On Metrics for Measuring Fragmentation of Federation over SPARQL Endpoints

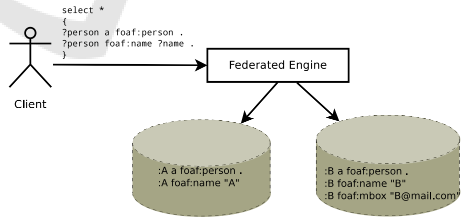
Nur Aini Rakhmawati, Marcel Karnstedt, Michael Hausenblas and Stefan Decker
INSIGHT Centre, National University of Ireland, Galway, Ireland

Keywords: Linked Data, Data Distribution, Federated SPARQL Query, SPARQL Endpoint.

Abstract: Processing a federated query in Linked Data is challenging because it needs to consider the number of sources, the source locations as well as heterogeneous system such as hardware, software and data structure and distribution. In this work, we investigate the relationship between the data distribution and the communication cost in a federated SPARQL query framework. We introduce the spreading factor as a dataset metric for computing the distribution of classes and properties throughout a set of data sources. To observe the relationship between the spreading factor and the communication cost, we generate 9 datasets by using several data fragmentation and allocation strategies. Our experimental results showed that the spreading factor is correlated with the communication cost between a federated engine and the SPARQL endpoints . In terms of partitioning strategies, partitioning triples based on the properties and classes can minimize the communication cost. However, such partitioning can also reduce the performance of SPARQL endpoint within the federation framework.

1 INTRODUCTION

Processing a federated query in the Linked Data is challenging because it needs to consider the number of the sources, the source locations and heterogeneous system such as the hardware, the software and the data structure and the distribution. A federated SPARQL query can be easily formulated by using the SERVICE keyword. Nevertheless, determining the datasource address that follows SERVICE keywords can be an obstacle in writing a query because prior knowledge data is required. To address this issue, several approaches (Rakhmawati et al., 2013) have been developed with the objective of hiding SERVICE keyword and data sources location from the user. In these approaches, the federated engines receive a query from the user, parse the query into sub queries, decide the location of each sub query and distribute the sub queries to the relevant sources. A sub query can be delivered to more than one data source if the desired answer occurs in the multiple sources. Thus, the distribution of the data can affect the federation performance (Rakhmawati and Hausenblas, 2012). As an example, consider two datasets shown in Figure 1. Each dataset contains a list of personal information using the FOAF(http://xmlns.com/foaf/spec/) vocabulary. If the user asks for the list of all person names, the federated engine must send a query to all data-



```
select *
{
?person a foaf:person .
?person foaf:name ?name .
}
```

Figure 1: Example of Federated SPARQL Query Involving Many Datasets.

sources. Consequently, the communication cost between the federated engine and data sources would be expensive.

In this study, we investigate the effect of data distribution on the federated engine performance. We propose two composite metrics to calculate the presence of classes and properties across datasets. These metrics can provide insight into the data distribution in the dataset which ultimately, it can determine the communication cost between the federated engine and SPARQL Endpoints. In order to evaluate our metrics, we use several fragmentation and allocation strategies to generate different shapes of data distribution. After that, we run a static query set over those data distributions. Our data distribution strategies could be useful for benchmarking and controlled systems such as organization system, but they can not be address the problem in the federated Linked

DOI: 10.5220/0004760101190126
In *Proceedings of the 10th International Conference on Web Information Systems and Technologies* (WEBIST-2014), pages 119-126
ISBN: 978-989-758-023-9

Open Data environment because the Linked Data publisher has the power to control the dataset generation. The existing evaluations for assessing the federation over SPARQL endpoints (Montoya et al., 2012; Schwarte et al., 2012) usually run their experiment over different datasets and different query sets. In fact, the performance of the federated engine is influenced by both dataset and query set. As a result, the performance results may vary. For benchmarking, a better comparison of federated engines performance can made with either static query sets over different datasets or static dataset with various query sets.

We only perform our observation on federation over SPARQL endpoints. Query with a SERVICE keyword is also out of the scope of our study because the query only goes to the specified source. In other words, the data distribution does not influence the performance of the federation engine in that query. Our contributions can be stated as follows: 1) We investigate the effects of data fragmentation and allocation on the communication cost of the Federated SPARQL query. 2) We introduce the spreading factor as a metric for calculating the distribution of data across a dataset. In addition, we present the relationship between the spreading factor and the communication cost of federated SPARQL queries. 3) Lastly, we create datasets for evaluating the spreading factor metric drawing from the real datasets. In particular, we provide datasets and a dataset generator that can be useful for benchmarking purpose.

2 RELATED WORKS

Primitive data metrics such as the number of triples, the number of literals are not sufficiently representative to reveal the essential characteristics of the datasets. Thus, Duan (Duan et al., 2011) introduced a structuredness notion. Since this notion is applied to a single RDF repository, it is not suitable for federated SPARQL queries which should consider the data allocation in each repository as well as the number of data sources involved in the dataset.

There are several data partitioning approaches for RDF data clustering repository such as vertical partitioning (Abadi et al., 2007) and Property Table partitioning (Huang et al., 2011). However, the communication in the RDF data clustering is totally different than the communication in the federated SPARQL query. In data clustering, several machines need to communicate with each other in order to execute a query, whereas in the federated SPARQL query, there is no interaction amongst SPARQL endpoints. The mediator has a role to communicate to each SPARQL endpoint during query execution in the federated SPARQL query. Nevertheless, we apply RDF data clustering strategies to generate the datasets for evaluation.

The existing evaluations of the federation frameworks used data partitioning in their experiment by adopting data clustering strategies. Prasser (Prasser et al., 2012) implemented three partitions: naturally-partitioned, horizontally-partitioned and randomly-partitioned. Fedbench(Schmidt et al., 2011) divided the SP2B (Schmidt et al., 2009) dataset into several partitions to run one of their evaluations. Our prior work (Rakhmawati and Hausenblas, 2012) observed the impact of data distribution on federated query execution which particularly focus on the number of sources involved, the number of links and the populated entities in several sources. In this work, we extend our previous evaluation by implementing more data partitioning schemes and we investigate the effect of the distribution of classes and properties throughout the dataset partitions on the performance of federated SPARQL query.

3 SPREADING FACTOR OF DATASET

Federated engines generally use a data catalogue to predict the most relevant sources for a sub query. The data catalogue mostly consists of a list of predicates and classes. Apart from deciding the destination of the sub queries, a data catalogue can help federated engine generate set of query execution plans. Hence, we consider computing the Spreading factor of dataset to analyse the distribution of classes and properties throughout the dataset. We initially define the dataset used in this paper as follows:

Definition 1. *Dataset D is a finite set of data sources d. In the context of federation over SPARQL endpoints, d denotes a set of triple statements t that can be accessed by a SPARQL endpoint. For each SPARQL endpoint, there exists multiple RDF graphs.*

In our work, we ignore the existence of graphs, because we are only interested in the occurrences of properties and classes in the SPARQL endpoint.

Definition 2. *Let U be the set of all URIs, B be the set of all BlankNodes, L be the set of all Literals, then a triple $t = (s, p, o) \in (U \cup B) \times U \times (U \cup L \cup B)$ where s is the subject, p is the predicate and o is the object of triple t.*

Later on, we determine the property and the class in the dataset as follows:

Definition 3. *Suppose d is a datasource in the dataset D, then the set $P_d(d,D)$ of properties p in the source d is defined as $P_d(d,D) = \{p | \exists(s,p,o) \in d \wedge d \in D\}$ and the set $P(D)$ of properties p in the dataset D is defined as $P(D) = \{p | p \in P_d(d,D) \wedge d \in D\}$*

Definition 4. *Suppose d is a datasource in the dataset D, then the set $C_d(d,D)$ of classes c in the source d is defined as $C_d(d,D) = \{c | \exists(s, rdftype, c) \in d \wedge d \in D\}$ and the set of classes c in the dataset D is defined as $C(D) = \{c | c \in C_d(d,D) \wedge d \in D\}$*

Given two datasets $D = \{d_1, d_2\}$ as shown in Figure 1. Then $P_d(d_1,D) = \{\text{rdf:type}, \text{foaf:name}\}$, $P_d(d_2,D) = P(D) = \{\text{rdf:type}, \text{foaf:name}, \text{foaf:mbox}\}$ and $C_d(d_1,D) = C_d(d_2,D) = C(D) = \{\text{foaf:person}\}$.

3.1 Spreading Factor of Dataset

With the above definitions of class, property and dataset, now we can describe how we calculate the spreading factor. The spreading factor of the dataset is based on whether or not classes and properties occur. Note that, we do not count the number of times a class and property that are found in the source d because the federated engine usually relies on the presence of property in order to predict the data location of a sub query. Given dataset D that contains a set of datasets d, the normalizing number of occurrences of properties in the Dataset D ($OCP(D)$)is calculated as follows: $OCP(D) = \frac{\sum_{\forall d \in D} |P_d(d,D)|}{|P(D)| \times |D|}$ And the normalizing number of occurrences of classes in Dataset D ($OCC(D)$) is computed as $OCC(D) = \frac{\sum_{\forall d \in D} |C_d(d,D)|}{|C(D)| \times |D|}$

$OCP(D)$ and $OCC(D)$ have a range value from zero to one. Inspired by the *F-Measure* function, we combine $OCP(D)$ and $OCC(D)$ into a single metric which is called the Spreading Factor $\Gamma(D)$ of the dataset D. $\Gamma(D) = \frac{(1+\beta^2) OCP(D) \times OCC(D)}{\beta^2 \times OCP(D) + OCC(D)}$ where $\beta = 0.5$

We assign $\beta = 0.5$ in order to put more stress on properties than classes. The intuition is that the highest number of the query pattern delivered to SPARQL endpoint mostly contains constant predicates (Arias et al., 2011). Moreover, the number of distinct properties in the dataset is usually higher than the number of distinct classes in the dataset. The high Γ value indicates that the class and properties are spread out over the dataset.

Look back at our previous example in which we define $P_d(d_1,D)$, $P_d(d_2,D)$, $P(D)$, $C(D)$, $C_d(d_1,D)$,$C_d(d_2,D)$, then we can calculate $OCP(D) = \frac{2+3}{3X2} = 0.833$ and $OCC(D) = \frac{1+1}{1X2} = 1$. Finally, we obtain $\Gamma(D) = 1.172$

3.2 Spreading Factor of Dataset Associated with the Queryset

The spreading factor of a dataset reveals how the whole of classes and properties are distributed over the dataset. However, a query only consists of partial properties and classes in the dataset. Thus, it is necessary to quantify the spreading factor of the dataset with respect to the queryset.

Definition 5. *A query consists of set of triple patterns τ which is formally defined as $\tau(s,p,o) \in (U \cup V) \times (U \cup V) \times (U \cup L \cup V)$ where V is a set of all variables.*

Given a queryset $Q = \{q_1, q_2, \cdots, q_n\}$, the Q-spreading factor γ of dataset D associated with queryset Q is computed as $\gamma(Q,D) = \sum_{\forall q \in Q} \frac{\sum_{\forall \tau \in q} OC(\tau,D)}{|Q|}$ where the occurrences of class and property for τ is specified as

$$OC(\tau,D) = \begin{cases} \frac{ofD(o_\tau,D)}{|D|} & \text{if } p_\tau \text{ is rdf:type} \wedge o_\tau \notin V \\ \frac{pfD(p_\tau,D)}{|D|} & \text{if } p_\tau \text{ is not rdf:type} \wedge p_\tau \notin V \\ \frac{\sum_{\forall d \in D} |P_d(d,D)|}{|D|} & \text{otherwise} \end{cases}$$

$ofD(o,D)$ denotes the occurrences of object o in the dataset D and $pfD(p,D)$ denotes the occurrences of predicate p in the dataset D which can be calculated as follows: $ofD(o,D) = \sum_{\forall d \in D} ofd(o,d,D)$ The occurrences of object o in the source d can be explained as follows:

$$ofd(o,d,D) = \begin{cases} 1 & \text{if } o \in C_d(d,D) \\ 0 & \text{otherwise} \end{cases}$$

$pfD(p,D) = \sum_{\forall d \in D} pfd(p,d,D)$ The occurrence of predicate p in the source d can be obtained from the following formula:

$$pfd(p,d,D) = \begin{cases} 1 & \text{if } p \in P_d(d,D) \\ 0 & \text{otherwise} \end{cases}$$

Consider an example, given a query and a dataset as shown in Figure 1, then $OC(\text{?person a foaf:person}, D) = 1$ and $OC(\text{?person foaf:name ?name}, D) = 1$ because `foaf:person` and `foaf:name` are located in two data sources. As a result, the q-Spreading factor $\gamma(Q,D)$ is $\frac{1+1}{1} = 2$

4 EVALUATION

We ran our evaluation on an Intel Xeon CPU X5650, 2.67GHz server with Ubuntu Linux 64-bit installed as

Listing 1: Dailymed Sample Triples.

```
dailymeddrug:82 a dailymed:drug
dailymeddrug:82 dailymed:activeingredient dailymeding:
        Phenytoin
dailymeddrug:82 rdfs:label "Dilantin-125␣(Suspension)"

dailymeddrug:201 a dailymed:drug
dailymeddrug:201 dailymed:activeingredient dailymeding:
        Ethosuximide
dailymeddrug:201 rdfs:label "Zarontin␣(Capsule)"

dailymedorg:Parke-Davis a dailymed:organization
dailymedorg:Parke-Davis rdfs:label "Parke-Davis"
dailymedorg:Parke-Davis dailymed:producesDrug
        dailymeddrug:82
dailymedorg:Parke-Davis dailymed:producesDrug
        dailymeddrug:201

dailymeding:Phenytoin a dailymed:ingredients
dailymeding:Phenytoin rdfs:label "Phenytoin"

dailymeding:Ethosuximide a dailymed:ingredients
dailymeding:Ethosuximide rdfs:label "Ethosuximide"
```

the Operating System and Fuseki 1.0 as the SPARQL Endpoint server. For each dataset, we set up Fuseki on different ports. We re-used the query set from our previous work (Rakhmawati and Hausenblas, 2012). We limited the query processing duration to one hour. Each query was executed three times on two federation engines, namely SPLENDID (Görlitz and Staab, 2011) and DARQ (Quilitz and Leser, 2008). These engines were chosen because SPLENDID employs VoID(http://www.w3.org/TR/void/) as data catalogue that contains a list of predicates and entities, while DARQ has a list of predicates which is stored in the Service Description(http://www.w3.org/TR/sparql11-service-description/). Apart from using VoID, SPLENDID also sends a SPARQL ASK query to determine whether or not the source can potentially return the answer. We explain the details of our dataset generation and metrics as follows:

4.1 Data Distribution

To determine the correlation between the communication cost of the federated SPARQL query and the data distribution, we generate 9 datasets by dividing the Dailymed(http://wifo5-03.informatik.uni-mannheim.de/dailymed/) into three partitions based on following strategies:

4.1.1 Graph Partition

Inspired by data clustering for a single RDF storage (Huang et al., 2011), we performed graph partition over our dataset by using METIS (Karypis and Kumar, 1998). The aim of this partition scheme is to reduce the communication needed between machines during the query execution process by storing the connected components of the graph in the same machine. We initially identify the connections of subject and object in different triples. We only consider the URI object which is also a subject in other triples. Intuitively, the reason is that the object which appears as the subject in other triples can create a connection if the triples are located in different dataset partitions. $V(D)$ denotes the set of pairs of subject and object that are connected in the dataset D which can be formally specified as $V(D) = \{(s,o) | \exists s,o,p,p' \in U : (s,p,o) \in D \wedge (o,p',o') \in D'\}$. We assign a numeric identifier for each $s,o \in V(D)$. After that, we create a list of sequential adjacent vertexes for each vertex then uses it as input of METIS API. Run METIS to divide the vertexes and get a list of the partition number of vertexes as output. Finally, we distribute each triple based on the partition number of its subject and object. Consider an example, given Listing 1 as a dataset sample, then

$$V(D)=\{(\text{dailymeddrug:82}, \text{dailymeding:Phenytoin}),(\text{dailymeddrug:201}, \text{dailymeding:Ethosuximide}),(\text{dailymedorg:Parke-Davis}, \text{dailymeddrug:82}),(\text{dailymedorg:Parke-Davis}, \text{dailymeddrug:201})\}$$

Starting an identifier value from one and increment the identifier later, we set the identifier for dailymeddrug:82 = 1, dailymeding:Phenytoin =2, dailymeddrug:201=3, dailymeding:Ethosuximide=4 and dailymedorg:Parke-Davis=5. After that, we can create list of sequential adjacent vertexes $V(D)$ is $\{(2,5),1,(4,5),3,(1,3)\}$. Suppose that we divide the sample of dataset into 2 partitions, then the output of METIS partition is $\{1,1,2,2,1\}$ where each value is the partition number for each vertex. According to the METIS output, we can say that dailymeddrug:82 belongs to partition 1, dailymeding:Phenytoin belongs to partition 1, dailymeddrug:201 belongs to partition 2 and so on. In the end, we have two following partitions:

Partition 1: all triples that contain dailymeddrug:82, dailymeding:Phenytoin and dailymedorg:Parke-Davis

Partition 2: all triples that contain dailymeddrug:201 and dailymeding:Ethosuximide

4.1.2 Entity Partition

The goal of this partition is to distribute the number of entities evenly in each partition. Different classes can be located in a single partition. However, the entities of the same class should be grouped in the same partition until the number of entities reaches the maximum number of entities for each source. We initially create a list of the subjects along with its class ($E(D)$). The set $E(D)$ of pairs of subject and its class in the dataset D is defined as $E(D) = \{(s,o) | \exists (s, rdftype, o) \in D\}$ Then, we sort $E(D)$ by its class o and store each pair of the subject and object in a partition until the number of pairs of subject and object equals to the total pairs of subject and object divided by the number of partitions. After that, we distribute the remainders of triples in the dataset based on the subject location. Given Listing 1 as a dataset sample, then

$$E(D)=\{(\text{dailymeddrug:82},\text{dailymed:drug}),(\text{dailymeddrug:201},\text{dailymed:drug}),(\text{dailymedorg:Parke-Davis},\text{dailymed:organization}),(\text{dailymeding:Phenytoin},\text{dailymed:ingredients}),(\text{dailymeding:Ethosuximide},\text{dailymed:ingredients})\}$$

Suppose that we split the dataset into two partitions, then the maximum number of entities for each partition is $\frac{|E(D)|}{numberofpartitions} = \frac{5}{2} = 3$ (ceiling 2.5). We place dailymeddrug:82, dailymeddrug:201 and dailymedorg:Parke-Davis in the partition 1 and store the remainders of entities in the partition 2. As the final step, we distribute the related triples based on its subject partition number.

4.1.3 Class Partition

Class Partition divides the dataset based on its classes. The related triples that belong to one entity are placed in the same machine. To begin with, we also create $E(D)$ which was used in Entity partition. Later, we distribute each triple based on the subject class. ike our previous entity partition example, we do the same step to generate $E(D)$. However, in the class partition, we divide the dataset to three partitions since we have three classes (dailymed:drug, dailymed:organization, dailymed:ingredients).

4.1.4 Property Partition

Wilkinson(Wilkinson, 2006) introduced a method for storing RDF data in traditional databases known as Property Table (PT). There are two types of PT partitions: Clustered Property Table and Property-class Table. In our property partition, we do not have a Property class table because we treat all properties in the same manner. We place the triples that have the same property in one data source. Because the number of properties in the dataset is generally high, we allow more than one property to be stored in the same partition as long as we get a balanced number of triples among the partitions. Firstly, we group the triples based on its property. Next, we store each group in a partition until the number of partition triples is less than or equal to the number of dataset triples divided by the number of partitions. For instance, given a dataset as shown in Listing 1, then we have four properties: rdf:type, dailymed:activeingredient, rdf:label and dailymed:producesDrug. Suppose that we want to divide the dataset into 2 partitions, then the maximum number of triples in each partition is $\frac{thenumberoftriples}{thenumberofpartitions} = \frac{14}{2} = 7$. As the following step, we store the triples based on its property as follows: *Partition 1*: five triples with rdf:type property, two triples with dailymed:activeingredient property and *Partition 2*: five triples with rdfs:label property, two triples with dailymed:producesDrug

4.1.5 Triples Partition

The federation framework performance is influenced not only by the federated engine solely, but also depends on the SPARQL Endpoints within the federation framework. In order to keep balanced workload for SPARQL Endpoints, we split up the triples of each source evenly because LUBM (Guo et al., 2005) mentioned that the number of triples can influence the performance of a RDF repository. We created three triple partition datasets (TD, $TD2$, $TD3$). TD is obtained by partitioning the native Dailymed dataset into three parts. $TD2$ and $TD3$ are generated by picking a random starting point within the Dailymed dump file(by picking a random line number).

4.1.6 Hybrid Partition

The Hybrid Partition is a partitioning method that combines two or more previous partition strategies. For instance, if the number of triples in a class is too high, we can distribute the triples to another partition to equalize the number of triples. Since the number of triples in each dataset of the Class Distribution CD are not equal, we create HD to distribute the triples evenly. However, rdf:type property and rdfs:label property are evenly through all partitions in dataset $HD2$. This distribution is intended for balancing the workload amongst SPARQL Endpoints since those properties are commonly used in our query set.

As shown in those figures, the classes and properties are distributed over most of the partitions in the GD dataset. The PD has the lowest Spreading Factor among the dataset because each property occurs in exactly one partition and only in one partition has a set of triples that contains rdf:type. The dataset generation code and the generation results can be found at DFedQ github(https://github.com/nurainir/DFedQ).

4.2 Metrics

To calculate the communication cost of the the federated SPARQL query, we compute the data transfer

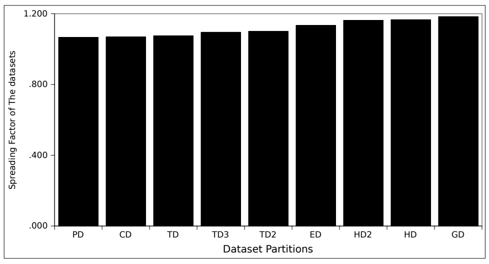


Figure 2: Spreading Factor of Dataset.

volume between the federated engine and SPARQL Endpoints. The data transfer volume includes the amount of data both sent and received by the mediator. Apart from capturing the data transmission, we also measure the requests workload (*RW*) during query execution. *RW* is calculated as $RW = \frac{RQ}{T*SS}$ where *RQ* refers to the number of requests sent by the federated engine to all SPARQL Endpoints, *T* denotes the duration between when a query is received by the federated engine and when its results starts to be dispatched to the client and *SS* is the number of selected sources. Furthermore, we also measure the response time that is required by a federated engine to execute a query.

For the sake of readability, we aggregate each performance metric results into a single value. In order to avoid trade-offs among queries, we assign a weight to each query using the the variable counting strategy from the ARQ Jena (Stocker and Seaborne, 2007). This weight indicate the complexity of the query based on the selectivity of the variable position and the impact of variables on the source selection process. The complexity of query can influence the federation performance. Hence, we normalize each performance metric result by dividing the metric value with the weight of the associated query. In the context of federated SPARQL queries, we set the weight of the predicate variable equals to the weight of the subject variable since most of the federated engines rely on a list of predicates to decide the data location. Note that, a triple pattern can contain more than one variable. The details of the weight of subject variable w_s, predicate variable w_p and object variable w_o for the triple pattern τ can be explained as follows:

$$w_s(\tau)=\begin{cases} 3 & \text{if the subject of triple pattern } \tau \in V \\ 0 & \text{otherwise} \end{cases}$$

$$w_p(\tau)=\begin{cases} 3 & \text{if the predicate of triple pattern } \tau \in V \\ 0 & \text{otherwise} \end{cases}$$

$$w_o(\tau)=\begin{cases} 1 & \text{if the object of triple pattern } \tau \in V \\ 0 & \text{otherwise} \end{cases}$$

Finally, we can compute the weight of query q: $weight(q) = \sum_{\forall \tau \in q} \frac{w_s(\tau)+w_p(\tau)+w_o(\tau)+1}{MAX\_COST}$ where $MAX\_COST = 8$ because if a triple pattern consists of variables that are located in all positions, the weight of the triple pattern is 8(3+3+1+1). By using the weight of a query, we can align the query performance results afterwards. We do not create a composite metric that combines the response time, the request workload and the data transfer, but rather we calculate each performance metric results individually. Given that Q is a set of queries q in the evaluation and that m is a set of performance metric results associated with the queryset Q, then the final metric μ for the evaluation is $\mu(Q,m) = \frac{\sum_{\forall q \in Q} \frac{mq}{weight(q)}}{|Q|}$

For instances, the query in Figure 1 has a weight = $\frac{3+1}{8} + \frac{3+1+1}{8} = 1.125$. Suppose that the volume of data transmission during this query execution is 10 Mb and we only have one query in the queryset, then $\mu(Q,m)$ can be calculated $\frac{\frac{10}{1.125}}{1} = 8.88$Mb.

5 RESULTS AND DISCUSSION

As seen in Figures 3 and 4, the data transmission between DARQ and SPARQL Endpoints is higher than the data transmission between SPLENDID and SPARQL Endpoints. However, Figures 5 and 6 show that the average requests workload in DARQ is less than the average requests workload in SPLENDID.

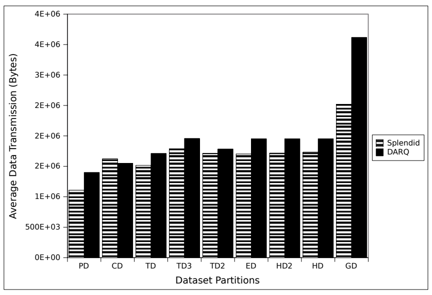


Figure 3: Average Data Transfer Volume Vs the Spreading Factor of Datasets (order by the Spreading Factor value).

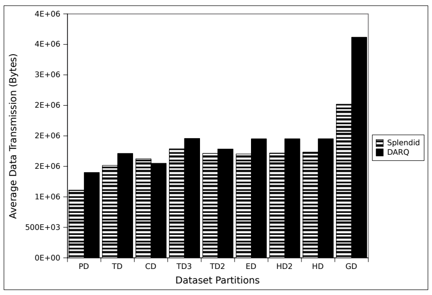


Figure 4: Average Data Transfer Volume Vs the Q-Spreading Factor of Datasets associated with the Queryset(order by the Spreading Factor value).

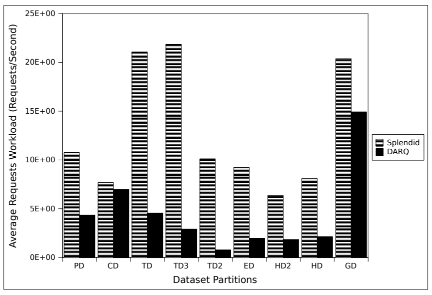


Figure 5: Average Requests Workload Vs the Spreading Factor of Datasets(order by the Spreading Factor value).

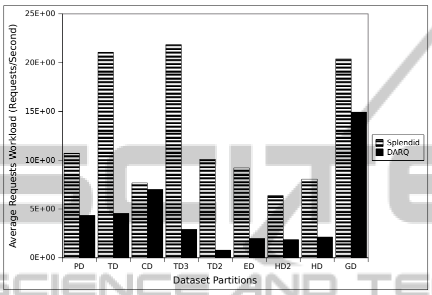


Figure 6: Average Requests Workload Vs the Q-Spreading Factor of Datasets associated with the Queryset(order by the Spreading Factor value).

This is because DARQ never sends SPARQL ASK queries in order to predict the most relevant source for each sub query.

Overall, data transmission increases gradually in line with the Spreading Factor of a dataset. However, the data transmission rises dramatically for *GD* distribution. This indicates that in the context of Federated SPARQL queries, data clustering based on its property and class is better than data clustering based on related entities such as Graph Partition. The reason behind this conclusion is that the source selection in federated query engine depends on classes and properties occurrences. Furthermore, when the federated engines generate query plans, they use optimization techniques based on the statistical predicates and classes.

Although a small Spreading Factor can minimize the communication cost, it can also reduce the SPARQL Endpoint performance. As shown in Figure 5 and 6, a small Spreading Factor can lead to the high number of requests received by SPARQL Endpoint in one second because in the property distribution, the federated engine mostly sends different query patterns to multiple datasource. Moreover, the SPARQL endpoint that stores the popular predicates such as rdf:type and rdfs:label will receive more requests than other SPARQL endpoints. Consequently, this such condition can lead to incomplete results because when overloaded, the SPARQL Endpoint might reject requests (e.g Sindice SPARQL endpoint(http://sindice.com/) only allows one client sending one query per second). Poor performance is also shown at the highest value of Spreading Factor of the dataset (GD) because the entities are spread over the dataset partitions. Hence, with the calculation of the spreading factor of the dataset, the federated engine can create a query optimization which attempts to adapt the dataset characteristic that is shown from the spreading factor value. For instance, if the dataset has too small Spreading Factor, the federated engine should maintain a timer to send several requests to the same SPARQL endpoint in order to keep the sustainability of the SPARQL endpoint as well as avoid the incomplete answer.

6 CONCLUSION

We have implemented various data distribution strategies to partition classes and properties over dataset partitions. We introduced two notions of dataset metrics, namely the Spreading Factor of a dataset and the Spreading Factor of a Dataset associated with the query set. These metrics expose the distribution of classes and properties over the dataset partitions. Our experiment results revealed that the class and property distribution effects on the communication cost between the federated engine and SPARQL endpoints. However, it does not significantly influence the request workload of a SPARQL endpoint. Partitioning triples based on the properties and classes can minimize the communication cost. However, such partitioning can also reduce the performance of SPARQL endpoints within the federation infrastructure. Further, it can also influence the overall performance of federation framework.

In future work, we will apply other dataset partitioning strategies and use more federated query engines which have different characteristics from DARQ and SPLENDID.

ACKNOWLEDGEMENTS

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289 and Indonesian Directorate General of Higher Education. Thanks to Soheila for a great discussion

REFERENCES

Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2007). Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd VLDB*, VLDB '07, pages 411–422. VLDB Endowment.

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

Duan, S., Kementsietsidis, A., Srinivas, K., and Udrea, O. (2011). Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *ACM SIGMOD*.

Görlitz, O. and Staab, S. (2011). SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the 2nd International Workshop on COLD*, Bonn, Germany.

Guo, Y., Pan, Z., and Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158 – 182.

Huang, J., Abadi, D. J., and Ren, K. (2011). Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134.

Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392.

Montoya, G., Vidal, M.-E., Corcho, Ó., Ruckhaus, E., and Aranda, C. B. (2012). Benchmarking federated sparql query engines: Are existing testbeds enough? In *ISWC(2)*, pages 313–324.

Prasser, F., Kemper, A., and Kuhn, K. A. (2012). Efficient distributed query processing for autonomous rdf databases. EDBT '12, pages 372–383, New York, NY, USA. ACM.

Quilitz, B. and Leser, U. (2008). Querying distributed rdf data sources with sparql. ESWC'08, pages 524–538, Berlin, Heidelberg. Springer-Verlag.

Rakhmawati, N. A. and Hausenblas, M. (2012). On the impact of data distribution in federated sparql queries. In *ICSC 2012*, pages 255 –260.

Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A., and Hausenblas, M. (2013). Querying over federated sparql endpoints - a state of the art survey. *CoRR*, abs/1306.1723.

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

Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. (2009). Spˆ2bench: a sparql performance benchmark. In *ICDE'09.*, pages 222–233. IEEE.

Schwarte, A., Haase, P., Schmidt, M., Hose, K., and Schenkel, R. (2012). An experience report of large scale federations. *CoRR*, abs/1210.5403.

Stocker, M. and Seaborne, A. (2007). Arqo: The architecture for an arq static query optimizer.

Wilkinson, K. (2006). Jena property table implementation. In *In SSWS*.