QFed: Query Set For Federated SPARQL Query Benchmark

Nur Aini Rakhmawati
INSIGHT Centre, National University of Ireland
National University of Ireland, Galway
nur.aini@deri.org

Muhammad Saleem
Universität Leipzi
IFI/AKSW, PO 100920,
D-04009 Leipzig
saleem@informatik.uni-leipzig.de

Sarasi Lalithsena
Kno.e.sis Center
Wright State University,Dayton, OH, USA
sarasi@knoesis.org

Stefan Decker
INSIGHT Centre
National University of Ireland, Galway
stefan.decker@deri.org

ABSTRACT

The increasing attention for federated SPARQL query systems emphasize necessity for benchmarking systems to evaluate their performance. Most of the existing benchmark systems rely on a set of predefined static queries over a particular set of data sources. Such benchmark are useful for comparing general purpose SPARQL query federation systems such as FedX, SPLENDID etc. However, special purpose federation systems such as TopFed, SAFE etc. cannot be tested with these static benchmarks since these systems only operate on a specific data sets and the corresponding queries. To facilitate the process of benchmarking for such special purpose SPARQL query federation systems, we propose QFed, a dynamic SPARQL query set generator that takes into account the characteristics of both dataset and queries along with the cost of data communication. Our experimental results show that QFed can successfully generate a large set of meaningful federated SPARQL queries to be considered for the performance evaluation of different federated SPARQL query engines.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Linked Data, Data Integration, Federation SPARQL Query

iiWAS '14 Hanoi, Vietnam

1. INTRODUCTION

Over the last years, the Web of Data has grown significantly and currently comprise of over 2000 data sources[1] from diverse domains. Consequently, SPARQL query federation approaches [15, 5, 1, 17, 11] has gained a significant importance to collect data from these globally distributed Linked Data sources. General purpose SPARQL query federation systems (e.g., [15, 5, 1] etc.) have been actively used for SPARQL query federation over multiple SPARQL endpoints (Linked Data storage servers). Beside these, specialised federation systems which are optimized for a specific use-case or data set (e.g., TopFed[2] [9, 10], SAFE[3] etc.) has gain a considerable attention as well. On the other hand, FedBench [13, 14] has been developed to compare and position such systems. While Fedbench are useful to test general purpose SPARQL query federation systems, it cannot be used for the performance evaluation of specialized federation systems. This is because this benchmark comprises of predefined queries and a set (can be singleton) of data sources which may be completely irrelevant for the specialized federation system in hand. For example, TopFed [9, 10] is a specialized federation system, particularly designed for Linked Cancer Genome Altas (LTCGA[4]) [10] data set . However, to the best of our knowledge, non of the existing SPARQL query federation benchmarks include LTCGA data set and queries. Similarly, SAFE is developed for SPARQL query federation over Linked statical data using RDF DataCube vocabulary and thus it is essential to test this system with RDF DataCube data sets and corresponding SPARQL queries. In a nutshell, to test such specialized systems, we need a dynamic query generator which can generate a variety of federated SPARQL queries over the given set of data sources.

To this end, we propose *QFed*, a dynamic query set generator for federated SPARQL query benchmarks that takes into account both key characteristics of the dataset and SPARQL queries which has a direct impact on the perfor-

[1]LOD Stats: `http://stats.lod2.eu/`
[2]TopFed federation engine: `https://code.google.com/p/topfed/`
[3]SAFE federation engine: `http://linked2safety.hcls.deri.org:8080/SAFE-Demo/`
[4]Linked TCGA: `http://tcga.deri.ie/`

mance evaluation of federated engines. *QFed* considers key SPARQL query characteristics such as the number of sources the query span (i.e, collect results), the type of triple pattern joins (star, path, hybrid [12]), use of different SPARQL clauses, the number of query triple patterns, and shared variables, etc. for SPARQL query generation. Characteristics of the dataset include the distributions of classes and properties, the frequencies of classes and properties, the number of sources, and the network cost etc [8]. Our evaluation shows *QFed* is successfully able to generate a large set of meaningful federated SPARQL queries for a given set of interlinked data sets. Selected queries can then later be used for the performance evaluation of any general or specialized SPARQL query federation engine.

In the remainder of this paper starts with reviewing related works in Section 2. Then we explain the query set generation in Section 3. The system is evaluated in Section 4 with concrete proposed metrics, query set, a dataset and system . Finally, we conclude it in Section 5.

2. RELATED WORKS

FedBench [13], to the best of our knowledge, is the only SPARQL query federation benchmark that encompass real queries (i.e., showing typical requests) spanning (collects data) over multiple real datasets. DAW [12] provides a set of static queries[5] based on characteristics of BSBM (Berlin SPARQL Benchmark) queries [4] from four[6] public datasets. These queries cover most of SPARQL operators and keywords. However, all the queries are statically generated thus cannot be used for specialized federation systems. Furthermore, these queries are simple in complexity (maximum of 4 triple patterns per query). SPLODGE [6] offers a tool that can generate a query set based on query structural, complexity, and cardinality constraints. LidaQ [16] also produces queries based on three main shapes (entity, star and path shapes) for federated queries benchmark. The star shape in Lidaq and Splodge is a local join subject-subject pattern which combines triple patterns in a single source. Both SPLODGE and LidaQ rely on the interlinks between datasets such as owl:sameAs, rdfs:seeAlso, etc. Since not all entities are interlinked with each other, we propose a query set generator which also uses objects and subjects comparisons between two or three sources without only relying on interlinkings between datasets. We will detail the differences among QFed, Splodge and Lidaq in Table 1.

[7] and [6] provide a list of query requirements for benchmarking federated SPARQL queries such as number of sources involved, complexity, selectivity/result size, general predicate (e.g rdfs:label, rdf:type), shape, variable position etc. Except for variable position, our query generator takes into account all the query requirements. With respect to communication cost, we add object value types (URI, blank-node or a literal) as one of requirements. The number of characters of a literal object can be a lot higher than the number of characters of an URI object. Consequently, querying such literal objects can lead to increased bandwidth usage. Furthermore, it can reduce the speed of the query execution, if the data sent or received is over the network capacity.

3. QUERY SET GENERATION

[5]https://sites.google.com/site/dawfederation/queries
[6]https://sites.google.com/site/dawfederation/data-sets

This section explains the foundations of federated SPARQL query processing. After that, we describe our methodology to generate a query set in Section 3.2.

3.1 Background

We provide a definition of the federated SPARQL query framework to formalize our queries generation method. In a nutshell, the federated SPARQL query framework consists of a federated engine as a mediator and a set of RDF sources. As the mediator, a federated engine plays an important role to distribute query to the most relevant sources [2]. Each source is normally accessed via a SPARQL endpoint and contains a set of triple statements which break down into three components: subject, predicate and object. Those components can be either an URI, literal or blank-node. Let U be the set of all URIs, L be the set of all literals and B be the set of all blank nodes. Then a triple t can be formally defined as $t = (s, p, o) \in (U \cup B) \times U \times (U \cup L \cup B)$. An entity is part of a dataset if it can be described by a triple that contains *rdf:type* predicate. An entity might be connected to other entities that are located in other sources. For instance, Figure 1 describes a relation between three entities: siderdrug:450(*Femring*) in the Sider dataset, diseasome:382(*Estrogen Resistance*) in the Diseasome datset and dailymeddrug:363(*Vivelle*) in the Dailymed dataset. The oval shape represents a URI or a blank node and the rectangle shape depicts a literal. Those components are connected by the predicates (arrows). According to Figure 1, siderdrug:450 is a subject, siderdrug:drugName is a predicate and Vivelle is a literal object. One of interlinked entities is shown that dailymeddrug:363 is linked to diseasome:382 via the dailymed:possibleDiseaseTarget predicate. In general, those links are useful for federated query processing from multiple sources.

3.2 Methodology

We deploy the following method for generating queries: Given a set of sources, we create query templates based on particular join patterns. Since we don't only want to create a join between sources, we also span the query by adding more triple patterns. In order to add those triple pattern and to take into account the cost of data communication, we utilize two predicate selection strategies: *properties distribution not considered* and *properties distribution considered*. In addition, we use big literal objects values (explained in Equation 2) to increase the communication cost. Furthermore, in order to change the selectivity value of a query, we cover two widely used keywords: FILTER and OPTIONAL and analyse the effect of those keywords on the data communication cost. In summary, our query set generation steps are described as follows:

1. We calculate the occurrences and frequencies of classes and properties as two relevant parameters for query set generation (Section 3.3)

2. We identify the list of joins that can be found between two entities at two different sources based on subject-object join pattern, subject-subject join pattern and object-object join pattern. After that, we create query patterns between two entities which form that join pattern.

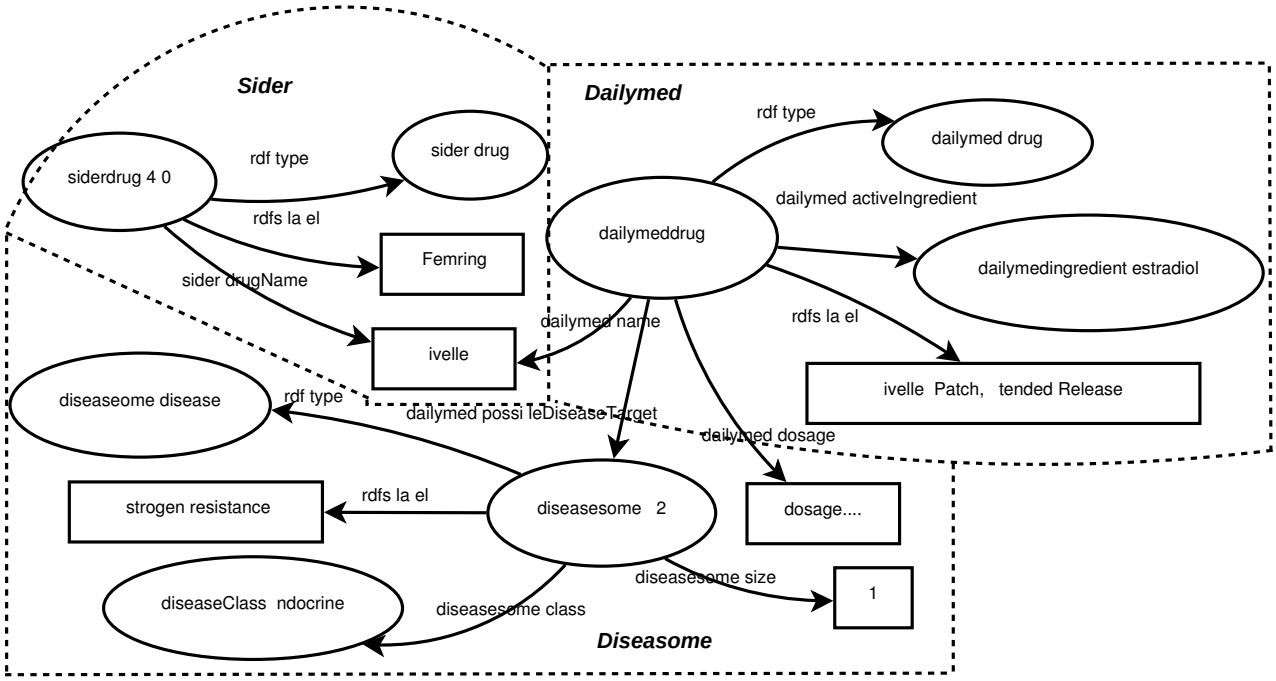


Figure 1: Dataset Example

3. We add more query patterns to create a star shape where the second entity from the previous step become the centre of the shape. The star shape is useful to retrieve a list of information related to the second entity. The query patterns consist of a URI and a literal object value. The predicates of the query pattern are selected based on two methods that will be explained in Section 3.5.

3.3 Data Preprocessing

We pre-process all datasets involved before the query set generation to obtain the characteristics of the datasets. We calculate the occurrences of classes and predicates regardless of the frequencies of classes and predicates in each source. The occurrence calculation aims to see how the predicate and class are spread across the datasets; as a federated engine — the query mediator — generally exploits a data catalogue which contains a list of predicates and classes. In other words, occurrence in this work is related to the distribution of class and property. Based on our previous work [8], if the classes and predicates are distributed over the dataset, the federated engine will send a request to more than one dataset for detecting the most relevant sources for a sub query. In addition, we compute the total frequencies predicates and classes in the datasets since they are highly associated with the number of intermediate results that are received by the federated engine during query execution.

DEFINITION 1. *Let $D = \{d_1, d_2, \cdots, d_n\}$ be a set of sources that are used in the federation framework, P be a set of known predicates and C being a set of known classes. Then the occurrence $O_p(p, D)$ of predicates $p \in P$ in the dataset D is computed as $O_p(p, D) = \sum_{d \in D} Op(p, d)$, where $Op(p, d)$ is a function that returns 1 if there exists one or more triple with predicate p in source d, otherwise it returns 0. The frequency $f_p(p, D)$ of predicate p in the dataset D is defined as*

$$f_p(p, D) = \sum_{d \in D} \sum_{t \in d} \begin{cases} 1 & \text{if } \exists s, o : t = (s, p, o) \\ 0 & \text{otherwise} \end{cases}$$

And the occurrence $O_c(c, D)$ of class $c \in C$ is calculated as $O_c(c, D) = \sum_{d \in D} Oc(c, d)$. Similar to Op(p,d), Oc(c,d) is a function that returns 1 if there exists one or more triple with class c, otherwise it returns 0. The frequency $f_c(c, D)$ of class c in the dataset D is defined as

$$f_c(c, D) = \sum_{d \in D} \sum_{t \in d} \begin{cases} 1 & \text{if } \exists s : t = (s, rdf : type, c) \\ 0 & \text{otherwise} \end{cases}$$

Consider Figure 1 as a dataset example, the $O_p(rdfs : label, D)$ and $f_p(rdfs : label, D)$ equal to three, while the $O_c(dailymed : drug, D)$ and $f_c(dailymed : drug, D)$ equal to one.

3.4 Query Join Pattern Types

In order to generate a SPARQL query template, we introduce the SPARQL query definition. A SPARQL query is a set of Basic Graph Patterns $(BGP)=(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$ where V is a set of all variables . A BGP might share one or more of the same variables with other BGP either in the subject, predicate or object position. These shared variables create AND join inter BGPs. There are six types of join triple patterns based on the position of the variable in the pattern: Subject-Subject, Predicate-Predicate, Object-Object, Subject-Predicate, Subject-Object, and Predicate-Object. We only consider Object-Object, Subject-Subject, Subject-Object join types since the joining shared variable in the predicate position is rarely used in real world queries [3].

3.4.1 Subject-Object

In federated SPARQL queries, the Subject-Object join pattern is normally used for discovering a relationship between two datasets. This join pattern exploits the links amongst datasets which are generated by publishers with

predicates such as owl:sameAs and dailymed:possibleDiseaseTarget in Figure 1. In the context of the federated SPARQL query, a link refers to a predicate which join two entities that are located in different sources.

DEFINITION 2. *Given sources $d, d' \in D$, the set of triples $\mathcal{L}(d, d')$ in source d which contains a link that joins entity s in source d to entity s' in source d' can be formally defined as follows:*
$\mathcal{L}(d, d') = \{(s, p, s') | \exists (s, p, s') \in d \wedge s' \in U \wedge \exists (s', p', o') \in d' \wedge d \neq d'\}$

Once we obtain the link p between entity s and entity s' from different sources, we add two triple patterns to create a star shape where the entity s' in source d' is the center of this star shape. For each entity s' in the $\mathcal{L}(d, d')$, we fetch the list of the predicates of the entity s' along with its objects in other source d' ($PObjType(s', d')$). A strategy for choosing which triple patterns should be added in the query will be described later in Section 3.5.

$$PObjType(s, d) = \{(p, typef(o, d)) | \exists (s, p, o) \in d \wedge p \neq rdf : type\} \quad (1)$$

where $typef(o, D)$ is a function to decide the type of object value which is defined as follows:

$$typef(o, D) = \begin{cases} u & \text{if } o \in U \\ l & \text{if } o \in L \wedge length(o) < \text{avgl(D)} \\ bl & \text{otherwise} \end{cases} \quad (2)$$

As shown in Equation 2, the big literal object (bl) is a literal which has length of more the average number characters in the literal objects in the dataset($avgl(D)$). u and l denote URI object and literal object respectively. The motivation of adding big literal object value in the query pattern is to asses how optimize a federated engine to deal with the cost of data communication and the restriction of SPARQL Endpoint.

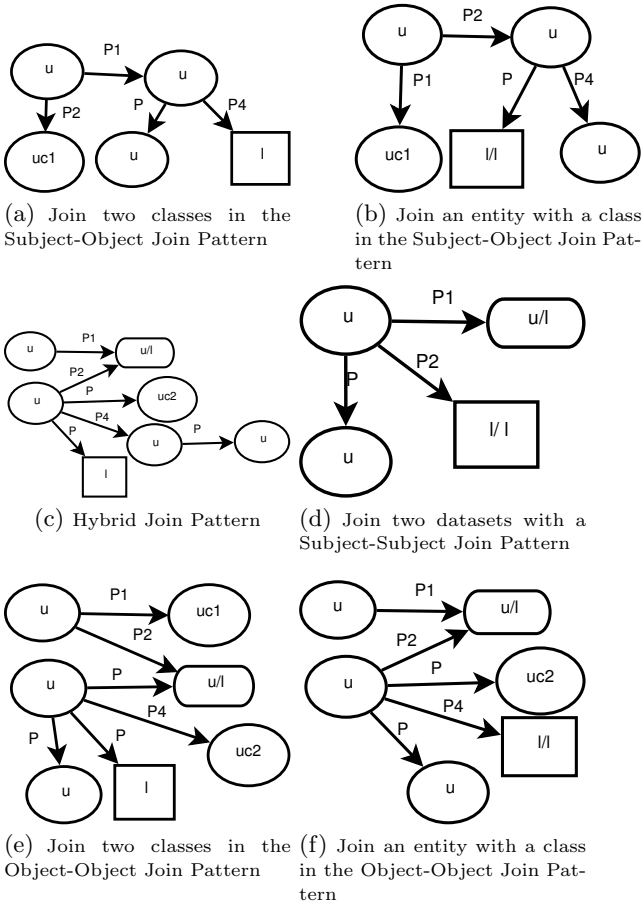
For subject-object joins, we provide two templates: 1) joining two classes (Figure 2(a)), and 2) joining an entity with a class (Figure 2(b)). The first query template is a low selectivity query because it maps the all entities that belong class $uc1$ in source d to all the entities in the source d'. $uc1$ is one of classes in source d. The following query pattern is an example of the first query template which join all entities in class dailymed:drug with entities in diseasome:disease.

```
?s1 a dailymed:drugs .
?s1 dailymed:possibleDiseaseTarget ?s2 .
?s2 diseasome:class ?URI .
```

We only generate the first query template if we find a link that occurs in more than one triples belonging to the same class, but belonging to different entities. According to [3], Constant subject-Constant predicate-Variable Object is widely used in DBPedia queries. Therefore, we also create the second query template with the aim of joining an entity in a source to other class in other sources.

```
dailymeddrug:363 dailymed:possibleDiseaseTarget ?
    s2 .
?s2 diseasome:class ?URI .
```

The above query patterns presents the second template that connects entity dailymeddrug:363 to all entities in diseasome:disease. We then append two triple patterns containing URI object variables (*?u*) and literal object variables (*?l*) in the first query template. Since the second query template join is a high selectivity query, we add big literal object variables (*?bl*) rather than literal object variable (?l) if we discover the big literal object.



(a) Join two classes in the Subject-Object Join Pattern

(b) Join an entity with a class in the Subject-Object Join Pattern

(c) Hybrid Join Pattern

(d) Join two datasets with a Subject-Subject Join Pattern

(e) Join two classes in the Object-Object Join Pattern

(f) Join an entity with a class in the Object-Object Join Pattern

Figure 2: Federated Query Templates

3.4.2 Object-Object

In order to deal with the problem of unavailability of interlinks amongst sources, we also generate a query that merges two sources by using object comparison. We create an object-object join pattern from two sources where two triple patterns share the same variable. We initially construct a list of quadruples $\mathcal{OO}(D)$ that consists of a subject object pair located in a single source and a subject object pair from another source that have the same object value.

DEFINITION 3. *Let D be a finite set of sources in the query federation frameworks, the list of quadruple for creating object-object join pattern can be formulated as:*
$\mathcal{OO}(D) = \bigcup_{d,d' \in D} \{(s, p, s', p') | (s, p, o) \in d \wedge (s', p', o) \in d' \wedge d \neq d'\}$

Like the Subject-Object join pattern, we propose two query templates that join two classes from different sources and join an entity in a source to a class in another source. Those two query templates also use the $PObjType(s, d)$ function to find other triple patterns that are associated with entity s'. Later on, the outcome of $PObjType(s', d')$ is chosen to be added to the query.

3.4.3 Subject-Subject

Having the same subject located in multiple sources normally happens when a source is divided into several partitions due to a special reason such as data clustering. A subject-subject join pattern is usually used for smushing data which has the same identifier. To begin with, we create a list of subjects that are located in multiple sources as follows:

DEFINITION 4. *Let D be a finite set of sources in the federation frameworks, the list of subjects for creating subject-subject join pattern can be formulated as:*
$\mathcal{SS}(D) = \bigcup_{d,d' \in D} \{s | (s,p,o) \in d \wedge (s,p',o') \in d' \wedge d \neq d'\}$

Figure 2(d) illustrates a query template for Subject-subject join pattern that links two different datasets. All of the query patterns share the same subject variables. The first query pattern is obtained from the first source (?u p ?u/l), and the rest of the query patterns are obtained from the second source. Similar to the subject-object join pattern and object-object join pattern, we utilize $PObjType(s,d)$ function to add the query patterns for all of the subjects in $\mathcal{SS}(D)$.

3.4.4 Hybrid Join

The aforementioned query templates only integrate two sources. Hence, we combine the object-object join pattern query template and subject-object query template. In the first step, we create a similar query template to the path that joins an entity to a class in the object-object template (Figure 2(b)). Then, we iterate each predicate of s' in the $PObjType(s',d')$ to find the predicate that also belongs to $DL(D)$.

DEFINITION 5. *Let D be a set of sources in the federation frameworks, a set of predicates $DL(D)$ that links an entity in source d to an entity in another source d' in dataset D is formulated as:*
$DL(D) = \{p | (s,p,o) \in \bigcup_{d \in D} \bigcup_{d' \in D} \mathcal{L}(d,d')\}$

In order to avoid the reciprocal query to the first source, we reject a predicate whose object equals to entity s which is the subject of the first source . Let s'' be an object of a triple in source d' whose predicate is an element of $DL(D)$, then we look up the list of predicates that belongs to s'' in the source d'' by using $PObjType(s'',d'')$. In the final step, we do the same procedure (Section 3.5) to select a predicate that will be the part of the last query triple pattern. For the last query pattern, we only choose a predicate with the literal object value or URI object value to reduce the complexity of the query.

3.5 The Predicates Selection

The objective of the predicate selection is to find suitable predicates that will be one component of triple query patterns added to the query. We propose two approaches for predicate selection: 1) *properties distribution not considered* (ND) 2) *properties distribution considered*(D). In the first approach, we just do an iteration for each pair of the predicate and its object type for entity s in $PObjType(s,d)$. The iteration will stop once we get a predicate with URI object value and a predicate with literal object value. In the case of enabled big literal options, we iterate $PObjType(s,d)$ until we discover a predicate with the big literal object value and a predicate with the URI object value. If we cannot find a predicate with the big literal object value, we choose a predicate with literal object value instead. For instance, we create a subject-object query from Dailymed and Diseasome sources based on data in Figure 1. The results of $\mathcal{L}(Dailymed, Diseasome)$ is ($dailymeddrug$: 363, $dailymed$: $possibleDiseasetarget$, $diseasome$: 382). Then, we find all predicates that belong to diseasome:382 by using $PObjType(diseasome : 382, Diseasome)$. Suppose we obtain (diseasome:size,l), (rdfs:label,l), (diseasome:class,u) sequentially, we only choose diseasome:size and diseasome:class. The rdfs:label is not selected since its position is after diseasome:size. As a result, we generate Query 1.

Query 1: Example of a Subject-Object Join Pattern Query

```
select * {   ?s1 a dailymed:drugs .
 ?s1 dailymed:possibleDiseaseTarget ?s2 .
 ?s2 diseasome:class ?URI .
 ?s2 diseasome:size ?LITERAL .  }
```

Query 2: Example of a Object-Object Join Pattern Query

```
select * {  ?s1 a sider:drug .
 ?s1 sider:drugName ?o .
 ?s2 a dailymed:drugs .  ?s2 dailymed:Name ?o .
 ?s2 dailymed:activeIngredient ?URI .
 ?s2 dailymed:dosage ?BIGLITERAL .  }
```

The second approach chooses the predicate p with the highest occurrence $O_p(p,D)$ since federated engines generally exploit a data catalogue that contains the list of predicates to select the relevant source of a query. Using predicates that are distributed over the dataset will increase the data communication cost between the federated engine and SPARQL endpoints [8]. To consider this example, we create a federated query by implementing an object-object join pattern from Sider and Dailymed sources in Figure 1. The $\mathcal{OO}(D)$ produces ($siderdrug$: 450, $sider$: $drugName$, $dailymeddrug$: 363, $dailymed$: $Name$) since sider:drugName and dailymed:Name have the same objects value ($Vivelle$). As the next step, we find all predicates and their type for entity dailymeddrug:363 ($PObjType(dailymeddrug : 363, Dailymed)$). We get $PObjType(dailymeddrug : 363, Dailymed) = \{(dailymed : activeIngredient, u), (rdfs : label, l), (dailymed : dosage, bl), (dailymed : possibleDiseaseTarget, u)\}$. In this query generation, we pick dailymed:activeIngredient and dailymed:dosage, since we consider the predicate occurrence and the big literal option. Although rdfs:label occurrence is higher than dailymed:dosage, dailymed:dosage is preferred rather than rdfs:label since we give higher priority to the predicate with the big literal than the predicate with the higher predicate occurrence.

In the next example, we extend our previous federated query example to create a hybrid join query. One element of $PObjType(dailymeddrug : 363, Dailymed)$ is $dailymed$: $possibleDiseaseTarget$ which is also a predicate that points to diseasome:382 at Diseasome source. Therefore, we can create a subject-object join pattern by employing $dailymed$: $possibleDiseaseTarget$. As shown in the first example, we retrieve values of $PObjType(diseasome : 382) = \{(diseasome : size, l), (rdfs : label, l), (diseasome : class, u)\}$. Since $O_p(rdfs :$

label) is greater than $O_p(diseasome : size)$ and $O_p(diseasome : class)$, we choose rdfs:label be the predicate for the triple pattern with URI object value. Note that, for the hybrid join, we disregard the type of object value. Furthermore, in order to simplify the query, we do not select a predicate with the big literal value .

The last example is a subject-subject join pattern federated query. Suppose that the related triples with dailymeddrug:363 as the subject that are located in two different sources, *dailymeddrug:363 dailymed:name "Vivelle"* is in the first source and the rest of the triples are in the second source. If we also consider big literal value as one of the parameters, then we can create Query 3.

Query 3: Example of a Subject-Subject Join Pattern Query

```
select * {
?s dailymed:Name ?LITERAL .
 ?s dailymed:possibleDiseaseTarget ?URI .
 ?s dailymed:dosage ?BIGLITERAL . }
```

Query 4: Example of a Hybrid Join Pattern Query

```
select * {
diseasome:382 sider:drugName ?o .
 ?s2 a dailymed:drugs . ?s2 dailymed:Name ?o .
 ?s2 rdfs:label ?LITERAL .
 ?s2 dailymed:possibleDiseaseTarget ?s3 .
 ?s3 rdfs:label ?LITERAL3 . }
```

Query 5: Example of a Federated SPARQL Query Using FILTER and OPTIONAL Keywords

```
select * { ?s1 a dailymed:drugs .
?s1 dailymed:possibleDiseaseTarget ?s2 .
?s2 diseasome:class ?URI .
OPTIONAL{ ?s2 diseasome:size ?LITERAL . }
FILTER (?LITERAL >= 1) }
```

3.6 Query set Generation Extension

3.6.1 query set Threshold

Since the number of entities is quite large, we limit the number of queries based on two parameters: the frequency of predicates ($f_p(p, D)$) and number of entities for each class. The goal of the first parameter is to take a subset of $\mathcal{L}(d, d')$, $\mathcal{OO}(D)$ or $\mathcal{SS}(D)$ that will be processed in the query generation. The subset consists of the top-K predicates with the highest $f_p(p, D)$. We span the results of the first parameter into several query generations. However, we give the second parameter as a constraint to limit the number of entities of each class. We restrict only n entities for each class with the same predicates to be generated since several entities in the same class may have the same predicate, which is also part of $\mathcal{L}(d, d')$ or $\mathcal{OO}(D)$.

3.6.2 OPTIONAL, FILTER and SERVICE Keywords

A predicate that belongs to an entity does not always belong to another entity, even when they are in the same classes. In order to retrieve more query results, we provide the OPTIONAL keyword that is inserted in the one of triple patterns that is associated to entity s'. The OPTIONAL and FILTER keyword are only applicable for joining interclasses query templates (Figure 2(a) and 2(e)) since those two templates cover all entities in the same class. For FILTER keywords, we only apply it in the query that contains a literal with integer value. To find the constraint value for a FILTER expression, we choose the median value of a set of literal answers. Then, we use the greater than sign or equal to ($\geq$) in the FILTER expression. The example of query using OPTIONAL and FILTER keywords can be found in Figure 5.

The SPARQL 1.1 standard is supported in most of SPARQL endpoint servers. We can execute federated SPARQL queries by employing *SERVICE* keywords in SPARQL 1.1. Therefore, we also provide a query that contains SERVICE keyword to asses the federated engine that supports SPARQL 1.1 feature. This implies that our query can be executed in the federated engine that does not support transparent query interface.

3.7 Comparison of Splodge, Lidaq and QFed

Table 1 compares the features and capabilities of QFed, Lidaq and Splodge based on seven dimensions. QFed creates three join patterns (S-S,O-O,S-O), while Lidaq and Splodge only provide subject-object join patterns. Note that, Lidaq and Splodge also provide other patterns but these patterns are a local join pattern which does not join entities between different datasets. Splodge generates unlimited triple patterns that depends on number of sources involved. Both Splodge and QFed generate queries that only contain bound predicates. QFed does not only generate query patterns, but it also extends the query by adding SPARQL keywords which may influence the number of intermediate results. Further, QFed takes into account the type of object values (URI, Literal, Big Literal). The selectivity value is useful for choosing a predicate that will be added in queries that are generated by Splodge and QFed. Both Splodge and QFed take into account how many sources are involved in a query. As shown in that table, those query generators produce different queries. Hence, in our evaluation, we will not compare the performance of those query generators in producing queries.

4. EVALUATION

The objective of our evaluation is to show that our queries generated can return results and involve more than one data source. Furthermore, we demonstrate the effect of the big literal object value, the predicate occurrence, FILTER and OPTIONAL keywords on the performance of federated engines.

4.1 Experimental Setup

This section describes the evaluation system for running our query set on the federated engines. The code for generating the queries can be found at `https://github.com/nurainir/QFed`.

4.1.1 Dataset

Our dataset consists of four life science datasets: Daily-

	Lidaq	Splodge	QFed
Join pattern between different datasets	S-O	S-O	S-S,O-O,S-O
Maximum Triple patterns	3	unlimited	6
Predicate	Unbound and Bound predicates	Bound predicate	Bound predicate
Keywords Supported	✗	✗	Filter, Optional, Service
Object value types	✗	✗	Literal, URI
Selectivity	✗	✓	✓
Number of sources	✗	✓	✓

Table 1: Features and Capabilities of QFed, Lidaq and Splodge. S=Subject,O=Object

med[7], Drugbank[8], Diseasesome[9] and Sider[10]. These datasets are interlinked with each other using a number of predicates. Additionally there are many of the same object values that can be found from different datasets which can be used for creating federated SPARQL queries such as rdfs:label, dailymed:name and drugbank:interactionDrug1. Due to the unavailability of triples with the same subject at different datasets, we divide the Drugbank dataset into two partitions. We distributed the triples that are related to class Drug to all partitions evenly. We then add the triples that contain class drug_interactions, references and Enzim to partition one, and add the rest of the triples to partition two. In total, we have 5 SPARQL endpoints for accessing five data-sources.

4.1.2 System

We set up five Fuseki[11] engines to a Linux virtual machine for storing four datasets. We bound Fuseki to five different ports. In a separate virtual machine, we installed FedX[15] and Fuseki as the federated engines. Notice that, we do not compare the performance of Fuseki and FedX. We run queries containing SERVICE keyword on Fuseki, whereas the query without the SERVICE keyword is executed on FedX. The FedX is chosen as the federated engine since it is able to support most of the operators and keywords in the SPARQL 1.1.

4.1.3 Metrics

We assess the performance of FedX and Fuseki based on the following metrics: data transmission, and run time. The data transmission refers to the amount of data sent and received between the federated engine and SPARQL Endpoints measured in bytes size during query execution. The run time is started when the federated engine receives a query from the client and ended when it dispatches the query results to the client. The data transmission is highly related to our goal to investigate the impact of big literal object values, the distributions of the predicates, and OPTIONAL and FILTER keywords on the performance of a federated engine.

4.1.4 Query set

In total, we produced 5088 queries in 32 categories. The detail of our queries generation results can be found at our Github Wiki page (`https://github.com/nurainir/QFed/wiki`). We distinguished the categories based on the query threshold, the existence of big literal objects value, OPTIONAL and SERVICE keywords. For simplicity, the threshold value of class and properties is two. Therefore, we produces 159 queries for each category. Although we insert SERVICE and OPTIONAL, the average of predicate occurrence and number of queries stay the same. Due to the limitation of Fuseki for running a query, we also append LIMIT keywords for the queries that run on Fuseki. Each query is executed three times and limited to 10 minutes of execution.

4.2 Results and Discussion

The main goal of including the SERVICE keyword in a query is to ensure that the query cover more than one dataset. If Fuseki returns an empty result for a query with SERVICE keyword, it means that the query does not involve more than one dataset. The results of SERVICE query execution shows that only 0.07% out of the queries generated failed to return answers.

In order to distinguish each query set category, we name query sets as follows: Label C following the number represents the constraint of number entities for each class. By adding label P and following with the number, we limit the number of properties based on their frequencies. B, the last label, denotes that query set contains big literal object value. The value of C and P are two. In our evaluation, we add label O, F and B to denote query with OPTIONAL, FILTER keywords and big literal object value respectively.

As depicted in Figure 3, the replacement of the literal object value with the big literal object value lead to an increased volume data transmission between the Fuseki and SPARQL endpoints. The OPTIONAL and FILTER keywords does not contribute to increasing of the communication cost since we add LIMIT keyword for the query with SERVICE keyword. LIMIT keyword aligns the selectivity values of queries with the same ID among categories. There is not much difference between query concerning properties distribution(D) and the query that does not take account properties distribution(ND). The reason is that Fuseki does not have a source selection mechanism because the user has to define the source of each sub query beforehand. As such, the distribution of properties does not influence the communication cost.

Due to the time out issue and Java memory heap space problem, FedX failed to execute 5.68% of the queries. We eliminated all the failed query result along with all the queries that have same IDs to the failed ones. Query selectivity value gets low when OPTIONAL keyword is added to the

[7] `http://wifo5-03.informatik.uni-mannheim.de/dailymed/`
[8] `http://wifo5-03.informatik.uni-mannheim.de/drugbank/`
[9] `http://wifo5-03.informatik.uni-mannheim.de/diseasesome/`
[10] `http://wifo5-03.informatik.uni-mannheim.de/sider/`
[11] `http://jena.apache.org/documentation/serving_data/`

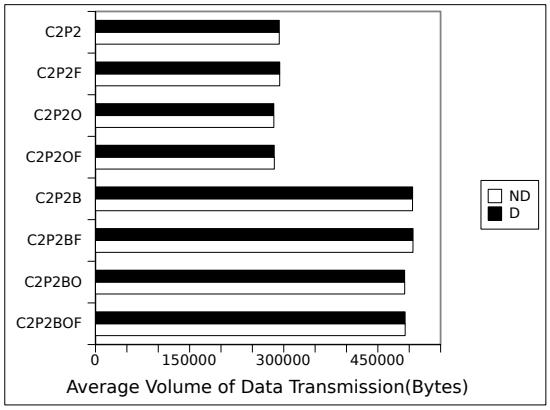


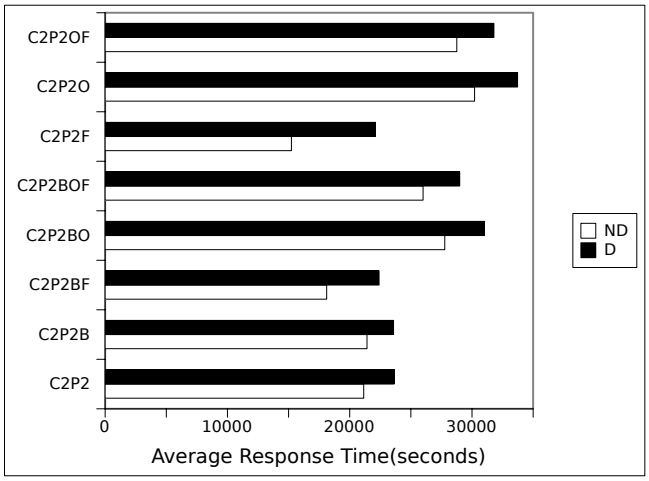
Figure 3: Data Transmision on Fuseki Execution (in bytes). D=Properties Distribution Considered, ND=Properties Distribution Not Considered

query. As a result, the response time and the volume of data transmission increase as shown in Figure 4(a) and 4(b). The using big literal object also causes an increased in volume of data transmission (Figure 4(b)), but if a query with the small literal object has a smaller selectivity value than a query with the big literal value, then in some cases, they require almost the same bandwidth consumption. If the properties distribution is considered, the FedX performance gets worse as FedX has to interrogate more SPARQL endpoints to execute a single query. The addition of FILTER keyword can cut the number of intermediate results and eventually reduces the communication cost between FedX and SPARQL endpoints.

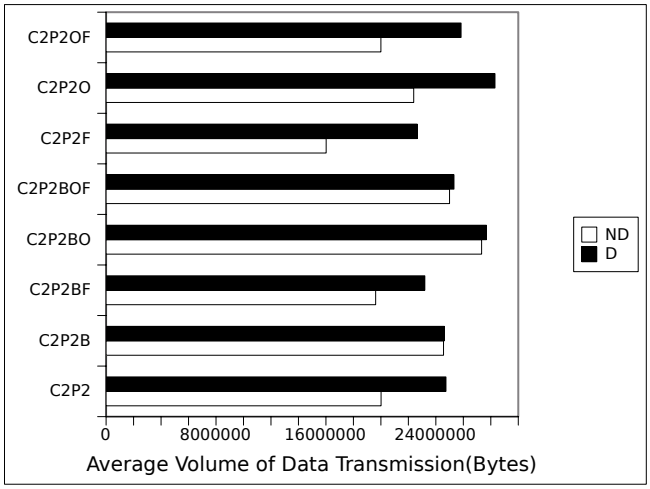
5. CONCLUSION

We have presented QFed, a tool for generating the queries to asses the performance of the federated engine. The generation of the queries considers the distribution of the predicates as the characteristic of the dataset. With respect to the cost of data communication, we add the big literal object, FILTER and OPTIONAL keyword in the query. In order to integrate data from different sources, we identify subject-object, subject-subject and object-object join patterns. The experimental result showed that big literal object has a significantly impact on Fuseki performance since Fuseki executes the query that declares SPARQL endpoints before execution. Using predicates that are distributed over the dataset, the big literal object, FILTER keyword and OPTIONAL keyword in the query influence the volume of data transmission between the FedX and the SPARQL endpoints and the FedX response time. The addition of OPTIONAL keyword contributes to the performance of FedX more significantly than the addition of a big literal value since it can reduce the selectivity value of a query.

Since the SPARQL 1.1 has supported update operation such as INSERT and DELETE, we plan to evaluate the update operation query. Furthermore, we will include more SPARQL keywords and operator. In terms of the metrics, we will take into account other metrics such as the federated engine runtime which is normally used as the performance indicator.



(a) Response Time



(b) Data Transmission

Figure 4: FedX Execution Results. D=Properties Distribution Considered, ND=Properties Distribution Not Considered

6. ACKNOWLEDGEMENT

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289.

7. REFERENCES

[1] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *ISWC*, 2011.

[2] N. Aini Rakhmawati, J. Umbrich, M. Karnstedt, A. Hasnain, and M. Hausenblas. Querying over Federated SPARQL Endpoints —A State of the Art Survey. *ArXiv e-prints*, June 2013.

[3] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world sparql queries. *CoRR*, abs/1103.5043, 2011.

[4] C. Bizer and A. Schultz. The berlin sparql benchmark. *IJSWIS*, 5(2):1–24, 2009.

[5] O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *COLD at ISWC*, 2011.

[6] O. Görlitz, M. Thimm, and S. Staab. Splodge: Systematic generation of sparql benchmark queries for linked open data. In *ISWC*, pages 116–132, 2012.

[7] G. Montoya, M.-E. Vidal, Ó. Corcho, E. Ruckhaus, and C. B. Aranda. Benchmarking federated sparql query engines: Are existing testbeds enough? In *ISWC*, pages 313–324, 2012.

[8] M. H. Nur Aini Rakhmawati, Marcel Karnstedt and S. Decker. On metrics for measuring fragmentation of federation over sparql endpoints. In *WEBIST*. SciTePress, 2014.

[9] M. Saleem, M. Kamdar, A. Iqbal, S. Sampath, H. Deus, and A.-C. Ngonga Ngomo. Big linked cancer data: Integrating linked tcga and pubmed. *Web semantics: Science, Services and Agents on the World Wide Web*, 2014.

[10] M. Saleem, M. R. Kamdar, A. Iqbal, S. Sampath, H. F. Deus, and A.-C. Ngonga. Fostering serendipity through big linked data. In *Semantic Web Challenge at ISWC2013*, 2013.

[11] M. Saleem and A.-C. N. Ngomo. Hibiscus: Hypergraph-based source selection for sparql endpoint federation. In *The Semantic Web: Trends and Challenges*, pages 176–191. Springer, 2014.

[12] M. Saleem, A.-C. N. Ngomo, J. X. Parreira, H. F. Deus, and M. Hauswirth. Daw: Duplicate-aware federated query processing over the web of data. In *The Semantic Web–ISWC 2013*, pages 574–590. Springer, 2013.

[13] M. Schmidt, O. GÃűrlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *ISWC*, volume 7031, pages 585–600. Springer, 2011.

[14] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Spˆ 2bench: a sparql performance benchmark. In *ICDE*, pages 222–233, 2009.

[15] A. Schwarte, P. Haase, K. Hoose, R. Schenkel, and M. Schmidt. Fedx: A federation layer for distributed query processing on linked open data. In *ESWC*, 2011.

[16] J. Umbrich, A. Hogan, A. Polleres, and S. Decker. Improving the recall of live linked data querying through reasoning. In *RR*, pages 188–204, 2012.

[17] X. Wang, T. Tiropanis, and H. C. Davis. Lhd: Optimising linked data query processing using parallelisation. In *LDOW at WWW*, 2013.