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. In addition, although our approach is partitioning-tolerant, different partitioning strategies result in different performances and we also evaluate different partitioning strategies for our approach. 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, it is necessary for us to design a distributed database system to process SPARQL queries. In many applications, the 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, European Bioinformatics Institute[1] has built up a uniform platform for users to query multiple bioinfomatics RDF datasets, including BioModels, Biosamples, ChEMBL, Ensembl, Atlas, Reactome and OLS. These datasets are provided by different data publishers and had better be administratively partitioned according to their data publishers. Thus, partitioning-tolerant SPARQL processing is desirable.

For partitioning-tolerant SPARQL processing on distributed RDF graphs, Peng et al.[17] discuss how to evaluate SPARQL queries in the "partial evaluation and assembly" framework. However, its efficiency has great potential to be improved. Its major bottleneck is the large volume of partial evaluation results, which causes such a high cost for generating and assembling them.

In this paper, we propose several optimizations for the "partial evaluation and assembly" framework [17] to prune the irrelevant partial evaluation results and assemble them efficiently to form the final results. 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* [3]. The second one is to assemble all local partial matches based on their LEC features as well. Last, to further avoid redundant computations within the sites, we propose an optimization that communicate variables' candidates among the sites to prune some irrelevant candidates. In addition, although our approach is partitioning-tolerant, different partitioning strategies result in different performances and we also evaluate different partitioning strategies for our approach.

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

- We explore the intrinsic structural characteristics of partial results to compress them into a compact data structure, *LEC feature*. We communicate and utilize the LEC features to pruning some irrelevant partial results. We prove theoretically that the LEC feature can guarantee the performance of the pruning optimization in both *response time* and *data shipment*.

---

[1] https://www.ebi.ac.uk/rdf/

- We propose an efficient LEC feature-based assembly algorithm to merge all partial results together and form the final results.
- We present an optimization based on the communication of the variables' internal candidates among different sites to further avoid redundant computations within the sites.
- We define a specific cost model for our method to measure the cost of different partitioning strategies and select the best partitioning from the existing partitionings.
- 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 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**) Let u and $\overrightarrow{uu'}$ denote the vertex and edge in an 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.,

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

5) Vertices in V_i^e are called *extended* vertices of F_i, vertices in V_i are called *internal* vertices of F_i;
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 $E_1^c = \{\overrightarrow{001, 006}, \overrightarrow{006, 005}, \overrightarrow{001, 012}\}$.□
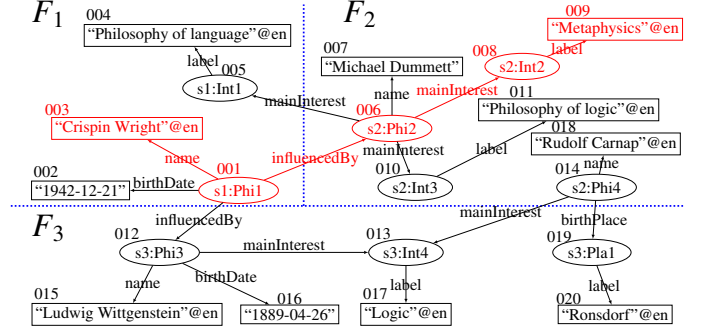

Fig. 1. Distributed RDF Graph

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.

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. □

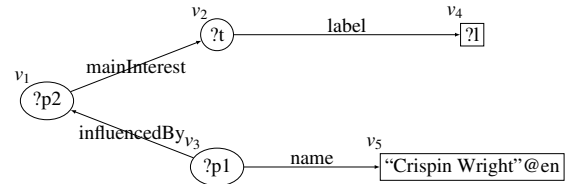Fig. 2. SPARQL Query Graph

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: If v_i is not a variable, $f(v_i)$ and v_i have the same URI or literal value $(1 \leq i \leq n)$; If v_i is a variable, there is no constraint over $f(v_i)$ except that $f(v_i) \in \{u_1, ..., u_m\}$; 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*

F_1

006 s2:Phi2
influencedBy
001 s1:Phi1 —name→ 003 "Crispin Wright"@en
[006,NULL,001,NULL,003]
PM_1^1

012 s3:Phi3
influencedBy
001 s1:Phi1 —name→ 003 "Crispin Wright"@en
[012,NULL,001,NULL,003]
PM_1^2

005 s1:Int1 —label→ 004 "Philosophy of language"@en
mainInterest
006 s2:Phi2
[006,005,NULL,004,NULL]
PM_1^3

F_2

008 s2:Int2 —label→ 009 "Metaphysics"@en
mainInterest
006 s2:Phi2
influencedBy
001 s1:Phi1
[006,008,001,009,NULL]
PM_2^1

010 s2:Int3 —label→ 011 "Philosophy of logic"@en
mainInterest
006 s2:Phi2
influencedBy
001 s1:Phi1
[006,010,001,011,NULL]
PM_2^2

005 s1:Int1
mainInterest
006 s2:Phi2
influencedBy
001 s1:Phi1
[006,005,001,NULL,NULL]
PM_2^3

F_3

013 s3:Int4 —label→ 017 "Logic"@en
mainInterest
012 s3:Phi3
influencedBy
001 s1:Phi1
[012,013,001,017,NULL]
PM_3^1

013 s3:Int4 —label→ 017 "Logic"@en
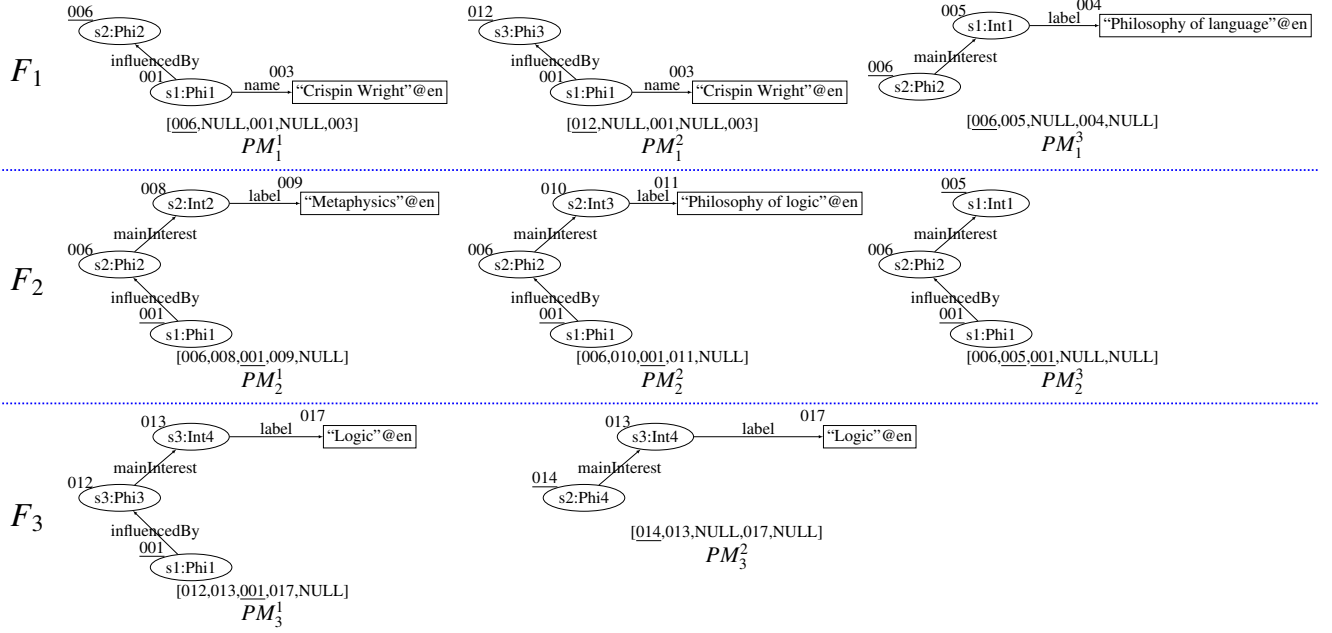mainInterest
014 s2:Phi4
[014,013,NULL,017,NULL]
PM_3^2

Fig. 3.   Example Local Partial Matches

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 sites such that F_i is located at S_i. Given a SPARQL BGP query 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. Finding matches in a site can be evaluated locally using a centralized RDF triple store. In this paper, we only focus on how to find the matches crossing multiple sites efficiently. In our prototype experiments, we modify gStore [23] to perform partial evaluation.

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 [17], we give its brief background here.

In our framework, each site S_i receives the full query graph Q and computes the partial answers (called *local partial matches*) based on the known input F_i (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. 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 $\{u_1, \ldots, u_n\}$ ($n \le |V^Q|$) in a fragment F_k, PM is a *local partial match* in F_k if and only if there exists a function $f : V^Q \rightarrow \{u_1, \ldots, u_n\} \cup \{NULL\}$ that holds the following conditions:

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, $f(v_i) \in \{u_1, \ldots, u_n\}$ or $f(v_i) = NULL$.
3) If there exists an edge $\overrightarrow{v_i v_j}$ in Q ($i \ne j$), 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_k^e, $f(v_i) = NULL$, 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 $f(v_j) \ne NULL$ 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 internal vertices of F_k, 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.

Vector $[f(v_1), \ldots, f(v_n)]$ 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.

Generally, a local partial match is a subset of a complete SPARQL match. The first three conditions in Definition 5 are

analogous to SPARQL match while vertices of query Q are allowed to match a special value NULL. The fourth condition requires that a local partial match must have at least one crossing edges, since it is used to form the possible crossing match. The fifth condition is that if vertex v (in query Q) is matched to an internal vertex, all neighbours of v should be matched in this local partial match as well. The sixth condition is to ensure the correctness of our framework [17].

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.

For example, PM_1^1 is the overlapping part between the crossing match discussed in Example 3 and fragment F_1. PM_1^1 contains a crossing edge $\overrightarrow{001, 006}$. In PM_1^1, the query vertices v_3 and v_5 are matched to the internal vertices 001 and 003 of F_1, so v_3 and v_5 are weakly connected and all neighbors of v_3 and v_5 are also matched. □

For a SPARQL query, its local partial matches bear structural similarities (see Section IV-A), so they can be represented as vectors of Boolean formulas associated with crossing edges (see Section IV-B). 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 [17] to find local partial matches.

III. Overview

We extend the *partial evaluation and assembly* [11] framework to answer SPARQL queries over a distributed RDF graph G as shown in Fig. 4. In our execution model, there are two stages: the partial evaluation stage and the assembly stage.

In the partial evaluation stage, each site S_i first receives the full query graph Q and find all sets of internal candidates. The coordinator site assembles all sets of internal candidates from different sites and gains the candidates sets of all variables (Section VI). The coordinator site distributes the candidates sets and each site use them to find out the local partial matches of Q in F_i, at each site S_i. We explore the intrinsic structural similarities of local partial matches to divide these local partial matches into some equivalence classes and propose a compact data structure, named *LEC feature* (Definition 8), to compress them. Only by joining LEC features can we determine which local partial matches can contribute to the complete matches (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 partitioning of the graph (Section IV-D).

In the assembly stage, we divide all local partial matches into some groups and propose a join algorithm based on the LEC features (Section V).
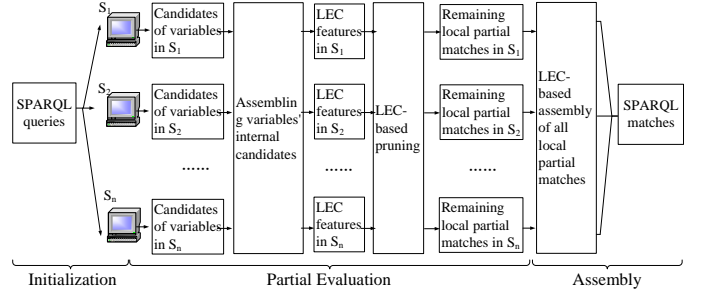

Fig. 4. Overview of Our Method

IV. LEC Feature-based Optimization

A. Local Partial Match Equivalence Class

As discussed in [17], 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 $\overrightarrow{v_3 v_1}$ 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 formalize the 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 $f_i^{-1}(PM_i) = f_j^{-1}(PM_j)$ where $f_i^{-1}(PM_i)$ and $f_i^{-1}(PM_i)$ are the subgraphs (of Q) induced by the matched vertices, if they meet the following conditions:

1) $\forall \overrightarrow{u_i u_j} \in PM_i(or \ PM_j)$, if $\overrightarrow{u_i u_j} \in E_k^c$, $\overrightarrow{u_i u_j} \in PM_j(or \ PM_i)$;
2) $\forall \overrightarrow{u_i u_j} \in PM_i(or \ PM_j)$, if $\overrightarrow{u_i u_j} \in E_k^c$, $f_i^{-1}(u_i) = f_j^{-1}(u_i)$ 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 internal 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 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 Ω denote all local partial matches and $\sim$ 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, all local partial matches equivalent to a local partial match PM_i can be combined together to form the *Local partial match Equivalence Class (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 that if two local partial matches can join together, then all other local partial matches in the corresponding LECs of the 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 [17], 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.

$F_1 : [PM_1^1] = \{PM_1^1\}$; $[PM_1^2] = \{PM_1^2\}$, $[PM_1^3] = \{PM_1^1\}$;
$F_2 : [PM_2^1] = [PM_2^2] = \{PM_2^1, PM_2^2\}$, $[PM_2^3] = \{PM_2^3\}$;
$F_3 : [PM_3^1] = \{PM_3^1\}$, $[PM_3^2] = \{PM_3^2\}$;

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* $LF([PM]) = \{F, g, LECSign\}$ consists of three components:

1) The fragment identifier, F, that PM is from;
2) A function g, which maps crossing edge $\overrightarrow{u_i u_j}$ 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 the i-th bit is set to '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; $\{\overrightarrow{001, 006} \to \overrightarrow{v_3 v_1}\}$ 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].

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

$LF([PM_1^1]) = \{F_1, \overrightarrow{\{001, 006} \to \overrightarrow{v_3 v_1}\}, [00101]\}$

$LF([PM_1^2]) = \{F_1, \overrightarrow{\{001, 012} \to \overrightarrow{v_3 v_1}\}, [00101]\}$

$LF([PM_1^3]) = \{F_1, \overrightarrow{\{006, 005} \to \overrightarrow{v_1 v_2}\}, [01010]\}$

$LF([PM_2^1]) = LF([PM_2^2]) = \{F_2, \overrightarrow{\{001, 006} \to \overrightarrow{v_3 v_1}\}, [11010]\}$

$LF([PM_2^3]) = \{F_2, \overrightarrow{\{006, 005} \to \overrightarrow{v_1 v_2}}, \overrightarrow{001, 006} \to \overrightarrow{v_3 v_1}\}, [10000]\}$

$LF([PM_3^1]) = \{F_3, \overrightarrow{\{001, 012} \to \overrightarrow{v_3 v_1}\}, [11010]\}$

$LF([PM_3^2]) = \{F_3, \overrightarrow{\{014, 013} \to \overrightarrow{v_1 v_2}\}, [01010]\}$ □
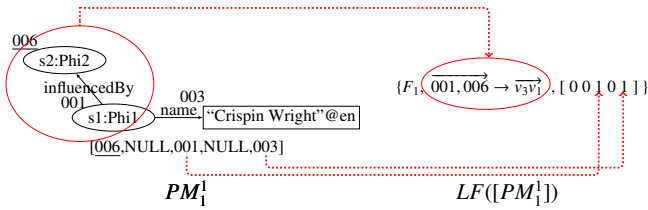

Fig. 5. LEC Feature $LF([PM_1^1])$ (PM_1^1 is the only element in $[PM_1^1]$)

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.

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

---

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 $\Psi(F_i)$.

1 **for** *each local partial match PM in $\Omega(F_i)$* **do**
2   Initialize a LEC feature LF;
3     $LF.F \leftarrow F_i$;
4   **for** *each mapping $(\overrightarrow{u_i u_j}, \overrightarrow{v_i v_j})$ in PM* **do**
5   **if** *u_i is an extended vertex of fragment F_i* **then**
6           $LF.LECSign[i] \leftarrow$ '0';
7           $LF.g \leftarrow LF.g \cup (\overrightarrow{u_i u_j}, \overrightarrow{v_i v_j})$;
8   **else**
9           $LF.LECSign[i] \leftarrow$ '1';
10   **if** *u_j is an extended vertex of fragment F_i* **then**
11           $LF.LECSign[j] \leftarrow$ '0';
12           $LF.g \leftarrow LF.g \cup (\overrightarrow{u_i u_j}, \overrightarrow{v_i v_j})$;
13   **else**
14           $LF.LECSign[j] \leftarrow$ '1';
15   **if** *$\Psi(F_i)$ does not contain LF* **then**
16        $\Psi(F_i) \leftarrow \Psi(F_i) \cup LF$;
17 Return $\Omega(F_i)$;

---

mapping $(\overrightarrow{u_i u_j}, \overrightarrow{v_i v_j})$, if u_i (or u_j) is an extended vertex, we set $LF.LECSign[i]$ (or $LF.LECSign[j]$) as '0' and $LF.LECSign[i]$ (or $LF.LECSign[j]$) as '1' and add $(\overrightarrow{u_i u_j}, \overrightarrow{v_i v_j})$ 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, so 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 $\overrightarrow{u_i u_j}$, such that $LF([PM_i]).g(\overrightarrow{u_i u_j}) = LF([PM_j]).g(\overrightarrow{u_i u_j})$;
3) There exist no two edges $\overrightarrow{u_i u_j}$ and $\overrightarrow{u_i' u_j'}$ in the domains of $LF([PM_i]).g$ and $LF([PM_j]).g$, respectively, such that $LF([PM_i]).g(\overrightarrow{u_i u_j}) = LF([PM_j]).g(\overrightarrow{u_i' u_j'})$.
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 [17]. 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 ($m \le |V^Q|$) 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 ($j \ne i$) that $[PM_i]$ and $[PM_j]$ are joinable;
2) $\forall 1 \le i \ne j \le m$, 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 are met, 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 ($i \le m$). 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. Only when all bits in $LECSign$ of the join result of some LEC features are '1', the corresponding local partial matches can join to form a SPARQL match.

Therefore, we can assemble all LEC Features and merge them together to prune some irrelevant local partial matches. 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 whether each pair of LEC features 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 $LF_i.LECSign = LF_j.LECSign$, 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, the LEC feature groups $\{P_1, P_2, P_3, P_4, P_5\}$ are as follows.

$P_1 = \{LF([PM_1^1]), LF([PM_1^2])\}$,
$P_2 = \{LF([PM_1^3])\}$, $P_3 = \{LF([PM_2^1]), LF([PM_3^1])\}$,
$P_4 = \{LF([PM_2^3])\}$, $P_5 = \{LF([PM_3^2])\}$. □

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 $\mathcal{P}$.
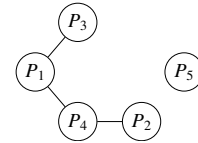

Fig. 6.   Join Graph

We propose an algorithm (Algorithm 2) based on the DFS traversal over the join graph to filter out the irrelevant LEC features. For example, $P_5 = LF([PM_3^2])$ in our example can be filtered out after we execute Algorithm 2.

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

1 $RS \leftarrow \emptyset$;
2 **while** $V^{JG} \ne \emptyset$ **do**
3   Find the vertex $v_{min} \in V^{JG}$ corresponding to LEC feature group P_{min}, where P_{min} has the smallest size;
4   Call Function **ComLECFJoin**($\{v_{min}\}, P_{min}, JG, RS$);
5   Remove v_{min} from V^{JG};
6   Remove all outliers remaining in JG;

Function ComLECFJoin(V, P, JG, RS)

---

1 **for** *each vertex v in JG adjacent to at least one vertex in V, where v corresponds to LEC feature group P'* **do**
2   Set $P'' \leftarrow \emptyset$;
3   **for** *each LEC feature LF_i in P* **do**
4   **for** *each LEC feature LF_j in P'* **do**
5   **if** *LF_i and LF_j are joinable* **then**
6 $LF_k \leftarrow LF_i \bowtie LF_j$;
7   **if** *all bits in $LF_k.LECSign$ are '1'* **then**
8   Insert all LEC features corresponding to vertices in V into RS;
9   **else**
10   Put LF_k into P'';
11   Call Function **ComParJoin**($V \cup \{v\}, P'', JG$);

D. Analysis

To analyse the complexity of the above optimization technique, we consider the communication cost as well as the computation costs. The communication cost is the data shipment needed in the above optimization technique, while 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 assemble the LEC features to prune the local partial matches. A general formula for determining the communication cost can be specified as $Cost_{LF} \times |\Psi|$, 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 a constant. The second component is the cost of the function g mapping the crossing edges 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 distributed RDF graph G, our optimization technique has the property that the communication cost of evaluating a query depends mainly on the size of the query and the partitioning 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* [3].

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 [17] 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. All local partial matches in the same partition maps to the internal vertices for a given variable. In [17], the authors prove that the local partial matches in the same partition cannot be joined.

The join space of the join algorithm in [17] is still large. As discussed before, we can determine whether two local partial matches in two different fragments can join according to their corresponding LEC features. Thus, we proposes an optimized technique based on the LEC features of the local partial matches to join the local partial matches.

The intuition of our method is that we divide all local partial matches into multiple groups based on their LEC features as proved in Theorem 5 such that two local partial matches in the same group cannot be joinable. Then, we only consider joining local partial matches from different groups.

Definition 11: **(LEC Feature-based Local Partial Match Group)** $\mathcal{G} = \{Gr_1, ..., Gr_n\}$ 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, after PM_3^2 is pruned during LEC feature-based optimization, the LECSign-based local partial match groups $\{Gr_1, Gr_2, Gr_3, Gr_4\}$ are as follows:

$Gr_1 = \{PM_1^1, PM_1^2\}, Gr_2 = \{PM_1^3\},$
$Gr_3 = \{PM_2^1, PM_2^2, PM_3^1\}, Gr_4 = \{PM_2^3\}$ □

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. Here, the join graph of $\mathcal{G}$ is the same to Fig. 6.
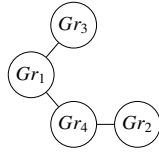
Fig. 7. Local Partial Match Group Join Graph

Then, we use the 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

1 **while** $V^{Gr} \neq \emptyset$ **do**
2   Find the vertex $v_{min} \in V^{Gr}$ corresponding to LEC feature-based local partial match group Gr_{min}, where Gr_{min} has the smallest size;
3   Call Function **ComParJoin**($\{v_{min}\}, Gr_{min}, LG, MS$);
4   Remove v_{min} from V^{Gr};
5   Remove all outliers remaining in LG;
6 Return false;

Function ComParJoin(V, Gr, LG, MS)

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

VI. 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 fragments often adopt the filter-and-evaluate framework. They first compute out the candidates of all variables, and then search matches over these candidates. The process of finding candidates is often very quick. Hence, we can modify the codes of these systems and assemble the internal candidates in the coordinator site. When the set of internal candidates for variable v (denoted as $C(Q, v)$) has been found, we do not find local partial matches directly but send the set of internal 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 they do not appear in any complete matches. To filter out these unnecessary candidates, the coordinator site can assemble and unions the candidates sets of a variable from all sites. If a candidate of variable v can appear in a complete match, it must belong to the v's internal candidate sets from all sites. Then, 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.

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 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. We can send the result bit vectors of all variables to different sites and filter out some false positive 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. Hence, it is beneficial for us to afford the cost of communicating the candidate bit vectors between the coordinator site and the sites.

Algorithm 4 describes the optimization of assembling variables' internal candidates. For the coordinator site, it receives and unions the bit vectors of candidates of all variables. Then, the coordinator site sends the result bit vectors of all variables to sites. For each site, it firstly finds out the candidates of variables locally and compresses them into bit vectors. It then sends all bit vectors to the coordinator site and 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 during computing the local partial matches.

VII. Impact of Partitioning Strategies

In this section, we analyze the impact of different partitioning strategies for our method.

According to the above analysis, the cost of our method are mainly dependent on the number of LEC features. The straightforward heuristic is to reduce the number of crossing edges. However, if we go deeply into the complexity of the cost, we can find that the small size of edge cut does not always result in small number of LEC features. For example, let us consider two example partitionings in Fig. 8. Although

Algorithm 4: Assembling Variables' Internal Candidates

Input: A distributed RDF graph G over sites $\{S_1, ..., S_m\}$, coordinator site S_c, and the SPARQL query Q.

Output: The internal bit vector B_v of any variable v.

1 **The Coordinator Site S_c:**
2 **for** *each variable v in Q* **do**
3 $B_v \leftarrow 0$;
4   **for** *each site S_i* **do**
5   Receive B'_v from S_i;
6 $B_v \leftarrow B_v \vee B'_v$;
7   **for** *each site S_i* **do**
8   Send B_v to S_i;
9 **The Site S_i:**
10 **for** *each variable v in Q* **do**
11   Find $C(Q, v)$ and $B'_v \leftarrow 0$;
12   **for** *each internal candidate c in $C(Q, v)$* **do**
13   Use a hash function h to map c to an integer $h(c)$;
14   Set the $h(c)$-th bit of B'_v to 1;
15   Send B_v to S_c;
16   Receive B_v to S_c;

---

the partitioning in Fig. 8(b) results in more crossing edges, its crossing edges are scattered to different boundary vertices. In contrast, all crossing edges in Fig. 8(a) are adjacent to one boundary vertex. When a star query Q of two edges as Fig. 8(c) is input, it maps to $\binom{4}{2} + \binom{4}{1} = 10$ LEC features for the partitioning in Fig. 8(a) and $\binom{3}{2} + \binom{3}{1} + \binom{2}{2} + \binom{2}{1} = 9$ LEC features for the partitioning in Fig. 8(b).
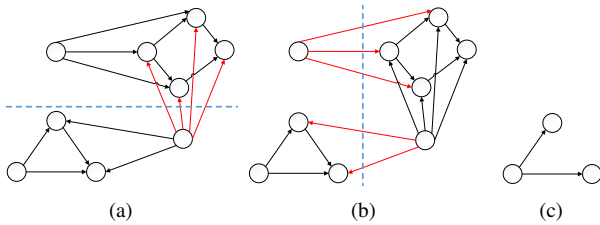

Fig. 8. Comparison of Different Partitionings

Based on the above observation, in a good partitioning for our method, the crossing edges need be scattered to as many vertices as possible. Given a partitioning $\mathcal{F} = \{F_1, F_2, ..., F_k\}$ and the set of its crossing edges E^c, we define the distribution of crossing edges over a vertex v, $p_{\mathcal{F}}(v)$, as follows.

$$p_{\mathcal{F}}(v) = |N(v) \cap E^c|/(2 \times |E^c|)$$

where $N(v)$ is the set of v's neighbors. Note that, an edge can be adjacent to two vertices, so the divisor in $p_{\mathcal{F}}(v)$ should be $2 \times |E^c|$, which can ensure that the sum of the distributions over all vertices is 1.

Then, the expectation of the number of crossing edge adjacent to a vertex v is as follows.

$$E_{\mathcal{F}}(v) = |N(v) \cap E^c| \times p_{\mathcal{F}}(v)$$

Then, the total expectation of the number of crossing edge distributed to all vertices is as follows.

$$E_{\mathcal{F}}(V) = \sum_{v \in V} E_{\mathcal{F}}(v)$$

To scatter the crossing edges to as many vertices as possible, the above expectation should be as small as possible.

In addition, when we partition the graph, we should also balance the sizes of all fragments. Thus, we should avoid generating a fragment with too many edges. Here, we use the edge number of the largest fragment to measure the balance of fragments. In summary, we combine the above two factors to define the cost of a partitioning as follows.

$$Cost_{Partitioning}(\mathcal{F}) = E_{\mathcal{F}}(V) \times \max_{1 \leq i \leq k} |E_i \cup E_i^c|$$

Here, the more sophisticated partitioning strategy is beyond the scope of this study. We only select the partitioning with the smallest cost from the existing partitioning strategies. For example, the cost of the partitioning in Fig. 8(a) is 27.5 and the cost of the partitioning in Fig. 8(b) is 23.4. Hence, the partitioning in Fig. 8(b) is better to be selected.

VIII. Experiments

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

A. Setting

LUBM. LUBM [5] is a benchmark that adopts an ontology for the university domain, and can generate RDF data scalable to an arbitrary size. We generate three datasets of triples from 100 million to 1 billion, whose sizes vary from 15 GB to 150 GB. The dataset of 100 million triples is denoted as LUBM 100M, the one of 500 million triples is LUBM 500M and the one of 1 billion triples is LUBM 1B. We use the 7 benchmark queries in [1] (denoted as $LQ_1 - LQ_7$) to test our methods.

YAGO2. YAGO2 [10] 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 of 44 GB. We use the benchmark queries in [1] (denoted as $YQ_1 - YQ_4$) to evaluate our methods.

BTC. BTC[2] is a real dataset used for the Billion Triples Track of the Semantic Web Challenge, which contains about 1 billion triples of 176 GB. We use the 7 queries (denoted as $BQ_1 - BQ_7$) in [17] 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 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++ for communication. By default, we use a hash function $H(v)$ to assign each

---

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

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)						
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

√ means that the query involves some selective triple patterns.

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)						
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 III
EVALUATION OF EACH STAGE ON BTC

		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

vertex v in RDF graph to the i-th fragment if $H(v)\ MOD\ N = i$, where $N = 12$ is the number of machines. Each machine stores a single fragment.

In this paper, we revise gStore [23] to find local partial matches at each site. We denote our method as gStore^D. We compare our approach with four other systems, including DREAM [7], S2X [18], S2RDF [19] and CliqueSquare [4]. 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, the query performance mainly depends on two 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 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. 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 we can directly compute out the results over each fragment without considering communications and our optimization techniques. In contrast, queries of other query shapes involve multiple fragments and generate local partial matches, which increase the search space of partial evaluation and the assembly process. Thus, evaluating them 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.

C. Evaluation of Different Optimizations

This experiment is to use LUBM 100M and YAGO2 to test the effect of the three optimization techniques proposed in this paper. Here, because star queries can be evaluated without involving any optimization techniques, we only consider the benchmark queries of other shapes (LQ_1, LQ_3, LQ_6 and LQ_7 in LUBM and all queries in YAGO2). We design a baseline that does not utilize any proposed optimization techniques (denoted as $gStore^D$-Basic), 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. 9 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.
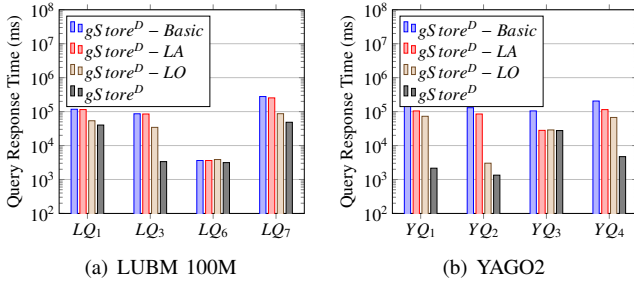

(a) LUBM 100M   (b) YAGO2

Fig. 9. Evaluation of Different Optimizations

D. Evaluation of Different Partitioning Strategies

The aim of this experiment is to highlight the differences among different partitioning strategies. In this experiment, we also use LUBM 100M and YAGO2 and test three partitioning strategies, hash partitioning, semantic hash partitioning [14] and METIS [13]. Here, we also only consider the benchmark queries of other shapes. Tables IV shows the costs of different partitionings defined in Section VII, while Fig. 10 show the evaluation times of our method over different partitionings.

The hash partitioning can uniformly distribute vertices and crossing edges among different fragments. Hence, the cost of the hash partitioning is not too bad. The semantic hash partitioning is based on the URI hierarchy. For LUBM 100M,

because different entities have different URI hierarchies, the semantic hash partitioning can partition the entities totally based on their domains, which greatly reduces its partitioning cost. In contrast, all entities in YAGO2 have the same URI hierarchy and the cost of the semantic hash partitioning is approximately same to the hash partitioning. Hence, the performance of our method over LUBM 100M in the semantic hash partitioning is better than other partitionings while the performance over YAGO2 is similar. In addition, although there are fewer crossing edges in METIS, its partitioning result is much more imbalanced than the hash partitioning, which indicates that the cost of METIS is high. This results in that the performance in METIS is always not better than the hash partitioning for YAGO2.

TABLE IV
$Cost_{Partitioning}$

	Hash	Semantic Hash	METIS
YAGO2	0.76×10^{14}	0.77×10^{14}	1.49×10^{14}
LUBM 100M	0.92×10^9	0.55×10^9	0.67×10^9

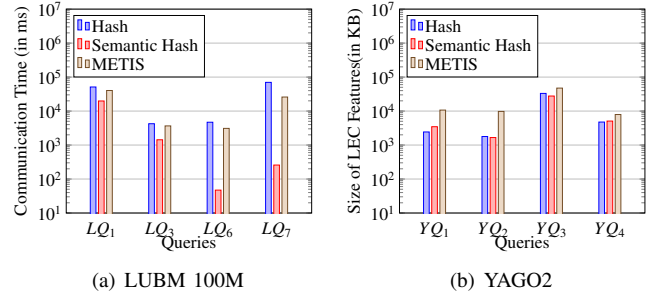(a) LUBM 100M   (b) YAGO2

Fig. 10. Evaluation Time of Different Partitioning Strategies

E. Scalability Test

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. 11 shows the experiment results. As mentioned in Section VIII-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).

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.

F. Online Performance Comparison

In this experiment, we evaluate the online performance of our method on different partitionings of three datasets, YAGO2, LUBM 1B and BTC. Fig. 12 shows the performances.
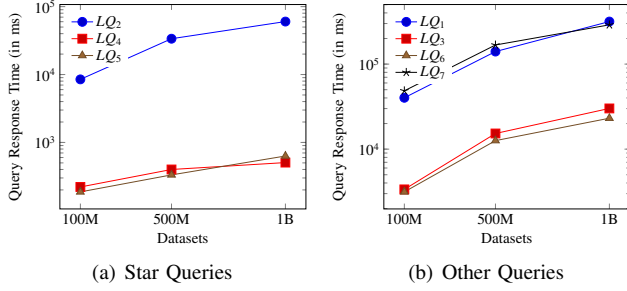
Fig. 11.   Scalability Test

Note that, METIS can only be used to YAGO2, and fails to partition LUBM 1B and BTC in our setting.

This experiment results include a comparative evaluation of our method against four state-of-the-art public disk-based distributed RDF systems proposed in recent three years, including DREAM [7], S2X [18], S2RDF [19] and CliqueSquare [4], which are provided by [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, S2X fails to run all queries on LUBM 1B. We also run DREAM and CliqueSquare over BTC, while S2X and S2RDF fails over BTC.

Generally, our method is partitioning-tolerant and the performances of our method over different partitionings show the superiority of our proposed approach.

In particular, S2X, S2RDF and CliqueSquare are three cloud-based systems, 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 may have better performances than DREAM and our approach running over ill-suited partitionings. However, when our method runs over partitionings with the smallest costs (hash partitioning for YAGO2 and semantic hash partitioning for LUBM 1B and BTC), our method can outperforms others.

On the other hand, when the queries (LQ_3, LQ_4, LQ_5 and LQ_6 in LUBM 1B and all queries in BTC) are selective or the RDF dataset (YAGO2) is not very large, DREAM [7] and our system can outperform the cloud-based systems in most cases. Here, DREAM builds a single RDF-3X database for the entire dataset in each site, and decomposes the input query into multiple star-shape subqueries where each subquery is answered by a single site. This can greatly reduce the performances over the selective queries and small datasets. However, DREAM exhibits excessive replication and causes huge overhead when processing complex queries. When a query is complex, it may lead to multiple large subqueries. Evaluating the large subqueries over a site of the entire dataset often results in many intermediate results, and joining these intermediate results is also costly. Our method running over the best partitionings can always be not much worse than DREAM. Note that, DREAM fails to process YQ_2.

IX. Related Work

Distributed SPARQL Query Processing. There have been many works on distributed SPARQL query processing, and a very good survey is [12]. In recent years, some approaches [4], [22], [21], [8], [9], [16], [7], [19] are proposed. We classify them into three classes: cloud-based approaches, partitioning-based approaches, and partitioning-tolerant approaches.

First, some recent works (e.g., [4], [19], [18]) focus on managing RDF datasets using cloud platforms. CliqueSquare [4] discuss how to build query plans relying on n-ary (star) equality joins in Hadoop. S2RDF [19] uses Spark SQL to store the RDF data in the vertical partitioning schema and materializes some extra join relations. In the online phase, S2RDF transforms the query into SQL queries and merges the results of the SQL queries. S2X [18] uses GraphX in Spark to evaluate SPARQL queries. S2X first distributes all triple patterns to all vertices. Then, vertices validate their triple candidacy with their neighbours by exchanging messages. Last, the partial results are collected and merged.

Second, some approaches [22], [21], [8], [9], [16] are partition-based. They divide an RDF graph into several partitions. Each partition is placed at a site which installs a centralized RDF system to manage it. At run time, a SPARQL query is decomposed into several subqueries that can be answered locally at a 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 [22] 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 [21] adopts the end-to-end path as the partition unit to partition the data and query graph; AdHash [8] and AdPart [9] directly use the subject values to partition the RDF graph and mainly discuss how to reduce the communication cost during distributed query evaluation; Peng et al. [16] mine some frequent patterns in the log as the partitioning units.

DREAM [7] and Peng et al. [17] are two other approaches that do not neither partition RDF graphs nor use existing cloud platforms. In DREAM [7], each site maintains the whole RDF dataset. For query processing, DREAM divides the input query into subqueries and executes each subquery in a site. The intermediate results are merged to produce the final matches. Peng et al. [17] propose a partition-tolerant distributed approach based on the "partial evaluation and assembly" framework. However, its efficiency has large potential to improve.

Partial Evaluation. As surveyed in [11], partial evaluation has found many applications ranging from compiler optimization to distributed evaluation of functional programming languages. Recently, partial evaluation has been used for evaluating queries on distributed graphs [15], [2], [3], [6], [20]. In [2], [6], the authors provide algorithms for evaluating reachability queries on distributed graphs based on partial evaluation. In [15], [3], the authors study partial evaluation algorithms and optimizations for distributed graph simulation.
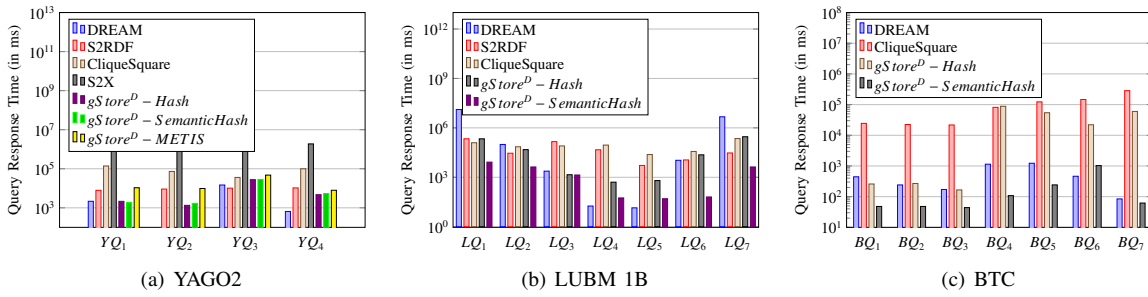
(a) YAGO2	(b) LUBM 1B	(c) BTC

Fig. 12. Online Performance Comparison

Peng et al. [17] discuss how to employ the "partial evaluation and assembly" framework to handle SPARQL queries, while Wang et al. [20] 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.

X. 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 reduce the search space. Moreover, we propose an optimization that communicate variables' candidates among the sites to avoid irrelevant local partial matches. We also discuss the impact of different partitioning over our approach. 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] W. Fan, X. Wang, and Y. Wu. Performance Guarantees for Distributed Reachability Queries. *PVLDB*, 5(11):1304–1315, 2012.

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

[4] 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.

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. In *ICPP*, pages 113–122, 1995.

[14] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*, 6(14):1894–1905, 2013.

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

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

[17] 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.

[18] 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.

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

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

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

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

[23] 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.