

QEF-LD: A Query Engine for Distributed Query Processing on Linked Data

Keywords: Linked Data, Federated Queries, Query Processing, Data Integration, Mashup

Abstract: The Web of Linked Data forms a single, globally distributed Web. Querying this dataspace poses new challenges that are not addressed by traditional research on federated query processing. Linked data integration applications must meet the requirements of efficient query processing in the distributed setting. This paper presents QEF-LD, a query engine that enables the efficient execution of federated query over multiple Linked Data sources. Experiments demonstrate the feasibility of QEF-LD when compared to available Federated Query Engines.

1 Introduction

The Linked Data initiative promotes the publication of data as Web accessible resources. By using standard protocols and representing data using the RDF model, autonomous datasources are published and can be queried using the SPARQL query language. The diversity of published data in a standard format makes the basis for new kinds of applications that combine data from different sources into a federated view.

Linked data integration applications express federated views using the SPARQL query language. In a SPARQL federated query (Prud'hommeaux and Buil-Aranda, 2011), the service keyword points to the distributed data sources, while joins and unions integrate the data in the federation. The integrated query is submitted to a federated query engine that processes it over the distributed SPARQL endpoints.

It turns out that achieving an efficient execution of such a SPARQL federated query is hard. This is mainly due to the fact that query processors have little or no statistical information about the data on endpoints. As a result, traditional query optimization strategies are jeopardized making it hard to define optimal join orderings and reacting to large bind sets, common operations on integration queries execution.

As an example, consider the scenario in which one wants to acquire information about drugs, side effects and drug formulae, each published by an autonomous SPARQL endpoint. Running the same SPARQL federated query integrating these data sources from four different query execution engines provides response times that vary to an extreme of approximately 4.500 times (Table 1).

There is, however, a particular kind of federated

application for which fine-tuned query strategies may be conceived. Data mashups are pre-defined data views that are computed by integrating distributed data sources. In these applications, the designer knows which data sources will provide the required data and may define from experience the best strategy to access them. Thus, inter-site join orderings, for instance, can be defined in design time. Note however that, depending on the query parameters, intermediate results size may vary considerably, so a query engine must also be able to react to this variation by dynamically setting the size of bind sets in joins.

In this paper, the QEF-LD is presented. The system enables designers to specify mashup queries over federated linked data sources. During mashup design, join ordering between distributed endpoints are defined, while local joins remain specified in SPARQL subqueries to be run by the endpoints themselves. Moreover, inter-site joins are implemented by the Set-BindJoin operator.

We conducted experiments running five SPARQL federated queries on three different SPARQL query engines and QEF-LD. The results show that the proposed approach produces a query elapsed-time that is up to 4500 times faster than one of the query engines (Table 1). Moreover, QEF-LD was able to run all the queries, whereas some of the other systems suffered from memory overflow or simply would not respond. The conclusion is that the combination of pre-defined inter-site join ordering and fine tuned bind set sizes were able to produce efficient execution profiles in a context of integrated mashup linked data views.

This paper is structured as follows. Section 2 covers other works related to federated query processing on Linked Data. Section 3 presents the QEF-LD component used to execute federated query plans on

Linked Data. Section 4 explains the proposed algorithms used in QEF-LD. Section 5 analyses the experiments performed to evaluate the feasibility of QEF-LD compared to other strategies for the execution of federated queries. Finally, section 6 contains the conclusions and suggestions for future work.

2 Related Work

Jena ARQ¹ and Sesame² are query processors that implement the federated query specification for SPARQL 1.1 (Prud'hommeaux and Buil-Aranda, 2011). The specification defines the SERVICE operator that in turn defines the SPARQL Endpoint URI and SPARQL query to be executed. However the specification is quite simple and does not provide enhancements or other strategies to improve query performance.

DARQ (Quilitz and Leser, 2008) – Distributed ARQ – extends Jena ARQ in order to allow SPARQL federated queries with transparent access to multiple SPARQL endpoints. DARQ divides the SPARQL query in subqueries and submits them to the corresponding SPARQL Endpoints. Next it combines the results. One limitation of DARQ related to query processing is that it can only execute queries with bound predicates. This is because data source selection in DARQ is based on matching query pattern predicates to predicates in capability patterns. Therefore, DARQ does not allow the use of SPARQL variables in predicates of BGPs (Basic Graph Patterns). The DARQ project emerged in 2006, though its development ceased as of 2008.

SemWIQ (Langegger, 2010) is another data integration system in which queries are expressed in SPARQL. Like DARQ, it also extends the Jena ARQ query processor. SemWIQ is based on a mediator-wrapper architecture and uses its own optimization strategy to generate execution plans. SemWIQ development is no longer maintained and its last update was in 2010. DARQ and SemWIQ were not used in our experiments (Section 5) since they were discontinued and also because they did not add new operators or features specifically aimed at improving the performance of their query processor.

FedX (Schwarte et al., 2011a; Schwarte et al., 2011b) – Linked Data in a Federation – is a framework which extends Sesame with a federation layer for transparent access to data sources through a federation. It enables efficient query processing on distributed Linked Data sources. FedX is compatible

with the SPARQL 1.0 query language, which allows clients to integrate with available SPARQL endpoints. It uses join reordering, bound joins and grouping of subquery results to reduce the number of intermediate results and thus to improve federated query performance. FedX allows concurrent processing of join and union operations through the use of threads. As an ad-hoc query processor, FedX can not adapt to non-expected results from federated subqueries. QEF-LD, on the contrary, applies user knowledge about the queries as much as techniques to dynamically react to intermediate result sizes, leading to better tuned execution profiles.

3 QEF-LD

QEF-LD is an extension of QEF – *Query Evaluation Framework* (Porto et al., 2007) – that enables the execution of integration queries on Linked Data. QEF is a framework for the deployment of data processing applications. The framework allows developers to extend it with new operators that implement the processing semantics of the application, as much as with new data sources enabling access to data under heterogeneous formats. The application specification is exposed to QEF as an XML document known as an application execution plan. There are two types of QEF operators: algebraic and control. The former corresponds to the application semantics, whereas the latter implements the execution model through operators for data transformation and transfer. In particular, in this implementation, the operations needed to integrate RDF data in the context of Linked Data compose the application semantics and are materialized as a set of Linked Data algebraic operators. Complementarily, operations that access the data sources or cache intermediate results would take part in the set of control operators.

Each operator implements the iterator interface, executing a loop on a set of input data. The iterator model (Graefe, 1990), additionally, forces a synchronization in the chain of operators, such that a consumer-producer relationship is established between pairs of operators, defining a pipeline of results from one operator to another.

In the QEF-LD extension proposed in this paper, data is retrieved from *SPARQL endpoints*, whose underlying sources may be RDF stores or any other source of data with a translation to RDF offered by a wrapper plugged on top of it.

¹<http://jena.apache.org/documentation/query/>

²<http://www.openrdf.org/>

3.1 QEF extensions to support the Linked Data integration algebra

One of the contributions of this work is the extension of the QEF framework to support the execution of SPARQL endpoints integration queries. This work implemented a SPARQL datasource that communicates with endpoints obtaining results from SPARQL queries, and transforming them into QEF tuples. The semantics of the data integration query is implemented by special join and union operators. The former has been modified to offer scalability to large result sets by the dynamic partitioning of result sets and parallel evaluation. Additionally, minimization of data retrieval is obtained, whenever possible, using bindings of literals to conditional predicates over endpoint data. In the remainder of this section we describe each operator.

SPARQL Endpoint Data Source

The access to SPARQL endpoints is implemented as an extension to the QEF DataSource class. The implementation is based on the Jena framework³ that accesses a SPARQL endpoint and builds a QEF tuple with the query result set instances. The class offers support for named parameters.

Service operator

The *Service* operator extends the QEF Access class to implement the interface with SPARQL endpoints. During initialization, instances of the *Service* operator receive: the data source name, used as a key to obtain connection information stored in the QEF catalog; the endpoint URI, used to drive the connection to the appropriate service; and the SPARQL query to be submitted to the endpoint. The SPARQL query string may include named parameters to be replaced by *Bind* values, as described below.

Project operator

The *Project* operator extracts from the SPARQL result set the values to be considered in the output of the query and passed to the tuple data structure.

BindJoin operator

The *BindJoin* operator is an implementation of a join between two SPARQL endpoints using an equality condition based on attributes shared by the sources. First, a SPARQL query is submitted to the producer on the left of the join tree. Next, for each retrieved tuple, the values in the shared attributes are used as literals in predicates over the shared attributes from the right-hand side data source. Note that for each tuple retrieved from the left-hand side a new SPARQL query is constructed probing the source on the right. Once the results from the right-hand side data source

are obtained, a concatenation with values from the left-hand tuple produces the output of the join.

SetBindJoin operator

The *SetBindJoin* is an extension to the *BindJoin* operator (Florescu et al., 1999) adapted to join tuples produced from SPARQL endpoints data and optimized to run in parallel with the dynamic partitioning of incoming tuples. The operator behaves asymmetrically obtaining partial results from the left-hand side expression, placing them in an in-memory hash table, and probing the data in the right-hand side. The data obtained from the left-hand side is split into blocks of tuples of a certain configurable size k . The data from each block b_i is used as a binding to restrict the data obtained from the right-hand side, akin to the behavior of the original *BindJoin*. Indeed, the query submitted to the right-hand side SPARQL endpoint is rewritten having the join predicate substituted by a restriction with values from b_i , in the spirit of the Ingres decomposition algorithm (Wong and Youssefi, 1976). Once the results from the rewritten query have been obtained a new probe is done against the hash table loaded during the first stage. The execution of the k queries, originated from blocks of tuples from the left-hand side, are managed by threads of the SetBindJoin. These threads share the common hash table enabling correct evaluation of the join operation.

Union operator

The *Union* operator implements the analogous operation from the Relational model over sets of tuples, extended to union data from multiple SPARQL endpoints sharing the same data schema. The operator uses threads to consume data in parallel from each of the incoming data producers.

4 Algorithms

SetBindJoin Algorithm

The *SetBindJoin* algorithm outputs results from the parallel processing of tuple sets generated by its left producer. The grouping of tuples obtained from the left producer of the join in sets allows a reduction in the number of remote requests to SPARQL Endpoints related to the right producer of the join. It also limits the number of returned tuples, since the binding of common variables used in producers leads to the formulation of a query with lower selectivity, i.e. a more restrictive query.

The processing of each set can be briefly divided into the following steps:

- (i) Create a tuple set S with elements retrieved from the left producer of the join.
- (ii) Retrieve tuples from the right producer of the

³<http://jena.apache.org/>

join that are related with tuples from the tuple set S .

(iii) Return the join results between tuples from the set S and tuples retrieved from the right producer.

The steps are detailed below:

(i) **Create a tuple set S with elements retrieved from the left producer of the join.** The *SetBindJoin* algorithm (algorithms 1 and 2) groups the tuples retrieved from the left producer of the join in sets (Lines 6–16 of Algorithm 2). The sets have a maximum number of tuples that is pre-configured in the *SetBindJoin* operator in the query plan. That configuration is represented in our algorithm by the variable *leftTuplesSetSize*.

(ii) **Retrieve tuples from the right producer of the join that are related with tuples from the tuple set S .** The right producer of the join is cloned and existing queries in the right producer are reformulated to bind the values of common variables between the left and right producers of the join. The reformulation ensures that the right producer will only retrieve results related to tuples from the tuple set S . Clone and reformulation are performed by the *cloneAndReformulate* method on line 17 of Algorithm 2. The reformulation changes the original query using UNION and FILTER features from the SPARQL query language in order to bind variables. For example, suppose the tuple set retrieved from the left producer of the join has the tuples illustrated in Figure 1 and that the right producer has the SPARQL query shown in Figure 2.

```
(?researcher = <http://dblp.l3s.de/d2r/page/authors/Tim_Berners-Lee>,
?pub = <http://dblp.l3s.de/d2r/resource/publications/conf/www/2010ldow>),
(?researcher = <http://dblp.l3s.de/d2r/page/authors/Christian_Bizer>,
?pub = <http://dblp.l3s.de/d2r/resource/publications/conf/esws/2007sfsw>)
```

Figure 1: Example of a tuple set S' retrieved from the left producer of the join.

```
prefix dc: <http://purl.org/dc/elements/1.1/>

SELECT * WHERE {
  ?pub dc:creator ?researcher .
  ?pub dc:title ?pub_title
}
```

Figure 2: Example of a SPARQL query Q' in the right producer of the join

Figure 3 shows the query Q' after its reformulation by the *cloneAndReformulate* method. One UNION operation is used for each tuple from the set S' , except for the first tuple. Thus, for every set of n tuples, there are $(n - 1)$ UNION operations after the reformulation.

Other reformulation strategies were tested, but

```
prefix dc: <http://purl.org/dc/elements/1.1/>

SELECT * WHERE {
  { ?pub dc:creator ?researcher .
    ?pub dc:title ?pub_title
    FILTER (
      ?researcher = <http://dblp.l3s.de/d2r/page/authors/Tim_Berners-Lee> &&
      ?pub = <http://dblp.l3s.de/d2r/resource/publications/conf/www/2010ldow>
    )
  }
  UNION
  { ?pub dc:creator ?researcher .
    ?pub dc:title ?pub_title
    FILTER (
      ?researcher = <http://dblp.l3s.de/d2r/page/authors/Christian_Bizer> &&
      ?pub = <http://dblp.l3s.de/d2r/resource/publications/conf/esws/2007sfsw>
    )
  }
}
```

Figure 3: Example of a reformulated SPARQL query from the right producer of the join

they were not feasible either due to some incompatibility with most available SPARQL Endpoints or because their performance was worse than the adopted strategy. Figure 4 shows a reformulation using the BINDINGS feature proposed in the SPARQL 1.1 specification.

```
prefix dc: <http://purl.org/dc/elements/1.1/>

SELECT * WHERE {
  ?pub dc:creator ?researcher .
  ?pub dc:title ?pub_title
}
BINDINGS ?researcher ?pub {
  (<http://dblp.l3s.de/d2r/page/authors/Tim_Berners-Lee>
  <http://dblp.l3s.de/d2r/resource/publications/conf/www/2010ldow>)
  (<http://dblp.l3s.de/d2r/page/authors/Christian_Bizer>
  <http://dblp.l3s.de/d2r/resource/publications/conf/esws/2007sfsw>)
}
```

Figure 4: Example of a reformulated query using the BINDINGS feature from SPARQL 1.1

Figure 5 shows the use of FILTER combined with disjunctions of conjunctions. The last strategy proved to be much slower than the adopted strategy. All these strategies are equivalent as they retrieve the same results. However, currently available SPARQL endpoints are still not able to generate syntactically equivalent and optimized plans to many equivalent SPARQL queries. Currently equivalent queries in SQL language are usually converted to the same query execution plan. We expect this evolution will also occur in future SPARQL implementations.

```

prefix dc: <http://purl.org/dc/elements/1.1/>

SELECT * WHERE {
  ?pub dc:creator ?researcher .
  ?pub dc:title ?pub_title
  FILTER (
    (?researcher = <http://dblp.l3s.de/d2r/page/authors/Tim_Berners-Lee> &&
     ?pub = <http://dblp.l3s.de/d2r/resource/publications/conf/www/2010ldow>))
    (?researcher = <http://dblp.l3s.de/d2r/page/authors/Christian_Bizer> &&
     ?pub = <http://dblp.l3s.de/d2r/resource/publications/conf/esws/2007sfsw>)
  )
}

```

Figure 5: Example of reformulated query using FILTER and disjunction of conjunctions

All the tuples retrieved by the left producer of the join are stored in a hash table called *leftTupleHashTable* (Lines 4, 8, 11 and 17 of Algorithm 2). The hash table key is a representation of the values of the common variables between the join producers and its value is a list of tuples that share the key.

(iii) Return the join results between tuples from the set S and tuples retrieved from the right producer.

For each tuple from the right producer of the join, we retrieve a list with all left side tuples from the *leftTupleHashTable* that share the same key. Next, we go over the list to join each of its elements with the element retrieved from the right in order to return the final result of the operation (Lines 20–30 of Algorithm 2).

The resulting tuples from all sets processed in parallel are stored in a single linked blocking queue called *resultBuffer*. The *take* method from the *resultBuffer* queue (Line 7 of Algorithm 1) retrieves and removes its first element if the queue is not empty. If the queue is empty, the *take* method waits until a new element is added. The *put* method is used to insert an element at the end of the queue (Lines 27 and 33 of Algorithm 2). The *put* method waits if no space is available to insert a new element in the queue. If space is available, the queue exits the wait state and allows the insertion of new elements.

The *END_TOKEN* element is used to flag the end of processing all tuples. It is added after the last resulting tuple. The *leftProducerSetCounter* variable is used to count sets that are processed in parallel. It is incremented when a set starts to be processed and decremented at the end of processing each set. When its value is zero and no more tuples are retrieved from the left producer of the join (Line 32 of Algorithm 2), there is nothing to process and so the *END_TOKEN* can be inserted (Line 33 of Algorithm 2).

Algorithm 1: SetBindJoin - getNext

Input: leftProducer, rightProducer,
leftTuplesSetSize, resultBuffer,
processStarted,
maxNumberOfLeftProducerSets

Output: tuple

```

1 if not processStarted then
2   processStarted ← true
3   parallel
4     processTuples (leftProducer,
5                   rightProducer, leftTuplesSetSize,
6                   resultBuffer,
7                   maxNumberOfLeftProducerSets)
5   end
6 end
7 tuple ← resultBuffer.take ()
8 if tuple = END_TOKEN then
9   tuple ← null
10 end
11 return tuple

```

The *SetBindJoin* implemented in QEF-LD is configurable from parameters defined in the query execution plan. The parameters allow the definition of (i) the maximum set size and (ii) the maximum number of concurrent threads. The parameter (ii) is also the maximum number of sets (*maxNumberOfLeftProducerSets*) that can be processed concurrently. Line 18 of Algorithm 2 implements this restriction in order to avoid having too many threads awaiting processing. Higher values to parameter (ii) can open an excessive number of sockets, which can interrupt the query processing.

Union Algorithm

The Union algorithm (Algorithm 3) performs the concurrent union of tuples from multiple producers. Each thread retrieves tuples from one producer and stores them in a linked blocking queue called *resultBuffer*. If the *resultBuffer* queue is not empty, the *take* method (Line 19 of Algorithm 3) retrieves and removes its first element. Otherwise, the *take* method waits until a new element is inserted. The *put* method is used to insert a new element at the end of the queue (Lines 9 and 14 of Algorithm 3). If there is no space available to insert a new element in the queue the *put* method goes into wait state. It leaves the wait state and allows the insertion of new elements as soon as the required space is available.

The *END_TOKEN* element is used to flag the end of processing all tuples. It is added after the insertion of the last resulting tuple. A counter of concurrent processed producers called *producersCounter* is used to help identify the end of processing all tuples. It is incremented in the beginning of processing of each producer and decremented after the end of processing.

Algorithm 2: SetBindJoin - processTuples

```
Input: leftProducer, rightProducer,
        leftTuplesSetSize, resultBuffer,
        maxNumberOfLeftProducerSets
1 leftTuple ← leftProducer.getNext ()
2 leftProducerSetCounter ← 0
3 while leftTuple ≠ null do
4   leftTuplesHashTable ← createHashtable ()
5   numberOfLeftTuples ← 0
6   while (numberOfLeftTuples <
leftTuplesSetSize) and leftTuple ≠ null do
7     key ← getKeyBasedOnSharedVars
(leftTuple)
8     leftList ← leftTuplesHashTable.get (key)
9     if leftList = null then
10      leftList ← createList ()
11      leftTuplesHashTable.put (key, leftList)
12    end
13    leftList.add (leftTuple)
14    leftTuple ← leftProducer.getNext ()
15    numberOfLeftTuples++
16  end
17  changedRightProducer ←
rightProducer.cloneAndReformulate
(leftTuplesHashTable)
18  Wait until leftProducerSetCounter <
maxNumberOfLeftProducerSets
19  parallel
20    leftProducerSetCounter++
21    rightTuple ←
changedRightProducer.getNext ()
22    while rightTuple ≠ null do
23      key ← getKeyBasedOnSharedVars
(rightTuple)
24      leftTuplesList ←
leftTuplesHashTable.get (key)
25      foreach leftTuple in leftTuplesList do
26        tuple ← join (leftTuple,
rightTuple)
27        resultBuffer.put (tuple)
28      end
29      rightTuple ←
changedRightProducer.getNext ()
30    end
31    leftProducerSetCounter--
32    if leftTuple = null and
leftProducerSetCounter = 0 then
33      resultBuffer.put (END_TOKEN)
34    end
35  end
36 end
```

Thus, when its value is zero (Line 13 of Algorithm 3) there is nothing to process and the *END_TOKEN* can be added (Line 14 of Algorithm 3).

Algorithm 3: Union - getNext

```
Input: producers, processStarted, resultBuffer
Output: tuple
1 if not processStarted then
2   processStarted ← true
3   producersCounter ← 0
4   for i = 0 to producers.size () - 1 do
5     parallel
6       producersCounter++
7       prodTuple ← producers[i].getNext ()
8       while prodTuple ≠ null do
9         resultBuffer.put (prodTuple)
10        tuple ← producers[i].getNext ()
11      end
12      producersCounter--
13      if producersCounter = 0 then
14        resultBuffer.put (END_TOKEN)
15      end
16    end
17  end
18 end
19 tuple ← resultBuffer.take ()
20 if tuple = END_TOKEN then
21   tuple ← null
22 end
23 return tuple
```

5 Experiments and Results

In order to quantitatively evaluate the proposed query engine under the mashup data integration scenario with parameterized queries, we have performed several experiments using QEF-LD, and the most widely used tools, to run federated SPARQL queries: Jena, Sesame and FedX. This section discusses the results of the experiments we carried out. For that, we used efficiency as a metric that is related to query processing time and memory footprint in each evaluated SPARQL query processor.

To carry out the tests we used the following datasets: *diseasome*, *dailymed*, *sider*, *drugbank*, *dblp*, *DBpedia* and *linkedgeodata*. For each dataset we imported its data for an RDF Store using the dumps available on the Web. The OpenLink Virtuoso⁴ was used to store the RDF data and to provide a SPARQL Endpoint service.

The workload comprised five synthetic SPARQL mashup queries. The Q1, Q2, and Q3 queries were designed to evaluate the join strategies, whilst queries Q4 and Q5 were prepared with the intention of analyzing the performance of the union operations.

Queries to evaluate the join strategies

Both queries Q1 and Q2 have a single join operation, but differ principally by the amount of data returned

⁴<http://virtuoso.openlinksw.com/>

(see Table 2). Query Q3 involves two join operations and retrieves a large number of results (86,516 tuples).

Query Q1 (Figure 6) gets resources' URIs from the *linkedgeodata* dataset, together with their respective latitudes and longitudes obtained from the *DBpedia* dataset. Query Q2 (Figure 7) gets URIs of diseases and possible drugs used to treat each disease from the *diseasome* data source. In addition to these data, the full names of the drugs used in treating each disease are obtained from the *dailymed* data source.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX geopos: <http://www.w3.org/2003/01/geo/wgs84_pos#>

SELECT ?s ?lat ?long
WHERE {
  SERVICE <http://linkedgeodata.arida.ufc.br/sparql> {
    ?s owl:sameAs ?geo .
  }
  SERVICE <http://dbpedia.arida.ufc.br/sparql> {
    ?geo geopos:lat ?lat ;
    geopos:long ?long .
  }
}
```

Figure 6: Federated SPARQL Query Q1

```
PREFIX ds:
  <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome/>
PREFIX dm: <http://www4.wiwiss.fu-berlin.de/dailymed/resource/dailymed/>

SELECT DISTINCT ?ds ?dg ?dgn
WHERE {
  SERVICE <http://diseasome.arida.ufc.br/sparql> {
    ?ds ds:possibleDrug ?dg .
  }
  SERVICE <http://dailymed.arida.ufc.br/sparql> {
    ?dg dm:fullName ?dgn .
  }
}
```

Figure 7: Federated SPARQL Query Q2

Query Q3 (Figure 8) gets, initially, the name of active pharmacological agents for some drugs in the *dailymed* dataset. From these values, Q3 checks: 1) the *owl:sameAs* links with *sider*, in order to get the side effects for each drug, and 2) the links *dailymed:genericDrug* with *drugbank* to retrieve chemical formulas of drugs.

Queries to evaluate the union strategies

Queries Q4 and Q5 differ in the number of union operations performed. While query Q4 has a single union operation, query Q5 has ten union operations. Query Q4 (Figure 9) performs the union of generic names of drugs and medical treatment indi-

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dm: <http://www4.wiwiss.fu-berlin.de/dailymed/resource/dailymed/>
PREFIX db: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX sider: <http://www4.wiwiss.fu-berlin.de/sider/resource/sider/>

SELECT ?dgain ?dgcf ?sen
WHERE {
  SERVICE <http://dailymed.arida.ufc.br/sparql> {
    ?dg dm:activeIngredient ?dgai .
    ?dgai rdfs:label ?dgain .
    ?dg dailymed:genericDrug ?dgd .
    ?dg owl:sameAs ?sa .
  }
  SERVICE <http://sider.arida.ufc.br/sparql> {
    ?sa sider:sideEffect ?se .
    ?se sider:sideEffectName ?sen .
  }
  SERVICE <http://drugbank.arida.ufc.br/sparql> {
    ?dgd db:chemicalFormula ?dgcf .
  }
}
```

Figure 8: Federated SPARQL Query Q3

cations between the datasets *drugbank* and *dailymed*. The query Q5 (Figure 10) performs the union of researchers names and their publications in the DBLP dataset.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX db: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX dm: <http://www4.wiwiss.fu-berlin.de/dailymed/resource/dailymed/>

SELECT ?gn ?indication
WHERE {
  {
    SERVICE <http://drugbank.arida.ufc.br/sparql> {
      ?dn db:genericName ?gn ;
      db:indication ?indication .
    }
  }
  UNION {
    SERVICE <http://dailymed.arida.ufc.br/sparql> {
      ?dn dm:name ?gn ;
      dm:indication ?indication .
    }
  }
}
```

Figure 9: Federated SPARQL Query Q4

Execution

To measure efficiency, we have submitted 10 execution cycles for each of the five queries comprising the designed workload. Soon, 50 execution cycles were performed. Each execution cycle involved two executions of the same query.

In an execution cycle, the first query usually un-

```

PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?pub_title where {
  {
    SERVICE <http://dblp01.arida.ufc.br/sparql> {
      ?publication dc:creator ?dblp_researcher ;
      dc:title ?pub_title .
      ?dblp_researcher rdfs:label ?label .
      FILTER regex(?label, "^Aab")
    }
  }
  ...
} UNION {
  SERVICE <http://dblp10.arida.ufc.br/sparql> {
    ?publication dc:creator ?dblp_researcher ;
    dc:title ?pub_title .
    ?dblp_researcher rdfs:label ?label .
    FILTER regex(?label, "^Jab")
  }
}
}

```

Figure 10: Federated SPARQL Query Q5

derperformed due to the startup of the Java virtual machine that prepares and allocates the necessary resources. However, the second query occurred on the same virtual machine instance, where all the resources were already available. For this reason, for each execution cycle, we ignored the response time of the first query run. Then, we took into account only the response time of the second query run.

We note that the SPARQL query engines Jena, Sesame and FedX are designed to evaluate ad-hoc SPARQL queries, dynamically generating a federated query execution plan. QEF-LD takes a different approach for dealing with mashup integration queries. In design time, an efficient data integration join ordering is computed for a given mashup query, which is associated to the corresponding SPARQL query during execution. In this scenario, a more adequate execution profile can be guaranteed.

Test Environment

Two nodes comprised the test environment: a server and a client. A local network connected these machines. The server machine hosted the OpenLink Virtuoso, which stored the RDF data and provided a SPARQL Endpoint service to each dataset used in the workload: *diseasome*, *dailymed*, *sider*, *drugbank*, *dblp*, *DBpedia* and *linkedgeodata*. The client machine hosted the evaluated SPARQL query engines: Jena, Sesame, FedX and QED-LD.

The server machine used in the experiments used an Intel Core i7 2.93GHz with 16 GB RAM DDR3 1333 MHz. The client machine used during the tests used an Intel Core 2 Duo 2.93GHz with 2GB RAM

667 MHz. During the experiments, both the server and the client machine were dedicated to the test tasks.

Experimental Results

In order to evaluate the efficiency of the SPARQL query engines Jena, Sesame, FedX and QED-LD regarding the federated SPARQL queries processing we used two metrics: 1) the query response time and 2) the maximum amount of memory used by the Java virtual machine during each query run.

Performance evaluation of join operations

The join operator used by QEF-LD was the *SetBindJoin*. This operator uses *threads* to run queries in parallel. A maximum of one hundred concurrent threads was chosen to be used for all queries involving the *SetBindJoin* operator. This value was chosen due to the following experimental observation. Fixing the other *SetBindJoin* parameters and varying only the number of concurrent threads, the best performance results were obtained when this value was near one hundred. Furthermore, limiting the maximum number of concurrent threads avoids problems arising from the opening of a large number of socket connections. Moreover, fewer concurrent threads may result in a lower throughput. Then, we observed that there are values for the maximum number of concurrent threads that lead to a balance between data production (SPARQL Endpoint) and data consumption (QEF-LD), which maximizes the throughput. For the environment used in our experiments, this value was close to the one hundred.

Sesame (version 2.6.5.) did not return data for Q1 (see nr – no results – reference in Table 1). The query Q1 was executed. Sesame did not crash. However, it did not return results even after running for hours. No result was obtained. No error message was returned. We also noted no excessive memory consumption (approximately 180MB).

FedX did not return data for Q1 and Q3. During the execution of Q1 and Q3, FedX used all available memory for the Java virtual machine and, after some time, threw an exception related to the lack of available memory (see oom – OutOfMemory – references in Table 1).

Figures 11 and 12 show that the most effective strategy, related to query response time, for queries Q1 to Q4 was QEF-LD. In the query Q5, obtained the results in slightly more time than FedX. However, in queries Q1, Q2 and Q3, QEF-LD obtained considerably smaller query response times than the other evaluated SPARQL query engines.

Moreover, the *SetBindJoin* operator implemented in QEF-LD generally consumed more memory than equivalent operators in others evaluated SPARQL

	Jena	Sesame	FedX	QEF-LD
Q1	382.649	nr	oom	50.808
Q2	39.530	47.239	12.576	1.017
Q3	88.531	339.741	oom	7.416
Q4	0.813	0.642	0.636	0.556
Q5	375.226	375.900	208.155	214.457

Table 1: Comparison table showing execution times (seconds) of queries Q1–Q5.

Query	Q1	Q2	Q3	Q4	Q5
# results	43,016	6,124	86,516	5,146	18,327

Table 2: Number of results of queries Q1–Q5.

query engines. This memory consumption was mainly due to the need to temporally store data required to build the joins results. The use of multiple threads also increases memory consumption. The Q3 query is quite different from Q1 and Q2 (other queries that use join operations) since Q3 performs two join operations (instead of just one) and returns more results. It is important to note that, in Jena, the amount of memory used by Q3 was much greater than that used by Q1. However, in QEF-LD queries Q1 and Q3 did not suffer a significant difference in memory consumption. Besides this, Sesame provided the lowest memory consumption among the evaluated tools.

Analyzing Figure 14 one can see that in query Q1, from the value 20, increasing the size of the sets makes queries slower. Therefore, it is important to find a balance between data production and consumption to maximize query performance. The maximum size of the sets used in queries Q1, Q2 and Q3 was 57. We could not use larger sets because the Virtuoso server does not allow queries with more than 57 union operations. Then, as the *BINDING* strategy used by *SetBindJoin* involves query reformulation using several union operations (See Section 4),

Performance evaluation of union operations

Regarding the union operation, Sesame and FedX stood out for their smaller memory consumption compared to other evaluated tools. For the first time in the experiments, FedX achieved satisfactory results with respect to memory usage. However, the QEF-LD memory footprint was larger compared to the other strategies, which means it is an important aspect to be improved in the QEF-LD Union algorithm.

In the query Q5, FedX and QEF-LD had similar response times. Furthermore, FedX and QEF-LD proved to be almost twice as fast as other evaluated query engines. This performance gain is due to the use of threads. However, the memory consumption presented by QEF-LD was greater than the other

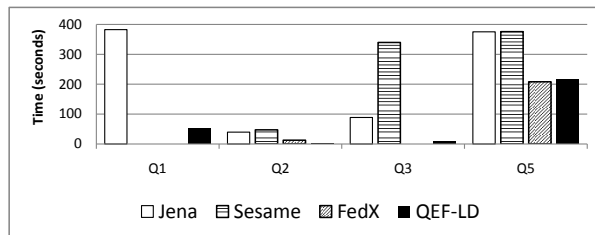


Figure 11: Comparison chart showing execution times of queries Q1, Q2, Q3 and Q5.

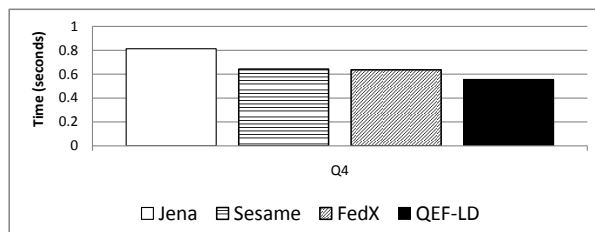


Figure 12: Comparison chart showing execution times of query Q4.

tools, which means it is an important aspect to be improved in the QEF-LD union algorithm.

We tried to run the designed workload (queries Q1 to Q5) over the original data available on the Web several times. However, these queries burdened the endpoints, sometimes causing service interruption. In other cases, the endpoint servers limited the results, threw exceptions, and added error messages (like *"Premature end of file"*). Unfortunately, these problems hindered the experiments in the real and uncontrolled Web environment. Still, in the future we intend to design and run over the Web environment a workload containing queries with greater selectivity in order to reduce the amount of data retrieved and, so, facilitate results achievement.

6 Conclusions and Future Work

This paper addresses the processing of federated query plans on the Web of Data using QEF-LD, which is a query execution engine that extends QEF – Query Evaluation Framework. QEF-LD exploits intraoperator parallelism, reduction in the number of remote calls and reduction in the selectivity of queries to remote endpoints in order to improve the performance of query execution. Furthermore, QEF-LD is fully compatible with the SPARQL 1.0 query language that allows clients to integrate with available SPARQL endpoints.

Experiments demonstrated the feasibility of using QEF-LD operators. The *SetBindJoin* operator implemented in QEF-LD obtained considerably smaller ex-

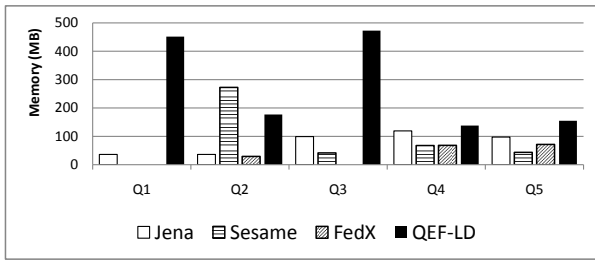


Figure 13: Comparison chart showing memory usage of queries Q1–Q5.

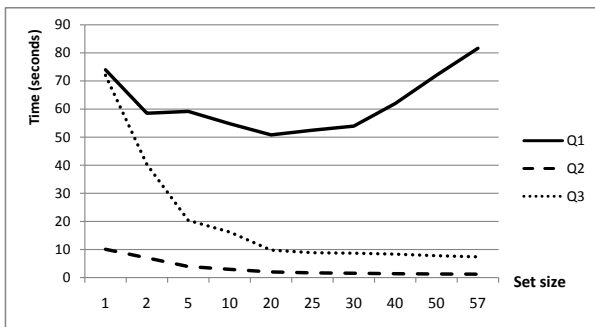


Figure 14: Comparison chart showing execution times of queries Q1–Q3 for different set sizes.

execution times than other strategies. The best results were obtained when larger sets were used, which reduced the number of calls to remote endpoints and the execution time. On the other hand, the storage of results in sets led to a larger memory footprint. Fortunately the memory footprint did not become excessive, since the space reserved for sets can be continuously released and allocated for new sets. The results related to the Union operator in QEF-LD were also satisfactory. The union operator in QEF-LD used parallelism to achieve considerably improved performance over other strategies. Therefore, the higher level of parallelism allows more connections to transfer data simultaneously, which increases the throughput.

The main challenges to be addressed in the future include: (i) adding new efficient operators to QEF-LD; (ii) creating adaptive operators to address the aspect of unpredictability in the Web of Data; (iii) using data cache, indexes and statistics to improve query performance; (iv) creating a framework to automate all steps of federated query processing, where QED-LD will be used as the query execution engine; (v) adding support for adaptive processing of ad-hoc queries.

REFERENCES

- Florescu, D., Levy, A., Manolescu, I., and Suciu, D. (1999). Query optimization in the presence of limited access patterns. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 311–322, New York, NY, USA. ACM.
- Graefe, G. (1990). Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 102–111, New York, NY, USA. ACM.
- Langegger, A. (2010). *A Flexible Architecture for Virtual Information Integration based on Semantic Web Concepts*. PhD thesis, J. Kepler University Linz.
- Porto, F., Tajmouati, O., Da Silva, V. F. V., Schulze, B., and Ayres, F. V. M. (2007). Qef - supporting complex query applications. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, CCGRID '07*, pages 846–851, Washington, DC, USA. IEEE Computer Society.
- Prud'hommeaux, E. and Buil-Aranda, C. (2011). SPARQL 1.1 Federated Query. <http://www.w3.org/TR/sparql11-federated-query/>.
- Quilitz, B. and Leser, U. (2008). Querying Distributed RDF Data Sources with SPARQL. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 524–538, Berlin, Heidelberg. Springer-Verlag.
- Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011a). Fedx: a federation layer for distributed query processing on linked open data. In *Proceedings of the 8th extended semantic web conference on The semantic web: research and applications - Volume Part II*, ESWC'11, pages 481–486, Berlin, Heidelberg.
- Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011b). Fedx: optimization techniques for federated query processing on linked data. In *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ISWC'11, pages 601–616, Berlin, Heidelberg. Springer-Verlag.
- Wong, E. and Youssefi, K. (1976). Decomposition – a strategy for query processing. *ACM Trans. Database Syst.*, 1(3):223–241.