Efficient distributed path computation on RDF knowledge graphs using partial evaluation

Qaiser Mehmood, Muhammad Saleem, Alokkumar Jha, Mathieu D'aquin

▶ To cite this version:

Qaiser Mehmood, Muhammad Saleem, Alokkumar Jha, Mathieu D'aquin. Efficient distributed path computation on RDF knowledge graphs using partial evaluation. World Wide Web, 2022, 25 (2), pp.1005-1036. 10.1007/s11280-021-00965-5 . hal-03659142

HAL Id: hal-03659142
https://hal.science/hal-03659142

Submitted on 4 May 2022

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

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

ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/355710532

Efficient Distributed Path Computation on RDF Knowledge Graphs Using Partial Evaluation

Article *in* World Wide Web · March 2022

DOI: 10.1007/s11280-021-00965-5

CITATIONS
0

READS
156

4 authors:

Qaiser Mehmood
National University of Ireland, Galway
36 PUBLICATIONS 411 CITATIONS
SEE PROFILE

Muhammad Saleem
University of Leipzig
97 PUBLICATIONS 1,180 CITATIONS
SEE PROFILE

Alokkumar Jha
Weill Cornell Medical College
72 PUBLICATIONS 1,236 CITATIONS
SEE PROFILE

Mathieu d'Aquin
National University of Ireland, Galway
203 PUBLICATIONS 3,109 CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

MK:Smart View project

Discourse Analysis for Automatic Fake News Classification View project

All content following this page was uploaded by Muhammad Saleem on 28 October 2021.

The user has requested enhancement of the downloaded file.

Noname manuscript No.
(will be inserted by the editor)

Efficient Distributed Path Computation on RDF Knowledge Graphs Using Partial Evaluation

Qaiser Mehmood · Muhammad Saleem · Alokkumar Jha · and Mathieu d'Aquin

Received: date / Accepted: date

Abstract A key property of Linked Data is the representation and publication of data as interconnected labelled graphs where different resources linked to each other form a network of meaningful information. Searching these important relationships between resources – within single or distributed graphs – can be reduced to a pathfinding or navigation problem, i.e., looking for chains of intermediate nodes. SPARQL1.1, the current standard query language for RDF-based Linked Data defines a construct – called Property Paths (PPs) – to navigate between entities within a single graph. Since Linked Data technologies are naturally aimed at decentralised scenarios, there are many cases where centralising this data is not feasible or even not possible for querying purposes. To address these problems, we propose a SPARQL PP-based graph processing approach – dubbed DpcLD – where users can execute SPARQL PP queries and find paths distributed across multiple, connected graphs exposed as SPARQL endpoints. To execute the distributed path queries we implemented an index-free, cache-based query engine that communicates with a shared algorithm running on each remote endpoint, and computes the distributed paths. In this paper, we highlight the way in which this approach exploits and aggregates partial paths, within a distributed environment, to

Q. Mehmood
Data Science Institute, NUI Galway
E-mail: qaiser.mehmood@insight-centre.org

M. Saleem
AKSW, Leipzig
E-mail: muhammad.saleem@informatik.uni-leipzig.de

Alokkumar Jha
Stanford University
E-mail: alokjha@stanford.edu

M. d'Aquin
Data Science Institute, NUI Galway
E-mail: mathieu.daquin@insight-centre.org

produce complete results. We perform extensive experiments to demonstrate the performance of our approach on two datasets: One representing 10 million triples from the DBPedia SPARQL benchmark, and another full benchmark dataset corresponding to 124 million triples. We also perform a scalability test of our approach using real-world genomics datasets distributed across multiple endpoints. We compare our distributed approach with other distributed and centralized pathfinding approaches, showing that it outperforms other distributed approaches by orders of magnitude, and provides a good trade-off for cases when the data cannot be centralised.

Keywords Distributed Graphs · Reachability · Federated Paths · Graph Traversal · Path Caching · Path Query · RDF · SPARQL 1.1 Property Path.

1 Introduction

In the past decade, graphs have become the defacto data model in some of the popular application domains such as bioinformatics, social, and road networks. One of the most used variants of the graph model is the labeled graph where vertices and edges are given names. In the context of Semantic Web technologies, this graph representation is called Linked Data, expressed using RDF (Resource Description Format), where entities are given their names with Uniform Resource Identifiers (URIs) which uniquely identify each entity. A plethora of such information, corresponding to various domains[1], exists over the Web and is exposed as RDF knowledge bases. Some of the prominent examples of such knowledge bases or datasets are YAGO [2], DBPedia [3] and Bio2RDF [4].

This large collection of interlinked and distributed data over the web has created new challenges in the context of data management, data integration and knowledge discovery. One of the key challenges is querying and analysing relevant information from such large-scale knowledge bases. For instance, in the context of graph analysis, navigational queries are at the basis of core tasks and processes. Several approaches have been proposed to perform path queries, with some supporting declarative queries, while others are based on imperative constructs. Moreover, some approaches work in a distributed manner while others in a central way. For instance, SPARQL1.1 [5], which is a declarative query language, introduced a feature called Property Paths (PPs). Using PPs, a user can check the existence of paths between two entities within a single graph. However, this variant is not very expressive, does not capture some of the important properties of path finding problems (e.g., path enumeration, K paths), and has performance issues [3] when dealing with large-scale data. To

[1] LOD Cloud: https://lod-cloud.net/
[2] https://yago-knowledge.org/
[3] https://www.dbpedia.org/
[4] https://bio2rdf.org/
[5] https://www.w3.org/TR/sparql11-query/

address these limitations, there has been an extensive amount of work done either for SPARQL path query extensions [33, 2, 35, 61, 21] or to efficiently find the paths using compressed data models [61, 21].

Many state-of-the-art RDF-based path finding approaches [25, 61, 33, 2, 35, 61, 21] work on single graph where complete RDF data is loaded into a single machine. However, due to autonomous and distributed, and linked nature of the RDF data, such approaches cannot exploit the full potential of the Linked Data. For example, the Bio2RDF data portal[6] provides a catalog of a total of 35 RDF datasets from health care domain. These datasets are interlinked and heterogeneous in nature. Practitioners often need to find the paths between different entities that can lead to various datasets [41]. In such cases, the single graph-based path finding approaches cannot be applied. Such challenges have justified the need for novel approaches for path queries to work in a distributed environment.

There have been some work done for distributed path queries. For example, ecosystems such as Hadoop [65] or Pregel [40] have become defacto standards for such tasks and are capable of navigating the path queries in a distributed manner. However, these systems are developed for customised data distribution within a controlled environment, and hence in the context of Linked Data such systems are not considered purely heterogeneous.

Most of systems (e.g., [22]) that are built on top of Hadoop ecosystems systems are categorised as homogeneous distributed systems. That is, they work in such environment where every distributed system or database must have the same software and configurations. Furthermore, the partition and distribution of data must follow certain procedures or constraints. Such systems are not feasible in many application where the partitioning strategy is not controlled by the distributed RDF system itself. There may be some administrative requirements that influence the data partitioning. For example, in some applications, the RDF knowledge bases are partitioned according to topics (i.e., different domains) or are partitioned according to different data contributors. On the other side, Pregel's vertex-centric computing model has been widely adopted in most of the recent distributed graph computing systems [19, 59, 74, 73, 44] because of being fault tolerant and also efficient as compared to Hadoop-based approaches. However, Pregel also suffered the aforementioned challenges that is the controlled environment.

Although partial evaluation is a well studied concept [6, 9, 10], in the context of SPARQL queries and Linked data, it has only recently been used [71, 36]. One of the most well-know challenge with partial evaluation is the large number of intermediate results. Moreover, complex queries suffer performance bottleneck [41] when evaluating on large-scale distributed RDF graphs. In the context of distributed paths, to assemble these partial paths into complete paths is also a challenge. There are some other approaches [41, 72, 24] exist which are capable of computing distributed paths over RDF data. However, some of these approaches (e.g., [41]) require a pre-computed up-to-date index.

[6] Bio2RDF data portal: https://download.bio2rdf.org/files/release/3/release.html

Hence, path completeness is only assured if the index is up-to-date according to the current status of the underlying distributed RDF datasets. Furthermore, some of these approaches, e.g., [24], are unable to perform lookups for non-dereferenceable URIs [68]. Hence, they may retrieve empty or incomplete results. Note that our previous work [69] shows that around 43% of the URIs of more than 660K RDF datasets from LODStats [16] and LODLaundromat [5] are non-dereferenceable.

In this paper, we address these challenges and limitations, exist in state-of-the-art approaches. Our contributions are as follows:

i) We propose an efficient path finding approach – using *partial evaluations* [29] – in distributed environments. Our solution allows users to find K number of paths globally.
ii) We proposed an efficient way to assemble partial paths to get the final K numbers of complete paths.
iii) Our approach is easy to adopt, and unlike the Hadoop or Pregel ecosystems, it does not require complex configuration and data management.
iv) Our approach is index-free, efficient and does not require dereferenceable URIs. Therefore, it is agnostic of the changes in the underlying distributed datasets.
v) We proposed a cache assisted technique that leads to significant performance improvement by avoiding duplicate requests and filtering irrelevant data sets for the given path query.
vi) We performed extensive experiments with *synthetic* and *real-world* data. We compare our approach with local and remote systems that support querying K paths. We measure the impact of complex queries in terms of performance (i.e, query execution time) and memory consumption.

The structure of this paper is as follows: Section 2 revies the related work. In Section 3 we present the preliminary definition to understand the related terms. In Section 4, we then explain the architecture of the system and our approach with a running example to make the core components easy to understand. We provide an extensive evaluation both for *synthetic* and *real-world* data, and also provide comparisons with state-of-the-art approaches (i.e., distributed and centralized). Finally, we present our conclusion and future work.

2 Related Work

In recent years, with the increasing availability of large graph datasets, executing complex queries or traversing large scale graphs has shown to be impractical. This is particularly true when a single system is used with limited memory, creating a bottleneck in the processing of large volumes of data. Therefore, to process such data, there is a need to distribute it and execute queries in a distributed manner with a high degree of parallelism. In contrast with the centralised systems, distributed architectures are characterized as higher processing capacity systems with large aggregated memory size. There are several

distributed architectures and approaches, of which we discuss the relevant ones to our work.

2.1 Hadoop-based approaches

To process big data, hadoop and its ecosystem have become the *de-facto* standard both in academia and industry. In the context of RDF graphs, there have been numerous works (e.g., [27,32,47,48]) done on RDF data management using this platform. Surveys such as [30,75] talk about such approaches in details. Many of them, for example [51,4] in path querying, follow the MapReduce paradigm and use HDFS [54,27] as their underlying data structure to store RDF triples. In the MapReduce paradigm, when a query is issued against such systems, HDFS files are explored and scanned to find adjacent nodes, which are then concatenated using the MapReduce join implementation (for more details see [39]). Approaches which follow the MapReduce paradigm differ mostly with respect to how the RDF data is stored and accessed from HDFS files, and how many MapReduce jobs are used.

The MapReduce programming model The MapReduce programming model was introduced by Google [13]. This model is scalable, and fault-tolerant and new nodes can easily be added to the cluster. Furthermore, the data distribution and parallel processing of the jobs is handled automatically. Developers do not need to take care of all these steps, therefore, they do not face the classical problems of data distribution (e.g., synchronization, network protocol, and etc.) The advantages and benefits of this programming model has led to a large number of applications built using it.

However, for most of the users this programming model is too low level: It does not provide a declarative syntax to execute the tasks, and therefore, even a simple computation has to be configured, and programmed, as Map and Reduce steps. To overcome these limitations, there have been a number of works done [45,43,28,62] to provide declarative query mechanisms on top of the MapReduce process.

In the context of path queries, Martin et al [51] proposed an approach named RDFPath; a path query processing method on large RDF graphs with MapReduce. Their approach is inspired by XPath [8] and designed according to the MapReduce model. A query in RDFPath is composed of a sequence of *location steps* where the output of the ith location step is used as input of the $(i+1)^{th}$ location step [51]. Conceptually, more edges and nodes are added by each location step to the intermediate path and this path can be restricted by applying filters. However, in this approach, the distance between two nodes and shortest paths are only partially supported. Further, RDFPath only supports paths of a fixed maximum length.

2.2 NoSQL-based approaches

Beside the hadoop ecosystem, there has also been work done related to RDF graph query processing in HBASE [7]. HBASE is a NoSQL distributed data store. The work in [32] provides a criterion to store and query RDF triples in HBASE. It is based on some permutations of subjects, predicates and objects which build indices that are then stored in HBase. Trinity [64] is an other example of NoSQL-based systems. Trinity is built on a distributed memory-cloud, and model RDF data in its native format (i.e., entities as node and relationships as edges). It allows SPARQL queries to be executed in an optimized way, and also support graph analytics (e.g., reachability) on RDF data. Survey papers [60, 11] highlight some prominent NoSQL databases, their characteristics and usage for storing and querying RDF data.

Trinity [77] is also claimed to support random walks, regular expression and reachability queries. Distributed NoSQL graph databases such as Neo4J [67] and Stardog [66] are optimized for graph navigation features or path querying.

2.3 Partition-based approaches

The partition-based approaches [50, 49, 18, 22, 46, 38, 26, 37] partition a large RDF graph into several fragments and distribute those fragments at different sites. Each remote site hosts its own centralized RDF storage and querying mechanism. To answer a posed query, it is first decomposed into multiple subqueries which are distributed across the sites. Each site answers the received query locally, and results from different sites are then aggregated.

In [38], an RDF graph is partitioned into n fragments, and each fragment is extended by including the N hop neighbors of boundary vertices. Their partitioning strategy restricts query processing in such a way that each decomposed subquery cannot be larger than N.

Partout [18] is an engine to query graph data which uses its own partitioned strategy. It extends the concepts of minterm predicates and uses the results of those predicates as its fragment units.

Lee et. al [26, 37] in their approaches proposed a "vertex block" concept where they define a partition unit as a vertex and its neighbors. They adopted a set of heuristic rules for distribution of those vertex blocks. A query is transformed into blocks and these blocks can be executed among all sites in parallel and without any communication.

TriAD [22] adopted METIS [31], a multilevel graph partitioning technique to divide the RDF graph into many partitions. Each partition is considered as a unit and distributed among different sites. At each site, TriAD maintains six large, in-memory vectors of triples, which correspond to all subject-predicate-object permutations of triples. It also constructs a summary graph to maintain the partitioning information.

[7] http://hbase.apache. org

2.4 Federated SPARQL query systems

SPARQL queries when run over multiple distributed heterogeneous SPARQL endpoints are considered federated queries. In the context of Linked Data, different repositories may be interconnected, and therefore provide a virtual integration of multiple data sources. A common technique in query federation is the precomputation of metadata for each SPARQL endpoint. Based on the metatdata, the original query is decomposed into subqueries in a way that each subquery is dispatched only to its relevant endpoint. On receiving answer from each subquery, the results are aggregated or joined together to answer the original query.

In [53], the metadata correspond to a service description that tells which triple patterns (i.e., predicate) can be answered. SPLENDID [20] is a federated system that uses Vocabulary of Interlinked Datasets (VOID) as the metadata schema for describing endpoints. HiBISCuS [57] relies on capabilities to compute the metadata. For each source, HiBISCuS defines a set of capabilities which map the properties to their subject and object authorities.

The systems above are categorised as *index-based*, i.e. they rely on the availability of an index of metadata about the federated endpoints. In contrast to these, FedX [63] does not require any index or preprocessed catalogue, but it sends ASK requests to collect the metadata on the fly. Based on the results of ASK requests, subqueries of the original query are dispatched towards the relevant endpoints.

Saleem et al [56] provide an extensive, fine-grains evaluation of existing query federated engines. They show the impact of different factors such as: number of sources selected, number of ASK requests, etc. They conclude that source selection time significantly affect the overall query performance. However, their proposal was for standard SPARQL queries not for the property paths. In this paper, we have introduced source selection mechanisms for distributed path computation.

2.5 Vertex-centric approaches

There have been many distributed graph processing systems [34] developed to efficiently analyse large graphs. We focus only those [19, 40, 76] which support navigational features. These systems are built on top of a cluster of low-cost commodity PCs and are deployed with a shared-nothing distributed computing paradigm.

Google's Pregel [40] is an example which adopts a vertex-centic computing paradigm. In-contrast with MapReduce, the Pregel-based approaches are inspired by *bulk synchronous parallel* models [70] and are more fault tolerant, scalable and efficient.

Different approaches (e.g., [44, 74, 73]) have adopted Pregel's architecture for distributed path or reachability queries. Nolé et al [44] for example proposed an evaluation technique according to the Pregel-based vertex-centric model.

In their approach, at each step of the computation, each vertex derives an input query according to the symbol labeling of outgoing edges and propagate derivative queries to its neighbours.

In [74,73], the authors proposed a distributed Pregel-based approaches named *DP2RPQ* to answer provenance-aware *regular path queries* Regular Path Queries (RPQs) using *Glushkov automata*. In their approach, at each superstep, one hop of edges in the path of the RDF graph is matched forward to obtain intermediate partial answers. Also they designed different optimization strategies e.g., reduce the vertex computation cost and edge-filtering, to improve the overall performance of computation.

2.6 Link traversal approaches

As explained previously, Hadoop, Partitions, and Vertex-centric computation are considered to be *homogeneous* infrastructures. However, Linked Data by nature is based on heterogeneous distributed systems, exposed by different data providers using different systems, each holding data based on requirements external to the distributed query processing approach. Consequently, a focus on a paradigm called Linked Traversal has emerged in this context, which makes use of the characteristics of Linked Data. In order to execute a given Linked Data query, for example a path query, live exploration systems perform a recursive URI lookup process during which they incrementally discover further URIs that also qualify for lookup [68,23]. Such live explorations provide valid results in the context of reachability queries. In contrast to Federated query approaches 2.4 where a source selection mechanism is required, the important characteristic of live exploration is that query execution do not require any *a priori* information about the distributed, remote data sources being queried.

Although Linked Traversal approaches seem to be fully compliant with the Linked Data principles, traversing live links may produce incomplete results. Furthermore, these approaches are not designed for high runtime performance [68].

2.7 Partial evaluation

The concept of partial evaluation has been used in many applications ranging from compiler optimization to distributed evaluation of functional programming languages [29]. In recent years, partial evaluation has also been used for evaluating queries on distributed tree-structured data such as XML, and graphs [7,10,17,71]. Authors in [7,10] introduced the concepts of partial evaluation in XPath [8] queries on distributed XML structured data. In their work, XPath queries are serialized to a vector of subqueries. Their approach finds partial answers for all subqueries at individual sites by using a top-down [9] or bottom-up [6] traversal, in a topological order, over the XML tree. Finally, all partial answers received from each site are assembled together at

the server. However, in contrast to XML tree structured data, RDF data and query language (i.e., SPARQL) are based on graphs. It is not feasible to serialize SPARQL queries and traverse the RDF graph in a topological order.

Partial evaluation on graphs has been considered in prior works [17, 71]. Wang et al. in [71] proposed a method for answering regular path queries (RPQs) on large-scale RDF graphs using partial evaluation. They partitioned RDF data and distributed it in a cluster, and used a dynamic programming model to evaluate in parallel the local and partial results of the RPQ on each computing site. To assemble these partial answers they designed and implemented an *automata-based* algorithm to produce complete answers.

3 Preliminaries

In this section we define the definitions and relevant background along with a running example that we will use throughout the paper to better understand the proposed approach.

Definition 1 *RDF Graph: An RDF graph G is a directed graph (V, E) where the set of edges E is represented by a set of RDF triples. Nodes V can be IRIs I, blank-nodes B or literals L. An RDF Triple $t := (s; p; o)$ is an element of the set $(I \cup B) \times I \times (I \cup L \cup B)$.*

In an RDF graph (V, E), if there exists an RDF triple (edge) between nodes v_i and v_j (i.e. a triple of the form (v_i, p, v_j)), then v_j is said to be a *successor* for v_i and v_i is a *predecessor* of v_j.

Definition 2 *Path: A Path $P(n_1, n_k)$ between two nodes n_1 and n_k within a graph $G = (V, E)$ is a sequence of nodes and relations $n_1 \xrightarrow{p_1} n_2 \xrightarrow{p_2} ....,$ $\xrightarrow{p_{k-1}} n_k$ such that for each step i, there exists a property p_i connecting v_i to v_{i+1} through an RDF triple $(v_i; p_i; v_{i+1})$ in E. Obviously, a single triple $T := (s; p; o)$ can also be viewed as a one hop path $s \xrightarrow{p} o$ between s and o.*

Definition 3 *Partial Path: A Partial Path $P'(n_1, n_k)$ is a path $P(n_1, n_i)$ which does not include n_k. In other words, it is a path starting at the source node, but which has not reached the target node. A path $P(n_1, n_k)$ is therefore referred to in this paper as a complete path.*

Definition 4 *Source Datasets: A graph $G_i = (V_i, E_i) \in \mathcal{D}$ is a source dataset in the context of a path query $PQ(s, t, K)$ if $s \in V_i$ and there is at least one triple in E_i with s as subject.*

Definition 5 *Reachability: In a graph $G = (V, E)$, if there exists at least one path between s and t we say that s and t are reachable and we denote it by $s \rightsquigarrow t$. Reachability can be expressed as a boolean value i.e.,:*

$$s \rightsquigarrow t = \begin{cases} true, & if\, \mathcal{P}s, t \neq \emptyset \\ false, & otherwise \end{cases}$$

Definition 6 *Direct Paths: A direct path is a path where at most two datasets are involved. For instance, $D1$ or $D2$ contributed to calculate the first three paths shown in Figure 1.*

Definition 7 *Indirect Paths: An indirect path is a path involving more than two datasets. For instance, $D1$, $D2$ and $D3$ contribute to calculate the 4th and 5th paths shown in Figure 1.*

The coverage of a class C in an RDF dataset D, denoted by $CV(C)$, is defined as follows [14]:

Definition 8 (Class Coverage) *Let D be a dataset. Moreover, let $P(C)$ denote the set of distinct properties of class C, and $I(C)$ denote the set of distinct instances of the class C. Let $I(p, C)$ count the number of entities for which property p has its value set in the instances of C. Then, the coverage of the class $CV(C)$ is*

$$CV(C) = \frac{\sum\limits_{p \in P(C)} I(p, C)}{|P(C)| \cdot |I(C)|}$$

In general, RDF datasets comprise multiple classes with a varying number of instances for different classes. The authors of [14] proposed a mechanism that considers the weighted sum of the coverage $CV(C)$ of individual classes. For each class C, the weighted coverage is defined below.

Definition 9 (Weighted Class Coverage) *The weighted coverage for a class C denoted by $WTCV(C)$ is calculated as:*

$$WTCV(C) = \frac{|P(C)| + |I(C)|}{\sum\limits_{C' \in D} |P(C')| + |I(C')|}$$

By using Definitions 8 and 9, we are now ready to compute the structuredness of a dataset D.

Definition 10 (Dataset Structuredness) *The overall structuredness or coherence of a dataset D denoted by $CH(D)$ is defined as*

$$CH(D) = \sum_{C \in D} CV(C) \cdot WTCV(C)$$

Definition 11 (Predicate Relationship Specialty) *Let d be the distribution that records the number of occurrences of a* relationship predicate r *associated with each resource and μ is the mean and σ is the standard deviation of d. The specialty value of r denoted as $\kappa(r)$ is defined as the Pearson's Kurtosis value of the distribution d.*

$$\kappa(r) = \frac{n(n+1)}{(n-1)(n-2)(n-3)} \cdot \frac{\sum_{x_i \in d} (x_i - \mu)^4}{\sigma^4} - \frac{3(n-1)^2}{(n-2)(n-3)}$$

Where n is the number of available values, i.e., sample size. The relationship specialty of a dataset is defined in the form of a weighted sum of specialty values of all relationship predicates:

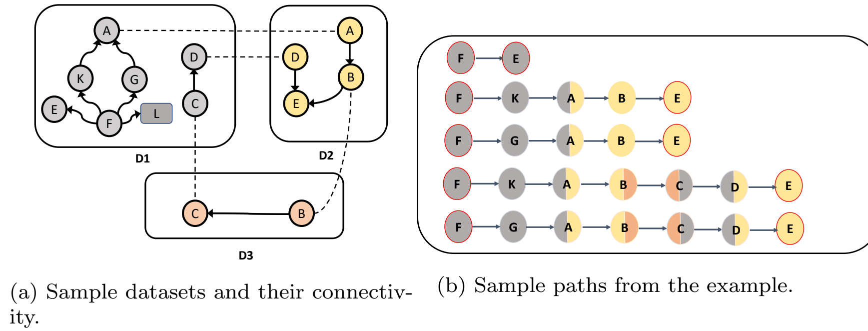
Definition 12 (Dataset Relationship Specialty) *The relationship specialty of dataset D denoted by $RS(D)$ is calculated as follows:*

$$RS(D) = \sum_{r_i \in R} \frac{|T(r_i)| \cdot \kappa(r_i)}{\sum\limits_{r_j \in R} |T(r_j)|}$$

where $|T(r_i)|$ is the number of triples in the dataset having predicate r_i, $\kappa(r_i)$ is the specialty value of relationship predicate r_i.

Running example

Figure 1a shows three fictional datasets distributed across the network and their connectivity. Suppose a user poses a path query PQ (ref. Listing 1) to find five paths (i.e., $K = 5$) between a source F and target in these datasets. As shown in Figure 1a, dataset $D1$ contains only a single path between F and E. However, dataset $D1$ is connected to other datasets (i.e., $D2$, $D3$) via some links. To fulfil the K requirement, the distributed path finding engine needs to exploit these links and finds partial paths from remote datasets and rearranges them to produce the complete final paths. Figure 1b depicts the generated paths that involve different datasets.



(a) Sample datasets and their connectivity.

(b) Sample paths from the example.

Fig. 1: Running example: Datasets and K paths between node F and node E.

Listing 1: Extended property path query

```
PREFIX : <http://insight-centre.org/sample/>
PREFIX ppfj: <java:org.centre.insight.property.path.>
SELECT * WHERE
{
?path ppfj:kPaths (:F  :E  5) .
}
```

Our defined path query PQ is denoted by $PQ = (s_i, t_i, K)$, where $s_i \in I$ is the source node, $t_i \in I$ is the target node, and $K \in \mathbb{Z}^+$ is the number of paths to be retrieved. The result set of such a query PQ on a graph G is the set $[\![PQ]\!]_G$ of up to K paths connecting s_i to t_i in G.

For the purpose of describing how DpcLD works, we additionally rely on the notion of partial paths(Definition 3). DpcLD relies on computing partial paths in individual graphs, and reconnecting them in order to form complete paths across the overall federated graph. We denote as $[\![PQ_G$ the set of partial paths $P'(s_i, t_i)$ resulting from the path query PQ in the graph G.

Formally, considering a set of federated, local graphs $\mathcal{D}$, the objective of DpcLD is to compute the set of paths $[\![PQ]\!]_\mathcal{G}$ where $\mathcal{G}$ represents the union of the graphs $G_i \in \mathcal{D}$. To simplify, we will also refer to this result set as $[\![PQ]\!]_\mathcal{D}$, substituting the set of graphs $\mathcal{D}$ with its union $\mathcal{G}$.

4 DpcLD

DpcLD is a path query processing engine that computes K paths between two nodes from distributed RDF graphs while communicating with a shared algorithm running on remote endpoints (triple stores). The DpcLD architecture is summarised in Figure 2, which shows its core components, i.e., (i) Source Selection: performs source selection and selects a relevant datasource to start the traversal from, (ii) Federated Path Computation: once a candidate datasource is selected, DpcLD starts path traversal, interacts with the Cache, and dispatches the queries to remote triple stores where it is required, and finally (iii) Paths RDFizer: breaks down (triplize) all retrieved paths (i.e., complete or partial paths) and stores into a temporary graph where a Bidirectional-BFS pathfinding algorithm computes the K paths.

In summary, posing a path query, the DpcLD engine delegates the requests to the data sources. The instances of the shared algorithm 2, running on remote endpoints, compute the paths (full or partial) against each query request and return the answers back to engine. The DpcLD engine, on receiving these answers, intelligently computes and generates the complete paths and finally the results are presented to the user.

4.1 DpcLD: Algorithm

To evaluate a path query PQ over distributed datasets $G_i \in \mathcal{D}$, the DpcLD engine not only finds the path set $[\![PQ]\!]_{G_i}$ within each graph, but also partial paths $[\![PQ_{G_i}$. Before we explain the DpcLD system in details, it is important to understand what type of cases the system has to cope with for a given path query $PQ = (s, t, K)$:

case 1: if $s \in G_i \wedge t \in G_i$, i.e. both the source and target nodes are within the local graph G_i, and the number of local paths satisfy K, then these will be directly pushed to the solution list and the algorithm will be terminated.

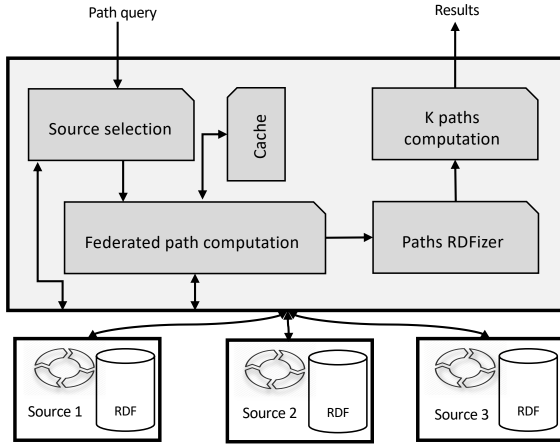


Fig. 2: DpcLD's high level architecture.

case 2: if $s \in G_i \wedge t \in G_i$, but the local paths do not satisfy the required K, then the algorithm will query the other graphs in $\mathcal{D}$ for (complete and partial paths).

case 3: if $s \in G_i \wedge t \notin G_i$, i.e. the source node is in the local graph, but not the target node, then the algorithm will query the other graphs in $\mathcal{D}$ for partial paths.

The pseudocode in Algorithm 1, shows how a path query $PQ = (s, t, K)$ is processed over the set of graphs $\mathcal{D}$. The steps performed are described in the following subsections.

4.1.1 Source Selection

DpcLD performs a partially index-free source selection of relevant graphs (triple stores, see Definition 4), where only the address URI of the endpoint for each graph is stored. The source selection of relevant data sources is performed by selecting the dataset in which the source node s has more outgoing links. For instance, in our running example, we assume a set of graphs $\mathcal{D} = G_1, \ldots, G_n$, where each graph G_i is a dataset accessible through a SPARQL endpoint. When a user poses a path query $PQ = (\text{:F}, \text{:E}, K)$, the DpcLD engine probes the relevant datastore in all datasets (Lines 3-9 of Algorithm 1). Figure 1a shows that only dataset $D1$ contains the source node :F. Therefore DpcLD, in this case, considers $D1$ as a local graph and starts computation on this dataset. We formalise the relevant data sources for s in Definition 4.

4.1.2 Federated Path Computation

This component computes the paths, communicates with the cache, and dispatches requests where required. The cache here plays a vital role in terms of

Algorithm 1: Pseudo-code of the algorithm implemented by DpcLD.

```
   Input  : A query Q = (s, t, K) over 𝒢
   Output : The answer set ⟦Q⟧_𝒢
1  𝒟 = {D_1, ..., D_n} ;                              /* index of data sources URIs */
2  D_l ← D_1;
3  max_oeg ← 0;
4  foreach D ∈ 𝒟 do
5  |  outgoing ←findOutGoingEdge(s, D);
6  |  if |outgoing| > | max_oeg | then
7  |  |  max_oeg ← outgoing;
8  |  |  D_l ← D
9  end
10 sol← ∅ ;                              /* solution (retrieved complete paths) */
11 q ← {T_node(s, null, null)} ;  /* initialise q as T_node which contains triple(tp) binary
     relations as transitive closure */
12 while q ≠ ∅ do                                   /* check until queue is empty */
13 |  tp ← q.poll()                          /* take triple tp from current T_node  */
14 |  for (tp_i)_{i=1}^n ∈ D_l do     /* iterate over all (p; o) of s for current triple tp */
15 |  |  if (tp(o)_i = literal) ∨ (isVisted(tp(o)_i) = true) then    /* if o is literal or
       current triple tp_i | (s; p; o) is already visited then skip */
16 |  |  |  continue;
17 |  |  if (tp(o)_i = t) then                                /* if target node found */
18 |  |  |  P← path(tp, tp(p; o)_i) ;    /* append tp(p; o) to the tp as a complete path
          */
19 |  |  |  sol←P ;                                           /* solution is found */
20 |  |  |  if |sol| ≥ K then
21 |  |  |  |  return sol;       /* if K is satisfied then terminate the algorithm
             (i.e., case:1) */
22 |  |  |  end
23 |  |  |  CacheCheck(P, 𝒟);      /* check the cache and perform necessary tasks */
24 |  |  else
25 |  |  |  P′ ← path(tp, tp(p; o)_i) ;    /* append tp(p; o) to the tp as a partial path
          */
26 |  |  |  q← q.add(tp_i);                                       /* update queue */
27 |  |  |  CacheCheck(p′, 𝒟);      /* check the cache and perform necessary tasks */
28 |  end
29 end
30 tempModel← RDFizer(Cache.getPartialAndfullPaths);   /* when Case:1&2 are not satisfied
     and local computation is finished */
31 sol← BidirectionalBFS(s, t, K, tempModel); /* run a bidirectionalBFS algorithm on temp
     Graph and computes the K paths */
32 return sol ;                                              /* return solutions */
33 Function CacheCheck(path, 𝒟)
34 |  foreach dataset D ∈ 𝒟 do
35 |  |  foreach (path_node)_{node=1}^n ;   /* iterate over each node of a path (P or P′)
        */
36 |  |  do
37 |  |  |  visited ← Cache.check(node, D) ;    /* check each node of path if visited
          against current dataset */
38 |  |  |  if visited = true then
39 |  |  |  |  continue;
40 |  |  |  else
41 |  |  |  |  p_r ← FedRequest(node, t, D) ;             /* obtain remote path through
             federated request */
42 |  |  |  |  PathStatus(p_r, D)
43 |  |  end
44 |  end
45 Function PathStatus(pathLst, D)
46 |  foreach path ∈ pathLst do
47 |  |  if path.startNode=s ∧ path.endNode=t then      /* check if current is a full
       path P */
48 |  |  |  P ← path;
49 |  |  |  Cache.put(D, BrkPath_Into_Nodes(P),P)
50 |  |  else
51 |  |  |  P′ ← path;
52 |  |  |  Cache.put(D, BrkPath_Into_Nodes(P′),(P′))
53 |  end
54 Function BrkPath_Into_Nodes(path)           /* break path into indvisual nodes */
55 |  nodes← {∅} ;
56 |  foreach (path_node)_{node=1}^n ;     /* iterate over each node of a path (P or P′) */
57 |  do
58 |  |  nodes.Add(node); ;         /* each node is added to the visited nodes list */
59 |  end
60 |  return nodes;
```

performance and reducing the number of HTTP requests. The DpcLD engine using cache avoids sending duplicate requests. We implemented the cache as a *key-value pair* store, where every dataset $D \in \mathcal{D}$ is a key and has multiple values (*visited nodes*, *full paths* P, *partial paths* P') against each key.

In our running example, the traversal starts (Line 10 of Algorithm 1) and the queue q is initialized with an object T_{node} which at the beginning contains a triple with only a source node :F as subject. The object T_{node} holds paths for each node as a transitive closure, i.e., it contains the *incoming edge, the current vertex,* and its *previous vertex (predecessor).* For instance, given a path query PQ with a source node :F, the traversal starts from :F and iterates over each of its triples/successors $T_i := (current\ vertex|s;p;o)$ (Line 14). The algorithm checks two conditions: (1) if any successor i.e., o is not a URI (i.e., a *literal*), and (2) if o is already *visited* for :F as a predecessor (to check for *cycles*). If any of these conditions is true, the particular iteration T_i will be skipped and the algorithm *continues* (Lines 15-16) to the next iteration. For example, a connected object :L to the source node (:F→L) is a literal, as depicted in Figure 1a. Thus, it will be skipped in the iteration $i = 1$.

Other than the previous two conditions explained above, the algorithm navigates along all the paths *(successors)* of :F (Lines 17-27) and checks the reachability (Definition 5) between source :F and target :E node.

In our running example, at iteration $i = 2$, the algorithm finds a complete path P (i.e.,:F→:E) within the local dataset $D1$, and the situation in this iteration represents the case 2. The algorithm does not terminate at iteration $i = 2$ but performs the following steps and updates the cache against the dataset $D1$:

- stores the path P (:F→:E) in the full path set.
- gets all the nodes (i.e., :F and :E) from the current path and put them in the *visited nodes* set against $D1$.
- *in parallel* checks the cache to see if these node are visited against other datasets. If the cache does not have these nodes visited for other datasets, it sends requests (i.e., Q *(:F,:E)*) to remote datasets (i.e., $D2, D3$, see Line 41).

Since the current path contains only two nodes, one request PQ *(:F,:E)* is dispatched to each remote datasets. It is notable that multiple requests will be dispatched in those cases where more than two nodes exist within a path. At current iteration $i = 2$, we can see that remote datasets did not return any paths, however, we still treat :F, :E as visited nodes against each dataset $D2, D3$ and store them as (*visited nodes*) in the cache. Table 1 shows the cache storage when iteration $i = 2$ is completed. The set of *visited nodes* is an important factor to minimise the number of remote requests because each element (i.e., visited node) stored in this set against individual datasets will not be counted for future requests, hence, reducing the network cost.

Before we explain the next steps, we can see that only one path has been found and the solution still does not satisfy the K parameter (i.e., 5). Therefore, the algorithm continues to next iterations. When the algorithm encoun-

Table 1: Cache at $i = 2$.

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,E]	[(F→E)]	[]
D2	[F,E]	[]	[]
D3	[F,E]	[]	[]

ters the iterations $i = 3$ and $i = 4$, then it has to cope with the situation that falls under the case 3, where only the source node :F does exist within the dataset $D1$. The algorithm (Line 25) finds two incomplete paths, at $i = 3$ (F→K→A) and at $i = 4$ (F→G→A) respectively, in local datasets $D1$. We denote such paths as *partial paths* $P\prime$ which do not reach target :E from source :F.

After iteration $i = 3$ is completed, the cache is updated and the partial path (F→K→A) along with the *visited nodes* are stored against key $D1$. In *parallel*, the algorithm constructs the path queries and delegates those to remote triple stores *i.e.*, $D2, D3$. The way the queries are generated is discussed in the following subsection.

4.1.3 Query Generation

As explained earlier, the number of generated queries directly depends on the nodes in a path. For example, for the path (F→K→A), the possible queries could be; (i) *PQ(:F,:E)*, (ii) *PQ(:K,:E)*, and (iii) *PQ(:A,:E)*.

However any node that is already visited, against relevant datasets $D \in \mathcal{D}$, will not be considered in the query generation. Moreover, while constructing a query, where *source* and *target* are the same (e.g., *(:E,:E)*), the algorithm will not generate the corresponding query. These two checks improve the query performance significantly by reducing the number of unnecessary remote requests.

It is also important to note while generating the requests that each query PQ will always have the *same* target:E.

For path (F→K→A) at iteration $i = 3$, the algorithm checks in the cache (Line 37) which nodes already exist against relevant datasets in $\mathcal{D}$. We can see that :F, :E are there against all datasets. Therefore, the query *PQ(:F,:E)* will not be generated and only two queries, i.e., *PQ(:K,:E)* and *PQ(:A,:E)*, are constructed and dispatched to datasets $D2$ and $D3$ (not $D1$ since :K and :A are already visited in $D1$). After receiving the responses from remote datasets (i.e., $D2$ and $D3$), the path status (Line 42) is checked and the cache is updated, where a partial path (A→B→E) against $D2$ is returned and no path against $D3$. The same procedure, as for $i = 3$, is followed by Algorithm 1 at iteration $i = 4$ where, for the path (F→G→A), only one query *PQ(:G,:E)* is constructed and dispatched to $D2$ and $D3$. The queries for node :F, :K, :A are not generated since these nodes are already visited against all datasets. For query *Q(:G,:E)*,

both $D2$ and $D3$ return no result. However, their *visited node* sets are updated with :G in the cache. After the completion of $i = 3$ and $i = 4$ respectively, the updated cache contains what is shown in Table 2.

Table 2: Cache at $i = 3$ and $i = 4$.

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,K,G,A,E]	[(F→E)]	[(F→K→A), (F→G→A)]
D2	[F,K,G,A,B,E]	[]	[(A→B→E)]
D3	[F,K,G,A,E]	[]	[]

In our running example, when the iteration $i = 4$ is finished, the local traversal for $D1$ is terminated because there is no more possible paths P or $P\prime$ from :F to its triples/successors $T_i := (current\ vertex|s; p; o)$ within $D1$.

Until now, the values *full path* or *partial path* in the cache included only direct paths, i.e., paths leading from $D1$ to $D2$ and $D3$. We explain the notion of direct path in (Definition 6). However, in many cases the paths, could be *indirect paths* (Definition 7), as shown in Figure 1a where $D1$, $D2$ and $D3$ participate to complete the 4th and 5th paths in Figure 1..

When local traversal is finished at dataset $D1$, however, we can get only 3 paths, (1, 2, and 3) (see Figure 1), and these are the direct paths where $D1$ and $D2$ contributed. We still miss the desired $K = 5$ solutions. How the remaining 2 paths are calculated is explained in the following paragraph.

After the iteration $i = 4$ when local traversal has finished, the algorithm does not terminate but recursively starts checking the *difference between sets of visited nodes* $A \Delta B = \{x : [x \in A \text{ and } x \notin B] \text{ or } [x \in B \text{ and } x \notin A]\}$ from one dataset to every other datasets within the cache $\mid PC \mid$. If any non-overlapping node – visited against one dataset but not for others – is found in a particular dataset then it is also checked against other unexplored datasets. For instance, in Table 2, node :B is in the *visited nodes* list against $D2$, but not against $D1$ and $D3$. So, when node :B is checked against these remaining datasets, i.e., $D1$ and $D3$, we get updated values in cache as shown in Table 3.

Table 3: *Cache update when :B is checked against* $D1$ *and* $D3$.

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,K,G,A,B,E]	[(F→E)]	[(F→K→A), (F→G→A)]
D2	[F,K,G,A,B,E]	[]	[(A→B→E)]
D3	[F,K,G,A,B,E,C]	[]	[(B→C)]

In the previous step performed for Table 3, we can see, in dataset $D3$, that node :C in the *visited nodes* list is not yet checked against other datasets. When node :C is checked against datasets $D1, D2$, we get an updated list as shown in Table 4.

Table 4: *Cache update when :C is checked against* $D1$ *and* $D2$

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,K,G,A,B,E,C,D]	[(F→E)]	[(F→K→A), (F→G→A),(C→D)]
D2	[F,K,G,A,B,E,C]	[]	[(A→B→E)]
D3	[F,K,G,A,B,E,C]	[]	[(B→C)]

As shown in Table 4, when the cache is updated, the node :D would appear in the difference between dataset $D1$ and others. When :D is checked for paths against $D2, D3$, the cache is updated as shown in Table 5. When all of the *visited nodes* appear for all datasets, Algorithm 1 stops checking cache and is terminated.

Table 5: *Cache update when :D is checked against* $D2$ *and* $D3$*.*

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,K,G,A,B,E,C,D]	[(F→E)]	[(F→K→A), (F→G→A),(C→D)]
D2	[F,K,G,A,B,E,C,D]	[]	[(A→B→E),(D→E)]
D3	[F,K,G,A,B,E,C,D]	[]	[(B→C)]

It is important to note that *visited nodes* in cache are only those that contribute to complete P or partial $P\prime$ paths, and therefore include a subset of the graph only.

4.1.4 Paths RDFizer:

When the main algorithm 1 is terminated, all the paths are broken down into triples by our *path rdfizer algorithm*. This is a simple mechanism, since in a path every node is connected to other node through one hop, like a triple (i.e., $T := (s; p; o)$) and we break a path in such a way that every object o becomes the subject s for a triple with the next connected node as object. For example, a path *(F→K→A)*, when triplified, will generate two triples: (i) $F \xrightarrow{p_1} K$, and (ii) $K \xrightarrow{p_3} A$.

An excerpt shown in Listing 2 represents the RDFized (N-Triples) data for *partial paths p'* retrieved in Table 5.

```
<http://node-C> <http://property-p6> <http://node-D> .
<http://node-A> <http://property-p7> <http://node-B> .
<http://node-F> <http://property-p4> <http://node-G> .
<http://node-F> <http://property-p1> <http://node-K> .
<http://node-K> <http://property-p3> <http://node-A> .
<http://node-D> <http://property-p6> <http://node-E> .
<http://node-B> <http://property-p7> <http://node-E> .
<http://node-B> <http://property-p8> <http://node-C> .
<http://node-G> <http://property-p5> <http://node-A> .
```

Listing 2: N-Triples format of paths given in Table 4.

A local traversal algorithm (we presented in [61]) is executed on this temporary graph G_{tmp} and all the *complete paths* p returned by this algorithm are added into the *solution* set. In the end, the query shown in Listing 1 returns the *answer set* $[\![Q]\!]_{\mathcal{D}}$ to the user with the $K = 5$ results.

4.2 Shared Algorithm

In the previous section, we explained that the DpcLD engine communicates with a shared algorithm. This algorithm (Algorithm 2) is distributed and deployed on remote SPARQL endpoints, to calculates the *paths*, either complete P or partial $P\prime$. It is implemented as a standard Breadth First Search (BFS) process with some modifications.

Lines 1-4 are the inputs for the algorithm. At Line 6 a *queue* q of type T_{node} is initialised with the start node s. The T_{node} object stores the triples in a chain such that the predecessor of each current triple can be accessible. At Line 6, the first time when Algorithm 2 is started, there will not be any predecessor for the source node s. At Line 8 the current triple is pulled from the *queue* q and at Lines 9-24 the algorithm traverses each vertex or the *object* o of that particular triple. At Line 10, the algorithm checks and continues to the next iteration if the *object* value of the current (s,p,o) triple is a *literal* or is already *visited*. At Line 13, if the leading *object* value is the actual target t, the algorithm adds it to the solution. At Line 15, the algorithm checks and continues to the next iteration if the *queue* already contains this triple in any of the triple's *predecessor* or *successor*. Line 15 is an extra check which allows only to store distinct paths and, therefore, restricts the unnecessary increase in the *queue* size. At Line 17 the calculated paths $P\prime$ are also stored into the solution list. At Line 19, if the size of the solution list, which contains both P and $P\prime$, becomes greater than K, Algorithm 2 terminates and returns the solution (i.e., complete P and partial $P\prime$ paths) towards the DpcLD engine.

5 Evaluation

In this section, we explain the evaluation setup, the synthetic and real-world datasets used, the queries and the results of our experiments.

Algorithm 2: Algorithm to find K paths locally between source and target datasets.

```
1  s← source ;                                              /* source node */
2  t← target ;                                              /* target node */
3  k← K_paths ;                                /* search number of TopK paths */
4  D← G ;                                                      /* dataset */
5  sol← ∅ ;                                /* solution (retrieved properties) */
6  q← {T_node(s, null, null)} ;  /* initialise q with Object T_node which can store
     current triple(tp) and all its predecessors */
7  while q ≠ ∅ do                                 /* check until queue is empty */
8  |  tp ← q.poll()                             /* take tp as current T_node object */
9  |  for (tp_i)_{i=1}^n do /* iterate over each child for a specific triple object
   |    */
10 |  |  if (tp(o)_i = literal) ∨ (isVisted(tp(o)_i)) then            /* move next */
11 |  |  |  continue;
12 |  |  if (tp(o)_i = t) then
13 |  |  |  sol ← path(tp, tp(p; o)_i) ;              /* append tp(p; o) to the tp as a
   |  |  |    complete path P  */
14 |  |  else
15 |  |  |  if (q.contains(tp_i) then  /* move next if queue q already contains
   |  |  |    this triple  */
16 |  |  |  |  continue;
17 |  |  |  sol ← path(tp, tp(p; o)_i) ;  /* append tp(p; o) to the tp as a partial
   |  |  |    path P/ */
18 |  |  |  q← q.add(tp_i)
19 |  |  |  if sol.size ≥ K then
20 |  |  |  |  return sol;
21 |  |  |  end
22 |  end
23 end
```

5.1 Experimental Setup

In this section, we discuss the datasets, the set of path queries over the selected datasets, the performance metrics we used in the evaluation, and the state-of-the-art path finding systems over Linked Data, we used in our evaluation.

5.1.1 Datasets

We used both *synthetic* and *real-world* RDF datasets in our evaluation. The *synthetic* datasets[8] were selected from the ESWC-2016 shortest path finding challenge and $real - world$ datasets were selected from biological data[9] provided by DisGeNET.

Datasets metrics. Before going in to the details, first we briefly explain the different RDF datasets metrics that are important to be considered while designing RDF benchmarking [15, 58]. For each dataset, we present total triples,

[8] https://bitbucket.org/ipapadakis/eswc2016-challenge/downloads/

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

subjects, predicates, objects, graph structuredness and the relationship specialty, explained as follow.

- Triple:
  A triple is an atomic entity, which consists a set of three entities i.e., subject, predicate, object. We have defined a triple in Definition 1
- Data Structuredness: This metric measures how well the concepts or classes (i.e., rdf:typ) are covered by different instances within the dataset. The value of structuredness lies between [0,1], where 0 stands for lowest possible structure and 1 indicates to a highest possible structured dataset. Duan et al. [15] in their paper concluded that synthetic dataset are highly structured in nature while real-world datasets's structuredness varies from low to high, covering the whole structuredess spectrum. Structuredness is considered one of the important matircs for RDF dataset benchmarking [15,1,55,58]. We have defined structuredness in Section 10.
- Relationship Specialty: It is often that some attribute within a datset are more common and have more associations with other resources. Moreover, some attributes have more then one values, e.g., a person entity within a dataset can have a multiple values for same property, for example, cell phone or professional skills. The number of occurrence of a property associated with each resource provides useful information of an RDF graph structure, and make some resources distinguishable from others [52]. The relationship specialty is commonplace in real datasets. For instance, a movie can be liked by several million people. Likewise, a research paper can be cited in several hundred other research papers. Qiao et al. [52] suggested that synthetic datasets are limited in how they reflect this relationship specialty. This could be due to the simulation of uniform relationship patterns for all the resource, or a random relationship generation process. We have defined relationship specialty in Definition 12.

Tables 6-8 show this statistical information both for synthetic and real-world datasets used in our evaluation. In general, we can see a good variation in these metrics, required for selecting RDF datasets for benchmarking [15,1,55,58]. However, as opposite to Duan et al. [15] conclusion, we can see that the structured values of the synthetic datasets are smaller as compared to real-world datasets. This is due to the fact that we have partitioned the complete synthetic dataset into 4 sub-datasets, therefore the completeness, structuredness, and relationship specialty of these datasets are affected.

Synthetic Datasets: The ESWC-2016 shortest path finding challenge provides both training and evaluation datasets. The synthetic dataset comprises of the information from DBPedia knowledge base which is built based on the information extracted from Wikipedia. The benchmark is based on query-log mining, clustering and SPARQL feature analysis [10] .

[10] https://aksw.org/Projects/DBPSB.html

- Training Dataset: The *training* data corresponds to the 10% transformation of the *DBpedia SPARQL Benchmark* [42] and comprises 9, 996, 907 triples, structuredness value with 0.445, and relationship specialty 874.491. To evaluate our approach (i.e., in distributed settings), we divided this data into 4 equal parts, producing 4 different graphs, based on the order in which the triples were provided in the overall dataset. Based on our experience, the 4 datasets represent a reasonable trade-off between the size of the individual graphs and the overhead generated by communication. Table 6 presents the statistics about this data.
- Evaluation Dataset: This dataset corresponds to 100% of the DBpedia SPARQL Benchmark and comprises 124, 743, 858 triples, structuredness value with 0.2695, and relationship specialty 1715.845. We also divided this data into 4 different graphs. Table 7 presents statistics about the evaluation data.

Table 6: Training datasets statistics.

Dataset	Triples	Subject	Predicates	Objects	Structuredness	Specialty
TDataset1	2925457	313036	7109	639743	0.087	937.62
TDataset2	2296074	92078	8536	1100275	0	333.15
TDataset3	2347536	95619	8135	1124389	0	407.73
TDataset4	2427840	97422	8267	1150104	0	460.98

Table 7: Evaluation datasets statistics.

Dataset	Triples	Subject	Predicates	Objects	Structuredness	Specialty
EDataset1	30740699	680727	18541	10157624	0.35	13.76
EDataset2	31892427	723768	17001	9948302	0.33	1256.11
EDataset3	30687403	686672	14119	9318528	0.33	782.15
EDataset4	31423329	674252	14175	9665835	0.33	737.20

Real-world Datasets: As mentioned before, the $real - world$ datasets are chosen from our previous evaluation [41] and are provided by DisGeNET. DisGeNET is a platform of datasets containing one of the largest publicly available collections of *genes* and *variants* associated with human diseases. The datasets involved are: *Disease, hpoClass, doClass, phenotype, Protein, Variant, Gene, and pantherClass*. We chose these datasets because they are highly interlinked and contain shared resources. The data comprises of 7, 265, 423 triples. Table 8 shows the statistics for each of the individual datasets.

5.1.2 Queries

Along with datasets, the ESWC-2016 shortest path finding challenge also provides four different path queries with different $source$ and $target$ and K values.

Table 8: Real-world datasets statistics.

Dataset	Triples	Subject	Predicates	Objects	Structuredness	Specialty
Disease	738626	60130	12	489756	0.656	317.38
doClass	101	21	11	63	0.524	3.281
Gene	1056346	119522	12	834502	0.655	13536.64
hpoClass	253	36	11	151	0.717	4.48
pantherClass	272	40	9	123	0.760	5.65
Phenotype	83292	8441	8	66249	0.998	1014.12
Protein	160537	14635	8	117034	0.941	1939.11
Variant	5225996	708405	16	3628674	0.930	118.89

These queries are available from Task 1[11] of the challenge. In our evaluation, we used these four queries. For the *real-world* data, we used the 12 path queries used in the evaluation of our previous work [41]. These queries were carefully chosen such that they show variation in terms of the number of possible paths between *source* and *target*, the number of distributed datasets contributing to the paths, and the length and complexity of the paths.

5.1.3 Evaluation Settings

In distributed computing, the network cost can be one of the key factors in the performance evaluation. For this reason we used two evaluation settings in our experiments:

- **Remote Setting With Network Cost:** To compare the distributed path finding systems, we loaded each RDF dataset (i.e., *synthetic, real-world*) into multiple Fuseki servers (version 1.3.0 2015-07-25T17) deployed on different physical machines with the following specifications: Ubuntu OS with 2.6GHz Intel Core i5 processors, 16GB 1600MHz DDR3 of RAM, and 500GB of storage capacity hard disks. It is important to note that each Fuseki server is integrated with our algorithm 2 and provides a public SPARQL endpoint to be queried remotely using SPARQL over HTTP requests.
- **Local Setting With Negligible Network Cost:** This represents the same settings as above, except that the four instances of the Fuseki server were started on the same machine with different ports for SPARQL endpoints. Since, all of the four instances of the Fuseki server were running on the same machine, the network cost is negligible.
- **Cache size:** The configuration of the DpcLD allows to set the maximum size of the cache depending on available RAM. We have set the cache limit to 2GB in our evaluation setup to allow making the maximum use of the memory available[12].

[11] https://bitbucket.org/ipapadakis/eswc2016-challenge
[12] In our stress testing, for a highly complex path query with over 400K distributed paths, the cache size reached 1.5GB.

5.1.4 Comparison

We selected state-of-the-art RDF path finding techniques based on the following criteria: (1) open source and configurable[13], (2) able to find K paths in *distributed RDF datasets*, (3) scaleable to medium-large datasets, such as DBpedia in our case, (4) no licensing issue of publishing benchmarking results, and (5) able to show/enumerate complete paths from source to target node. Based on this criteria, we selected three – TPF [12], QPPDs [41], and ESWC2016_Winner [25] – state-of-the-art approaches that are able to find K paths in *RDF datasets*, which are loaded into triplestores with SPARQL endpoints. Table 9 shows the existing approaches including DpcLD that meet the above selection criteria.

The performance metrics we used in our evaluation are: (1) the path computation time (in seconds), and (2) the memory consumption during the paths computation.

Table 9: Systems that support RDF and path queries

System	Support		
	Single-graph	Distributed-graphs	K-Paths
DpcLD	✓	✓	✓
TPF [12]	✓	✓	✓
QPPDs [41]	✓	✓	✓
ESWC2016_winner [25]	✓	✗	✓

5.2 Performance Analysis

In this section, we compare the runtime performance of DpcLD, both with centralized and distributed path finding approaches for RDF datasets.

5.2.1 Comparison with Distributed Approach

As shown in Table 9, QPPDs and TPF are the two approaches that support finding the K paths in distributed *RDF datasets*. We compared DpcLD on exactly the same benchmarks used in the QPPDs and LDF evaluations.

DpcLD vs QPPDs In QPPDs's evaluation, the aforementioned 8 $real-world$ datasets were used. Each dataset was loaded into a dedicated Fuseki triplestore with a SPARQL endpoint. We used the same settings in both approaches. Figure 3 shows the runtime comparison of our approach with QPPDs

[13] We have explicitly asked authors for the availability of their code and/or data used in the evaluation

on 12 benchmark queries. As an overall performance evaluation, our approach is clearly faster than QPPDs on all of the 12 benchmark queries. The average (over all 12 queries) runtime of DpcLD is 4.0 seconds, while QPPDs took 17.2 seconds on average, leading to a performance improvement of greater than 400%. The main reason for this performance improvement is the lesser number of distributed path requests sent by our approach compared to QPPDs. Furthermore, DpcLD makes use of the cache to avoid sending duplicate requests while that is not controlled in QPPDs. In Q1, Q4, Q9, and Q10, we noticed that the results contain the large number of nodes connectivity (i.e., path hops, as well as high degrees for different nodes involved in those paths) as compare to the other queries results. To process those paths, we observed that the QPPDs sends more requests for such complex queries while on the other hand, the DpcLD keeps record of visited nodes and does not send the duplicate requests.

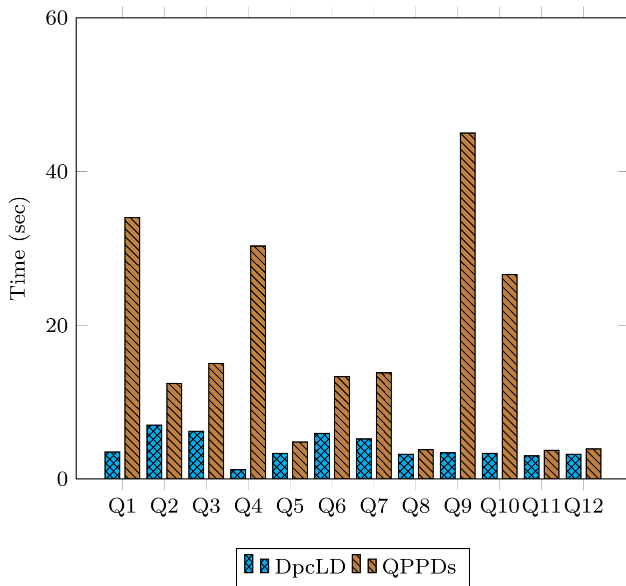


Fig. 3: Response time per query with DpcLD vs. QPPDs.

DpcLD vs TPF We compared our approach to the results presented in the paper about TPF [12]. To have a fair comparison, we used machines for DpcLD with the same specifications as was used for the evaluation of TPF. Figure 4 depicts the comparison between DpcLD and the TPF approach in terms of response time. We outperformed TPF by several orders of magnitude. In general, the TPF servers only perform executing single triple patterns SPARQL queries. The load of the query execution is distributed among TPF client and

server, thus ensuring high availability with a slight performance loss. While on the otherside, DpcLD does not allow to send the duplicates request and also the aggregated results are returned from remote endpoints against each query request which is not in the case of TPF. Therefore, DpcLD outperformed TPF in several order of magnitudes.

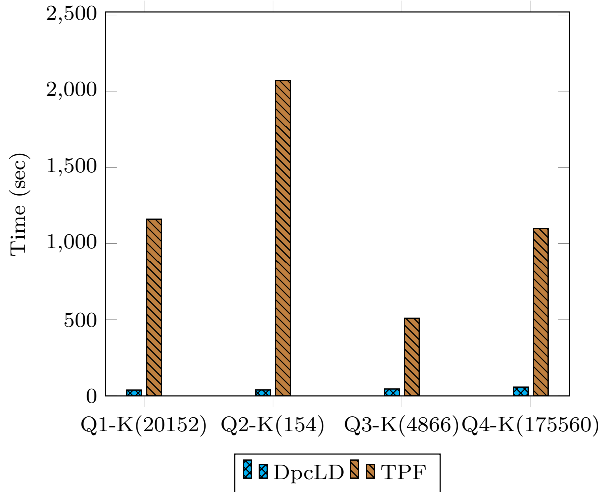


Fig. 4: Response time per query with DpcLD vs. TPF.

5.2.2 Comparison with centralised approaches

In this section, we compare our distributed DpcLD approach with a centralized path finding solution over RDF data. The goal of the comparison is to show how DpcLD scales as compared to centralized solutions. To this end, we compare DpcLD with the winner of the ESWC2016 challenge based on the datasets and queries used in the ESWC2016 challenge, i.e, the synthetic datasets (both training and evaluation) and the four queries Q1-Q4. We used both of the aforementioned evaluation settings (i.e., remote and local settings).

Training data results Figure 5 shows the comparison of the ESWC2016_winner with DpcLD when deployed both in local and remote settings. The results clearly suggest that the centralized solution is much faster for smaller K values shown in all four queries Q1, Q2, Q3 and Q4. On the other hand, the performance of DpcLD is rather slow for smaller K. The reason DpcLD performs slower with smaller K values is that it first collects the global partial paths and then performs the local processing.

However, the performance of ESWC2016_winner significantly drops when we increase the required K number of paths. This can be seen in Q1, Q2, Q3

that when we increased the K beyond 50000, the performance of ESWC2016_winner exponentially decreased. In Q4, we also noticed that the performance of ESWC2016_winner dramatically decreased even before $K = 30000$. This was because some of the nodes involved in the path processing were having high degrees, therefore ESWC2016_winner occupied to much memory and this became performance bottlneck for ESWC2016_winner. While, DpcLD scales better and perform linearly as compared to ESWC2016_winner as we increase K.

The results also suggest that DpcLD is approximately 4 times slower in remote settings as compared to the local settings. This means that network costs play an important role in the performance of DpcLD.

The runtime performance results shown in Figure 5 are highly correlated to the memory consumption by each of the systems shown in Figure 6. In general, DpcLD consumes less memory resources when deployed in remote settings, followed by DpcLD in local settings, and then ESWC2016_winner. The results suggest that the significant performance drop in ESWC2016_winner is due to the large amount of memory consumed, as we increase the value of K. On the other hand, memory consumption in DpcLD is more controlled as compared to ESWC2016_winner for large K values.

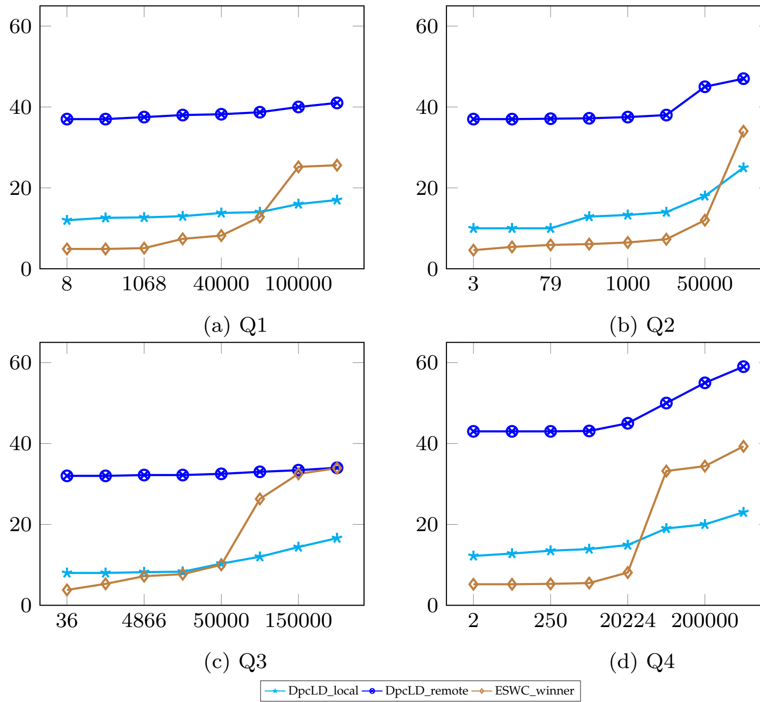


Fig. 5: Training datasets:- Response time per query (with different K paths) for DpcLD vs. ESWC2016_winner.

Figure 6 shows the memory consumed by each system. We noticed that DpcLD in remote settings outperformed both DpcLD_local and ESWC2016_winner in all four queries. This was due to the path computation being distributed on each remote machine instead of a single machine. In Q4 we also noticed that DpcLD_local unexpectedly used less memory when computing paths for $K >= 250$ as opposite to the trending line (i.e., with the increase of K the memory consumption should increase in theory). We concluded that the internal resource/memory allocation by CPU could be the reason.

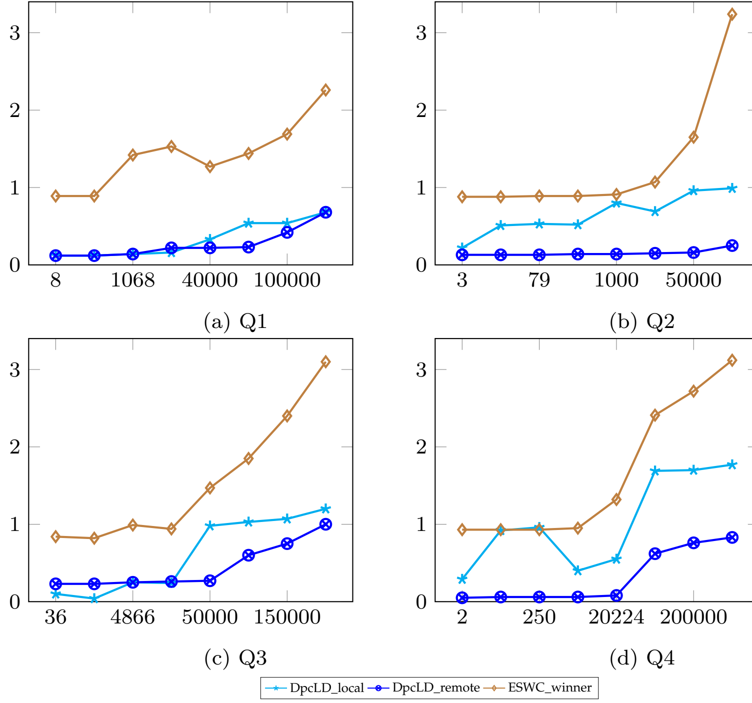
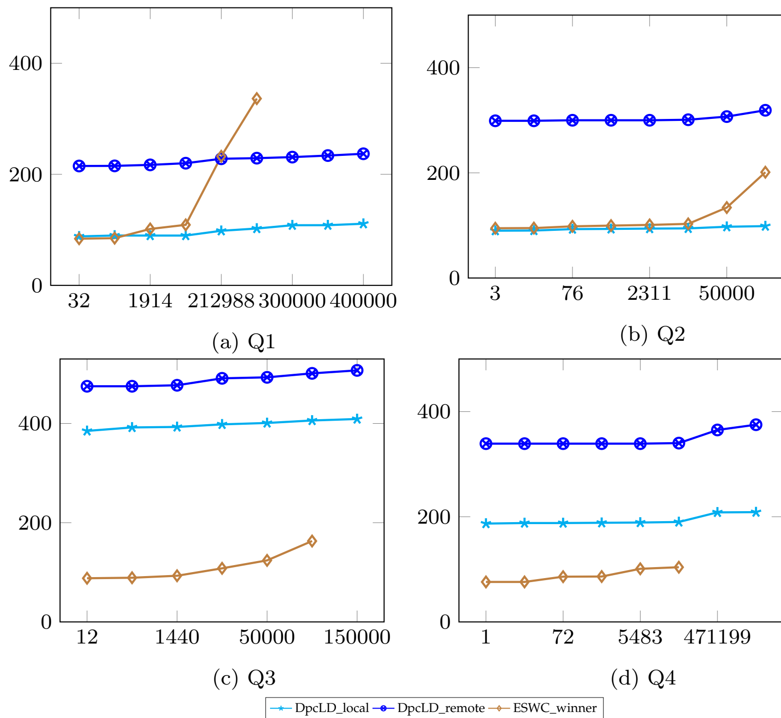


Fig. 6: Training datasets:- Memory consumption per query (with different K paths) for DpcLD vs. ESWC2016_winner.

Evaluation data results Figure 7 shows the runtime performance for the bigger dataset (i.e, 100% DBpedia) as compared to the training dataset (i.e, 10% DBpedia). In general, it can be seen that the performance of ESWC2016_winner degrades much faster, in all four queries, with the increase in values of K compared to the training data. On the other hand, the performance graph of DpcLD is almost linear with the increase of K value, confirming that DpcLD scales better as compared to ESWC2016_winner. For Q1 we set the maximum $K = 400,000$, where ESWC2016_winner was only able to retrieve paths up

to $K = 250,000$. After that limit, it started sending *out of memory* exceptions. DpcLD, on the other hand, was able to retrieve paths to the maximum K limit. For query Q2, it can be seen that DpcLD performed better than ESWC2016_winner throughout from the minimum to the maximum values of K. For queries Q3 and Q4, ESWC2016_winner outperformed DpcLD up to a certain path limits (i.e, around $K = 100,000$). However, it started sending *out of memory* exceptions after that limit. DpcLD was able to retrieve paths up to the highest tested values of K for both queries Q3 and Q4. The results suggest that with huge volumes of data and higher K values, the algorithms that work on a single graph may face performance issues or *out of memory* exceptions. This comparison shows that DpcLD could be applicable as an alternative choice to local traversal approaches when it comes to dealing with large amount of data. Figure 8 shows the memory consumed by each system over the evaluation data. In general, we noticed that DpcLD required about 8 times less memory than ESWC2016_winner. As already mentioned, ESWC2016_winner throws *out-of-memory* exceptions for Q1, Q3 and Q4 over large values of K. This shows that DpcLD can be a good candidate when finding paths over large distributed datasets using machines with low memory resources.



(a) Q1 (b) Q2 (c) Q3 (d) Q4

Fig. 7: Evaluation datasets:- Response time per query (with different K paths) over evaluation dataset for DpcLD vs. ESWC2016_winner.

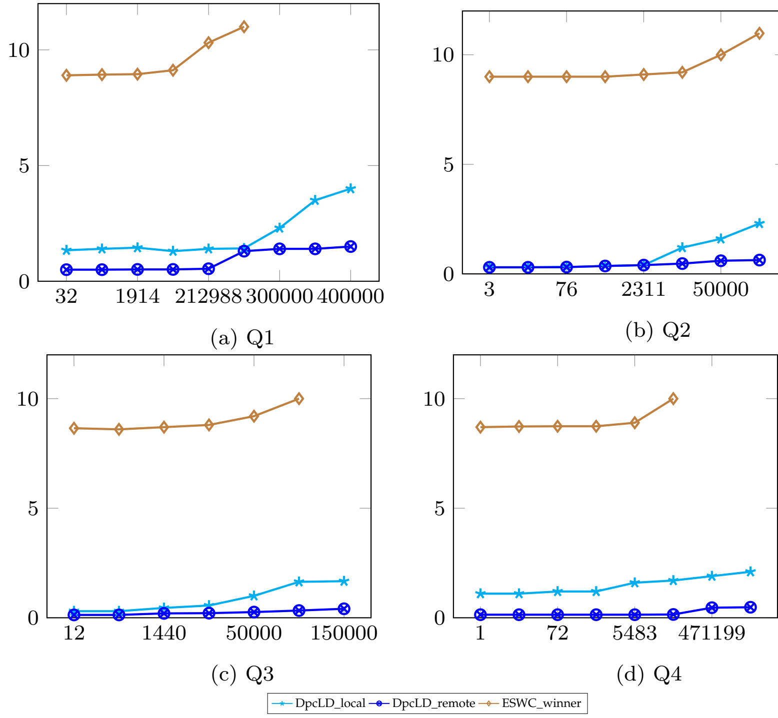


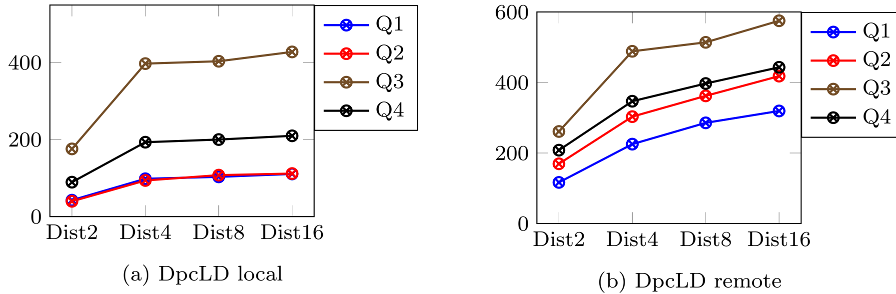
Fig. 8: Evaluation datasets:- Memory consumption per query (with different K paths) over evaluation dataset for DpcLD vs. ESWC2016_winner.

5.2.3 Scalability Analysis

We used the evaluation dataset to test the saclability of the proposed approach with respect to increasing number of partitions. For this purpose, we distributed the data into 2, 4, 8, and 16 distributions. We executed the same four queries that we previously used in the evaluation data. We wanted to test the scalability, therefore, we selected the maximum K value of individual query that is shown in Figure 8, i.e., for Q1 ($K = 400000$), Q2 ($K = 50000$), Q3 ($K = 150000$), and Q4 ($K = 200000$). We used both local and remote deployments explained in the previous section.

Figure 9 shows the response time for each query. As an overall scalability analysis, for all queries, we were able to get the required K paths within a reasonable amount of time (i.e. less than 10 minutes). In the local setting, our approach response time is almost linear even with increasing number of distributions. However for the remote setup, as expected, the response time is slightly increased with increasing number of distributions. This is because with more distributions the network cost was increased, and the response time is around 30% increased with with doubling the number of partitions.

For the local distributions setup, the increase in response time is marginal with increasing the number of distributions as shown in Figure 9. Finally, as we used synthetic data for partitioning and distribution, we noticed that with the increase of partitioning, our experimental data started to become a kind of random data as we discussed in section 5.2.4). This randomization of data resulted into increase in communication cost. In contrast to the synthetic data, the real-world datasets where each dataset represents a complete graph, our approach was more scalable even for remote experiments as we discussed in section 5.2.1 when we compared DpcLD with QPPDs.



(a) DpcLD local

(b) DpcLD remote

Fig. 9: Response time per query for different distributions

5.2.4 Randomized data and DpcLD performance

In this section, we show the result of testing DpcLD when the dataset is randomly distributed, rather than being divided in the order provided. In the original dataset, the triples are provided ordered by subject, meaning that triples with the same subject are more likely to end up in the same partition. The objective here is to show the extent of the impact of such a distribution on the performance of DpcLD.

We randomized the training data using the *shuf* command[14], partitioned it as before, and loaded it into 4 remotely running Fuseki servers. This represents an extreme case, unlikely to be encountered in practice, where triples are distributed in a way that does not follow any specific pattern. It therefore represents a “worst case scenario” for DpcLD, as a way of *torture testing* the system.

Figure 10 shows the response time of DpcLD to compute answers for each of the queries Q1-Q4 with the indicated K values on the training dataset (10% DBpedia). Compared to the results obtained with the same K values as depicted in Figure 4, the effect of random data distribution becomes obvious. This emphasises how distributed approaches like DpcLD do rely on data being distributed in a way that is meaningful and that supports balancing local

[14] https://shapeshed.com/unix-shuf/

and remote computations. It is worth mentioning here that QPPDs failed to respond when tested on the same setting, and that results are not available with randomised datasets for TPF.

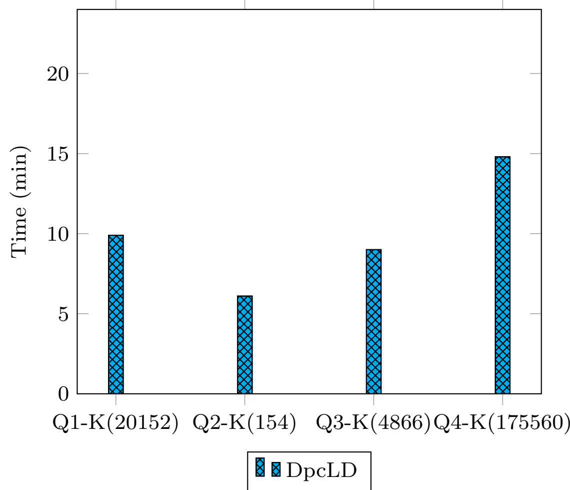


Fig. 10: Response time per query from DpcLD with randomized dataset.

6 Conclusion and Futurework

In this paper, we propose DpcLD, an engine for finding paths in distributed RDF datasets exposed as SPARQL endpoints. DpcLD is an index-free approach and hence does not require any pre-computations like QPPDs [41] for generating the index. Since DpcLD does not require an index, it has the capability of querying up-to-date data. DpcLD exploits and aggregates partial paths within a distributed environment to retrieve the required K paths. We used both synthetic and real-world datasets and compared the performance of DpcLD with distributed and centralized path finding approaches over RDF datasets. Furthermore, we used two different evaluation settings to measure the effect of the network cost on the runtime performance. The results suggest that DpcLD outperforms other distributed methods and scales better as compared to the best centralized approach for path retrieval over RDF datasets. The results suggest that DpcLD, when tested for local computation, could be applicable as an alternate choice to local traversal approaches when it comes to deal with large amounts of data with less memory.

Currently, the DpcLD engine communicates only with a “shared” algorithm running on remote endpoints and answering local path queries. However, with some modifications, databases that support SPARQL1.1 property paths (e.g.,

Virtuoso, Blazegraph, etc) could be adapted to achieve the same results, reducing the need for a specific shared algorithm. The extent of those adaptations and whether reference implementations could be provided for them is an interesting area to further explore. We plan to test DpcLD over larger *real-wolrd* datasets, e.g., bio2rdf. Finally, the current implementation of the DpcLD engine navigates only *reachability* queries, i.e., arbitrary paths between *source* and *target*. We plan to introduce the regex-based *regular* path queries.

Acknowledgment

This work is partly supported by a research grant from Science Foundation Ireland, co-funded by the European Regional Development Fund, for the Insight SFI Research Centre for Data Analytics under Grant Number SFI/12/RC/2289_P2. The work conducted in the University of Leipzig has been supported by the project LIMBO (Grant no. 19F2029I), OPAL (no. 19F2028A), KnowGraphs (no. 860801), 3DFed(Grant no. 01QE2114B), and SOLIDE (no. 13N14456)

References

1. G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of rdf data management systems. In *International Semantic Web Conference*, pages 197–212. Springer, 2014.
2. K. Anyanwu, A. Maduko, and A. Sheth. Sparq2l: towards support for subgraph extraction queries in rdf databases. In *Proceedings of the 16th international conference on World Wide Web*, pages 797–806. ACM, 2007.
3. M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st international conference on World Wide Web*, pages 629–638, 2012.
4. Y. Bai, C. Wang, and X. Ying. Para-g: Path pattern query processing on large graphs. *World Wide Web*, 20(3):515–541, 2017.
5. W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach. Lod laundromat: a uniform way of publishing other people's dirty data. In *International Semantic Web Conference*, pages 213–228. Springer, 2014.
6. P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 211–222, 2006.
7. P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 211–222. VLDB Endowment, 2006.
8. J. Clark, S. DeRose, et al. Xml path language (xpath) version 1.0, 1999.
9. G. Cong, W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 509–520, 2007.
10. G. Cong, W. Fan, A. Kementsietsidis, J. Li, and X. Liu. Partial evaluation for distributed xpath query processing and beyond. *ACM Transactions on Database Systems (TODS)*, 37(4):1–43, 2012.
11. P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, and M. Wylot. Nosql databases for rdf: an empirical evaluation. In *International Semantic Web Conference*, pages 310–325. Springer, 2013.
12. L. De Vocht, R. Verborgh, and E. Mannens. Using triple pattern fragments to enable streaming of top-k shortest paths via the web. In *Semantic Web Evaluation Challenge*, pages 228–240. Springer, 2016.
13. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
14. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156, 2011.
15. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156, 2011.
16. I. Ermilov, J. Lehmann, M. Martin, and S. Auer. LODStats: The data web census dataset. In *International Semantic Web Conference*, pages 38–46. Springer, 2016.
17. W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *arXiv preprint arXiv:1208.0091*, 2012.
18. L. Galárraga, K. Hose, and R. Schenkel. Partout: a distributed engine for efficient rdf processing. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 267–268, 2014.
19. Giraph Team. Apache giraph. http://giraph.apache.org.
20. O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *Proceedings of the Second International Conference on Consuming Linked Data-Volume 782*, pages 13–24. CEUR-WS. org, 2011.
21. A. Gubichev and T. Neumann. Path query processing on very large rdf graphs. In *WebDB*. Citeseer, 2011.

22. S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300, 2014.
23. O. Hartig. *Querying a web of linked data: foundations and query execution*, volume 24. Ios Press, 2016.
24. O. Hartig and G. Pirrò. Sparql with property paths on the web. *Semantic Web*, 8(6):773–795, 2017.
25. S. Hertling, M. Schröder, C. Jilek, and A. Dengel. Top-k shortest paths in directed labeled multigraphs. In *Semantic Web Evaluation Challenge*, pages 200–212. Springer, 2016.
26. J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.
27. M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1312–1327, 2011.
28. M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data intensive query processing for large rdf graphs using cloud computing tools. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 1–10. IEEE, 2010.
29. N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.
30. Z. Kaoudi and I. Manolescu. Rdf in the clouds: a survey. *The VLDB Journal*, 24(1):67–91, 2015.
31. G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing'95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, pages 29–29. IEEE, 1995.
32. V. Khadilkar, M. Kantarcioglu, B. Thuraisingham, and P. Castagna. Jena-hbase: A distributed, scalable and efficient rdf triple store. In *Proceedings of the 11th International Semantic Web Conference Posters & Demonstrations Track, ISWC-PD*, volume 12, pages 85–88. Citeseer, 2012.
33. K. J. Kochut and M. Janik. Sparqler: Extended sparql for semantic association discovery. In *European Semantic Web Conference*, pages 145–159. Springer, 2007.
34. D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.
35. E. V. Kostylev, J. L. Reutter, and M. Ugarte. Construct queries in sparql. In *18th International Conference on Database Theory (ICDT 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
36. V. Le Anh and A. Kiss. Efficient processing regular queries in shared-nothing parallel database systems using tree-and structural indexes. In *ADBIS Research Communications*, 2007.
37. K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment*, 6(14):1894–1905, 2013.
38. K. Lee, L. Liu, Y. Tang, Q. Zhang, and Y. Zhou. Efficient and customizable data partitioning framework for distributed big rdf data processing in the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 327–334. IEEE, 2013.
39. F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Computing Surveys (CSUR)*, 46(3):1–42, 2014.
40. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.
41. Q. Mehmood, M. Saleem, R. Sahay, A.-C. N. Ngomo, and M. D'Aquin. Qppds: Querying property paths over distributed rdf datasets. *IEEE Access*, 7:101031–101045, 2019.
42. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. Dbpedia sparql benchmark–performance assessment with real queries on real data. In *International semantic web conference*, pages 454–469. Springer, 2011.
43. J. Myung, J. Yeon, and S.-g. Lee. Sparql basic graph pattern processing with iterative mapreduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, pages 1–6, 2010.

44. M. Nolé and C. Sartiani. Regular path queries on massive graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2016.
45. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.
46. T. Padiya and M. Bhise. Dwahp: workload aware hybrid partitioning and distribution of rdf data. In *Proceedings of the 21st International Database Engineering & Applications Symposium*, pages 235–241, 2017.
47. N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st International Conference on World Wide Web*, pages 397–400, 2012.
48. N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H2rdf+ an efficient data management system for big rdf graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 909–912, 2014.
49. P. Peng, L. Zou, L. Chen, and D. Zhao. Adaptive distributed rdf graph fragmentation and allocation based on query workload. *IEEE Transactions on Knowledge and Data Engineering*, 31(4):670–685, 2018.
50. P. Peng, L. Zou, and R. Guan. Accelerating partial evaluation in distributed sparql query evaluation. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 112–123. IEEE, 2019.
51. M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, and G. Lausen. Rdfpath: path query processing on large rdf graphs with mapreduce. In *Extended Semantic Web Conference*, pages 50–64. Springer, 2011.
52. S. Qiao and Z. M. Özsoyoğlu. Rbench: Application-specific rdf benchmarking. In *Proceedings of the 2015 acm sigmod international conference on management of data*, pages 1825–1838, 2015.
53. B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In *European semantic web conference*, pages 524–538. Springer, 2008.
54. K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *Programming support innovations for emerging distributed applications*, pages 1–5. 2010.
55. M. Saleem, A. Hasnain, and A.-C. N. Ngomo. Largerdfbench: a billion triples benchmark for sparql endpoint federation. *Journal of Web Semantics*, 48:85–125, 2018.
56. M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web*, 7(5):493–518, 2016.
57. M. Saleem and A.-C. N. Ngomo. Hibiscus: Hypergraph-based source selection for sparql endpoint federation. In *European semantic web conference*, pages 176–191. Springer, 2014.
58. M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, and A.-C. Ngonga Ngomo. How representative is a sparql benchmark? an analysis of rdf triplestore benchmarks. In *The World Wide Web Conference*, pages 1623–1633, 2019.
59. S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2013.
60. L. H. Z. Santana and R. d. S. Mello. An analysis of mapping strategies for storing rdf data into nosql databases. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 386–392, 2020.
61. 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 *Proceedings of the 13th International Conference on Semantic Systems*, pages 97–103. ACM, 2017.
62. A. Schätzle, M. Przyjaciel-Zablocki, T. Hornung, and G. Lausen. *Pigsparql: Übersetzung von sparql nach pig latin.* Gesellschaft für Informatik eV, 2011.
63. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: a federation layer for distributed query processing on linked open data. In *Extended Semantic Web Conference*, pages 481–486. Springer, 2011.
64. B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013.

65. K. Shvachko, H. Kuang, S. Radia, R. Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.
66. E. Sirin. Stardog - a path of our own. https://www.stardog.com/blog/a-path-of-our-own/.
67. N. Team. Neo4j. https://neo4j.com.
68. J. Umbrich, A. Hogan, A. Polleres, and S. Decker. Link traversal querying for a diverse web of data. *Semantic Web*, 6(6):585–624, 2015.
69. A. Valdestilhas, T. Soru, M. Nentwig, E. Marx, M. Saleem, and A.-C. N. Ngomo. Where is my uri? In *European Semantic Web Conference*, pages 671–681. Springer, 2018.
70. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
71. X. Wang, J. Wang, and X. Zhang. Efficient distributed regular path queries on rdf graphs using partial evaluation. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1933–1936, 2016.
72. X. Wang, J. Wang, and X. Zhang. Efficient distributed regular path queries on rdf graphs using partial evaluation. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1933–1936. ACM, 2016.
73. X. Wang, S. Wang, Y. Xin, Y. Yang, J. Li, and X. Wang. Distributed pregel-based provenance-aware regular path query processing on rdf knowledge graphs. *World Wide Web*, pages 1–32, 2019.
74. Y. Xin, X. Wang, D. Jin, and S. Wang. Distributed efficient provenance-aware regular path queries on large rdf graphs. In *International Conference on Database Systems for Advanced Applications*, pages 766–782. Springer, 2018.
75. J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. Springer, 2010.
76. J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. Springer, 2010.
77. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013.

View publication stats