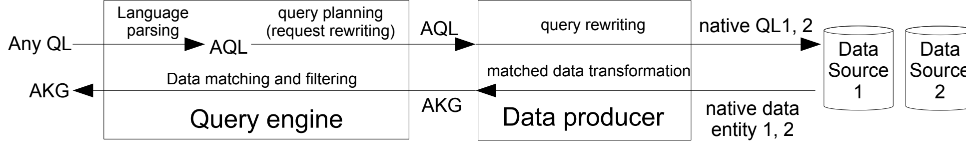

Figure 1. KGRAM principles

XML data sources can be transformed into equivalent AKG representations as well.

A. Query Process

KGRAM's query process principle is diagrammed in Figure 1. KGRAM is decomposed into two main components: a *Query Engine* (QE), capable of handling different query languages, planning queries over multiple data sources and joining data entities retrieved from multiple sources; and a *Data Producer* capable of interfacing to heterogeneous data sources. The QE internally uses AQL to decompose incoming queries from any (supported) Query Language and performs query planning through query rewriting. Input queries can thus be transformed into as many subqueries as deemed necessary to efficiently query multiple and heterogeneous data sources. AQL subqueries are transferred to a *Data Producer* that acts as an abstraction for a native data source. It rewrites inbound AQL queries into the native query language of its associated data source, and conversely transforms the native data entities retrieved (*e.g.* triples or tuples) into KGRAM's AKG homogeneous representation. The AKGs retrieved while handling a query are post-processed by KGRAM's QE, which may filter out and join the received AKGs to produce the final AKG query result.

B. Remote Data Federation

To enable multiple and remote data sources querying, KGRAM's *Producers* may need to be architected according to different concerns, as illustrated in Figure 2. Multiple sources querying (top) requires rewriting inbound AQL subqueries into different native Query Languages. The data entities retrieved are transformed into pivot AKG representations and can be joined before being delivered to the QE. Remote sources querying (bottom) requires the integration of a client, a server and a communication channel. The inbound AQL subqueries are serialized for transfer between the client and the server. On the server side, the query rewriting and results transforming actions are undertaken, before AKG resulting entities are serialized for delivery to the client.

Depending on the precise data querying infrastructure needed, both multiple- and remote-querying capabilities may be needed. KGRAM is architected as a versatile toolbox facilitating the design of a specific query infrastructure by deployment of tailored generic software components.

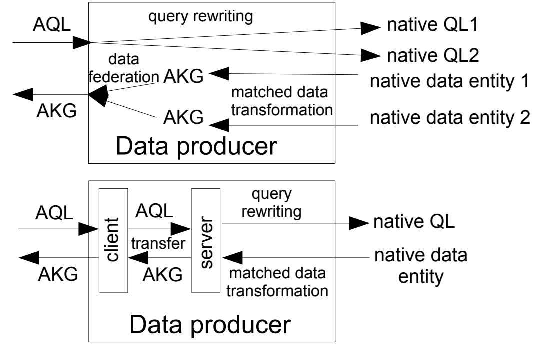

Figure 2. Data producer principles. Top: multiple data sources federation. Bottom: remote data sources access.

III. Query Engine

KGRAM query engine relies on AQL to decompose input queries into homogeneous subqueries and a planner to efficiently evaluate query parts and join results.

A. Abstract Query Language (AQL)

To query and manipulate knowledge graphs, KGRAM searches for homomorphisms in oriented labelled hypergraphs through a comparator of graph node and edge labels. AQL, which grammar and semantics are detailed in [5], is a SPARQL 1.1 extension. SELECT-WHERE clauses extended to n-ary relations are represented through labelled *Nodes*, n-ary *Edges*, *Filters* defining constraints on the searched nodes, *Paths* of binary relations between two nodes in a graph, as well as conjunction, disjunction, options, negation and existence checking of the formers. AQL models nested sub-queries, SPARQL's aggregates, and SERVICE clauses. The engine supports entailment regimes and builds new structures in the query result AKG to model CONSTRUCT, UPDATE and INSERT clauses. Note however that updates and insertions cause changes to the in-memory result graph, but they are not materialized in the remote data sources. An heterogeneous, distributed implementation of UPDATE and INSERT clauses would imply modifying the data sources through data producers able to transform new/changed structures into the native data sources representation and the associated native query language. AQL encompasses various existing query languages such as the Simple Conceptual Graph model [4] or conceptual graphs with constraints [2].

Figure 3 gives an AQL example querying authors and titles of documents linked through edges labelled with

'hasCreated' and 'hasTitle'. The result of this query evaluation is the binding of node variables '?author' and '?title' with matching values in the queried graph.

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

Figure 3. Query example.

B. Query Planning and Processing

The core of KGRAM is its evaluation function. The operationalization of AQL involves the invocation of the connected *Data Producer* to fetch AKG-formatted candidate nodes and edges. Candidates are processed through a customizable *Matcher* implementing search of homomorphisms on labelled edges, possibly with *Filter* constraints.

1) Query evaluation algorithm: Algorithm 1 provides a simplified representation of the KGRAM query evaluator. Input queries are parsed and transformed into a tree representation which nodes contain elementary AQL expressions. The tree structure is flatten into a $queryArray$ so that the recursive eval function (lines 11 and 17) can browse the query structure depth first and iteratively process each expression. NODE and EDGE expressions in particular invoke the data producer connected to fetch the graph nodes and edges to match (lines 8 and 14). The candidate AKG nodes and edges returned by the data producer are matched against the query expression by the *matcher*. The evaluation of filters (constraints) is delegated to an abstract *filter.test* function. All the other AQL expressions have been implemented by specific blocks integrated to the backbone of this algorithm.

eval is initially called with an index i pointing to the first query expression ($i \leftarrow 0$) and an empty environment env. All along the query evaluation, KGRAM stores node bindings and keeps track of all (partial) homomorphisms found between the AQL expressions and the data graph in env. The eval function processes each expression in the context of the known environment, which is progressively enriched so that partial matches found are taken into account when evaluating subsequent elementary expressions in the expressions tree. At the end of the evaluation, the homomorphisms are complete and returned as a list of bindings from the environment (line 3). Consider the query given in Figure 3 as an example. The query engine would first search for relations between authors and documents they wrote, returning a number of bindings for the pair of nodes author-document in the environment. It would then query for relations between the only documents found in the previous step and their title. In case a partial homomorphism represented by the current environment cannot be completed, a backtrack in the evaluation function invocation stack reverses to a previous level in the expressions tree to enumerate new candidates and filter them in an updated environment.

Algorithm 1: Simplified KGRAM query algorithm.

Variables: $producer$ data producer interfaced to KGRAM engine
$matcher$ graph homomorphisms matcher
$filter$ filter evaluator used
Arguments: $queryArray$ array of all query elements to be evaluated iteratively,
i index of the query element currently evaluated,
env environment
Result: list of bindings found

```
1  function eval(queryArray, i, env)
2    if (i = queryArray.size()) then
3      store(env) ;        // accumulate bindings found
4      return;
5    exp ← queryArray[i];
6    switch (exp) do
7      case EDGE
8        for (Edge e : producer.getEdges(exp, env)) do
9          if (matcher.match(exp, e)) then
10           env.bind(e) ;
11           eval(queryArray, i + 1, env) ;
12           env.unbind(e) ;
13     case NODE
14       for (Node n : producer.getNodes(exp, env)) do
15         if (matcher.match(exp, n)) then
16           env.bind(n) ;
17           eval(queryArray, i + 1, env) ;
18           env.unbind(n) ;
19     case FILTER
20       if (filter.test(exp, env)) then
21         eval(queryArray, i + 1, env) ;
22     ...
```

2) Querying multiple data sources: The KGRAM query evaluation algorithm makes it easy to query multiple data sources through multiple data producers. A meta-producer, interfaced to KGRAM's QE on the one side and to multiple data producers on the other side, just forwards the graph node and edge queries of Algorihm 1 (lines 8 and 14) to all attached producers before merging all resulting bindings received into the QE's environment.

3) Remoting Producers: The implementation of the data producer and its capability to return only the necessary AKG components to the QE is critical regarding the performance of KGRAM, especially in a distributed environment were data sources are remote and the AKG components are communicated over the network. In a naive implementation, a remote producer can passively deliver all AKG nodes and edges corresponding to its entire data base for matching and filtering by the QE. However, this strategy is fairly inefficient in case a large number of the AKG components sent to the QE are later on discarded by the matcher or the filter evaluator. Consequently, the interface to the producer also includes parameters to transfer the known bindings and

the AQL filters to it. This allows for the implementation of advanced operations such as source data filtering and partial matching inside data producers.

C. Query optimizations

KGRAM implements various strategies to minimize the query processing time. Some performance optimizations are structural, related to the AQL design and the query algorithm parallelization. Some are related to the QE planning capability. Others are related to the work locally performed by the Data Producers. KGRAM's QE internally implements optimizations and its interface was designed to allow for advanced and asynchronous Data Producer implementations.

1) Nodes search: The ability of KGRAM to evaluate homomorphisms both by node search and edge search allows for optimizing the process of query languages which make use of the NODE expression.

2) Exploiting partial query results: Partial query results may be exploited to improve the processing of subqueries that have not yet been performed. In particular, KGRAM implements *bind joins*. The QE manages an index of already known mappings between variables and values in its environment. These already known bindings are transferred to the data producer at query runtime for exploitation when querying the native data source. As a consequence, the order in which subqueries are evaluated may have a significant impact on performance. KGRAM favors evaluation of selective queries which return few candidates first to reduce the amount of data that will be transferred and joined at the level of the QE. The heuristics consists in considering first, edges with fewer candidates through a pre-analysis of the graph edges cardinality. Edges are also reordered in such a way that, whenever possible, two consecutive edges share at least a node.

3) Backtracking: When a partial homomorphism cannot be completed, the recursive evaluation function call stack is backtracked, restoring former states of the corresponding environment. A backtrack optimization called *backjump* was implemented, enabling several backtrack levels in the query stack and the associated environment simultaneously. A backjump returns directly to an expression whose evaluation provides a new binding for at least one query node from the last expression that just failed.

4) Asynchronous querying: Concurrent querying of multiple data sources is an obvious optimization when querying multiple data sources, especially if these are remotely located and each source query is processed by different computing units. A parallel meta-producer that implements a meta-producer connected to data sources queried simultaneously rather than sequentially was added to the KGRAM software suite.

Algorithm 2 shows the simplified distributed query processor. The algorithm consists in iterating over each NODE or EDGE expression in the initial AQL query (line 1) and,

Algorithm 2: Simplified parallel distributed query processing.

Data: $Producers$ set of data source endpoints,
Exp set of **node** and **edge** expressions from the AQL query,
$scheduler$ a thread pool allowing for parallel execution.
Result: $Results$ the set of AQL query results.

```
1 foreach (e ∈ Exp) do
2     foreach (p ∈ Producers) do in parallel
3         scheduler.submit(p.matchingExp(e)) ;
4     wait for scheduler ;
5     foreach (task ∈ scheduler.getFinished()) do
6         Results ← task.getResults() ;
```

for each one, in concurrently querying all remote *Producers* (line 3). The meta-producer then waits for all connected endpoints to finish through a synchronization barrier (line 4). Results are finally accumulated for the current NODE or EDGE expression (lines 5 and 6) and the iteration over the next expression can be processed (line 1). To soften the synchronization barrier a pipelining approach was adopted, in which a synchronized blocking queue allows the meta-producer to post-process results as soon as they become available. Due to space limitation, the pipelining algorithm is not detailed.

5) Data sources selection: KGRAM is using a cache index dynamically created with SPARQL ASK queries, which prevents from unnecessary communications. For each AQL edge expression, this index stores the identified data sources providing candidate AKG edges and thus potentially contributing to the result set. By using its index, KGRAM is able to send remote edge requests only to data sources hosting candidates. This strategy is specially adapted for Linked Data querying scenarios where each remote data source is specialized into one kind of data.

IV. Reference Implementation

KGRAM principles have been operationalized in several java implementations. As a proof of concept, a first implementation was first provided with the existing Corese[1] search engine, based on the Conceptual Graphs model. A Jena-based[2] [8] implementation has also been experimented by creating lightweight KGRAM API adaptors for Jena objects. The last version of KGRAM is a reference implementation available from the Web[3], based on an optimized specific implementation of the AKG model. The KGRAM software suite is architected as a modular and versatile toolbox aimed at covering many use cases such as heterogeneous data sources manipulation, access to remote data stores, data source querying and inference processing (RDFS entailments and rule-based inferences) on data sources. KGRAM's

[1]Corese: http://wimmics.inria.fr/corese
[2]Jena: http://jena.sourceforge.net
[3]KGRAM: http://wimmics.inria.fr/corese

modules deployment can easily be tailored to address different data querying and federation scenarios. The reference implementation particularly focuses on SPARQL 1.1, which is the *de facto* query language for knowledge graphs models like RDF.

A. Modular Software Architecture

The KGRAM software architecture is modular with clearly defined, decoupled interfaces and extensible components. KGRAM's QE and *Data Producers* (see Section II) are implemented as decoupled modules communicating through a *Producer* interface to exchange AQL queries and AKG results. The QE adapts to different input Query Languages through a query language *Parser* interface. The AQL language can be finely customized through the *Matcher* and *Filter* interfaces. A *Matcher* computes edge homomorphisms. Depending on its implementation, labels comparison may *e.g.* consist in simple string equality, or take into account class and property subsumption, compute approximate matches, etc. A *Filter* evaluates query constraints by implementing the filtering dialect of the target language.

Figure 4 diagrams a KGRAM QE. The KGRAM API defines abstract interfaces for the different KGRAM modules to interface with each others. The engine has 4 required interfaces and it is completely defined only when the 4 corresponding modules are connected to these interfaces: a data source interfaced through the *Producer* API, a query parser interfaced through the *Query* API, a graph matcher interfaced through the *Matcher* API and a constraint evaluator interfaced through the *Filter* API. Internally, the KGRAM query engine manipulates a *Query Stack* used to decompose complex queries into elementary AQL expressions, and an *Environment Manager* used to accumulate partial AKG results as detailed in section III-B.

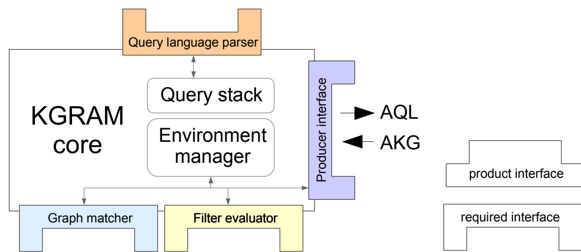

Figure 4. KGRAM core query engine

Modules implementing the *Producer* interface can be created to match the different target data sources, connect to multiple data sources or introduce network communications between data sources and the query server as illustrated in Figure 5. The top row of Figure 5 (5a) illustrates a local data *Producer* adapted to a specific data model (*e.g.* relational, RDF or XML data). It implements the *Producer API* and internally connects to the target data source through its native API (*e.g.* an SQL client interface).

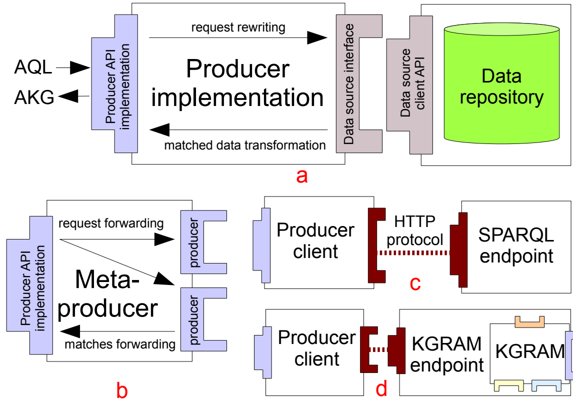

Figure 5. KGRAM *Producers*. Top: regular data source producer. Bottom left: meta-producer. Bottom right: producer towards remote data sources.

B. KGRAM Software Suite

The core QE included in the KGRAM software suite comes with a *Parser* implementing the SPARQL 1.1 query language, a *Matcher* exploiting RDFS entailments, and a *Filter* implementing SPARQL operators and XML Schema datatypes.

Several base *Producer* implementations are included in the software suite (Figure 5a): for RDF datasets in raw RDF/XML files, N3 text files or Jena databases. The bottom row of Figure 5 also shows three other specific *Producer* implementations available. The Meta-Producer (5b) is a module enabling the connection to multiple data sources. It exposes a *Producer* interface and requires multiple other *Producer* components to be connected through the same interface. It simply forwards input AQL queries to the connected *Producers* and conversely merges all AKG results delivered by all subsequent *Producers* before delivering them to the QE. The Producer client (5c) is an interface to any SPARQL remote endpoint. It transforms inbound AQL queries into SPARQL queries that are sent over HTTP to any SPARQL-compliant endpoint. The entities returned by the endpoint are then transformed into AKG results. Since AQL extends SPARQL 1.1, KGRAM can connect to any SPARQL endpoint. However, some AQL queries may not be fully interpreted by a regular SPARQL endpoint as they contain non-SPARQL expressions. KGRAM software suite therefore contains a KGRAM-enabled endpoint as illustrated in Figure (5d). The producer client communicates queries to the KGRAM endpoint using AQL rather than SPARQL. The endpoint uses a local KGRAM instance to parse and evaluate the inbound AQL queries.

KGRAM interfaces are designed in order to minimize the glue code necessary to interface to external software components. As a result, Corese's and Jena's portings to KGRAM required very few source lines of code. Corese's porting was almost immediate because KGRAM was partly designed as an abstraction of Corese principles. Jena's

porting has required less than 1000 lines of source code.

C. KGRAM Extensibility Examples

Many different deployment scenarios can be implemented taking advantage of the KGRAM component-based architecture. Both the query language and the knowledge representation language can be customized to different needs through the implementation of various *Parsers*, data *Filters* and graph *Matchers*. Furthermore, any kind of data source can be queried simultaneously, given that a *Producer* interface can be developed for it. The meta-producer combined with the scalable query algorithm allow for the integration of multiple data sources. To illustrate the versatility of KGRAM architecture, two complex deployments inspired by real data federation platforms are presented below.

Figure 6 illustrates an implementation that was experimented in the context of the ISICIL project[4]. In this scenario, RDF data is distributed over three servers, each of them in charge of inferences on a specific type of data: (1) social network and user profiles, online communities, activity tracking and trust model; (2) tag model, document metadata, terminologies, thesaurus; (3) Web resource model with low level data such as MIME type, production context, format, duration, etc. Some of the Web applications developed in this context require to query data distributed over these three servers. A KGRAM core component is therefore instantiated with a SPARQL *Parser*, a *Matcher* exploiting RDFS entailments, and a *Filter* supporting XSD datatypes. A Meta-Producer component is connected to this engine to interface to the three data sources, each of them interfaced through an instance of RDF *Producer*.

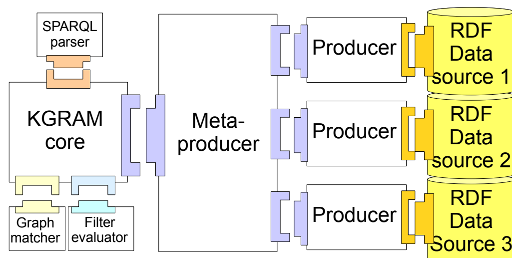

Figure 6. ISICIL data query architecture federating 3 RDF stores.

Another complex deployment, inspired by the NeuroLOG collaborative platform for neurosciences[5], is illustrated in Figure 7. The use case is the joint querying of two heterogeneous data sources (an SQL and an RDF repository) located at different places, using the SPARQL query language. Similarly to the previous case, a SPARQL-enabled KGRAM query engine interfaced to a meta-producer is deployed. To access remote data sources, KGRAM endpoints and their associated *Producer* clients are used. Each endpoint is connected to a specific *Producer* (SQL or RDF data producer) adapting to the site data source.

[4]ISICIL: http://isicil.inria.fr

[5]NeuroLOG: http://neurolog.polytech.unice.fr

V. Evaluation

There are several facets of the KGRAM query system to consider for evaluation. First, the strict compliance of the reference implementation to standards is evaluated with the W3C SPARQL 1.1 Query Test case[6] (with 4 failures over 465 queries). Second, AQL's and AKG's high expressiveness is illustrated (V-A). Third, KGRAM's performance is evaluated by comparison to a high-performance relational databases federation product (V-B). Finally, KGRAM's ability to implement complex Linked Data scenarios is demonstrated (V-C).

A. Expressiveness

The seminal NeuroLOG platform (see Section IV-C) was federating relational databases from 5 neuroscience centers in France. Its data federation layer was based on the high-performance *DataFederator* commercial tool[7]. The platform was re-deployed using KGRAM for data federation. KGRAM can access NeuroLOG relational databases either directly through an ad-hoc SQL *Producer*, or indirectly through its RDF *Producer* after conversion of the relational database into an RDF repository with the METAMorphoses mapping tool[8]. For this experiment, the SQL Producer is a simple component implementing query-specific mappings allowing to retrieve SQL entities matching the query and transform them into AKG entities.

Query Q_1 from Figure 8 is a real use-case that searches for image datasets (acquired with Gadolinium contrast agent) associated to patients (join in line 4) in the context of multi-centric studies addressing the Multiple Sclerosis pathology. During the evaluation of this query, all types of medical images are searched in order to match a "GADO" tag associated to the use of the Gadolinium contrast agent. The evaluation of this query can be considerably enhanced by exploiting clinical knowledge represented in an RDFS vocabulary. Indeed Gadolinium is used in the context of magnetic resonance (MR) acquisitions (T1 or T2 weighted MRIs for instance) but generally not in the context of any other modality (Ultrasound for instance). By exploiting this domain knowledge, the query time can be significantly reduced since all non-MR datasets (such as Ultrasound datasets) are excluded. In other words, when provided with an implementation of its *Matcher* interface supporting RDFS entailments, KGRAM performs inferences which enables it to produce better query results.

When provided with a SPARQL 1.1 *Parser*, KGRAM provides a mean to handle the intelligent distribution of

[6]W3C tests: http://www.w3.org/2009/sparql/docs/tests

[7]SAP DataFederator: http://www.sap.com/solutions/sapbusinessobjects/large/eim/datafederator/index.epx

[8]METAmorphoses: http://metamorphoses.sourceforge.net

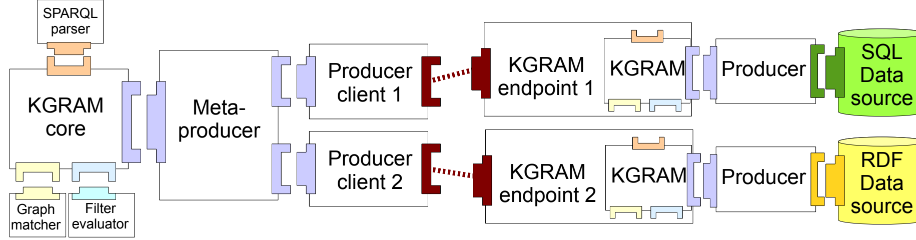

Figure 7. NeuroLOG data sharing architecture deployment example.

```
1 SELECT distinct ?patient ?study ?dataset ?dsName WHERE {
2         ?dataset linguistic−expression:has−for−name ?dsName .
3         ?patient examination−subject:has−for−subject−identifier ?clinID .
4         ?patient iec:is−referred−to−by ?dataset .
5         ?study study:involves−as−patient ?patient .
6         FILTER (regex(?clinID, 'MS') && regex(?dsName, 'GADO')) }
```

Figure 8. Sample SPARQL query Q_1: search for patient, study and dataset information in the context of the Multiple Sclerosis disease.

highly expressive and graph-based semantic queries. For instance, let us consider again query Q_1. Line 2 and 4 of Figure 8 could be merged with the following PATH expression:

?patient iec:is-referred-to-by / linguistic-exp:has-for-name ?dsName.

KGRAM thus allows to perform this kind of graph-based information retrieval which cannot be easily expressed in SQL.

B. Performance Evaluation

The query processing time using the seminal NeuroLOG relational databases federator was experimentally compared against KGRAM's heterogeneous platform deployment (federating both RDF-converted and original SQL data sources). Two queries are considered below: Q_1, introduced in Section V-A, and Q_2, a query for datasets acquired through the T2-weighted MRI modality. It is a very selective query, leading to 5 results only, on the data sources considered. To be robust against variability observed in real distributed systems, results are averaged over three query runs. The average query execution time $\pm$ one standard deviation is displayed in the following table. For Q_1, leading to 336 remote invocations, the query times are better with the optimized SQL federation engine than with KGRAM, but it remains in the same order of magnitude. For the very selective query Q_2, we observe comparable query times for both environments.

Query	KGRAM	relational federator
Q_1	11.76 s $\pm$ 0.05	3.03 s $\pm$ 0.25
Q_2	1.53 s $\pm$ 0.14	1.52 s $\pm$ 0.62

C. Linked Data Scenario

NeuroLex [7], supported by The Neuroscience Information Framework and the International Neuroinformatics Coordinating Facility, is a dynamic Neuroscience Lexicon which describes 18,490 neuroscience terms (287 neurons and 952 brain parts, 151 spinal cord parts and 30 other parts of the nervous system). It aims at delivering a standard lexicon for neuroscience entities, covering their meaning and their classification, to address data integration issues generally faced in the neuroscience area.

In this Linked Data experiment, neuroimaging data from the NeuroLOG federation is linked with neuroscience "open" knowledge capitalized through the NeuroLex initiative, allowing neuroscientists involved in the NeuroLOG federation to benefit from the NeuroLex lexicon and its semantic wiki interface. Thanks to KGRAM versatility, the NeuroLOG platform is easily extended with a new data source that exposes the NeuroLex ontology. To bridge the two knowledge bases, an ad-hoc semantic alignment was implemented, in which all NeuroLOG datasets are annotated with the NeuroLex *Label* property corresponding to their medical image modality.

```
1 SELECT ?patient ?dataset WHERE {
2         ?t property:Label \"MRI protocol\"^^xsd:string .
3         ?s rdfs:subClassOf* ?t .
4         ?s property:Label ?label .
5         ?dataset property:Label ?label .
6         ?patient iec:is−referred−to−by ?dataset .}
```

Figure 9. Sample SPARQL query exploiting the NeuroLex taxonomy for medical image modalities to search images for the NeuroLOG platform

Figure 9 illustrates a SPARQL query which is evaluated by KGRAM by first listing all NeuroLEX subclasses of *MRI protocol* and their associated *labels*, and then exploiting these labels to search for relevant medical images, and their corresponding patients provided by the distributed NeuroLOG data providers. There exist many neuroscience ontologies with different modeling objectives. Some may focus on particular pathologies while others may focus on neuro-data processing. In that context, and beyond the neuroscience area, the KGRAM query system provides a transparent mean to query distributed and heterogeneous data sources while potentially benefiting from several ontologies. This is possible by implementing the *Producer* associated to each data source and its ontology with a *Matcher* exploiting RDFS entailment.

VI. Related Works

The recent W3C working draft [9] addresses the federated querying of distributed knowledge bases through a set of language extensions for SPARQL 1.1 (SERVICE and VALUES clauses). These clauses help implementing distributed query processing when the content of data sources is well partitioned and known at query design time. However, this assumption does not hold in many real use cases. This approach is not suitable in the context of dynamic knowledge base federations in which pre-designed SPARQL 1.1 queries must be adapted to take into account the data source availability.

Transparent federating approaches generally address performance issues. For instance, DARQ [10], SPLENDID [6] or FedX [11] propose a set of static and dynamic optimizations. KGRAM implements similar strategies to exploit value constraints at query rewriting time and to enhance the querying at runtime through bind joins. Moreover, in line with FEDX, KGRAM also exploits the parallelism of both distributed data sources and modern multi-core CPUs with a multithreaded implementation. However, there remain several optimization opportunities for KGRAM. Query planning techniques implementing data sources selection could be applied. Block bind joins introduced by FEDX also seem to be an interesting optimization.

Addressing the SPARQL querying of traditional relational databases through a declarative language, Bizer et al. highlight in [3] the user needs for both weird mappings, and real data federation (not directly addressed through D2RQ). In line with these observations, KGRAM could benefit from this approach by letting legacy database experts populate remote producers with relational to semantic RDF mappings.

To the best of our knowledge, none of the state-of-the art transparent federating approaches address both the issues of efficient and heterogeneous distributed querying. The strength of the KGRAM model is to provide a versatile and extensible framework allowing to efficiently and dynamically access to multiple data sources internally using different data representations.

VII. Conclusion

The KGRAM software suite is architected following a modular approach to ease the integration of multiple, heterogeneous and distributed data sources. Its modules can easily be combined to architect and deploy dedicated platforms tackling specific data query use cases while delivering high query performance. KGRAM's versatility makes it a strong candidate to tackle the challenges raised by the Web of Linked Data.

KGRAM is an abstract machine for querying knowledge graph models. Its Abstract Query Language covers, but is not limited to, the SPARQL 1.1 standard graph-based query language promoted by the W3C. It also enables querying and reasoning through other forms of graph pattern matching such as Conceptual Graphs Homomorphism. Through AQL, KGRAM can easily adapt to future language evolutions and provides an extensible framework in which new language constructs can be experimented.

Although most SPARQL clauses, including SELECT-WHERE and CONSTRUCT, are supported while querying distributed heterogeneous data sources, the UPDATE clause is more challenging and its implementation covers local RDF sources only. This opens new data coherency challenges to tackle when considering knowledge graph insertions, deletions, or modifications in a distributed setup.

Acknowledgment

This work is partly funded by the French National Agency for Research under grant ANR-09-COSI-03 "VIP".

References

[1] J.F. Baget, O. Corby, R. Dieng-Kuntz, C. Faron-Zucker, F.Gandon, A. Giboin, *et al*. GRIWES: Generic Model and Preliminary Specifications for a Graph-Based Knowledge Representation Toolkit. *Intl Conf. on Conceptual Structures*, ICCS'08, LNCS vol. 5113, pages 297–310. Springer, 2008.

[2] J.F. Baget and M.L. Mugnier. Extensions of Simple Conceptual Graphs: the Complexity of Rules and Constraints. *J. Artif. Intell. Res. (JAIR)*, 16:425–465, 2002.

[3] C. Bizer and R. Cyganiak. D2rq - lessons learned. *W3C Workshop on RDF Access to Relational Databases*, Oct. 2007.

[4] M. Chein and M.L. Mugnier. *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Springer London Ltd, 2009.

[5] O. Corby and C. Faron-Zucker. The KGRAM Abstract Machine for Knowledge Graph Querying. In *Intl Conf. on Web Intelligence*, WI'10, 2010.

[6] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Intl Workshop on Consuming Linked Data*, Bonn, Germany, 2011.

[7] F. T. Imam, S. D. Larson, J. S. Grethe, A. Gupta, A. Bandrowski, and M. E. Martone. NIFSTD and NeuroLex: Comprehensive Neuroscience Ontology Development Based on Multiple Biomedical Ontologies and Community Involvement. In *ICBO*, 2011.

[8] B. McBride. Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[9] E. Prud'hommeaux, C. Buil Aranda, A. Seaborne, A. Polleres, L. Feigenbaum, and G. T. Williams. SPARQL 1.1 Federated Query [http://www.w3.org/TR/sparql11-federated-query], November 2011.

[10] B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. *The Semantic Web Research and Applications*, 5021:524–538, 2008.

[11] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: optimization techniques for federated query processing on linked data. *Intl Semantic Web Conf.*, ISWC'11, pages 601–616, Berlin, Heidelberg, 2011. Springer-Verlag.