IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 16, 2019, accepted July 3, 2019, date of publication July 22, 2019, date of current version August 9, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2930416

QPPDs: Querying Property Paths Over Distributed RDF Datasets

QAISER MEHMOOD[1], MUHAMMAD SALEEM[2], RATNESH SAHAY[1], AXEL-CYRILLE NGONGA NGOMO[3], AND MATHIEU D'AQUIN[1]

[1]Insight Centre for Data Analytics, Data Science Institute, NUI Galway, Galway, Ireland
[2]Leipzig University, Leipzig, Germany
[3]Department of Computer Science, University of Paderborn (UPD), Paderborn, Germany

Corresponding author: Qaiser Mehmood (qaiser.mehmood@insight-centre.org)

This work was supported in part by the Science Foundation Ireland (SFI) under Grant SFI/12/RC/2289, in part by the European Regional Development Fund, and in part by the Project HOBBIT, Germany, under Grant 688227.

ABSTRACT A key property of linked data, i.e., the web-based representation and publication of data as interconnected labeled graphs, is that it enables querying and navigating through datasets distributed across the network. SPARQL1.1, the current standard query language for RDF-based linked data, defines a construct—called property paths (PP)—to navigate between the entities of a graph. This is potentially very useful in a number of use cases, e.g., in the biomedical domain, where large datasets are available as linked data graphs. However, the use of PP in SPARQL 1.1. is possible only on a single local graph, requiring us to merge all distributed datasets into one large, centrally stored graph, therefore reducing the value of using linked data in the first place. We propose an index-based approach—called QPPDs—for answering queries for paths distributed across multiple, distributed datasets. We provide a heuristic-based source selection mechanism to select the relevant datasets (also called data sources) for a given path query, and a technique that federates queries to selected sources, and assembles (merges) the paths (i.e., partial or complete) retrieved from those remote datasets. We demonstrate our approach on a genomics use-case, where the description of biological entities (e.g., genes, diseases, and drugs) is scattered across multiple datasets. In our preliminary investigation, we evaluate the QPPDs approach with real-world path queries—on biological data that are very heterogeneous in nature—in terms of performance (overall path retrieval time) and result completeness, i.e., the number of paths retrieved.

INDEX TERMS Linked data, SPARQL, distributed querying, federated querying, path retrieval, graph traversal.

I. INTRODUCTION

The potential benefits of using Linked Data (also known as the Web of Data or Semantic Web data), have been increasingly considered in a variety of domains where rich, multi-source data need to be explored, e.g., bioinformatics, geography, literature, etc. In those domains, several, often large datasets have been made available using RDF to represent them as labelled directed graphs on the Web. Support for querying and exploring those datasets has also been evolving. SPARQL is, for example, the standard query language for RDF graph data, and the recent SPARQL 1.1. specification introduced new navigational features (see example in Listing 1), where users can check the existence of paths between two entities (i.e., nodes).

```
Prefix : <urn:exe:>
SELECT ?s ?p ?o
{ :cnv (:|!:)* ?s. ?s ?p ?o. ?o
    (:|!:)*  :gene }
Group By ?s ?p ?o
```

Listing 1. Property Path query in SPARQL1.1.

There are however a number of limitations in those specifications, and their implementation in various reference platforms. For instance, the above query can only be used to check the existence of paths, however, it will not enumerate paths. To overcome this limitation, in our previous work [1], we proposed an extension of SPARQL which allows finding, rendering and enumerating the top-k shortest paths. Due to its importance as a feature in a number of use cases, including

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

in the biomedical domain, other earlier approaches [2]–[5] also tried to address the issue of querying paths. In 2016, the European Semantic Web Conference (ESWC) hosted a challenge[1] to find the "Top-K Shortest Paths in Large Typed RDF Graphs". Recently, the Stardog tool[2] also started supporting path retrieval via SPARQL querying. However, all the previous approaches are only able to work on a *single graph*. None of them were aimed at finding paths that span across several *distributed* graphs, which seems an obvious requirement to enable the full benefit of the use of Linked Data. Indeed, in the biomedical domain for example, a lot of data is available publicly from multiple, heterogeneous sources. In such a case, it is very common for two biological entities (e.g., gene, protein, drug, pathway, etc.) to be related through paths formed of links going across several of those datasets. To query such paths neither SPARQL 1.1, even if it supports some form of federated querying, nor the aforementioned techniques would be effective.

Hence, to find paths between two entities, the centralized approaches adopted by current systems pose some challenges such as: (i) querying multiple datasets requires the user to first merge them into a single graph, which is a cumbersome task; (ii) copied data need to be synchronized; and therefore (iii) merged data might not be as up-to-date and fresh as in the original source; (iv) data is not always under control or fully accessible by the person querying it, and finally (v) scalability is a major issue in the centralized approaches. We propose an approach called QPPDs that can efficiently federate path queries and return a list of paths that connect entities (i.e., a *source* entity and a *target* entity) across distributed, interconnected graphs. The QPPDs adopts an index-based, top-down approach and federates navigational queries across multiple datasets which are hosted as SPARQL endpoints. To find distributed paths between given source and target resources, the QPPDs approach works in four steps: (i) first it selects all datasets which contain the source and target resources; (ii) in the second step it retrieves all paths from each source *dataset* to each target *dataset*; (iii) in the third step it retrieves all distributed paths from the source *resource* to the target *resource*; (iv) in the final step it merges the paths.

The rest of the paper is organized as follow: We start by presenting two motivating scenarios (i.e., a running example, and a real-world use case) that relate to retrieving paths between source and target nodes of a query, hosted in different triple stores. We then introduce preliminary notions and notations and discuss related work on traversing and retrieving paths in relational and graph databases. We show the details of the QPPDs approach and provide an evaluation of its performance and completeness, discussing the insight obtained from the results. Finally, we present our conclusions and various routes to optimize the navigation across federated RDF graphs.

[1] http://2016.eswc-conferences.org/top-k-shortest-path-large-typed-rdf-graphs-challenge.html

[2] https://www.stardog.com/docs/#_path_queries

II. MOTIVATING SCENARIO

In this section, we present two motivating scenarios: (1) a real-world scenario showing the use of distributed property paths in RDF datasets for Cancer Genomics; and (2) a toy scenario which is used as a running example to explain the proposed approach.

A. CANCER GENOMICS

In the bio-informatics domain, for a practitioner to understand cancer progression, several genetic features (e.g. diseases, medical history, etc.) are often studied together. Therefore, one of the key challenges in cancer genomics –a cornerstone of precision medicine– is to discover gene-disease-drug associations, i.e. understanding which gene is effected by what disease and how this disease can be treated with what drug. At data level, such associations provide insight into the drug development process –tailored specifically for an individual patient (or a group of patients)– targeting prevention, diagnosis and treatment of the diseases [6].

For instance, consider a scenario where a biomedical expert is trying to discover the paths between a gene `rs769022521` and the associated disease `HP_0000024`, and (s)he only has access to a single local dataset (e.g., Variant, see Figure 1). Even if those two entities are present in that dataset, existing paths between them might not be, in which case, the expert would not be able to find the association between `rs769022521` and `HP_0000024`, unless (s)he does have access and knowledge about other datasets. Even if that was the case, however, without an approach to identify paths across those datasets, the expert would have a hard time finding paths without first integrating all datasets into one.

Figure 1 shows how a path between `rs769022521` and `HP_0000024` can be obtained where each of the dataset i.e., (Variant, Gene, Protein, PantherClass, Phenotype, and Disease) hosted at different SPARQL endpoints contributes to this particular path.

Based on the above scenario we believe that a technology –implemented here through the QPPDs approach– that can enable querying paths/associations among two or more biological entities across distributed datasets/endpoints would be of great help to biologist and practitioners working in cancer genomics as well as in the larger healthcare and life sciences area. This is further demonstrated in the evaluation section of this paper.

B. RUNNING EXAMPLE

In this scenario, we present a toy/fictional example, used in the rest of the paper to illustrate the approach. It uses three synthetic RDF datasets given in Figure 2. The goal of presenting this use case is to explain the proposed approach by using a simple and easy to follow example. Suppose that we want to find all the paths from resource F of Dataset $D1$ to resource E which is present in dataset $D2$. A simple SPARQL property path query on each dataset individually will not be able to retrieve any path. This is because both source F and

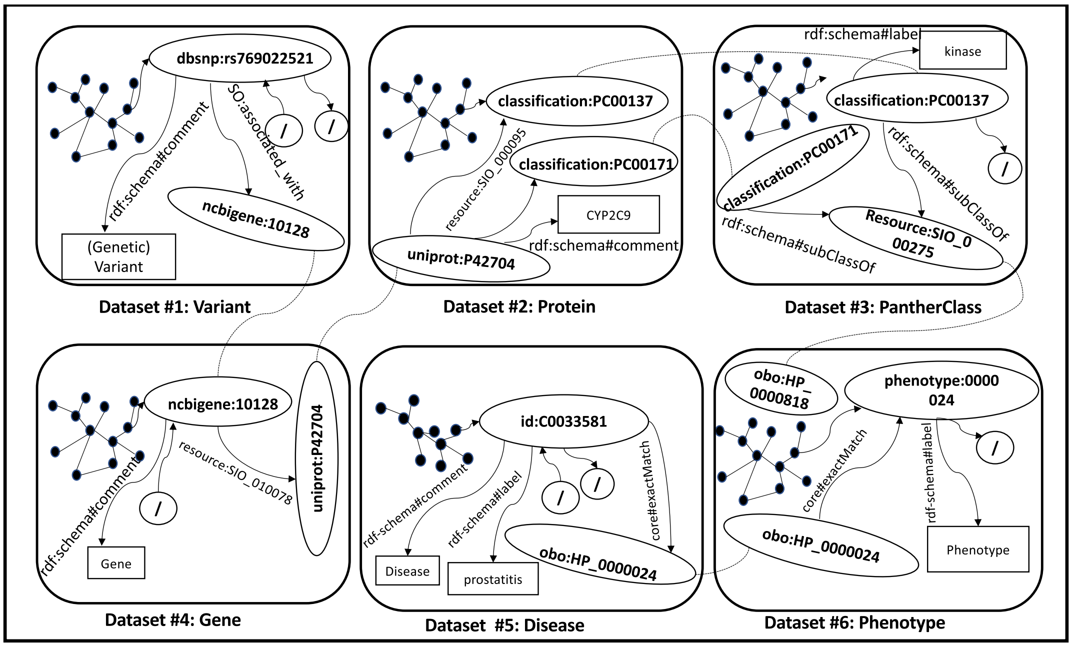
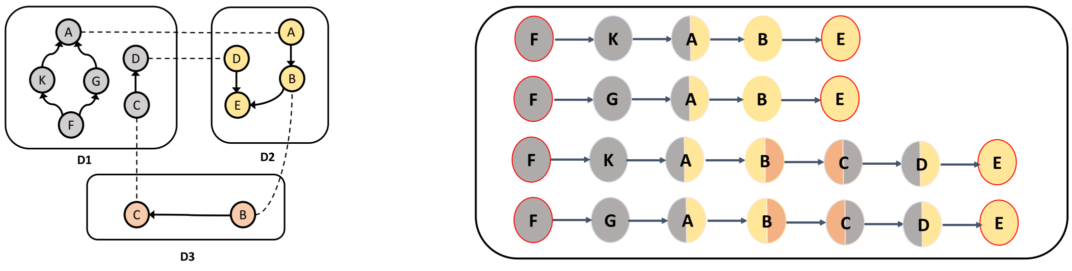


FIGURE 1. A real-world example of distributed paths between RDF datasets.



(a) Sample datasets and their connectivity.

(b) Federated Paths.

FIGURE 2. Running example: Datasets and paths between node F and node E.

target E nodes are not present within the same dataset in any of the datasets. However, we can obtain partial paths from different datasets which can later be assembled or re-arranged to form complete paths between the two distributed resources as given in Figure 2b.

III. PRELIMINARIES

Consider a network of distributed datasets $\mathcal{G}$ where each dataset $G = (V, P)$ is a directed graph. V is the set of vertices and P is the set of edges. The vertices (v_i, v_j) are associated through a set of edges $E = \{e_{in}, \ldots, e_{n'j}\}$. If there exists an edge $e_{ij} = (v_i, v_j)$ then v_j is a *successor* of v_i and v_i is a predecessor of v_j. For RDF graphs, we define these notations as follows:

Definition 1 (Path): A ***Path*** within a graph $G = (V, P)$ is a sequence of nodes and relations $n_1 \xrightarrow{t_1} n_2 \xrightarrow{t_2} \ldots, \xrightarrow{t_{k-1}} nk$ such that, there exists a property (edge) e of type $t_k{-}1$ between two consecutive nodes in a sequence $n_k{-}1, n_k$.

In a distributed environment, if a path between source and target entities does not exist within a local dataset then remote datasets are scanned and queried to find the paths connecting those entities across graphs.

Definition 2 (RDF Triple & Graph): The set of RDF terminologies consists of the set of IRIs I, the set of blank-nodes B and the set of literals L. An RDF Triple $T := (s; p; o)$ is an element of the set $G := (\text{I}\cup\text{B}) \times \text{I} \times (\text{I}\cup\text{L}\cup\text{B})$. The set G is a finite set of triples called RDF graph. An RDF graph G_i of a single triple T_i is represented by an edge or property p_i through which a vertex (subject) s_i is associated with another vertex (object) o_i.

Definition 3 (Relevant Path Sources): Given a path query Q (see Listing 6) corresponding to finding a path between a subject s_i and object o_i where the set of relevant sources for T in $\mathcal{G}$ is the set $\mathcal{D}_T \subseteq \mathcal{G}$ of data sources that can provide (partial) paths when queried with T. We use the notation $\mathcal{D}_T$ to denote the set of relevant data sources for source (subject) and target (object) nodes and use $\mathcal{D}$

when the context does not require specifying the query patterns.

Definition 4 (Path Reachability): For a labeled directed RDF graph G, the reachability relation is the transitive closure of RDF properties *p(s,o)* such that i.e., for the set of all ordered pairs (s, o) there exists a sequence of subjects and objects $v_0 = s, v_1, v_2, \ldots, v_k = o$ where the property $p(v_{i-1}, v_i)$ is in *p(s,o)* for all $1 \leq i \leq k$.

For a given query Q to find the path between s_i and o_i, if these two are connected through any number of paths $p_i, \ldots, p_n$ we say that paths exist and source (s_i) and target (o_i) are reachable. While if there is no path between s_i and o_i, it means that (s_i, o_i) are not reachable.

IV. RELATED WORK

As linked data is growing, different query federation approaches have been introduced to query such distributed data sources. We categorize the related work based on the general underlying technology, as well as the relation to navigational queries and the identification of paths between entities.

A. DISTRIBUTED DATABASES

Some relational database systems such as [7] provide the facility to store graph data in distributed relational tables. In contrast to the graph databases, the relational databases are not designed to provide public endpoints or access to the data and therefore most of them are used to process graph queries within the enterprise context. There are non-relational distributed systems, e.g., [8], [9], which manage distributed data via different data structures and support simple navigational queries (e.g., shortest path, neighborhood, degree, etc). Distributed graph databases, such as Stardog,[3] Neo4j,[4] Blazegraph,[5] are optimized for graph navigational features, but those are not implemented for navigation across the distributed datasets. Trinity[6] and HyperGraphDB[7] are two systems that include distributed query based approaches.

B. DISTRIBUTED PATH FINDING ALGORITHMS

Google in 2014, introduced a framework named Pregel [10] where distributed querying is possible based on message passing. The basic idea of Pregel was to partition a graph and distribute it on different servers. It works similarly to a master-slave approach, where the master assigns tasks to different slave machines. These slave machines communicate with each other through message passing.

Comparing Pregel to our approach, we can identify a number of differences: (i) Pregel does not work like a federated systems but like master-slave and is file based, i.e. data is partitioned and distributed in a closely controlled way, while the approach we adopted is purely distributed where data can be served by different providers, exposing different interfaces supporting path queries, (ii) our system is specific to RDF-based distributed graphs and SPARQL path queries. Our system is index-based, meaning that message passing is done only between already known connected datasets. In [11] they proposed a distributed system for dynamic road network.

The previous two approaches are based on the shared algorithms disseminated across the network. Also, to query the paths, these systems require to have get access to data, partition it and restructure the data according to their implementations. However, this cannot be achieved using data from the linked open data cloud (LOD). Our approach relies on whatever pathfinding algorithm (e.g., BFS, DFS, A*, Bidirectional) is available in the underlying implementation of the triple stores behind a dataset's endpoint. Our approach therefore gets results from the endpoint's underlying algorithms, merges these and presents the complete results to users. Finally, neither of the two systems described above support the SPARQL query language.

C. SPARQL-BASED NAVIGATIONAL APPROACHES

SPARQL is a standard language to query RDF and linked data. Recently, SPARQL 1.1. introduced navigational features called Property Paths. However, SPARQL property path queries allow only to traverse a single graph. To support this feature, several algorithms i.e., [2]–[5], [12]–[16] have been developed to navigate within a single graph. All of those approaches were aimed to either improve the efficiency of the pathfinding feature, handle larger graphs, or work for specific data models/formats (e.g., RDF3X, HDT). In our previous work [1], we proposed an extension to the Property Path query feature with some additional features, which are missing in the current implementation of SPARQL property paths. However, this work also considered property path queries within a single graph/data source. All of the systems are therefore unable to leverage the benefits of path query federation, although different engines e.g., Apache Jena-based Virtuoso,[8] Stardog, etc. implement SPARQL query federation (i.e. through the SERVICE clause) for other kinds of queries. In recent years there has been extensive work [17]–[28] on query federation, however, surprisingly none of them addressed or implemented the path query federation. Here, we extend our previous work from a single graph to federated graphs and address the specific challenges associated with federating property path queries.

D. PATH INDEXING

To efficiently calculate paths, several techniques such as [29]–[31] have been proposed in the past. Some of these approaches are designed for relational databases and employ a B+tree-based path index, while others are developed for the RDF model. However, as above, all previous indexing approaches are developed for *single graph* traversal

[3]hhttps://www.stardog.com/blog/graphql-and-paths/

[4]https://neo4j.com/developer/kb/all-shortest-paths-between-set-of-nodes/

[5]https://wiki.blazegraph.com/wiki/index.php/PropertyPaths

[6]https://www.microsoft.com/en-us/research/project/trinity/

[7]http://hypergraphdb.org

[8]https://virtuoso.openlinksw.com

algorithms. We, in contrast, adopted an approach that is used to calculate paths over the distributed datasets.

E. RDF-BASED PATH FINDING IN DISTRIBTUED GRAPHS

In recent years, in the context of the Semantic Web, initiatives have appeared [32], [33] and [34], [35] proposing to process paths over distributed graph data. In [32], [33] they proposed approaches to retrieve paths over the 'Web', through Linked Traversal. This approach is based on traversing RDF web documents and does not rely on SPARQL endpoints. Also, in the link traversal-based query procedure, the biggest challenge is that it might lead to following dead links, and there is no guarantee of the completeness of the results. Moreover, as they rely on blind search approaches, they are most costly in performance, have to rely on a large number of `http` requests, and do not employ source selection criteria. In [34] partial path evaluations are carried out over distributed datasets. However, this approach follows a customized *shared algorithm* deployed over every dataset. Moreover, in [34], [35] the distribution of the data is controlled centrally (i.e. partitioned), which is not feasible when using external data endpoints. In our approach, we do not rely on a customized distribution of the data but perform pathfinding queries across heterogeneous data distributed (publicly exposed by different providers) across the network.

To sum-up the related work, existing approaches differ from our approach mostly by focusing on single data sources or relying on specific data distribution mechanisms, assuming control over it. While achieving similar tasks, none of those systems can therefore be straightforwardly compared to our approach.

V. THE QPPDS APPROACH

In this section, we explain the QPPDs approach in details. The QPPDs system has four main components: (1) the QPPDs index for distributed path computation, (2) path computation between the connected *datasets*, (3) distributed path computation between the resources within the connected datasets, and (4) the path merger. We explain these main components in the next sub-sections where we use a running example to walk through each step in QPPDs.

A. THE QPPDS INDEX

The QPPDs approach makes use of a pre-computed index for fast retrieval of K possible paths between the source and target resources in distributed RDF graphs. We assume a set of data sources $\mathcal{D} := \{D_1, \ldots D_n\}$ where each data source $D \in \mathcal{D}$ is a SPARQL endpoint. We say a connection $\Psi(D, D')$ holds between two datasets D and D' if both datasets have a common node, i.e., $\{\Psi(D, D') \mid \exists N : N \in \mathcal{D} \wedge N \in \mathcal{D}'\}$. Furthermore, $Rs(D, D')$ represents all such common nodes between $D \wedge D' \in \mathcal{D}$. For each data source $D \in \mathcal{D}$, QPPDs stores the following as index:

1) The set of datasets connected to D: $Cs(D) := \forall_{D' \in \mathcal{D}}\{D' \mid \Psi(D, D') \wedge D \neq D'\}$ (`:connectedTo`).

```
## ConnectedTo --------
D1 :connectedTo D2,D3 .
D2 :connectedTo D1,D3 .
D3 :connectedTo D1,D2.
## isCommonIn --------
A :isCommonIn D1, D2 .
B :isCommonIn D2, D3 .
C :isCommonIn D1, D3 .
D :isCommonIn D1, D2 .
```

Listing 2. The QPPDs index of the datasets of motivating example given in Figure 2a.

2) For each dataset $D' \in Cs(D)$ the set of resources which connects (i.e., common) D and D', i .e., $Rs(D, D')$ (`:isCommonIn`).

Listing 2 shows an excerpt of the corresponding index of the datasets of the example given in Figure 2a. Please note that we used a simplified representation for the sake of simplicity. In reality, our index is represented as valid RDF in the NTriples format. We compute the index of the datasets by simply sending relevant SPARQL queries to the corresponding endpoints of the datasets.

Algorithm 1 shows the QPPDs index generation, which takes the set of all datasets $\mathcal{D}$ as input and returns the corresponding QPPDs index I as output. For each dataset $D \in \mathcal{D}$, we first get the set of all URI nodes (i.e., resources) $\mathcal{N}$ (Lines 1-2 of Algorithm 1). The set of all URI nodes from a dataset are retrieved by using a single SPARQL `SELECT` query given in Listing 3. For each node $N \in \mathcal{N}$, we then create a SPARQL `ASK` query (given in Listing 4) and send it to all datasets D (Lines 3-5 of Algorithm 1). The datasets that return **true** to given SPARQL `ASK` are treated as connected to D and share a common node N. All common nodes are added to the index (Lines 6-9 of Algorithm 1).

Algorithm 1 QPPDs Index Construction

```
input : $\mathcal{D} := \{D_1, \ldots D_n\}$ ; /* All datasets */
output: I;                                  /* The QPPDs index */
1  foreach $D \in \mathcal{D}$ do
2      $\mathcal{N} \leftarrow$ getURINodes($D$) ;
3      foreach $N \in \mathcal{N}$ do
4          foreach $D' \in \mathcal{D} \wedge D' \neq D$ do
5              b $\leftarrow$ ASK($N$,$D'$) ;
6              if ($b$ == 'true') then
7                  I.add($D$ : connectedTo $D'$) ;
8                  I.add($N$ : isCommonIn $D'$) ;
9              end
10         end
11     end
12 end
13 return I
```

B. PATHS COMPUTATION BETWEEN DATASETS

In Linked Data (our use case), RDF datasets are interconnected via various links. Consider our motivating example

```
SELECT ?s ?o {?s ?p ?o. Filter isUri(?o)}
```

Listing 3. SPARQL to select all subject and object.

```
ASK WHERE {?s ?p ?o
  Filter(?s = <%N> ?o= <%N> )
  }
```

Listing 4. SPARQL to ASK subject and object.

given in Figure 2a, all three datasets are interconnected via different links as shown in Figure 3. In order to compute paths – for a given query (see Listing 6) – between resources within the distributed datasets (algorithm 3), we first need to compute all the possible paths to reach from one dataset to another dataset. For example, in our synthetic motivating example, the possible paths between dataset D1 and dataset D2 (under the condition that the maximum allowed repetition of nodes in a path equals 2) are given in Listing 7. Thus, to compute all the possible paths between two datasets, we first need to make a graph of datasets similar to Figure 3 and then apply some algorithm on this graph to retrieve the required paths. We make use of the index (i.e Listing 5) to get the required graph of datasets and then apply a simplified version of Breadth-First-Search (BFS) with some modifications to calculate the paths over the given graph. Algorithm 2 explains how the paths are calculated from the source and target datasets in a multi-graph of interconnected datasets.

Algorithm 2 finds k paths between the source and target nodes in multi-graph. Thus it requires the source node s (source dataset in our case), target node t, number of paths k, and multi-graph G as input and retrieves the top-k shortest paths between the source and target nodes as output. Lines 6 to 11 correspond to a standard BFS-based approach. However, at Line 12 we slightly diverge from BFS: During the traversal, if a node is already visited, we do not consider it as a visited node, but this visited node is again queued to traverse until it reaches the maximum visited count of the maximum links between any two nodes in G. For example, the maximum links between two nodes in the multi-graph given in Figure 3 is 2. Using this modification, we are able to retrieve paths involving cycles in the given graph. The rest of the algorithm corresponds again to a standard BFS approach where queue q stores all the paths starting from the start D_{source} (e.g. $D1$) in Listing 7 ordered by path length. In every next iteration, the path p is extracted from the queue q extending p by one hop edge. The new extended path p' along with the previous path is again queued and whenever the next extended edge leads to the D_{target}, (i.e. $D2$) in Listing 7, this complete path is stored in the list of solutions sol (Line 18 of Algorithm 2). When the number of the solution reaches k, the traversing process is terminated.

Algorithm 2 Algorithm to Find k Paths Between Source and Target Datasets

```
1  s ← $D_{source}$ ;                          /* source node */
2  t ← $D_{target}$ ;                          /* target node */
3  k ← $K_{paths}$ ;            /* search number of TopK paths */
4  D ← $G$ ;   /* A multi graph generated by CONSTRUCT query over index */
5  sol ← $\emptyset$ ;          /* solution (retrieved properties) */
6  q[$\emptyset$] ← $|T_i, \ldots, T_n|$ ;     /* queue of triples T(s,p,o) */
7  while $(\mathcal{D}_i)_{i=1}^{n} \neq \emptyset$ do
8      q ← $\{T(s,p,o)\}$ ;   /* pull and remove triple(T) with source node into queue */
9      while $q \neq \emptyset$ do   /* check until queue is empty */
10         $(tp_i)_{i=1}^{n} \leftarrow q.T(s,p,o)$ ; /* get all child nodes of source node(s) and corresponding triples(tp) */
11         for $(tp_i)_{i=1}^{n}$ do
12             if $(tp(o)_i = literal) \vee (isVisted(tp(o)_i) >= maxNodesLinks)$ then ;   /* move next */
13
14                 continue;
15             else if $(tp(o)_i = t)$ then ;   /* if target node found */
16
17                 p ← $path(tp(o)_i)$ ;   /* store path */
18                 sol ← $p$ ;   /* solution is found */
19                 flag ← true ;   /* do not store in queue since path is found */
20                 if $sol$.size $\geq K$ then
21                     return sol;
22                 end
23                 else if $flag \leftarrow false$ then
24                     q ← $q$.add($tp(o)_i$)
25             return sol ;   /* return solution */
26         end
27     end
28 end
```

C. DISTRIBUTED PATH COMPUTATION

In this section we explain, in detail, the QPPDs distributed path computation given in Algorithm 3. The algorithm takes the source node n_{source} and target node n_{target} for which all paths are required to be computed over the set of distributed datasets $\mathcal{D}$. The algorithm –incorporating with previous algorithms 1 and 2– retrieves all distributed paths between the required two nodes.

The source and target datasets (i.e., D_{source} and D_{target}, respectively) at step 1: are selected, by sending SPARQL

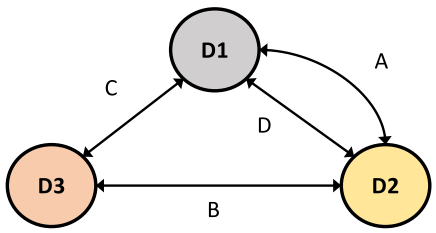


FIGURE 3. Multi-graph of datasets connectivity of our running example.

```
prefix feds: <http://vocab.org.centre.insight/
    feds#>

CONSTRUCT {?s feds:connectedTo ?o}
WHERE {?s feds:connectedTo ?o}
```

Listing 5. SPARQL Construct query to find the :connectedTo.

ASK queries to all of the datasets $\mathcal{D}$, before the actual path query in Listing 6 for n_{source} and n_{target} is executed. The datasets which return **true** for n_{source} are added to set S and datasets which return **true** for n_{target} are added to set T (Lines 2-5 of Algorithm 3). In our running example, n_{source} is F and n_{target} is E. Since F can only be found in $D1$, hence $S := \{D1\}$ and E is only present in $D2$, therefore, $T := \{D2\}$.

In step 2:, we need to compute possible paths from source datasets S to target datasets T within the generated index. To do so, we compute the multi-graph G, showing how the datasets in S are connected to the datasets in T (Line 7 of Algorithm 3). This graph can be constructed from the QPPDs index by using a SPARQL CONSTRUCT query given in Listing 5. Figure 3 shows the corresponding multi-graph of the datasets used in our running example. Now graph G can be considered as a single RDF dataset and Algorithm 2 is used to find paths within the graph G. To this end, we run Algorithm 2 over graph G to find all paths from all the datasets in S to all the datasets in T (Lines 8-12 of Algorithm 3). In our running example, since $S := \{D1\}$ and $T := \{D2\}$, we need to find all possible paths from $D1$ to $D2$ in the multi-graph G given in Figure 3. Since G is a multi-graph with cycles, it is possible that we can have an infinite number of possible paths. However, according to graph theory, we can still retrieve complete paths while allowing a maximum number of repetition of a node within a path equal to the maximum number of edges between two nodes in graph G. In the multi-graph given in Figure 3, the maximum number of edges between any two datasets is 2 (i.e., between D1 and D2). Under this condition, the paths retrieved by Algorithm 2 over the graph of Figure 3 are given in Listing 7.

Until step 2: of the algorithm 3, we have computed all paths from the source datasets (containing resource n_{source}) to target datasets (containing resource n_{target}). However, these paths are only at the dataset level, i.e. linking a dataset to

Algorithm 3 QPPDs Distributed Property Paths Finding Algorithm

```
input : $Q = n_{source} \wedge n_{target}$, $\mathcal{D} := \{D_1, \ldots D_n\}$ ;
        /* query, set of all datasets */
output: P ; /* Set of distributed paths */
1  /* Step 1: get source and target datasets */
2  foreach $D \in \mathcal{D}$ do
3      S ← $D_{source}$ + ASK ($n_{source}$,$D$) ;
4      T ← $D_{target}$ + ASK ($n_{target}$,$D$) ;
5  end
6  /* Step 2: get all paths from each source to target datasets */
7  G ← getConnectedGraph ($S$,$T$) ;
8  foreach $s \in S$ do
9      foreach $t \in T$ do
10         SP ← SP + getPaths ($G$,$s$,$t$)
11     end
12 end
13 /* Step 3: get distributed paths from source resource to
   target resource */
14 foreach $p \in SP$ do
15     pathLength ← p.length ;
16     count ← 0 ;
17     sources [] ← Ø ;        /* array of source
       resources */
18     curDataset ← p.getNextDataset () ;
19     sources.addResource ($n_{source}$) ;
20     while (count < pathLength) do
21         nextDataset ← p.getNextDataset () ;
22         if (nextDataset ≠ Ø) then
23             targets [] ← getCommonIn (curDataset,
               nextDataset) ;
24             sources []=ifNodeHasChild (sources,
               curDataset);
25             targets []= ifNodeHasParent (targets,
               curDataset);
26             /* These nested loops work in batch style ;
27             and generate query accordingly */
28             foreach source in sources [] do
29                 foreach target in targets [] do
30                     Q ← generateQuery (source,
                       target);
31                     P' ← FedRequest (curDataset,Q) ;
32                 end
33             end
34             curDataset ← nextDataset ;
35             sources [] ← targets [];
36         end
37         else
38             target ← $n_{target}$ ;
39             foreach source in sources [] do
40                 Q ← generateQuery (source, target);
41                 P' ← FedRequest (curDataset,Q) ;
42             end
43         end
44         count++;
45     end
46     return P ← mergePaths (P') ;
47 end
```

```
PREFIX ppfj: <java:org.centre.insight.property.
    path.>
SELECT * WHERE{
  ?path ppfj:topk (< $n_{source}$ > < $n_{target}$ > k)
       }
```

Listing 6. QPPDs SPARQL path query.

```
p1: D1 → D2
p2: D1 → D3 → D2
p3: D1 → D2 → D1 → D2
p4: D1 → D2 → D3 → D2
p5: D1 → D3 → D1 → D2
p6: D1 → D2 → D1 → D3→ D2
p7: D1 → D2 → D3 → D1→ D2
p8: D1 → D3 → D2 → D1→ D2
```

Listing 7. Possible paths from D1 to D2 in multi-graph given in Figure 3 with a condition of max. allowed repetition of node in a path equals 2.

```
SELECT DISTINCT ?node WHERE {
   {?node :isCommonIn <%curDataset> .
    ?node :isCommonIn <%nextDataset> .
 }
```

Listing 8. SPARQL query to find the common nodes between datasets.

another dataset, and do not tell us how to go from a node to another node within the distributed datasets. We achieve this in step 3 of the algorithm. In step 3:, we follow dataset paths computed in the previous step to get actual paths at node-level.

We maintain a list of all the dataset paths $p \in SP$ calculated bewteen D_{source} and D_{target}. For each path $p \in SP$, we first compute its length, i.e. the number of datasets to be tested for query $Q = n_{source}, n_{target}$ (Line 15 of Algorithm 3). For example, in Listing 7 the first dataset path $p1$ is $D1 \rightarrow D2$ with length 2. We then initialize a counter, an array of source resources, get first dataset from the path, and add n_{source} (i.e., F) into array of source resources. For $p1$ of our running example, $curDataset$ would be $D1$ and resource F would be added into the $sources$ array. Now at Line 20, we check if the $count$ is less than path length, which is true in the current state of our running example (i.e., $count = 0, pathLenghty = 2$). We get the next dataset ($D2$ at current) from the path and store in $nextDataset$. Since the next dataset is not empty, we get all the common nodes in $curDataset$ and $nextDataset$ (Lines 21-23 of Algorithm 3) from the index. In our running example at present, the common nodes between $D1$ and $D2$ are A, D and these common nodes are added in to the $target$ set. The common nodes between two datasets are retrieved by using the query given in Listing 8.

Walking through the running example, when we have two sets of $sources = \{F\}, targets = \{A, D\}$ and dataset $curDataset = D1$ as well as the $nextDataset = D2$ (Lines 18-21 of Algorithm 3), we perform three steps to optimize our approach, just before sending the remote requests to $curDataset = D1$.

1) We check the set $sources$ for all its elements (i.e., nodes) in the $curDataset = D1$ if they have children (Line 24 of Algorithm 3). After filtering out all nodes with no children, we get an updated set of $sources$. We do this because a path search should stop at a node having no further children. At the current stage of our running example, the $sources$ set contains F, which has children, so we add it to the query generation.
2) Similarly, we filter out the $targets$ set for the $curDataset = D1$ and discard all target nodes who do not have parent nodes except the n_{target} (Line 25 of Algorithm 3). At the current stage of our running example, nodes $\{A, D\}$ in set $targets$ both have parents and are therefore not to be discarded.
3) We also check if $n_{source} = n_{target}$. If true we do not count this combination in path search, because this points to the same node.

In our running example, after applying the first two optimization techniques, we still maintain $sources = \{F\}$, $targets = \{A, D\}$, however, it is important to note that it would not always be the case in real-life data. From Line 28-32 of Algorithm 3, nested loop for $sources$ and $target$ respectively iterate in a batch style (depending on user settings) and generate the nested SPARQL path query Q while considering the third optimization technique (Line 30 of Algorithm 3). We discuss the impact of a batch-based nested query in the results section. In our running example, the generated SPARQL query (see Listing 6) is sent to the $curDataset = D1$. The results P', i.e., $\{F \rightarrow K \rightarrow A\}, \{F \rightarrow G \rightarrow A\}$ retrieved from $D1$ are stored against this dataset. It is important to note that P' represents the $partial\ paths$. However, it does not mean that partial paths set P' will maintain always incomplete paths, but in real-world scenarios there may be many cases of complete paths retrieved and stored in P'.

When the iteration for $curDataset = D1$ is finished and the next dataset (i.e., $D2$) becomes the $curDataset = D2$ and set of $targets$ (i.e., $\{A, D\}$) becomes the $sources$ (Lines 34-35 of Algorithm 3), the procedure jumps back to Line 21 of Algorithm 3, where it checks if more datasets are available. In the current scenario of p1: (see Listing 7) $D2$ is the last dataset, therefore, control goes to Line 39 of Algorithm 3.

Now at Line 39, we get $sources$ (i.e. $\{A, D\}$) and the $targets$ set will only contain the actual target resource n_{target} since there is no further dataset to explore, which means no more common node needs to be checked. From Lines 39-42 of Algorithm 3, the query is generated for each element of the loop as source, while keeping target node static –i.e., actual target node E of the query executed by user– and sent to $curDataset = D2$. we get results P', i.e., $\{A \rightarrow B \rightarrow E\}$, which are stored against $D2$. At this stage, the length of $count$ reaches to maximum, therefore the processing of p1: (see Listing 7) is terminated and control goes to Line 46 of Algorithm 3.

Now at Line 46 of Algorithm 3, a mergePaths Algorithm (Algorithm 4) is called, which merges all the remote paths, $P'\{F \rightarrow K \rightarrow A\}, \{F \rightarrow G \rightarrow A\}$ from $D1$ and $\{A \rightarrow B \rightarrow E\}$ from $D2$, to complete full paths P. After merging the paths we get a set of; $P\{F \rightarrow G \rightarrow A \rightarrow B \rightarrow E\}, \{F \rightarrow K \rightarrow A \rightarrow B \rightarrow E\}$.

So far, we have obtained the paths that are direct paths, meaning one-to-one relations between two connected datasets and no third dataset contributes in path computation. However, there may be many cases where indirect paths exist as shown in Figure 2b, where paths *#3 and #4* are *indirect paths*.

We step through in the following paragraph to understand that how Algorithm 3 works when it encounters indirect paths.

Indirect Paths :

In our running example, we take path #3 i.e., $(F \rightarrow K \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow E)$. Unlike the previous paths discussed before, which involve only two datasets (i.e. $D1$ and $D2$), the indirect path involves three datasets (i.e. $D1$, $D2$ and $D3$) to complete it between F and E.

Having provided a detailed explanation before, we here provide only a simplified description.

This particular path #3 is calculated when Line 10 of Algorithm 3 faces p7: $D1 \rightarrow D2 \rightarrow D3 \rightarrow D1 \rightarrow D2$ dataset path connectivity given in Listings 7. If we notice here, the first two datasets of p7: i.e., $D1 \rightarrow D2$, have already been explained for p1:. For simplicity, we skip their computation since we know that $D1$ and $D2$ return P' $\{F \rightarrow K \rightarrow A\}, \{A \rightarrow B \rightarrow E\}$ respectively. In the case of p7:, the algorithm does not terminate, since dataset $D2$ is further connected to $D3$, but further iterations are performed.

Lets say two iterations $(Itr_{1,2})$ were carried out for previous datasets. Hereon, we will use Itr_i for iteration number.

At Itr_3, $D2$ becomes $curDataset = D2$ and *sources* set is initialized with $\{A, D\}$ (i.e. common nodes of $D1$ and $D2$). During Itr_3, $D3$ becomes $nextDataset = D3$. Common nodes (i.e. $\{B\}$) between $D2$ and $D3$ are added to *targets*. At the end of Itr_3, when $\{A, D\} \times \{B\}$ is checked against $D2$, we get P' $\{A \rightarrow B\}$ against $D2$.

At Itr_4, $D3$ becomes $curDataset = D3$ and $\{B\}$ is added to *sources*. During Itr_4, $D1$ becomes $nextDataset = D1$. Common nodes (i.e. C) between $D3$ and $D1$ are added to the *targets*. At the end of Itr_4, when $\{B\} \times \{C\}$ is checked against $D3$, we get P' $\{B \rightarrow C\}$ against $D3$.

At Itr_5, $D1$ becomes $curDataset = D1$ and $\{C\}$ is added to *sources*. During Itr_5, $D2$ becomes $nextDataset = D2$. Common nodes $\{A, D\}$ between $D1$ and $D2$ are added to *targets*. At the end of Itr_5, when $\{C\} \times \{A, D\}$ is checked against $D1$, we get P' $\{C \rightarrow D\}$ against $D1$.

At Itr_6, $D2$ becomes $curDataset = D2$ and $\{A, D\}$ are added to *sources*. During Itr_6, Algorithm 3 finds no further dataset to explore (see p7: in Listings 7). At this stage the *targets* will only be one (i.e., $n_{target} = E$). At the end of Itr_6, when $\{A, D\} \times \{E\}$ is checked against $D2$, we get P' $\{D \rightarrow E\}$,$\{A \rightarrow B \rightarrow E\}$ against $D2$.

When computation is completed for p7: (Listings 7), the set $\{P'\}$ contains all the direct and indirect paths computed by Algorithm 3. Now, the mergePaths (see Algorithm 4) component takes the set of $\{P'\}$ and produces the complete paths P.

A complete list of paths shown in Figure 2b is generated when datasets paths (p1:–p8:) with the above-mentioned procedures are completed.

D. PATH MERGER

When the QPPDs Algorithm 3 has finished its task, we get a set of P' against each dataset. This P' is given to Algorithm 4, which assembles all the paths in such a way that we get a list of complete paths P. To explain the path merger algorithm, let's take our running example. We step through the two cases direct paths and the indirect paths for given query where $n_{source} = F$ and $n_{target} = E$.

Algorithm 4 Path Merger Algorithm

```
input : $[\mathcal{D}]_{P'}$;        /* $P'$ against relevant
        datasets */
output: P ;   /* Set of distributed paths
        */
1 for $(\mathcal{D}_i)_{i=1}^{n} \neq \emptyset$ do
2     if $[D_i]_{p'}$.prefix $= n_{source} \wedge [D_i]_{p'}$.postfix $= n_{target}$
      then
3         $P \leftarrow p'$;
4     else if $[D_i]_{p'}$.prefix $= n_{source} \wedge [D_i]_{p'}$.postfix $=$
      $[D_{i+1}]_{p'}$.prefix $\wedge [D_{i+1}]_{p'}$.postfix $= n_{target}$ then
5         $P \leftarrow p'$ ;            /* store path */
6     else if $[D_i]_{p'}$.postfix! $= n_{target} \wedge [D_{i+1}]_{p'}$.prefix $=$
      $[D_i]_{p'}$.postfix then
7         $[D_{i+1}]_{p'} \leftarrow [D_i]_{p'}$.concat($[D_{i+1}]_{p'}$)
8 end
9 return $P$ ;                 /* return solution */
```

Case-1 Direct Paths:

We explained earlier that the direct paths are calculated when Algorithm 3 comes into contact with the p1: shown in Listings 7. In this case, we get two paths P' $\{F \rightarrow K \rightarrow A\}, \{F \rightarrow G \rightarrow A\}$ against $D1$, and also two paths P' $\{A \rightarrow B \rightarrow E\}, \{D \rightarrow E\}$ against $D2$. Note that the algorithm 4 iteratively performs all steps, where it can go back-and-forth during the iteration. At Line 2 of Algorithm 4, it checks if any path in $D1$ either starts with $n_{source} = F$ or ends with $n_{target} = E$. If true, it will be stored in P. In the $D1$ case there is no such path that satisfies the previous condition. So all paths P' of $D1$ are checked against the next dataset paths P'. At Line 4 of Algorithm 4, it checks if any path P' from $D1$ starting with $n_{source} = F$ and ending with any node that is equal to the starting node of any path P' exists in $D2$. For p1:, in this case, condition becomes true (Line 4 of Algorithm 4) and we get complete paths in P $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow E\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow E\}$. The path P' $\{D \rightarrow E\}$ is discarded, as it does not satisfy the applied condition at Line 4 of Algorithm 4.

Case-2 Indirect Paths:

In the case of Indirect paths, when Algorithm 3 has solved p2:–p8 shown in Listings 7, we get the following paths P' $\{F \rightarrow K \rightarrow A\}, \{F \rightarrow G \rightarrow A\}, \{C \rightarrow D\}$ against $D1$, paths P' $\{A \rightarrow B \rightarrow E\}, \{D \rightarrow E\}, \{A \rightarrow B\}$ against $D2$, and also one path P' $\{B \rightarrow C\}$ against $D3$. At Line 4 of Algorithm 4, we get two complete paths P $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow E\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow E\}$, however, Algorithm 4 does not terminate at this stage. At Line 6 of Algorithm 4 concatenates the path from $D1$ and $D2$, i.e. $\{F \rightarrow K \rightarrow A \rightarrow B\}, \{F \rightarrow G \rightarrow A \rightarrow B\}$. In next iteration when $D1$ is checked against $D3$, we get $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow C\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow C\}$ at Line 6 of Algorithm 4. Next iteration at Line 6 of Algorithm 4 generates $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow C \rightarrow D\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow C \rightarrow D\}$. In the next iteration when algorithm at Line 4 of Algorithm 4 checks the condition, we get complete paths P $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow E\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow E\}$ and algorithm 4 is terminated.

VI. EVALUATION

In this section, we present the evaluation results of the QPPDs approach. We first explain the evaluation setup, followed by evaluation results and discussion.

A. EXPERIMENTAL SETUP

Since, to the best of our knowledge, there exists no publicly available benchmark to test the distributed property paths retrieval systems on top of distributed RDF datasets, we had to create a benchmark by ourselves. Now we explain the datasets and path queries used in our evaluation.

1) DATASETS

In our evaluation, we used 8 datasets – Disease, hpoClass, doClass, phenotype, Protein, Variant, Gene, and pantherClass – from a life-sciences domain with a combined total of 7.26 millions of triples. The "Disease" dataset contains information about the disease associated with at least one gene. The "Human Phenotype Ontology" (hpoClass), and "Disease Ontology" (doClass) both individually are the classification of the genes. The dataset "Phenotype" contains the cross-referenced IDs extracted from HPO. The dataset "Protein" is the UniProt IDs encoded by genes. "Variant" is the dataset that contains the variants associated to diseases. The "Gene" dataset contains gene information associated to disease. The dataset "PantherClass" classifies the genes' attributes according to the molecular functions.

We chose these datasets because they are interconnected and contain resources that are relevant to each other. Some of the high-level statistics of these datasets are shown in Table 1. All of the datasets used in our evaluation are publicly available from disgenet providers.[9]

[9]http://rdf.disgenet.org/download/v5.0.0/

TABLE 1. Disgenet datasets statistics.

Query	Triples	Subject	Predicates	Objects
Disease	738626	60130	12	489756
doClass	101	21	11	63
Gene	1056346	119522	12	834502
hpoClass	253	36	11	151
pantherClass	272	40	9	123
Phenotype	83292	8441	8	66249
Protein	160537	14635	8	117034
Variant	5225996	708405	16	3628674

Setting: We loaded each dataset into different Fuseki server instances, where our baseline algorithm already knit-in to each Fuseki server calculates and finds the paths for a given query request.

2) PATH QUERIES

We wanted to test our approach on the most complex SPARQL1.1 property path[10] queries, where we did not specify any *regex* expression but to find the arbitrary paths based on the wild card (i.e., (:|! :)∗).

We chose a total of 12 path queries for benchmarking, where each path query contains the source and target resources from selected benchmark datasets. The total number of possible paths P, the number of hops or nodes in the given path (along with max., min., avg., and std.), and the number of datasets (along with max., min., avg., and std.) involved in P are given in Table 2. While constructing the path queries, we considered carefully that paths between source and target must have multiple datasets involved. We fixed the query time-out to 90 seconds, meaning that if a query is not executed within the time limit, it is considered as a failure.

3) HARDWARE AND IMPLEMENTATION SETUP

We conducted path experiments on a local setup (i.e., local network) to maintain the network cost as low as possible. The 8 datasets used in our benchmark were loaded into Fuseki server version (1.3.0 2015-07-25T17). We used a cluster of 8 machines (Ubuntu OS) with 2.9GHzx8 Intel Core i7 processors, 4GB of RAM, and 250GB of storage capacity to run 8 Fuseki server. The QPPDs engine is implemented in Java 1.8, using Jena API. To run QPPDs, we used MacBook Pro (Mojave OS) with 2.6GHz Intel Core i5 processor, 16GB of RAM, and 500GB of storage capacity. The code and all the configurations are available at GitHub.[11]

4) METRICS

The metrics with which these queries were evaluated are: (i) the index generation time (ii) the index compression ratio,[12] i.e, the index size to dataset size ratio (iii) the number of sources selected for each query, (iv) source selection time,

[10]https://www.w3.org/TR/sparql11-property-paths/

[11]https://github.com/InsightGalway/Path-Federation

[12]Index ratio: indexSize/datasetSize

TABLE 2. Various characteristics of the benchmark path queries.

Query	Path	Hops				Datasets			
	Total	max	min	avg	std	max	min	avg	std
1	22	9	1	5	2.82	4	1	2.25	1.5
2	257	37	3	20	10.37	3	3	3	0
3	1	3	3	1	0	3	3	2	0
4	1	1	1	1	0	1	1	1	0
5	1	1	1	1	0	1	1	1	0
6	1	3	3	1	1	3	2	2.5	0.5
7	1	2	2	1	0	3	1	2.5	0.68
8	1	2	2	1	0	3	2	2.5	0.5
9	45	1	10	5.5	2.87	3	1	2.5	0.5
10	9	2	8	5	2.23	3	3	2	0
11	5	4	4	5	0	4	4	2	0
12	5	4	4	5	0	4	4	2	0

(v) number of hops in the paths (vi) the time taken to retrieve k paths. We provide a URL[13] for a SPARQL endpoint that contains the information about retrieved paths. On the GitHub page, we provide sample queries to check the different metrics.

5) BASELINE

The focus of this paper is to introduce a federation-based approach to path finding in distributed RDF datasets. To do so and considering the overhead involved in federation, we presented above several algorithms that include a number of features aimed at reducing the response time for individual queries. We therefore here assess the contributions of those features by comparing response times with and without those features, starting from a baseline corresponding to an implementation of QPPDs without the following optimization steps:

1) **Nodes Connectivity:** As mentioned before, the QPPDs approach only searches for complete paths if the intermediate nodes have both child and parent nodes. This is because a node without children or parent will terminate the current path computation. Therefore, before doing any processing on the given node, the QPPDs approach first checks for its child and parent node. In the baseline approach, we are not applying this filter until the algorithm itself discovers that the path is complete and terminates the processing for the current path search.
2) **Streaming approach:** The QPPDs approach works in batches of paths (as explained in section V-C), i.e, a streaming approach for path computation. The baseline algorithm does not work in batches.
3) **Requests Grouping:** Our distributed path computation algorithm sends multiple path requests to the underlying datasource, i.e., SPARQL endpoints. The QPPDs approach has this feature to combine multiple path requests into a single composite request. However, in the baseline algorithm, only one path request is sent to the endpoint at a given time.
4) **Same Source-Target Filtering:** It is also possible that for a given path query the source and target nodes are exactly the same. The QPPDs approach filters out such requests before sending them to the remote SPARQL endpoints, hence the number of requests is reduced.

[13]http://vmurq09.deri.ie:8030/sparql.tpl

The impact of these missing features on the performance of the baseline algorithm is discussed in the next section. While comparing with other path querying approaches would be valuable, at this point, that those are applicable to different contexts, and often unavailable, prevents us from being able to conduct a fair comparison.

B. RESULTS

1) INDEX GENERATION TIME

The QPPDs and the baseline approach both use exactly the same index of the given RDF dataset. Even though it is a one time process (assuming that no datasets updates), the index should be generated in a reasonable amount of time. Figure 4 shows the index generation time for each of the benchmark datasets. We can clearly see that the index generation time is dependent upon the size of the underlying dataset. The maximum time to compute the index for the largest dataset, i.e Variant, is only 136.6 seconds. The overall time to compute the indexes for the complete benchmark datasets is less than 6 minutes.

2) INDEX SIZE

The index should be small in size for fast lookups at runtime. The contribution of each of the benchmark datasets to the size of the index is shown in Table 3. We can clearly see that the indexes generated by our approach is very small in size. The cumulative size of the QPPDs index generated for all 8 datasets is only 6.1MB.

3) SOURCE SELECTION

As mentioned before, each of the benchmark path queries contains the source and target nodes, distributed in multiple

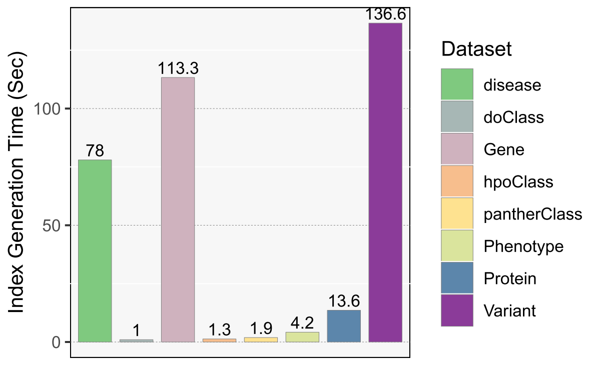


FIGURE 4. Index generation time.

TABLE 3. Index size for each dataset.

Dataset	Index Size (MB)	Dataset Size (MB)
Disease	0.3	116
doClass	0.004	0.014
Gene	3	170
hpoClass	0.013	0.038
pantherClass	0.008	0.047
Phenotype	0.015	12.6
Protein	2	22
Variant	1.1	995

TABLE 4. Number of datasets selected for the source and targets nodes of path queries.

Query	Source Dataset	Target Dataset
Q1	2	3
Q2	1	2
Q3	1	2
Q4	2	3
Q5	1	2
Q6	1	2
Q7	1	2
Q8	1	2
Q9	1	2
Q10	2	1
Q11	1	3
Q12	1	3

datasets. Thus, it is possible that the given source or target node is found in more than a single dataset. We define source selection results in terms of: (1) the number of datasets which contain the source and target nodes of a given path query (see Table 4), (2) the time required to identify the source and target datasets for the given path query (see Figure 5). Please note that the source selection of both baseline and QPPDs approach is exactly the same, thus these results are the same for both approaches.

From the results given in Table 4 and Figure 5, we can see that the QPPDs approach is able to quickly filter out (requiring milliseconds) the irrelevant sources.

4) QUERY RUNTIME PERFORMANCE

Figure 6 shows the query execution time for all 12 queries used in our evaluation. We observed that in two-third of the cases – for queries **Q1, Q2, Q3, Q4, Q6, Q7, Q9, and Q10** – QPPDs outperformed the baseline algorithm with almost

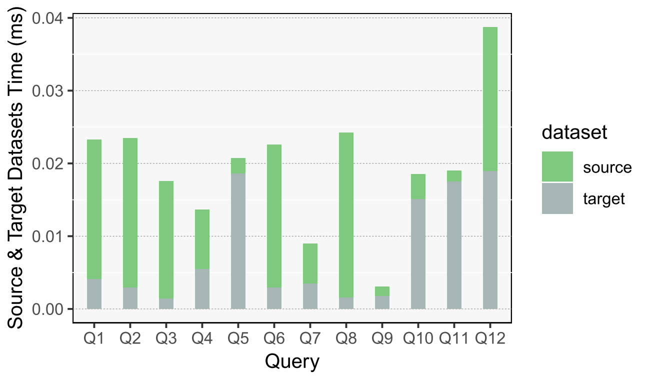


FIGURE 5. Time required to select the source and target datasets of benchmark path queries.

one order of magnitude. It is because, for all these queries, the possible paths between the connected datasets are very high comparing other queries. Thus, these queries generate more endpoint requests. The QPPDs, with its salient features, (in particular checking for the child and parents of nodes beforehand) generates fewer requests as compared to the baseline approach. For the other queries **Q5, Q8, Q11 and Q12** the baseline approach performs better as compared to QPPDs. This is because these queries are rather simple and need smaller time as compare to other queries. Thus, the QPPDs approach spends extra time for checking the child and parent of nodes involved in the final path. We can conclude that the QPPDs approach generally performs better for complex path queries.

a: SCALABILITY OVER NUMBER OF SITES AND DATA SIZE

We noticed that the datasets involved in the results obtained from queries **Q1, Q2, Q3, Q4, Q6, Q7, Q9, and Q10**, are big in size (see Table 2) and have more connected nodes between these datasets. This shows that the full QPPDs approach improves response time, as compared to the baseline, in the following situations for a given query:

- if the required path is distributed across more sites (datasets)
- if datasets are large in size

It is important to note that the comparison shown in Figure 6 is when a single path request is sent to the underlying triplestores, i.e, the *request grouping* feature of the QPPDs is not activated. In the next section, we will see that the performance of QPPDs further improves with this optimization parameter.

5) EFFECTS OF QPPDS OPTIMIZATION PARAMETERS

Now we show how much the query performance is effected either by enabling or disabling the QPPDs optimization parameters.

a: REQUESTS GROUPING

Figure 7 shows the query runtime performances by using a single request per query (QPPDs-1), grouping two requests

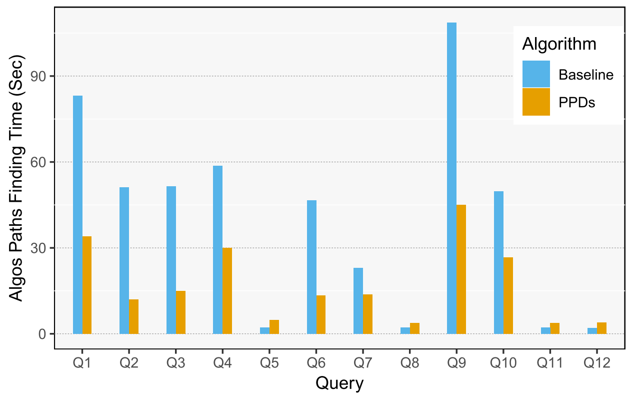


FIGURE 6. Query runtime performances of the QPPDs and baseline approaches when *request grouping* feature is not activated in QPPDs.

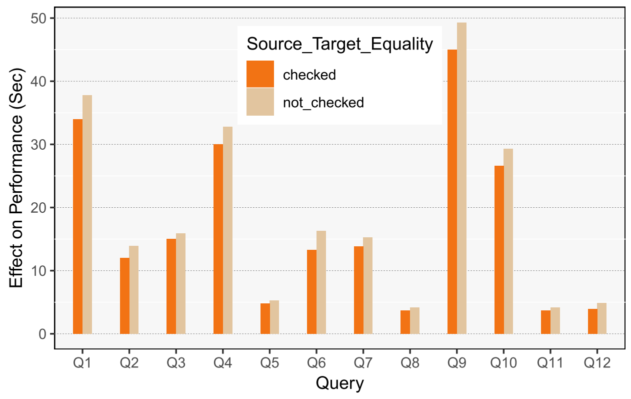


FIGURE 8. Effect of filtering same source and target nodes.

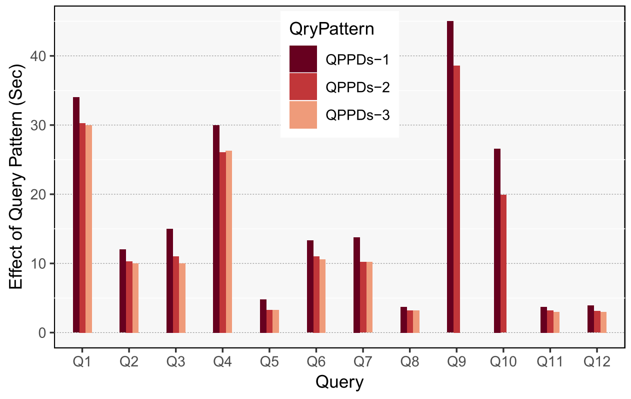


FIGURE 7. Effect of *request grouping* feature.

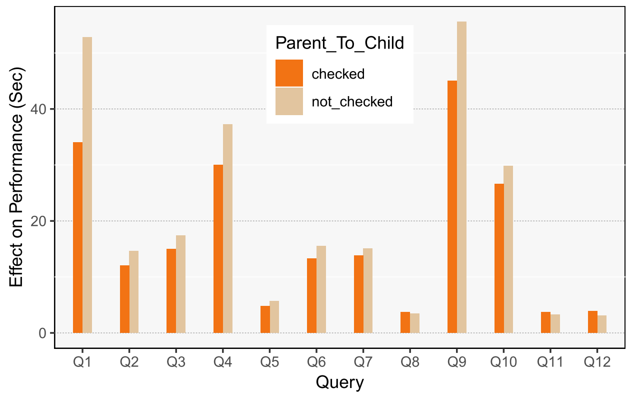


FIGURE 9. Effect of checking if a given node has child node.

per query (QPPDs-2) and grouping three requests per query (QPPDs-3). The overall results show that the performance is improved with the grouping of results. In most of the queries the performance is further improved when we increased the grouping size of requests. However, for QPPDs-3, the queries **Q9, Q10** resulted in a time-out (90 sec). This leads to the question of whether the performance will be further improved if we further increase the grouping size. We tested with QPPDs-4 (i.e, grouped 4 requests per query) and noticed that the performance significantly dropped compared to QPPDs-3. We investigated the reason and noticed that the SPARQL endpoints hosting the underlying datasets were not able to handle more complex requests and hence started giving timeouts. Thus, the performance of QPPDs is also strongly dependent on the query processing capabilities of the underlying triplestores. We believe that if efficient path finding algorithms on remote endpoints are deployed, the performance of the QPPDs can further be enhanced.

b: SAME SOURCE-TARGET FILTERING

Figure 8 shows the effect of checking if the source and target represent the same node and hence skipping the path calculation process. Overall, enabling this feature improved the query runtime performance for all of the 12 benchmark queries. On average, the runtime is improved by 19% by enabling this feature.

c: NODE HAS CHILD

Figure 9 shows the effect of checking if a node has missing child nodes and if the current path is broken, i.e it cannot reach the desired target node. Enabling this feature has resulted in improving the runtime performance for 9/12 queries. For queries Q8, Q11, Q12 the performance is degraded. The reason for this is that the datasets involved in each query have more parent to child relations. However, the connected relations are not involved in the construction of the path. Hence, processing these relations takes more time compared to disabling this feature. On average, the runtime performance is improved by 22% by enabling this feature.

d: NODE HAS PARENT

Figure 10 shows the effect of checking if a node has a missing parent node and the current path is broken, i.e, it cannot reach the desired target node. Enabling this feature resulted in improving the runtime performance for 5/12 queries. For queries Q3, Q5, Q7,Q8,Q10,Q11 the performance is degraded. The reason for this is that the targeted datasets

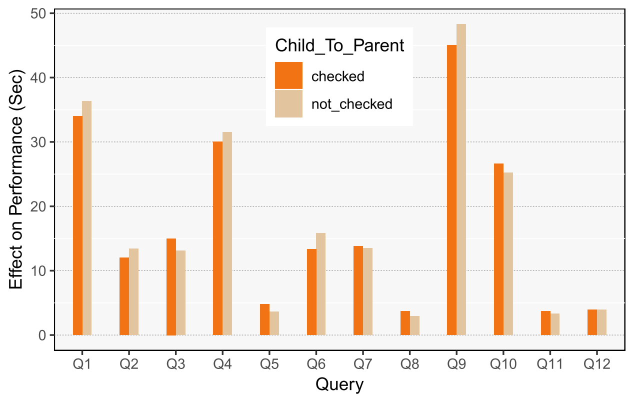


FIGURE 10. Effect of checking if a given node has parent nodes.

TABLE 5. Comparison of the paths retrieved by the baseline approach and our QPPDs approach.

Query	QPPDs		Baseline		TopK	
	tot-paths	max-hops	tot-paths	max-hops	tot-paths	max-hops
1	22	9	22	9	22	9
2	**257**	37	**81**	15	**257**	37
3	1	3	1	3	1	3
4	1	1	1	1	1	1
5	1	1	1	1	1	1
6	1	3	1	3	1	3
7	1	1	1	1	1	1
8	1	2	1	2	1	2
9	45	10	45	10	45	10
10	9	8	9	8	9	8
11	5	4	5	4	5	4
12	5	4	5	4	5	4

involved in query processing have more nodes that have parent nodes, and calculating these relations by enabling this feature takes more time. On average, the runtime is improved by 17% by enabling this feature.

6) RESULTS COMPLETENESS

To verify the QPPDs results completeness, we merged these 8 datasets into single graph and tested with the centralized approach "TopK" presented in our previous work [1]. Table 5 shows the comparison between the QPPDs, TopK, and baseline approaches in terms of *total number of paths* retrieved, maximum and minimum length (number of hops) of these paths. We noticed that for all queries the returned results are the same in the three approaches, except for one query. The query **Q2** with the baseline algorithm returned fewer paths (i.e., 81) and with a maximum path hops 15. While on the other-side QPPDs returned 257 paths with maximum path hops 37. This is because the baseline algorithm sent more parallel requests to some of the remote endpoints, exceeding the endpoint limit to handle the requests and hence started to give the exception *Cannot assign requested address*.

VII. CONCLUSION

The motivation behind this work is the need of the BIOOPENER project, which aims at linking and discovery of linked data across cancer and biomedical data at publicly available distributed triple stores. In this paper, we laid the foundation for distributed path queries and propose QPPDs, a path traversal approach that federates path queries across multiple SPARQL endpoints. The current SPARQL 1.1 Property Path specification and the standard traversal algorithms (BFS, DFS, A*, etc.) assume (or require) a single graph – or many graphs merged into a centralized graph – for graph traversal. QPPDs proposes a four-step approach that enables graph traversal in a federated environment. Our initial evaluation results are encouraging where QPPDs retrieves the paths in a competing query processing time. QPPDs contributes to path query federation in terms of (i) the source selection criteria to find the relevant datasets, (ii) the distribution of path queries in a batch style to remote endpoints. QPPDs, at the moment, applies to RDF data graphs. However, the approach is generalizable to other kinds of graphs (e.g., weighted, unweighted, property, etc.).

In terms of future work, there are a number of possible routes to optimize the current four-step process: we plan (i) to implement a heuristic indexing approach, so we can send a fewer number of remote requests, (ii) to test our approach on other available open access data engines which support path queries (e.g., Neo4J, Stardog, etc.), (iii) to extend its support for all types of Regular Path Queries `RPQ`, and (iv) to optimize QPPDs in such a way that, instead of traversing all commonIn combinations, if the user asks for K paths, it should terminate its processing when K is reached.

ACKNOWLEDGMENT

This article is based on the preliminary work [36] – traversing federated RDF graphs – presented previously.

REFERENCES

[1] V. Savenkov, Q. Mehmood, J. Umbrich, and A. Polleres, "Counting to k or how SPARQL1.1 property paths can be extended to top-k path queries," in *Proc. 13th Int. Conf. Semantic Syst.*, Sep. 2017, pp. 97–103.
[2] K. J. Kochut and M. Janik, "SPARQLeR: Extended Sparql for semantic association discovery," in *Proc. Eur. Semantic Web Conf.* Berlin, Germany: Springer, 2007, pp. 145–159.
[3] K. Anyanwu, A. Maduko, and A. Sheth, "SPARQ2L: Towards support for subgraph extraction queries in rdf databases," in *Proc. 16th Int. Conf. World Wide Web*, May 2007, pp. 797–806.
[4] A. Gubichev and T. Neumann, "Path query processing on very large RDF graphs," in *Proc. WebDB*, Jan. 2011, pp. 1–6.
[5] E. V. Kostylev, J. L. Reutter, and M. Ugarte, "Construct queries in sparql," in *Proc. 18th Int. Conf. Database Theory (ICDT)*, 2015, pp. 212–229.
[6] R. Simon and S. Roychowdhury, "Implementing personalized cancer genomics in clinical trials," *Nature Rev. Drug Discovery*, vol. 12, no. 5, pp. 358–369, Apr. 2013.
[7] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. Springer, 2011.
[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. MSST*, vol. 10, 2010, pp. 1–10.
[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, Jun. 2010, p. 95.
[10] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2010, pp. 135–146.
[11] D. Yang, D. Zhang, K.-L. Tan, J. Cao, and F. Le Mouël, "CANDS: Continuous optimal navigation via distributed stream processing," *Proc. VLDB Endowment*, vol. 8, no. 2, pp. 137–148, Oct. 2014.
[12] A. Gubichev, S. Bedathur, S. Seufert, and S. Seufert, "Fast and accurate estimation of shortest paths in large graphs," in *Proc. 19th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2010, pp. 499–508.

[13] E. Filtz, V. Savenkov, and J. Umbrich, "On finding the k shortest paths in RDF data," in *Proc. 5th Int. Workshop Intell. Explor. Semantic Data (ISWC)*, vol. 18, Oct. 2016.

[14] M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, and G. Lausen, "RDF-Path: Path query processing on large RDF graphs with mapreduce," in *Proc. ESWC*. Berlin, Germany: Springer, 2011, pp. 50–64.

[15] E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoč, "SPARQL with property paths," in *Proc. Int. Semantic Web Conf.* Cham, Switzerland: Springer, 2015, pp. 3–18.

[16] S. Hertling, M. Schröder, C. Jilek, and A. Dengel, "Top-k shortest paths in directed labeled multigraphs," in *Semantic Web Evaluation Challenge*. Cham, Switzerland: Springer, 2016, pp. 200–212.

[17] S. Lynden, I. Kojima, A. Matono, and Y. Tanimura, "ADERIS: An adaptive query processor for joining federated SPARQL endpoints," in *Proc. OTM Conf. Int. Conf. Move Meaningful Internet Syst.* Berlin, Germany: Springer, 2011, pp. 808–817.

[18] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus, "ANAPSID: An adaptive query processing engine for SPARQL endpoints," in *Proc. Int. Semantic Web Conf.* Berlin, Germany: Springer, 2011, pp. 18–34.

[19] C. Basca and A. Bernstein, "Avalanche: Putting the spirit of the Web back into semantic Web querying," in *Proc. Posters Demonstrations Track, Collected Abstr.*, vol. 658, 2010, pp. 177–180.

[20] B. Quilitz and U. Leser, "Querying distributed RDF data sources with SPARQL," in *Proc. Eur. Semantic Web Conf.* Berlin, Germany: Springer, 2008, pp. 524–538.

[21] 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 *Proc. Int. Semantic Web Conf.* Springer, 2013, pp. 574–590.

[22] A. Nikolov, A. Schwarte, and C. Hütter, "Fedsearch: Efficiently combining structured queries and full-text search in a SPARQL federation," in *Proc. Int. Semantic Web Conf.* Springer, 2013, pp. 427–443.

[23] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "FedX: A federation layer for distributed query processing on linked open data," in *Proc. Extended Semantic Web Conf.*, Springer, 2011, pp. 481–486.

[24] O. Görlitz and S. Staab, "SPLENDID: SPARQL endpoint federation exploiting VOID descriptions," in *Proc. 2nd Int. Conf. Consuming Linked Data*, 2011, pp. 13–24.

[25] M. Saleem and A.-C. N. Ngomo, "Hibiscus: Hypergraph-based source selection for SPARQL endpoint federation," in *Proc. Eur. Semantic Web Conf.* Springer, 2014, pp. 176–191.

[26] G. Montoya, H. Skaf-Molli, P. Molli, and M.-E. Vidal, "Federated SPARQL queries processing with replicated fragments," in *Proc. Int. Semantic Web Conf.* Springer, 2015, pp. 36–51.

[27] A. Hasnain, Q. Mehmood, S. S. E. Zainab, M. Saleem, C. Warren, Jr., D. Zehra, S. Decker, and D. Rebholz-Schuhmann, "BioFed: Federated query processing over life sciences linked open data," *J. Biomed. Semantics*, vol. 8, no. 1, p. 13, Mar. 2017.

[28] Y. Khan, M. Saleem, M. Mehdi, and A. Hogan, "SAFE: SPARQL federation over RDF data cubes with access control," *J. Biomed. Semantics*, vol. 8, no. 1, p. 5, Dec. 2017.

[29] G. H. Fletcher, J. Peters, and A. Poulovassilis, "Efficient regular path query evaluation using path indexes," Tech. Rep., 2016.

[30] B. Liu and B. Hu, "Path queries based RDF index," in *Proc. 1st Int. Conf. Semantics, Knowl. Grid*, Nov. 2005, p. 91.

[31] J. Sirén, N. Välimäki, and V. Mäkinen, "Indexing graphs for path queries with applications in genome research," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 11, no. 2, pp. 375–388, Mar./Apr. 2014.

[32] O. Hartig and G. Pirrò, "SPARQL with property paths on the Web," *Semantic Web*, vol. 8, no. 6, pp. 773–795, Aug. 2017.

[33] J. Umbrich, A. Hogan, A. Polleres, and S. Decker, "Link traversal querying for a diverse Web of data," *Semantic Web*, vol. 6, no. 6, pp. 585–624, 2015.

[34] X. Wang, J. Wang, and X. Zhang, "Efficient distributed regular path queries on RDF graphs using partial evaluation," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2016, pp. 1933–1936.

[35] A. Davoust and B. Esfandiari, "Processing regular path queries on arbitrarily distributed data," in *Proc. OTM Conf. Int. Conf. Move Meaningful Internet Syst.* Cham, Switzerland: Springer, 2016, pp. 844–861.

[36] Q. Mehmood, A. Jha, D. Rebholz-Schuhmann, and R. Sahay, "FedS: Towards traversing federated rdf graphs," in *Proc. Int. Conf. Big Data Anal. Knowl. Discovery*. Cham, Switzerland: Springer, 2018, pp. 34–45.

QAISER MEHMOOD received the M.Sc. degree in computer engineering from Mid Sweden University, Sweden. He is currently pursuing the Ph.D. degree with the Insight Centre for Data Analytics [formerly Digital Enterprise Research Institute (DERI)], National University of Ireland Galway, Ireland, with a focus on distributed path queries and graph mining. His research interests include semantic web, query federation, and data cataloguing and linking.

MUHAMMAD SALEEM received the Ph.D. degree in computer science from AKSW, University of Leipzig. He is currently the Unit Leader of the Data Storage and Querying Group, DICE, University of Paderborn, and AKSW, University of Leipzig. His research interests include SPARQL query processing and benchmarking, graph partitioning, and question answering over linked data.

RATNESH SAHAY received the bachelor's degree in information technology from the University of Southern Queensland, Australia, in 2002, the M.S. degree in distributed systems from the KTH Royal Institute of Technology, Sweden, in 2006, and the Ph.D. degree in computer science in healthcare informatics from the National University of Ireland, Galway, in 2012. He has more than 12 years of working/research experiences in healthcare and life sciences domain. He has been active and leading European and National Research and Development projects with an emphasis on e-health and semantic interoperability. He has served as a member of the OASIS SEE Technical Committee. He is also a member of the W3C OWL 2, W3C HCLS, and HL7 working groups.

AXEL-CYRILLE NGONGA NGOMO is currently a Full Professor with the University of Paderborn. He also leads the Agile Knowledge Engineering and Semantic Web research (AKSW) and the Data Science (DICE) Groups. His research interests include semantic web technologies, especially link discovery, federated queries, machine learning, and natural language processing.

MATHIEU D'AQUIN was a Senior Research Fellow with the Knowledge Media Institute of the Open University, where he led the Data Science Group. He is currently an Established Professor (Chair) of data analytics with the Data Science Institute, National University of Ireland Galway, and the Site-Director of the Insight Centre for Data Analytics. He is also leading research and development activities around the meaningful sharing and exploitation of distributed information. He has worked on applying the technologies coming out of my research, especially semantic web/linked data technologies, in various domains, including medicine, education especially through learning analytics, smart cities, the Internet of Things, and personal data management.