

# Subgraph Matching over Graph Federation

Ye Yuan  
Beijing Institute of Technology  
yuan-ye@bit.edu.cn

Delong Ma  
Northeastern University, China  
delong\_ma@126.com

Zhenyu Wen  
Zhejiang University of Technology  
zhenyuwen@zjut.edu.cn

Zhiwei Zhang  
Beijing Institute of Technology  
zwzhang@bit.edu.cn

Guoren Wang  
Beijing Institute of Technology  
wanggr@bit.edu.cn

## ABSTRACT

Many real-life applications require processing graph data across heterogeneous sources. In this paper, we define the *graph federation* that indicates that the graph data sources are temporarily federated and offer their data for users. Next, we propose a new framework FedGraph to efficiently and effectively perform subgraph matching, which is a crucial application in graph federation. FedGraph consists of three phases, including query decomposition, distributed matching, and distributed joining. We also develop new efficient approximation algorithms and apply them in each phase to attack the NP-hard problem. The evaluations are conducted in a real test bed using both real-life and synthetic graph datasets. FedGraph outperforms the state-of-the-art methods, reducing the execution time and communication cost by  $37.3 \times$  and  $61.8 \times$ , respectively.

## PVLDB Reference Format:

Ye Yuan, Delong Ma, Zhenyu Wen, Zhiwei Zhang, and Guoren Wang. Subgraph Matching over Graph Federation. PVLDB, 15(3): 437-450, 2022. doi:10.14778/3494124.3494129

## 1 INTRODUCTION

Graph data (e.g., web graphs, social graphs, knowledge graphs, and biological graphs) are widely used in many applications across various domains, including bioinformatics, finance, and healthcare. The development of big data increases the demand for processing (or analyzing) graph data over several sources with various domains, formats, and query interfaces. In this paper, we name this new computational paradigm *graph federation*, and an illustrative example is shown as follows.

### Example 1: Financial Graph Federation for Risk Management.

The China Banking Regulatory Commission (CBRC) includes thousands of financial institutions that may build their own graphs or networks for internal data management and analysis [40]. However, there are increasing demands for performing analytics across graphs. For example, before lending money to an enterprise, a bank should conduct due diligence for risk assessment. One of the investigations is the *related-party* analysis, which attempts to find the relation patterns among the bank accounts related to this enterprise. The related-party analysis can help identify illegal behaviors

such as money laundering by monitoring the complex sequence of banking transfers. The related-party analysis can be formalized as a party query (or subgraph matching) [18]. Figure 2(a) shows that query pattern  $Q$  is a set of normal accounts, including the target enterprise's account. However, if these accounts have a set of suspicious connections, the target enterprise may be involved in money laundering. We aim to identify these suspicious connections (i.e., query patterns) from three independent graphs ( $G_1$ ,  $G_2$  and  $G_3$  in Figure 2(b)) provided by different financial institutions of the CBRC for risk assessment. Unfortunately, many entities from these graphs have the same or similar meaning but are represented in semantical differences. As a result, we need a unified solution to build the connections among the graphs, namely, *financial graph federation*.

Graph federation is not limited to the financial field. Increasingly, enterprises have embraced data lakes. To uniformly analyze heterogeneous data, some state-of-the-art methods (e.g., federated GraphQL [47]) build a knowledge graph for every data source [9, 13, 39]. These knowledge graphs are composed of a *knowledge graph federation*. Moreover, the resource description framework (RDF) graphs in the Linked Data cloud [1] are semantically different and distributed across the world wide web. Therefore, the linked data cloud implies a *RDF graph federation*.

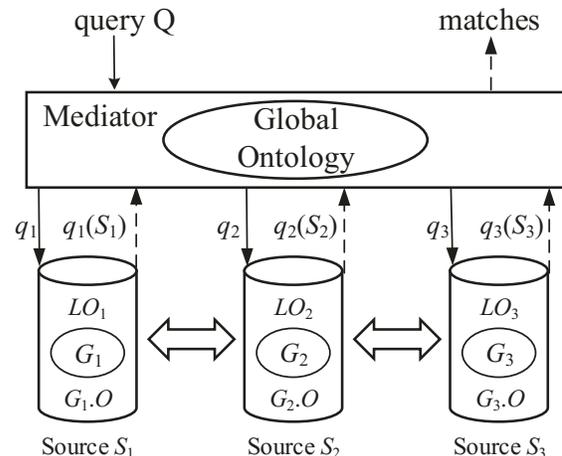


Figure 1: An architecture of graph federation.

These examples indicate that a graph federation  $GF$ -based application has a common architecture, as shown in Figure 1. To

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.  
doi:10.14778/3494124.3494129



Dec and the distributed matching algorithm Mat. Dec identifies different optimization algorithms to decompose a query  $Q$  into a set  $SQ$  of star-shaped subqueries. Mat progressively generates star matches for the pipeline joining. (4) In Section 5, we develop a distributed joining algorithm that includes logical joining and physical joining. The logical joining applies the pipeline technique to output an efficient corrected joining tree. Based on the joining tree, the physical joining generates and executes an optimized execution plan to minimize the joining process's parallel execution time and communication cost. (5) In Section 5, we design machine learning algorithms to build the global ontology that bridges the semantic gaps between different graphs. (6) In Section 7, using real-life and synthetic graphs, we experimentally verify the efficiency and scalability of our algorithms.

**Position of the work.** The graph federation also has dynamic features (i.e., every data graph changes automatically) and private features (i.e., every source may contain sensitive and private data that should be protected). In this paper, however, we concentrate on the features of autonomy, heterogeneous semantics, and different processing capabilities. We propose novel and systematic solutions to address issues incurred by these features. The dynamic and private features are very different from these features. We will extend our proposed algorithms to solve the issues incurred by the dynamic and private features in other works.

## 2 PROBLEM DEFINITION

### 2.1 Basic Concepts

**Data graph.** We consider labeled, directed, attributed graphs, defined as  $G = (V, E, L, F_A)$ , where (1)  $V$  is a finite set of nodes and  $E \subseteq V \times V$  is a set of edges. For each node  $v \in V$  (resp. edge  $e \in E$ )  $L(v)$  (resp.  $L(e)$ ) is a *type* (resp. *relation*) from a finite alphabet. The value of each node  $v$  is denoted as  $v.val$ .  $v.val$  is an example of an *attribute* of  $v$ , describing a node property. For each node  $v$ , its attributes  $A_i \in A$ ,  $i \in [1, n]$  are captured in its *property tuple*,  $F_A(v)$ , defined as a sequence of attribute-value pairs  $\{(v.A_1, a_1), \dots, (v.A_n, a_n)\}$ . Each pair  $(v.A_i, a_i)$  states that the attribute  $v.A_i = a_i$ .

**Graph federation.** The integration of distributed graph sources via a central mediator implies a *graph federation*  $GF$ .  $GF$  consists of a mediator and a set of graph sources. The mediator and graph sources also maintain *ontology graphs* (introduced later). The ontology graphs capture the semantics of different data graphs in sources and can be used to map queries to sources. More specifically, a graph federation needs the following basic components and functionalities:

**Graph sources.** A collection of  $n$  graph sources, or simply sources  $GS = \{S_1, \dots, S_n\}$ , agree to collectively support query services over their data graphs. Each source  $S_i$  manages its own data graph  $G_i$ . An  $S_i$  also maintains a set of *out-nodes*  $G_i.O$ , each of which is also a node of  $G_j$  in another source  $S_j$ . An out-node  $v \in G_i.O$  can be owned by multiple sources. In practice, multiple knowledge graphs have the same entity. In Section 6, we introduce how to determine the same out-nodes (e.g., entities) between different data graphs.

In summary, a source  $S_i$  maintains the graph  $G_i$  as well as the out-node set  $G_i.O$ . Considering all the sources, a graph federation

has a *virtual graph*  $VG$  composed of data graphs from every  $S_i$ ; i.e.,  $VG = \bigcup_{i=1}^n G_i$ . The virtual graph is physically distributed across  $n$  sources. This graph is used to define our problems in the next section.

Figure 2(b) shows three data graphs  $G_1, G_2$  and  $G_3$  for sources  $S_1, S_2$  and  $S_3$ , respectively. Every  $S_i$  also maintains an out-node set  $G_i.O$ , denoted by bold circles. For example,  $S_1$  maintains  $G_1.O = \{co_1, co_2\}$ . To see this, nodes  $co_1$  and  $co_2$  are also maintained in  $S_2$ , represented as  $en_1$  and  $en_2$ <sup>1</sup>. For the same reason, nodes  $sp_1$  and  $cr_2$  are the out-nodes of  $S_2$  and  $S_3$ , respectively.

Below, we introduce two features of graph sources, based on which we model the protocols of computation and communication over the graph federation  $GF$ .

#### (1) Heterogeneous data sizes and semantics.

Different sources build their own data graphs without a uniform standard. Different data graphs thus may have heterogeneous data sizes and semantics. Heterogeneous data semantics refer to those nodes of different data graphs with distinct labels that may refer to the same entity. Ontology-based mapping is an effective mechanism to bridge heterogeneous data semantics [23, 57]. For example,  $LO_1$  and  $LO_2$  (in Figure 3) are the local ontologies of  $G_1$  and  $G_2$  in (Figure 2), respectively.  $LO_1$  and  $LO_2$  form a global ontology  $GO$ . There are mappings from  $LO_1$  (or  $LO_2$ ) to  $GO$ , such as  $\pi(co) = en$  (represented by the dotted lines in Figure 3), which means they are the same concept but have different names.

**Ontology.** An ontology graph  $O = (V_o, E_o)$  is a directed graph, where  $V_o$  is a set of concept labels or attributes and  $E_o \subseteq V_o \times V_o$  is a set of semantic relations among the nodes. In practice, an edge  $(v, v') \in E_o$  may encode 6 types of relations [30, 36]: (a) *equivalence*, which states that  $v$  and  $v'$  are semantically equivalent; (b) *hyponym*, which is that  $v$  is a kind (or subclass) of  $v'$ ; (c) *property*, which states that  $v$  is a property of  $v'$  in terms of 'association' or 'part-of' relation; (d) *cause*, which states that  $v$  is caused by  $v'$ ; (e) *location*, which states that  $v$  is a location of  $v'$ ; and (f) *temporal*, which states that  $v$  is the temporal information of  $v'$ .

The relations in an ontology may vary with different domains, but most domains include the above 6 relations.

A node  $u \in V_o$  is a *label ancestor* (resp. *label offspring*) of a node  $v \in V_o$  if there is a path from  $u$  (resp.  $v$ ) to  $v$  (resp.  $u$ ) in  $O$  with a sequence of "hyponyms" and "equivalences". Intuitively,  $u$  is a subclass of  $v$  if  $u$  is a label ancestor of  $v$ .

Figure 3(c) shows that nodes  $km$  and  $sp$  have an equivalent relation and that node  $en$  is a subclass of node  $eu$ . Additionally, in the figure, node  $ie$  is an offspring of node  $eu$ , and thus,  $ie$  is a subclass of  $eu$ .

Every source maintains a local ontology  $LO_i$  of  $G_i$ . We assume that  $LO_i$  contains all the labels of  $G_i$ .  $LO_i$  can be obtained from  $G_i$  by merging nodes with the same type of label, as introduced in the works [59].

The mediator  $M$  keeps a global ontology  $GO$  of  $VG$ .  $LO_i$  (resp.  $GO$ ) can be viewed as a schema that is a summary of the node and edge labels of  $G_i$  (resp.  $VG$ ). For any concept label  $cl$  of a  $LO_i$ , there is a mapping  $\pi$  from  $cl$  to a concept label  $cl'$  of  $GO$ ; i.e.,

<sup>1</sup>co and en are short for company and enterprise, as shown in Figure 3. co and en refer to the same concept via ontology mapping.

$\pi(cl) = cl'$ .  $cl$  and  $cl'$  refer to the same concept, although they may have different names. These concepts are generalizations (i.e., superconcepts) of related concepts in local ontologies. In Section 6, we introduce how to compute a global ontology  $GO$  from local ontologies. Users can write queries based on  $GO$ .

(2) *Heterogeneous computational and storage capabilities.*

Every source  $S_i$  is autonomous and manages its data without any interference from other sources. Each  $S_i$  voluntarily participates in a graph federation  $GF$  by using a *capsule*  $C_i$ , which is a logic unit.  $C_i$  indicates the shared data graph of  $S_i$ . Since each  $S_i$  is autonomous and voluntary, it offers limited and various storage and computing resources. We define the resource constraints of each  $S_i$  as  $c_i$ ; i.e., the consumed memory at every computing time is less than  $c_i$ .

## 2.2 Subgraph Matching

**Query graph.** A query graph is a directed graph defined as  $Q = (V_q, E_q, L_q, F_q)$ , where (1)  $V_q$  and  $E_q$  are a set of query nodes and query edges, respectively; (2)  $L_q$  is a labeling function such that for each node  $v \in V_q$  (resp.  $e \in E_q$ ),  $L_q(v)$  (resp.  $L_q(e)$ ) is a node (resp. edge) label; and (3) for each node  $v \in V_q$ ,  $F_q(v)$  specifies the set  $\{A_1, \dots, A_k\}$  of its attributes. Note that  $L_q(v)$  or  $F_q(v)$  (resp.  $L_q(e)$ ) is a node (resp. edge) label of the global ontology  $GO$ . Users can format their queries based on  $GO$ .

**Ontology-based subgraph matching (OSM).** Given a query graph  $Q = (V_q, E_q, L_q, F_q)$ , a data graph  $G = (V, E, L, F_A)$  and an ontology  $O$ , the OSM finds the subgraphs  $G' = (V', E', L', F'_A)$  of  $G$ , such that there is a bijective function  $h$  from  $V_q$  to  $V$  where (1) for each node  $u \in V_q$ , (a)  $L(h(u))$  is a label ancestor (or label offspring) of  $L_q(u)$  in  $O$ , and (b)  $F_A(h(u)).A_i$  is a label ancestor (or label offspring) of  $F_q(u).A_i$  in  $O$ ; and (2)  $(u, u')$  is a query edge if and only if  $(h(u), h(u'))$  is an edge of  $G'$ . We refer to  $G'$  as a match of  $Q$  in  $G$  induced by the mapping  $h$  and denote all the matches in  $G$  for  $Q$  as  $Q(G)$ .

**Problem definition.** Given a query graph  $Q = (V_q, E_q, L_q, F_q)$  and a graph federation  $GF$  with the virtual graph  $VG$ , our problem is to find a set of matches  $Q(VG)$  of  $Q$  in  $VG$  via OSM. A match  $m \in Q(VG)$  is called a crossing match if  $m$  contains nodes in different sources.

**Example 2:** Figure 4 shows the query answers of OSM  $Q$  over the virtual graph  $VG$ . We take an example of query node  $km$  mapping to the data node  $ac_1$  of  $G_1$  as follows. First,  $L'_s(ac_1) = ac$  in  $LO_1$ , and  $L_q(km) = km$  in  $GO$ . Then, based on the mapping  $\pi$  from  $LO_1$  to  $GO$ ,  $\pi(ac)$  is the child of  $km$  in  $GO$ , i.e., equivalence.  $\square$

Clearly, subgraph matching over graph federation is an NP-hard problem. This is because its subproblem—subgraph isomorphism—is NP-hard [15]. In the next section, we propose a novel algorithmic framework to address it.

## 3 FRAMEWORK OF THE QUERYING ALGORITHM

In this section, we propose a general federated graph querying framework, namely, FedGraph, which includes an online phase and an offline phase. The online phase solves the OSM problem in a decomposition-matching-joining manner, consisting of three steps.

Step 1) *Query decomposition.* The inputs of FedGraph are a graph federation  $GF = (M, GS = \{S_1, \dots, S_n\})$  and a query graph  $Q$ . Once a  $Q$  is submitted to  $M$ , a procedure Dec is invoked to decompose  $Q$  into a set of star queries  $SQ$  (detailed in Section 4.1). A star query contains a pivot node and a set of leaves as its neighbors in  $Q$ . After query decomposition,  $SQ$  is sent to  $GS$  for performing distributed matching.

Step 2) *Distributed matching.* Taking the input from Step 1, FedGraph invokes a procedure Mat to efficiently generate all matches for each star query in  $SQ$  over  $VG$  (see Section 4.2). The procedure Mat progressively generates the matches, which are fed into the pipeline join in the next step.

Step 3) *Distributed joining.* The matches produced by Mat for multiple star queries are then joined in pipeline parallel by a Join procedure to produce complete matches of  $Q$  (see Section 5). Join includes logical and physical joining, aiming at minimizing the parallel execution time and network overhead. Specifically, logical joining outputs a joining plan that minimizes the time cost. Physical joining executes a joining plan to minimize the traffic cost.

The offline phase of FedGraph is the determination of the out-nodes of graph sources and the construction of the ontologies (see Section 6). The online phase needs the out-nodes and ontologies to execute the three steps.

## 4 QUERY DECOMPOSITION & MATCHING

### 4.1 Query Decomposition

In this section, we propose an algorithm Dec that decomposes a query  $Q$  into a set of star subqueries  $SQ = \{q_1, \dots, q_m\}$ . Dec aims to achieve the following two goals.

**Goal 1:** The number of decomposed star queries should be as small as possible, which intuitively reduces the number of joins.

**Goal 2:** The number of out-nodes of star matches should be as small as possible, which intuitively reduces the communication costs through these out-nodes.

**Achieving goal 1.** To achieve goal 1, we have to uncover the following problem: Let  $Q$  be a query graph and  $SQ = \{q_1, \dots, q_m\}$  be a set of stars such that any edge of  $Q$  belongs to only one star  $q_i \in SQ$ .  $SQ$  is called the *star cover* of  $Q$ . The problem consists of computing the minimum star cover of  $Q$ . It is not difficult to prove that the minimum star cover problem is polynomial equivalent to the minimum node cover problem, which is NP-hard. As a result, our problem is an NP-hard problem.

We leverage the 2-approximate algorithm [51] to construct a star cover from a node cover in polynomial steps, detailed as follows. In every step, we randomly select an edge  $(a, b)$ , add  $a$  and  $b$  to the answer, and remove all edges incident to  $a$  or  $b$ . This process is repeated until we remove all of the edges. We use the same process to create a 2-approximate star cover.

To realize goal 2, Dec revises the query decomposition algorithm by picking edges with higher *selectivity* as follows.

**Achieving goal 2.** We first calculate a selectivity  $s(u)$  of a node  $u \in Q$ .  $VG.O$  denotes a set of out-nodes of  $VG$ ; i.e.,  $VG.O = \bigcup_{i=1}^n G_i.O$ .

For nodes  $u \in Q$  and  $v \in VG.O$ ,  $|LO(u)|$  denotes the number of out-nodes  $v$  such that  $h(u) = v$ . We then define the selectivity as  $s(u) = \frac{1}{|LO(u)|}$ . Intuitively, the larger  $s(u)$  is, the smaller the

probability of  $h(u)$  being an out-node, where  $h$  is the mapping of  $Q$  to  $VG$  (i.e.,  $u$  and  $v$  have an ancestor/descendent relationship in the global ontology).

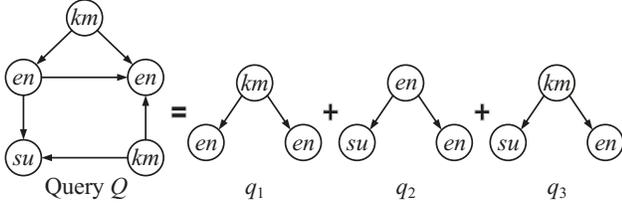


Figure 5: Query decomposition of  $Q$  in Figure 2(a).

Figure 6 outlines the detailed steps of Dec. Dec simulates the process of the approximate algorithm for the node cover problem. Instead of selecting a common edge  $(u, v)$  of  $Q$  for the node cover, Dec picks an edge  $(u, v)$  such that  $s(u) + s(v)$  is the largest. This difference allows Dec to achieve the above two goals.

#### Algorithm Dec

*Input:* a query  $Q$ .

*Output:* a set of decomposed star subqueries  $SQ$ .

1. initialize  $SQ = \phi$ ;  $R = \phi$ ;
2. **while**  $Q$  has edges **do**
3.   **if**  $R = \phi$  **then**
4.     select an edge  $(u, v)$  such that  $s(u) + s(v)$  is the largest;
5.   **else**
6.     select an edge  $(u, v)$  such that  $u \in R$  and  $s(u) + s(v)$  is the largest
7.    $T_u =$  the stars rooted at  $u$ ;
8.   add  $T_u$  to  $SQ$ ;
9.    $R = R \cup \text{neighbor}(u)$ ;
10.   remove  $T_u$  from  $Q$ ;
11.   **if**  $\text{deg}(v) > 0$  **then**
12.      $T_v =$  the star rooted at  $v$ ;
13.     add  $T_v$  to  $SQ$ ;
14.     remove all edges in  $T_v$  from  $Q$
15.      $R = R \cup \text{neighbor}(u)$ ;
16.   remove  $u, v$  and all nodes with degree 0 from  $R$ ;
17. **return**  $SQ$ ;

Figure 6: Algorithm of query decomposition Dec.

Based on the two goals, the query  $Q$  in Figure 2(a) can be decomposed into 3 star subqueries with the smallest number of stars and out-nodes, as illustrated in Figure 5.

## 4.2 Distributed Matching

After query decomposition, we obtain a set of star queries  $SQ$ . Our distributed matching scheme aims to maximize the parallelism of the query  $q \in SQ$  over all sources  $\{S_i\}$ . To achieve this, the mediator  $M$  sends  $q \in SQ$  to every source  $S_i$ . Every  $S_i$  then computes the

#### Algorithm Mat

*Input:* a star query  $q$  and source  $S_i$ .

*Output:* the match set  $q(G_i)$ .

1. let  $u$  be the pivot node of  $q$ ;
2. **for each** node  $l \in V(LO_i)$  **do**
3.   **if**  $\text{Reach}(L(u), \pi(L(l)))$  or  $\text{Reach}(\pi(L(l)), L(u))$  **then**
4.     let  $\text{Cand}(u) = \{v\}$  be the set of nodes of  $G_i$  with label  $L(l)$ ;
5.     **if**  $\text{Reach}(u.A_i, v.A_i)$  or  $\text{Reach}(v.A_i, u.A_i)$
6.       add edges from  $v$  to nodes of  $q(G_i)$ ;
7.   **for each**  $v \in \text{Cand}(u)$  **do**
8.     **for each**  $w \in \text{Neighbor}(v)$  **do**
9.       **for each** leaf node  $x$  of  $q$  **do**
10.        **if**  $\text{Reach}(L(w), \pi(L(x)))$  or  $\text{Reach}(\pi(L(x)), L(w))$  **then**
11.         **if**  $\text{Reach}(w.A_i, x.A_i)$  or  $\text{Reach}(x.A_i, w.A_i)$
12.         add edges from  $w$  to nodes of  $q(G_i)$ ;

Figure 7: Algorithm of distributed matching Mat.

match set  $q(G_i)$  of  $q$  in  $G_i$  in parallel. Figure 7 shows the detailed steps of Mat, where the inputs of Mat are a star query  $q$  and the source  $S_i$ . Its output is the match set  $q(G_i)$ .

Mat takes the following steps. (1) For the pivot node  $u$  of  $q$ , Mat computes the candidate match set  $\text{Cand}(u)$  of  $u$  in  $G_i$  (lines 1-6). To obtain  $\text{Cand}(u)$ , Mat needs the local ontology  $LO_i$  of  $G_i$  and the global ontology  $GO$ . Recall that a node  $l \in LO_i$  has a mapping node  $\pi(l)$  in  $GO$  and that the label  $L(u)$  of  $u$  is a node of  $GO$ . We thus justify that  $u$  matches  $l$  if  $L(u)$  is an ancestor or offspring of  $L(\pi(l))$  in  $GO$  (line 3) (the same as for their properties (line 5)). In this case,  $u$  may be a subclass or superclass of  $l$ . The function  $\text{Reach}(L(u), L(v))$  determines such a case.  $\text{Reach}()$  speeds up the determination based on a reachability index [53]. (2) Using the same idea, Mat calculates the matches of leaf nodes of  $q$  (lines 7-12). Once all leaf nodes find their matches, Mat returns the match set  $q(G_i)$ . Mat also optimizes this process based on a good match order [55].

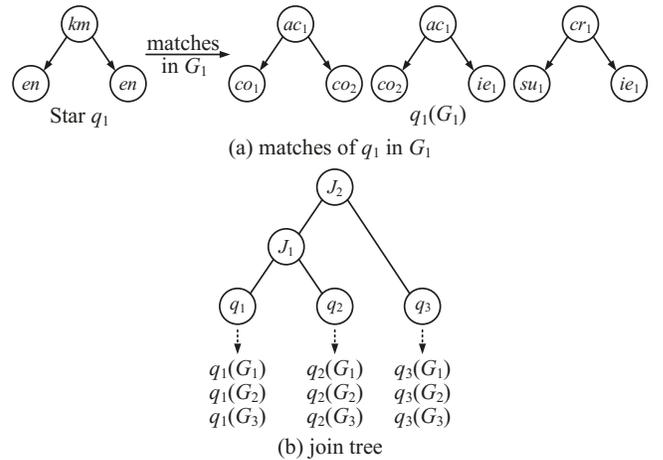


Figure 8: Example of matching and joining tree.

For example, Figure 8(a) shows the matches of star  $q_1$  in  $G_1$ , represented as  $q_1(G_1)$ . Query  $Q$  in Figure 2(a) is decomposed into 3 star subqueries  $q_1, q_2$  and  $q_3$ , as shown in Figure 5. The mediator sends these stars to every source. We aim to obtain all the matches. We use  $q_1$  to match  $G_1$  as an example (see Figure 8(a)). We first identify the node matches  $ac_1$  and  $cr_1$  for the pivot node  $km$ . By probing their neighbors, we then find all the matches for the leaf nodes of  $q_1$  (i.e.,  $en$ ). Finally, we have all the matches of  $q_1$  in  $G_1$ , as shown in Figure 8(a).

Note that any node of a star query has only one-hop neighbors in the star. Additionally, a source contains also one-hop neighbor nodes (i.e., out-nodes) of other sources. Therefore, a match  $q(G_i)$  is entirely within a source.

## 5 DISTRIBUTED JOINING

Given  $Q$  decomposed to a set of star queries  $SQ = \{q_1, \dots, q_m\}$ , distributed joining finds all the matches for  $Q$  by assembling the matches retrieved by Mat on each  $q_i$ . We aim to achieve two goals in this section:

(1) Logical joining. A complete match is constructed from star matches. Logical joining outputs a joining tree that guarantees a corrected joining answer.

(2) Physical joining. A logical joining plan is executed over graph sources. Physical joining aims at minimizing the parallel execution time and the network overhead.

### 5.1 Logical Joining

In this section, we develop the algorithm of logical joining LogJoin to obtain a complete match of  $Q$ . Given a query  $Q$ , we decompose it to a set of star queries  $SQ = \{q_1, \dots, q_m\}$ , denoted  $R(q_i)$  as the star matches ( $i \in [1, m]$ ). LogJoin computes the set of matches  $R(Q)$  as:

$$R(Q) = R(q_1) \bowtie R(q_2) \bowtie \dots \bowtie R(q_m). \quad (1)$$

**Joining tree.** A join plan determines the order for solving the above joining and processes  $m$  rounds of two-way joins. We denote  $P_i$  as the  $i$ -th partial pattern, whose results are produced in the  $i$ th round of the joining plan. Obviously, we have  $P_m = Q$ . The joining plan is presented in a tree structure, where the leaf nodes are the stars and the internal nodes are the partial patterns. A joining tree  $T$  uniquely specifies a joining plan. If all the internal nodes of the joining tree have at least one joining operation as their child, the tree is called a *left-deep tree* [27]. Otherwise, it is a bushy tree [27]. Note that a left-deep tree is a version of bushy tree.

LogJoin advocates the left-deep tree; i.e., LogJoin applies a sequence of two-way joins to compute  $R(Q)$ . This is because a left-deep tree incurs much fewer partitions (i.e., disjoint subtrees) than a bushy tree. This feature allows the following algorithms to be effectively applied to the distributed joining.

Figure 8(b) demonstrates a joining tree  $T$  for the three stars in Figure 5. The leaf nodes of  $T$  are  $q_1, q_2$  and  $q_3$ , each of which relates to the star matches.

**Optimizing the joining procedure.** We use the joining pipeline to perform Equation 1; i.e., we progressively join star matches generated by Mat. The pipeline performs joining operations once star matches are generated without waiting for all the matches to be

produced. The pipeline technique ensures that the consumed memory of the algorithms in each source satisfies the storage constraint  $c_i$ . We can select a joining order that minimizes the intermediary results of Equation 1. In this paper, we apply the sample-based joining cost estimation method and cost-based joining order selection method [14] to compute an optimal joining order.

### 5.2 Physical Joining

The previous sections discuss the algorithms used to obtain a logical joining plan; this plan needs to be transferred to a physical joining plan for distributed execution. Given a joining tree  $T$ , the physical joining plan aims to maximize the parallel execution and minimize the network overhead of  $T$ .

**Idea of physical joining.** The key idea of physical joining is to optimize the parallel execution and communication costs via *logical scheduling* and *physical scheduling*, respectively.

Logical scheduling partitions  $T$  into disjoint subtrees such that these subtrees can be executed in a maximum parallel form. Physical scheduling is an assignment of these subtrees to sources such that the communication cost of executing every subtree is minimized.

Both optimization problems are NP-complete as stated later. To solve the two problems, we propose two polynomial time algorithms with approximation ratio bounds  $8\epsilon$  ( $\epsilon$  is a constant) and  $\log n$ , respectively.

**5.2.1 Logical Scheduling.** Given a left-deep joining tree  $T$ ,  $T$  has  $m$  leaf nodes based on Equation 1. A leaf node of  $T$  represents a set of star matches  $R(q_i)$  across sources.<sup>2</sup> These sources can be viewed as a *virtual machine*. Thus,  $T$  corresponds to a set of  $m$  virtual machines. Logical scheduling is a partitioning of  $T$  to  $m$  virtual machines such that the parallel execution time of  $T$  is *minimized*.

This idea is similar to the classical problem of multiprocessor scheduling with homogeneous environments (i.e., machines with the same computational speeds [16]). However, our environment is heterogeneous, as the  $m$  virtual machines have a variety of computing resources. This is because every source has heterogeneous processing capabilities, as introduced in Section 2. In this paper, we propose novel solutions to the heterogeneous scenario. Before introducing our solutions, we first define the following.

We first define a *weighted operator tree* PT originating from the joining tree  $T$ . PT shares the same structure with  $T$ , i.e., the same nodes and edges of  $T$ . The weight  $t_k$  of node  $k$  is the time to run the operator. The operator is joined if  $k$  is an internal node of  $T$ , and the operator is matched otherwise. The weight  $c_{kj}$  of the edge from node  $k$  to node  $j$  is the network overhead that both  $k$  and  $j$  incur for communication if they are scheduled on different virtual machines. The values of  $t_k$  and  $c_{kj}$  can be obtained by the sample-based cost estimation method for the joining order [14].

Figure 10(a) shows the PT of the joining tree in Figure 8(b). We also need the following definitions.

**Definition 5.1.** Given  $m$  virtual machines and an operator tree  $PT = (V_{pt}, E_{pt})$ , a scheduling of PT is a partition of  $V_{pt}$  into  $m$  sets  $F_1, \dots, F_m$ , with set  $F_k$  allocated to virtual machine  $k$ . The load  $L_k$  on the virtual machine  $k$  is the cost of executing all nodes in  $F_k$

<sup>2</sup> $R(q_i)$  might not span all the sources.

plus the overhead of communicating with nodes on other virtual machines;  $L_k = \sum_{i \in F_k} [t_i + \sum_{j \notin F_k} c_{ij}]$ .

Every virtual machine has a relative computing speed  $s_i$  ( $1 \leq i \leq m$ ). We assume these speeds have been normalized such that  $s_1 = 1$ ,  $s_i \geq 1$  for  $2 \leq k \leq m$ . The response time of scheduling is  $\lambda = \max_{1 \leq k \leq m} \frac{L_k}{s_k}$ . The logical scheduling computes a partition of  $V$  into  $F_1, \dots, F_m$  that minimizes  $\lambda$ .

This problem is intractable since the special case in which all edge weights are zero is the NP-complete problem of multiprocessor scheduling [15]. Note that our algorithm (LogSch) is not parameterized by  $s_i$  ( $1 \leq i \leq m$ ), which is used to analyze only LogSch. Therefore, we do not need to estimate the values of  $s_i$ .

To schedule a PT, two operations, introduced as follows, are used to modify it:

*Definition 5.2.*  $Collapse(i_1, i_2)$  collapses nodes  $i_1$  and  $i_2$  in tree PT, where  $i_1$  and  $i_2$  are replaced by a new node  $i$ . The weight of the new node  $i$  is the sum of the weights of the two deleted nodes; i.e.,  $t_i = t_{i_1} + t_{i_2}$ . The edge between  $i_1$  and  $i_2$  is detected if it exists. All other edges connected to either  $i_1$  or  $i_2$  are instead connected to  $i$ .

$Cut(i, j)$  modifies a PT by deleting edge  $(i, j)$  and adding its weight to that of nodes  $i$  and  $j$ ; i.e.,  $t_i^{new} = t_i^{old} + c_{ij}$  and  $t_j^{new} = t_j^{old} + c_{ij}$ .  $\square$

Now, we propose the logical scheduling algorithm LogSch. Figure 9 shows the steps of LogSch. LogSch consists of two phases: *partitioning* and *the actual scheduling*.

Given an operator tree PT, LogSch performs a subprocess ColCut (for partitioning, line 1), followed by another subprocess LPT (for scheduling, line 2). LPT is a classic scheduling algorithm [15]. LPT assigns tasks to machines in decreasing order of their processing times. A task is taken from this list and is assigned to a machine whose finishing time is the earliest. ColCut partitions PT by iteratively applying  $Collapse(i_1, i_2)$  and  $Cut(i, j)$  to PT.

As shown in Figure 9, ColCut iteratively selects a leaf and decides whether to collapse or cut the edge from the leaf of its parent (line 1). It determines the operation based on the ratio of the leaf weight to the edge weight to its parent. If the ratio is greater than an input parameter  $\alpha$  ( $\alpha > 1$ ), it cuts the edge (lines 2-3). If the ratio is less than  $\alpha$ , it collapses the leaf to its parent (lines 4-5). This is because the weight of the parent node does not greatly increase. In ColCut, a parent node is defined as a node whose child nodes are leaves. In our algorithm, we set the value of  $\alpha$  to 3.

**Example 3:** Figure 10 shows the process of LogSch for the PT in Figure 10(a). LogSch collapses the leaf node  $q_1$  to its parent  $J_1$  because the new weight of  $J_1$  does not increase substantially. For the same reason, LogSch also collapses the leaf node  $q_3$  to its parent  $J_2$ . LogSch cuts the edge between the leaf node  $q_2$  and its parent  $J_1$  since this operation does not incur the weight of the resulting partitions. Figure 10(b) gives the final partition of PT.  $\square$

In the following, we apply LogSch to the problem in Definition 5.1, for which we prove that LogSch provides a constant approximation ratio bound. Here, denote  $L$  as the total load; i.e.,  $L = \sum_{k=1}^p L_k$ . The superscript  $*$  is used to denote the quantities in the optimal scheduling.

We first have a lemma for a partitioning the  $\phi$  of the PT of the optimal scheduling.

---

### Algorithm LogSch

*Input:* an operator tree PT,  $p$  virtual machines.

*Output:* a schedule of PT to  $p$  virtual machines.

1. invoke ColCut(PT) to partition PT into sets  $F_1, \dots, F_p$ ;
2. invoke LPT() to schedule  $F_1, \dots, F_p$  over  $p$  virtual machines;

#### Procedure ColCut(PT)

1. **while** (there exists a parent node  $m$  with child  $j$ ) **do**
  2.   **if** ( $p_j > \alpha c_{jm}$ ) **then**
  3.      $Cut(j, m)$ ;
  4.   **else**
  5.      $Collapse(j, m)$ ;
- 

Figure 9: Algorithm of logical scheduling LogSch.

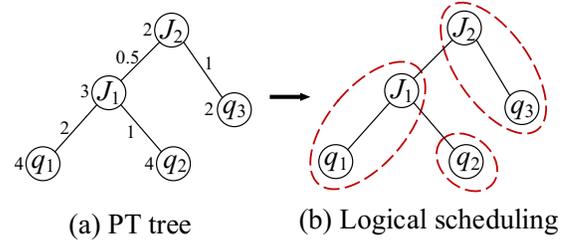


Figure 10: Example of logical scheduling.

**LEMMA 5.3.** *Suppose that there exists a mapping function  $f$  from the fragments produced by partitioning the  $\phi$  of the PT to fragments of the optimal scheduling that satisfies the following conditions: (1)  $f$  is total on its domain, and (2) for all  $F_k$  ( $k \geq 1$ ) such that  $f(F_k) = F_j^*$  ( $j \geq 1$ ), we have  $\frac{\sum L_k}{L_j^*} \leq r$  ( $r > 0$ ).*

*After LPT is executed for the partitions of  $\phi$ , we have the approximation ratio bound at most  $r\epsilon$ , where  $\epsilon$  is the ratio bound of LPT.*  $\square$

**Proof.** First, we make a schedule  $\Psi$  from the optimal schedule. Each fragment  $F_k$  of  $\phi$  is assigned to the virtual machine such that  $f(F_k)$  has been assigned to the optimal scheduling. Because of totality of  $f$ , we can schedule all fragments of  $\phi$  in this manner. From our assumption, if  $\lambda_1$  is the response time of  $\Psi$ , we have  $\frac{\lambda_1}{\lambda_1^*} \leq r$ . Clearly, if  $\lambda_1^*$  is the response time of the optimal scheduling of fragments produced by  $\phi$ , we have  $\lambda_1^* \leq \lambda_1$ . Therefore, scheduling these fragments by LPT yields a response time of  $\lambda \geq \epsilon\lambda_1$ . Then  $\frac{\lambda}{\lambda^*} \leq r\epsilon$ .  $\square$

To define the following theorem, we need a concept *main node*. The main node (denoted by  $\mu(F)$ ) of a partition  $F$  is the highest level node in the partition. Clearly, every partition has one and only one main node.

**THEOREM 5.4.** *LogSch provides an approximation ratio bound of  $8\epsilon$ , where  $\epsilon$  is the ratio bound of LPT.*  $\square$

**Proof.** We define a relation  $f$  from partitions of ColCut to those of optimal solution such that  $f(F_k) = F_j^*$ , if and only if  $\mu(F_k)$

belongs to  $F_j$ . Clearly,  $f$  is a total function on its domain because each partition of ColCut has one and only one main node and each node of PT belongs to one and only one partition in its optimal solution.

The value of  $r = \frac{\sum L_k}{L_j^*}$  such that  $\forall k f(F_k) = F_j^*$  is maximized when  $\sum L_k$  is maximized and  $L_j^*$  is minimized. Assume that  $f$  maps  $p$  partitions  $F_1, \dots, F_p$  to one partition  $F_j^*$ . Because of the definition of  $f$ ,  $F_j^*$  must have at least  $p$  nodes.  $F_1, \dots, F_p$  must be connected because they are mapped into a partition.  $L_j^*$  is minimized, if  $F_j^*$  cuts all incident edges of the main nodes of  $F_1, \dots, F_p$ , except those edges that connect the main nodes to each other. This is because for every edge  $e_{kj}$  we have,  $c_{kj} < L_k - c_{kj}$ .  $L_j^*$  is also minimized if  $F_j^*$  collapses all edges which connect  $\mu(F_1), \dots, \mu(F_p)$  to each other. We therefore have,  $\sum_{k=1}^p L_k = \sum_{k=1}^p [m_k + \sum_{j=1}^q t_j + \sum_{l=1}^p c_l]$ , in which  $m_k$  is the weight of  $\mu(F_k)$ ,  $t_j$  is the weight of the child node  $j$  of  $\mu(F_k)$  that belongs to  $F_k$  and  $c_l$  is the weight of the connecting edge between partitions incident  $\mu(F_k)$ . For each boundary node  $j$  of  $F_k$ ; we assume that  $m_j$  and  $t_j$  are the sum of node weights plus the weights of all edges incident to  $j$  that are not in  $F_k$ .

Because  $F_1, \dots, F_p$  form a subtree, and each connecting edge between  $F_k$  and  $F_j$  is considered for computing the cost of both  $F_k$  and  $F_j$ , we have,  $\sum_{k=1}^p L_k = \sum_{k=1}^p [m_k + \sum_{j=1}^q t_j] + 2 \sum_{l=1}^{p-1} c_l$ .

Since connecting edges are cut by ColCut, we have

$$\sum_{k=1}^p L_k < \sum_{k=1}^p [m_k + \sum_{j=1}^q t_j] + \frac{2}{\alpha} \sum_{k=2}^p L_k$$

Let  $\alpha > 2$  so

$$\sum_{k=1}^p L_k < \frac{\alpha}{\alpha-2} \sum_{k=1}^p [m_k + \sum_{j=1}^q t_j]$$

For  $L_j^*$  we have,  $L_j^* = \sum_{k=1}^p [m_k + \sum_{j=1}^q c_j]$  in which  $c_j$  is the weight of the cut edge  $j$  incident to  $\mu(F_k)$ . Note that  $j$  cannot be a connecting edge between the main nodes. Since these edges are collapsed by ColCut, we have

$$\frac{\sum_{k=1}^p L_k}{L_j^*} < \frac{\sum_{k=1}^p [m_k + \sum_{j=1}^q t_j] + \frac{2}{\alpha} \sum_{k=2}^p L_k}{\sum_{k=1}^p [m_k + \frac{1}{\alpha} \sum_{j=1}^q t_j]} < \frac{\alpha}{\alpha-2} < \frac{\alpha^2}{\alpha-2}$$

The above ratio is minimized when  $\alpha = 4$ , and the minimum value of  $r$  is 8. This completes the proof.  $\square$

**5.2.2 Physical Scheduling.** Consider a joining operation  $J = A \bowtie B$  (i.e., an internal node of T). Since T is a left-deep tree, pattern A corresponds to partial matches  $R(A)$ , and pattern B corresponds to star matches  $R(B)$ . All these matches are distributed over the sources in  $MS = \{S_1, \dots, S_n\}$ . Physical scheduling studies how to assign  $R(A)$  and  $R(B)$  to the sources in  $MS$  such that the communication cost of executing  $A \bowtie B$  is *minimized*. Physical scheduling is performed for every joint node of T from bottom to top.

The problem is an NP-hard reduction from the distributed joining optimization for relational databases [52]. We thus propose an approximation scheme (PhySch) in this subsection. Specifically, PhySch generates a plan  $\xi$  for  $Q$  by a reduction from the minimum set cover (MSC) problem [51].

The MSC problem is defined as follows: given a universe set of elements  $U$ ,  $X$  is a collection of weighted sets whose union covers  $U$ . We aim to find an  $X'$  that covers all elements in  $U$  and

simultaneously has the minimum total weight of all such subsets of  $X$ .

We design PhySch by applying an approximation-preserving reduction [24] from the MSC such that PhySch can optimize the joining overhead by the current approximate algorithms for the MSC.

**Reduction.** The idea of the reduction is (a) to represent  $J = A \bowtie B$  over  $R(A) \cup R(B)$  as a cost plan that assigns necessary data movement for answering  $J$  and (b) to transform the assignment problem as a variant of the MSC that admits a PTIME logarithmic-factor approximation algorithm [51].

Consider  $J = A \bowtie B$  over  $R(A) \cup R(B)$ , and an instance of MSC is constructed as follows. Given a universe  $U$  of elements and a set  $X$  of weighted subsets of  $U$ , each solution to MSC with an approximation ratio of  $c$  encodes a distributed joining plan for  $J$  with a cost of at most  $c$ -times the minimum cost of all plans for  $J$ .

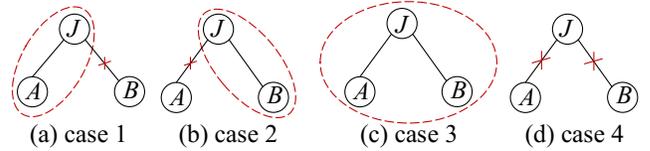


Figure 11: Four cases of physical scheduling.

Four cases from logical scheduling affect PhySch; these cases are introduced as follows.  $VM$  is the virtual machine to which logical scheduling assigns a node of PT.  $I_A$  is the set of sources making up  $VM_J$ ; similarly, for  $I_B$  corresponding to  $B$ .  $I_J$  is the set of candidate sources to which PhySch performs the join  $J$ . Figure 11 illustrates the four cases.

- Case 1:  $J$  and  $A$  are in the same  $VM$ , and  $B$  is in another  $VM$ . Then,  $I_J = I_A \setminus I_B$ .
- Case 2:  $J$  and  $B$  are in the same  $VM$ , and  $A$  is in another  $VM$ . Then,  $I_J = I_B \setminus I_A$ .
- Case 3:  $J$ ,  $A$  and  $B$  are in the same  $VM$ . Then,  $I_J = I_A \cup I_B$ .
- Case 4:  $J$ ,  $A$  and  $B$  are in different  $VM$ s. Then,  $I_J = MS \setminus I_A \setminus I_B$ .

Case 1 means that the join operation  $J$  is executed at the sources in  $I_J = I_A \setminus I_B$ . The other cases follow the similar meanings.

$R_i(A)$  includes the matches of  $A$  at source  $S_i$  ( $S_i \in I_A$ ), similar to  $R_j(B)$  ( $S_j \in I_B$ ). For any  $i, j \in [1, n]$ ,  $u_{ij} = [R_i(A), R_j(B)]$  is called a *unit joining* of  $J$  in  $R(A) \cup R(B)$ . Then:

- (1)  $U$  consists of all unit joinings of  $J$  in  $R(A) \cup R(B)$ ; and
- (2)  $X$  consists of pairs  $(i, X)$  for all  $i \in [1, n]$  and  $X \subseteq U$ . We say that  $(i, X)$  covers element  $u_{jk} = [R_j(A), R_k(B)]$  in  $U$  if  $u_{jk} \in X$ . The weight of  $(i, X)$ , denoted by  $t(i, X)$ , is defined as the sum of the total travel cost of fetching  $R_j(A)$  and  $R_k(B)$  from sources  $S_j$  and  $S_k$  to source  $S_i$  and the total cost of computing  $R_j(A) \bowtie R_k(B)$  for all units  $[R_j(A), R_k(B)]$  in  $X$ .
- (3) The sources  $S_i, S_j$  and  $S_k$  obey the rules in the above four cases for  $S_i \in I_J, S_j \in I_A$  and  $S_k \in I_B$ .

It is not difficult to verify that the reduction is approximation preserving [51]. Based on the reduction, PhySch iteratively selects  $(i_*, X_*)$  such that the ratio of its weight  $t(i_*, X_*)$  to the number of new unit joinings covered by  $X_*$  is the minimum of all set covers.

After the greedy process, it returns a distributed plan that consists of exactly these atomic operations. It is an  $O(\log n)$ -approximation for computing minimum cost joining plans since it is an  $O(\log |U|)$ -approximation for the reduced MSC [51], where  $|U| = n^2$ .

## 6 LINKING NODES OF GRAPHS

To obtain an out-node of a source, we need to identify the same entity for different graphs. To obtain a global ontology  $GO$ , we merge different local ontologies  $LO$  by identifying the same concept of different  $LO$ s.

To this end, we develop *node linking* that can identify the same node (concept) for two different graphs (resp. ontologies). To obtain the global ontology  $GO$ , we apply node linking to every pair of local ontology graphs in the graph federation  $GF$ .

Node linking uses the following concepts: Given a directed graph  $G$  and two nodes  $a, b$  in  $V(G)$ ,  $a$  is a *child* of  $b$  if  $(a, b) \in E(G)$ ;  $a$  is a *grandson* of  $b$  if there are two consecutive edges from  $a$  to  $b$  in  $G$ ; and  $a$  is a *descendant* of  $b$  if there is a path from  $a$  to  $b$  in  $G$ .

Given two disjoint graphs  $G_1 = (V_1, E_1, L_1)$  and  $G_2 = (V_2, E_2, L_2)$ , node linking has the following intuitive idea. If a node  $u$  in  $G_1$  matches a node  $v$  in  $G_2$ , the children of  $u$  are “similar” to those of  $v$ ; the grandsons of  $u$  are “similar” to those of  $v$ ; and the descendants of  $u$  are “similar” to those of  $v$ . That is, node linking inductively considers the “similarity” of the descendants of  $u$  and the descendants of  $v$ .

To realize the idea, node linking first embeds nodes in  $G_1$  and  $G_2$  into the same feature space based on representation learning. Node linking then computes the distances of these node embeddings and returns the matching nodes. In the representation learning, node linking considers similar *descendant semantics* and *attribute semantics*.

We first illustrate the descendant semantics. For a node  $u \in G$ , to obtain the embedding about  $u$  and its descendant  $u_j$ , node linking aggregates the paths from  $u$  to  $u_j$  based on a *path-level attention*. There can be many paths from  $u$  to  $u_j$  in  $G$ , which reveals the rich semantic information between  $u$  and  $u_j$ . To learn a more comprehensive semantics for  $u$  and  $u_j$ , node linking needs to merge these paths.

Specifically, we design a graph-specific transformation matrix for  $G$ , denoted by  $M_g$ , to project the features of these paths in  $G$  into a unified feature space. That is, given a path  $p$  in  $G$ , its projected path feature  $h'_p$  can be obtained by:

$$h'_p = M_g \cdot h_p; \quad (2)$$

where  $M_g$  is the transformation matrix for  $G$  and  $h_p$  is the initial embedding vector of path  $p$  created by a natural language model (e.g., BERT [12]).

Denote by  $P_{i,j}$  the set of paths from  $u$  to  $u_j$  in  $G$ . Node linking computes the attention weight  $\alpha_{i,j}^p$  for each  $p \in P_{i,j}$ , to indicate how important the path  $p$  for expressing the semantics between  $u$  and  $u_j$ . Formally, the attention weight  $\alpha_{i,j}^p$  is calculated as follows:

$$\alpha_{i,j}^p = \text{ReLU}(h'_p), \quad (3)$$

where  $\text{ReLU}$  is the non-linear activation function.

After obtaining the attention weights of paths from  $u$  to  $u_j$ , node linking normalizes them to obtain the weight coefficient  $c_{i,j}^p$  using

the softmax function:

$$c_{i,j}^p = \text{softmax}_p(\alpha_{i,j}^p) = \frac{\alpha_{i,j}^p}{\sum_{q \in P_{i,j}} \exp(\alpha_{i,j}^q)}; \quad (4)$$

Then, node linking obtains the descendant semantics between  $u$  and  $u_j$  as follows:

$$hs_{i,j} = \sigma \left( \sum_{p \in P_{i,j}} c_{i,j}^p h'_p \right), \quad (5)$$

where  $\sigma$  is the activation function.

Consider a set of node pairs  $\{(u, u_{j1}), (u, u_{j2}), \dots, (u, u_{jl})\}$ , where  $u_j$  is a descendant of  $u$  in  $G$ . Feeding their path sets into the path-level attention, node linking obtains  $l$  descendant embeddings, denoted by  $\{hs_{i,j1}, hs_{i,j2}, \dots, hs_{i,jl}\}$ .

To balance the computation complexity and the representations of descendant semantics, node linking focuses on the descendants within  $k$  hops away from the node  $u$ , denoted by  $N_k(u)$ . To aggregate the embeddings between  $u$  and its descendants within  $N_k(u)$ , node linking takes such  $l$  embeddings  $\{hs_{i,j1}, hs_{i,j2}, \dots, hs_{i,jl}\}$  as inputs and returns the attention coefficients  $\{\beta_{i,j1}, \beta_{i,j2}, \dots, \beta_{i,jl}\}$ .

Based on these results, the descendant representation of node  $u$ , denoted by  $hs_u$ , is computed as follows:

$$hs_u = \sigma \left( \sum_{u_j \in N_h(u)} \beta_{i,j} hs_{i,j} \right); \quad (6)$$

where  $\sigma$  is the activation function.

In addition to the descendant semantics, we should also consider the semantic (embedding) of node  $u$ 's attributes, denoted by  $ha_u$ . Given nodes  $u$  and  $v$  in  $G_1$  and  $G_2$ , node linking inputs their attributes and returns the semantic similarity of  $ha_u$  and  $ha_v$  by utilizing a sentence embedding model [29].

In summary, we say  $u$  and  $v$  is a match if  $\text{sim}(ha_u, ha_v) \geq \mu$  and  $\text{sim}(hs_u, hs_v) \geq \nu$ , where  $\mu$  and  $\nu$  are thresholds for attribute and descendant similarities. After obtaining all the node representations for  $G_1$  and  $G_2$ , node linking applies a cluster-based matching to reduce unnecessary comparisons of pairwise embeddings. Specifically, node linking groups the node embeddings into a set of clusters using a cluster algorithm, e.g.,  $K$ -means. For each cluster, node linking then computes the related similarities of node embeddings to find the matching nodes.

## 7 EVALUATION

We evaluate FedGraph using a real test bed with 7 real-life datasets and compare it with 3 baselines. Our highlights are illustrated as follows:

- FedGraph is effective at reducing both parallel execution and network cost, outperforming its competitors by 37.3- and 61.8-fold on average, respectively.
- The performance of FedGraph is less sensitive with an increasing number of tasks than that of its competitors.
- FedGraph has better scalability than its competitors due to the effective techniques for reducing the communication overhead among the sources.
- NodeLink of FedGraph provides high accuracy graph and ontology integrations.

## 7.1 Setup

*Test bed.* FedGraph is deployed on a cluster with 13 machines connected with a high-speed kilomega network, where one machine is selected as the mediator, and the remaining machines are sources. Each machine runs CentOS Linux 7.6 with a 4 Intel Core i7-880 3.06 GHz CPU, 32 GB memory, and 1 TB HDD.

**Table 1: Real-life datasets**

Dataset	CS	Mater	Engin	Chem	Phy	CrossD	YAGO2
$ V(G) $	11.9M	4.6M	5.2M	12.2M	18.1M	27.2M	3.5M
$ E(G) $	107.2M	42.2M	36.1M	159.5M	79.5M	51.6M	7.35M
$ V(O) $	38	27	53	41	35	103	13
$ E(O) $	107	67	189	105	87	856	36
$ G_i.O $	15.5K	2.7K	7.1K	2.2K	1.8K	156.8K	78.8K

*Real-life dataset.* We use 7 real-life property graphs, as shown in Table 1. We first generate five graphs from the Open Academic Graph (OAG) [58] dataset (with 178 million nodes and 2.236 billion edges); these graphs are domain-specific subgraphs from OAG: computer science (CS), materials science (Mater), engineering (Engin), chemistry (Chem), and physics (Phy). We then supplement two widely used graph datasets, CrossDomain (CrossD) [1] and YAGO2 [25]. These property graphs have labels and properties such as papers, authors, venues, and institutes. We use the state-of-the-art scheme [59] to generate an ontology from each data graph. All the statistics of the 7 graphs are listed in Table 1. The global ontology (GO) has 296 nodes and 1357 edges. The GO is very small and is maintained in every source.

Every graph is maintained in a source (machine). Thus, five (i.e., 12-7=5) sources do not have preloaded graphs. To be more practical for real applications, each source must contain a dataset. We select the 3 largest data graphs and partition each into three subsubgraphs. Every subgraph is distributed to one source, and its ontology is also copied to the corresponding source. Finally, we apply NodeLink to merge the 12 local ontologies into one global ontology that is maintained in the mediator and every source.

*Query workload.* We first randomly derive star queries from the 7 property graphs. The star queries have labels and properties. We then extend the stars by adding nodes and edges to generate queries with more complex and larger structures, e.g., cliques. We use  $Q_i$  to denote the size of a query  $Q$ , where  $i$  is the number of nodes of  $Q$ . For each  $Q_i$ , we generate a set of 20 queries and report the average performance. In the experiment, we set the query sets as  $Q_2$ ,  $Q_4$ ,  $Q_6$ ,  $Q_8$  and  $Q_{10}$ , where  $Q_6$  is the default set.

*Implementation.* We develop a prototype system of FedGraph, which employs Neo4j as the database at each source. As we discuss in Section 3, there are three key phases of the FedGraph framework. The algorithms of distributed matching and joining are developed in Neo4j and deployed at each source. The algorithm of query decomposition is developed in the mediator.

To reflect **autonomy**, we partition the 12 sources into 4 groups, i.e., 3 sources in each group. The **heterogeneity** is reflected in two ways: (1) the memories of the sources belonging to 4 groups are constrained to 50%, 40%, 30% and 20% of their capacities; and (2)

the datasets deployed on various sources have different sizes and semantics.

*Baselines.* We compare FedGraph with the algorithms:

(1) Match is an OSM algorithm for a centralized environment [57]. It follows the filtering-verification strategy, which directly computes matches from the extracted small subgraph without searching the entire graph. To apply it to graph federation, we make the following adaptation. We extract a subgraph from each  $G_i$  using Match. The subgraph retains the out-nodes of  $G_i$ . We then send  $Q$  to every  $S_j$  to execute Match. Note that Match may span multiple sources through out-nodes since the matches of  $Q$  might span multiple  $G_i$ .

(2) DisRDF processes subgraph matching over a distributed RDF graph by using a “partial evaluation and assembly” framework [41]. DisRDF cannot support ontology-based matching. To adapt it to graph federation, we feed our matching algorithm (see Figure 7) into the “partial evaluation” phase of DisRDF. During the assembly, DisRDF relies on a straightforward joining strategy that cannot schedule any intermediate computations as FedGraph.

(3) BinJoin computes subgraph matching by solving a series of binary joins [31]. It first decomposes the query graph into a set of joined units (e.g., TwinTwig) whose matches can serve as the base relations of the join. BinJoin then joins the base relations based on a near optimal joining order. Similar to DisRDF, BinJoin focuses only on the optimal joining order but neglects the heterogeneity of graph federation.

*Metrics.* In the experiments, we evaluate the *total execution time* and *communication cost* of the algorithms. The communication cost is measured by the number of exchanged messages, including the messages exchanged between the mediator and the sources, as well as the messages exchanged between the sources.

## 7.2 Experimental Results

*Exp-1: accuracy of the node linking.* To evaluate NodeLink, we randomly initialize all the learnable parameters and optimize the model using Adam optimizer [28]. The initial features of attributes and paths are generated by the word2vec model [12] and the distributed bag of words (DBOW [34]), respectively. The negative sample number is 10 for each pre-matched node pair. The activation functions are  $\tanh(\cdot)$  and  $\text{ReLU}(\cdot)$  for the node representations. We use early stopping to terminate training with a patience of 5 epochs.

NodeLink is executed for every pair of graphs from the 7 datasets, and its quality is evaluated by the F-measure. The F-measure is defined with the precision and recall [7]. Here, the precision, recall and F-measure are (1) the ratio of true matches to matches returned, (2) the ratio of true matches to annotated match pairs in the dataset, and (3)  $2 \cdot (\text{precision} \cdot \text{recall}) / (\text{precision} + \text{recall})$ , respectively.

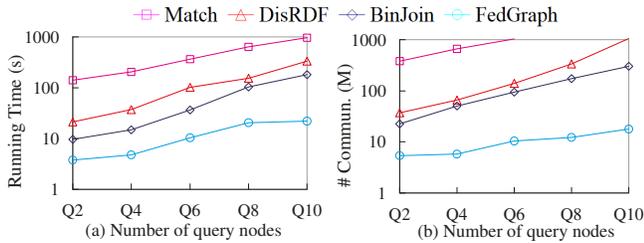
We perform our evaluation with three settings: (1) we vary  $k$  from 1 to 5 by fixing  $\mu = 0.8$  and  $\nu = 2$ ; (2) we vary  $\mu$  from 0.5 to 0.9 by fixing  $k = 20$  and  $\nu = 2$ ; and (3) we vary  $\nu$  from 1.0 to 3.0 by fixing  $k = 20$  and  $\mu = 0.8$ . Table 2 reports the evaluation results of precision (Acc), recall (Rec) and F-measure ( $f$ -m).

With setting (1), the F-measure first increases and then remains stable after reaching a  $k$  of approximately 4. With setting (2), the F-measure first grows steadily when  $\mu$  increases; it reaches the peak at  $\mu = 0.8$  and then drops sharply with larger  $\mu$ . The accuracy with

**Table 2: Accuracy of node linking (Exp-1).**

$k$	$\nu=2 \mu=0.8$			$\nu=2 k=20$			$k=20 \mu=0.8$		
	Acc, Rec, $f$ -m	$\mu$	Acc, Rec, $f$ -m	$\nu$	Acc, Rec, $f$ -m				
1	0.61, 0.09, 0.15	0.5	0.58, 0.11, 0.17	1	0.64, 0.08, 0.15				
2	0.66, 0.24, 0.35	0.6	0.6, 0.19, 0.29	1.5	0.63, 0.13, 0.21				
3	0.72, 0.56, 0.62	0.7	0.67, 0.51, 0.58	2	0.73, 0.95, 0.88				
4	0.82, 0.98, 0.88	0.8	0.75, 0.97, 0.88	2.5	0.52, 0.08, 0.11				
5	0.91, 0.87, 0.88	0.9	0.9, 0.6, 0.72	3	0.43, 0.05, 0.09				

setting (3) exhibits a similar trend to that with setting (2). From the evaluation, we observe that the F-measure is the best at  $\nu = 2.0$ . Therefore, we set the parameters as  $\mu = 0.8$ ,  $\nu = 2$  and  $k = 4$  for the following experiments.



**Figure 12: Impact of query sizes (Exp-2).**

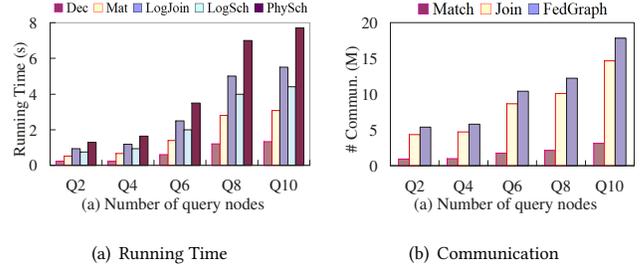
*Exp-2: impact of query sizes.* To evaluate the impact of query sizes, we vary the query size from Q2 to Q10 and then apply FedGraph as well as the comparison algorithms.

Figure 12 shows that both execution time and communication cost of Match, DisRDF and BinJoin grow exponentially, while that of FedGraph has less of an upward trend than the others. On average, FedGraph exhibits 107.5, 23.8, and 18.7 times fewer communications than Match, DisRDF and BinJoin, respectively, and is 39.6, 16.5 and 11.2 times faster. Notably, this advantage is more significant with increasing query size. This is because FedGraph optimizes all three phases of the framework instead of the exhaustive search of the compared algorithms and thereby has a lower chance of being stuck in a local optimum.

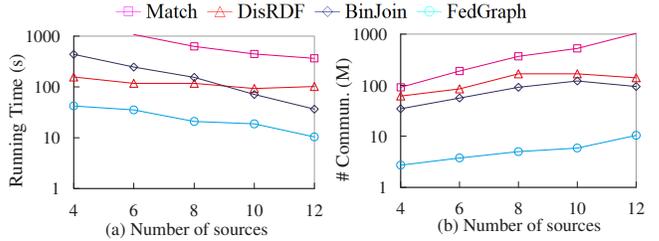
Figure 13 shows the performance of each phase of FedGraph with respect to query size. Specifically, Figure 13(a) reports the running time of Dec, Mat, LogJoin, LogSch and PhySch, and Figure 13(b) gives the communication cost of the distributed matching (Match) and the joining (Join).

We observe that PhySch dominates most of the running time, followed by LogJoin, LogSch, Mat, and Dec. This result is consistent with our intuition that PhySch tries every feasible joining plan to select the optimal one and that Dec executes only in the local mediator. For communication, Join contributes more than 80% of the traffic of FedGraph, whereas Match incurs less than 20% of that of FedGraph.

*Exp-3: impact of source numbers.* To evaluate the scalability, we vary the source number from 4 to 12. Figure 14 shows that FedGraph outperforms other methods in terms of scalability, yielding the minimum execution time and communication cost. In general, all execution times decrease with increasing source number, and

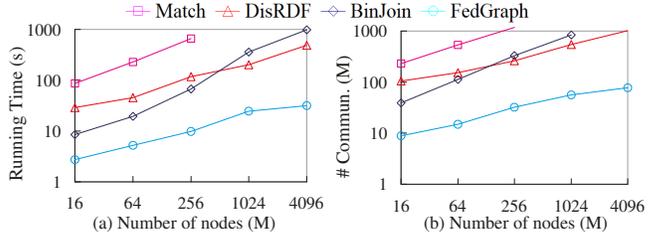


**Figure 13: Performance of each phase of FedGraph.**



**Figure 14: Impact of source numbers (Exp-3).**

adding more sources causes an increase in the communication cost. Because of the source heterogeneity, the decrease in execution time and the increase in communication cost are not completely proportional to the increase in the source number. Moreover, Figure 14(b) shows that FedGraph introduces less network overhead with a varying number of sources than other algorithms; i.e., the growth of the communication cost of FedGraph is much smaller than that of the others with increasing sources.



**Figure 15: Impact of graph sizes (Exp-4).**

*Exp-4: impact of graph sizes.* Finally, we examine the impact of the data graph size. Here, we use a new experimental setting as follows. We develop a graph generator to randomly produce synthetic graphs; this generator is controlled by three parameters: the number of nodes  $|V|$ , the number of edges  $|E|$ , and the size  $|L|$  of the node label set. Additionally, we generate ontology graphs for the set of synthetic graphs sharing the same set of labels  $L$ , controlled by the same set of parameters. Based on the setting, we generate 5 data graphs and 5 ontology graphs. Every data graph is randomly partitioned and distributed across the 12 sources. NodeLink is used

to identify the out-nodes and the global ontology. We also generate a set of 20 Q6 queries according to  $L$ . We report the average results.

Figure 15 presents the experimental results by varying the node numbers of the data graphs from 16M to 4,096M. FedGraph consistently performs the best of all the methods. This advantage amplifies as the graph size increases. For example, on average, the execution time of FedGraph is  $32.5 \times$  less than other algorithms, and it sends 85.3% fewer messages than the other algorithms. FedGraph is very efficient; e.g., it takes 30 seconds for the graph of 40 billion nodes. FedGraph sends fewer than 80M messages for the same graph. This experiment shows that FedGraph has good scalability for very large graphs.

## 8 RELATED WORK

We categorize the related work as follows.

*Graph pattern matching.* There have been a large number of algorithms developed for graph pattern matching thus far; these algorithms can generally be classified into backtracking [11, 19, 20, 50], encoding and indexing [5, 22, 46, 57], and decomposition [6, 49]. The first category of algorithm started with [50] and was followed by [11]. An approach of rewriting the query into a neighborhood equivalence class (NEC) tree in terms of neighborhood equivalence is proposed in [20]; dynamic programming is exploited to determine the matching order according to a directed acyclic graph (DAG) established from the pattern in [19]. Backtracking-based subgraph Enumeration (BENU) [54] divides a subgraph enumeration task into a group of local search tasks that can be executed in parallel, each of which follows a backtracking-based execution plan. In the second category, the labels of a node within a radius are encoded as signatures in [22]. The neighbors or degrees of each node are encoded as indices in [46]. Match is an ontology-based index proposed for subgraph matching [57]. Recently, to find the candidates for every edge of the query trees, an index using a compact embedding cluster was proposed in [5]. In the third category, the pattern is decomposed into a series of tree-like subgraphs in [49] and a dense subgraph along with a forest in [6]. The same approaches have been proposed for distributed subgraph matching [3, 4, 31–33].

*Federated RDF query.* A number of algorithms have been developed for federated RDF query processing, classified into metadata-based queries [2, 17, 21, 42, 44] and ask queries [45]. Due to the autonomy of RDF sources, the major differences among existing algorithms are query decomposition and source selection. In the first category, a vocabulary of interlinked dataset-based OWL language is used as the metadata in [17], and a variant of the R-tree where its leaf nodes store a set of source identifiers is used in [21, 42]. [2] generates the query plan dynamically, considering both data availability and runtime conditions. Recently, a set of capabilities that can map the properties to corresponding subjects and objects were defined and used in [44]. In the second category, ask queries for every triple pattern are sent to the RDF sources and the patterns are annotated with relevant sources according to the answers in [45]. [9] is the first to consider heterogeneous securities for federated databases and proposed related efficient query processing algorithms.

*Ontology-based query processing.* Ontological information has been studied and used for keyword queries [23, 30, 36, 37], pattern mining [8] and semantic queries [10, 35, 57]. In the first category, an

ontology-based multifaceted search paradigm that links keyword queries to a number of entities in multidistinct ontology views is explored in [37]. In the second category, an approach [8] of mining frequent patterns over graphs uses generalized labels in input taxonomies. In the third category, class hierarchy is exploited in [10] to evaluate queries specified by SPARQL and OWL on RDF graphs, in which distance metrics identify approximate results. Template graph searching is extended by interpolating ontologies to data graphs in [35, 36]. Moreover, [57] leverages the ontology graph to develop filtering strategies to identify semantically related matches.

FedGraph is designed for a brand new graph computation scenario, namely, graph federation. It differs from all the prior works in the following ways. (1) Rather than exact graph pattern matching using identical labels, this work identifies the matches semantically close to the query graph through ontology from distributed graph sources via a mediator in the graph federation. (2) This work proposes a machine learning-based solution for integrating graphs and ontologies that has wider applications than RDF graphs and OWL languages, where common uniform resource identifiers (URIs) often need manual identification. (3) This work aims to propose a general framework for graph federation that is more suitable to be formulated by a graph model. Compared to the RDF, the graph-based model has the advantage of identifying instances of a relationship of the same type [26]. In other words, the RDF-based model is not suitable for performing subgraph matching, which is one of the critical features in graph federation. (4) The graph federation is autonomous and heterogeneous, which is not considered in prior works, e.g., logical and physical scheduling algorithms. The experiments show that these algorithms can very efficiently process subgraph matching, e.g., 53 times faster than the SPARQL-based solution.

## 9 CONCLUSION

We made the first attempt to study query answering in graph federation by incorporating the features of autonomy and heterogeneity. We abstracted these features, formalized the subgraph matching problem in graph federation, studied its complexity, and developed approximate algorithms for generating distributed query plans. We proposed machine learning techniques to identify common entities (resp. concepts) between different graphs (resp. ontologies). Our experimental study showed that this framework is promising for reducing communication costs and minimizing execution time. In future work, we will study the challenges incurred by graph federation dynamic and private features. Additionally, we will extend the techniques to other types of graph queries in graph federation.

## ACKNOWLEDGMENTS

Ye Yuan is supported by the NSFC (Grant No. 61932004). Guoren Wang is supported by the NSFC (Grant No. 61732003, No. 61729201 and No. U2001211). Zhenyu Wen is supported by the NSFC (Grant No. 62072408 and No. 62073292) and Zhejiang Provincial NSFC (No. LY20F020030). Zhiwei Zhang is supported by the Open Research Projects of Zhejiang Lab (Grant No. 2020KE0AB04) and NSFC (Grant No. 62072035).

## REFERENCES

- [1] 2020. The Linked Open Data Cloud. <https://lod-cloud.net/>.
- [2] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. 2011. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *International Semantic Web Conference*. Springer, 18–34.
- [3] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. 2013. Enumerating subgraph instances using map-reduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 62–73.
- [4] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment* 11, 6 (2018), 691–704.
- [5] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.
- [6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.
- [7] David C. Blair. 1979. Information Retrieval, 2nd ed. C.J. Van Rijsbergen. London: Butterworths. *Journal of the American Society for Information Science* 30, 6 (1979), 374–375.
- [8] Ali Cakmak and Gultekin Ozsoyoglu. 2008. Taxonomy-superimposed graph mining. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 217–228.
- [9] Yang Cao, Wenfei Fan, Yanghao Wang, and Ke Yi. 2020. Querying shared data with security heterogeneity. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 575–585.
- [10] Olivier Corby, Rose Dieng-Kuntz, Fabien Gandon, and Catherine Faron-Zucker. 2006. Searching the semantic web: Approximate query processing based on ontologies. *IEEE Intelligent Systems* 21, 1 (2006), 20–27.
- [11] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.
- [12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [13] Kemele M Endris, Philipp D Rohde, Maria-Esther Vidal, and Sören Auer. 2019. Ontario: Federated query processing against a semantic data lake. In *International Conference on Database and Expert Systems Applications*. Springer, 379–395.
- [14] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. 2000. *Database system implementation*. Vol. 654. Prentice Hall Englewood Cliffs.
- [15] Michael R. Garey and David S. Johnson. 1979. *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman.
- [16] Mimos N Garofalakis and Yannis E Ioannidis. 1996. Multi-dimensional resource scheduling for parallel queries. *ACM SIGMOD Record* 25, 2 (1996), 365–376.
- [17] Olaf Görlitz and Steffen Staab. 2011. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proceedings of the Second International Conference on Consuming Linked Data—Volume 782*. 13–24.
- [18] Glen L Gray and Roger S Debreceeny. 2014. A taxonomy to guide research on the application of data mining to fraud detection in financial statement audits. *International Journal of Accounting Information Systems* 15, 4 (2014), 357–380.
- [19] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.
- [20] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 337–348.
- [21] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. 2010. Data summaries for on-demand queries over linked data. In *Proceedings of the 19th international conference on World wide web*. 411–420.
- [22] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 405–418.
- [23] Hanh Huu Hoang and A Min Tjoa. 2006. *The state of the art of ontology-based query systems: A comparison of existing approaches*. Citeseer.
- [24] Dorit S Hochba. 1997. Approximation algorithms for NP-hard problems. *ACM Sigact News* 28, 2 (1997), 40–52.
- [25] Johannes Hoffart, Fabian M Suchanek, Klaus Berberich, and Gerhard Weikum. 2013. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence* 194 (2013), 28–61.
- [26] <https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/>. [n.d.]. [n. d.].
- [27] Yannis E Ioannidis and Younkyung Cha Kang. 1991. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*. 168–177.
- [28] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).
- [29] Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Skip-Thought Vectors. In *Advances in Neural Information Processing Systems*. 3294–3302.
- [30] Rasmus Knappe, Henrik Bulskov, and Troels Andreasen. 2007. Perspectives on ontology-based querying. *International Journal of Intelligent Systems* 22, 7 (2007), 739–761.
- [31] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.
- [32] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment* 10, 3 (2016), 217–228.
- [33] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1099–1112.
- [34] Quoc V. Le and Tomáš Mikolov. [n.d.]. Distributed Representations of Sentences and Documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014 (JMLR Workshop and Conference Proceedings)*, Vol. 32. 1188–1196.
- [35] Eric Little, Kedar Sambhoos, and James Llinas. 2008. Enhancing graph matching techniques with ontologies. In *2008 11th International Conference on Information Fusion*. IEEE, 1–8.
- [36] Hanchao Ma, Morteza Alipourlangouri, Yinghui Wu, Fei Chiang, and Jiaxing Pi. 2019. Ontology-based entity matching in attributed graphs. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1195–1207.
- [37] Eetu Mäkelä, Eero Hyvönen, and Samppa Saarela. 2006. Ontogator<sup>+</sup>: a semantic view-based search engine service for web applications. In *International Semantic Web Conference*. Springer, 847–860.
- [38] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.
- [39] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. 2019. Data lake management: challenges and opportunities. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1986–1989.
- [40] Xiaofeng Ouyang, Liang Hong, and Lujia Zhang. 2018. Query Associations Over Big Financial Knowledge Graph. In *International Conference on Big Scientific Data Management*. Springer, 199–211.
- [41] Peng Peng, Lei Zou, M. Tamer Ozsu, Lei Chen, and Dongyan Zhao. 2016. Processing SPARQL queries over distributed RDF graphs. *The VLDB journal* 25, 2 (2016), 243–268.
- [42] Fabian Prasser, Alfons Kemper, and Klaus A Kuhn. 2012. Efficient distributed query processing for autonomous RDF databases. In *Proceedings of the 15th International Conference on Extending Database Technology*. 372–383.
- [43] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment* 11, 2 (2017), 176–188.
- [44] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. 2014. Hibiscus: Hypergraph-based source selection for sparql endpoint federation. In *European semantic web conference*. Springer, 176–191.
- [45] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. Fedx: Optimization techniques for federated query processing on linked data. In *International semantic web conference*. Springer, 601–616.
- [46] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.
- [47] Patrick Stünkel, Ole von Barga, Adrian Rutle, and Yngve Lamo. 2020. GraphQL Federation: A Model-Based Approach. (2020).
- [48] Shixuan Sun and Qiong Luo. 2019. Scaling up subgraph query processing with efficient subgraph matching. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 220–231.
- [49] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment* 5, 9 (2012), 788–799.
- [50] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.
- [51] Vijay V. Vazirani. 2001. *Approximation algorithms*. Springer.
- [52] Chihping Wang and Ming-Syan Chen. 1996. On the complexity of distributed query optimization. *IEEE Transactions on Knowledge and Data Engineering* 8, 4 (1996), 650–662.
- [53] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. 2006. Dual labeling: Answering graph reachability queries in constant time. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 75–75.

- [54] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed subgraph enumeration with backtracking-based framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 136–147.
- [55] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.
- [56] Wentao Wu, Hongsong Li, Haixun Wang, and Kenny Q Zhu. 2012. Probase: A probabilistic taxonomy for text understanding. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 481–492.
- [57] Yinghui Wu, Shengqi Yang, and Xifeng Yan. 2013. Ontology-based subgraph querying. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 697–708.
- [58] Fanjin Zhang, Xiao Liu, Jie Tang, Yuxiao Dong, Peiran Yao, Jie Zhang, Xiaotao Gu, Yan Wang, Bin Shao, Rui Li, et al. 2019. OAG: Toward linking large-scale heterogeneous entity graphs. In *Proceedings of the 25th ACM SIGKDD*. 2585–2595.
- [59] Amal Zouaq and Felix Martel. 2020. What is the schema of your knowledge graph? leveraging knowledge graph embeddings and clustering for expressive taxonomy learning. In *Proceedings of The International Workshop on Semantic Big Data*. 1–6.