Accelerating Partial Evaluation in Distributed SPARQL Query Evaluation

Peng Peng #1, Lei Zou *2, Runyu Guan #3

Hunan University, Changsha, China

* Peking University, Beijing, China

1 hnu16pp@hnu.edu.cn, 2 zoulei@pku.edu.cn, 3 guanrunyu@hnu.edu.cn

Abstract—**Partial evaluation has recently used for processing SPARQL queries over a large RDF graph in a distributed environment. However, the previous approach is inefficient to deal with complex queries. In this paper, we further improve the "partial evaluation and assembly" framework for answering SPARQL queries over a distributed RDF graph while providing performance guarantees. Our key idea is to explore the intrinsic structural characteristics of partial matches to filter out some irrelevant partial results while providing performance guarantees on the network traffic (data shipment) or the computational cost (response time). We also propose an efficient assembly algorithm to utilize the characteristics of partial matches to merge them and form the final results. To further improve the efficiency of finding partial matches, we propose an optimization that communicates variables' candidates among the sites to avoid redundant computations. Experiments over both real and synthetic RDF datasets confirm the superiority of our approach.**

I. Introduction

RDF is a semantic web data model that represents data as a collection of triples of the form ⟨subject, property, object⟩. An RDF dataset can also be represented as a graph, where subjects and objects are vertices and triples are edges with labels between vertices. On the other hand, SPARQL is a query language designed for retrieving and manipulating an RDF dataset, and its primary building block is the basic graph pattern (BGP). A BGP query can also be seen as a query graph, and answering a BGP Q is equivalent to finding subgraph matches of the query graph over RDF graph. In this paper, we focus on the evaluation of BGP queries. An example SPARQL query of four triple patterns (e.g., ?t label ?l) is listed in the following, which retrieves all people influencing Crispin Wright and their interests.

```
Select ?p2, ?l where {?t label ?l.
?p1 influencedBy ?p2.  ?p2 mainInterest ?t.
?p1 name ``Crispin Wright''@en.}
```

With the increasing size of RDF data published on the Web, the computational requirements of evaluating SPARQL queries over large RDF graphs have stressed the limits of single machine processing. To process SPARQL queries over a distributed database system, the RDF graph is often divided into multiple subgraphs named *fragments*. In many applications, the fragmented RDF graph are geographically or administratively distributed over the sites, and the RDF repository partitioning strategy is not controlled by the distributed RDF system itself. For example, Freebase [7] is an RDF dataset partitioned in domains; DBpedia [16] is multilingual and is divided into multiple subsets in different languages; LOD are provided by different publishers and many publishers do not allow other systems to repartition their datasets. Therefore, partitioning-tolerant SPARQL processing is desirable.

For partitioning-tolerant SPARQL processing on distributed RDF graphs, Peng et al.[19] discuss how to evaluate SPARQL queries in the "partial evaluation and assembly" framework. However, its efficiency has great potential to be improved and it does not provide performance guarantees. Generally speaking, the major bottleneck in the approach proposed in [19] is the large volume of partial evaluation results, which causes such a high cost for generating and assembling them.

In this paper, we improve the "partial evaluation and assembly" framework in [19]. To prune the irrelevant partial evaluation results and assemble them efficiently to form the final results, we propose three optimizations while providing some performance guarantees. The first is to compress all partial evaluation results into a compact data structure named *LEC feature*. Then, we can communicate the LEC features among sites to filter out some irrelevant partial evaluation results. We can prove that the proposed optimization technique is *partition bounded* in both *response time* and *data shipment* [5]. The details are discussed in Section IV. The second one is to join all local partial matches based on their LEC features to compute the final results. This is discussed in Section V. Last, to further avoid redundant computations within the sites, we propose an optimization that communicate variables' candidates among the sites to prune some candidates that cannot be contained in the final results.

In a nutshell, we make the following contributions in this paper.

- We explore the intrinsic structural characteristics of partial answers to compress the partial evaluation result of SPARQL queries into a compact data structure, *LEC feature*. We communicate and utilize the LEC features to filter out some irrlevant. We prove theoretically that the LEC feature can guarantee the performance of our framework in both *response time* and *data shipment*.
- We propose an efficient assembly algorithm to merge all LEC features found in different sites together and form

the final results.

- We present a framework based on the communication of the variables' candidates among different sites, which can further avoid redundant computations within the sites.
- We do experiments over both real and synthetic RDF datasets to confirm the superiority of our approach.

II. Background

A. Distributed RDF Graph and SPARQL Query

An RDF dataset can be represented as a graph where subjects and objects are vertices and triples are labeled edges. In the context of this paper, an RDF graph G is vertex-disjoint partitioned into a number of *fragments*, each of which resides at one site. The vertex-disjoint partitioning methods guarantee that there are no overlapping vertices between fragments. Here, to guarantee data integrity and consistency, we store some replicas of crossing edges. Formally, we define the *distributed RDF graph* as follows.

Definition 1: **(Distributed RDF Graph)** A distributed RDF graph $G = \{V, E, \Sigma\}$ consists of a set of fragments $\mathcal{F} = \{F_1, F_2, ..., F_k\}$ where each F_i is specified by $(V_i \cup V_i^e, E_i \cup E_i^c, \Sigma_i)$ $(i = 1, ..., k)$ such that

1) $\{V_1, ..., V_k\}$ is a partitioning of V, i.e., $V_i \cap V_j = \emptyset, 1 \leq i, j \leq k, i \neq j$ and $\bigcup_{i=1,...,k} V_i = V$;
2) $E_i \subseteq V_i \times V_i$, $i = 1, ..., k$;
3) E_i^c is a set of crossing edges between F_i and other fragments, i.e.,

$$E_i^c = (\bigcup_{1 \leq j \leq k \wedge j \neq i} \{\overrightarrow{uu'} | u \in F_i \wedge u' \in F_j \wedge \overrightarrow{uu'} \in E\}) \bigcup (\bigcup_{1 \leq j \leq k \wedge j \neq i} \{\overrightarrow{u'u} | u \in F_i \wedge u' \in F_j \wedge \overrightarrow{u'u} \in E\})$$

4) A vertex $u' \in V_i^e$ if and only if vertex u' resides in other fragment F_j and u' is an endpoint of a crossing edge between fragment F_i and F_j $(F_i \neq F_j)$, i.e.,

$$V_i^e = (\bigcup_{1 \leq j \leq k \wedge j \neq i} \{u' | \overrightarrow{uu'} \in E_i^c \wedge u \in F_i\}) \bigcup (\bigcup_{1 \leq j \leq k \wedge j \neq i} \{u' | \overrightarrow{u'u} \in E_i^c \wedge u \in F_i\})$$

5) Vertices in V_i^e are called *extended* vertices of F_i, vertices in V_i are called *internal* vertices of F_i, and vertices in V_i adjacent to vertices in V_i^e are called *boundary* vertices of F_i;
6) Σ_i is a set of edge labels in F_i.

Example 1: Fig. 1 shows a distirbuted RDF graph G consisting of three fragments F_1, F_2 and F_3. The numbers besides the vertices are vertex IDs that are introduced for ease of presentation. In Fig. 1, $\overrightarrow{001, 006}$ and $\overrightarrow{006, 005}$ are crossing edges between F_1 and F_2. As well, edges $\overrightarrow{001, 012}$ is a crossing edge between F_1 and F_3. Hence, $V_1^e = \{006, 012\}$ and $E_1^c = \{\overrightarrow{001, 006}, \overrightarrow{006, 005}, \overrightarrow{001, 012}\}$.□

Similarly, a SPARQL query can also be represented as a query graph Q. In this paper, we focus on basic graph pattern (BGP) queries as they are foundational to SPARQL, and focus on techniques for handling these.

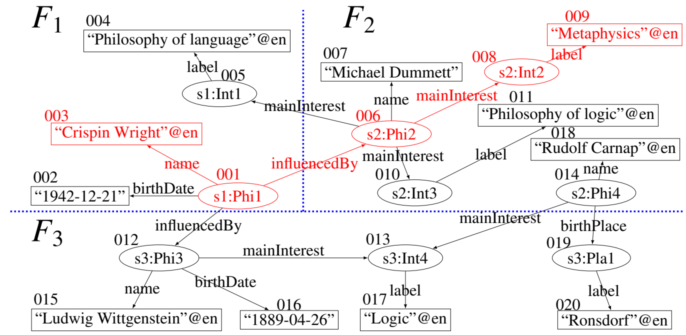


Fig. 1. Distributed RDF Graph

Definition 2: **(SPARQL BGP Query)** A *SPARQL BGP query* is denoted as $Q = \{V^Q, E^Q, \Sigma^Q\}$, where $V^Q \subseteq V \cup V_{Var}$ is a set of vertices, where V denotes all vertices in RDF graph G and V_{Var} is a set of variables; $E^Q \subseteq V^Q \times V^Q$ is a multiset of edges in Q; Each edge e in E^Q either has an edge label in Σ (i.e., property) or the edge label is a variable.

Example 2: Fig. 2 shows the query graph corresponding to the example query shown in Section I. There are four edges in the query graph and each edge maps to a triple pattern in the example query. Both vertices and edges in the query graph can be variable.□

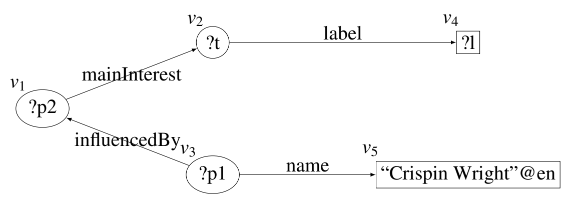


Fig. 2. SPARQL Query Graph

We assume that Q is a connected graph; otherwise, all connected components of Q are considered separately. Answering a SPARQL query is equivalent to finding all subgraphs of G homomorphic to Q. The subgraphs of G homomorphic to Q are called *matches* of Q over G.

Definition 3: **(SPARQL Match)** Consider an RDF graph G and a connected query graph Q that has n vertices $\{v_1, ..., v_n\}$. A subgraph M with m vertices $\{u_1, ..., u_m\}$ (in G) is said to be a *match* of Q if and only if there exists a *function* f from $\{v_1, ..., v_n\}$ to $\{u_1, ..., u_m\}$ $(n \geq m)$, where the following conditions hold: 1). if v_i is not a variable, $f(v_i)$ and v_i have the same URI or literal value $(1 \leq i \leq n)$; 2). if v_i is a variable, there is no constraint over $f(v_i)$ except that $f(v_i) \in \{u_1, ..., u_m\}$; 3). if there exists an edge $\overrightarrow{v_i v_j}$ in Q, there also exists an edge $\overrightarrow{f(v_i) f(v_j)}$ in G. Let $L(\overrightarrow{v_i v_j})$ denote a multi-set of labels between v_i and v_j in Q; and $L(\overrightarrow{f(v_i) f(v_j)})$ denote a multi-set of labels between $f(v_i)$ and $f(v_j)$ in G. There must exist an *injective function* from edge labels in $L(\overrightarrow{v_i v_j})$ to edge labels in $L(\overrightarrow{f(v_i) f(v_j)})$. Note that a variable edge label in $L(\overrightarrow{v_i v_j})$ can match any edge label in $L(\overrightarrow{f(v_i) f(v_j)})$.

Definition 4: (**Problem Statement**) Let G be a distributed RDF graph that consists of a set of fragments $\mathcal{F} = \{F_1, \ldots, F_k\}$ and let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be a set of computing nodes such that F_i is located at S_i. Given a SPARQL query graph Q, our goal is to find all matches of Q over G.

Note that for simplicity of exposition, we are assuming that each site hosts one fragment. Determining whether there exist matches in a site can be evaluated locally using a centralized RDF triple store, such as Virtuoso [3] and Jena[23]. In our prototype development and experiments, we modify gStore, a graph-based SPARQL query engine [26], to perform partial evaluation. The main issue of evaluating SPARQL queries over the distributed RDF graph is how to find the matches crossing multiple sites efficiently. That is a major focus of this paper.

Example 3: Given a SPARQL query graph Q in Fig. 2, there exists a crossing match mapping to the subgraph induced by vertices 003,001,006,008 and 009 (shown in the red vertices and edges in Fig. 1).□

B. Partial Evaluation-based SPARQL Query Evaluation

As we extend the distributed SPARQL query evaluation approach based on the "partial evaluation and assembly" framework in [19], we give its brief background here.

In the "partial evaluation and assembly" framework, each site S_i receives the full query graph Q (i.e., there is no query decomposition). In order to answer query Q, each site S_i computes the partial answers (called *local partial matches*) based on the known input F_i (recall that, for simplicity of exposition, we assume that each site hosts one fragment as indicated by its subscript). Intuitively, a local partial match PM_i is an overlapping part between a crossing match M and fragment F_i at the partial evaluation stage. Moreover, M may or may not exist depending on the yet unavailable input G' . Based only on the known input F_i, we cannot judge whether or not M exists.

Definition 5: (**Local Partial Match**) Given a SPARQL query graph Q and a connected subgraph PM with n vertices $\{u_1, ..., u_n\}$ ($n \leq |V^Q|$) in a fragment F_k, PM is a *local partial match* in fragment F_k if and only if there exists a function $f : V^Q \rightarrow \{u_1, ..., u_n\} \cup \{NULL\}$, where the following conditions hold: 1). If v_i is not a variable, $f(v_i)$ and v_i have the same URI or literal or $f(v_i) = NULL$. 2). If v_i is a variable, $f(v_i) \in \{u_1, ..., u_n\}$ or $f(v_i) = NULL$. 3). If there exists an edge $\overrightarrow{v_i v_j}$ in Q ($i \neq j$), then PM should meet one of the following five conditions: there also exists an edge $\overrightarrow{f(v_i)f(v_j)}$ in PM with property p and p is the same to the property of $\overrightarrow{v_i v_j}$, there also exists an edge $\overrightarrow{f(v_i)f(v_j)}$ in PM with property p and the property of $\overrightarrow{v_i v_j}$ is a variable, there does not exist an edge $\overrightarrow{f(v_i)f(v_j)}$ but $f(v_i)$ and $f(v_j)$ are both in V^e_k, $f(v_i) = NULL$, or $f(v_j) = NULL$. 4). PM contains at least one crossing edge, which guarantees that an empty match does not qualify. 5). If $f(v_i) \in V_k$ (i.e., $f(v_i)$ is an internal vertex of F_k) and $\exists \overrightarrow{v_i v_j} \in Q$ (or $\overrightarrow{v_j v_i} \in Q$), there must exist $f(v_j) \neq NULL$ and $\exists \overrightarrow{f(v_i)f(v_j)} \in PM$ (or $\exists \overrightarrow{f(v_j)f(v_i)} \in PM$). Furthermore, if $\overrightarrow{v_i v_j}$ (or $\overrightarrow{v_j v_i}$) has a property p, $\overrightarrow{f(v_i)f(v_j)}$ (or $\overrightarrow{f(v_j)f(v_i)}$) has the same property p. 6). If $f(v_i)$ and $f(v_j)$ are both internal vertices in PM, then there exist a *weakly connected path* π between v_i and v_j in Q and each vertex in π maps to an internal vertex of F_k in PM.

Vector $[f(v_1), ..., f(v_n)]$ is a serialization of a local partial match. $f^{-1}(PM)$ is the subgraph (of Q) induced by a set of vertices, where for any vertex $v \in f^{-1}(PM)$, $f(v)$ is not NULL.

Example 4: Given a query Q in Fig. 2 and a distributed RDF graph G in Fig. 1, Fig. 3 shows all local partial matches and their serialization vectors in each fragment. A local partial match in fragment F_i is denoted as PM_i^j, where the superscripts distinguish local partial matches in the same fragment. Furthermore, we underline all extended vertices in serialization vectors.□

Although there may exist many local partial matches for a SPARQL query, these local partial matches bear structural similarities (see Section IV-A). Based on the structural similarities, all local partial matches can be represented as vectors of Boolean formulas associated with crossing edges (see Section IV-B) and we can utilize these formulas to filter out some irrelevant local partial matches (see Section IV-C). Last, the remaining local partial matches are assembled to get the final answer (see Section V). Note that, in this paper, we focus on how to represent the local partial matches in a compact way and prune some irrelevant local partial matches. We directly use the algorithm in [19] to find local partial matches.

III. Overview

We also adopt the *partial evaluation and assembly* [14] framework to answer SPARQL queries over a distributed RDF graph G. Each site S_i treats fragment F_i as the known input s and other fragments as yet unavailable input. The special framework based on the above two optimization techniques for distributed SPARQL query processing as Fig. 4.

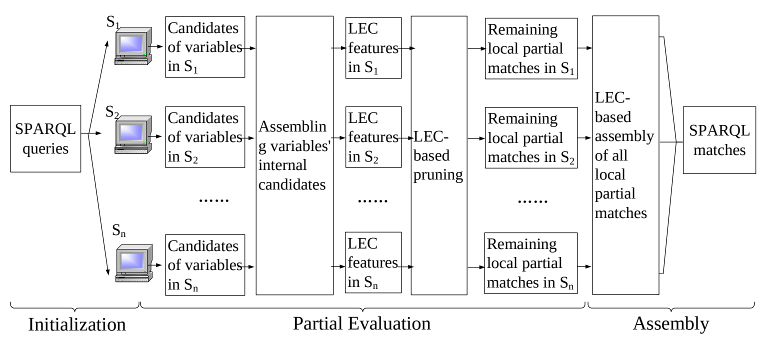


Fig. 4. Overview of Our Method

In our execution model, each site S_i receives the full query graph Q. In the partial evaluation stage, at each site S_i, we first use algorithms proposed in [19] to find all local partial matches of Q in F_i. Then, we explore the intrinsic structural similarities of local partial matches to divide these local partial matches into some equivalence classes. For each equivalence class, we propose a compact data structure, named *LEC feature* (Definition 8), to compress it. The LEC features maintain

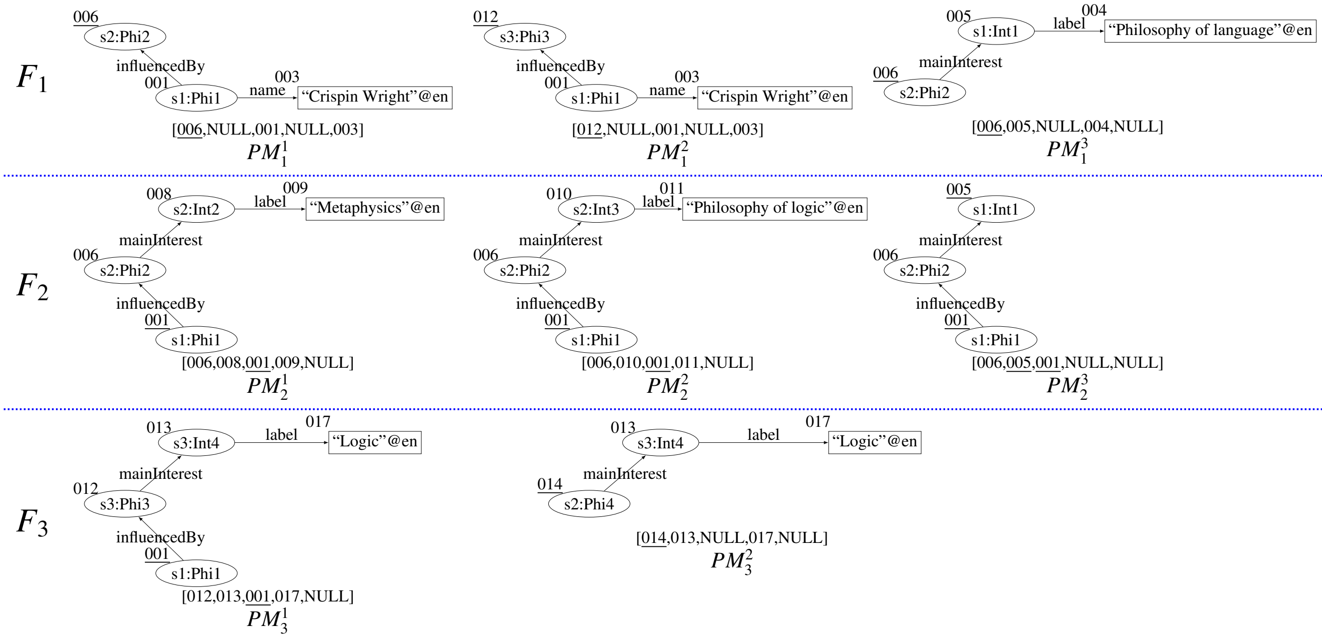


Fig. 3. Example Local Partial Matches

enough structural information of local partial matches, and only by joining LEC features can we determine which local partial matches can contribute to the complete matches (as discussed in Section IV). In addition, we can also prove that the communication cost of all LEC features only depends on the size of the query and the fragmentation of the graph (as discussed in Section IV-D).

To speed up the process of finding out all local partial matches, the proposed approach is to communicate all variables' internal candidates in different sites, as discussed in Section VI. Each site sends the sets of internal candidates to the coordinator site before the computing the local partial matches, the coordinator site assembles all sets of internal candidates from different sites and gains the candidates sets of all variables. Last, the coordinator site distributes the candidates sets and each site use them to find out the local partial matches.

In the assembly stage, all remaining local partial matches are assembled in a coordinator site and determine whether crossing matches exist. The naive assembly method is to try to join any two local partial matches. However, the join space of the naive method is too large. Hence, we propose a LEC-based method assembly algorithm to reduce the join space. First, we divide all local partial matches into some partitions based on their corresponding LECs such that two local partial matches in the same partition cannot join together. Then, we propose an algorithm based on the structural relevances among all partitions to determine the join orders and join all partitions of LEC features for final matches.

IV. LEC Feature-based Optimization

A. Local Partial Match Equivalence Class

As discussed in [19], only two local partial matches with common crossing edges from different fragments may join together via their common crossing edges. Hence, if two local partial matches generated from the same fragment contains the same crossing edges and these crossing edges map to the same query edges, then they can join with the same other local partial matches, which means that they should have the similar structures. For example, let us consider two local partial matches, PM_2^1 and PM_2^2 in Fig. 3. They contain the common crossing edge $\overrightarrow{001,006}$, and $\overrightarrow{001,006}$ maps to the query edge $\overrightarrow{v_3v_1}$ in both PM_2^1 and PM_2^2. Thus, PM_2^1 and PM_2^2 are homomorphic to the same subgraph of the query graph. Any other local partial match (like PM_1^1) that can join with PM_2^1 can also join with PM_2^2.

We can formalize the above observation as the following theorem.

Theorem 1: Given two local partial matches PM_i and PM_j from fragment F_k with functions f_i and f_j, we can find out that $f_i^{-1}(PM_i) = f_j^{-1}(PM_j)$, if they meet the following conditions:

1) $\forall \overrightarrow{u_iu_j} \in PM_i(or\ PM_j)$, if $\overrightarrow{u_iu_j} \in E_k^c$, $\overrightarrow{u_iu_j} \in PM_j(or\ PM_i)$;
2) $\forall \overrightarrow{u_iu_j} \in PM_i(or\ PM_j)$, if $\overrightarrow{u_iu_j} \in E_k^c$, $f_i^{-1}(u_i) = f_j^{-1}(u_i)$ and $f_i^{-1}(u_j) = f_j^{-1}(u_j)$.

Proof: First, we prove that $\forall v \in f_i^{-1}(PM_i)$, $v \in f_j^{-1}(PM_j)$. For any vertex $v \in f_i^{-1}(PM_i)$, there are two cases: 1), PM_i contains an edge $e \in E_k^c$ and $f_i(v)$ is an endpoint of e; 2), all edges adjacent to $f_i(v)$ in PM_i are not crossing edges.

If PM_i contains an edge $e \in E_k^c$ and $f_i(v)$ is an endpoint of e, since $e \in E_k^c$, $e \in PM_j$. Hence, $f_i(v) \in PM_j$. Furthermore, because of condition 2, $v = f_i^{-1}(f_i(v)) = f_j^{-1}(f_i(v))$. Thus, $v \in f_j^{-1}(PM_j)$.

Then, let us consider the case that all edges adjacent to $f_i(v)$ in PM_i are not crossing edges. Because $f_i(v)$ does not belong to any crossing edges in PM_i, $f_i(v)$ is an internal vertex of F_k. According to condition 6 of Definition 5, there exists a weakly connected path between v and any other vertices mapping to internal vertices in PM_i. Therefore, given a crossing edge $\overrightarrow{f_i(v_1)f_i(v_2)} \in PM_i$ where $f_i(v_1)$ is an internal vertex, there exists a weakly connected path $\pi = \{v_1, v_2, ..., v\}$ in $f_i^{-1}(PM_i)$ and all vertices in π map to internal vertices of F_k.

Let us consider vertices in π from v_1 to v one by one. Since $f_i(v_1)$ is an endpoint of a crossing edge, $v_1 \in f_j^{-1}(PM_j)$. As well, because PM_i and PM_j are from the same fragment, $f_j(v_1)$ in PM_j is still an internal vertex. According to condition 5 of Definition 5, all neighbors of v_1 have been matched in PM_j, so v_2 has been matched in PM_j. Furthermore, $f_j(v_2)$ must be an internal vertex. Otherwise, $\overrightarrow{f_j(v_1)f_j(v_2)}$ is a crossing edge, so $v_2 = f_j^{-1}(f_j(v_2)) = f_i^{-1}(f_j(v_2))$. In other words, $f_j(v_2)$ is an extended vertex of F_k and also maps to v_2 in $f_i^{-1}(PM_i)$. This is in conflict with the fact that all vertices in π map to internal vertices of F_k. By that analogy, we can prove that all other vertices in π have been matched in PM_j. Hence, $v \in f_j^{-1}(PM_j)$ and $f_j(v)$ is an internal vertex.

Similarly, we can prove that $\forall v \in f_j^{-1}(PM_j)$, $v \in f_i^{-1}(PM_i)$. Therefore, the vertex set of $f_i^{-1}(PM_i)$ is equal to the vertex set of $f_j^{-1}(PM_j)$. Moreover, for each vertex v in $f_i^{-1}(PM_i)$ and $f_j^{-1}(PM_j)$, both of $f_i(v)$ and $f_j(v)$ are internal vertices or extended vertices.

On the other hand, for each edge $\overrightarrow{v_1v_2} \in f_i^{-1}(PM_i)$, due to the condition 3 of Definition 5, at least one vertex of $f_i(v_1)$ and $f_i(v_2)$ is an internal vertex. Supposing that $f_i(v_1)$ is an internal vertex, $f_j(v_1)$ should also be an original vertex, so $\overrightarrow{v_1v_2} \in f_j^{-1}(PM_j)$. In the same way, we can prove that $\forall \overrightarrow{v_1v_2} \in f_j^{-1}(PM_j)$, $\overrightarrow{v_1v_2} \in f_i^{-1}(PM_i)$. Hence, the edge set of $f_i^{-1}(PM_i)$ is equal to the edge set of $f_j^{-1}(PM_j)$.

In conclusion, $f_i^{-1}(PM_i) = f_j^{-1}(PM_j)$. ■

Based on the above theorem, we can avoid exhaustive enumerations among irrelevant local partial matches with the same crossing edges and their matches which do not contribute to the final matches and result in significant data communication. Our strategy explores the intrinsic structural characteristics of the local partial matches to only generate combinations. If a generated combination cannot contribute to a valid match, we can filter out the local partial matches corresponding to the combination.

To define the combination of multiple local partial matches, we first define the concept of *local partial match equivalence relation* as follows.

Definition 6: **(Local Partial Match Equivalence Relation)** Let $\sim$ be an equivalence relation over all local partial matches in Ω such that, $PM_i \sim PM_j$ if PM_i(with function f_i) and PM_j(with function f_j) satisfy the following three conditions:

1) PM_i and PM_j are from the same fragment F_k.
2) $\forall \overrightarrow{u_iu_j} \in PM_i(or\ PM_j)$, if $\overrightarrow{u_iu_j} \in E_k^c$, $\overrightarrow{u_iu_j} \in PM_j(or\ PM_i)$;
3) $\forall \overrightarrow{u_iu_j} \in PM_i(or\ PM_j)$, if $\overrightarrow{u_iu_j} \in E_k^c$, $f_i^{-1}(u_i) = f_j^{-1}(u_i)$ and $f_i^{-1}(u_j) = f_j^{-1}(u_j)$.

Based on the above equivalence relation, there is a natural grouping of local partial matches that are related to one another. All local partial matches equivalent to a local partial match PM_i in Ω can be combined together to form the *L*ocal partial match *E*quivalence *C*lass (LEC) of PM_i as follows.

Definition 7: **(Local Partial Match Equivalence Class)** The *local partial match equivalence class (LEC)* of a local partial match PM_i is denoted $[PM_i]$ and defined as the set

$$[PM_i] = \{PM_j \in \Omega \mid PM_j \sim PM_i\}$$

Then, we can prove in the following theorem that if two local partial matches can join together, then all other local partial matches in the corresponding LECs of these two local partial matches can also join together. In other word, we only need to select one local partial match of a LEC as a representative to check whether all local partial matches in the LEC can join with other local partial matches. It prunes out many permutations of joining local partial matches of two LECs.

Theorem 2: Given two LEC $[PM_i]$ and $[PM_j]$, if local partial match PM_i can join with local partial match PM_j, then any local partial matches in $[PM_i]$ can join with any local partial matches in $[PM_j]$.

Proof: As discussed in [19], if PM_i and PM_j can join together, then they are generated from different fragments, they share at least one common crossing edge that corresponds to the same query edge and the same query vertex cannot be matched by different vertices in them.

Since PM_i and PM_j are from different fragments, according to Definition 6, any local partial match in $[PM_i]$ is generated from different fragments from any local partial match in $[PM_j]$. Furthermore, all local partial matches in $[PM_i]$ (or $[PM_j]$) contain the same crossing edges that map to the same query edges, so any local partial match in $[PM_i]$ (or $[PM_j]$) shares at least one common crossing edge with any local partial match in $[PM_j]$ (or $[PM_i]$).

In addition, since our fragmentation is vertex-disjoint, the query vertices that the internal vertices in PM_i map to should be different from the query vertices that the internal vertices in PM_j map to. Hence, the internal vertices in any local partial match of $[PM_i]$ (or $[PM_j]$) cannot conflict with the internal vertices in any local partial match of $[PM_j]$ (or $[PM_i]$) map to. On the other hand, since the crossing edges in PM_i does not conflict with the crossing edges in PM_j and Definition 6 defines that the local partial matches in the same LEC share the same crossing edges and their mappings, the extended vertices in any local partial match of $[PM_i]$ (or $[PM_j]$) cannot conflict the vertices in any local partial match of $[PM_j]$ (or $[PM_i]$) map to.

In summary, any two local partial matches in $[PM_j]$ and $[PM_i]$ meet all conditions that two joinable local partial matches should meet. Hence, the theorem is proven. ■

Example 5: Given all local partial matches in Fig 3, there are seven LECs as follows.

$$F_1 : [PM_1^1] = \{PM_1^1\};\ [PM_1^2] = \{PM_1^2\};\ [PM_1^3] = \{PM_1^1\};$$

$$F_2 : [PM_2^1] = [PM_2^2] = \{PM_2^1, PM_2^2\}; \quad [PM_2^3] = \{PM_2^3\};$$

$$F_3 : [PM_3^1] = \{PM_3^1\};\ [PM_3^2] = \{PM_3^2\};$$

Since PM_1^1 can join with PM_2^1 and PM_2^1 and PM_2^2 are in the same LEC, PM_1^1 can also join with PM_2^2.□

B. LEC Feature

Theorems 1 and 2 show that many local partial matches have the same structures and can be combined together as a LEC to join with local partial matches of other LECs through their common crossing edges. The observations imply that we can only use the same structure of local partial matches in a LEC and the common crossing edges of the LEC to determine whether the local partial matches of the LEC can join with the local partial matches of other LECs.

Hence, given a LEC $[PM]$, we maintain it into a compact data structure called *LEC feature* that only contains the same structure of local partial matches in $[PM]$ and the common crossing edges of $[PM]$ as follows.

Definition 8: **(LEC Feature)** Given a local partial match PM with function f and its LEC $[PM]$, its *LEC feature* $LF([PM]) = \{F, CE, LECSign\}$ consists of three components:

1) The fragment identifier, F, that PM is from;
2) A function g, which maps crossing edge $\overrightarrow{u_iu_j}$ in PM to its corresponding mapping $\overrightarrow{f^{-1}(u_i)f^{-1}(u_j)}$ in E^Q;
3) A bitstring of the length $|V^Q|$, $LECSign$, where we set i-th bit to be '1' if $f(v_i)$ maps to an internal vertex of F.

Fig. 5 shows a LEC feature $LF([PM_1^1])$ for the LEC $[PM_1^1]$ that is shown in Example 5. In $LF([PM_1^1])$, F_1 is the fragment identifer of the fragment that PM_1^1 is generated from; $\{\overrightarrow{001, 006} \rightarrow \overrightarrow{v_3v_1}\}$ is the set of crossing edges in PM_1^1 and their corresponding query edges; since the internal vertices in PM_1^1 match the query vertices v_3 and v_5 that correspond to the third and fifth bits of $LECSign$, the $LECSign$ in $LF([PM_1^1])$ is [00101].

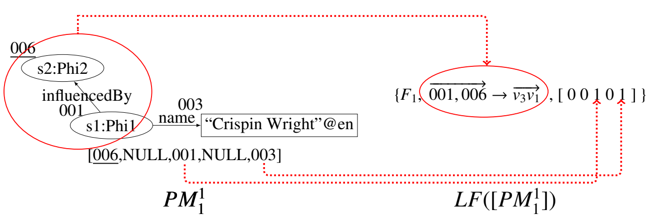


Fig. 5. LEC Feature $LF([PM_1^1])$ (PM_1^1 is the only element in $[PM_1^1]$)

Example 6: Given the LECs in Example 5, their LEC features are as follows:

$$LF([PM_1^1]) = \{F_1, \{\overrightarrow{001, 006} \rightarrow \overrightarrow{v_3v_1}\}, [00101]\}$$

$$LF([PM_1^2]) = \{F_1, \{\overrightarrow{001, 012} \rightarrow \overrightarrow{v_3v_1}\}, [00101]\}$$

$$LF([PM_1^3]) = \{F_1, \{\overrightarrow{006, 005} \rightarrow \overrightarrow{v_1v_2}\}, [01010]\}$$

$$LF([PM_2^1]) = LF([PM_2^2]) = \{F_2, \{\overrightarrow{001, 006} \rightarrow \overrightarrow{v_3v_1}\}, [11010]\}$$

$$LF([PM_2^3]) = \{F_2, \{\overrightarrow{006, 005} \rightarrow \overrightarrow{v_1v_2}, \overrightarrow{001, 006} \rightarrow \overrightarrow{v_3v_1}\}, [10000]\}$$

$$LF([PM_3^1]) = \{F_3, \{\overrightarrow{001, 012} \rightarrow \overrightarrow{v_3v_1}\}, [11010]\}$$

$$LF([PM_3^2]) = \{F_3, \{\overrightarrow{014, 013} \rightarrow \overrightarrow{v_1v_2}\}, [01010]\}$$

□

Given a SPARQL query Q and a fragment F_i, we can find all LEC features (according to Definition 5) in F_i and utilize them together to filter some irrelevant local partial matches. In this paper, we mainly focus on how to compress all local partial matches into LEC features. A high-level description of computing LEC features is outlined in Algorithm 1.

Algorithm 1: Computing LEC Features

Input: The set of all local partial matches in fragment F_i, denoted as $\Omega(F_i)$.
Output: The set of all LEC features in F_i, denoted as $\Omega(F_i)$, denoted as $\Psi(F_i)$.

for *each local partial match PM in* $\Omega(F_i)$ **do**
  Initialize a LEC feature LF;
  $LF.F \leftarrow F_i$;
  for *each mapping* $(\overrightarrow{u_iu_j}, \overrightarrow{v_iv_j})$ *in PM* **do**
    if u_i *is an extended vertex of fragment* F_i **then**
      $LF.LECSign[i] \leftarrow$ '0';
      $LF.g \leftarrow LF.g \cup (\overrightarrow{u_iu_j}, \overrightarrow{v_iv_j})$;
    else
      $LF.LECSign[i] \leftarrow$ '1';
    if u_j *is an extended vertex of fragment* F_i **then**
      $LF.LECSign[j] \leftarrow$ '0';
      $LF.g \leftarrow LF.g \cup (\overrightarrow{u_iu_j}, \overrightarrow{v_iv_j})$;
    else
      $LF.LECSign[j] \leftarrow$ '1';
  if $\Psi(F_i)$ *does not contain LF* **then**
    $\Psi(F_i) \leftarrow \Psi(F_i) \cup LF$;
Return $\Omega(F_i)$;

The above process consists of determining what the LEC feature of a local partial match PM is. We first initialize a LEC feature LF with the fragment identifer F_i. Then, we scan all mappings in PM. For each mapping $(\overrightarrow{u_iu_j}, \overrightarrow{v_iv_j})$, if u_i (or u_j) is an extended vertex, we set $LF.LECSign[i]$ (or $LF.LECSign[j]$) as '0', otherwise we set $LF.LECSign[i]$ (or $LF.LECSign[j]$) as '1'. Furthermore, if one of u_i and u_j is an extended vertex, we add $(\overrightarrow{u_iu_j}, \overrightarrow{v_iv_j})$ into $LF.g$. Last, we insert LF into the set of all LEC features in F_i. This above step iterates over each local partial match. Constructing all LEC features only requires a linear scan on the local partial matches. Therefore, it can be done on-the-fly as the local partial matches is streamed out from the evaluation.

C. LEC Feature-based Pruning Algorithm

In this section, based on the definition of LEC feature and its properties, we propose an optimization technique that prune some irrelevant local partial matches.

First, we define the conditions under which two local partial matches can join together as Definition 9 and prove the correctness of the join conditions as Theorem 3.

Definition 9: **(Joinable)** Given two local partial matches PM_i and PM_j, they are joinable if their LEC features $LF([PM_i])$ and $LF([PM_j])$ meet the following conditions:

1) $LF([PM_i]).F \neq LF([PM_j]).F$;
2) There exist at least one edge $\overrightarrow{u_iu_j}$, such that $LF([PM_i]).g(\overrightarrow{u_iu_j}) = LF([PM_j]).g(\overrightarrow{u_iu_j})$;
3) There exist no two edges $\overrightarrow{u_iu_j}$ and $\overrightarrow{u_i'u_j'}$ in the domains of $LF([PM_i]).g$ and $LF([PM_j]).g$, respectively, such that $LF([PM_i]).g(\overrightarrow{u_iu_j}) = LF([PM_j]).g(\overrightarrow{u_i'u_j'})$.
4) All bits in $LF([PM_i]).LECSign \wedge LF([PM_j]).LECSign$ are '0'.

Theorem 3: Given two LEC $[PM_i]$ and $[PM_j]$, if the LEC features of $[PM_i]$ and $[PM_j]$ are joinable, then any local partial match in $[PM_i]$ can join with any local partial match in $[PM_j]$.

Proof: Due to Condition 1 of Definition 9, any local partial match in $[PM_i]$ is generated from different fragments that any local partial match in $[PM_j]$ generated from. Condition 2 of Definition 9 means that any local partial matches in $[PM_i]$ shares at least one common crossing edge mapping to the same query edge with any local partial matches in $[PM_j]$. Condition 3 of Definition 9 implies that the same query vertex cannot be matched by different vertices in crossing edges of local partial matches in $[PM_i]$ and $[PM_j]$. Condition 4 of Definition 9 means that the same query vertex cannot be matched by different internal vertices edges of local partial matches in $[PM_i]$ and $[PM_j]$.

In summary, all conditions of Definition 9 imply all local partial matches in $[PM_i]$ and $[PM_j]$ meet all joining conditions discussed in [19]. Hence, any local partial match in $[PM_i]$ can join with any local partial match in $[PM_j]$. ■

Further, we prove in the following theorem that only using all LEC features can determine whether the local partial matches of a LEC can contribute to the complete matches.

Theorem 4: Given m ($m \leq |V^Q|$) local partial matches $PM_1, PM_2, ..., PM_m$, they can join together to form a match of Q if their corresponding LEC features meet the following conditions:

1) For any PM_i, there exists a local partial match PM_j ($j \neq i$) that $[PM_i]$ and $[PM_j]$ are joinable;
2) $\forall 1 \leq i \neq j \leq m$, all bits in $LF([PM_i]).LECSign \wedge LF([PM_j]).\ LECSign$ are '0';
3) All bits in $LF([PM_1]).LECSign \vee LF([PM_2]).LECSign \vee ... \vee LF([PM_m]).LECSign$ are '1'.

Proof: Here, we prove that if the three conditions in Theorem 4, then $PM_1 \bowtie PM_2 \bowtie ... \bowtie PM_m$ is a match of Q.

Conditions 1 and 2 in Theorem 4 guarantees that the m local partial matches can join together. Conditions 3 in Theorem 4 means that each vertex u in $PM_1 \bowtie PM_2 \bowtie ... \bowtie PM_m$ is an internal vertex of one local partial match PM_i ($i \leq m$). Since u is an internal vertex in PM_i, all u's adjacent edges have been matched. Then, we can know all edges in $PM_1 \bowtie PM_2 \bowtie ... \bowtie PM_m$ have been matched. Hence, $PM_1 \bowtie PM_2 \bowtie ... \bowtie PM_m$ is a match of Q. ■

Theorem 4 implies that we only need assemble all LEC features to determine which local partial matches can contribute to the complete match. If there exists a SPARQL match, there should be some LEC features that can be merged together and all bits in the union of these LEC features' $LECSign$ should be '1'. Hence, when the all bits in $LECSign$ of the join result of some LEC Features are '1', we can determine that there exists a SPARQL match by joining their corresponding local partial matches.

Therefore, before we assemble all local partial matches to form the complete matches, we assemble all LEC Features and merge them together. If a LEC feature cannot contribute to a union result of some LEC features' $LECSign$ where all bits are '1', then all local partial match corresponding to the LEC feature can be pruned.

The straightforward approach of merge all LEC features is to check each pair of LEC features whether they are joinable. However, the join space of the straightforward approach is very large, so we proposes an partitioning-based optimized technique to reduce the join space. The intuition of our partitioning-based technique is that we divide all LEC features into multiple groups such that two LEC features in the same group cannot be joinable. Then, we only consider joining LEC features from different groups.

Theorem 5: Given two LEC features LF_i and LF_j, if $LF_i.LECSign$ is equal to $LF_j.LECSign$, LF_i and LF_j are not joinable.

Proof: Since $LF_i.LECSign$ is equal to $LF_j.LECSign$, $LF_i.\ LECSign \wedge LF_j.LECSign = LF_i.LECSign = LF_i.\ LECSign$. According to Condition 4 of Definition 5, there are at least one internal vertices in a local partial match, so there are at least one '1' in $LF_i.LECSign$ and $LF_i.LECSign$. Therefore, there are at least one '1' in $LF_i.LECSign \wedge LF_j.LECSign$, which is in conflict with Condition 4 of Definition 9. ■

Definition 10: **(LEC Feature Group)**. Let Ψ denote all LEC features. $\mathcal{P} = \{P_1, ..., P_n\}$ is a set of *LEC feature groups* for Ψ if and only if each group P_i ($i = 1, ..., n$) consists of a set of LEC Features, all of which have the same $LECSign$.

Example 7: Given all LEC features in Example 6, their corresponding LEC feature groups $\{P_1, P_2, P_3, P_4\}$ are as follows.

$$P_1 = \{LF([PM_1^1]), LF([PM_1^2]), LF([PM_3^2])\}, P_2 = \{LF([PM_1^3])\}$$

$$P_3 = \{LF([PM_2^1]), LF([PM_3^1])\}, P_4 = \{LF([PM_2^3])\}$$

□

Given a set $\mathcal{P}$ of LEC feature groups, we build a *join graph* (denoted as $JG = \{V^{JG}, E^{JG}\}$) as follows. In a join graph, one vertex indicates a LEC feature group. We introduce an edge between two vertices in the join graph if and only if their corresponding LEC feature groups are joinable. Fig. 6 shows the join graph of $\mathcal{P}$.

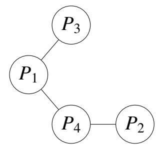


Fig. 6. Join Graph

We propose an algorithm (Algorithm 2) based on the DFS traversal over the join graph to filter out the irrelevant LEC features.

Algorithm 2: LEC Feature-based Pruning Algorithm

Input: A set $\mathcal{P} = \{P_1, ..., P_n\}$ of LEC feature groups and the join graph JG

Output: The set RS of LEC features that can contribute to complete matches

$RS \leftarrow \emptyset$;
while $V^{JG} \neq \emptyset$ **do**
  Find the vertex $v_{min} \in V^{JG}$ corresponding to LEC feature group P_{min}, where P_{min} has the smallest size;
  Call Function **ComParJoin**($\{v_{min}\}, P_{min}, JG, RS$);
  Remove v_{min} from V^{JG};
  Remove all outliers remaining in JG;

Function ComParJoin(V, P, JG, RS)

---

for *each vertex v in JG adjacent to at least one vertex in V, where v corresponds to LEC feature group* P' **do**
  Set $P'' \leftarrow \emptyset$;
  for *each LEC feature* LF_i *in* P **do**
    for *each LEC feature* LF_j *in* P' **do**
      if LF_i *and* LF_j *are joinable* **then**
        $LF_k \leftarrow LF_i \bowtie LF_j$;
        if *all bits in* $LF_k.LECSign$ *are '1'* **then**
          Insert all LEC features corresponding to vertices in V into RS;
        else
          Put LF_k into P'';
  Call Function **ComParJoin**($V \cup \{v\}, P'', JG$);

D. Analysis

To analyse the complexity of the above optimization technique, we consider its communication cost as well as the computation costs for evaluating a SPARQL query Q on a distributed RDF graph G. The communication cost is the data shipment needed during the distributed query evaluation. In contrast, the computation cost is the response time needed for evaluating the query at different sites in parallel. Generally speaking, our method can guarantee the following.

Communication cost. As discussed before, our optimization technique only needs to assemble the LEC features of all LECs to find out the final results. A general formula for determining the communication cost can be specified as follows:

$$Cost = Cost_{LF} \times |\Psi|$$

where $Cost_{LF}$ is the size of a LEC feature and $|\Psi|$ is the number of LEC features.

For any LEC feature $\{F, g, LECSign\}$, its cost, $Cost_{LF}$, consists of three components. The first component is the cost of the fragment identifer F, which is obvious to be a constant. The second component is the cost of the function g that maps the crossing edges in a local partial match to the query edges. The number of crossing edges is at most $|E^Q|$, so the complexity of g is $O(|E^Q|)$. The last component, $LECSign$, is defined as a bitstring of fixed-length $|V^Q|$, so the cost of $LECSign$ is also $Q(|V^Q|)$. In summary, the cost of any LEC feature is $O(|E^Q| + |V^Q|)$.

On the other hand, the number of LEC features, $|\Psi|$, only depends on the number of crossing edges in fragment F_i , i.e., $|E_i^c|$, due to the LEC features only introduced by these crossing edges. In the worst case, each query edge can map to any edge in E_i^c, and then the number of LEC features is $O(|E_i^c|^{|E^Q|})$. Hence, the number of LEC features is $O(\sum_{i=1}^{|\mathcal{F}|} |E_i^c|^{|E^Q|})$.

Overall, the total communication cost is $O(\sum_{i=1}^{|\mathcal{F}|} |E_i^c|^{|E^Q|} \times (|E^Q| + |V^Q|))$. Thus, given a fragmentation of an RDF graph G to a set of fragments, our optimization technique has the property that the communication cost of evaluating a query is independent of the size of the graph, and depends mainly on the size of the query and the fragmentation of the graph.

Computation cost. There are two parts of our optimization technique: partial evaluation for computing LEC features and assembly for joining LEC features to get the final answer. We discuss the costs of the two stages as follows.

First, computing local partial matches to find out LEC features is performed on each fragment F_i in parallel, and it takes $O(|V_i \cup V_i^e|^{|V^Q|})$ time to compute all local partial matches for each fragment. Hence, it takes at most $O(|V_m \cup V_m^e|^{|V^Q|})$ time to get all LEC features from all sites, where $V_m \cup V_m^e$ is the vertex set of the largest fragment in $\mathcal{F}$.

Second, we only need scan all LEC features once to partition them, so it takes $O(|\Psi|)$ to partition all LEC features. In addition, given a partitioning $\mathcal{P} = \{P_1, ..., P_n\}$, joining all LEC features costs $\prod_{i=1}^{i=n} |P_i|$, which is bounded by $O((\frac{|\Psi|}{|V^Q|})^{|V^Q|})$. As discussed before, $|\Psi|$ is independent of the entire graph G, so the response time is also independent of G.

In summary, the data shipment of our method depends on the size of query graph and the number of crossing edges only; and the response time of our method depends only on the size of query graph, the largest fragment and the number of edges across different fragments. Thus, our method is *partition bounded* in both *data shipment* and *response time* [5].

In real applications, we can expect that the number of crossing edges in a fragmentation will be small compared to the size of the graph itself, i.e., $\sum_{i=1}^{|\mathcal{F}|} |E_i^c| \ll |V|$. Furthermore, after we study the real SPAQRL query workload, the DBpedia query workload[1], the size of a real SPARQL query is often smaller than ten edges. Last, we find out that the query edge in real SPARQL queries often only map to a limited number of edges in E_i^c.

V. LEC Feature-based Assembly

After we gain all local partial matches, we need assemble and join all them to form all complete matches. In this section, we discuss the join-based assembly of local partial matches to compute the final results.

The join method proposed in [19] is a partitioning-based join algorithm, where the local partial matches are divided into multiple partitions based on their internal candidates such that two local partial matches in the same partitions cannot be joinable. The join space of the join algorithm in [19] is still large. Two local partial matches in two different partitions still cannot be joinable according to their corresponding LEC features. Thus, we proposes an optimized technique based on the LEC features of the local partial matches to further reduce the join space.

The intuition of our method is that we divide all local partial matches into multiple groups based on their LEC features such that two local partial matches in the same group cannot be joinable. Then, we only consider joining local partial matches from different groups.

Theorem 6: Given two local partial matches PM_i and PM_j and their LEC features LF_i and LF_j, if $LF_i.LECSign$ is equal to $LF_j.LECSign$, PM_i and PM_j are not joinable.

Proof: Since $LF_i.LECSign$ is equal to $LF_j.LECSign$, $LF_i.\ LECSign \wedge LF_j.LECSign = LF_i.LECSign = LF_i.\ LECSign$. According to Condition 4 of Definition 5, there are at least one internal vertices in a local partial match, so there are at least one '1' in $LF_i.LECSign$ and $LF_i.LECSign$. Therefore, there are at least one '1' in $LF_i.LECSign \wedge LF_j.LECSign$, which is in conflict with Condition 4 of Definition 9. ■

Definition 11: **(LEC Feature-based Local Partial Match Group)**. Let Ω denote all local partial matches. $\mathcal{G} = \{Gr_1, ..., Gr_n\}$ is a set of local partial match groups for Ω if and only if each group Gr_i $(i = 1, ..., n)$ consists of a set of local partial matches, the corresponding LEC features of which have the same $LECSign$.

Example 8: Given all local partial matches in Fig. 3, their corresponding LECSign-based local partial match groups $\{Gr_1, Gr_2, Gr_3, Gr_4\}$ are as follows.

$$Gr_1 = \{PM_1^1, PM_1^2, PM_3^2\}, Gr_2 = \{PM_1^3\}$$

$$Gr_3 = \{PM_2^1, PM_2^2, PM_3^1\}, Gr_4 = \{PM_2^3\}$$

□

Given a set $\mathcal{G}$ of LECSign-based local partial match groups, we also build a *local partial match group join graph* (denoted as $LG = \{V^{Gr}, E^{Gr}\}$) as follows. In a join graph, one vertex indicates a LEC feature-based local partial match group. We introduce an edge between two vertices in the join graph if and only if their corresponding LEC features are joinable. Fig. 7 shows the join graph of $\mathcal{G}$.

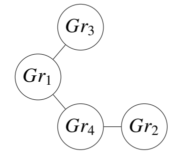


Fig. 7. Local Partial Match Group Join Graph

We propose an algorithm (Algorithm 3) based on the DFS traversal over the local partial match group join graph to get the complete matches.

Algorithm 3: LEC Feature-based Assembly Algorithm

Input: A set $\mathcal{G} = \{Gr_1, ..., Gr_n\}$ of LEC feature-based local partial match groups and its join graph LG
Output: The set of complete matches, MS

while $V^{Gr} \neq \emptyset$ **do**
    Find the vertex $v_{min} \in V^{Gr}$ corresponding to LEC feature group Gr_{min}, where Gr_{min} has the smallest size;
    Call Function **ComParJoin**($\{v_{min}\}, Gr_{min}, LG, MS$);
    Remove v_{min} from V^{Gr};
    Remove all outliers remaining in JG;
Return false;

Function ComParJoin(V, Gr, LG, MS)

for *each vertex v in JG adjacent to at least one vertex in V, where v corresponds to* Gr' **do**
    $Gr'' \leftarrow \emptyset$;
    for *each local partial match* PM_i *in* Gr **do**
        for *each local partial match* PM_j *in* Gr' **do**
            if PM_i *and* PM_j *are joinable* **then**
                $PM_k \leftarrow PM_i \bowtie PM_j$;
                if *all vertices in* PM_k *are matched* **then**
                    Put PM_k into MS;
                else
                    Put PM_k into Gr'';
    Call Function **ComParJoin**($V \cup \{v\}, Gr'', LG, MS$);

VI. Further Optimization – Assembling Variables' Internal Candidates

In this section, we present another optimization technique: *assembling variables' internal candidates*. This technique is based on the internal candidates of all variables in each site to filter out some false positives.

Existing RDF database systems used in sites storing individual partitions often adopt the filter-and-evaluate framework. These systems first compute out the candidates of all variables, and then search matches over all candidates. The process of finding candidates is very quick and it often does not take much time. Hence, we can modify the code of these systems and assemble the internal candidates in the coordinator site.

[1] http://aksw.org/Projects/DBPSB.html

Algorithm 4: Assembling Variables' Internal Candidates

Input: Fragments $\mathcal{F} = \{F_1, ..., F_m\}$ of RDF graph G over sites $\{S_1, ..., S_m\}$, coordinator site S_c, and the SPARQL query Q.
Output: The internal candidate set $C(Q, v)$ of any variable v in Q.

The Coordinator Site S_c**:**
for *each variable* v *in* Q **do**
  $B_v \leftarrow 0$;
  for *each site* S_i **do**
    Receive B'_v from S_i;
    if B'_v *is the first bit vector that* S_c *receives* **then**
      $B_v \leftarrow B'_v$;
    else
      $B_v \leftarrow B_v \vee B'_v$;
for *each site* S_i **do**
  for *each variable* v *in* Q **do**
    Send B_v to S_i;
The Site S_i**:**
Find $C(Q, v)$ and $B_v \leftarrow 0$;
for *each candidate* c *in* $C(Q, v)$ **do**
  Use a hash function h to map c to a integer $h(c)$;
  Set the $h(c)$-th bit of B_v to 1;
Send B_v to S_c;
for *each variable* v *in* Q **do**
  Receive B_v to S_c;
  for *each candidate* c *in* $C(Q_i, v)$ **do**
    Use a hash function h to map c to a integer $h(c)$;
    if *the* $h(c)$*-th bit of* B_v *is equal to 0* **then**
      Remove c from $C(Q_i, v)$;

When a set of internal candidates for variable v has been found, we do not find local partial matches directly but send the set of candidates to the coordinator site.

The major benefit for assembling variables' internal candidates is to avoid some false positive local partial matches. When a site finds local partial matches, it does not consider how to join with local partial matches in other sites. Hence, many unnecessary candidates may be generated, and these candidates do not appear in any complete matches. To filter out these unnecessary candidates, the coordinator site can assemble all these internal candidates and unions the candidates sets of a variable from different sites. If a candidate of variable v can appear in a complete match, it belongs to the v's internal candidate sets from all sites

In practice, there may be too many internal candidates for each variable, which result in high communication cost. For reducing the communication cost, we compress the information of all internal candidates for each variable into a fixed length bit vector. For variable v, we associate it with a fixed length bit vector B_v. We define a hash function to map the URI of each v's internal candidate in a site to a bit in B_v. Then, all v's internal candidates can be compressed in B_v. Thus, the coordinator site only needs to assemble all bit vectors of variables from different sites and do bitwise OR operations over bit vectors of a variable from different sites. The result bit vectors compresses the information of all internal candidates. We can send the result bit vectors of all variables to different sites and filter out some false positive candidates. When we compute the local partial matches, we avoid forming the local partial matches over those extended candidates that do not appears in the assembled internal candidates. Because the length of a bit vector is fixed, the communication cost is not too expensive.

Smaller search space can speed up evaluating the SPARQL query, meanwhile modern distributed environments have much faster communication networks than in the past. Therefore, it is beneficial for us to afford the cost of communicating the candidate numbers of all variables between the coordinator site and the sites.

Algorithm 4 describes the optimization of assembling variables' internal candidates. For the coordinator site, it receives the information of all variables' candidates from different sites. The information includes two parts: the numbers and bit vectors of candidates of all variables. The coordinator site assembles all internal candidates by unioning all the bit vectors. Then, the coordinator site sends the distributed execution plan and the result bit vectors of all variables to sites. For each site, it firstly finds out the candidates of variables locally and compresses these candidates into bit vectors. It then sends the number of candidates and all bit vectors to the coordinator site. The site waits for the bit vectors of all variables from the coordinator site. With the received bit vectors of all variables, the site can filter out many false positive extended candidates. Finally, the site also starts to its local partial matches.

VII. Experiments

In this section, we use some real and synthetic RDF datasets to conduct our experiments.

A. Setting

LUBM. LUBM [8] is a benchmark that adopts an ontology for the university domain, and can generate synthetic OWL data scalable to an arbitrary size. We generate three datasets of triples from 100 million to 1 billion. We use the 7 benchmark queries in [1] (denoted as $LQ_1 - LQ_7$) to test our methods.

YAGO2. YAGO2 [13] is a real RDF dataset that is extracted from Wikipedia. YAGO2 also integrates its facts with the WordNet thesaurus. It contains about 284 million triples. We use the benchmark queries in [1] (denoted as $YQ_1 - YQ_4$) to evaluate our methods.

BTC. BTC[2] is a real dataset that serves as the basis of submissions to the Billion Triples Track of the Semantic Web Challenge. After eliminating all redundant triples, this dataset contains about 1 billion triples. We use the 7 queries (denoted as $BQ_1 - BQ_7$) in [19] to test our methods.

We conduct all experiments on a cluster of 12 machines running Linux, each of which has two CPU with six cores of

[2]http://km.aifb.kit.edu/projects/btc-2012/

		Partial Evaluation						Assembly	Total Time (in ms)	Local Partial Matches' Number	Matches' Number	Crossing Matches' Number
		Assembling Variables' Internal Candidates		Time of Local Partial Match Computation (in ms)	LEC Feature-based Optimization		Time(in ms)	Time of LEC Feature-based Assembly (in ms)				
		Time(in ms)	Data Shipment (in KB)		Time(in ms)	Data Shipment (in KB)						
LQ_1		4,029	2,032	21,550	2,054	38,882	27,633	12,539	40,172	276,327	21	21
LQ_2		0	0	8,488	0	0	0	0	8,488	0	864,197	0
LQ_3		568	16	2,795	0	0	3,363	0	3,363	0	0	0
LQ_4	√	0	0	221	0	0	0	0	221	0	10	0
LQ_5	√	0	0	187	0	0	0	0	187	0	10	0
LQ_6	√	1,556	136	1,516	61	1	3,133	9	3,142	228	125	114
LQ_7		7,827	2,268	25,779	2,323	5,057	35,929	12,582	48,511	973,255	35,434	35,077

TABLE I
EVALUATION OF EACH STAGE ON LUBM 100M

		Partial Evaluation						Assembly	Total Time (in ms)	Local Partial Matches' Number	Matches' Number	Crossing Matches' Number
		Assembling Variables' Internal Candidates		Time of Local Partial Match Computation (in ms)	LEC Feature-based Optimization		Time(in ms)	Time of LEC Feature-based Assembly (in ms)				
		Time(in ms)	Data Shipment (in KB)		Time(in ms)	Data Shipment (in KB)						
YQ_1		188	13	1,007	879	6	2,094	79	2,153	811	17	17
YQ_2		315	15	999	26	1	1,340	0	1,340	0	0	0
YQ_3		1,341	137	3,292	1,599	1,317	6,232	21,404	27,636	816,382	605,993	588,390
YQ_4		388	27	2,036	1,602	293	4,026	686	4,712	16,661	226	224

TABLE II
EVALUATION OF EACH STAGE ON YAGO2

		Partial Evaluation						Assembly	Total Time (in ms)	Local Partial Matches' Number	Matches' Number	Crossing Matches' Number
		Assembling Variables' Internal Candidates		Time of Local Partial Match Computation (in ms)	LEC Feature-based Optimization		Time(in ms)	Time of LEC Feature-based Assembly (in ms)				
		Time(in ms)	Data Shipment (in KB)		Time(in ms)	Data Shipment (in KB)						
BQ_1	√	0	0	259	0	0	0	0	259	0	1	0
BQ_2	√	0	0	269	0	0	0	0	269	0	2	0
BQ_3	√	0	0	187	0	0	0	0	187	0	0	0
BQ_4	√	39,842	2,699	45,723	2,511	1	88,076	93	88,169	5	4	4
BQ_5	√	45,962	1,929	6,858	1,504	1	54,324	2	54,326	16	12	11
BQ_6		19,663	1,047	1,589	756	1	22,008	2	22,010	0	0	0
BQ_7		35,849	3,071	21,233	2,848	1	59,930	24	59,954	0	0	0

TABLE III
EVALUATION OF EACH STAGE ON BTC

1.2GHz. Each machine has 128GB memory and 28TB disk storage. We select one of these machines as the coordinator machine. We use MPICH-3.0.4 running on C++ to join the partial results. We randomly partition all datasets into several fragments. We assign each vertex v in RDF graph to the i-th fragment if $H(v) MOD N = i$, where $H(v)$ is a hash function and $N = 12$ is the number of fragments. Each machine stores a single fragment.

In this paper, we revise gStore [26] to find local partial matches at each site. We denote our method as gStore^D. We compare our approach with three state-of-the-art disk-based distributed RDF systems in recent three years, including DREAM [10], S2X [20], S2RDF [21] and CliqueSquare [6]. The codes of these systems are released by [1] in GitHub[3]. We also release our codes in GitHub[4].

B. Evaluation of Each Stage

In this experiment, we study the performance of our approaches at each stage (i.e., partial evaluation and assembly process) with regard to different queries in LUBM 100M, YAGO2 and BTC. We report the running time of each stage, the size of data shipment, the number of intermediate and complete results, and the communication time, with regard to different queries in Tables I, II and III. Generally speaking, the query performance mainly depends on three factors: the shape of the query graph and the existence of the selective triple patterns.

For the shape of the query graph, we divide all benchmark SPARQL queries into two categories according to the complexities of their structures: star and other shapes. The evaluation times for star queries (LQ_2, LQ_4 and LQ_5 in LUBM, and BQ_1, BQ_2 and BQ_3 in BTC) are short, while queries of other shapes have longer times. Each crossing edge in the distributed RDF graph is replicated, so any results of star queries are certain to be in a single fragment and there are not any local partial matches generated during the query processing. Because our optimization techniques focus on reducing the number of local partial matches, we can directly compute out the results over each fragment without considering communications and our optimization techniques.

[3]https://github.com/ecrc/rdf-exp
[4]https://github.com/bnu05pp/gStoreD

Thus, the evaluation times of star queries are short. In contrast, queries of other query shapes involve multiple fragments and generate local partial matches, which increase the search space of partial evaluation and cause the optimization techniques and the assembly process. Thus, queries of other query shapes has worse performance.

For the selective triple patterns, our method processes queries with selective triple patterns faster than queries without selective triple patterns. The performance of our method is dependent on the computation and assembly of local partial matches. The selective triple patterns can be used to filter out many irrelevant candidates and local partial matches, which greatly reduce the search space for computing and joining the local partial matches. Thus, if there are some selective triple patterns in the query, the performance of the query is better.

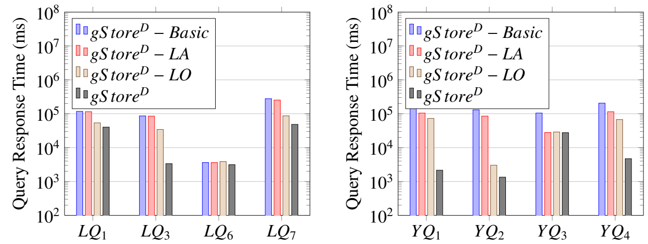
C. Evaluation of Different Optimizations

The aim of this experiment is to use LUBM 100M and YAGO2 to test the effect of the three optimization techniques proposed in this paper. Here, because some queries can be answered at each fragment locally and they can be evaluated without involving any optimization techniques, we only consider the benchmark queries that the assembly process (LQ_1, LQ_3, LQ_6 and LQ_7 in LUBM and all queries in YAGO2) in our experiments. We use the method proposed in [19] that does not utilize any optimization techniques proposed in this paper as a baseline (denoted as $gStore^D$-Basic); we also design a baseline only using the optimization of the LEC feature-based assembly (denoted as $gStore^D$-LA) and a baseline only using the optimizations of the LEC feature-based assembly and LEC feature-based optimization (denoted as $gStore^D$-LO). Fig. 8 shows the experiment results.

Generally speaking, the optimization of LEC feature-based assembly only repartitions the local partial matches to reduce the join space and does not leads to the extra communications, so $gStore^D$-LA has the same partial evaluation stage to $gStore^D$-Basic and their difference is only on the assembly stage. Because $gStore^D$-LA optimizes the joining order without the extra communications, it always faster than $gStore^D$-Basic. On the other hand, for the optimizations of assembling variables' internal candidates and LEC feature-based optimization, they lead to the extra communications for internal candidates and local partial matches, so they may result in extra processing times. However, the optimizations are effective and improve the performance in most cases. Especially for the selective queries of complex shapes (LQ_3 in LUBM and YQ_1, YQ_2, YQ_4 in YAGO2), the optimizations can improve the performance by orders of magnitude.

D. Scalability Test

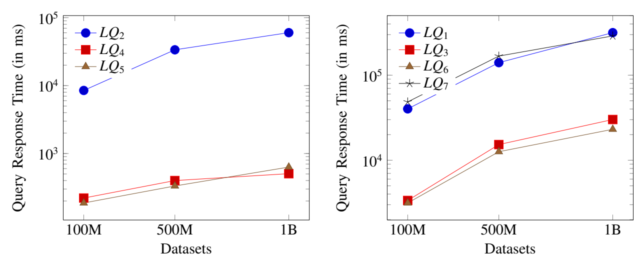
In the above two experiments the data size was kept constant, and we investigate the effect of data size on query evaluation times in this experiment. We generate three LUBM datasets varying the from 100 million to 1 billion triples to test our method. Fig. 9 shows the experiment results. As mentioned in Section VII-B, we divide the queries into four categories according to their structures: star queries (LQ_2, LQ_4 and LQ_5) and other queries (LQ_1, LQ_3, LQ_6 and LQ_7).



(a) LUBM 100M (b) YAGO2

Fig. 8. Evaluation of Different Optimizations

Generally speaking, since the number of crossing edges linearly increases as the data size increases and our approach is partition bounded, the query response time also increases proportional to the data size. Here, for queries of other shapes, the query response times may grow faster. This is because the other query graph shapes cause more complex operations in query processing, such as joining and assembly, and larger number of local partial matches. However, even for queries of complex structures, the query performance is scalable with RDF graph size on the benchmark datasets.



(a) Star Queries (b) Other Queries

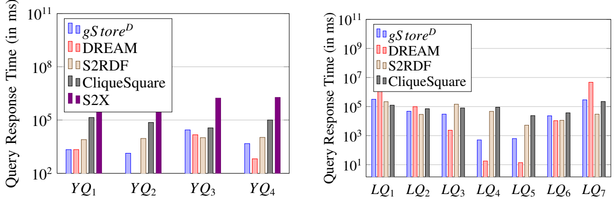
Fig. 9. Scalability Test

E. Online Performance Comparison

This experiment is a comparative evaluation of our method against several distributed SPARQL processing systems proposed in recent three years on the very large datasets with more than one billion triples, LUBM 1B. Fig. 10 shows the performance of different approaches.

The competitors are all of three state-of-the-art public disk-based distributed RDF systems proposed in recent three years, including DREAM [10], S2X [20], S2RDF [21] and CliqueSquare [6]. As surveyed in [1], other distributed RDF systems in recent three years are either not released or memory-based systems that are in different environments than what we target in this paper. Note that, all results are provided in [1] and S2X fails to run all queries on LUBM 1B.

Generally, although our approach sacrifices the efficiency to be adaptive for any partitioning strategies, its efficiency is still comparable to other distributed RDF systems. Specifically, our approach and DREAM are two specialized distributed



(a) YAGO2

(b) LUBM 1B

Fig. 10. Online Performance Comparison

RDF systems, while S2X, S2RDF and CliqueSquare are three cloud-based systems. The cloud-based systems use existing cloud platforms, like Hadoop and Spark, to manage large RDF datasets, which suffer from the expensive overhead of scans and joins in the cloud. Only when the queries (LQ_1, LQ_2 and LQ_7 in LUBM) are unselective and evaluated over the very large RDF dataset (LUBM 1B) that can generate many intermediate results, they have better performances. For these queries, our approach can outperform DREAM, which builds a database of the entire RDF dataset at each site. DREAM incurs significant overhead for querying the entire database on a single machine and generating many results.

On the other hand, when the queries (LQ_3, LQ_4, LQ_5 and LQ_6 in LUBM) are selective or the RDF dataset (YAGO2) is not very large, our system can outperform the cloud-based systems in most cases. Here, DREAM [10] utilizes full data replication. Although the full data replication in DREAM causes higher space cost than our system, it can greatly reduce the evaluation times of these queries. Note that, DREAM fails to process YQ_2

VIII. Related Work

Distributed SPARQL Query Processing. There have been many works on distributed SPARQL query processing, and a very good survey is [15]. In the last three years, some recent approaches [2], [6], [25], [24], [11], [12], [18], [10], [21] are proposed. We classify them into three classes: cloud-based approaches, partitioning-based approaches, and partitioning-tolerant approaches.

First, there have been some recent works (e.g., [2], [6], [21], [20]) focusing on managing large RDF datasets using existing cloud platforms. CliqueSquare [2], [6] first discuss how to build query plans relying on n-ary (star) equality joins in Hadoop. S2RDF [21] uses the relational interface of Spark to store the RDF data in the vertical partitioning schema and materializes some extra join relations between some vertical partitioning tables. In the online phase, S2RDF transforms the query into many SQL queries and merges the results of the SQL queries. S2X [20] uses GraphX in Spark to evaluate SPARQL queries. Query evaluation starts by distributing all triple patterns to all vertices. Vertices cooperatively validate their triple candidacy with their neighbours by exchanging messages. Then, the partial results are collected and merged.

Second, the partition-based approaches [25], [24], [11], [12], [18] divide an RDF graph G into several partitions. Each partition is placed at a site which installs an existing centralized RDF system to manage it. At run time, a SPARQL query is decomposed into several subqueries, and each subquery can be answered locally at one site. The results of the subqueries are finally merged. Each of these papers approaches has its own data partitioning strategy, and different partitioning strategies result in different query decomposition methods. DiploCloud [25] asks the administrator to define some templates as the partition unit. Then, DiploCloud stores the instantiations of the templates in compact lists as in a column-oriented database system; PathBMC [24] adopts the end-to-end path as the partition unit to partition the data and query graph; AdHash [11] and AdPart [12] uses a straightforward partitioning framework by using the subject values and mainly discuss how to optimize the distributed query evaluation to reduce the communication cost; Peng et al. [18] first mine some frequent patterns in the query log, and use them to define the partitioning unit.

DREAM [10] and Peng et al. [19] are two other approaches that do not neither partition RDF graphs nor use existing cloud platforms. In DREAM [10], each site maintains the whole RDF dataset. For query processing, DREAM runs a query planner that divides the input query into subqueries. Then, DREAM executes each subquery in a separate site and merge the intermediate results to produce the final matches. Peng et al. [19] propose a partition-tolerant distributed approach based on the “partial evaluation and assembly” framework. However, its efficiency has large potential to improve.

Partial Evaluation. Recently, partial evaluation has been used for evaluating queries on distributed graphs [17], [4], [5], [9], [22]. In [4], [9], the authors provide algorithms for evaluating reachability queries on distributed graphs based on partial evaluation. In [17], the authors provides partial evaluation algorithms and optimizations for graph simulation in a distributed setting, while [5] further studies what is doable and what is undoable for distributed graph simulation. Recently, Peng et al. [19] discuss how to employ the “partial evaluation and assembly” framework to handle SPARQL queries, while Wang et al. [22] discuss how to answer regular path queries on large-scale RDF graphs using partial evaluation. However, both of them do not provide the performance guarantees on the total network traffic and the response time.

IX. Conclusion

In this paper, we propose three optimizations to improve the partial evaluation-based distributed SPARQL query processing approach. The first is to compress the partial evaluation results in a compact data structure named *LEC feature* and communicate them among sites to filter out some irrelevant partial evaluation results while providing some performance guarantees. The second one is the LEC feature-based assembly of all local partial matches to filter out some intermediate results. Last, we propose an optimization that communicate variables’ candidates among the sites to prune some irrelevant candidates. In addition, we do extensive experiments to confirm our approach.

References

[1] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB*, 10(13):2049–2060, 2017.

[2] B. Djahandideh, F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis. CliqueSquare in Action: Flat Plans for Massively Parallel RDF Queries. In *ICDE*, pages 1432–1435, 2015.

[3] O. Erling. Virtuoso, A Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[4] W. Fan, X. Wang, and Y. Wu. Performance Guarantees for Distributed Reachability Queries. *PVLDB*, 5(11):1304–1315, 2012.

[5] W. Fan, X. Wang, Y. Wu, and D. Deng. Distributed Graph Simulation: Impossibility and Possibility. *PVLDB*, 7(12):1083–1094, 2014.

[6] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis. CliqueSquare: Flat Plans for Massively Parallel RDF Queries. In *ICDE*, pages 771–782, 2015.

[7] Google. Freebase data dumps. *https://developers.google.com/freebase/data*, 2018.

[8] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[9] S. Gurajada and M. Theobald. Distributed Set Reachability. In *SIGMOD Conference*, pages 1247–1261, 2016.

[10] M. Hammoud, D. A. Rabbou, R. Nouri, S. Beheshti, and S. Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *PVLDB*, 8(6):654–665, 2015.

[11] R. Harbi, I. Abdelaziz, P. Kalnis, and N. Mamoulis. Evaluating SPARQL Queries on Massive RDF Datasets. *PVLDB*, 8(12):1848–1859, 2015.

[12] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning. *The VLDB Journal*, pages 1–26, 2016.

[13] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[14] N. D. Jones. An Introduction to Partial Evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.

[15] Z. Kaoudi and I. Manolescu. RDF in the Clouds: A Survey. *VLDB J.*, 24(1):67–91, 2015.

[16] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[17] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed Graph Pattern Matching. In *WWW*, pages 949–958, 2012.

[18] P. Peng, L. Zou, L. Chen, and D. Zhao. Query Workload-based RDF Graph Fragmentation and Allocation. In *EDBT*, pages 377–388, 2016.

[19] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao. Processing SPARQL Queries over Distributed RDF Graphs. *VLDB J.*, 25(2):243–268, 2016.

[20] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. S2X: Graph-Parallel Querying of RDF with GraphX. In *Big-O(Q)/DMAH@VLDB*, pages 155–168, 2015.

[21] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF Querying with SPARQL on Spark. *PVLDB*, 9(10):804–815, 2016.

[22] X. Wang, J. Wang, and X. Zhang. Efficient Distributed Regular Path Queries on RDF Graphs Using Partial Evaluation. In *CIKM*, pages 1933–1936, 2016.

[23] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[24] B. Wu, Y. Zhou, P. Yuan, L. Liu, and H. Jin. Scalable SPARQL Querying Using Path Partitioning. In *ICDE*, pages 795–806, 2015.

[25] M. Wylot and P. Mauroux. DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud. *TKDE*, PP(99), 2015.

[26] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gStore: a Graph-based SPARQL Query Engine. *VLDB J.*, 23(4):565–590, 2014.