Scalable SPARQL Querying using Path Partitioning

Buwen Wu [†#], Yongluan Zhou [#*], Pingpeng Yuan [†*], Ling Liu [§], Hai Jin [†]
[†]SCTS/CGCL, Huazhong University of Science and Technology, Wuhan, China
[#]University of Southern Denmark, Denmark [§]Georgia Institute of Technology, Atlanta, USA
Email: [†]{wubuwen, ppyuan, hjin}@hust.edu.cn, [#]zhou@imada.sdu.dk, [§]lingliu@cc.gatech.edu

***Abstract*—The emerging need for conducting complex analysis over big RDF datasets calls for scale-out solutions that can harness a computing cluster to process big RDF datasets. Queries over RDF data often involve complex self-joins, which would be very expensive to run if the data are not carefully partitioned across the cluster and hence distributed joins over massive amount of data are necessary. Existing RDF data partitioning methods can nicely localize simple queries but still need to resort to expensive distributed joins for more complex queries. In this paper, we propose a new data partitioning approach that takes use of the rich structural information in RDF datasets and minimizes the amount of data that have to be joined across different computing nodes. We conduct an extensive experimental study using two popular RDF benchmark data and one real RDF dataset that contain up to billions of RDF triples. The results indicate that our approach can produce a balanced and low redundant data partitioning scheme that can avoid or largely reduce the cost of distributed joins even for very complicated queries. In terms of query execution time, our approach can outperform the state-of-the-art methods by orders of magnitude.**

I. Introduction

RDF (Resource Description Framework) [3] data model represents information in the form of triple statements: (subject, predicate, object). The high growth rate of big data and the simplicity and flexibility of RDF model have driven an increasing number of organizations storing their data in RDF format. The statistics from Linked Open Data Project show that more than 31 billion triples had been published till Sep. 2011 [1]. As the amount of RDF data continues to grow rapidly, it will soon exceed the processing capacity of a single machine. To achieve desirable performance for massive RDF data analysis, it is inevitable to employ a cluster of computing nodes to manage big RDF datasets.

While RDF data can generally be regarded as a long table with three columns in traditional relational databases, it can also be modelled as a graph structure with the subjects and objects of triples modelled as vertices and the predicates modelled as labelled edges. Due to a large number of common subjects and objects shared among the triples, many edges are incident to common vertices and hence an RDF graph often exhibits a complex structure. Figure 1(a) shows an example RDF graph, which is formed by using part of the RDF structure in the LUBM benchmark [11].

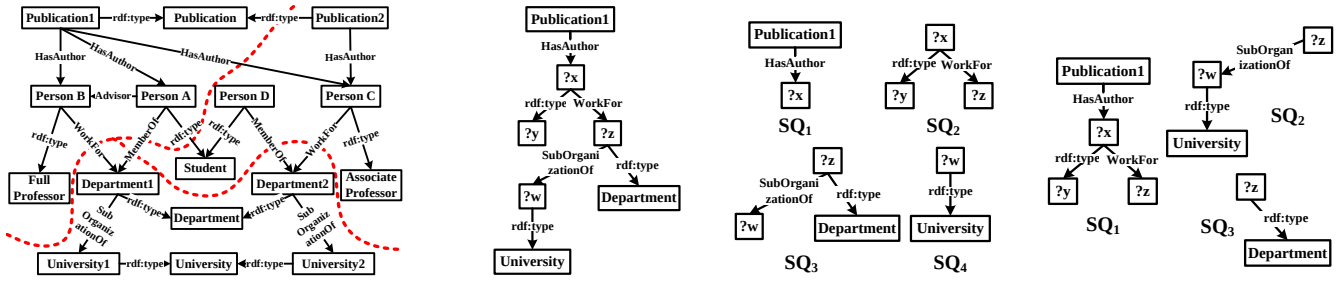
Correspondingly, queries over RDF graph specified using SPARQL [4], a standard RDF query language, can also be modelled as graph patterns to be matched with the RDF graph, which would also have complex and diverse structures. Example structures of query graphs include star, chain, tree and cycle etc. Figure 1(b) is an example SPARQL query graph over the RDF data in Figure 1(a). This query is to find the authors of $Publication1$, as well as the $Department$ and $University$ that the authors work in.

In a scale-out RDF data processing system, RDF data would be partitioned among the computing nodes and accordingly, a complex SPARQL query can be decomposed into subqueries which will be run on different computing nodes. Distributed joins may have to be performed over the intermediate outputs from the subqueries to produce the final query output. Such distributed joins are often expensive and require frequent interaction, communication and synchronization among the computing nodes, which would significantly diminish the benefit of adopting a scale-out system [15]. Therefore, a carefully designed RDF data partitioning scheme needs to consider both the complex structures of RDF graph and query graph and minimize the necessary number of expensive distributed joins.

A popular approach to partition RDF data is hash partitioning, which is adopted by a majority of the existing distributed RDF engines [13], [14], [18], [24]. This approach distributes RDF triples across different partitions by computing a hash key over either the subject or the object of each triple. Suppose the hash key is computed on the subject, then we can ensure that all triples sharing a common subject would be located on a single computing node. Therefore a complex query can be decomposed into a number of subqueries with a star shape, where each subject variable/constant in the query is the center of a star. For instance, the query in Figure 1(b) can be decomposed into the star subqueries shown in Figure 1(c). Take SQ_2 as an example. We are sure that all triples with the same person as the subject are hashed to the same partition and hence we can execute SQ_2 in parallel without the need for distributed joins. Therefore, each of these subqueries can be parallelized onto the computing nodes without interactions between the nodes. We call these subqueries *inner-partition* subqueries. However, to obtain the final output, we have to execute expensive distributed joins over the intermediate outputs produced from these subqueries, which could be quite many for a complex query.

Hash partitioning does not capture the rich structural information in the RDF graph and hence often miss opportunities to generate better data partitioning. In [15], the authors proposed using graph partitioning methods to optimize data partitioning. This approach initially uses a graph partitioner, e.g. METIS [2], to place vertices into different partitions. Then, for the boundary vertices in each partition, it will extend the partition by replicating the vertices that are within m-hop distances from

*corresponding authors: Pingpeng Yuan (ppyuan@hust.edu.cn), Yongluan Zhou (zhou@imada.sdu.dk)



(a) RDF Graph (b) SPARQL (c) With Hash Partition (d) With 1-hop Extension

Fig. 1: RDF graph, SPARQL query and query decomposition results with regard to two data partitioning methods: Hash Partitioning and Graph Partitioning with 1-hop Vertex Extension.

the boundary vertices and place them into the current partition. With such a partitioning method, we can decompose a query into inner-partition subqueries that contain paths with lengths up to $2m$. For instance, the Figure 1(b) can be decomposed into the subqueries illustrated in Figure 1(d) with 1-hop vertex extension. SQ_1 has paths whose lengths are equal to 2, which is the maximum path length in an inner-partition subquery with 1-hop vertex extension.

One can increase the value of m to minimize the necessary number of subqueries. For example, setting m to 2 in this example can reduce the number of subquery to 1. However a greater m would incur massive duplicate data. For example, the dotted lines in Figure 1(a) shows the partition boundaries set by applying graph partitioning to divide the vertices into 3 partitions. We can see if we extend each partition using 2-hop vertex extension, then each partition would nearly duplicate the complete graph. In this case, even though we have only one inner-partition subquery, all the three computing nodes have to perform duplicate computations over almost identical data and hence diminish the benefit of employing a scale-out system. Furthermore, if there exist a few high-degree vertices, a greater m would also incur high data skewness and hence render a few computing nodes becoming the bottleneck in parallel processing. A more recent work [19] is aimed at solving the problem of data skewness by replacing the step of graph partitioning with a hashing approach. However, the data duplication problem still remains due to the vertex extension.

In this paper, we argue that a data partitioner should not only consider the structure of the RDF graph, but also take into account the structure of the query graph, and propose a radically different partitioning framework to address the challenges. We introduce the notion of end-to-end path in RDF graph and use such path as the finest partition element. In this way, a complex query containing longer paths does not need to be decomposed into unnecessarily more number of subqueries. To reduce data redundancy and further lower the number of inner-partition subqueries that a complex query has to be decomposed to, we propose the technique of vertex merging, which attempts to collocate paths that have share vertices. With this new partitioning framework, the problem of optimizing data partitioning is transformed into choosing an optimal set of vertices to merge so that the query efficiency is maximized. In particular, we make the following major contributions in this paper:

- We propose a new RDF data partitioning framework, which adopts the end-to-end path as the basic partition element and employs vertex merging to combine paths into partitions. We formally formulate the balanced path partitioning problem under this new framework and proof the problem is NP-Hard and APX-Hard.

- In view of the hardness of the problem, we introduce a new version of the problem with a relaxed balance constraint. Then we propose an approximate algorithm that provides a performance guarantee.

- To enhance the efficiency, we also present two bottom-up path merging algorithms to partition the paths. The resulting data partitioning can localize many queries with complex structures, such as star, chain, cycle and tree, while maintaining low data duplication and data skewness.

- We propose a partition-aware query decomposition method to decompose a complex SPARQL query to minimize the possible cross-node communication. Our data partitioning method allows a complex query to be decomposed into fewer number of subqueries and hence be evaluated more efficiently.

- To perform a fair comparison with the state-of-the-art approaches [15], [19], we implement an experimental system by adopting a similar architecture as proposed in [15], [19], where each single node RDF store is powered by RDF-3X [22] and cross-node communication is implemented on a Hadoop platform. We conduct an extensive experimental study on LUBM [11] and SP^{2}Bench [25] benchmarks as well as a large real-world RDF dataset UniProt [5]. The results show that our method outperforms the previous approaches by up to two orders of magnitude, especially for complex queries.

II. Related Work

Hadoop based RDF data systems, such as [16], [23], [24] directly store RDF data as HDFS files, and distribute these files by using the file partitioning and placement policies in the vanilla Hadoop. However, previous studies [9], [17] showed that, without carefully designed data partitioning algorithms and data localization strategies, massive I/O cost and communication overhead would be incurred in these kind of systems.

A popular data partitioning algorithm for RDF data is hash partitioning [13], [14], [18]. This approach distributes RDF triples across different partitions by computing a hash key over the subject or the object of each triple. Hence, hash partitioning can work well for star queries, but for chain or more complex queries, its performance is inefficient. Huang et al. [15] use a graph partitioner, e.g. METIS [2], to place vertices into

partitions and extend each partition by replicating the vertices that are within m-hop distances from its boundary vertices. But this approach suffers from a skewed data distribution and a potentially large amount of data duplication. Wang et al. [26] propose a more efficient graph partitioner that is able to partition billion-node graphs. A more recent work [19] is aimed at solving the problem of data skewness by replacing the step of graph partitioning with hash partitioning. This method still cannot solve the data duplication problem. The partitioning algorithm in [27] uses rooted sub-graphs as the partition elements and a k-means clustering algorithm to allocate the rooted sub-graphs to the computing nodes. Instead of making an in-depth analysis of the RDF data partitioning problem, this work is mainly focused on designing an efficient distributed RDF engine.

Another research direction of RDF data partitioning is dynamic run-time data partitioning, which adapts the data partitioning scheme according to the run-time changes of system workload [28]. Our data partitioning algorithm can be used as the initial partitioning method in [28]. Moreover, the idea of using our coarse-grained partition element, end-to-end path, can be applied to the dynamic partitioning algorithms to further improve their performance.

Trinity.RDF [30] and TriAD [12] are probably the most recently proposed distributed RDF engines for web-scale RDF data. Trinity.RDF uses main memory to store the RDF data and hence can achieve very low data access latency. Instead of using joins, the authors proposed an efficient operator, namely graph exploration, to perform SPARQL queries based on the MPI protocol. TriAD uses a full set of $(subject, predicate, object)$ permutations as the local data index and provides a global indexing by the summary graph technique. To perform joins, it leverages multi-threaded and distributed executions based on an asynchronous MPI protocol. While Trinity.RDF and TriAD mainly focus on designing the scale-out systems, our data partitioning algorithm can be employed in these systems to further improve their performance by reducing the communication cost.

III. Preliminaries

A. RDF Graph and SPARQL Query

RDF data, a set of triples $(subject, predicate, object)$, can be represented as a graph according to the following definition.

Definition 1 (RDF Graph): An RDF graph is a directed labeled graph, denoted as $G = (V, E, L_E)$. V is a set of vertices, corresponding to all the subjects and the objects of the RDF triples. $E \subseteq V \times V$ is a set of directed edges from the subjects to the objects. L_E is a set of edge labels, referring to the predicates associated with the edges. A vertex with zero indegree is called a **source vertex** and a vertex with zero outdegree is called a **sink vertex**. □

A SPARQL [4] query Q can be modelled as a set of triple patterns. Each triple pattern tp is a triple that contains variables in the subject, predicate or object. Then Q can be denoted as $Q = \{tp_1, tp_2, ..., tp_m\}$, where tp_i $(1 \leqslant i \leqslant m)$ is a triple pattern. In fact, triple patterns are connected by shared subject or object and a join occurs on the shared subject or object between two connected triple patterns. Thus, the exemplary types of joins in SPARQL are subject-subject join (S-S join), subject-object join (S-O join) and object-object join (O-O join). In this paper, we do not consider predicate join, which is not very common as shown in a previous study [10].

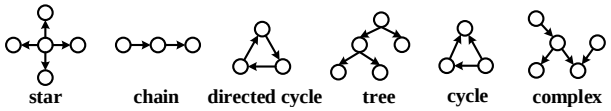


Fig. 2: Query Types

Based on the joins that a SPARQL query contains, we can classify queries into different categories: star query that only contains S-S joins, chain and directed cycle query that only contains S-O joins, tree queries that contains S-S join and S-O join and some more complex queries. Figure 2 gives some illustrating examples of different queries.

For distributed SPARQL query processing, there could be two kinds of query workload: **inner-partition query** and **cross-partition query**. If a query can be processed in parallel on each computing node without any cross-node data communication, then it is called an inner-partition query. Otherwise, the query has to be processed by joining distributed data from multiple partitions and it is called a cross-partition query.

B. End-to-End Path

Previous researches on RDF data partitioning usually use RDF triples or vertices as partition elements. As analyzed above, with these types of partition elements, complicated queries have to be decomposed into many inner-partition subqueries and hence incur many expensive distributed joins.

To reduce the number of inner-partition subqueries that a complex query has to be decomposed to, and hence to save the cost of distributed joins, we adopt end-to-end paths in an RDF graph as the partition elements, which is defined as follows:

Definition 2 (End-to-End Path): Let G be an RDF graph. A path $v_0e_1v_1e_2v_2...e_mv_m$ is called an end-to-end path if it satisfies all the following conditions: (i) v_0 is a source vertex or one of the vertices in a directed cycle that does not contain any vertex with incoming edges from vertices outside of the cycle, (ii) v_m is a sink vertex or there exists a vertex v_i in this path and $v_m = v_i$ (i.e. there exists a directed cycle). We call v_0 as the **start vertex** and v_m as the **end vertex**. □

Example 1: Figure 3(a) shows the collection of all end-to-end paths in an RDF graph. The start vertices include the three source vertices v_1, v_2 and v_3 as well as vertex v_6, one of the vertices in the directed cycle $v_6 \rightarrow v_7 \rightarrow v_{10} \rightarrow v_6$. Here we use IDs instead of URI and Literal for simplicity. □

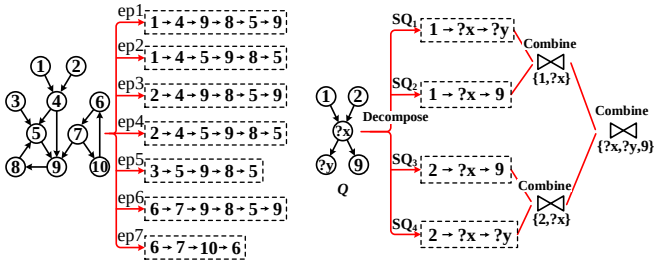
Hereafter, unless otherwise specified, we refer to *end-to-end paths* as *paths*.

Theorem 1: Given an RDF graph G, if it is decomposed into a set of paths according to Definition 2, then every vertex v and every edge (u, w) in G exist in at least one path. □

The proof is omitted here (see Appendix for details).

IV. Problem Formulation

In this section, we present the model of our new partitioning framework and a formal problem statement.



(a) End-to-End Paths (b) Query Decomposition

Fig. 3: Paths and Query Decomposition

A. Path Partitioning Model

The k-way path partitioning plan is defined as follows:

Definition 3 (k-way Path Partitioning Plan): Given an RDF graph $G{=}(V, E)$, a k-way path partitioning plan $\mathscr{P}$ over G is to divide all the end-to-end paths of G into k nonempty and disjoint partitions $\{P_1, ..., P_k\}$, where P_i contains an exclusive subset of end-to-end paths. □

A path partitioning plan has the following property.

Theorem 2: Given a path partitioning plan $\mathscr{P}$, all queries that only contain S-O joins (i.e. chain and directed cycle queries) are inner-partition queries. □

Proof: Suppose we have an RDF graph G and a query Q only contains S-O joins. Each subgraph $\mu(Q)$ of G that matches Q is either a general path or a directed cycle in G. Thus, there must exist at least one end-to-end path ep such that ep contains $\mu(Q)$. In other words, $\mu(Q)$ is a subgraph of end-to-end path ep. ■

In our path partitioning model, each path is assigned to exactly one partition and each partition contains a number of paths that may or may not have common vertices. Given a path partitioning plan, we can perform query decomposition by making use of the concept of merged vertex defined as follows:

Definition 4 (Merged Vertex): A vertex v is called *merged* if all paths that contain v are in the same partition. □

We will see how this concept can be used for query decomposition in the next subsection.

B. Query Decomposition Model

A query will be decomposed into a set of inner-partition subqueries. And the final query output can be calculated by joining the results of these subqueries. To reduce the cost of performing such expensive distributed joins, we need to reduce the number of subqueries as much as possible.

Note that a SPARQL query can be represented as a graph, which also consists of a set of paths. Therefore a query Q, can be first decomposed into a set of subqueries $Q{=}\{SQ_1, ..., SQ_m\}$, each of which is a chain or a directed query only contains S-O joins. As stated in Theorem 2, each SQ_i is an inner-partition query for any given path partitioning plan. If two subqueries are connected by shared vertices, it means there is a join between these two subqueries, denoted by $SQ_i \bowtie_{V_{i,j}} SQ_j$, where $V_{i,j}$ is the set of their shared vertices.

To further reduce the number of subqueries, we attempt to exploit the opportunity to combine some subqueries. Specifically, if the join between two subqueries can be performed in parallel without cross-node communication, then we can combine them into a new inner-partition subquery. Whether it can be done depends on the location of their shared vertices, which is stated in the following theorem.

Theorem 3: Given two inner-partition subqueries SQ_i and SQ_j that share a set of vertices $V_{i,j}$, the join between SQ_i and SQ_j can be evaluated locally, if there exists one vertex $v \in V_{i,j}$ such that all matching vertices of v in the RDF graph are merged. □

Example 2: Given the RDF graph in Figure 3(a) and the query Q in Figure 3(b), Q can be first decomposed into four subqueries: from SQ_1 to SQ_4. Let us look at SQ_1 and SQ_2. They have two shared vertices: a constant v_1 and a variable $v_{?x}$. Each of them has only one matching vertex in the RDF data shown in Figure 3(a), which is vertex v_1 and vertex v_4 respectively. Here, as long as one of them is merged in the partitioning plan, we can combine these two subqueries into a new inner-partition subquery. □

C. Metrics for Path Partitioning

To define the path partitioning problem, we have to define the metrics to measure the optimality of different path partitioning plans. In particular, we consider the following three metrics.

Balance. A well balanced data partitioning plays an important role in efficient query processing. A substantial skewed data partitioning would cause imbalanced query workload distribution. In addition, the overloaded partition may exceed the memory capacity of a single machine. Both would lead to a performance bottleneck in the cluster. Here, we use the number of the paths in the partition P_i to measure its load, denoted by $|L(P_i)|$.

Data Duplication. Note that paths are not independent and in fact, some paths may share common edges and vertices. If two paths share some edges and vertices and they are assigned to two different partitions, then there will be duplicated triples in the two partitions. The duplicated data will not only incur extra storage overhead, but also impose duplicated processing load and possibly produce duplicated outputs. Hence the number of duplicated triples has significant impact on processing cost and communication overhead. Formally, we use $Dup(\mathscr{P})$ to denote the duplicate ratio of a path partitioning plan $\mathscr{P}$, which is defined as:

$$Dup(\mathscr{P}) = \frac{1}{|E|} \sum\nolimits_{e \in E} (|\mathscr{P}(e)| - 1) \quad (1)$$

where $|\mathscr{P}(e)|$ denotes the total number of copies of e in the path partitioning plan $\mathscr{P}$.

Query-Efficiency. As stated earlier, minimizing the number of inner-partition subqueries that a query has to be decomposed to, will reduce the cost of performing distributed joins, thus enhance the query efficiency. According to Theorem 3, the more vertices that we can merge, the more subqueries that

we can combine. Formally, the set of vertices are merged is denoted by V_+. Then, the query efficiency depends on the number of vertices in V_+, denoted by

$$QE(\mathscr{P}) = |V_+| \tag{2}$$

D. Problem Statement

In order to find a path partitioning plan for efficient SPARQL query processing, we aim to make more queries as inner-partition queries while keeping low data duplication and balanced load distribution. In the following theorem, we relate the duplicate ratio to the number of merged vertices.

Theorem 4: $Dup(\mathscr{P})$ satisfies the following property:

$$Dup(\mathscr{P}) \leqslant \frac{(|V|-|V_+|)^2}{|E|}(k-1).$$

□

The proof is omitted (see Appendix for details).

According to Theorem 4, we find that, the duplicate ratio can be confined within a bound positively correlated to $|V_+|$, the number of merged vertices. In other words, maximizing $|V_+|$ would also reduce the duplicate ratio, which is consistent with our intuition. Therefore, we relax the constraint on data duplication and focus on maximizing $|V_+|$. The **$(k,1)$-balanced path partitioning problem**, denoted by $(k,1)$-BPP problem, is formally stated as follows:

Definition 5 ($(k,1)$-BPP Problem): Given an RDF graph G, find a k-way path partitioning plan $\mathscr{P}$ with the following objective functions:

$$\textbf{Maximize} \quad |V_+| \quad \textbf{s.t.} \quad |L(P_i)| \leqslant \lceil \frac{n}{k} \rceil, 1 \leqslant i \leqslant k$$

where $P_i \in \mathscr{P}$, and n is the number of paths in G. □

Theorem 5: The $(k,1)$-BPP problem is (1) NP-Hard, (2) APX-Hard. □

Proof: Proof sketch. The hardness of $(k,1)$-BPP problem is verified by a reduction from the balanced hypergraph partition problem, which is NP-Hard [20]. The APX-hardness of $(k,1)$-BPP problem is proofed by a reduction from the 3-Partition problem [20]. One can see Appendix for the details. ■

According to Theorem 5, the $(k,1)$-BPP problem has no polynomial time approximation algorithm with a constant approximation factor unless $P{=}NP$.

V. Approximate Algorithm

The $(k,1)$-BPP problem is hard to approximate. Hence, we introduce a problem with a relaxed balance constraint, namely $(k,2)$-balanced path partitioning problem, or $(k,2)$-BPP for short, which is to find a k-way path partitioning plan to maximize $|V_+|$ under the constraint that each partition contains at most $\lceil \frac{2n}{k} \rceil$ paths, where n is the number of paths.

We present a approximate algorithm for $(k,2)$-BPP problem with an assumption that the length of path (i.e. the number of distinct edges in this path) is bounded by a constant l. We will show that the algorithm gives a k-way path partitioning plan such that no partition contains more than $\lceil \frac{2n}{k} \rceil$ paths,

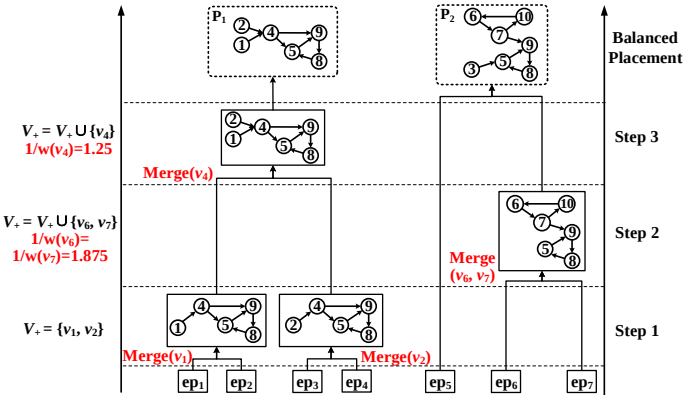


Fig. 4: Approximate Algorithm. The solid-line rectangle indicates the path group.

and the number of merged vertices is at least $(1{-}e^{-\frac{1}{kl}})|V_+^*|$, where V_+^* denotes the set of merged vertices of the optimum $(k,1)$-balanced path partitioning.

Our algorithm for solving $(k,2)$-BPP problem proceeds in two phases. In the first phase, we try to merge the vertices as much as possible. To merge a vertex v, we should combine all the paths passing v into a path group. In the meantime, we should keep the sizes of all the path groups less than or equal to $\lceil \frac{n}{k} \rceil$. Then, since the size of each path group is not more than $\lceil \frac{n}{k} \rceil$, we can pack these path groups into k partitions evenly, such that no partition contains more than $\lceil \frac{2n}{k} \rceil$ paths.

The approximate algorithm is presented in Algorithm 1. First of all, we introduce some notations used in the algorithm. l_{ep} denotes the length of a path ep. $\mathscr{E}(v)$ is the set of all paths passing v. The profit for merging a vertex is counted as 1, because mering a vertex increments $|V_+|$ by 1. The weight of merging a vertex is denoted as $w(v) = \sum_{ep \in \mathscr{E}(v)} \frac{1}{l_{ep}}$. A higher weight means merging this vertex might end up with a larger path group. Then profit-weight ratio of merging v is counted as $\frac{1}{w(v)}$, which is used to rank the vertices in the algorithm.

The algorithm starts by generating all the paths within an RDF graph. Here, the critical step is to generate the set of start vertices used to generate the paths we need. First, all the source vertices are included in the set. Since there could exist some vertices that are not reachable from any source vertex, (e.g. a directed cycle $v_6 \rightarrow v_7 \rightarrow v_{10} \rightarrow v_6$ in Figure 3(a)), we choose the vertex with the minimal ID in such a directed cycle as a start vertex (e.g. v_6). After generating the set of start vertices, we use a depth-first search algorithm starting from each start vertex to generate all the paths. For brevity, the details of this step is omitted in the pseudo-code .

Since no vertex has been merged yet, in the initial set of path groups is in the form of $Res{=}\{\{ep_1\},\{ep_2\},...,\{ep_n\}\}$ (line 3), i.e. each group contains one path. Then for each subsets of V that contain two vertices, we attempt to merge its two vertices. If the merging dose not make the size of any path group exceeds size constraint (line 5, $\max_{pg}(V_+)$ denotes the size of the largest path group after merging all the vertices in V_+), then we complete the merging solution by using a greedy heuristic. This is achieved by choosing vertices to be merged one by one from the remaining vertices in descending order of their profit-weight ratios (i.e. $\frac{1}{w(v)}$), while maintaining the

Algorithm 1: Approximate Algorithm

Input: a set of path $EP = \{ep_1, ..., ep_n\}$ and vertex set V
Output: k-way path partitioning plan $\mathscr{P}$

1 **foreach** $v \in V$ **do**
2 | $\mathscr{E}(v) \leftarrow$ all paths contain v; $w(v) \leftarrow \sum_{ep \in \mathscr{E}(v)} \frac{1}{l_{ep}}$;
3 $Res \leftarrow \{\{ep_1\}, \{ep_2\}, ..., \{ep_n\}\}$;
4 $Res' \leftarrow \emptyset$; $V'_+ \leftarrow \emptyset$;
5 **foreach** $V_+ \subset V$ *such that* $|V_+|=2$ **and** $max_{pg}(V_+) \leqslant \lceil \frac{n}{k} \rceil$ **do**
6 | `Greedy` $(V_+, V \backslash V_+, Res)$;
7 | $V'_+ \leftarrow$ argmax$(|V_+|, |V'_+|)$; $Res' \leftarrow Res(V'_+)$;
8 **foreach** $PG_i \in Res'$ **do**
9 | $P_j \leftarrow$ `AssignPGToPartition` (PG_i)

Function: `Greedy` (V_+, V', Res)

10 **while** $V' \neq \emptyset$ **do**
11 | Choose v with the largest $\frac{1}{w(v)}$;
12 | **if** $max_{pg}(V_+ \cup \{v\}) \leqslant \lceil \frac{n}{k} \rceil$ **then** `// Merge(v)`
13 | | $Res \leftarrow Res$.`Union` (v); $V_+ \leftarrow V_+ \cup \{v\}$;
14 | $V' \leftarrow V' \backslash \{v\}$

constraint on the size of each path group (line 10-14). To merge v, we should combine all the path groups that contains paths passing v. This operations is implemented by using the `Union` function of the *disjoint set* data structure [8] (line 13). Finally, we choose the merging solution with the largest cardinality (line 7).

After obtaining the best merging solution, we need to assign the path groups into balanced partitions. It is known that the k-way balanced partition problem is NP-Complete [20]. The function `AssignPGToPartition` gives a greedy strategy for selecting the partition that the path group is assigned to (line 8-9). To get a balanced distribution, it sorts all path groups by their numbers of paths in descending order. Then it proceeds to assign path groups one by one in sorted order to the partitions and at each step, the partition with the smallest size is chosen.

Complexity Analysis. The algorithm first generates the paths which scans all the edges, thus the complexity is $O(|V| \cdot |E|)$. For each subset of V that contains two vertices, it scans all remaining vertices in $V \backslash V_+$, whose complexity is $O(|V|^3)$. The complexity of the union operator of the disjoint set data structure is nearly $O(1)$ [8]. Thus, the total complexity is $O(|V|^3)$. The final placement is mainly the cost of sorting, whose complexity is $O(|Res'| \log^{|Res'|})$. □

Example 3: Figure 4 shows an example that runs the Algorithm 1 on the RDF graph in Figure 3(a). First, $\mathscr{E}(v_1)$=$\{ep_1, ep_2\}$, $\mathscr{E}(v_2)$=$\{ep_3, ep_4\}$, $\mathscr{E}(v_3)$=$\{ep_5\}$, $\mathscr{E}(v_4)$=$\{ep_1, ..., ep_4\}$, $\mathscr{E}(v_5)$=$\mathscr{E}(v_8)$=$\mathscr{E}(v_9)$=$\{ep_1, ..., ep_6\}$, $\mathscr{E}(v_6)$=$\mathscr{E}(v_7)$=$\{ep_6, ep_7\}$ and $\mathscr{E}(v_{10})$=$\{ep_7\}$. Initially, we have Res=$\{\{ep_1\}, ..., \{ep_7\}\}$, and suppose that V_+=$\{v_1, v_2\}$. Then we merge v_1 and v_2 in Step 1. Take merging v_1 as an example, it is to union all elements in Res that contain the paths in $\mathscr{E}(v_1)$. After merging v_1 and v_2, we have that Res=$\{\{ep_1, ep_2\}, \{ep_3, ep_4\}, \{ep_5\}, \{ep_6\}, \{ep_7\}\}$. Then, v_6 and v_7 have the largest value of $\frac{1}{w(v)}$, thus we merge v_6 and v_7 and add them into V_+. We have Res=$\{\{ep_1, ep_2\}, \{ep_3, ep_4\}, \{ep_5\}, \{ep_6, ep_7\}\}$ as shown in Step 2. In the Step 3, we merge v_4, and get Res=$\{\{ep_1, ep_2, ep_3, ep_4\}, \{ep_5\}, \{ep_6, ep_7\}\}$. After that we can not merge any other vertices. Suppose this merging solution is the best one. Thus, we assign these three path groups into two partitions evenly, P_1=$\{ep_1, ep_2, ep_3, ep_4\}$ and P_2=$\{ep_5, ep_6, ep_7\}$. In this example, we do not consider v_3 and v_{10}. This is because $\mathscr{E}(v_3)$ and $\mathscr{E}(v_{10})$ only have one element, which means merging v_3 and v_{10} do not make any changes on Res. We can directly add them into V_+. □

Theorem 6: Algorithm 1 achieves an approximation factor of $(1 - e^{-\frac{1}{kl}})$, i.e., $|V'_+| \geqslant (1 - e^{-\frac{1}{kl}})|V^*_+|$, where V'_+ denotes the set of merged vertices of the plan generated by Algorithm 1 and V^*_+ denotes the set of merged vertices of the optimum $(k, 1)$-balanced path partitioning plan. □

The proof is complicated and hence omitted here (one can see Appendix for details).

VI. Bottom-Up Path Merging Algorithm

In view of the high complexity of the approximate algorithm, we propose two bottom-up path merging approaches in this section, which can dramatically reduce the complexity and work well in practice.

A. Merging Start Vertices

A previous empirical study [10] analyzed queries generated by both human and machine agents over two datasets, and concluded that the most common types of joins are S-S joins (60%) and S-O joins (35%). We also find the same conclusion from the benchmark queries in LUBM [11] and SP^{2}Bench [25] and queries over UniProt data in the literature [7], [21], [29]. S-S and S-O joins would mainly form the common types of queries, namely star, chain, cycle and tree queries.

According to this observation, we perform our bottom-up path merging algorithms in two major steps: (1) we merge all start vertices, which means all start vertices are in V_+; (2) we design a new weighting method for ordering the remaining vertices, then merge them according to this order.

The benefits of the first step are two-fold. First, as stated by the following theorem, merging all the start vertices can make all star, chain, cycle and tree queries as inner-partition queries, which are the most common queries as described above.

Theorem 7: If all start vertices are merged, the queries that only contain S-S and/or S-O joins (such as star, chain, directed cycle and tree queries) are inner-partition queries. □

Proof: Given an RDF graph G, for each vertex v in G, there must be a start vertex v_s that can reach v. Thus all the vertices reachable from v can also be reached from v_s. If v_s is merged, then all paths starting from v_s are in one partition and hence all vertices reachable from v are also in one partition. For a query Q only contains S-S and/or S-O joins, there exists a vertex v that can reach all other verices. Thus, it is an inner-partition query. ■

Second, the number of paths in a large RDF dataset would be huge. If we generate and store all the paths as done in the approximate algorithm, the space complexity would be very high, which is $\Theta(\overline{l_{ep}} \cdot n)$, where $\overline{l_{ep}}$ denotes the average length of the paths and n is the number of the paths. Merging all the start vertices can significantly reduce the space complexity of the algorithm to $\Theta(|V_s|)$, where V_s denotes the set of the start vertices. This is because we can use each start vertex to

represent all the paths starting from it, and thus save the cost of generating and storing them.

Example 4: In Figure 3(a), after merging all start vertices, start vertex v_1 is used to represent ep_1 and ep_2. □

B. Vertex Weighting

After merging all the start vertices, we attempt to order the remaining vertices for merging by using their profit-weight ratios as done in the approximate algorithm. Here, the profit for merging a vertex v is also counted as 1, however the weight of merging v cannot be easily calculated if we do not generate all the paths. Thus we adopt a heuristic that merging a vertex v shared by a larger number of paths might end up with a larger path group. Then, the weight of merging a vertex is denoted by $w'(v) = N_p(v)$, where $N_p(v)$ is the number of paths that contain v. The value of $N_p(v)$ can be estimated by the following method without generating all the paths.

First, we give a theorem to show how to exactly compute the number of the paths that contain a vertex v as follows:

Theorem 8: Given an RDF graph $G = (V, E)$, $N_p(v)$ can be computed by the following equation:

$$N_p(v) = I_p(v) \times O_p(v) \tag{3}$$

where $I_p(v)$ denotes the number of distinct paths from the start vertices to v that do not contain any cycle and $O_p(v)$ is the number of distinct paths from v to the end vertices. If v is a start vertex, then $I_p(v)$=1. Similarly, $O_p(v)$=1 if v is an end vertex. □

Example 5: In Figure 3(a), $I_p(v_4)$=2 and $O_p(v_4)$=2, since the number of paths from start vertices to v_4 is two (i.e., $v_1 \rightarrow v_4$ and $v_2 \rightarrow v_4$) and the number of paths from v_4 to end vertices is also two (i.e., $v_4 \rightarrow v_9 \rightarrow v_8 \rightarrow v_5 \rightarrow v_9$ and $v_4 \rightarrow v_5 \rightarrow v_9 \rightarrow v_8 \rightarrow v_5$). Thus, we have $N_p(v_4)$=4. □

According to this theorem, we can divide the estimation into two parts: estimating $I_p(v)$ and estimating $O_p(v)$. We define a recursive equation for estimating $I_p(v)$ and $O_p(v)$ as follows:

$$I_p(v) = \sum\nolimits_{u \in I(v)} I_p(u) \text{ and } O_p(v) = \sum\nolimits_{u \in O(v)} O_p(u) \tag{4}$$

where $I(v)$ denotes the in-neighbors of v and $O(v)$ denotes the out-neighbors of v.

We only discuss the estimation of $I_p(v)$ below and we can estimate $O_p(v)$ in a similar way. A solution to compute $I_p(v)$ can be implemented by iterations as follows:

$$I_p(v)_k = (1-\alpha) + \alpha \cdot \frac{\sum_{u \in I(v)} I_p(u)_{k-1}}{\sqrt{\sum_{u \in I(v)} (I_p(u)_{k-1})^2}} \tag{5}$$

where $I_p(v)_0 = 1$, α denotes a decay factor, and $\sqrt{\sum_{u \in I(v)} (I_p(u)_{k-1})^2}$ is used to normalize.

Let $\boldsymbol{A}$ denotes the adjacency matrix of RDF graph G and $\boldsymbol{I_p}$ denotes the estimated values of all the vertices. The equivalent matrix equation form is:

$$\boldsymbol{I_p} = (1-\alpha) \cdot \mathbf{1} + \alpha \cdot \frac{\boldsymbol{A} \cdot \boldsymbol{I_p}}{||\boldsymbol{A} \cdot \boldsymbol{I_p}||_2} \tag{6}$$

This computation can be easily implemented in a parallel processing framework, such as MapReduce framework, where each vertex iteratively gets the update information from its neighbors (see Appendix for details).

Theorem 9: Equation (6) is convergent. □

This can be proved by power iteration theory.

C. Class-based Vertex Weighting

In this section, we propose an alternative heuristic to compute the weight of each vertex, which is suitable for RDF graphs that contain *rdf:type* labels and can provide an even more efficient query decomposition approach.

According to the RDF W3C Recommendation, predicate *rdf:type* can be used to state that a resource, i.e. a vertex in the RDF graph model, is an instance of a class (we consider the resources without a predicate *rdf:type* are of the same class). To provide a more efficient query decomposition, we can leverage the class hierarchical information implied in an RDF dataset.

To understand how this can be done, let us first consider under what conditions we can combine two inner-partition subqueries, say SQ_i and SQ_j, into a new inner-partition subquery. Suppose SQ_i and SQ_j share a common vertex v. They can be combined to an inner-partition subquery if all the vertices in the RDF graph that can match v have been merged. To determine if this condition is met, we need to consider two cases: (i) v is a constant and (ii) v is a variable. In case (i), v only has one matching vertex in the unpartitioned RDF graph, so we can build an index to quickly determine whether this vertex has been merged.

Unfortunately, for case (ii), v could have an unknown number of matching vertices in the unpartitioned RDF graph and there is no efficient way to determine if all these vertices have been merged without actually evaluating the query. Fortunately, we observe that many queries often use *rdf:type* to indicate which class a variable v belongs to. For example, almost all the queries specified in the popular RDF benchmarks, such as LUBM and SP^{2}Bench, and those used in previous studies [7], [11], [19], [21], [25], [29], exhibit such property. So if we know that all the vertices in the RDF graph belonging to the same class as v have been merged, then we can ensure that SQ_i and SQ_j can be combined into a new inner-partition subquery. Taking use of such side-information can effectively help reduce the number of subqueries.

Therefore, we use the following weighting method so that the algorithm will merge vertices with the same class at the same time and then we can use such information to generate the better query decomposition.

Definition 6 (Class-based Vertex Weighting): Given an RDF graph $G = (V, E)$, let $C = \{T(v) | v \in V\}$ be a set of classes of the vertices, where $T(v)$ represent the class of v. We use the average weight score of all the vertices in class C_i ($C_i \in C$) as the weight of C_i, denoted as $w_{class}(C_i)$. □

Then, we assign a weight value to each vertex using the following strategy. All the vertices in each class are assigned the same weight value $w_{class}(v)$ ($v \in V \backslash V_s$), denoted by:

$$w_{class}(v) = w_{class}(T(v)),\ T(v) \in C \tag{7}$$

Algorithm 2: Generating Start Vertices List

Input: $G = (V, E)$
Output: a list of start vertices $S(v)$ for each vertex v

```
1  Vs ← GeneratingStartVertex(V, E);
2  foreach v ∈ V do
3  |  Activate(v);
4  |  if v ∈ Vs then S(v).Add(v);
5  |  else S(v) ← ∅ ;
6  repeat
7  |  foreach active v ∈ V do
8  |  |  foreach u ∈ out-neighbor(v) do
9  |  |  |  S(u).Add(S(v));
10 |  |  |  if S(u) is updated then
11 |  |  |  |  Activate(u); Signal;
12 |  |  |  else Deactivate(u);
13 until no update signal;
```

Algorithm 3: Bottom-Up Path Merging Algorithm

Input: $\{S(v)|v \in V\}$, all start vertices V_s
Output: a k-way partitioning result $\{P_1, ..., P_k\}$

```
1  Res ← {{vs1}, {vs2}, ..., {vsm}}, where vsj∈Vs ;
2  V' ← V\Vs ;
3  while V' ≠ ∅ do
4  |  Choose v with the largest 1/w_class(v) (or 1/w'(v)) ;
5  |  if max_pg(V+ ∪ {v}) ⩽ ⌈|Vs|/k⌉ then
6  |  |  Res ← Res.Union (v); V+ ← V+ ∪ {v} ;
7  |  V' ← V'\{v} ;
8  foreach PGi ∈ Res do
9  |  PGi ← ExtendByPaths(PGi);
10 foreach PGi do
11 |  Pj ← AssignPGToPartition(PGi);
```

D. The Complete Algorithm

In this section, we present the bottom-up path merging algorithm and its implementation details.

Our algorithm can be divided into two phases. In the first phase, Algorithm 2 is run to search a list of start vertices $S(v)$ for each vertex v, where $S(v)$ contains all the start vertices that can reach v. Specifically, the algorithm initially sets the $S(v)$=v if v is a start vertex, otherwise $S(v)$ is set as empty. Then the algorithm iteratively propagate the $S(v)$ of each vertex to its out-neighbors, until there is no update of $S(v)$ occurred.

Example 6: In Figure 4, all the lists of start vertices of each vertex are $S(v_4) = \{v_1, v_2\}$, $S(v_5) = S(v_8) = S(v_9) = \{v_1, v_2, v_3, v_6\}$ and $S(v_7) = S(v_{10}) = \{v_6\}$. □

After the previous phase, we run Algorithm 3 to perform path merging. Initially each path group contains one start vertex, which means all paths starting from that start vertex are merged (line 1). Then the algorithm merges the remaining vertices in descending ordering of $\frac{1}{w_{class}(v)}$ or $\frac{1}{w'(v)}$ (line 3-7). After the merging iterations stop, each resulting path group will be generated by extending each start vertex v_s in this path group with all the paths starting from v_s (line 8-9). Finally, the function `AssignPGToPartition` selects the partition that the path group is assigned to (line 10-11).

Example 7: Using the same example in Figure 3(a), the first step is to merge all start vertices, then we have Res=$\{\{v_1\}, \{v_2\}, \{v_3\}, \{v_6\}\}$. Suppose that $\frac{1}{w_{class}(v_4)}$ is the largest and $S(v_4)$=$\{v_1, v_2\}$. Then, at step 2, v_4 is merged, thus Res=$\{\{v_1, v_2\}, \{v_3\}, \{v_6\}\}$. We also merge v_7 and v_{10}, after that Res=$\{\{v_1, v_2\}, \{v_3\}, \{v_6\}\}$. Finally, the path groups are assigned onto two partitions, P_1=$\{v_1, v_2\}$ and P_2=$\{v_3, v_6\}$. □

Complexity Analysis. Algorithm 2 scans all the edges, and hence the total complexity is $O(|E|)$. In Algorithm 3, the $while$ loop is to merge all the vertices in $V\backslash V_s$. Thus, the complexity of path merging is $O(|V|)$. In addition, the total complexity of path extension is $O(|E|)$. The final placement of path groups to the final partitions is mainly the cost of sorting, whose complexity is $O(|Res| \log^{|Res|})$. □

VII. Query Decomposition

In the previous sections, we have presented the path-based partitioning approaches, which are able to carefully partition a big RDF graph into path preserving data partitions. If a query is inner-partition query which can be executed locally without collaboration between computing nodes, it will be sent to the local engines running at the individual nodes. The query will be processed and the results will be returned directly. However, not all queries are inner-partition queries. If a query is a cross-partition query, which has to be evaluated involving distributed joins, the query has to be split into inner-partition subqueries. Then the final results will be processed by joining the results of all the inner-partition subqueries in a distributed manner using MapReduce jobs [15]. In this section, we present how to perform query decomposition.

According to our query decomposition model and Theorem 7, given a query Q, we first decompose Q into a set of subqueries, SQ=$\{SQ_1, ..., SQ_m\}$, where each SQ_i consists of a start vertex of Q and all vertices that can be reachable from this start vertex and the corresponding edges. Thus each SQ_i is a subquery that only contains S-S and/or S-O joins. Then we attempt to further reduce the number of subqueries by combining some of them. Specifically, given two subqueries SQ_i and SQ_j, suppose $V_{i,j}$ is the set of shared vertices between them. According to Theorem 3, if one of the vertices $V_{i,j}$ is a constant and it appears to be merged after looking into our index, then we merge these two subqueries into a new subquery, which replaces SQ_i and SQ_j.

If the shared vertex is a variable and we use class-based weighting, then we see if this variable has a known class (i.e. *rdf:type*) and all vertices in this class are merged. If both are true, then these two subqueries can be merged. We can continue this process until we get a number of inner-partition subqueries that cannot be further merged. This will produce a minimal number of subqueries given a path-based data partitioning plan.

VIII. Experiment

We use a cluster of 20 computing nodes for all experiments. Each node in the cluster has two processors at 2.4GHz, 6GB RAM and a 500GB hard disk.

We compare our approach with the state-of-the-art RDF data partitioning algorithms proposed in [15] and [19]. To

TABLE I: RDF Datasets

Dataset	#Triple ($\|E\|$)	#Entity ($\|V\|$)	#Class	N3 File Size
LUBM-Tiny	6K	2K	14	1 MB
LUBM-1	100K	27K	14	18 MB
LUBM-10	1,200K	315K	14	222 MB
LUBM-1000	138M	33M	14	24.1 GB
LUBM-2000	276M	66M	14	47.4 GB
LUBM-5000	691M	164M	14	118.8 GB
LUBM-10000	1,382M	329M	14	237.8 GB
UniProt	1,975M	619M	120	543.3 GB
SP^{2}Bench-500M	500M	222M	12	51.4 GB

TABLE II: Queries

	Star	Chain	Tree	Complex
LUBM	Q1, Q2	Q8, Q11, Q12	Q7, Q9	Q3, Q4, Q5, Q6, Q10
UniProt	Q2	Q5	Q1, Q3, Q4, Q6	
SP^{2}Bench	Q1	Q2	Q3	

perform a fair comparison, we follow the implementation and setup of the distributed RDF processing system proposed in [15], [19] and take use of RDF-3X [22] and Hadoop to build the system. We refer to our approximate algorithm as **Path-AX**, and refer to the bottom-up path merging algorithms with vertex weighting and class-based vertex weighting as **Path-BM** and **Path-BMC** respectively. We also implement a baseline path partitioning algorithm, called **Path-Hash**, which partitions paths by hashing the start vertex of each path. Furthermore, we implement the algorithm proposed in [15], which uses METIS [2] as the graph partitioning tool and applies undirected one-hop (**un-one**) and undirected two-hop (**un-two**) vertex extension. Finally, we implement the semantic hash partitioning algorithm, 2-hop forward (**2f**), in [19] that is reported to perform the best in most cases. The 2f extends each vertex with a 2-hop forward expansion to form a subgraph and then hash such subgraphs to the computing nodes.

For the experiments, we use nine datasets, from LUBM [11], UniProt [5] and SP^{2}Bench [25]. The properties of these data are listed in Table I. The LUBM and SP^{2}Bench are benchmark generators. The UniProt is a real-world protein dataset. All the benchmark queries (see Appendix for details) come from [7], [11], [19], [21], [25], [29], covering most types of queries mentioned earlier in the paper, as shown in Table II.

Table III shows the comparison among different path partitioning algorithms. As one can see, it takes 175 minutes to run Path-AX on an RDF graph with 6K edges and 2K vertices. Path-AX cannot finish processing over LUBM-1 and LUBM-10 within 3 days. However, Path-BM and Path-BMC only need several seconds to partition those two datasets. In addition, they perform well in maximizing the number of merged vertices. Hence, we only consider Path-BMC in the following experiments. A comparison of query performance between Path-BM and Path-BMC is shown in Section VIII-D.

A. Partitioning Time, Data Balance and Duplication

We explore the impact of parameters on the partitioning time, data load balance and data duplication of the data partitioning algorithms. Note that, we cannot get METIS to work on larger datasets, such as LUBM-5000, LUBM-10000, SP^{2}Bench-500M and UniProt, due to insufficient memory. Therefore we can only collect results for the graph partitioning algorithm over the LUBM-2000 dataset.

Table IV shows the data partitioning time for different algorithms (METIS and path merging phase in Path-BMC run on a server with 24GB RAM). The un-one and un-two approaches have the longest partitioning time, since running METIS takes a lot of time. Path-Hash, 2f and Path-BMC can achieve more efficient data partitioning, while Path-BMC is slightly slower due to the cost of calculating the weights and merging the vertices.

TABLE III: Comparison of Path Partitioning Algorithms

Algorithms	LUBM-Tiny		LUBM-1		LUBM-10	
	$\|V_+\|$	Time	$\|V_+\|$	Time	$\|V_+\|$	Time
Path-AX	1.6K	175 min	N/A	$>$ 3 days	N/A	$>$ 3 days
Path-BM	1.7K	0.9 sec	24.2K	2.8 sec	303.8K	20.6 sec
Path-BMC	1.3K	0.9 sec	18.4K	2.7 sec	220.8K	20.1 sec

TABLE IV: Data Partitioning Time

	un-one	un-two	2f	Path-Hash	Path-BMC
LUBM-2000	220 min	294 min	79 min	65 min	81 min
LUBM-10000	N/A	N/A	273 min	314 min	376 min
UniProt	N/A	N/A	852 min	896 min	958 min
SP^{2}Benck-500M	N/A	N/A	156 min	149 min	169 min

The duplication ratios (Eq. (1)) achieved by all the algorithms are listed in Table V. As one can see, the numbers of duplicated triples of un-one and un-two approaches are very large. It can be attributed to the existence of high-degree vertices. Furthermore, by increasing the vertex extension from one-hop to two-hop, the number of duplicated triples grows rapidly. The 2f algorithm can significantly improve the data duplication, which is consistent with the results reported in [19]. Our Path-BMC achieves a even more dramatic reduction on data duplication. This is because many vertices are merged in this case and the triples involving the merged vertices are ensured to have only one copy. For the case of Path-Hash, although the load is balanced, the data duplication is rather high. This is because this algorithm fails to consider the common vertices and edges among paths.

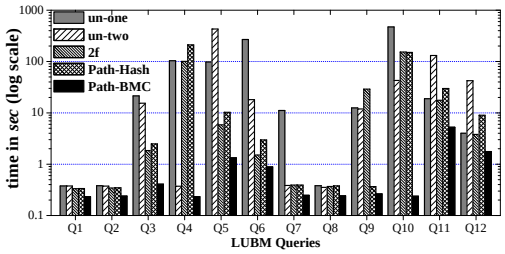
For data distribution, in Table V, we reported the standard deviation (σ) of the percentage of triples allocated to each partition. The un-one and un-two approaches have a very skewed data distribution. Moreover, un-two could cause a large number of duplicated triples at some small number of nodes. For example, one computing node contains 21.2% of the total triples, which is almost a duplication of the LUBM-2000 dataset. This is because some high-degree vertices are allocated to one partition. The highly loaded nodes will be the bottleneck for parallel processing. Path-Hash and 2f do not have such problems and only have some slight data imbalance. Path-BMC has the most balanced data distribution.

B. Query Performance

As un-one and un-two use METIS, which cannot handle larger datasets, we only use LUBM-2000 dataset for the

TABLE V: Data Balance and Data Duplication

	Algorithms	Data Balance		Data Duplication
		SD (σ)	Max Partition Size	$Dup(\mathscr{P})$ (Eq. (1))
LUBM-2000	un-one	0.0215	11.8%	0.22
	un-two	0.0438	21.2%	2.00
	2f	0.0036	6.1%	0.12
	Path-Hash	0.0012	5.9%	1.66
	Path-BMC	**0.0001**	**5.0%**	**0.03**
UniProt	2f	0.0032	8.8%	0.45
	Path-Hash	0.0063	8.1%	2.02
	Path-BMC	**0.0021**	**5.3%**	**0.28**
SP^{2}Benck-500M	2f	0.0025	6.9%	0.20
	Path-Hash	0.0043	7.5%	1.78
	Path-BMC	**0.0018**	**5.1%**	**0.17**

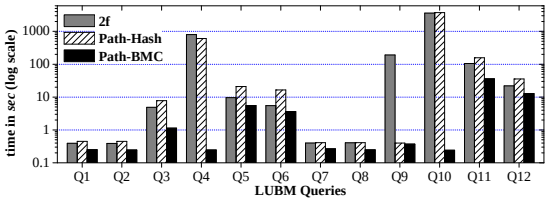


(a) Query Performance (LUBM-2000)

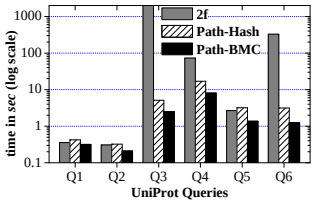
		LUBM Queries											
		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
un-one	#SQ	1	1	3	2	3	3	2	1	3	3	1	1
	IR (MB)	0	0	6.0	1190.6	660.6	1945.9	67.4	0	124.8	3857.5	0	0
un-two	#SQ	1	1	1	1	1	1	1	1	2	2	1	1
	IR (MB)	0	0	0	0	0	0	0	0	0.1	316.2	0	0
2f	#SQ	1	1	1	2	1	1	1	1	3	3	1	1
	IR (MB)	0	0	0	1190.6	0	0	0	0	127.5	1765.7	0	0
Path-Hash	#SQ	1	1	1	2	1	1	1	1	1	2	1	1
	IR (MB)	0	0	0	566.2	0	0	0	0	0	1602.3	0	0
Path-BMC	#SQ	1	1	1	1	1	1	1	1	1	1	1	1
	IR (MB)	0	0	0	0	0	0	0	0	0	0	0	0

(b) Query Decomposition and Intermediate Results Size (LUBM-2000)

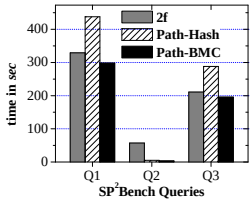
Fig. 5: Query Performance, Query Decomposition and Intermediate Results Size (LUBM-2000)



(a) LUBM-10000



(b) UniProt



(c) SP^{2}Bench-500M

Fig. 6: Query Processing on Large Datasets

comparison with these two approaches. The results are shown in Figure 5. We can run 2f, Path-Hash and Path-BMC on larger datasets, such as LUBM-10000, UniProt and SP^{2}Bench, whose results are shown in Figure 6.

All the reported numbers are the averages of at least three runs of the queries. To carefully follow the configuration presented in [15], we remove the Hadoop's start-up overhead. Figure 5(b) shows the number of subqueries into which the query should be decomposed (**#SQ**) and the size of the intermediate results need to be transferred during distributed joins (**IR**). For Path-BMC, all queries are inner-partition queries, i.e. they can be processed locally without decomposition.

The first observation is that Path-BMC outperforms other algorithms on all the benchmark queries and the improvement is more significant on the queries with long chains or complex structures, or with large input and low selectivities.

Let us look deeper into Figure 5(a) and analyze each individual query. Q1 and Q2 are simple and highly selective star queries, which only contains S-S join with a constant as the star center, and Q8 is a two-hop chain query. All these three queries are inner-partition queries for all five partitioning algorithms. As they are all highly selective and the local RDF-3X engines can use local indexes to efficiently retrieve the results, the only reason for the differences in query performance is the different amount of data duplication, which would incur duplicate computation and result generation.

Q11 and Q12 are one-hop chain queries and hence are also inner-partition queries. But they are not very selective and hence each local engine has to process a large amount of data. Thus the performance affected by both the data duplication and distribution. So we can see un-two performs significantly than all the other approaches due to its worst data balance.

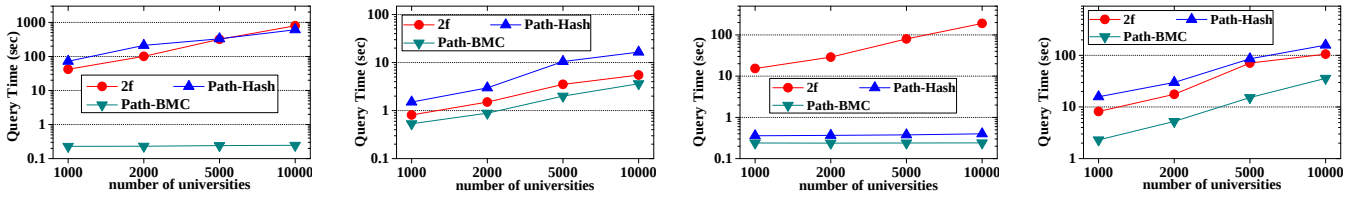
Q3, Q5, Q6 and Q7 are queries whose paths' lengths are at most equal to 2, which means only the un-one needs to decompose them into 2 or more subqueries. Therefore, un-one performs significantly worse than the others in most cases.

Q9 is a highly selective tree query whose paths are quite long (up to 5). Thus, except Path-BMC and Path-Hash, the other algorithms have to decompose it into multiple subqueries and then join the intermediate results using MapReduce on Hadoop. As shown in Figure 5(b), un-two decomposes Q9 into 2 subqueries, while un-one and 2f have to decompose it into 3 subqueries, some of which are not very selective. Therefore, for un-one and 2f, a large amount intermediate results have to be transfered across computing nodes to perform distributed joins. In this case, Path-BMC improves over these approaches by up to two orders of magnitude.

Q4 and Q10 are more complex queries that contain all three types of joins. In particular, Q4 has to be decomposed in the approaches of un-one, 2f and Path-Hash, which perform significantly worse than both un-two and Path-BMC due to the large amount of intermediate results. Q10 is a selective query, however, all approaches except Path-BMC have to decompose it into several un-selective subqueries, which could incur a large amount of intermediate results. Thus, Path-BMC outperforms all the other algorithms two to three orders of magnitudes. We also observe that the memory consumption is another critical aspect in addition to I/O cost and communication cost for expensive queries, which will further discussed in Section VIII-C.

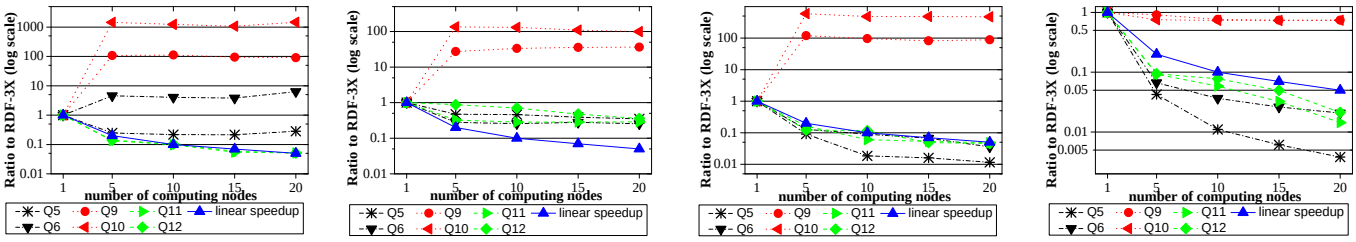
In Figure 6(a), we find that the results of query performance in LUBM-2000 is similar to those for LUBM-10000. However, with a greater size of datasets, a greater performance improvement can be achieved by using Path-BMC in comparing to 2f or Path-Hash, especially for complex queries like Q4, Q9 and Q10. This is because the sizes of intermediate results become larger when the sizes of datasets increase.

Figure 6(b) presents the results for UniProt dataset. Q3, Q4 and Q6 are tree queries with a height of 3. The 2f needs to decomposed them into 2 subqueries. In addition, one of the subqueries is not very selective, leading to expensive



(a) LUBM Q4 (b) LUBM Q6 (c) LUBM Q9 (d) LUBM Q11

Fig. 7: Query Performance in Varying Dataset Size (from LUBM-1000 to LUBM-10000)



(a) Speedup of un-one (b) Speedup of un-two (c) Speedup of 2f (d) Speedup of Path-BMC

Fig. 8: Query Performance in Varying Cluster Size (LUBM-2000)

distributed join and large amount of communication cost. Thus, 2f has very bad performance for these two queries. The rest queries in UniProt are inner-partition query for all three algorithms, the query performance only depends on the data redundancy and data distribution.

In Figure 6(c), Q2 needs to be decomposed in 2f. Q1 and Q3 are inner-partition query for all three algorithms. The performance of the different methods is consistent with the previous datasets.

C. Scalability

Data size. In Figure 7, we choose four typical queries to show the scalability of different partitioning algorithms. For Path-BMC, as the number of triples increases, the execution time of each query also increases. But, we observe that the increase of execution time is sublinear or nearly linear, which means Path-BMC achieves a good property for scaling. In particular, Q4 and Q9 are highly selective queries, thus their performance is not sensitive to the data sizes. Q6 and Q11 are non-selective queries that have complex structures and large input. Hence they can benefit from the well-balanced data distribution and low data duplication of Path-BMC and have nearly linear scalability.

For 2f and Path-Hash, the execution time increases more rapidly than Path-BMC. It is interesting to look at Q4 in particular. Both 2f and Path-Hash need to decompose Q4 into 2 subqueries. When the data size is small, the number of duplicate triples plays the main role in query performance. As the data size increases, the size of intermediate results increases significantly, which incurs a large communication cost and expensive MapReduce jobs for joining the intermediate results.

Cluster size. This section reports the comparison of the four algorithms (the results of Path-Hash are similar to those of 2f and hence we omit Path-Hash here) with varying size of server cluster. We run LUBM-2000 on the server cluster with varying sizes, from 1, 5, 10, 15 to 20 computing nodes and measure the performance of queries Q5, Q6 , Q9, Q10, Q11 and Q12. The results are displayed in Figure 8. We normalize the query execution time of each query by the execution time with a single-node RDF-3X engine, and include a linear speedup line (1, 0.2, 0.1, 0.07 and 0.05 for 1, 5, 10, 15 and 20 computing nodes) for better presentation.

With regard to Path-BMC in Figure 8(d), all queries except Q9 and Q10 can achieve a faster than linear speedup with an increasing cluster size. This is because each benchmark query can be performed locally without the expensive distributed join operations with Hadoop jobs. For Q9 and Q10, the network communication delay for aggregating the results from all computing nodes take up the most portion of query execution time, and the network communication delays are the same for different cluster sizes. Therefore, the execution times from 5 to 20 are roughly the same.

For 2f, the query processing time of Q5, Q6, Q11, Q12 is nearly linear. But, for Q9 and Q10, it has a poor performance largely due to the fact that these two queries need to be decomposed.

For un-one and un-two, almost all queries, except Q11 and Q12 in un-one, cannot benefit from a cluster size increasing from 5 to 20. There are a number of reasons. The data distributions in un-one and un-two are highly imbalanced (Section VIII-A), and the overloaded partitions become the bottleneck. E.g., for Q10 in un-one, the intermediate results are approximately 4 GB (Figure 5(b)), regardless of the cluster size. In addition, the high I/O communication cost, memory consumption becomes another critical bottleneck [7], [29]. This is because the index, candidate triples and intermediate results need to reside in the main memory and we observe frequent swapping happens in this case.

D. Parameter Analysis

In this section, we evaluate the effects of different configuring parameters. In particular, we vary the number of merged classes in Path-BMC, from merging all vertices in first 6

TABLE VI: Different Configurations

	Data Balance		Data Duplication	#Path Group
	SD (σ)	Max Partition Size	$Dup(\mathscr{P})$ (Eq. (1))	
Path-Hash	0.0012	5.9%	1.66	N/A
Path-BMC-6	0.0001	5.1%	0.88	20.6M
Path-BMC-9	0.0001	5.1%	0.19	13.4M
Path-BMC-13	0.0001	5.0%	0.03	80.4K
Path-BM	0.0001	5.0%	0.03	72.4K

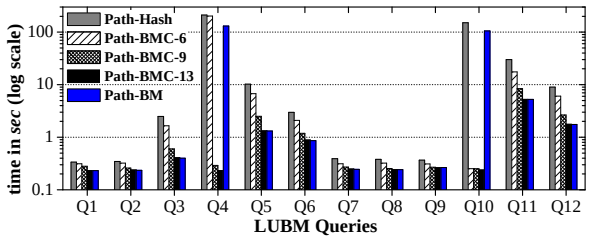


Fig. 9: Performance of Different Configurations

classes to first 13 classes (Path-BMC-6, Path-BMC-9 and Path-BMC-13), and compare them with Path-BM and Path-Hash on LUBM-2000 dataset.

Table VI shows the experimental results of data balance, data duplication and the number of remained path groups. As expected, as the number of merged classes increases, the duplication are decreased rapidly and the number of remained path groups is reduced. Benefiting from the path group placement strategy, the data distribution is very balanced. In Figure 9, we explore the differences on query answering time. For Q4 and Q10, we observe a substantial performance difference among different parameters. The performance of Path-BMC with the lower number of merged classes is much poorer, even underperforms the other parameters up to nearly two orders of magnitude. This is because Q4 (or Q10) need to be decomposed into two subqueries with the number of class merged smaller or equal to 7 (or 5). Path-BM can achieve good data balance and low duplication and can deliver acceptable query performance. However, Q4 and Q10 need to be decomposed into 2 subqueries with Path-BM due to the inferior query decomposition strategy associated with Path-BM, and hence it does not perform well for these two queries.

IX. Conclusion

In this paper, we analyze the limitation of the existing RDF data partitioning methods and then propose path partitioning, a novel and effective data partitioning technique for scalable SPARQL query processing over big RDF graphs. By partitioning the big RDF dataset into multiple path preserving data partitions, we can make many complex SPARQL queries to be inner-partition queries and hence can successfully avoid or largely reduce the cost of distributed joins over the large RDF dataset. Our experimental results verify that the proposed approach can localize many complex queries while maintaining balanced data distribution and minimizing data duplication, and hence dramatically outperforms the state-of-the-art data partitioning approaches by orders of magnitude.

Acknowledgment

This work is carried out while Buwen Wu is a visiting PhD student at the University of Southern Denmark partially sponsored by DASTI's International Network Programme. The authors from HUST are supported by National High Technology Research and Development Program of China (863 Program) under grant No.2012AA011003 and National Science Foundation of China (61073096). Ling Liu is partially sponsored by NSF CISE under under grants IIS-0905493, CNS-1115375, IIP-1230740 and a grant from Intel ISTC on Cloud Computing.

References

[1] *Linking Open Data*. http://lod-cloud.net/state/.

[2] *METIS*. http://www.cs.umn.edu/~metis/.

[3] *RDF*. http://www.w3.org/TR/rdf-concepts/.

[4] *SPARQL*. http://www.w3.org/TR/rdf-sparql-query/.

[5] *Universal Protein Resource*. http://www.uniprot.org/.

[6] K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[7] M. Atre, V. Chaoji, et al. Matrix bit loaded: a scalable lightweight join query processor for RDF data. In *WWW*, 2010.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*. MIT Press, 2001.

[9] M. Eltabakh, Y. Tian, et al. CoHadoop: Flexible data placement and its exploitation in Hadoop. *PVLDB*, 4(9):575–585, 2011.

[10] M. A. Gallego, J. D. Fernández, et al. An empirical study of real-world SPARQL queries. In *USEWOD*, 2011.

[11] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics*, 3(2):158–182, 2005.

[12] S. Gurajada, S. Seufert, et al. Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. In *SIGMOD*, 2014.

[13] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *SSWS*, pages 94–109, 2009.

[14] A. Harth, J. Umbrich, et al. YARS2: A federated repository for querying graph structured data from the web. In *ISWC*, 2007.

[15] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134.

[16] M. Husain, J. McGlothlin, et al. Heuristics based query processing for large RDF graphs using cloud computing. *IEEE TKDE*, 23(9), 2011.

[17] D. Jiang, B. Ooi, L. Shi, and S. Wu. The performance of MapReduce: An in-depth study. *PVLDB*, 3(1-2):472–483, 2010.

[18] Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. SPARQL query optimization on top of DHTs. In *ISWC*, pages 418–435, 2010.

[19] K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *PVLDB*, 6(14):1894–1905, 2013.

[20] R. G. Michael and S. J. David. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

[21] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, pages 627–640, 2009.

[22] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[23] P. Ravindra, S. Hong, et al. Efficient processing of RDF graph pattern matching on MapReduce platforms. In *DataCloud-SC*, 2011.

[24] K. Rohloff and R. E. Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In *DIDC*, 2011.

[25] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2bench: a SPARQL performance benchmark. In *ICDE*, pages 222–233, 2009.

[26] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *ICDE*, pages 568–579, 2014.

[27] B. Wu, Y. Zhou, P. Yuan, H. Jin, and L. Liu. Semstore: A semantic-preserving distributed rdf triple store. In *CIKM*, pages 509–518, 2014.

[28] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD*, pages 517–528, 2012.

[29] P. Yuan, P. Liu, B. Wu, L. Liu, H. Jin, and W. Zhang. TripleBit: a fast and compact system for large scale RDF data. *PVLDB*, 6(7), 2013.

[30] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *VLDB*, pages 265–276, 2013.

APPENDIX

Queries:

LUBM:

Q1: SELECT ?x WHERE { ?x rdf:type ub:ResearchGroup. ?x ub:subOrganizationOf <http://www.Department0.University0.edu>. }

Q2: SELECT ?x WHERE { ?x rdf:type ub:FullProfessor. ?x ub:emailAddress ?y2. ?x ub:name ?y1. ?x ub:worksFor <http://www.Department0.University0.edu>. ?x ub:telephone ?y3. }

Q3: SELECT ?x ?y ?z WHERE { ?z rdf:type ub:Department. ?x ub:memberOf ?z. ?x rdf:type ub:UndergraduateStudent. ?y rdf:type ub:University. ?z ub:subOrganizationOf ?y. ?x ub:undergraduateDegreeFrom ?y. }

Q4: SELECT ?x ?y WHERE { ?x rdf:type ub:GraduateStudent. <http://www.Department0.University0.edu/AssociateProfessor0> ub:teacherOf ?y. ?y rdf:type ub:GraduateCourse. ?x ub:takesCourse ?y. }

Q5: SELECT ?x ?y ?z WHERE { ?z ub:subOrganizationOf ?y. ?y rdf:type ub:University. ?z rdf:type ub:Department. ?x rdf:type ub:GraduateStudent. ?x ub:memberOf ?z. ?x ub:undergraduateDegreeFrom ?y. }

Q6: SELECT ?x ?y ?z WHERE { ?y ub:teacherOf ?z. ?y rdf:type ub:FullProfessor. ?z rdf:type ub:Course. ?x ub:takesCourse ?z. ?x rdf:type ub:UndergraduateStudent. ?x ub:advisor ?y. }

Q7: SELECT ?x ?y WHERE { ?x ub:worksFor ?y. ?y rdf:type ub:Department. ?x rdf:type ub:FullProfessor. ?y ub:subOrganizationOf <http://www.University0.edu>. }

Q8: SELECT ?x ?y WHERE { ?x ub:worksFor ?y. ?y ub:subOrganizationOf <http://www.University0.edu>. }

Q9: SELECT ?x ?w WHERE { ?x ub:advisor ?y. ?y ub:worksFor ?z. ?x rdf:type ub:GraduateStudent. ?z ub:subOrganizationOf ?w. ?w ub:name ?u. ?z rdf:type ub:Department.?w rdf:type ub:University. <http://www.Department12.University0.edu/FullProfessor0/Publication0> ub:publicationAuthor ?x. }

Q10: SELECT ?x ?p WHERE { ?x ub:advisor ?y. ?y ub:worksFor ?z. ?x rdf:type ub:GraduateStudent. <http://www.Department0.University0.edu/FullProfessor0/Publication0> ub:publicationAuthor ?x. ?p ub:name ?n. ?z rdf:type ub:Department. ?z ub:subOrganizationOf ?w. ?p ub:publicationAuthor ?x. }

Q11: SELECT ?x WHERE { ?x rdf:type ub:UndergraduateStudent . }

Q12: SELECT ?x WHERE { ?x rdf:type ub:GraduateStudent . }

UniProt:

Q1: SELECT ?p2 ?i ?p1 WHERE { ?p1 rdf:type uni:Protein. ?p1 uni:enzyme <http://purl.uniprot.org/enzyme/2.7.7.->. ?i uni:participant ?p1. ?i rdf:type uni:Interaction. ?i uni:participant ?p2. ?p2 rdf:type uni:Protein. ?p2 uni:enzyme <http://purl.uniprot.org/enzyme/3.1.3.16>. }

Q2: SELECT ?a ?vo WHERE { ?a uni:encodedBy ?vo. ?a schema:seeAlso <http://purl.uniprot.org/refseq/NP_346136.1>. ?a schema:seeAlso <http://purl.uniprot.org/tigr/SP_1698>. ?a schema:seeAlso <http://purl.uniprot.org/pfam/PF00842>. ?a schema:seeAlso <http://purl.uniprot.org/prints/PR00992>. }

Q3: SELECT ?a ?vo ?c WHERE { ?a schema:seeAlso ?vo. ?b uni:mnemonic ?c. ?a uni:classifiedWith <http://purl.uniprot.org/keywords/67>. ?ab uni:replacedBy ?b. ?a uni:replaces ?ab. ?a schema:seeAlso <http://purl.uniprot.org/embl-cds/AAN81952.1>. }

Q4: SELECT ?p ?a WHERE { ?p uni:annotation ?a. ?p rdf:type uni:Protein. ?a uni:range ?r. ?a rdf:type <http://purl.uniprot.org/core/Transmembrane_Annotation>. }

Q5: SELECT ?p ?a WHERE { ?p uni:annotation ?a. ?p rdf:type uni:Protein. ?p uni:organism taxon:9606. ?a rdfs:comment ?t. ?a rdf:type <http://purl.uniprot.org/core/Disease_Annotation>. }

Q6: SELECT ?a ?vo WHERE { ?a schema:seeAlso ?vo. ?b schema:seeAlso <http://purl.uniprot.org/geneid/1025922>. ?b schema:seeAlso <http://purl.uniprot.org/smr/P0A7A1>. ?b schema:seeAlso <http://purl.uniprot.org/embl-cds/AAP18215.1>. ?a uni:replaces ?ab. ?ab uni:replacedBy ?b. }

SP^{2}Bench:

Q1: SELECT ?inproc ?author ?booktitle ?title ?proc ?ee ?page ?url ?yr ?abstract WHERE { ?inproc rdf:type bench:Inproceedings. ?inproc dc:creator ?author. ?inproc bench:booktitle ?booktitle. ?inproc dc:title ?title. ?inproc dcterms:partOf ?proc. ?inproc rdfs:seeAlso ?ee. ?inproc swrc:pages ?page. ?inproc foaf:homepage ?url. ?inproc dcterms:issued ?yr. }

Q2: SELECT ?name WHERE { ?author foaf:name ?name. <http://publications/inprocs/Proceeding21/1983/Inproceeding1017> dcterms:references ?doc. ?x dc:creator ?author. ?doc ?p ?x. }

Q3: SELECT ?yr ?name ?doc WHERE { ?doc rdf:type ?class. ?class rdfs:subClassOf foaf:Document. ?doc dcterms:issued ?yr. ?doc dc:creator ?author. ?author foaf:name ?name. }

Proof of Theorem 1

Proof: (i) If v is a source, then v is in the path started from v. If v is not a source and there exists a source v_s can reach v, then there must exist a path started from v_s and pass through v. If v is not a source and none of the sources can reach v, then there must exist a minimal-ID vertex can reach v (according to Algorithm 4) and v is in the path started from that vertex and passes through v. Therefore every vertex in G must exist in at least one path. (ii) For an edge $(u, w) \in E$, according to (i), we can ensure that u must at least be in one path. Suppose that this path is denoted as a sequence of vertices $v_s...u...v_e$. If u is not the end vertex of this path, then there must be a path starting from v_s passing (u, w). If u is the end vertex of this path, according to (i), we have that u must appear in this path twice, then this path can be denoted by $v_s...u...u$. Also there must be a path starting from v_s passing (u, w). ■

Proof of Theorem 4

Proof: We divide the edges E into two disjoint subsets: E_+=$\{(u,v)|u \in V_+ \ or \ v \in V_+\}$ and E_-=$\{(u,v)|u \in V\backslash V_+ \ and \ v \in V\backslash V_+\}$. Since an edge $e \in E_+$ occurs only in one partition, i.e. $|\mathscr{P}(e)|$=1, then,

$$Dup(\mathscr{P})=\frac{1}{|E|}\sum\nolimits_{e\in E}(|\mathscr{P}(e)|-1)=\frac{1}{|E|}\sum\nolimits_{e\in E_-}(|\mathscr{P}(e)|-1).$$

Since $1\leqslant|\mathscr{P}(e)|\leqslant k$ and $|E_-|\leqslant|V\backslash V_+|^2=(|V|-|V_+|)^2$, then,

$$\frac{1}{|E|}\sum\nolimits_{e\in E_-}(|\mathscr{P}(e)|-1)\leqslant\frac{|E_-|}{|E|}(k-1)\leqslant\frac{(|V|-|V_+|)^2}{|E|}(k-1).$$

■

Proof of Theorem 5

Proof: (1) To prove that this problem is NP-hard, we show a reduction from the k-balanced hypergraph partitioning problem. Given a hypergraph $H = (V_H, E_H)$, where V_H is a set of vertices and E_H is a set of edges. Each edge is a non-empty subset of V_H denoted by $eh = \{vh_i, ..., vh_m\}$, where vh_i is a vertex in V_H. The k-balanced hypergraph partitioning problem is to partition the vertices of H into k equal partitions, such that the total number of edges crossing between partitions is minimized. Then given a hypergraph H, the following local replacement can be performed. We replace every $vh \in V_H$ with a path ep and each edge $eh \in E_H$ with a vertex v with $|eh|$ paths passing it. If an edge is not across different partitions, then all its vertices are in the merged vertex set V_+. Thus, the total number of edges crossing between partitions is minimized if and only if $|V_+|$ is maximized.

(2) we use a reduction from the 3-Partition problem to prove that this problem is AXP-hard, similar to the proof in [6]. ■

Proof of Theorem 6

Let V_+^* be the set of vertices merged in optimal solution. Let us order the vertices in V_+^* in the deceasing order by the profit-weight ratio $\frac{1}{w(v)}$. Let Y be the first two vertices in this order. Let Y' be the set of vertices added by the function `Greedy` to Y, denoted by $v_1, ..., v_r$. Let v_{r+1} be the first vertex from $V_+^* \backslash Y$ that is considered but not added to Y'. Let B denotes the constraint on the size of the path group, i.e. $B = \lceil \frac{n}{k} \rceil$, where n denotes the number of the paths.

Before giving the proof of this theorem, we first prove the following lemmas.

Lemma 1: For each i=1, ..., $r+1$, the following holds,

$$\frac{k \cdot B}{w(v_i)} \geqslant |V_+^* \backslash Y| - \sum_{j=1}^{i-1} 1 \tag{8}$$

where 1 means the profit for merging a vertex. □

Proof: For all $j \in V_+^* \backslash (Y \cup \{v_1, ..., v_{i-1}\})$, we have that

$$\frac{1}{w(v_j)} \leqslant \frac{1}{w(v_i)} \tag{9}$$

Thus,

$$\begin{aligned} |V_+^* \backslash Y| - \sum_{j=1}^{i-1} 1 &= \sum_{j \in V_+^* \backslash (Y \cup \{v_1, ..., v_{i-1}\})} 1 \\ &\leqslant \sum_{j \in V_+^* \backslash (Y \cup \{v_1, ..., v_{i-1}\})} \frac{w(v_j)}{w(v_i)} \\ &\leqslant n \cdot \frac{1}{w(v_i)} \\ &\leqslant k \cdot B \cdot \frac{1}{w(v_i)}. \end{aligned} \tag{10}$$

■

Lemma 2: For each i=1, ..., $r+1$, we have

$$\sum_{j=1}^{i} 1 \geqslant (1 - \prod_{j=1}^{i} (1 - \frac{w(v_i)}{k \cdot B}))(|V_+^* \backslash Y|) \tag{11}$$

□

Proof: We prove the lemma by induction. Clearly, it is true for i=1. Let us assume it is true for the case $i-1$, then we will show it holds for the case i.

$$\begin{aligned} \sum_{j=1}^{i} 1 &= \sum_{j=1}^{i-1} 1 + 1 \\ &\geqslant \sum_{j=1}^{i-1} 1 + \frac{w(v_i)}{k \cdot B}(|V_+^* \backslash Y| - \sum_{j=1}^{i-1} 1) \\ &= (1 - \frac{w(v_i)}{k \cdot B}) \sum_{j=1}^{i-1} 1 + \frac{w(v_i)}{k \cdot B}(|V_+^* \backslash Y|) \\ &\geqslant (1 - \prod_{j=1}^{i} (1 - \frac{w(v_i)}{k \cdot B}))(|V_+^* \backslash Y|) \end{aligned} \tag{12}$$

Algorithm 4: Generating Start Vertices and Paths

Input: $G = (V, E)$, Source vertex set $Source$
Output: all paths

```
1  V_s ← GeneratingStartVertex(V, E);
2  foreach v_s ∈ V_s do
3  |  Using DFS to generate the paths start by v_s;
Function: GeneratingStartVertex(V, E)
4  V_s.Add(Source);
5  foreach v ∈ V do // init all vertices
6  |  Activate(v);
7  |  if v ∈ Source then S(v).Add(v); min(v) ← v;
8  |  ;
9  |  else S(v) ← ∅; min(v) ← v;
10 |  ;
11 repeat
12 |  foreach active v ∈ V do
13 |  |  foreach u ∈ out-neighbor(v) do
14 |  |  |  S(u).Add(S(v)); min(u)← min{min(u), v};
15 |  |  |  if S(u) or min(u) is updated then
16 |  |  |  |  Activate(u); Signal;
17 |  |  |  else Deactivate(u);
18 |  |  |  ;
19 until no update signal;
20 foreach v ∈ V do
21 |  if S(v) == ∅ then V_s.Add(min(v));
22 |  ;
```

The first inequality follows by Lemma 1, and the second follows from the induction hypothesis. ■

Finally, Theorem 6 is proved as follows.

Proof: We define $\widetilde{w}$ as follows,

$$\widetilde{w} = \sum_{i=1}^{r+1} w(v_i) = \sum_{i=1}^{r+1} \sum_{ep \in \mathscr{E}(v_i)} \frac{1}{l_{ep}} \tag{13}$$

Then,

$$l \cdot \widetilde{w} = \sum_{i=1}^{r+1} \sum_{ep \in \mathscr{E}(v_i)} \frac{l}{l_{ep}} \geqslant \sum_{i=1}^{r+1} \sum_{ep \in \mathscr{E}(v_i)} 1 \geqslant B \tag{14}$$

In addition, we observe that if $a_1, ..., a_m \in R^+$ and $\sum_{j=1}^{m} a_j = \alpha \cdot A$, then the function $(1 - \prod_{j=1}^{m}(1 - \frac{a_j}{A}))$ achieves its minimum value of $(1 - (1 - \frac{\alpha}{m})^m)$ when $a_1 = a_2 = ... = a_m = \frac{\alpha \cdot A}{m}$.

Thus, from Lemma 2 we have the following,

$$\begin{aligned} \sum_{j=1}^{r+1} 1 &\geqslant (1 - \prod_{j=1}^{r+1} (1 - \frac{w(v_i)}{k \cdot B}))(|V_+^* \backslash Y|) \\ &\geqslant (1 - \prod_{j=1}^{r+1} (1 - \frac{w(v_i)}{k \cdot l \cdot \widetilde{w}}))(|V_+^* \backslash Y|) \\ &\geqslant (1 - (1 - \frac{1}{k \cdot l \cdot (r+1)})^{r+1})(|V_+^* \backslash Y|) \\ &\geqslant (1 - e^{-\frac{1}{kl}})(|V_+^* \backslash Y|) \end{aligned} \tag{15}$$

Algorithm 5: Generating Start Vertices

Input: $G = (V, E)$, Source vertex set $Source$
Output: the start vertices set V_s

```
1  foreach v ∈ V do
2  |  Activate(v);
3  |  if v ∈ Source then Out_0.Add((v,(v,v)));
4  |  ;
5  |  else Out_0.Add((v,(v,null)));
6  |  ;
7  i ← 0;
8  repeat
9  |  Propagation(E, Out_i);
10 |  i ← i + 1;
11 until no update signal;
```

Function: `Propagation(`E, Out_i`)`

```
   Map1: Input: edges E = {(u,w)}, Out_i = {(v,(v_m,v_s))}
1  if input is an edge (u,w) then Emit(u,w);
2  ;
3  else Emit(v,(v_m,v_s));
4  ;
   Reduce1: Input: key, values
1  foreach val ∈ values do
2  |  m ← ∞; sour ← ∅;
3  |  if val is a vertex and val is active then
4  |  |  m ← v_m; sour ← v_s;
5  |  |  Emit(key, val);
6  |  if val is a neighbor w and val is active then
7  |  |  Emit(w,(min{w, m}, sour)); // update
   Map2: Input: {(v,(v_m,v_s))}
1  Emit(v,(v_m,v_s));
   Reduce2: Input: key, values
1  Update the minimal vertex and root set (v_m,v_s) of key;
2  if nothing was updated then
3  |  Deactivate(key);
4  |  Out_{i+1} ← Emit(key,(v_m,v_s));
5  else
6  |  Activate(key); Signal;
7  |  Out_{i+1} ← Emit(key,(v_m,v_s));
```

Finally,

$$\begin{aligned} |V'_+| &= |Y| + |Y'| \\ &= 2 + r \\ &\geqslant 2 + (1 - e^{-\frac{1}{kl}})(|V^*_+ \backslash Y|) - 1 \\ &\geqslant (1 - e^{-\frac{1}{kl}})(|Y| + |V^*_+ \backslash Y|) \\ &\geqslant (1 - e^{-\frac{1}{kl}})|V^*_+| \end{aligned} \tag{16}$$

■

Algorithm of Generating All Paths

The pseudo-code of generating all start vertices and paths is given in Algorithm 4. The algorithm uses function `GeneratingStartVertex` to generate all start vertices. Specifically, all source vertices are included in the set of start vertex V_s (line 4). Initially, for each vertex v, the minimum-ID of the vertices that can reach v is itself, and a set of source vertices that can reach v contains v, if v is a source vertex or otherwise is an empty set (line 5-8). Then, the algorithm iteratively propagates the minimum-ID and the set of source

Algorithm 6: Vertex Weighting

Input: $G = (V, E)$, Source vertex set $Source$
Output: $w(v)$ for each $v \in V$

```
1  foreach v ∈ V do
2  |  Out_0.Add((v,1)) ;
3  for i ← 1 to Iteration Time do
4  |  Out_i ← VertexWeighting(E, Out_{i-1});
```

Function: `VertexWeighting(`E, Out_{i-1}`)`

```
   Map1: Input: E, Out_{i-1}
1  if input is an edge (v,u) then Emit((v,u));
2  ;
3  else Emit((v,w(v)));
4  ;
   Reduce1: Input: key,values
1  if value is a weight then w(key) ← value;
2  ;
3  else v_out.Add(value);
4  ;
5  foreach v ∈ v_out do // propagate v's weight to
   its neighbors
6  |  Emit((v,w(key)));
   Map2: Input: output of Reduce1:
1  Emit(key, value);
   Reduce2: Input: key,values
2  sum ← 0;
3  foreach value ∈ values do
4  |  sum ← sum + value;
5  Emit(key, 1 − α + α · sum/√sum);
```

vertices of an active vertex v to its out-neighbors (line 10-12). If v's out-neighbor u updates its set of source vertices or minimum-ID by those of v, u is activated, otherwise u is deactivated (line 13-15). The iteration will continue until there is no update occurred (line 16). Then, for each vertex v which has no source can reach to it, the algorithm adds the minimum-ID of v into V_s (line 18-19). After generating all start vertices, the algorithm adopts the depth-first search algorithm to generate all paths. The function `GeneratingStartVertex` can be implemented in MapReduce framework, as shown in Algorithm 5.

Algorithm of Calculating Vertex Weights

The pseudo-code of calculating vertex weights in MapReduce framework is given in Algorithm 6.