ABSTRACT

We propose techniques for processing SPARQL queries over linked data. We follow a graph-based approach where answering a query \( Q \) is equivalent to finding its matches over a distributed RDF data graph \( G \). We adopt a “partial evaluation and assembly” framework. Partial evaluation results of query \( Q \) over each repository—called local partial match—are found. In the assembly stage, we propose a centralized and a distributed assembly strategy. We analyze our algorithms both theoretically and the experimentally. Extensive experiments over both real and benchmark RDF repositories with billion triples demonstrate the high performance and scalability of our methods compared with that of the existing solutions.

1. INTRODUCTION

The semantic web data model, called the “Resource Description Framework”, or RDF, represents data as a collection of triples of the form (subject, property, object). A triple can be naturally seen as a pair of entities connected by a named relationship or an entity associated with a named attribute value. Linked Open Data (LOD), which is the focus of this paper, uses RDF to link web objects through their Uniform Resource Identifiers. LOD forms a directed graph where each vertex represents a Web document that contains data in the form of a set of RDF triples. These documents can be retrieved by looking up HTTP-scheme based URIs. A pair of document vertices is connected by a directed edge if any of the RDF triples in the source document contains a URI in its subject, predicate, or object position such that a lookup of this URI results in retrieving the target document. LOD forms a federated (or virtually integrated) database with the documents stored at autonomous sites some of which have the ability to run SPARQL queries over their own data (these are called “SPARQL endpoints”).

In this paper we address the issue of executing SPARQL queries over this type of federated RDF repositories. The approach we take to evaluating SPARQL queries is graph-based, which is consistent with RDF and SPARQL specifications. Consequently, answering a SPARQL query \( Q \) is equivalent to finding subgraph matches of query graph \( Q \) over an RDF graph \( G \), as employed, for example, in the gStore system [36]. We note that the problem in this context is different than graph subgraph matching in general graph processing systems since these depend on graph isomorphism while SPARQL query semantics are based on graph homomorphism.

There are several ways to address this issue [14]. One method is analogous to data warehousing where all of the data is downloaded into a single RDF database and queries are run over it using centralized techniques, such as Jena [32], RDF-3x [24], SW-store [1], and gStore [36]. This is a reasonable approach, but at web scale, the feasibility of these approaches is a serious concern as is the freshness of query results.

A second approach that favors query result freshness is live querying [15]. In this approach, one starts with a set of seed LOD documents and follows the RDF links inside these until no further documents are reachable. These approaches provide up-to-date results, but with high latency that is hard to optimize.

The third alternative is to treat the interconnected RDF repositories as a virtually integrated distributed database, where each individual RDF repository is regarded as a fragment of this database. Each fragment \( F_i \) resides at its own site \( S_i \) and each query is executed at the relevant sites. Within this context, there are two approaches. One is to decompose queries and evaluate each subquery at the appropriate sites. The query decomposition strategy requires having full knowledge about the underlying schema of each fragment \( F_i \), which is not always feasible in RDF datasets since their fundamental characteristic is that they are self-describing without requiring an explicit schema. This is the approach followed by most existing works [18] [25] [27]. For example, DARQ [27] needs the service description that describes which triple patterns can be answered in each site, based on which, the SPARQL query is decomposed into several subqueries. The alternative approach is to use the “partial evaluation and assembly” strategy [17]. The basic idea of the partial evaluation is the following: given a function \( f(s, d) \), where \( s \) is the known input and \( d \) is yet unavailable input, the part of \( f \)'s computation that depends only on \( s \) generates a partial answer [8]. In our setting, each site \( S_i \) treats fragment \( F_i \) as the known input in the partial evaluation stage. The partial evaluation technique has been used in compiler optimization [17], querying XML trees [4], and evaluating reachability queries [8] and graph simulation [23] [9] over graphs. In this paper, we follow the partial evaluation and assembly strategy to evaluate SPARQL queries over a distributed and interconnected set of RDF repositories.

Because of interconnections between datasets, processing of queries requires special care. For example, consider the four RDF datasets at different sites in LOD shown in Figure 1 and its graph representation in Figure 2. Each entity in RDF is represented by a URI (uniform resource identifier), the prefix of which always denotes the location of the dataset. For example, “s1:dir1” has the prefix “s1”, meaning that the entity is located at site s1. There are cross-
ing links between two datasets identified in bold font. For example, "(s2:act1 isMarriedTo s1:dir1)" is a crossing link (links between different datasets), which means that act1 (at site s2) is married to dir1 (at site s1).

Now consider the following SPARQL query Q that consists of five triple patterns (e.g., ?a isMarriedTo ?d) over these four interconnected RDF repositories:

\[
\text{SELECT} \ ?a \ ?d \ \text{WHERE} \ \\
\{ ?a \text{ isMarriedTo } ?d. \ ?a \text{ actedIn } ?f1. \ ?f1 \text{ rdf:s:label } ?n1. \ ?d \text{ directed } ?f2. \ ?f2 \text{ rdf:s:label } ?n2. \}
\]

Some of the triple patterns may be resolved completely within a fragment, which we call an inner match or local partial match. These inner matches can be found locally by existing centralized techniques at each site. However, if we consider the four datasets independently and ignore the crossing links, some correct answers will be missed, such as \((?a=s2:act1, ?d=s1:dir1)\). The key issue in the distributed environment is how to find subgraph matches that cross multiple fragments—these are called crossing matches. For the query \(Q\) in Figure 3, the subgraph induced by vertices 014, 007, 001, 002, 009 and 018 is a crossing match between fragments \(F_1\) and \(F_2\) in Figure 2 (shown in the shaded vertices and red edges). This is the focus of this paper.

There are two important issues to be addressed in this framework. The first is to compute the partial evaluation results at each site given a query graph \(Q\) (i.e., the local partial match), which is intuitively the overlapping part between a crossing match and a fragment. This is discussed in Section 4. The second one is the assembly of these local partial matches to compute crossing matches. We consider two different strategies: centralized assembly, where all local partial matches are sent to a single site (Section 5.2); and distributed assembly, where the local partial matches are assembled at a number of sites in parallel (Section 5.3).

This is the first work that considers the LOD repositories (or similarly distributed and autonomous RDF repositories) as a virtually integrated distributed database, and proposes the partial evaluation and assembly framework for executing SPARQL queries. Within this context we make the following contributions.

1. We formally define the partial evaluation result for SPARQL queries and propose an efficient algorithm to find them. We prove theoretically the correctness and the optimality of our algorithm.
2. We propose centralized and distributed methods to assemble the local partial matches and experimentally study their performance advantages and bottlenecks in different settings.

2. RELATED WORK

There are two threads of related work: distributed SPARQL query processing and partial evaluation.

**Distributed SPARQL Query Processing.** As noted above, generally speaking, there are two different approaches to distributed SPARQL query processing: federated SPARQL query processing, and data partition-based query processing.

1. Federated SPARQL Query Processing

Federated queries run SPARQL queries over multiple distributed and autonomous RDF datasets. A typical example is linked data, where different RDF repositories are interconnected, providing a virtually integrated distributed database.
A common technique is to precompute metadata for each individual RDF repository. Based on the metadata, the original SPARQL query is decomposed into several subqueries, where each subquery is sent to some RDF repositories that could answer the query. These subquery results are then joined together to answer the original SPARQL query. In DARQ [27], the metadata is called service description that describes which triple patterns (i.e., predicate) can be answered. Consider each triple pattern \( t \) in a SPARQL query \( Q \). If the triple pattern \( t \) matches exactly one data source, the triple will be added to the set of a subquery for this RDF data source. If \( t \) matches multiple data sources, the triple must be sent to all matching RDF data sources as separate subqueries. In [25, 12], the metadata is called Q-Tree, based on which, the SPARQL is also decomposed into several parts.

FedX [28] follows the same approach, but it does not require preprocessing. However, for each triple pattern \( t \) in a SPARQL query \( Q \), FedX sends “SPARQL ASK” to collect the metadata on the fly. Based on the returned results, it annotates each triple pattern with its relevant sources. Each triple pattern is a basic query unit. In order to improve the query performance, several triple patterns in the same exclusive groups [28] can be sent in a single subquery to the respective source.

The problem studied in this paper belongs to federated SPARQL query approach, but, we adopt a different strategy, namely partial evaluation and assembly. Rather than decomposing a query, we send the whole SPARQL query to each RDF data source. The partial evaluation results from each RDF data source are merged together. The main benefits of our solution are twofold: (1) The number of remote requests is minimized. For example, for each triple pattern \( t \) in SPARQL \( Q \), FedX sends “SPARQL ASK” to each RDF data source. For each data source, the number of remote requests is \(|E(Q)|\) (the number of edges in SPARQL query graph \( Q \)). However, we only send a single SPARQL query to each RDF data source. (2) The intermediate result size is reduced. We propose several graph structure-based pruning techniques to reduce the intermediate result size. Experimental results show the our approach outperforms the existing ones. We note that our approach requires minor modifications to SPARQL endpoints to facilitate partial query evaluation.

(2) Data Partition-based Method

A number of approaches [20, 15, 16, 25, 24, 21, 22, 11] start with a large RDF graph \( G \) (i.e., follow what we referred to as the data warehousing approach), and partition \( G \) into several fragments. At run time, a SPARQL query \( Q \) is decomposed into several subqueries such that each subquery can be answered locally at one site. Note that each of these works proposes its own data partitioning strategy, and different partitioning strategies result in different processing methods. Generally speaking, there are three major partitioning strategies: table-based, graph-based, and unit-based.


Graph-based methods [15, 16, 33] partition the RDF graph using graph partitioning techniques. In GraphPartition [15], an RDF graph \( G \) is partitioned into \( n \) fragments, and each fragment is extended by including \( N \)-hop neighbors of boundary vertices. According to the partitioning strategy, the diameter of the graph corresponding to each decomposed subquery should not be larger than \( N \) to enable subquery processing at each local site. In EAGRE [15], the graph fragmentation is based on the predicates adjacent to the entities. If two entities’ incident predicates are similar, they are in one class and should be in the same fragment. At run time, query graph \( Q \)'s decomposition always depends on the predicates incident to each variable in \( Q \). Trinity.RDF [33] stores the whole RDF graph in a distributed graph system over a memory cloud (called Trinity [30]). Trinity.RDF uses the vertex in the RDF graph as the key and its adjacency list as the value. Since Trinity is a main memory system, it can randomly access all neighbors of a vertex. Hence, Trinity.RDF uses a graph exploration-based method to answer SPARQL queries.

Unit-based methods [21, 22, 11] decompose the RDF data into some partition units, and distribute these units among different sites. Kising Lee et al. [21, 22] define the partition unit as a vertex and its neighbors, which they call a “vertex block”. The vertex blocks are distributed based on a set of heuristic rules. A query is partitioned into blocks that can be executed among all sites in parallel and without any communication. TriAD [11] distributes RDF-3x [24]. TriAD uses METIS [19] to divide the RDF graph into many partitions and the number of result partitions is much more than the number of sites. Each result partition is considered as a unit and distributed among different sites. Meanwhile, TriAD constructs a summary graph to maintain the partitioning information. During the query processing phase, TriAD determines a query plan based on this summary graph.

All of the above methods require partitioning and distributing the RDF data according to specific requirements of their approaches. In contrast, in LOD, the RDF are already distributed over different sites. Therefore, these data partition-based solutions cannot be used for processing queries over LOD unless LOD are collected in one central site and redistributed again, which is impractical.

Partial Evaluation. Partial evaluation has been used in many applications ranging from compiler optimization to distributed evaluation of functional programming languages [17]. Recently, partial evaluation has also been used for evaluating queries on distributed XML trees and graphs [8, 4, 5, 9]. In [8, 4, 5], partial evaluation is used to evaluate some XPath queries on distributed XML. These works serialize XPath queries to a vector of subqueries, and find the partial results of all subqueries at each site by using a top-down [4] or bottom-up [3] traversal over the XML tree. Finally, all partial results are assembled together at the server site to form final results. Note that, since XML is a tree-based data structure, these works serialize XPath queries and traverse XML trees in a topological order. However, the RDF data and SPARQL queries are graphs rather than trees. Serializing the SPARQL queries and traversing the RDF graph in a topological order is not intuitive.

There are some prior works that consider partial evaluation on graphs. For example, Fan et al. [8] study reachability query processing over distributed graphs using the partial evaluation strategy. Partial evaluation-based graph simulation is well studied in Fan et al. [9] and Shuai et al. [23]. However, SPARQL query semantics is different than these (i.e., SPARQL is based on graph homomorphism [25]) and pose additional challenges. To the best of our knowledge, there is no prior work in applying partial evaluation to SPARQL query processing. As discussed in [9], graph simulation defines a relation between vertices in query graph \( Q \) (i.e., \( V(Q) \)) and that in the data graph \( G \) (i.e., \( V(G) \)), but, the graph homomorphism is a function (not a relation) between \( V(Q) \) and \( V(G) \). Thus, the solutions proposed in [9, 25] cannot be applied to the problem studied in this paper.

3. BACKGROUND
An RDF data set can be represented as a graph where subjects and objects are vertices and triples are labeled edges.

**Definition 3.1. (RDF Graph)** An RDF graph is denoted as \( G = (V, E, \Sigma) \), where \( V \) is a set of vertices that correspond to all subjects and objects in RDF data; \( E \subseteq V \times V \) is a set of directed edges that correspond to all triples in RDF data; \( \Sigma \) is a set of edge labels. For each edge \( e \in E \), its edge label is its corresponding property.

Similarly, a SPARQL query can also be represented as a query graph \( Q \). For simplicity, we ignore FILTER statements in SPARQL syntax and only consider BGP (basic graph pattern) in this paper.

**Definition 3.2. (SPARQL Query)** A SPARQL query is denoted as \( Q = (V^0, E^0, \Sigma^0) \), where \( V^0 \subseteq V \cup V_{var} \) is a set of vertices, where \( V \) denotes all vertices in RDF graph \( G \) and \( V_{var} \) is a set of variables; \( E^0 \subseteq V^0 \times V^0 \) is a set of edges in \( Q \). Each edge \( e \) in \( E^0 \) either has an edge label in \( \Sigma \) (i.e., property) or the edge label is a variable.

In this paper, 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 subgraph matches (Definition 3.3) of \( Q \) over RDF graph \( G \).

**Definition 3.3. (SPARQL Match)** Consider a RDF graph \( G \) and a connected \( Q \) 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 exist a function \( f \) from \( \{v_1, ..., v_n\} \) to \( \{u_1, ..., u_m\} \) \((m \geq n)\), 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 \( G \); furthermore, \( f(v_i)f(v_j) \) has the same property as \( v_i v_j \) unless that the label of \( v_i v_j \) is a variable.

Vector \( [f(v_1), ..., f(v_n)] \) is a serialization of a SPARQL match. Note that we allow that \( f(v_i) = f(v_j) \) when \( 1 \leq i \neq j \leq n \). In other words, a match of SPARQL \( Q \) defines a graph homomorphism.

In the context of this paper, RDF graph \( G \) is distributed over different sites. We call \( G \) an integrated distributed RDF graph. Each constituent RDF repository is called a fragment of \( G \). For simplicity, we assume that each entity is only resident in a single site, i.e., we ignore replication (see condition (1) of Definition 3.4), but handling the more general case is not complicated.

**Definition 3.4. (Integrated Distributed RDF Graph)** Given \( k \) interconnected RDF repositories, each of them as represented as an RDF graph \( G_i \) \((i = 1, ..., k) \), the integrated distributed RDF graph \( G = (V, E, \Sigma) \) consists of a set of \( \{F_1, F_2, ..., F_k\} \). Each \( F_i \) is called a fragment of \( G \), which is specified by \( (V_i, E_i, \Sigma_i) \) \((i = 1, ..., k) \) such that

1. \( \{V_1, ..., V_k\} \) is a partition 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. \( E_i \subseteq V_i \times V_i, i = 1, ..., k \);

1In this paper, we assume that query graph \( Q \) is connected; otherwise, we can process each connected connected component of \( Q \) separately.

3. \( E_i \) is a set of crossing edges between \( F_i \) and other fragments, i.e.,
   \[ E_i = (\bigcup_{j \neq i} (E_j \cap V_i \times V_i)) \cup \bigcup_{j \neq i} (E_j \cap V_i \times V_j) \ ];
4. A vertex \( v' \in V_i \) if and only if vertex \( v' \) 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., \( V_i = (\bigcup_{j \neq i} (E_j \cap V_i \times V_i)) \cup \bigcup_{j \neq i} (E_j \cap V_i \times V_j) \);
5. Vertices in \( V_i \) are called external vertices of \( F_i \); and all vertices in \( V_i \) are called internal vertices of \( F_i \);
6. \( \Sigma \) is a set of edge labels in \( F_i \).

**Example 1.** Given four interconnected RDF repositories in Figure 7 the integrated RDF graph \( G \) consists of four constituent RDF fragment graphs \( F_1, F_2, F_3 \) and \( F_4 \) in Figure 2. The numbers besides the vertices are vertex IDs that are introduced for ease of presentation. In Figure 2, \( \{002, 001\} \) is a crossing edge between \( F_1 \) and \( F_2 \). As well, edges \( 004, 011, 001, 012 \) and \( 006, 008 \) are crossing edges between \( F_1 \) and \( F_3 \). Hence, \( V_1' = \{002, 006, 012, 004\} \) and \( E_1' = \{002, 001, 004, 011, 001, 012, 006, 008\} \).

**Definition 3.5. (Problem Statement)** Given an integrated distributed RDF graph \( G \) that consists of \( k \) fragments \( \{F_1, F_2, ..., F_k\} \), each located at a different computing node, and a SPARQL query graph \( Q \), our goal is to find all SPARQL matches of \( Q \) in \( G \).

Inner matches can be computed locally by a centralized RDF triple store, such as RDF-3x [24], SW-store [1] and gStore [36]. The main issue of answering SPARQL queries over the distributed RDF graph is finding crossing matches efficiently. That is the focus of this paper.

**4. PARTIAL EVALUATION**

We first discuss what is a local partial match (Section 4.1) and then discuss how to compute it efficiently (Section 4.2).

**4.1 Local Partial Match—Definition**

Recall that each site \( S_i \) receives the full query graph \( Q \) (i.e., there is no query decomposition). In order to answer a SPARQL query \( Q \), each site \( S_i \) computes the partial answers (called local partial matches) based on the known input \( F_i \). Intuitively, a local partial match \( P M_i \) (in fragment \( F_i \)) is an overlapping part between an unknown crossing match \( M \) and fragment \( F_i \). An unknown crossing match \( M \) means that we do not know \( M \) at the partial evaluation stage. Moreover, the unknown crossing match \( M \) may or may not exist depending on the yet unavailable input \( d \), i.e., other fragments of \( G \). Based on the known input, i.e., fragment \( F_i \), we cannot judge whether or not \( M \) exists. For example, the subgraph induced by vertices \( 014, 007, 001 \) and \( 002 \) (shown in shared vertices and red edges) in Figure 3 is a local partial match between \( M \) and \( F_1 \).

**Definition 4.1. (Local Partial Match)** Given a SPARQL query graph \( Q \) with \( n \) vertices \( \{v_1, ..., v_n\} \) and a connected subgraph \( PM \) with \( m \) vertices \( \{u_1, ..., u_m\} \) \((m \leq n) \) in a fragment \( F_i = (V_i, E_i, \Sigma_i) \), \( PM \) is a local partial match in fragment \( F_i \) if and only if there exists a function \( f : \{v_1, ..., v_n\} \rightarrow \{u_1, ..., u_m\} \cup \{NULL\} \), 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, \( f(v_i) \in \{u_1, ..., u_m\} \) or \( f(v_i) = NULL \).
3. If there exists an edge \( v_i \rightarrow v_j \) in \( Q \) (1 \( \leq i \neq j \leq n \)), there also exists an edge \( f(v_i) \rightarrow f(v_j) \) in \( PM \) with property \( p \) unless \( f(v_i) \) and \( f(v_j) \) are both in \( V_i \) or \( f(v_i) = NULL \) or \( f(v_j) = NULL \). Furthermore, \( f(v_i) \rightarrow f(v_j) \) has the same property as \( v_i \rightarrow v_j \) unless the label of \( v_i \rightarrow v_j \) is a variable.

4. \( PM \) contains at least one crossing edge.

5. If \( f(v_i) \in V_i \) (i.e., \( f(v_i) \) is an internal vertex in \( F_i \)) and \( \exists v_j \in Q \) (or \( v_j \in Q \)), there must exist \( f(v_i) \neq NULL \) and \( \exists f(v_i)f(v_j) \in PM \) (or \( \exists f(v_i)f(v_j) \in PM \)). Furthermore, if \( v_i \rightarrow v_j \) (or \( v_j \rightarrow v_i \)) has a property \( p \), \( f(v_i)f(v_j) \) (or \( f(v_j)f(v_i) \)) has the same property \( p \).

6. Any two vertices \( v_i \) and \( v_j \) in \( Q \), where \( f(v_i) \) and \( f(v_j) \) are both internal vertices in \( PM \), are weakly connected (see Definition 4.2) in \( Q \).

**Example 2.** Given a SPARQL query with six vertices in Figure 3, the subgraph induced by vertices 002, 001, 007 and 014 (shown in shaded circles and red edges) is a local partial match of \( Q \) in fragment \( F_1 \). The function is \( (v_1, \{002, 001, 007, 014\}) \). Actually, there are five different local partial matches in \( F_1 \). We show them in Figure 4.

**Definition 4.1** formally defines a local partial match, which is a subset of a complete SPARQL match. Therefore, some conditions in Definition 4.1 are analogous to SPARQL match with some slight differences. In Definition 4.2, some vertices of \( Q \) are not matched in a local partial match. They are allowed to match a special value NULL. For example, \( v_i \) and \( v_j \) in Example 2 are matched to NULL. As mentioned earlier, a local partial match is the overlapping part of an unknown crossing match and a fragment \( F_i \). Therefore, it must have a crossing edge, i.e., Condition 4.

The basic intuition of Condition 5 is that if vertex \( v_i \) in \( Q \) is matched to an internal vertex, all \( v_i \)'s neighbors should be matched in this local partial match as well. The following example illustrates the intuition.

**Example 3.** Let us recall the local partial match \( PM_i^1 \) of Fragment \( F_i \) in Figure 4. An internal vertex 001 in fragment \( F_i \) is matched to vertex \( v_2 \) in \( Q \). Assume that \( PM \) is an overlapping part between a crossing match \( M \) and fragment \( F_i \). Obviously, \( v_2 \)'s neighbors, such as \( v_1 \) and \( v_3 \), should also be matched in \( M \). Furthermore, the matching vertices should be 001's neighbors. Since 001 is an internal vertex in \( F_i \), 001's neighbors are also in fragment \( F_i \).

According to the above analysis, if a subgraph \( PM \) violates Condition 5, \( PM \) cannot be a subgraph of a crossing match. In other words, we are not interested in these subgraphs when finding local partial matches, since they do not contribute to any crossing match.

**Definition 4.2.** Two vertices are weakly connected in a directed graph if and only if there exists a connected path between the two vertices if we replace all directed edges with undirected edges. The path is called a weakly connected path between the two vertices.

Condition 6 will be used to prove the correctness of our algorithm in Theorems 4.1 and 4.2. We will discuss them soon. The following example shows all local partial matches in the running example.

**Example 4.** Given a query \( Q \) in Figure 3 and an RDF graph \( G \) in Figure 2. Figure 2 shows all local partial matches and their serialization vectors in each fragment. A local partial match in fragment \( F_i \) is denoted as \( PM_i \), where the superscript distinguishes different local partial matches in the same fragment. Furthermore, we underline all extended vertices in serialization vectors.

The correctness of our method are stated in the following theorem.

1. The overlapping part between any crossing match \( M \) and fragment \( F_i \) must be a local partial match.

2. Missing any local partial match may lead to result dismissal. Thus, our algorithm should find all local partial matches in each fragment.

3. It is impossible to find two local partial matches \( M \) and \( M' \) in fragment \( F_i \), where \( M' \) is a subgraph of \( M \), i.e., each local partial match is maximal.

**Theorem 4.1.** Given any crossing match \( M \) of SPARQL query \( Q \) in an RDF graph \( G \), if \( M \) overlaps with some fragment \( F_i \), the overlapping part between \( M \) and fragment \( F_i \), denoted as \( PM \), must be a local partial match in fragment \( F_i \).

Let us recall Example 4. There are some local partial matches that do not contribute to any crossing match, such as \( PM_i \) in Figure 4. We call these local partial matches false positives. However, the partial evaluation stage only depends on the known input. If we do not know the structures of other fragments, we cannot judge whether or not \( PM_i \) is a false positive. Formally, we have the following theorem, which means that we have to find all local partial matches in each fragment \( F_i \) in the partial evaluation stage.

**Theorem 4.2.** The results generated by the partial-evaluation-and-assembly algorithm satisfy no-false-negative requirement if and only if all local partial matches in each fragment are found in the partial evaluation stage.

Finally, we discuss another feature of a local partial match \( PM \), in fragment \( F_i \). Any \( PM \) cannot be enlarged by introducing more vertices or edges to become a larger local partial match. The following theorem formalizes this.

**Theorem 4.3.** Given a query graph \( Q \) and a RDF graph \( G \), if \( PM_i \) is a local partial match under function \( f \) in fragment \( F_i = (V_i \cup V'_i, E_i \cup E'_i) \), there exists no local partial match \( PM'_i \) under function \( f' \) in \( F_i \), where \( f' \subset f \).

### 4.2 Computing Local Partial Matches

Given a SPARQL query \( Q = (V^0, E^0) \) and a fragment \( F_i = (V_i \cup V'_i, E_i \cup E'_i) \), the partial evaluation goal is to find all local partial matches (according to Definition 4.1) in \( F_i \). The matching process consists of determining a function \( f \) that associates vertices of \( Q \) with vertices of \( F_i \). The matches are expressed as a set of pairs \((v, u) (v \in Q \text{ and } u \in F_i)\). A pair \((v, u)\) represents the matching of a vertex \( v \) of query \( Q \) with a vertex \( u \) of fragment \( F_i \). The set of vertex pairs \((v, u)\) constitutes function \( f \) referred to in Definition 4.1.

A high-level description of finding local partial matches is outlined in Algorithm 1 and Function ComParMatch. According to Conditions 1 and 2 of Definition 4.1, each vertex \( v \) in query graph \( Q \) has a candidate list of vertices in fragment \( F_i \). Since function \( f \) is as a set of vertex pairs \((v, u) (v \in Q \text{ and } u \in F_i)\), we start with an empty set. In each step, we introduce a candidate vertex pair \((v, u) (v \in Q \text{ and } u \in F_i)\).
to expand the current function $f$, where vertex $u$ (in fragment $F_i$) is a candidate of vertex $v$ (in query $Q$).

Assume that we introduce a new candidate vertex pair $(v', u')$ into the current function $f$ to form another function $f'$. If $f'$ violates any condition except for Conditions 4 and 5 of Definition 4.1, the new function $f'$ cannot lead to a local partial match (Lines 6-7 in Function ComParMatch). If $f'$ satisfies all conditions except for Conditions 4 and 5, it means that $f'$ can be further expanded (Lines 8-9 in Function ComParMatch). If $f'$ satisfies all conditions, then $f'$ specifies a local partial match and it is reported (Lines 10-11 in Function ComParMatch).

**Algorithm 1: Computing Local Partial Matches**

**Input:** A fragment $F_i$ and a query graph $Q$.

**Output:** The set of all local maximal partial matches in $F_i$, denoted as $\Omega(F_i)$.

1. Select one vertex $v$ in $Q$.
2. for each candidate vertex $u$ with regard to $v$ do
3. Initialize a function $f$ with $(v, u)$
4. Call Function ComParMatch($f$)
5. Return $\Omega(F_i)$;

**Function ComParMatch($f$)**

1. if all vertices of query $Q$ have been matched in the function $f$ then
2. return;
3. select an unmatched $v'$ adjacent to a matched vertex $v$ in the function $f$ for each candidate vertex $u'$ with regard to $v'$ do
4. if $f''$ violates any condition (except for condition 4 and 5 of Definition 4.1) then
5. continue
6. if $f''$ satisfies all conditions (except for condition 4 and 5 of Definition 4.1) then
7. ComParMatch($f''$)
8. if $f''$ satisfies all conditions of Definition 4.1 then
9. $f$ specifies a local partial match $PM$ that will be inserted into the answer set $\Omega(F_i)$;

As shown in Algorithm 1, in each step, we introduce a new candidate vertex pair $(v', u')$, where $v'$ is a vertex in query graph $Q$. The order of selecting query vertex can be arbitrarily defined. However, QuickSI [29] proposes several heuristic rules to select an optimized order that can speed up the matching process. These rules are also utilized in our experiments.

5. **ASSEMBLY**

Using Algorithm 1, each site $S_i$ finds all local partial matches in fragment $F_i$. The next step is to assemble partial matches to form crossing matches and compute the final result. We propose two assembly strategies: centralized and distributed (or parallel). In centralized, all local partial matches are sent to a single site for assembly. For example, in a client/server system, all local partial matches may be sent to the server. In distributed/parallel, local partial matches are combined at a number of sites in parallel. In Section 5.1, we define a basic join operator for assembly. Then, we propose a centralized assembly algorithm in Section 5.2 using the join operator. In Section 5.3, we study how to assemble local partial matches in a distributed manner.

5.1 **Join-based Assembly**

We first define the conditions under which two partial matches are joinable. Obviously, crossing matches can only be formed by assembling partial matches from different fragments. If local partial matches from the same fragment could be assembled, this would result in a larger local partial match in the same fragment, which is contrary to Theorem 4.3.

**Definition 5.1. (Joinable)** Given a query graph $Q$ and two fragments $F_i$ and $F_j$ ($i \neq j$), let $PM_i$ and $PM_j$ be two local partial matches over fragments $F_i$ and $F_j$ under functions $f_i$ and $f_j$, respectively. $PM_i$ and $PM_j$ are joinable if and only if the following conditions hold: 1) There exist no internal vertices $u$ and $u'$ in $PM_i$ and $PM_j$, respectively, such that $f_i^{-1}(u) = f_j^{-1}(u')$; 2) There exists at least one crossing edge $uu'$ such that $u$ is an internal vertex and $u'$ is an extended vertex in $F_j$, while $u'$ is an extended vertex and $u$ is an internal vertex in $F_i$. Furthermore, $f_i^{-1}(u) = f_j^{-1}(u)$ and $f_j^{-1}(u') = f_i^{-1}(u')$.

The first condition says that there does not exist different vertices in separate partial matches that match the same query vertex. The second condition says that two local partial matches share at least one common crossing edge that corresponds to the same query edge.
Example 5. Let us recall query two local partial matches $PM_i^j$ and $PM_i^k$ in Figure 4. There do not exist two different vertices in the two local partial matches that matches the same query vertex. Furthermore, they share a common crossing edge 002, 001, where 002 and 001 match query vertices $v_2$ and $v_1$ in the two local partial matches, respectively. Hence, they are joinable.

The join result of two joinable local partial matches is defined as follows.

Definition 5.2. Join Result. Given a query graph $Q$ and two fragments $F_i$ and $F_j$, $i \neq j$, let $PM_i$ and $PM_j$ be two joinable local partial matches of $Q$ over fragments $F_i$ and $F_j$, under functions $f_i$ and $f_j$, respectively. The join of $PM_i$ and $PM_j$, is defined under a new function $f$ (denoted as $PM = PM_i \triangleright f_i PM_j$), which is defined as follows for any vertex $v$ in $Q$:

1. if $f_i(v) \neq NULL \land f_j(v) = NULL$ then $f(v) \leftarrow f_i(v)$
2. if $f_i(v) = NULL \land f_j(v) \neq NULL$, $f(v) \leftarrow f_j(v)$;
3. if $f_i(v) \neq NULL \land f_j(v) \neq NULL$, $f(v) \leftarrow f_i(v)$ (In this case, $f_i = f_j$);
4. if $f_i(v) = NULL \land f_j(v) = NULL$, $f(v) \leftarrow NULL$.

5.2 Centralized Assembly

In centralized assembly, all local partial matches are sent to a single assembly site. We propose an iterative join algorithm (Algorithm 2) to find all crossing matches. In each iteration, a pair of local partial matches are joined. When the join is complete (i.e., a match has been found), the result is returned (Lines 12-13 in Algorithm 2); otherwise, it is joined with other local partial matches in the next iteration (Lines 14-15). There are $|V(Q)|$ iterations of Lines 4-16 in the worst case, since at each iteration only a single new matching vertex is introduced (worst case) and $Q$ has $|V(Q)|$ vertices. If no new intermediate results are generated at some iteration, the algorithm can stop early (Lines 5-6).

5.2.1 Partitioning-based Join Processing

The join space in Algorithm 2 is large, since we need to check all partial matches from different fragments. For each local partial match, there is an internal vertex that matches $v_1$ in query graph. The underline indicates the extended vertex in the local partial match. According to Theorem 5.1 none of them are joinable.

Algorithm 2: Centralized Join-based Assembly

Input: $\Omega(F_i)$, i.e., the set of local partial matches in each fragment $F_i$, $i = 1, \ldots, k$

Output: All crossing matches set $RS$

1. Each fragment $F_i$ sends the set of local partial matches in each fragment $F_j$ (i.e., $\Omega(F_j)$) to a single site for the assembly
2. Let $\Omega \leftarrow \bigcup_{i=1}^{k} \Omega(F_i)$
3. Set $MS \leftarrow \Omega$
4. for $N \leftarrow 1$ to $|V(Q)|$ do
5. if $|MS| = 0$ then
6. Break;
7. Set $MS' \leftarrow \phi$
8. for each local partial match $PM$ in $MS$ do
9. for each local partial match $PM'$ in $\Omega$ do
10. if $PM$ and $PM'$ are joinable then
11. Set $PM'' = PM \triangleright f_i PM'$
12. if $PM''$ is a complete match of $Q$ then
13. put $PM''$ into $RS$
14. else
15. put $PM''$ into $MS'$
16. $MS \leftarrow MS'$
17. Return $RS$

Matching to $v_1$

$PM_1^1 = \{002, 001, 000, NULL, NULL, NULL\}$
$PM_2^1 = \{002, 001, 009, NULL, 018, NULL\}$
$PM_3^1 = \{002, 001, 010, NULL, 019, NULL\}$
$PM_4^1 = \{004, 001, 003, NULL, NULL\}$

Figure 5: Partitioning Local Partial Matches on $v_1$

Definition 5.3. Local Partial Match Partitioning. Consider a SPARQL query $Q$ with $n$ vertices $\{v_1, \ldots, v_n\}$. Let $\Omega$ denote all local partial matches. $\mathcal{P} = \{P_1, \ldots, P_n\}$ is a partitioning of $\Omega$ if and only if the following conditions hold: 1). Each partition $P_i$ ($i = 1, \ldots, n$) consists of a set of local partial matches, each of which has an internal vertex that matches $v_i$; 2). $P_i \cap P_j = \emptyset$, where $1 \leq i \neq j \leq n$; 3). $P_1 \cup P_2 \cup \ldots \cup P_n = \Omega$.

Example 7. Let us consider all local partial matches of our running example in Figure 4. Figure 6 show an example partitioning.

As mentioned earlier, we only need to consider joining local partial matches from different partitions. Given a partitioning $\mathcal{P} = \{P_1, \ldots, P_n\}$, Algorithm 3 shows how to perform partitioning-based join of local partial matches. Note that different partitionings and the different join orders in the partitioning will impact the performance of Algorithm 3. In Algorithm 3, we assume that the partitioning $\mathcal{P} = \{P_1, \ldots, P_n\}$ is given, and that the join order is from $P_1$ to $P_n$, i.e., the order in $\mathcal{P}$. Choosing a good partitioning and the optimal join order will be discussed in Section 5.2.8 and Section 5.2.9.

The basic idea of Algorithm 3 is to iterate the join process on each partition. First, we set $MS \leftarrow P_i$ (Line 1 in Algorithm 3). Then, we try to join local partial matches $PM$ in $MS$ with local partial matches $PM'$ in $P_i$ (the first loop of Line 3-13). If the join result is a complete match, it is inserted into the answer set $RS$ (Lines 8-9). If the join result is an intermediate result, we insert it into a temporary set $MS'$ (Lines 10-11). We also need to insert $PM'$ into $MS'$, since the local partial match $PM'$ in $P_{i+1}$ will join local partial matches in the later partitions (Line 12).

At the
end of the iteration, we insert all intermediate results (in \(MS'\)) into \(MS\), which will join local partial matches in the later partitions in the next iterations (Line 13). We iterate the above steps for each partition in the partitioning (Lines 3-13).

**Algorithm 3:** Partitioning-based Joining Local Partial Matches

**Input:** A partitioning \(\mathcal{P} = \{P_1, ..., P_n\}\) of all local partial matches.

**Output:** All crossing matches set \(RS\).

1. \(MS \leftarrow P_1\)
2. for \(i \leftarrow 2\) to \(n\) do
   3. \(MS' \leftarrow \emptyset\)
   4. for each partial match \(PM\) in \(MS\) do
      5. for each partial match \(PM'\) in \(P_i\) do
         6. if \(PM\) and \(PM'\) are joinable then
            7. Set \(PM'' \leftarrow PM \bowtie PM'\)
            8. if \(PM''\) is a complete match then
               9. Put \(PM''\) into the answer set \(RS\)
            10. else
               11. Put \(PM''\) into \(MS'\)
      12. end for
   13. end for
   14. Insert \(MS'\) into \(MS\)
   15. end for
16. Return \(RS\)

### 5.2.2 Finding the Optimal Partitioning

Obviously, different partitionings lead to different join costs. Thus, we define a cost-based “best” partitioning in this section.

**Definition 5.4.** Join Cost. Given a query graph \(Q\) with \(n\) vertices \(v_1, ..., v_n\) and a partitioning \(\mathcal{P} = \{P_1, ..., P_n\}\) over all local partial matches \(\Omega\), the join cost is defined as follows.

\[
\text{Cost}(\Omega) = \text{Cost}(\Omega_{\text{opt}}) + \text{Cost}(\Omega_{\text{opt}}) - \text{Cost}(\Omega_{\text{opt}}) = |U_{v_i}| \times \text{Cost}(\Omega_{\text{opt}})
\]

where \(|P_v|\) is the number of local partial matches in \(P_v\) and \(I\) is introduced to avoid the “0” element in the product.

Definition 5.4 assumes that each pair of local partial matches (from different partitions) are joinable, which is the worst case, so that we can quantify the worst-case performance. For example, the cost of the partitioning in Figure 6 is \(6 \times 3 \times 4 = 72\). Of course, more sophisticated and more realistic cost functions can be used instead, but, determining the most appropriate cost function is a major research issue in itself and outside the scope of this paper. We leave it as an interesting future research to study.

The key issue is how to find the optimal partitioning (i.e., the partitioning with the minimal join cost). A naive enumeration method has \(O(n^{2^k})\) time complexity, where \(n\) is the number of vertices in query \(Q\) and \(\Omega\) denotes all local partial matches. Obviously, the naive enumeration is not efficient, since \(|\Omega|\) is very large in practice. Therefore, we need to find better solutions for finding the optimal partitioning.

Actually, we can reduce a 0-1 integer planning problem (a classical NP-complete problem) to find the optimal partitioning. Thus, finding the optimal partitioning is a NP-complete problem.

**Theorem 5.2.** Finding the optimal partitioning (i.e., the join cost defined in Definition 5.4 is minimal) is a NP-complete problem.

Even that finding the optimal partitioning is a NP-complete problem, we can still find a much faster algorithm with time complexity \(O(2^n \times |\Omega|)\). Note that this is a fixed-parameter tractable algorithm since \(n\) is small in practice. For example, \(n\) is no larger than 10 in most benchmark SPARQL queries (such as LUBM benchmarks).

Our algorithm is based on the following characteristic of the optimal partitioning. Consider a query graph \(Q\) with \(n\) vertices \(v_1, ..., v_n\). Let \(U_{v_i}\) (\(i = 1, ..., n\)) denote all local partial matches in \(\Omega\) that have internal vertices matching \(v_i\). Unlike the partitioning defined in Definition 5.3, \(U_{v_i}\) and \(U_{v_j}\) (\(1 \leq i \neq j \leq n\)) may have overlaps. For example, \(PM_2\) (in Figure 6) contains an internal vertex 002 that matches \(v_1\), thus, \(PM_2\) in \(U_{v_1}\) also has internal vertex 010 that matches \(v_1\), thus, \(PM_2\) is also in \(U_{v_1}\). However, the partitioning defined in Definition 5.3 does not allow overlapping partitions.

If a partitioning \(\mathcal{P}_{\text{opt}} = \{P_{v_1}, ..., P_{v_n}\}\) is optimal, the following theorem holds.

**Theorem 5.3.** Given a query graph \(Q\) with \(n\) vertices \(v_1, ..., v_n\) and a set of all local partial matches \(\Omega\), let \(U_{v_i}\) (\(i = 1, ..., n\)) be all local partial matches in \(\Omega\) that have internal vertices matching \(v_i\). For the optimal partitioning \(\mathcal{P}_{\text{opt}} = \{P_{v_1}, ..., P_{v_n}\}\) where \(P_{v_i}\) has the highest partition size (i.e., the number of local partial matches in \(P_{v_i}\) is maximum) in \(\mathcal{P}_{\text{opt}}\), \(P_{v_i} = U_{v_i}\).

Based on Theorem 5.3, we propose a dynamic programming algorithm. The intuition of the algorithm is as follows. Assume that the optimal partitioning is \(\mathcal{P}_{\text{opt}} = \{P_{v_1}, P_{v_2}, ..., P_{v_n}\}\). We order the partitions in \(\mathcal{P}_{\text{opt}}\) in non-descending order of sizes, i.e., \(|P_{v_i}| \geq |P_{v_j}| \geq ... \geq |P_{v_k}|\). According to Theorem 5.3, \(P_{v_i} = U_{v_i}\). Let us denote by \(\Omega_{\text{opt}}\) all local partial matches in \(\Omega\) that do not contain internal vertices matching \(v_i\), i.e., \(\Omega_{\text{opt}} = \Omega - U_{v_i}\). It is straightforward to have the following optimal substructure in Equation 2.

\[
\text{Cost}(\Omega_{\text{opt}}) = |P_{v_i}| \times \text{Cost}(\Omega_{\text{opt}}) = |U_{v_i}| \times \text{Cost}(\Omega_{\text{opt}}) = \text{Cost}(\Omega_{\text{opt}})
\]

Since we do not know which vertex is \(v_i\), we introduce the following dynamic programming structure that is used in our dynamic programming algorithm.

\[
\text{Cost}(\Omega) = \text{MIN}_{\mathcal{P}_{\text{opt}}}(\text{Cost}(\Omega_{\text{opt}})) = \text{MIN}_{\mathcal{P}_{\text{opt}}}(|U_{v_i}| \times \text{Cost}(\Omega_{\text{opt}}))
\]

Obviously, it is easy to design a naive dynamic algorithm based on Equation 3. However, it can be further optimized by recording a problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. This property is often used in dynamic programming formulations.
some intermediate results. Based on Equation 3, we can prove the following equation.

\[
\text{Cost}(\Omega_{opt}) = \min_{\text{bipartite}}(\text{Cost}(P_{i_1} \times P_{i_2} \times \text{Cost}(\Omega_{opt})) = \min_{\text{bipartite}}(\text{Cost}(U_{i_1} \times \text{Cost}(\Omega_{opt}))
\]

where \(\Omega_{opt}\) denotes all local partial matches that do not contain internal vertices matching \(v_1\) or \(v_2\), and \(U_{i_1}\) denotes all local partial matches (in \(\Omega\)) that contain internal vertices matching vertex \(v_1\).

However, if Equation 4 is used naively in the dynamic programming formulation, it would result in repeated computations. For example, \(\text{Cost}(\Omega_{opt})\) will be computed twice in both \(|U_{i_1}| \times |U_{i_2}| \times \text{Cost}(\Omega_{opt})\) and \(|U_{i_2}| \times |U_{i_1}| \times \text{Cost}(\Omega_{opt})\). To avoid this, we introduce a map that records \(\text{Cost}(\Omega)\) that have been already calculated (Line 16 in Function OptComCost), so that subsequent uses of \(\text{Cost}(\Omega)\) can be serviced directly from searching the map (Lines 8-10 in Function ComCost).

We can prove that there are \(2^n\) items in the map (worst case), where \(n = |V(Q)|\). Thus, the time complexity of the algorithm is \((2^n \times |Q|)\). Although the time complexity is still exponential, \(n\) i.e., \(|V(Q)|\) is small in practice. For example, \(n\) is no larger than 10 in most benchmark queries (such as LUBM benchmarks [10]).

### 5.2.3 Join Order

When we determine the optimal partitioning of local partial matches, the join order is also determined. If the optimal partitioning is \(P_{opt} = \{P_{i_1}, \ldots, P_{i_k}\}\) and \(|P_{i_1}| \geq |P_{i_2}| \geq \ldots \geq |P_{i_k}|\), then the join order must be \(P_{i_1} \bowtie P_{i_2} \bowtie \ldots \bowtie P_{i_k}\). The reasons are as follows.

First, changing the join order often cannot prune any intermediate results. Let us recall the example optimal partitioning \(\{P_{i_1}, P_{i_2}, P_{i_3}\}\) shown in Figure 6. The join order should be \(P_{i_1} \bowtie P_{i_2} \bowtie P_{i_3}\), and any changes in the join order can not prune intermediate results. However, if we first join \(P_{i_2}\) with \(P_{i_3}\), we can not prune the local partial matches in \(P_{i_1}\) that can not join with any local partial matches in \(P_{i_2}\). This is because there may be some local partial matches \(P_{i_3}\) that have an internal vertex matching \(v_1\) and can join with local partial matches in \(P_{i_2}\). In other words, the results of \(P_{i_3} \bowtie P_{i_1}\) is not smaller than \(P_{i_2}\). Similarly, any other changes of the join order of the partitioning have no effects.

Second, in some special cases, the join order may have an effect on the performance. Given a partitioning \(P_{opt} = \{P_{i_1}, \ldots, P_{i_k}\}\) and \(|P_{i_1}| \geq |P_{i_2}| \geq \ldots \geq |P_{i_k}|\), if the set of the first \(n'\) vertices, \(\{v_{i_1}, v_{i_2}, \ldots, v_{i_n'}\}\), is a vertex cut of the query graph, the join order for the remaining \(n - n'\) partitions has an effect. In the worst case, if the query graph is a complete graph, the join order has no effect on the performance.

### 5.3 Distributed Assembly

An alternative to centralized assembly is to assemble the local partial matches in a distributed fashion. We adopt Bulk Synchronous Parallel (BSP) model [31] to design our synchronous algorithm for distributed assembly. A BSP system consists of processors connected by a communication network. A BSP computation proceeds in a series of global supersteps. A superstep consists of three components: local computation, communication and barrier synchronisation. In the following, we discuss how we apply this strategy to distributed assembly.

**Local Computation**. Each processor performs some computation based on the data stored in the local memory. The computations on different processors are independent in the sense that different processors perform the computation in parallel.

Consider the \(m\)-th superstep. For each fragment \(F_i\), let \(\Delta_m(F_i)\) denote all received intermediate results in the \(m\)-th superstep and \(\Omega_m(F_i)\) denote all local partial matches and the intermediate results generated the first \((m - 1)\) supersteps. In the \(m\)-th superstep, we join local partial matches in \(\Delta_m(F_i)\) with local partial matches in \(\Omega_m(F_i)\) by Algorithm 4. The basic idea of Algorithm 4 is similar to Algorithm 2. It is an iterative algorithm. For each intermediate result \(PM\), we check if it can join with some local partial match \(PM'\) in \(\Omega_m(F_i) \cup \Delta_m(F_i)\). If the join result \(PM'' = PM \bowtie PM'\) is a complete crossing match, it is returned. If the join result \(PM''\) is an intermediate result, we will check if \(PM''\) can further join with another local partial match in \(\Omega_m(F_i) \cup \Delta_m(F_i)\) in the next iteration. We also insert the intermediate result \(PM''\) into \(\Delta_m(F_i)\) that will be sent to other fragments in the communication step discussed below. Of course, we can also use the partitioning-based solution (in Section 5.2.1) to optimize join processing, but we do not discuss that due to space limitation.

**Algorithm 4**: Local Computation in Each Fragment \(F_i\)

**Input**: \(\Omega_m(F_i)\), the local partial matches in fragment \(F_i\)

**Output**: \(RS\), the crossing matches found at this superstep; \(\Delta_m(F_i)\), the intermediate results that will be sent

1. Let \(\Omega = \Omega_m(F_i) \cup \Delta_m(F_i)\)
2. Set \(MS = \Delta_m(F_i)\)
3. for \(N = 1\) to \(|V(Q)|\) do
4. if \(|MS| = 0\) then
5. Break;
6. Set \(MS' = \phi\)
7. for each local partial match \(PM\) in \(MS\) do
8. for each local partial match \(PM'\) in \(\Omega_m(F_i) \cup \Delta_m(F_i)\) do
9. if \(PM\) and \(PM'\) are joinable then
10. \(PM'' = PM \bowtie PM'\)
11. if \(PM''\) is a SPARQL match then
12. Put \(PM''\) into the answer set \(RS\)
13. else
14. Insert \(MS'\) into \(\Delta_m(F_i)\)
15. Clear \(MS\) and \(MS'\)
16. \(\Omega^{n+1}(F_i) = \Omega_m(F_i) \cup \Delta_m(F_i)\)
17. Return \(RS\) and \(\Delta_m(F_i)\)

**Communication**. All processors exchange data among themselves. Consider the \(m\)-th superstep. The straightforward communication strategy works as follows. If an intermediate result \(PM\) in \(\Delta_m(F_i)\) shares a crossing edge with fragment \(F_j\), \(PM\) will be sent to site \(S_j\) from \(S_i\) (assuming fragments \(F_i\) and \(F_j\) are stored in sites \(S_i\) and \(S_j\), respectively).

However, the above communication strategy may generate duplicate results. For example, as shown in Figure 4, we can assemble \(PM_i\) at site \(S_i\) and \(PM_j\) at site \(S_j\) to form a complete crossing match. According to the straightforward communication strategy, \(PM_i\) will be sent to \(S_j\) from \(S_i\). Then, we can get \(PM_j \bowtie PM_i\) at site \(S_j\). Analogously, we also send \(PM_j\) from \(S_j\) to \(S_i\) and assemble them at site \(S_1\). In other words, we obtain the join result \(PM_j \bowtie PM_i\) at both sites \(S_1\) and \(S_3\). This wastes resources and increases total evaluation time.

To avoid duplicate result computation, we introduce a “divide-and-conquer” approach. We define a total order of fragment \(F_i\) \((i = 1, \ldots, k)\) in a non-descending order of \(|\Omega_m(F_i)|\), i.e., the number of local partial matches in fragment \(F_i\) found at the partial evaluation stage. Formally, the definition is as follows.

**Definition 5.5.** Given any two fragments \(F_i\) and \(F_j<br />(i and only if |\Omega_m(F_i)| \leq |\Omega_m(F_j)| (1 \leq i, j \leq n).**
Without loss of generality, we assume that $F_1 < F_2 < \ldots < F_n$ in the remainder. The basic idea of the divide-and-conquer approach is as follows. Assume that a crossing match $M$ is formed by joining local partial matches that are from different fragments $F_{i_1}, \ldots, F_{i_m}$, where $F_{i_1} < F_{i_2} < \ldots < F_{i_m}$ (1 ≤ $i_1, \ldots, i_m$ ≤ $n$). The crossing match should only be generated at fragment site $S_{i_m}$ rather than other fragment sites.

For example, at site $S_2$, we generate crossing matches by joining local partial matches from $F_1$ and $F_2$. The crossing matches generated at $S_2$ should not contain any local partial match from $F_3$ or even larger fragments (such as $F_4, \ldots, F_n$). Similarly, at site $S_3$, we should generate crossing matches by joining local partial matches from $F_3$ and fragments smaller than $F_3$. The crossing matches should not contain any local partial match from $F_4$ or even larger fragments (such as $F_5, \ldots, F_n$). As mentioned earlier, we define the total order for fragments $F_i$ according to the non-descending order of $|\Omega(F_i)|$. The basic intuition is the “load balancing” of each site.

The above “divide-and-conquer” framework can avoid duplicate results, since each crossing match can be only generated at a single site according to the “dividing space”. To enable the “divide-and-conquer” framework, we need to introduce some constraints over data communication. The transmission (of local partial matches) from fragment site $S_i$ to $S_j$ is allowed only if $F_i < F_j$.

Let us consider an intermediate result $PM$ in $\Delta_{in}^{out}(F_i)$. Assume that $PM$ is generated by joining intermediate results from $m$ different fragments $F_{i_1}, \ldots, F_{i_m}$, where $F_{i_1} < F_{i_2} < \ldots < F_{i_m}$. We send $PM$ to another fragment $F_j$ if and only if the two conditions hold: 1) $F_j > F_{i_m}$; and 2) $F_j$ shares common crossing edges with at least one fragment of $F_{i_1}, \ldots, F_{i_m}$.

**Barrier Synchronisation.** All communication in the $m$-th superstep should finish before entering in the $(m+1)$-th superstep. We now discuss the initial state (i.e., 0-th superstep) and the system termination condition.

**Initial State.** In the 0-th superstep, each fragment $F_i$ has only local partial matches in $F_i$, i.e., $\Omega_i$. Since it is impossible to assemble local partial matches in the same fragment, the 0-th superstep requires no local computation. It Enter comes from stage directly. Each site $S_j$ sends $\Omega_j$ to other fragments according to the communication strategy that has been discussed before.

**System Termination Condition.** A key problem in the BSP algorithm is the number of the supersteps to terminate the system. In order to facilitate the analysis, we propose utilizing a fragmentation graph topology graph.

**Definition 5.6. (Fragmentation Topology Graph)** Given a fragmentation $\overline{T} = (F_1, F_2, \ldots, F_n)$ over an RDF graph $G$, the corresponding fragmentation topology graph $T$ is defined as follows: 1) Each node in $T$ is a fragment $F_i$, $i = 1, \ldots, k$; 2) There is an edge between nodes $F_i$ and $F_j$ in $T$, $1 \leq i \neq j \leq n$, if and only if there is at least one crossing edge between $F_i$ and $F_j$ in RDF graph $G$.

Let $Dia(T)$ be the diameter of $T$. We need at most $Dia(T)$ supersteps to transfer the local partial matches in one fragment $F_i$ to any other fragment $F_i$. Hence, the number of the supersteps in the BSP-based algorithm is $Dia(T)$.

### 6. EXPERIMENTS

We evaluate our method over both real and synthetic RDF datasets, and compare our approach with the state-of-the-art distributed RDF systems, including both federated SPARQL query systems (DARQ [27], FedEx [28] and Q-Tree [26]) and RDF data partition-based methods (GraphPartition [15] and EAGRE [33]).

**Setting.** We use one synthetic dataset with different sizes, and two real datasets in our experiments. Table 1 summarizes the statistics of these datasets. All sample queries are shown in Appendix.

1) LUBM [10] is a benchmark that adopts an ontology for the university domain, and can generate synthetic OWL data scalable to an arbitrary size. We vary the university number from 1000 to 10000. The number of triples is 133 million to 1.33 billion. On these experiments, we partition the LUBM datasets according to the university identifiers. We use the 7 benchmark queries in [2] to evaluate our method.

2) BTC 2012 (http://km.aifb.kit.edu/projects/btc-2012/) is a dataset that serves as basis for 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 METIS [19], a popular min-cut based graph partition algorithm, to partition it into several fragments. In addition, we use 7 queries in [34].

3) LOD is a dataset generated from ten datasets in Linked Open Data to further evaluate our methods in a real federated database system environment. We download ten datasets (DBpedia, LinkedMDB, Yago2, FreeBase, NYTimes, MusicBrainz, Disease, Geonames, Timbl, LinkedGeoData). By randomly selecting one as the primary and others as the copies of the primary, we merge entities in different datasets connected using predefined property “owl:sameAs”.

Each dataset maps to a fragment. We also define six sample queries across different fragments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Triples</th>
<th>RDF N3 File Size(KB)</th>
<th>Number of Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM 1000</td>
<td>133,553,834</td>
<td>35,136,798</td>
<td>21,715,108</td>
</tr>
<tr>
<td>LUBM 5000</td>
<td>667,292,653</td>
<td>76,522,983</td>
<td>108,498,627</td>
</tr>
<tr>
<td>LUBM 10000</td>
<td>1,334,481,197</td>
<td>153,256,699</td>
<td>217,006,852</td>
</tr>
<tr>
<td>BTC</td>
<td>1,056,184,911</td>
<td>238,970,296</td>
<td>183,233,054</td>
</tr>
<tr>
<td>LOD</td>
<td>1,055,264,649</td>
<td>169,175,000</td>
<td>146,228,189</td>
</tr>
</tbody>
</table>

**Table 1: Datasets**

We conduct all experiments on a cluster of 10 machines running Linux, each of which has one CPU with four cores of 3.06GHz, 16GB memory and 500GB disk storage. At each site, we install gStore [16] to find inner matches, since it supports the graph-based SPARQL evaluation paradigm. We revise gStore to find all local partial matches in each fragment as discussed in Section 4. All implementations are in standard C++. We use MPICH-3.0.4 library for communication.

**Exp 1. Evaluating Each Stage’s Performance.** In this experiment, we study the performance of our system in each stage (i.e., partial evaluation and assembly process) with regard to different queries in LUBM1000. We report the running time of each stage (i.e., partial evaluation and assembly) and the number of local partial matches, inner matches, crossing matches, with regard to different queries in Table 2. We also compare the centralized and distributed assembly strategies. Note that we classify SPARQL queries into three categories according to query graphs’ structures: star, snowflake (several stars linked by a path) and complex (a combination of the above with complex structure).

**Partial Evaluation:** Table 3 shows that if there are some consistent entities (such as “http://www.Department.University.edu” in $Q_3$ of LUBM) in the query, the partial evaluation is much faster than others. Our partial evaluation algorithm (Algorithm 1) is based on a state transformation, while the consistent entities can reduce the search space. Furthermore, the running time also depends on the number of inner matches and local partial matches, as shown in Table 2. More inner matches and local partial matches lead to higher running time in the partial evaluation stage.

**Assembly:** In this experiment, we compare centralized and distributed assembly approaches. Obviously, there is no assembly process for a star query. Thus, we only study the performance of snowflake and complex queries. We find that distributed assembly can beat the centralized one when there are lots of local partial
matches and crossing matches. The reason is as follows: in centralized assembly, all local partial matches need to be sent to the server where they are assembled. Obviously, if there are lots of local partial matches, the server becomes the bottleneck. However, in distributed assembly, we can take advantage of the parallelization to speed up both the network communication and assembly. For example, in Q7, there are 167621 local partial matches. It takes a long time to transfer the local partial matches to the server and assemble them in the server in centralized assembly. So, distributed assembly outperforms the centralized one by six times. However, if the number of local partial matches and the number of crossing matches are small, the barrier synchronisation cost dominates the total cost in distributed assembly. In this case, the advantage of distributed assembly is not clear. A quantitative comparison between distributed assembly and centralized one needs more statistics about the network communication, CPU and other parameters. A sophisticated quantitative study is beyond the scope of this paper and is left as future work.

Exp 2: Evaluating Optimizations in Assembly. In this experiment, we evaluate two different optimization techniques in the assembly: partition-based join strategy (Section 5.1) and the divide-and-conquer approach in the distributed assembly (Section 5.3). Star queries Q2, Q3, and Q4 do not need the assembly process. Query Q5 does not have any match in RDF graph G. Therefore, we only use the three benchmark queries Q1, Q6, and Q7 in our experiments.

Partitioning-based Join. First, we compare partitioning-based join (i.e., Algorithm 3) with naive join processing (i.e., Algorithm 2) in Table 3, which shows that the partitioning-based strategy can greatly reduce the join cost. Second, we evaluate the effectiveness of our cost model. Note that the join order depends on the partitioning strategy, which is based on our cost model as discussed in Section 5.3. In other words, once the partitioning is given, the join order is fixed. So, we use the cost model to find the optimal partitioning and report the running time of the assembly process in Table 3. We find that the assembly with optimal partitioning is significantly faster than that with the random partitioning, which confirms the effectiveness of our cost model.

<table>
<thead>
<tr>
<th>Star</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td># of LPMs</td>
<td># of IMs</td>
<td>Centralized</td>
<td>Distributed</td>
<td># of CMs</td>
<td>Time (ms)</td>
<td>Centralized</td>
</tr>
<tr>
<td>Q1</td>
<td>1818</td>
<td>0</td>
<td>1081187</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1818</td>
</tr>
<tr>
<td>Q2</td>
<td>82</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>82</td>
<td>82</td>
</tr>
<tr>
<td>Q3</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 2: Evaluation of Each Stage

<table>
<thead>
<tr>
<th>Snowflake</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td># of LPMs</td>
<td># of IMs</td>
<td>Centralized</td>
<td>Distributed</td>
<td># of CMs</td>
<td>Time (ms)</td>
<td>Centralized</td>
</tr>
<tr>
<td>Q1</td>
<td>52548</td>
<td>3033</td>
<td>2324</td>
<td>53</td>
<td>60</td>
<td>4</td>
<td>52601</td>
</tr>
<tr>
<td>Q2</td>
<td>920</td>
<td>3358</td>
<td>0</td>
<td>36</td>
<td>48</td>
<td>0</td>
<td>956</td>
</tr>
<tr>
<td>Q3</td>
<td>3945</td>
<td>10721</td>
<td>42749</td>
<td>211670</td>
<td>35858</td>
<td>1709</td>
<td>215615</td>
</tr>
</tbody>
</table>

Exp 3: Scalability Test. In this experiment, we vary the RDF dataset size from 133 million triples (LUBM 1000) to 1.3 billion triples (LUBM 10000) to study the scalability of our methods. Table 5 shows the performance of different queries of both centralized and distributed assembly.

From LUBM1000 to LUBM10000, the data size increases 10 times and this causes the query response times to increase 10-20 times. The increase is also affected by the query graph’s shape. For star queries, the query response time increases proportional to the data size, as shown in Table 5. For snowflake/complex queries, the query response times may grow faster than the data size, as shown in Table 5. Especially for Q3, the query response time increases 20 times as the data size increases 10 times. This is because the complex query graph shape causes more complex operations in query processing, such as joining and assembly. However, even for complex queries, the growth of the query response times are approximatively linear with the RDF graph size.

Table 4: Dividing VS. No Dividing (in ns)

<table>
<thead>
<tr>
<th>Distributed Assembly Time (in ns)</th>
<th>Dividing</th>
<th>No Dividing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>60</td>
<td>86</td>
</tr>
<tr>
<td>Q2</td>
<td>125</td>
<td>127</td>
</tr>
<tr>
<td>Q3</td>
<td>35858</td>
<td>35856</td>
</tr>
</tbody>
</table>

Exp 4: Comparing with State-of-the-Art Algorithms. In this experiment, we compare our approach with some state-of-the-art distributed RDF systems, including both federated SPARQL query systems (DARQ [27], FedX [28] and Q-Tree [25]) and RDF data partition-based methods (GraphPartition [15] and EAGRE [24]).

Comparing with Federated SPARQL queries. For further evaluating our methods, we download ten datasets (DBpedia, LinkedMDB, Yago2, FreeBase, NYTtimes, MusicBrainz, Diseasome, Geonames, Timbl, LinkedGeoData) from LOD to integrate a real LOD
and DARQ cannot answer and DARQ by almost one order of magnitude. Furthermore, FedX is decomposed into a set of subqueries and each subquery is performed locally on each fragment. They do not consider the whole query graph structure to prune the intermediate results. However, in our method, each fragment receives the whole query graph, and can utilize the query graph structure to prune more false positive intermediate results. Second, both GraphPartition and EAGRE are implemented on Hadoop while our system is running with MPI, a lightweight parallel library. According to our experiments, starting Hadoop jobs always spends 20-30 seconds.

In particular, for star queries (such as Q2, Q4 and Q5 of LUBM), since their diameters are not larger than 2, GraphPartition can answer all these queries entirely without data communication in Hadoop. Therefore, the query response time of GraphPartition for these queries is as fast as our methods. However, for snowflake/complex queries whose diameters are larger than 2, GraphPartition needs to decompose the queries and join the intermediate results. Then, the query response time grows greatly. The performance differences also depend on the intermediate result sizes. Q1 has more intermediate results, thus, the performance difference is not clear comparing with GraphPartition and EAGRE.

Finally, our method does not depend any specific graph partitioning strategy (which is known to be NP-hard). However, the query algorithms of GraphPartition and EAGRE depend on their own graph partitions. Therefore, neither GraphPartition nor EAGRE can be used directly in the virtually integrated distributed RDF graphs, such as LOD data, since there is no possibility of changing the partitioning of the data.

### 7. CONCLUSION

In this paper, we propose a distributed graph-based approach to executing SPARQL queries in a federated RDF repository as in the linked open data. We treat the interconnected RDF repositories (in LOD) as a virtually integrated distributed database, and adopt a partial function evaluation approach. This is different from most dataset. Each dataset maps to a fragment and is stored at a site. We also define six queries across different fragments. Two of these six queries (Q1 and Q2) are stars, two (Q3 and Q4) are snowflakes and two (Q5 and Q6) are complex queries. Furthermore, three queries (Q1, Q2 and Q3) contain selective triple patterns, while the others (Q4, Q5 and Q6) do not. We report these six sample queries in Appendix.

We compare our method with FedX and DARQ in the LOD dataset in Figure 7 which shows that our method outperform FedX and DARQ by almost one order of magnitude. Furthermore, FedX and DARQ cannot answer Q4 and Q6 because it runs of memory. FedX and DARQ decompose the SPARQL into a set of subqueries, and join the intermediate results of all subqueries together to find the final results. When the intermediate results of two subqueries join together, FedX and DARQ use the bound join. This means that FedX and DARQ first use the intermediate results to rewrite the subquery with bound join variables. Then, they send out and evaluate the rewritten queries in the corresponding sites. Therefore, when the SPARQL queries (like Q2, Q4 and Q6) do not contain any selective triple pattern, the size of intermediate results is so large that evaluation of bound joins takes lots of time. In Q1 and Q6, the intermediate results exhausted the available memory. Q-Tree [20] optimizes the query process by encoding the intermediate results in bitsets and uses bitwise AND-operations to filter out irrelevant intermediate results before join. However, our method still outperforms Q-Tree by at least three times except for Q1, as shown in Figure 8. Our method is faster than Q-Tree on Q1 by 60%, since Q1 is a simple star query with a selective triple pattern.

Comparing with RDF Data Partition-based Methods. In this experiment, we compare our method with data partition-based methods. It means that we assume that all RDF data are downloaded into a server. Then, we partition the whole RDF graph into several fragments that are stored in a cluster of machines. We consider two state-of-the-art algorithms, i.e., GraphPartition [15] and EAGRE [14] on two RDF datasets with more than one billion triples, BTC and LUBM 10000. Figure 8 shows the performance of different approaches in the two RDF datasets. We find out that our method outperforms GraphPartition and EAGRE greatly in most cases. For example, in BTC, our method is faster than both GraphPartition and EAGRE by an order of magnitude for queries Q1, Q2 and Q6.

There are two reasons for this phenomena. First, both GraphPartition and EAGRE are based on query decomposition. A query is decomposed into a set of subqueries and each subquery is performed locally on each fragment. They do not consider the whole query graph structure to prune the intermediate results. However, in our method, each fragment receives the whole query graph, and can utilize the query graph structure to prune more false positive intermediate results. Second, both GraphPartition and EAGRE are implemented on Hadoop while our system is running with MPI, a lightweight parallel library. According to our experiments, starting Hadoop jobs always spends 20-30 seconds.

In particular, for star queries (such as Q2, Q4 and Q5 of LUBM), since their diameters are not larger than 2, GraphPartition can answer all these queries entirely without data communication in Hadoop. Therefore, the query response time of GraphPartition for these queries is as fast as our methods. However, for snowflake/complex queries whose diameters are larger than 2, GraphPartition needs to decompose the queries and join the intermediate results. Then, the query response time grows greatly. The performance differences also depend on the intermediate result sizes. Q1 has more intermediate results, thus, the performance difference is not clear comparing with GraphPartition and EAGRE.

Finally, our method does not depend any specific graph partitioning strategy (which is known to be NP-hard). However, the query algorithms of GraphPartition and EAGRE depend on their own graph partitions. Therefore, neither GraphPartition nor EAGRE can be used directly in the virtually integrated distributed RDF graphs, such as LOD data, since there is no possibility of changing the partitioning of the data.
existing distributed SPARQL query algorithms, since they require re-partitioning the whole RDF graph as their own methods require. In contrast, in LOD, the RDF data are already distributed over different sites, thus, re-partitioning and redistributing are usually not possible. Our experiments show that the proposed algorithms perform very well. It does require minor modifications to SPARQL endpoints to evaluate partial queries. There are a number of extensions we are currently working on. An important one is handling data at sites that do not have query capability (i.e., they are not SPARQL endpoints). Data at these sites need to be moved for processing that will affect the algorithm and cost functions. Furthermore, similarity search over LOD is also an ongoing work. Users may not specify the exact SPARQL query statements, because they have no full knowledge of the underlying schema of RDF graph in LOD.

8. REFERENCES

APPENDIX

A. PROOFS

Theorem 4.1 Given any crossing match \( M \) of SPARQL query \( Q \) in an RDF graph \( G \), if \( M \) overlaps with some fragment \( F_j \) (i.e., exists in \( F_j \)), the overlapping part between \( M \) and fragment \( F_j \) denoted as \( PM_j \), must be a local partial match in fragment \( F_j \).

Proof. Since \( M \) is a SPARQL query match and the overlapping part is a subset of the SPARQL match, it is easy to know that Conditions 1-6 of Definition 4.1 hold.

Since \( M \) is a crossing match, the overlapping part between \( M \) and fragment \( F_j \) must contain a crossing edge, thus, Condition 7 holds.

Since the overlapping part \( PM \) is a subgraph of match \( M \), according to Claim 1, we can prove that Condition 8 holds.

If the overlapping part between \( M \) and fragment \( F_j \) has several weakly connected components, we consider each weakly connected component individually. Obviously, each weakly connected component satisfies Condition 9.

To summarize, the overlapping part between \( M \) and fragment \( F_j \) satisfies all conditions in Definition 4.1. Thus, Theorem 4.1 holds.

Theorem 4.2 The results generated by the partial-evaluation-and-assembly algorithm satisfy no-false-negative requirement if and only if all local partial matches in each fragment are found in the partial evaluation stage.

Proof. In two parts:

1) The “If” part: (proven by contradiction).

Assume that all local partial matches are found in each fragment \( F_j \), but a cross match \( M \) is not found in the answer set.

Since \( M \) is a crossing match, suppose that \( M \) overlaps with \( m \) fragments \( F_1,...,F_m \). According to Theorem 4.1, the overlapping part between \( M \) and fragment \( F_j \) (i.e., \( PM_j \)) must be a local partial match \( PM \) in \( F_j \). According to the assumption, these local partial matches have been found in the partial evaluation stage. Obviously, we can assemble these local partial matches \( PM \) (i = 1, ..., \( m \)) to form the complete cross match \( M \).

In other words, \( M \) would not be missed if all local partial matches are found. This contradicts the assumption.

2) The “Only” part: (proved by contradiction).

We assume that a local partial match \( PM \) in fragment \( F_j \) is missed and the answer set cannot satisfy the no-false-negative requirement.

Suppose that \( PM \) matches a part of \( Q \), denoted as \( Q' \). Assume that there exists another local partial match \( PM' \) in \( F_j \) that matches a complementary graph of \( Q' \), denoted as \( Q'' \), i.e., \( Q'' \cap Q' = Q \). In this case, we can obtain a complete match \( M \) by assembling the two local partial matches. If \( PM \) is missed, then match \( M \) is missed. In other words, it cannot satisfy the no-false-negative requirement. This also contradicts the assumption.

Theorem 5.1 Given two local partial matches \( PM_i \) and \( PM_j \) from fragments \( F_i \) and \( F_j \) with functions \( f_i \) and \( f_j \), respectively, if there exists a query vertex \( v \) where both \( f_i(v) \) and \( f_j(v) \) are internal vertices of fragments \( F_i \) and \( F_j \), respectively, \( PM_i \) and \( PM_j \) are not joinable.

Proof. If \( f_i(v) \neq f_j(v) \), then a vertex \( v \) in query \( Q \) matches two different vertices in \( PM_i \) and \( PM_j \), respectively. Obviously, \( PM_i \) and \( PM_j \) cannot be joinable.

If \( f_i(v) = f_j(v) \), since \( f_i(v) \) and \( f_j(v) \) are both internal vertices, both \( PM_i \) and \( PM_j \) are from the same fragment. As mentioned earlier, it is impossible to assemble two local partial matches from the same fragment (see the first paragraph of Section 5.1), thus, \( PM_i \) and \( PM_j \) cannot be joinable.

Theorem 5.2 Finding the optimal partition is NP-complete problem.

Proof. We can reduce a 0-1 integer planning problem to finding the optimal partition. We build a bipartite graph \( B \), which contains two vertex groups \( B_1 \) and \( B_2 \). Each vertex \( a_i \) in \( B_1 \) corresponds to a local partial match \( PM_j \) in \( \Omega \), \( j = 1, ..., n \). Each vertex \( b_i \) in \( B_2 \) corresponds to a query vertex \( v_i \), \( i = 0, ..., n \). We introduce an edge between \( a_i \) and \( b_i \) if \( f_i(v) \) has an internal vertex that is matching query vertex \( v_i \). Let a variable \( x_{ij} \) denote the edge label of the edge \( a_i \rightarrow b_j \). Let us recall all local partial matches in Figure 4.

Figure 9 shows an example bipartite graph.

We define the 0-1 integer planning problem as follows:

\[
\min \prod_{i=0}^{n} \left( \sum_{j} x_{ij} + 1 \right) \\
st. \sum_{j} x_{ij} = 1
\]

The above equation means that each local partial match should be assigned to only one query vertex.

It is easy to know the equivalence between the 0-1 integer planning and finding the optimal partition. The former is a classical NP-complete problem. Thus, we know the theorem holds.

Theorem 5.3 Given a query graph \( Q \) with \( n \) vertices \( \{v_1,...,v_n\} \) and a set of all local partial matches \( \Omega \), let \( U_v \) (\( i = 1, ..., n \)) be all local partial matches (in \( \Omega \)) that have internal vertices matching \( v_i \). For the optimal partitioning \( P_{\text{opt}} = \{P_{v_1},...,P_{v_n}\} \) where \( P_{v_i} \) has the largest partition size (i.e., the number of local partial matches in \( P_{v_i} \) is maximum) in \( P_{\text{opt}} \), \( U_v = U_{v_i} \).

Proof. (by contradiction) Assume that \( P_{v_i} \neq U_v \) in the optimal partitioning \( P_{\text{opt}} = \{P_{v_1},...,P_{v_n}\} \). Then, there exists a local

Claim (1), edge \( vv' \) should also be matched in \( PM \), since \( PM \) is a local partial match. However, edge \( uu' \) (matching \( vv' \)) does not exist in \( PM \). This contracts \( PM \) being a local partial match. Thus, Theorem 4.3 holds.
partial match \( PM \notin P_{yj} \) and \( PM \in U_{vy} \). We assume that \( PM \in P_{yj} \), \( j \neq n \). The cost of \( \mathcal{P}_{opt} = \{ P_{y1}, ..., P_{yn} \} \) is:

\[
\text{Cost}(\mathcal{P}_{opt}) = \left( \prod_{1 \leq i \leq n} |P_{yi}| + 1 \right) \times |P_{yj}| + 1 \times (|P_{yn}| + 1)
\]

(5)

Since \( PM \in U_{vy} \), \( PM \) has an internal vertex matching \( v_{vy} \). Hence, we can also put \( PM \) into \( P_{yj} \). Then, we get a new partitioning \( \mathcal{P}' = \{ P_{y1}, ..., P_{yj} - \{ PM \}, ..., P_{yn} \cup \{ PM \} \} \). The cost of the new partitioning is:

\[
\text{Cost}(\mathcal{P}) = \left( \prod_{1 \leq i \leq n} |P_{yi}| + 1 \right) \times |P_{yj}| \times (|P_{yn}| + 2)
\]

(6)

Let \( C = \prod_{1 \leq i \leq n} (|P_{yi}| + 1) \), which exists in both Equations 5 and 6. Obviously, \( C > 0 \).

\[
\text{Cost}(\mathcal{P}_{opt}) - \text{Cost}(\mathcal{P}) = C \times (|P_{yi}| + 1) \times (|P_{yj}| - C) \times (|P_{yn}| + 2)
\]

Because \( P_{yn} \) is the largest partition in \( \mathcal{P}_{opt} \), \( |P_{yi}| + 1 - |P_{yj}| > 0 \). Furthermore, \( C > 0 \). Hence, \( \text{Cost}(\mathcal{P}_{opt}) - \text{Cost}(\mathcal{P}) > 0 \), meaning that the optimal partitioning has larger cost. Obviously, this cannot happen.

Therefore, in the optimal partitioning \( \mathcal{P}_{opt} \), we cannot find a local partial match \( PM \), where \( |P_{yi}| \) is the largest, \( PM \notin P_{yi} \) and \( PM \in U_{vy} \). In other words, \( P_{yn} = U_{vn} \) in the optimal partitioning. □

B. QUERIES IN EXPERIMENTS

B.1 LUBM Queries


Q₂: Select ?x, ?y, ?z where { ?x rdf:type ub:Course. ?x ub:name ?y. }


B.2 BTC Queries


Q₅: Select ?d where { ?a <http://dbpedia.org/property/education> ?d. }

Q₆: Select ?d where { ?a <http://dbpedia.org/property/nationality> ?d. }

B.3 LOD Queries


Q₂: Select ?a where { ?a <http://www.mpii.de/yago/resource/wordnet_scientist_110560637> }

Q₃: Select ?a where { ?a <http://www.mpii.de/yago/resource/wasBornIn> <http://www.mpii.de/yago/resource/Chicago> }

Q₄: Select ?a where { ?a <http://www.mpii.de/yago/resource/wordnet_actor_109765278> }


Q₈: Select ?a where { ?a <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat. ?a <http://www.w3.org/2003/01/geo/wgs84_pos#long> }


Q6: Select distinct ?n1, ?n2 where {
"f <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Film>.
"person1 <http://www.mpii.de/yago/resource/isMarriedTo> ?person2.
"person2 <http://www.mpii.de/yago/resource/hasPreferredName> ?n2.
"f <http://data. linkedmdb.org/resource/movie/producer> ?person1.
"person1 <http://data. linkedmdb.org/resource/movie/producer_name> ?n1.
}

B.4 Queries with Different Diameters in Experiment 1

<table>
<thead>
<tr>
<th>Diameter</th>
<th>Queries</th>
</tr>
</thead>
</table>
| 1        | Select ?x, ?y where {
| 2        | Select ?x, ?y where {
| 3        | Select ?x, ?y where {
|          | ?x ub:undergraduateDegreeFrom ?y.
|          | ?y ub:name University0. |
| 4        | Select ?x, ?y where {
|          | ?x ub:undergraduateDegreeFrom ?y.
|          | ?y ub:name University0. |

Table 6: Queries with Different Diameters