Noname manuscript No.
(will be inserted by the editor)

Processing SPARQL Queries Over Distributed RDF Graphs

Peng Peng · Lei Zou · M. Tamer Özsu · Lei Chen · Dongyan Zhao

the date of receipt and acceptance should be inserted later

Abstract We propose techniques for processing SPARQL queries over a large RDF graph in a distributed environment. We adopt a "partial evaluation and assembly" framework. Answering a SPARQL query Q is equivalent to finding subgraph matches of the query graph Q over RDF graph G. Based on properties of subgraph matching over a distributed graph, we introduce *local partial match* as partial answers in each fragment of RDF graph G. For assembly, we propose two methods: centralized and distributed assembly. We analyze our algorithms from both theoretically and experimentally. Extensive experiments over both real and benchmark RDF repositories of billions of triples confirm that our method is superior to the state-of-the-art methods in both the system's performance and scalability.

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. Hence, an RDF dataset can be represented as a graph where subjects and objects are vertices, and triples are edges with property names as edge labels. With the increasing amount of RDF data published on the Web, system performance and scalability issues have become increasingly pressing. For example, LOD (Linking Open Data) project builds a RDF data cloud by linking more than 3000 datasets, which currently have more than 84 billion triples [1]. The recent work [40] shows that the number of data sources has doubled within three years (2011-2014). Obviously, the computational and storage requirements coupled with rapidly growing datasets have stressed the limits of single machine processing.

There have been a number of recent efforts in distributed evaluation of SPARQL queries over large RDF datasets [20]. We broadly classify these solutions into three categories: *cloud-based*, *partition-based*, and *federated* approaches. These are discussed in detail in Section 2; the highlights are as follows.

Cloud-based approaches (e.g., [27,37,23,49,48,33,34]) maintain a large RDF graph using existing cloud computing platforms, such as Hadoop (http://hadoop.apache.org) or Cassandra (http://cassandra.apache.org), and employ triple pattern-based join processing most commonly using MapReduce.

Partition-based approaches [22,21,15,28,29,18] divide the RDF graph G into a set of subgraphs (fragments) $\{F_i\}$, and decompose the SPARQL query Q into subqueries $\{Q_i\}$. These subqueries are then executed over the partitioned data using techniques similar to relational distributed databases.

Federated SPARQL processing systems [36,19,16,38,39] evaluate queries over multiple SPARQL endpoints. These systems typically target LOD and follow a query processing

Peng Peng, Lei Zou, Dongyan Zhao
Institute of Computer Science and Technology,
Peking University, Beijing, China
Tel.: +86-10-82529643
E-mail: {pku09pp,zoulei,zhaody}@pku.edu.cn

M. Tamer Özsu
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Canada
Tel.: +1-519-888-4043
E-mail: Tamer.Ozsu@uwaterloo.ca

Lei Chen
Department of Computer Science and Engineering,
Hong Kong University of Science and Technology,
Hong Kong, China
Tel.: +852-23586980
E-mail: leichen@cse.ust.hk

[1] The statistic is reported in http://stats.lod2.eu/

over data integration approach.These systems operate in a very different environment we are targeting, since we focus on exploiting distributed execution for speed-up and scalability.

In this paper we propose an alternative strategy that is based on only partitioning the data graph but not decomposing the query. Our approach is based on the "partial evaluation and assembly" framework [24]. An RDF graph is partitioned using some graph partitioning algorithm such as METIS [26] into vertex-disjoint fragments (edges that cross fragments are replicated in source and target fragments). Each site receives the full SPARQL query Q and executes it on the local RDF graph fragment providing data parallel computation. To the best of our knowledge, this is the first work that adopts the partial evaluation and assembly strategy to evaluate SPARQL queries over a distributed RDF data store. The most important advantage of this approach is that the number of involved vertices and edges in the intermediate results are minimized, which is proven theoretically (see Proposition 3 in Section 4).

The basic idea of the partial evaluation strategy is the following: given a function $f(s,d)$, where s is the known input and d is the yet unavailable input, the part of f's computation that depends only on s generates a partial answer. In our setting, each site S_i treats fragment F_i as the known input in the partial evaluation stage; the unavailable input is the rest of the graph ($\overline{G} = G \setminus F_i$). The partial evaluation technique has been used in compiler optimization [24], and querying XML trees [7]. Within the context of graph processing, the technique has been used to evaluate reachability queries [13], and graph simulation [31,14] over graphs. However, SPARQL query semantics is different than these — SPARQL is based on graph homomorphism [35] — and pose additional challenges. Graph simulation defines a *relation* between vertices in the query graph Q (i.e. $V(Q)$) and that in the data graph G (i.e., $V(G)$), but, graph homomorphism is a *function* (not a relation) between $V(Q)$ and $V(G)$ [14]. Thus, the solutions proposed for graph simulation [14] and graph pattern matching [31] cannot be applied to the problem studied in this paper.

Because of interconnections between graph fragments, application of graph homomorphism over graphs requires special care. For example, consider a distributed RDF graph in Figure 1. 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 $s1$. Here, the prefix is just for simplifying presentation, not a general assumption made by the approach. There are *crossing 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 $s1$).

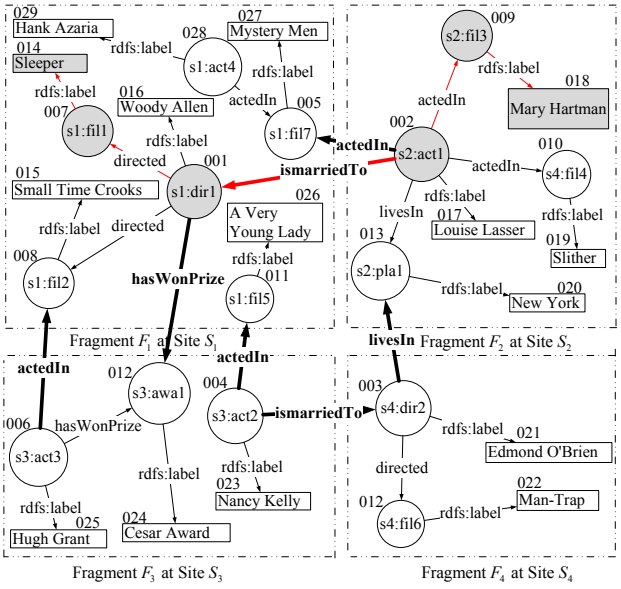


Fig. 1 A Distributed RDF Graph

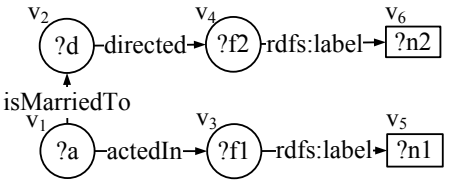


Fig. 2 SPARQL Query Graph Q

Now consider the following SPARQL query Q that consists of five triple patterns (e.g., ?a isMarriedTo ?d) over this distributed RDF graph:

```
SELECT ?a ?d WHERE
{?a isMarriedTo ?d. ?a actedIn ?f1.
?f1 rdfs:label ?n1. ?d directed ?f2.
?f2 rdfs:label ?n2.}
```

Some SPARQL query matches are contained within a fragment, which we call *inner matches*. 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 query Q in Figure 2, 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 1 (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 re-

sults at each site given a query graph Q (i.e., the *local partial match*), which, intuitively, is 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).

The main benefits of our solution are twofold:

- Our solution does not depend on any specific partitioning strategy. In existing partition-based methods, the query processing always depends on a certain RDF graph partitioning strategy, which may be difficult to enforce in certain circumstances. The partition-agnostic framework enables us to adopt any partition-based optimization, although this is orthogonal to our solution in this paper.
- Our method guarantees to involve fewer vertices or edges in intermediate results than other partition-based solutions, which we prove in Section 4 (Proposition 3). This property often results in smaller number of intermediate results and lowers the cost of our approach, which we demonstrate experimentally in Section 7.

The rest of the paper is organized as follows. We discuss related work in the areas of distributed SPARQL query processing and partial query evaluation in Section 2. Section 3 provides the fundamental definitions that form the background for this work and introduces the overall execution framework. Computation of local matches at each site is covered in Section 4 and the centralized and distributed assembly of partial results to compute the final query result is discussed in Section 5. We also study how to evaluate general SPARQLs in Section 6. We evaluate our approach, both in terms of its internal characteristics and in terms of its relative performance against other approaches in Section 7. Section 8 concludes the paper and outlines some future research directions.

2 Related Work

Distributed SPARQL Query Processing. As noted above, there are three general approaches to distributed SPARQL query processing: *cloud-based* approaches, *partition-based* approaches and *federated SPARQL query* systems.

(1) Cloud-based Approaches

There have been a number of works (e.g., [27,37,23,49, 48,47,33,34]) focused on managing large RDF datasets using existing cloud platforms; a very good survey of these is [25]. Many of these approaches follow the MapReduce paradigm; in particular they use HDFS [37,23,49,48], and store RDF triples in flat files in HDFS. When a SPARQL query is issued, the HDFS files are scanned to find the matches of each triple pattern, which are then joined using one of the MapReduce join implementations (see [30] for more detailed description of these). The most important difference among these approaches is how the RDF triples are stored in HDFS files; this determines how the triples are accessed and the number of MapReduce jobs. In particular, SHARD [37] directly stores the data in a single file and each line of the file represents all triples associated with a distinct subject. HadoopRDF [23] and PredicateJoin [49] further partition RDF triples based on the predicate and store each partition within one HDFS file. EAGRE [48] first groups all subjects with similar properties into an entity class, and then constructs a compressed RDF graph containing only entity classes and the connections between them. It partitions the compressed RDF graph using the METIS algorithm [26]. Entities are placed into HDFS according to the partition set that they belong to.

Besides the HDFS-based approaches, there are also some works that use other NoSQL distributed data stores to manage RDF datasets. JenaHBase [27] and H_2RDF [33,34] use some permutations of subject, predicate, object to build indices that are then stored in HBase (http://hbase.apache.org). Trinity.RDF [47] uses the distributed memory-cloud graph system Trinity [44] to index and store the RDF graph. It uses hashing on the vertex values to obtain a disjoint partitioning of the RDF graph that is placed on nodes in a cluster.

These approaches benefit from the high scalability and fault-tolerance offered by cloud platforms, but may suffer lower performance due to the difficulties of adapting MapReduce to graph computation.

(2) Partition-based Approaches

The partition-based approaches [22,21,15,28,29,18] partition an RDF graph G into several fragments and place each at a different site in a parallel/distributed system. Each site hosts a centralized RDF store of some kind. At run time, a SPARQL query Q is decomposed into several subqueries such that each subquery can be answered locally at one site, and the results are then agregated. Each of these papers proposes its own data partitioning strategy, and different partitioning strategies result in different query processing methods.

In GraphPartition [22], 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. WARP [21] uses some frequent structures in workload to further extend the results of GraphPartition. Partout [15] extends the concepts of minterm predicates in relational database systems, and uses the results of minterm predicates as the fragmentation units. Lee et. al. [28,29] 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 uses METIS [26] 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. At each site, TriAD maintains six large, in-memory vectors of triples, which correspond to all SPO permutations of triples. Meanwhile, TriAD constructs a summary graph to maintain the partitioning information.

All of the above methods require partitioning and distributing the RDF data according to specific requirements of their approaches. However, in some applications, the RDF repository partitioning strategy is not controlled by the distributed RDF system itself. There may be some administrative requirements that influence the data partitioning. For example, in some applications, the RDF knowledge bases are partitioned according to topics (i.e., different domains), or are partitioned according to different data contributors. Therefore, partition-tolerant SPARQL processing may be desirable. This is the motivation of our partial-evaluation and assembly approach.

As well, these approaches evaluate the SPARQL query based on query decomposition, which generate more intermediate results. We provide a detailed experimental comparison in Section 7.

(3) Federated SPARQL Query Systems

Federated queries run SPARQL queries over multiple SPARQL endpoints. A typical example is linked data, where different RDF repositories are interconnected, providing a *virtually integrated distributed database*. Federated SPARQL query processing is a very different environment than what we target in this paper, but we discuss these systems for completeness.

A common technique is to precompute metadata for each individual SPARQL endpoints. Based on the metadata, the original SPARQL query is decomposed into several subqueries, where each subquery is sent to its relevant SPARQL endpoints. The results of subqueries are then joined together to answer the original SPARQL query. In DARQ [36], the metadata is called *service description* that describes which triple patterns (i.e., predicate) can be answered. In [19], the metadata is called Q-Tree, which is a variant of RTree. Each leaf node in Q-Tree stores a set of source identifers, including one for each source of a triple approximated by the node. SPLENDID [16] uses Vocabulary of Interlinked Datasets (VOID) as the metadata. HiBISCuS [38] relies on capabilities to compute the metadata. For each source, HiBISCuS defines a set of capabilities which map the properties to their subject and object authorities. TopFed [39] is a biological federated SPARQL query engine. Its metadata comprises of an N3 specification file and a Tissue Source Site to Tumour (TSS-to-Tumour) hash table, which is devised based on the data distribution.

In contrast to these, FedX [42] does not require preprocessing, but sends "SPARQL ASK" to collect the metadata on the fly. Based on the results of "SPARQL ASK" queries, it decomposes the query into subqueries and assign subqueries with relevant SPARQL endpoints.

Global query optimization in this context has also been studied. Most federated query engines employ existing optimizers, such as dynamic programming [3], for optimizing the join order of local queries. Furthermore, DARQ [36] and FedX [42] discuss the use of semijoins to compute a join between intermediate results at the control site and SPARQL endpoints.

Partial Evaluation. Partial evaluation has been used in many applications ranging from compiler optimization to distributed evaluation of functional programming languages [24]. Recently, partial evaluation has also been used for evaluating queries on distributed XML trees and graphs [6–8, 13]. In [6–8], 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 [7] or bottom-up [6] 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 [13] study reachability query processing over distributed graphs using the partial evaluation strategy. Partial evaluation-based graph simulation is well studied by Fan et al. [14] and Shuai et al. [31]. However, SPARQL query semantics is based on graph homomorphism [35], not graph simulation. The two concepts are formally different (i.e., they produce different results) and the two problems have very different complexities. Homomorphism defines a "function" while simulation defines a "relation" – relation allows "one-to-many" mappings while function does not. Consequently, the results are different. The computational hardness of the two problems are also different. Graph homomorphism is a classical NP-complete problem [11], while graph simulation has a polynomial-time algorithm ($O((|V(G)|+|V(Q)|)(|E(G)|+|E(Q)|))$) [12], where $|V(G)|$ ($|V(Q)|$) and $|E(G)|$ ($|E(Q)|$) denote the number of vertices and edges in RDF data graph G (and query graph Q). Thus, the solutions based on graph simulation cannot be applied to the problem studied in this paper. To the best of our

knowledge, there is no prior work in applying partial evaluation to SPARQL query processing.

3 Background and Framework

An RDF dataset can be represented as a graph where subjects and objects are vertices and triples are labeled edges.

Definition 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 multiset of directed edges that correspond to all triples in RDF data; Σ 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. In this paper, we first focus on basic graph pattern (BGP) queries as they are foundational to SPARQL, and focus on techniques for handling these. We extend this discussion in Section 6 to general SPARQL queries involving FILTER, UNION, and OPTIONAL.

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.

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) of Q over RDF graph G.

Definition 3 (SPARQL Match) Consider an RDF graph G and a connected query graph Q that has n vertices $\{v_1, ..., v_n\}$. A subgraph M with m vertices $\{u_1, ..., u_m\}$ (in G) is said to be a *match* of Q if and only if there exists a *function* f from $\{v_1, ..., v_n\}$ to $\{u_1, ..., u_m\}$ $(n \geq m)$, where the following conditions hold:

1. if v_i is not a variable, $f(v_i)$ and v_i have the same URI or literal value $(1 \leq i \leq n)$;
2. if v_i is a variable, there is no constraint over $f(v_i)$ except that $f(v_i) \in \{u_1, ..., u_m\}$;
3. if there exists an edge $\overrightarrow{v_i v_j}$ in Q, there also exists an edge $\overrightarrow{f(v_i)f(v_j)}$ in G. Let $L(\overrightarrow{v_i v_j})$ denote a multi-set of labels between v_i and v_j in Q; and $L(\overrightarrow{f(v_i)f(v_j)})$ denote a multi-set of labels between $f(v_i)$ and $f(v_j)$ in G. There must exist an *injective function* from edge labels in $L(\overrightarrow{v_i v_j})$ to edge labels in $L(\overrightarrow{f(v_i)f(v_j)})$. Note that a variable edge label in $L(\overrightarrow{v_i v_j})$ can match any edge label in $L(\overrightarrow{f(v_i)f(v_j)})$.

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, an RDF graph G is vertex-disjoint partitioned into a number of *fragments*, each of which resides at one site. The vertex-disjoint partitioning has been used in most distributed RDF systems, such as GraphPartition [22], EAGRE [48] and TripleGroup [28]. Different distributed RDF systems utilize different vertex-disjoint partitioning algorithms, and the partitioning algorithm is orthogonal to our approach. Any vertex-disjoint partitioning method can be used in our method, such as METIS [26] and MLP [46].

The vertex-disjoint partitioning methods guarantee that there are no overlapping vertices between fragments. However, to guarantee data integrity and consistency, we store some replicas of crossing edges. Since the RDF graph G is partitioned by our system, metadata is readily available regarding crossing edges (both outgoing and incoming edges) and the endpoints of crossing edges. Formally, we define the *distributed RDF graph* as follows.

Definition 4 (Distributed RDF Graph) A distributed RDF graph $G = \{V, E, \Sigma\}$ consists of a set of fragments $\mathcal{F} = \{F_1, F_2, ..., F_k\}$ where each F_i is specified by $(V_i \cup V_i^e, E_i \cup E_i^c, \Sigma_i)$ $(i = 1, ..., k)$ such that

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

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

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

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

5. Vertices in V_i^e are called *extended* vertices of F_i and all 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 Figure 1 shows a distirbuted RDF graph G consisting of four fragments F_1, F_2, F_3 and F_4. The numbers besides the vertices are vertex IDs that are introduced for ease of presentation. In Figure 1, $\overrightarrow{002, 001}$ is a crossing edge between F_1 and F_2. As well, edges $\overrightarrow{004, 011}$, $\overrightarrow{001, 012}$ and

$\overrightarrow{006, 008}$ are crossing edges between F_1 and F_3. Hence, V_1^e = {002, 006, 012, 004} and E_1^c = {$\overrightarrow{002, 001}$, $\overrightarrow{004, 011}$, $\overrightarrow{001, 012}$, $\overrightarrow{006, 008}$}.

Definition 5 (**Problem Statement**) Let G be a distributed RDF graph that consists of a set of fragments $\mathcal{F} = \{F_1, \ldots, F_k\}$ and let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be a set of computing nodes such that F_i is located at S_i. Given a SPARQL query graph Q, our goal is to find all *SPARQL matches* of Q in G.

Note that for simplicity of exposition, we are assuming that each site hosts one fragment. Inner matches can be computed locally using a centralized RDF triple store, such as RDF-3x [32], SW-store [1] or gStore [50]. In our prototype development and experiments, we modify gStore, a graph-based SPARQL query engine [50], to perform partial evaluation. The main issue of answering SPARQL queries over the distributed RDF graph is finding crossing matches efficiently. That is a major focus of this paper.

Example 2 Given a SPARQL query graph Q in Figure 2, the subgraph induced by vertices 014,007,001,002,009 and 018 (shown in the shaded vertices and the red edges in Figure 1) is a crossing match of Q.

We utilize a *partial evaluation and assembly* [24] framework to answer SPARQL queries over a distributed RDF graph G. Each site S_i treats fragment F_i as the known input s and other fragments as yet unavailable input $\overline{G}$ (as defined in Section 1) [13].

In our execution model, each site S_i receives the full query graph Q. In the partial evaluation stage, at each site S_i, we find all *local partial matches* (Definition 6) of Q in F_i. We prove that an overlapping part between any crossing match and fragment F_i must be a local partial match in F_i (see Proposition 1).

To demonstrate the intuition behind dealing with crossing edges, consider the case in Example 2. The crossing match M overlaps with two fragments F_1 and F_2. If we can find the overlapping parts between M and F_1, and M and F_2, we can assemble them to form a crossing match. For example, the subgraph induced by vertices 014, 007, 001 and 002 is an overlapping part between M and F_1. Similarly, we can also find the overlapping part between M and F_2. We assemble them based on the common edge $\overrightarrow{002, 001}$ to form a crossing match, as shown in Figure 3.

In the assembly stage, these local partial matches are assembled to form crossing matches. In this paper, we consider two assembly strategies: centralized and distributed (or parallel). In centralized, all local partial matches are sent to a single site for the assembly. In distributed/parallel, local partial matches are combined at a number of sites in parallel (see Section 5).

There are three steps in our method.

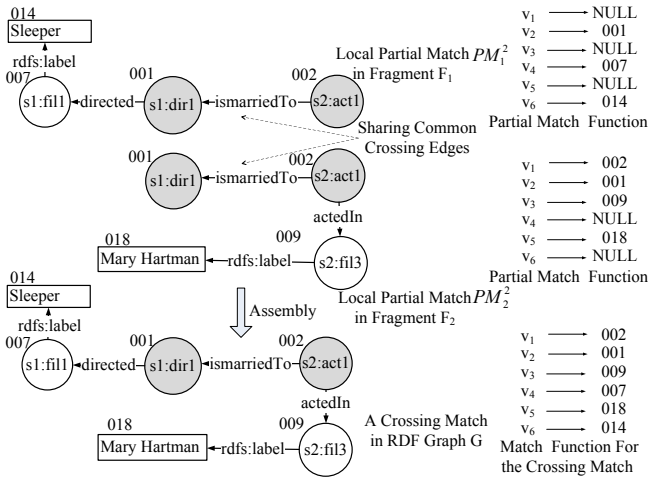


Fig. 3 Assemble Local Partial Matches

Step 1 (*Initialization*): A SPARQL query Q is input and sent to each site in $\mathcal{S}$.

Step 2 (*Partial Evaluation*): Each site S_i finds *local partial matches* of Q over fragment F_i. This step is executed in parallel at each site (Section 4).

Step 3 (*Assembly*): Finally, we assemble all local partial matches to compute complete crossing matches. The system can use the centralized (Section 5.2) or the distributed assembly approach (Section 5.3) to find crossing matches.

4 Partial Evaluation

We first formally define 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 query Q, each site S_i computes the partial answers (called *local partial matches*) based on the known input F_i (recall that, for simplicity of exposition, we assume that each site hosts one fragment as indicated by its subscript). Intuitively, a local partial match PM_i is an overlapping part between a crossing match M and fragment F_i at the partial evaluation stage. Moreover, M may or may not exist depending on the yet unavailable input $\overline{G}$. Based only on the known input 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 1 is a local partial match between M and F_1.

Definition 6 (Local Partial Match) Given a SPARQL query graph Q with n vertices $\{v_1, ..., v_n\}$ and a connected sub-

graph PM with m vertices $\{u_1, ..., u_m\}$ ($m \leq n$) in a fragment F_k, PM is a *local partial match* in fragment F_k if and only if there exists a function $f : \{v_1, ..., v_n\} \rightarrow \{u_1, ..., u_m\} \cup \{NULL\}$, 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 $\overrightarrow{v_i v_j}$ in Q ($1 \leq i \neq j \leq n$), then PM should meet one of the following five conditions: (1) 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 $\overrightarrow{v_i v_j}$; (2) 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; (3) 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; (4) $f(v_i) = NULL$; (5) $f(v_j) = NULL$.
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 in F_k) and $\exists \overrightarrow{v_i v_j} \in Q$ (or $\overrightarrow{v_j v_i} \in Q$), there must exist $f(v_j) \neq NULL$ and $\exists \overrightarrow{f(v_i)f(v_j)} \in PM$ (or $\exists \overrightarrow{f(v_j)f(v_i)} \in PM$). Furthermore, if $\overrightarrow{v_i v_j}$ (or $\overrightarrow{v_j v_i}$) has a property p, $\overrightarrow{f(v_i)f(v_j)}$ (or $\overrightarrow{f(v_j)f(v_i)}$) has the same property p.
6. Any two vertices v_i and v_j (in query Q), where $f(v_i)$ and $f(v_j)$ are both internal vertices in PM, are *weakly connected* (see Definition 7) in Q.

Vector $[f(v_1), ..., f(v_n)]$ is a serialization of a local partial match.

Example 3 Given a SPARQL query Q with six vertices in Figure 2, the subgraph induced by vertices 001, 002, 007 and 014 (shown in shaded circles and red edges) is a *local partial match* of Q in fragment F_1. The function is $\{(v_1, 002), (v_2, 001), (v_3, NULL), (v_4, 007), (v_5, NULL), (v_6, 014)\}$. The five different local partial matches in F_1 are shown in Figure 4.

Definition 6 formally defines a *local partial match*, which is a subset of a complete SPARQL match. Therefore, some conditions in Definition 6 are analogous to SPARQL match with some subtle differences. In Definition 6, some vertices of query Q are not matched in a local partial match. They are allowed to match a special value NULL (e.g., v_3 and v_5 in Example 3). 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 query Q) is matched to an internal vertex, all of v_i's neighbours should be matched in this local partial match as well. The following example illustrates the intuition.

Example 4 Let us recall the local partial match PM_1^2 of Fragment F_1 in Figure 4. An internal vertex 001 in fragment F_1 is matched to vertex v_2 in query Q. Assume that PM is an overlapping part between a crossing match M and fragment F_1. Obviously, v_2's neighbors, such as v_1 and v_4, should also be matched in M. Furthermore, the matching vertices should be 001's neighbors. Since 001 is an internal vertex in F_1, 001's neighbors are also in fragment F_1.

Therefore, if a PM violates Condition 5, it 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 7 Two vertices are *weakly connected* in a directed graph if and only if there exists a connected path between the two vertices when all directed edges are replaced 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 Propositions 1 and 2. The following example shows all local partial matches in the running example.

Example 5 Given a query Q in Figure 2 and an RDF graph G in Figure 1, Figure 4 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 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 propositions.

1. The overlapping part between any crossing match M and internal vertices of fragment F_i ($i = 1, ..., k$) must be a local partial match (see Proposition 1).
2. Missing any local partial match may lead to result dismissal. Thus, the algorithm should find all local partial matches in each fragment (see Proposition 2).
3. It is impossible to find two local partial matches M and M' in fragment F, where M' is a subgraph of M, i.e., each local partial match is maximal (see Proposition 4).

Proposition 1 *Given any crossing match M of SPARQL query Q in an RDF graph G, if M overlaps with some fragment F_i, let $(M \cap F_i)$ denote the overlapping part between M and fragment F_i. Assume that $(M \cap F_i)$ consists of several weakly connected components, denoted as $(M \cap F_i) = \{PM_1, ..., PM_n\}$. Each weakly connected component PM_a ($1 \leq a \leq n$) in $(M \cap F_i)$ must be a* local partial match *in fragment F_i.*

Proof (1) Since PM_a ($1 \leq a \leq n$) is a subset of a SPARQL match, it is easy to show that Conditions 1-3 of Definition 6 hold.

(2) We prove that each weakly connected component PM_a ($1 \leq a \leq n$) must have at least one crossing edge (i.e., Condition 4) as follows.

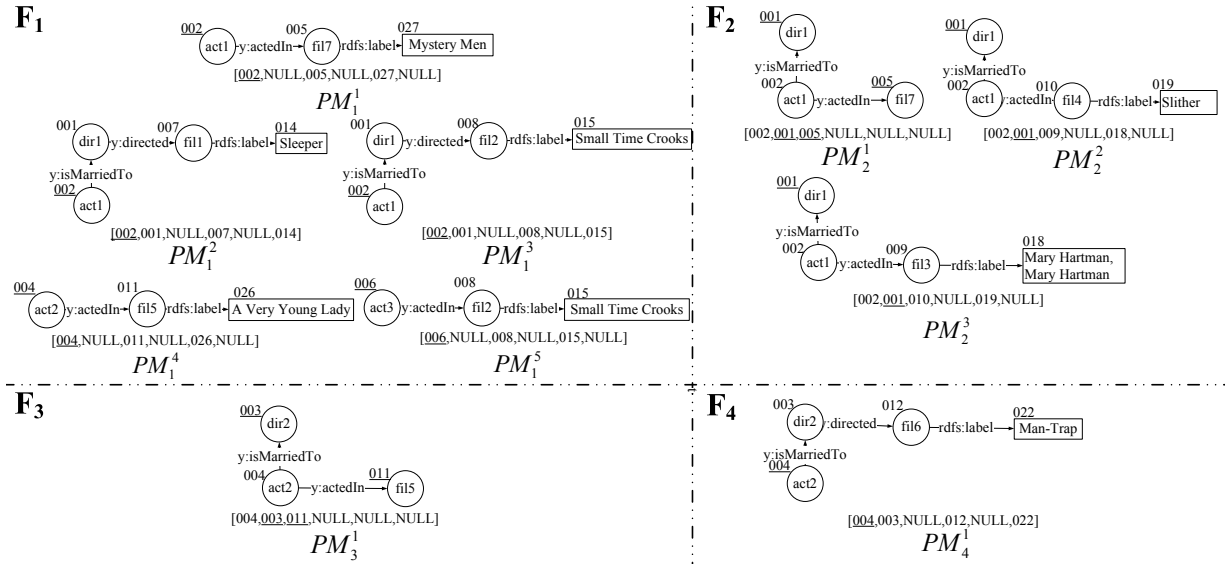


Fig. 4 Local Partial Matches of Q in Each Fragment

Since M is a crossing match of SPARQL query Q, M must be weakly connected, i.e., any two vertices in M are weakly connected. Assume that $(M \cap F_i)$ consists of several weakly connected components, denoted as $(M \cap F_i) = \{PM_1, ..., PM_n\}$. Let $M = (M \cap F_i) + \overline{(M \cap F_i)}$, where $\overline{(M \cap F_i)}$ denotes the complement of $(M \cap F_i)$. It is straightforward show that $\overline{(M \cap F_i)}$ must occur in other fragments; otherwise it should be found at $(M \cap F_i)$. PM_a $(1 \leq a \leq n)$ is weakly disconnected with each other because we remove $\overline{(M \cap F_i)}$ from M. In other words, each PM_a must have at least one *crossing edge* to connect PM_a with $\overline{(M \cap F_i)}$. $\overline{(M \cap F_i)}$ are in other fragments and only crossing edges can connect fragment F_i with other fragments. Otherwise, PM_a is a separated part in the crossing match M. Since, M is weakly connected, PM_a has at least one crossing edge, i.e, Condition 4.

(3) For Condition 5, for any internal vertex u in PM_a $(1 \leq a \leq n)$, PM_a retains all its incident edges. Thus, we can prove that Condition 5 holds.

(4) We define PM_a $(1 \leq a \leq n)$ as a weakly connected part in $(M \cap F_i)$. Thus, Condition 6 holds.

To summarize, the overlapping part between M and fragment F_i satisfies all conditions in Definition 6. Thus, Proposition 1 holds. □

Let us recall Example 5. There are some local partial matches that do not contribute to any crossing match, such as PM_1^5 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_1^5 is a false positive. Formally, we have the following proposition, stating that we have to find all local partial matches in each fragment F_i in the partial evaluation stage.

Proposition 2 *The partial-evaluation-and-assembly algorithm does not miss any crossing matches in the answer set* 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_i but a cross match M is missed in the answer set. Since M is a crossing match, suppose that M overlaps with m fragments F_1,...,F_m. According to Proposition 1, the overlapping part between M and F_i $(i = 1, ..., m)$ must be a local partial match PM_i in F_i. According to the assumption, these local partial matches have been found in the partial evaluation stage. Obviously, we can assemble these partial matches PM_i $(i = 1, ..., m)$ 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 If" part: (proven by contradiction).

We assume that a local partial match PM_i in fragment F_i is missed and the answer set can still satisfy no-false-negative requirement. Suppose that PM_i matches a part of Q, denoted as Q'. Assume that there exists another local partial match PM_j in F_j that matches a complementary graph of Q', denoted as $\overline{Q} = Q \setminus Q'$. In this case, we can obtain a complete match M by assembling the two local partial matches. If PM_i in F_i is missed, then match M is missed. In other words, it cannot satisfy the no-false-negative requirement. This also contradicts the assumption. □

Proposition 2 guarantees that no local partial matches will be missed. This is important to avoid false negatives. Based on Proposition 2, we can further prove the following proposition, which guarantees that the intermediate results in our method involve the smallest number of vertices and edges.

Proposition 3 *Given the same underlying partitioning over RDF graph G, the number of involved vertices and edges in the intermediate results (in our approach) is not larger than that in any other partition-based solution.*

Proof In Proposition 2, we prove that every local partial match should be found for result completeness (i.e., false negatives). The same proposition proves that our method produces complete results. Therefore, if a partition-based solution omits some of the partial matches (i.e., intermediate results) that are in our solution (i.e., has intermediate result smaller than ours) then it cannot produce complete results. Assuming that they all produce complete results, what remains to be proven is that our set of partial matches is a subset of those generated by other partition-based solutions. We prove that by contradiction.

Let A be a solution generated by an alternative partition-based approach. Assume that there exists one vertex u in a local partial match PM produced by our method, but u is not in the intermediate results of the partition-based solution A. This would mean that during the assembly phase to produce the final result, any edges adjacent to u will be missed. This would produce incomplete answer, which contradicts the completeness assumption.

Similarly, it can be argued that it is impossible that there exists an *edge* in our local partial matches (i.e., intermediate results) that it is not in the intermediate results of other partition-based approaches.

In other words, all vertices and edges in local partial matches must occur in the intermediate results of other partition-based approaches. Therefore, Proposition 3 holds. □

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

Proposition 4 *Given a query graph Q and an RDF graph G, if PM_i is a local partial match under function f in fragment F_i, there exists no local partial match PM'_i under function f' in F_i, where $f \subset f'$.*

Proof (by contradiction) Assume that there exists another local partial match PM'_i of query Q in fragment F_i, where PM_i is a subgraph of PM'_i. Since PM_i is a subgraph of PM'_i, there must exist at least one edge $e = \overrightarrow{uu'}$ where $e \in PM'_i$ and $e \notin PM_i$. Assume that $\overrightarrow{uu'}$ is matching edge $\overrightarrow{vv'}$ in query Q. Obviously, at least one endpoint of e should be an internal vertex. We assume that u is an internal vertex. According to Condition (8) of Definition 6 and Claim (1), edge $\overrightarrow{vv'}$ should also be matched in PM, since PM is a local partial match. However, edge $\overrightarrow{uu'}$ (matching $\overrightarrow{vv'}$) does not exist in PM. This contracts PM being a local partial match. Thus, Proposition 4 holds. □

4.2 Computing Local Partial Matches

Given a SPARQL query Q and a fragment F_i, the goal of partial evaluation is to find all local partial matches (according to Definition 6) 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) ($v \in Q$ 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 6.

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 6, 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) ($v \in Q$ and $u \in F_i$), we start with an empty set. In each step, we introduce a candidate vertex pair (v, u) 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 6, 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 Ω(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 Ω(F_i);
```

At each step, a new candidate vertex pair (v', u') is added to an existing function f to form a new function f'. The order of selecting the query vertex can be arbitrarily defined. However, QuickSI [43] proposes several heuristic rules to

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
4 **for** *each candidate vertex u′ with regard to v′* **do**
5 | $f' \leftarrow f \cup (v', u')$
6 | **if** *f′ violates any condition (except for condition 4 and 5 of Definition 6)* **then**
7 | | Continue
8 | **if** *f′ satisfies all conditions (except for condition 4 and 5 of Definition 6)* **then**
9 | | **ComParMatch**(f')
10 | **if** *f′ satisfies all conditions of Definition 6* **then**
11 | | f specifies a local partial match PM that will be inserted into the answer set $\Omega(F_i)$

select an optimized order that can speed up the matching process. These rules are also utilized in our experiments.

To compute local partial matches (Algorithm 1), we revise a graph-based SPARQL query engine, gStore, which is our previous work. Since gStore adopts "subgraph matching" technique to answer SPARQL query processing, it is easy to revise its subgraph matching algorithm to find "local partial matches" in each fragment. gStore adopts a state transformation technique to find SPARQL matches. Here, a state corresponds to a partial match (i.e. a function from Q to G).

Our *state transformation* algorithm is as follows. Assume that v matches vertex u in SPARQL query Q. We first initialize a state with v. Then, we search the RDF data graph for v's neighbor v' corresponding to u' in Q, where u' is one of u's neighbors and edge $\overrightarrow{vv'}$ satisfies query edge $\overrightarrow{uu'}$. The search will extend the state step-by-step. The search branch terminates when a state corresponding to a match is found or search cannot continue. In this case, the algorithm backtracks and tries another search branch.

The only change that is required to implement Algorithm 1 is in the termination condition (i.e., the final state) so that it stops when a partial match is found rather than looking for a complete match.

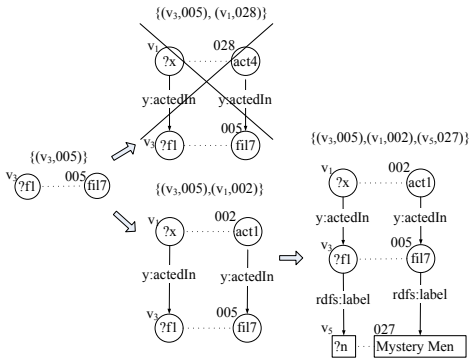


Fig. 5 Finding Local Partial Matches

Example 6 Figure 5 shows how to compute Q's local partial matches in fragment F_1. Suppose that we initialize a function f with $(v_3, 005)$. In the second step, we expand to v_1 and consider v_1's candidates, which are 002 and 028. Hence, we introduce two vertex pairs $(v_1, 002)$ and $(v_1, 028)$ to expand f. Similarly, we introduce $(v_5, 027)$ into the function $\{(v_3, 005), (v_1, 002)\}$ in the third step. Then, $\{(v_3, 005), (v_1, 002), (v_5, 027)\}$ satisfies all conditions of Definition 6, thus, it is a local partial match, and is returned. In another search branch, we check the function $\{(v_3, 005), (v_1, 028)\}$, which cannot be expanded, i.e., we cannot introduce a new matching pair;without violating some conditions in Definition 6. Therefore, this search branch is terminated.

5 Assembly

Each site S_i finds all local partial matches in fragment F_i. The next step is to assemble partial matches to compute crossing matches and compute the final results. 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. Here, when S_i sends the local partial matches to the final assembly site for joining, it also tags which vertices in local partial matches are internal vertices or extended vertices of F_i. This will be useful for avoiding some computations as discussed in this section.

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 Proposition 4.

Definition 8 (Joinable) Given a query graph Q and two fragments F_i and F_j ($i \neq j$), let PM_i and PM_j be the corresponding local partial matches over fragments F_i and F_j under functions f_i and f_j. PM_i and PM_j are *joinable* if and only if the following conditions hold:

1. There exist no 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 $\overrightarrow{uu'}$ such that u is an internal vertex and u' is an extended vertex in F_i, while u is an extended vertex and u' is an internal vertex in F_j. Furthermore, $f_i^{-1}(u) = f_j^{-1}(u)$ and $f_i^{-1}(u') = f_j^{-1}(u')$.

The first condition says that the same query vertex cannot be matched by different internal vertices in joinable partial matches. The second condition says that two local partial matches share at least one common crossing edge that corresponds to the same query edge.

Example 7 Let us recall query Q in Figure 2. Figure 3 shows two different local partial matches PM_1^2 and PM_2^2. We also show the functions in Figure 3. There do not exist two different vertices in the two local partial matches that match the same query vertex. Furthermore, they share a common crossing edge $\overrightarrow{002, 001}$, 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 9 (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 $PM = PM_i \bowtie_f PM_j$), which is defined as follows for any vertex v in Q:

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

Figure 3 shows the join result of $PM_1^2 \bowtie_f PM_2^2$.

5.2 Centralized Assembly

In centralized assembly, all local partial matches are sent to a final 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).

[2] $f_j(v) = NULL$ means that vertex v in query Q is not matched in local partial match PM_j. It is formally defined in Definition 6 condition (2)

[3] In this paper, we use "←" to denote the assignment operator.

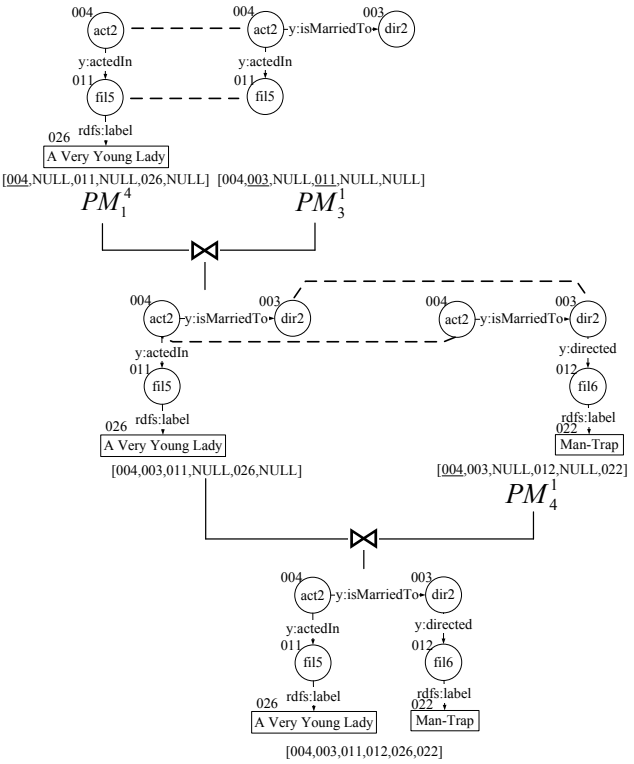


Fig. 6 Joining PM_1^4, PM_3^1 and PM_4^1

Example 8 Figure 3 shows the join result of $PM_1^2 \bowtie_f PM_2^2$. In this example, we consider a crossing match formed by three local partial matches. Let us consider three local partial matches PM_1^4, PM_4^1 and PM_3^1 in Figure 4. In the first iteration, we obtain the intermediate result $PM_1^4 \bowtie_f PM_3^1$ in Figure 6. Then, in the next iteration, $(PM_1^4 \bowtie_f PM_3^1)$ joins with PM_4^1 to obtain a crossing match.

5.2.1 Partitioning-based Join Processing

The join space in Algorithm 2 is large, since we need to check if every pair of local partial matches PM_i and PM_j are joinable. This subsection proposes an optimized technique to reduce the join space.

The intuition of our method is as follows. We divide all local partial matches into multiple partitions such that two local partial matches in the same set cannot be joinable; we only consider joining local partial matches from different partitions. The following theorem specifies which local partial matches can be put in the same partition.

Algorithm 2: Centralized Join-based Assembly

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

1 Each fragment F_i sends the set of local partial matches in each fragment F_i (i.e., $\Omega(F_i)$) to a single site for the assembly
2 Let $\Omega \leftarrow \bigcup_{i=1}^{i=k} \Omega(F_i)$
3 Set $MS \leftarrow \Omega$
4 **while** $MS \neq \emptyset$ **do**
5 Set $MS' \leftarrow \emptyset$
6 **for** *each local partial match* PM *in* MS **do**
7 **for** *each local partial match* PM' *in* Ω **do**
8 **if** PM *and* PM' *are joinable* **then**
9 Set $PM'' = PM \bowtie PM'$
10 **if** PM'' *is a complete match of* Q **then**
11 put PM'' into RS
12 **else**
13 put PM'' into MS'
14 $MS \leftarrow MS'$
15 Return RS

Theorem 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. □

Example 9 Figure 7 shows the serialization vectors (defined in Definition 6) of four local partial matches. 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 1, none of them are joinable.

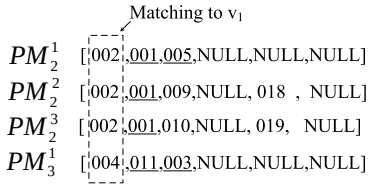


Fig. 7 The Local Partial Match Partition on v_1

Definition 10 (Local Partial Match Partitioning). Consider a SPARQL query Q with n vertices $\{v_1, ..., v_n\}$. Let Ω denote all local partial matches. $\mathcal{P} = \{P_{v_1}, ..., P_{v_n}\}$ is a partitioning of Ω if and only if the following conditions hold.

1. Each partition P_{v_i} $(i = 1, ..., n)$ consists of a set of local partial matches, each of which has an internal vertex that matches v_i.
2. $P_{v_i} \cap P_{v_j} = \emptyset$, where $1 \leq i \neq j \leq n$.
3. $P_{v_1} \cup ... \cup P_{v_n} = \Omega$

Example 10 Let us consider all local partial matches of our running example in Figure 4. Figure 8 shows two different partitionings.

As mentioned earlier, we only need to consider joining local partial matches from different partitions of $\mathcal{P}$. Given a partitioning $\mathcal{P} = \{P_{v_1}, ..., P_{v_n}\}$, 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 $\mathcal{P} = \{P_{v_1}, ..., P_{v_n}\}$ is given, and that the join order is from P_{v_1} to P_{v_n}, i.e. the order in $\mathcal{P}$. Choosing a good partitioning and the optimal join order will be discussed in Sections 5.2.2 and 5.2.3.

Algorithm 3: Partitioning-based Joining Local Partial Matches

Input: A partitioning $\mathcal{P} = \{P_{v_1}, ..., P_{v_n}\}$ of all local partial matches.
Output: All crossing matches set RS.

1 $MS \leftarrow P_{v_1}$
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_{v_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 Put PM' into MS'
13 Insert MS' into MS
14 Return RS

The basic idea of Algorithm 3 is to iterate the join process on each partition of $\mathcal{P}$. First, we set $MS \leftarrow P_{v_1}$ (Line 1 in Algorithm 3). Then, we try to join local partial matches PM in MS with local partial matches PM' in P_{v_2} (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_{v_2}) will join local partial matches in the later partition of $\mathcal{P}$ (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 partition of $\mathcal{P}$ in the next iterations (Line 13). We iterate the above steps for each partition of $\mathcal{P}$ in the partitioning (Lines 3-13).

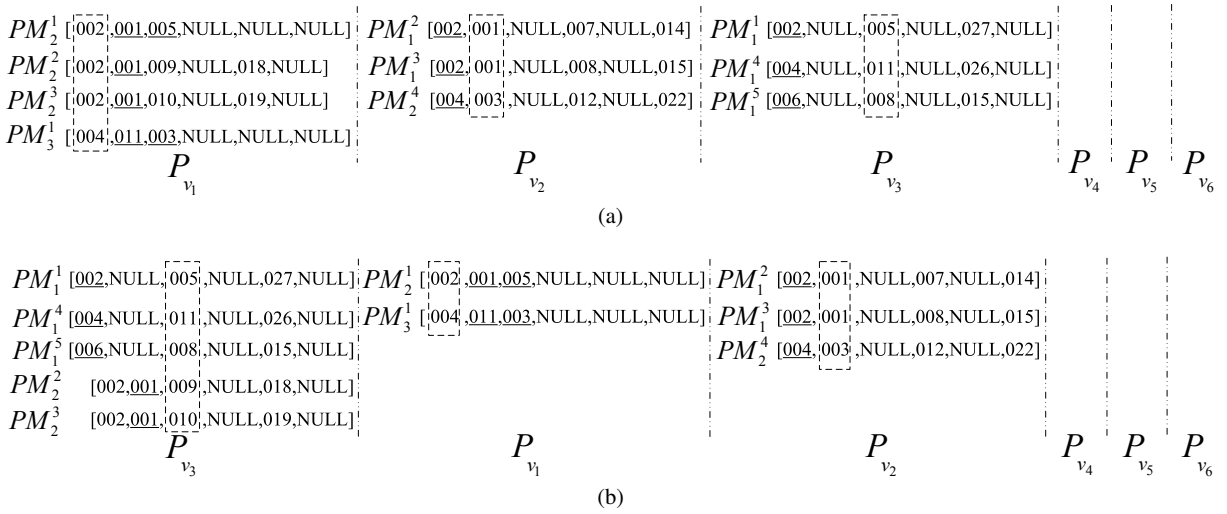


Fig. 8 Evaluation of Two Partitionings of Local Partial Matches

5.2.2 Finding the Optimal Partitioning

Obviously, given a set Ω of local partial matches, there may be multiple feasible local partial match partitionings, each of which leads to a different join performances. In this subsection, we discuss how to find the "optimal" local partial match partitioning over Ω, which can minimize the joining time of Algorithm 4.

First, there is a need for a measure that would define more precisely the *join cost* for a local partial match partitioning. We define it as follows.

Definition 11 (Join Cost). Given a query graph Q with n vertices v_1,...,v_n and a partitioning $\mathcal{P} = \{P_{v_1},...,P_{v_n}\}$ over all local partial matches Ω, the join cost is

$$Cost(\Omega) = O(\prod_{i=1}^{i=n} (|P_{v_i}| + 1)) \quad (1)$$

where $|P_{v_i}|$ is the number of local partial matches in P_{v_i} and 1 is introduced to avoid the "0" element in the product.

Definition 11 assumes that each pair of local partial matches (from different partitions of $\mathcal{P}$) are joinable so that we can quantify the worst-case performance. Naturally, more sophisticated and more realistic cost functions can be used instead, but, finding the most appropriate cost function is a major research issue in itself and outside the scope of this paper.

Example 11 The cost of the partitioning in Figure 8(a) is $5 \times 4 \times 4 = 80$, while that of Figure 8(b) is $6 \times 3 \times 4 = 72$. Hence, the partitioning in Figure 8(b) has lower join cost.

Based on the definition of join cost, the "optimal" local partial match partitioning is one with the minimal join cost. We formally define the *optimal partitioning* as follows.

Definition 12 (Optimal Partitioning). Given a partitioning $\mathcal{P}$ over all local partial matches Ω, $\mathcal{P}$ is the optimal partitioning if and only if there exists no another partitioning that has smaller join cost.

Unfortunately, Theorem 2 shows that finding the optimal partitioning is NP-complete.

Theorem 2 *Finding the optimal partitioning is NP-complete problem.*

Proof We can reduce a 0-1 integer planning problem to finding the optimal partitioning. We build a bipartite graph B, which contains two vertex groups B_1 and B_2. Each vertex a_j in B_1 corresponds to a local partial match PM_j in Ω, $j = 1, ..., |\Omega|$. Each vertex b_i in B_2 corresponds to a query vertex v_i, $i = 0, ..., n$. We introduce an edge between a_j and b_i if and only if PM_j has a internal vertex that is matching query vertex b_i. Let a variable x_{ji} denote the edge label of the edge $\overline{a_j b_i}$. Figure 9 shows an example bipartite graph of all local partial matches in Figure 4.

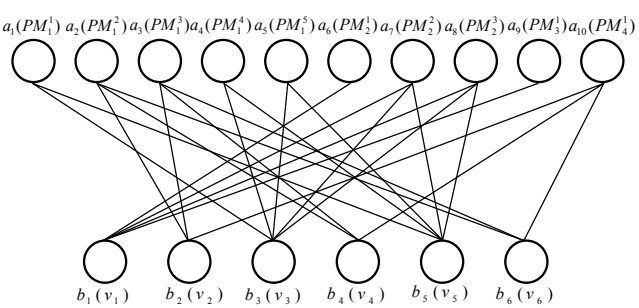


Fig. 9 Example Bipartite Graph

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

$\min \prod_{i=0}^{i=n} (\sum_j x_{ji} + 1)$
$st. \forall j, \sum_i x_{ji} = 1$

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

The equivalence between the 0-1 integer planning and finding the optimal partitioning is straightforward. The former is a classical NP-complete problem. Thus, the theorem holds. □

Although finding the optimal partitioning is NP-complete (see Theorem 2), in this work, we propose an algorithm with time complexity $(2^n \times |\Omega|)$, where n (i.e., $|V(Q)|$) is small in practice. Theoretically, this algorithm is called fixed-parameter tractable [10] [4].

Our algorithm is based on the following feature of optimal partitioning (see Theorem 3). 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 Ω) that have internal vertices matching v_i. Unlike the partitioning defined in Definition 10, U_{v_i} and U_{v_j} $(1 \le i \ne j \le n)$ may have overlaps. For example, PM_2^3 (in Figure 10) contains an internal vertex 002 that matches v_1, thus, PM_2^3 is in U_{v_1}. PM_2^3 also has internal vertex 010 that matches v_3, thus, PM_2^3 is also in U_{v_3}. However, the partitioning defined in Definition 10 does not allow overlapping among partitions of $\mathcal{P}$.

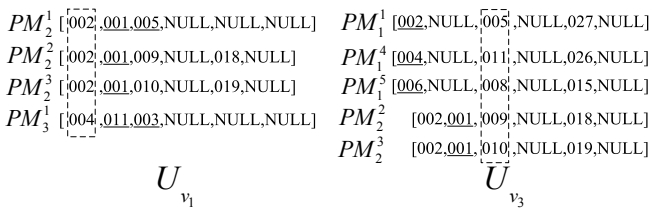


Fig. 10 U_{v_1} and U_{v_3}

Theorem 3 *Given a query graph Q with n vertices* $\{v_1,...,v_n\}$ *and a set of all local partial matches* Ω*, let* U_{v_i} $(i = 1, ..., n)$ *be all local partial matches (in* Ω*) that have internal vertices matching* v_i*. For the optimal partitioning* $\mathcal{P}_{opt} = \{P_{v_1}, .., P_{v_n}\}$ *where* P_{v_n} *has the largest size (i.e., the number of local partial matches in* P_{v_n} *is maximum) in* $\mathcal{P}_{opt}$*,* $P_{v_n} = U_{v_n}$*.*

Proof (by contradiction) Assume that $P_{v_n} \ne U_{v_n}$ in the optimal partitioning $\mathcal{P}_{opt} = \{P_{v_1}, .., P_{v_n}\}$. Then, there exists a local partial match $PM \notin P_{v_n}$ and $PM \in U_{v_n}$. We assume that $PM \in P_{v_j}$, $j \ne n$. The cost of $\mathcal{P}_{opt} = \{P_{v_1}, .., P_{v_n}\}$ is:

$$Cost(\Omega)_{opt} = (\prod_{1 \le i < n \wedge i \ne j} (|P_{v_i}| + 1)) \times (|P_{v_j}| + 1) \times (|P_{v_n}| + 1) \quad (2)$$

Since $PM \in U_{v_n}$, PM has an internal vertex matching v_n. Hence, we can also put PM into P_{v_n}. Then, we get a new partitioning $\mathcal{P}' = \{P_{v_1}, ..., P_{v_j} - \{PM\}, ..., , P_{v_n} \cup \{PM\}\}$. The cost of the new partitioning is:

$$Cost(\Omega) = (\prod_{1 \le i < n \wedge i \ne j} (|P_{v_i}| + 1)) \times |P_{v_j}| \times (|P_{v_n}| + 2) \quad (3)$$

Let $C = \prod_{1 \le i < n \wedge i \ne j} (|P_{v_i}| + 1)$, which exists in both Equations 2 and 3. Obviously, $C > 0$.

$$\begin{aligned} &Cost(\Omega)_{opt} - Cost(\Omega) \\ &= C \times (|P_{v_n}| + 1) \times (|P_{v_j}| + 1) - C \times (|P_{v_n}| + 2) \times (|P_{v_j}|) \\ &= C \times (|P_{v_n}| + 1 - |P_{v_j}|) \end{aligned}$$

Because P_{v_n} is the largest partition in $\mathcal{P}_{opt}$, $|P_{v_n}| + 1 - |P_{v_j}| > 0$. Furthermore, $C > 0$. Hence, $Cost(\Omega)_{opt} - Cost(\Omega) > 0$, 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 $|P_{v_n}|$ is the largest, $PM \notin P_{v_n}$ and $PM \in U_{v_n}$. In other words, $P_{v_n} = U_{v_n}$ in the optimal partitioning. □

Let Ω denote all local partial matches. Assume that the optimal partitioning is $\mathcal{P}_{opt} = \{P_{v_1}, P_{v_2}, ..., P_{v_n}\}$. We re-order the partitions of $\mathcal{P}_{opt}$ in non-descending order of sizes, i.e., $\mathcal{P}_{opt} = \{P_{v_{k_1}}, ..., P_{v_{k_n}}\}$, $|P_{v_{k_1}}| \ge |P_{v_{k_2}}| \ge ... \ge |P_{v_{k_n}}|$. According to Theorem 3, we can conclude that $P_{v_{k_1}} = U_{v_{k_1}}$ in the optimal partitioning $\mathcal{P}_{opt}$.

Let $\Omega_{\overline{v_{k_1}}} = \Omega - U_{v_{k_1}}$, i.e., the set of local partial matches excluding the ones with an internal vertex matching v_{k_1}. It is straightforward to know $Cost(\Omega)_{opt} = |P_{v_{k_1}}| \times Cost(\Omega_{\overline{v_{k_1}}})_{opt} = |U_{v_{k_1}}| \times Cost(\Omega_{\overline{v_{k_1}}})_{opt}$. In the optimal partitioning over $\Omega_{\overline{v_{k_1}}}$, we assume that $P_{v_{k_2}}$ has the largest size. Iteratively, according to Theorem 3, we know that $P_{v_{k_2}} = U'_{v_{k_2}}$, where $U'_{v_{k_2}}$ denotes the set of local partial matches with an internal vertex matching v_{k_2} in $\Omega_{\overline{v_{k_1}}}$.

According to the above analysis, if a vertex order is given, the partitioning over Ω is fixed. Assume that the optimal vertex order that leads to minimum join cost is given as $\{v_{k_1}, ..., v_{k_n}\}$. The partitioning algorithm work as follows.

Let $U_{v_{k_1}}$ denote all local partial matches (in Ω) that have internal vertices matching vertex v_{k_1} [5]. Obviously, $U_{v_{k_1}}$ is fixed if Ω and the vertex order is given. We set $P_{v_{k_1}} = U_{v_{k_1}}$. In the second iteration, we remove all local partial matches in $U_{v_{k_1}}$

[4] An algorithm is called fixed-parameter tractable for a problem of size l, with respect to a parameter n, if it can be solved in time $O(f(n)g(l))$, where $f(n)$ can be any function but $g(l)$ must be polynomial [10].

[5] When we find local partial matches in fragment F_i and send them to join, we tag which vertices in local partial matches are internal vertices of F_i.

from $\Omega_{\overline{v_{k_1}}}$, i.e, $\Omega_{\overline{v_{k_1}}} = \Omega - U_{v_{k_1}}$. We set $U'_{v_{k_2}}$ to be all local partial matches (in Ω') that have internal vertices matching vertex v_{k_2}. Then, we set $P_{v_{k_2}} = U'_{v_{k_2}}$. Iteratively, we can obtain $P_{v_{k_3}}, ..., P_{v_{k_n}}$.

Example 12 Consider all local partial matches in Figure 11. Assume that the optimal vertex order is $\{v_3, v_1, v_2\}$. We will discuss how to find the optimal order later. In the first iteration, we set $P_{v_3} = U_{v_3}$, which contains five matches. For example, $PM^1_1 = [\underline{002}^6, NULL, 005, NULL, 027, NULL]$ is in U_{v_3}, since internal vertex 005 matches v_3. In the second iteration, we set $\Omega_{\overline{v_3}} = \Omega - P_{v_3}$. Let U'_{v_1} to be all local partial matches in $\Omega_{\overline{v_3}}$ that have internal vertices matching vertex v_1. Then, we set $P_{v_1} = U'_{v_1}$. Iteratively, we can obtain the partitioning $\{P_{v_3}, P_{v_1}, P_{v_2}\}$, as shown in Figure 11.

Therefore, the challenging problem is how to find the optimal vertex order $\{v_{k_1}, ..., v_{k_n}\}$. Let us denote by $\Omega_{\overline{v_{k_1}}}$ all local partial matches (in Ω) that do not contain internal vertices matching v_{k_1}, i.e., $\Omega_{\overline{v_{k_1}}} = \Omega - U_{v_{k_1}}$. It is straightforward to have the following *optimal substructure*[7] in Equation 4.

$$\begin{aligned} &Cost(\Omega)_{opt} = |P_{v_{k_1}}| \times Cost(\Omega_{\overline{v_{k_1}}})_{opt} \\ &= |U_{v_{k_1}}| \times Cost(\Omega_{\overline{v_{k_1}}})_{opt} \end{aligned} \tag{4}$$

Since we do not know which vertex is v_{k_1}, we introduce the following optimal structure that is used in our dynamic programming algorithm (Lines 3-7 in Algorithm 4).

$$\begin{aligned} &Cost(\Omega)_{opt} = MIN_{1 \le i \le n}(|P_{v_i}| \times Cost(\Omega_{\overline{v_i}})_{opt}) \\ &= MIN_{1 \le i \le n}(|U_{v_i}| \times Cost(\Omega_{\overline{v_i}})_{opt}) \end{aligned} \tag{5}$$

Obviously, it is easy to design a naive dynamic algorithm based on Equation 5. However, it can be further optimized by recording some intermediate results. Based on Equation 5, we can prove the following equation.

$$\begin{aligned} &Cost(\Omega)_{opt} = MIN_{1 \le i \le n; 1 \le j \le n; i \ne j}(|P_{v_i}| \times |P_{v_j}| \times Cost(\Omega_{\overline{v_i v_j}})_{opt}) \\ &= MIN_{1 \le i \le n; 1 \le j \le n; i \ne j}(|U_{v_i}| \times |U'_{v_j}| \times Cost(\Omega_{\overline{v_i v_j}})_{opt}) \end{aligned} \tag{6}$$

where $\Omega_{\overline{v_i v_j}}$ denotes all local partial matches that do not contain internal vertices matching v_i or v_j, and U'_{v_j} denotes all local partial matches (in $\Omega_{\overline{v_i}}$) that contain internal vertices matching vertex v_j.

However, if Equation 6 is used naively in the dynamic programming formulation, it would result in repeated computations. For example, $Cost(\Omega_{\overline{v_1 v_2}})_{opt}$ will be computed twice in both $|U_{v_1}| \times |U'_{v_2}| \times Cost(\Omega_{\overline{v_1 v_2}})_{opt}$ and $|U_{v_2}| \times |U'_{v_1}| \times Cost(\Omega_{\overline{v_1 v_2}})_{opt}$. To avoid this, we introduce a map that records $Cost(\Omega')$ that is already calculated (Line 16 in Function OptComCost), so that subsequent uses of $Cost(\Omega')$ can be serviced directly by searching the map (Lines 8-10 in Function ComCost).

We can prove that there are $\sum_{i=1}^{n} \binom{n}{i} = 2^n$ items in the map (worst case), where $n = |V(Q)|$. Thus, the time complexity of the algorithm is $(2^n \times |\Omega|)$. Since n (i.e., $|V(Q)|$) is small in practice, this algorithm is fixed-parameter tractable.

Algorithm 4: Finding the Optimal Partitioning

Input: All local partial matches Ω
Output: The Optimal Partitioning $\mathcal{P}_{opt}$ and $Cost_{opt}(\Omega)$

```
1  minID ← Φ
2  Cost(Ω)_opt ← ∞
3  for i = 1 to n do
4      Cost_opt(Ω − U_{v_i}), P'_i ← ComCost(Ω − U_{v_i}, {v_i}) /* Call
           Function ComCost, U'_{v_i} denotes all
           local partial matches (in Ω') that
           have vertices match v_i.                    */
5      if Cost(Ω)_opt > |U_{v_i}| × Cost_opt(Ω − U_{v_i}) then
6          Cost(Ω)_opt ← |U_{v_i}| × Cost_opt(Ω − U_{v_i})
7          minID = i
8  P_opt ← {U_{v_minID}} ⋃ P'_minID
9  Return P_opt
```

5.2.3 Join Order

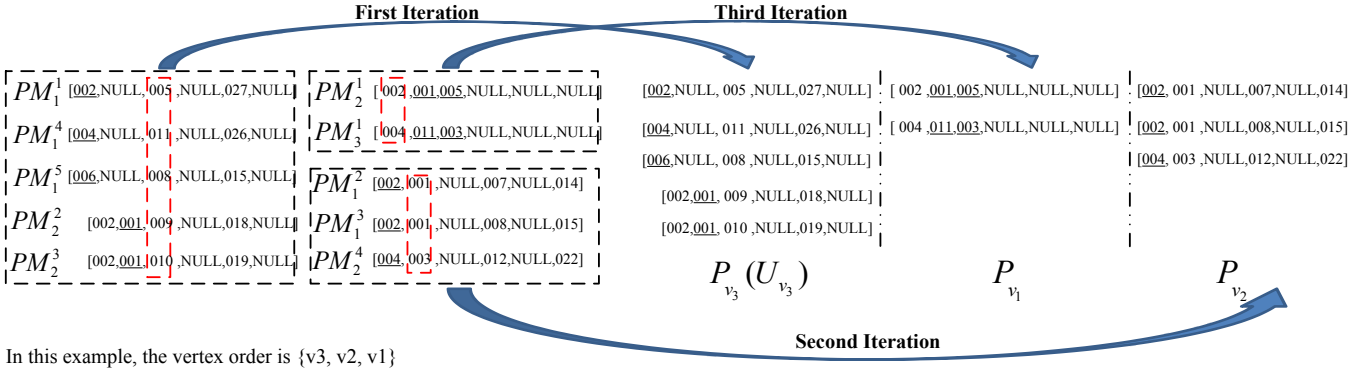
When we determine the optimal partitioning of local partial matches, the join order is also determined. If the optimal partitioning is $\mathcal{P}_{opt} = \{P_{v_{k_1}}, ..., P_{v_{k_n}}\}$ and $|P_{v_{k_1}}| \ge |P_{v_{k_2}}| \ge ... \ge |P_{v_{k_n}}|$, then the join order must be $P_{v_{k_1}} \bowtie P_{v_{k_2}} \bowtie ... \bowtie P_{v_{k_n}}$. The reasons are as follows.

First, changing the join order may not prune any intermediate results. Let us recall the example optimal partitioning $\{P_{v_3}, P_{v_2}, P_{v_1}\}$ shown in Figure 8(b). The join order should be $P_{v_3} \bowtie P_{v_2} \bowtie P_{v_1}$, and any changes in the join order would not prune intermediate results. For example, if we first join P_{v_2} with P_{v_1}, we can not prune the local partial matches in P_{v_2} that can not join with any local partial matches in P_{v_1}. This is because there may be some local partial matches P_{v_3} that have an internal vertex matching v_1 and can join with local partial matches in P_{v_2}. In other words, the results of $P_{v_2} \bowtie P_{v_1}$ is not smaller than P_{v_2}. Similarly, we can prove that 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 $\mathcal{P}_{opt} = \{P_{v_{k_1}}, ..., P_{v_{k_n}}\}$ and $|P_{v_{k_1}}| \ge |P_{v_{k_2}}| \ge ... \ge |P_{v_{k_n}}|$, if the set of the first n' vertices, $\{v_{k_1}, v_{k_2}, ..., v_{n'}\}$, is a vertex cut of the query graph, the join order for the remaining $n - n'$ partitions of $\mathcal{P}$ has an effect. For example, let us consider the partitioning $\{P_{v_1}, P_{v_3}, P_{v_2}\}$ in Figure 8(a). If the partitioning is optimal,

[6] We underline all extended vertices in serialization vectors.

[7] A problem is said to have *optimal substructure* if an optimal solution can be constructed efficiently from optimal solutions of its subproblems [9]. This property is often used in dynamic programming formulations.



In this example, the vertex order is {v3, v2, v1}

Fig. 11 Example of Partitioning Local Partial Matches

Function ComCost(Ω', W)

Input: local partial match set Ω' and a set W of vertices that have been used
Output: The Optimal Partitioning $\mathcal{P}'_{opt}$ over Ω' and $Cost_{opt}(\Omega')$

1 **if** $\Omega' = \Phi$ **then**
2 Return $\mathcal{P}'_{opt} \leftarrow \Phi$; $Cost(\Omega')_{opt} \leftarrow 1$;
3 **else**
4 $minID \leftarrow \Phi$
5 $Cost(\Omega')_{opt} \leftarrow \infty$
6 **for** $i = 1$ *to* n **do**
7 **if** $v_i \notin W$ **then**
8 **if** *MAP consists the key* $(\Omega'\text{-}U'_{v_i})$ **then**
/* if $Cost(\Omega'-U'_{v_i}))_{opt}$ has been calculated before */
9 $Cost_{opt}(\Omega' - U'_{v_i}), \mathcal{P}'_i \leftarrow MAP[(\Omega'\text{-}U'_{v_i})]$
/* finding the corresponding join cost and the optimal partitioning over $(\Omega'-U'_{v_i})$ from the map */
10 **else**
11 $Cost_{opt}(\Omega' - U'_{v_i}), \mathcal{P}'_i \leftarrow$ **ComCost**$(\Omega' - U'_{v_i}, W \cup \{v_i\})$
/* Call Function ComCost, U'_{v_i} denotes all local partial matches (in Ω') that have vertices match v_i. */
12 **if** $Cost(\Omega')_{opt} > |U'_{v_i}| \times Cost_{opt}(\Omega' - U'_{v_i})$ **then**
13 $Cost(\Omega')_{opt} \leftarrow |U_{v_i}| \times Cost_{opt}(\Omega' - U'_{v_i})$
14 $minID = i$
15 $\mathcal{P}'_{opt} \leftarrow \{U_{v_{minID}}\} \bigcup \mathcal{P}'_{minID}$
16 Insert (key=Ω', value=$(Cost(\Omega')_{opt}, \mathcal{P}'_{opt})$) into the *MAP*.
17 Return $Cost(\Omega')_{opt}$ and $\mathcal{P}'_{opt}$

then both joining P_{v_1} with P_{v_2} first and joining P_{v_1} with P_{v_3} first can work. However, it is possible for their cost to be different.[8] In the worst case, if the query graph is a complete graph, the join order has no effect on the performance.

In conclusion, when the optimal partitioning is determined as $\mathcal{P}_{opt} = \{P_{v_{k_1}}, ..., P_{v_{k_n}}\}$ and $|P_{v_{k_1}}| \geq |P_{v_{k_2}}| \geq ... \geq |P_{v_{k_n}}|$, then the join order must be $P_{v_{k_1}} \bowtie P_{v_{k_2}} \bowtie ... \bowtie P_{v_{k_n}}$.

[8] Note that, in this example, their cost values are the same, but they are possible to be different.

The join cost can be estimated based on the cost function (Definition 11).

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 [45] to design a synchronous algorithm for distributed assembly. A BSP computation proceeds in a series of global *supersteps*, each of which 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 $\Delta^m_{in}(F_i)$ 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 in the first $(m-1)$ supersteps. In the m-th superstep, we join local partial matches in $\Delta^m_{in}(F_i)$ with local partial matches in $\Omega^m(F_i)$ by Algorithm 5. For each intermediate result PM, we check if it can join with some local partial match PM' in $\Omega^m(F_i) \bigcup \Delta^m_{in}(F_i)$. 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 $\Omega^m(F_i) \bigcup \Delta^m_{in}(F_i)$ in the next iteration. We also insert the intermediate result PM'' into $\Delta^m_{out}(F_i)$ 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.

Communication. Processors exchange data among themselves. Consider the m-th superstep. A straightforward communication strategy is as follows. If an intermediate result PM in $\Delta^m_{out}(F_i)$ shares a crossing edge with fragment F_j, PM

Algorithm 5: 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; $\Delta^m_{out}(F_i)$, the intermediate results that will be sent

```
Let Ω = Ω^m(F_i) ∪ Δ^m_in(F_i)
Set MS = Δ^m_in(F_i)
for N = 1 to |V(Q)| do
    if |MS|=0 then
        Break;
    Set MS' = φ
    for each local partial match PM in MS do
        for each local partial match PM' in Ω^m(F_i) ∪ Δ^m_in(F_i) do
            if PM and PM' are joinable then
                PM'' ← PM ⋈ PM'
                if PM'' is a SPARQL match then
                    Put PM'' into the answer set RS
                else
                    Put PM'' into MS'
    Insert MS' into Δ^m_out(F_i)
    Clear MS and MS ← MS'
Ω^{m+1}(F_i) = Ω^m(F_i) ∪ Δ^m_out(F_i)
Return RS and Δ^m_out(F_i)
```

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^4_1 (at site S_1) and PM^1_3 (at site S_3) to form a complete crossing match. According to the straightforward communication strategy, PM^4_1 will be sent to S_1 from S_3 to produce $PM^4_1 \bowtie PM^1_3$ at S_3. Similarly, PM^1_3 is sent from S_3 to S_1 to assemble at site S_1. In other words, we obtain the join result $PM^4_1 \bowtie PM^1_3$ 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* ($\prec$) over fragments $\mathcal{F}$ in a non-descending order of $|\Omega(F_i)|$, i.e., the number of local partial matches in fragment F_i found at the partial evaluation stage.

Definition 13 Given any two fragments F_i and F_j, $F_i \prec F_j$ if and only if $|\Omega(F_i)| \leq |\Omega(F_j)|$ $(1 \leq i, j \leq n)$.

Without loss of generality, we assume that $F_1 \prec F_2 \prec ... \prec F_n$ 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},...,F_{i_m}, where $F_{i_1} \prec F_{i_2} \prec ... \prec F_{i_m}$ $(1 \leq i_1, ..., i_m \leq n)$. 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 matches from F_3 or even larger fragments (such as F_4,...,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,...,F_n).

The "divide-and-conquer" framework can avoid duplicate results, since each crossing match can be only generated at a single site according to the "divided search 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 $F_i \prec F_j$.

Let us consider an intermediate result PM in $\Delta^m_{out}(F_i)$. Assume that PM is generated by joining intermediate results from m different fragments $F_{i_1}, ..., F_{i_m}$, where $F_{i_1} \prec F_{i_2} \prec ... \prec F_{i_m}$. We send PM to another fragment F_j if and only if two conditions hold: (1) $F_j \succ F_{i_m}$; and (2) F_j shares common crossing edges with at least one fragment of F_{i_1},...,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, Ω_{F_i}. Since it is impossible to assemble local partial matches in the same fragment, the 0-th superstep requires no local computation. It enters the communication stage directly. Each site S_i sends Ω_{F_i} 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 using a fragmentation graph topology graph.

Definition 14 (**Fragmentation Topology Graph**) Given a fragmentation $\mathcal{F}$ over an RDF graph G, the corresponding *fragmentation topology graph* T is defined as follows: Each node in T is a fragment F_i, $i = 1, ..., k$. 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_j. Hence, the number of the supersteps in the BSP-based algorithm is $Dia(T)$.

6 Handling General SPARQL

So far, we only consider BGP (basic graph pattern) query evaluation. In this section, we discuss how to extend our method to general SPARQL queries involving UNION, OPTIONAL and FILTER statements.

```
Select ?name ?city where{
?person actedIn ?film .
?film rdfs:label ?name.
{ ?person rdfs:label "Hank Azaria". }
UNION
{?person rdfs:label "Slither". }
OPTIONAL{?person livedIn ?city . }
FILTER (?regex ( str(?name), "Mystery"))
}
```

Fig. 12 Example General SPARQL Query with UNION, OPTIONAL and FILTER

A general SPARQL query and SPARQL query results can be defined recursively based on BGP queries.

Definition 15 (General SPARQL Query) Any BGP is a SPARQL query. If Q_1 and Q_2 are SPARQL queries, then expressions $(Q_1\ AND\ Q_2)$, $(Q_1\ UNION\ Q_2)$, $(Q_1\ OPT\ Q_2)$ and $(Q_1\ FILTER\ F)$ are also SPARQL queries.

Figure 12 shows an example general SPARQL query with multiple operators, including UNION, OPTIONAL and FILTER. The set of all matches for Q is denoted as $[\![Q]\!]$.

Definition 16 (Match of General SPARQL Query) Given an RDF graph G, the match set of a SPARQL query Q over G, denoted as $[\![Q]\!]$, is defined recursively as follows:

1. If Q is a BGP, $[\![Q]\!]$ is the set of matches defined in Definition 3 of Section 3.
2. If $Q = Q_1\ AND\ Q_2$, then $[\![Q]\!] = [\![Q_1]\!] \bowtie [\![Q_2]\!]$
3. If $Q = Q_1\ UNION\ Q_2$, then $[\![Q]\!] = [\![Q_1]\!] \cup [\![Q_2]\!]$
4. If $Q = Q_1\ OPT\ Q_2$, then $[\![Q]\!] = ([\![Q_1]\!] \bowtie [\![Q_2]\!]) \cup ([\![Q_1]\!] \backslash [\![Q_2]\!])$
5. If $Q = Q_1\ FILTER\ F$, then $[\![Q]\!] = \Theta_F([\![Q_1]\!])$

We can parse each SPARQL query into a parse tree[9], where the root is a *pattern group*. A pattern group specifies a SPARQL statement, and consists of a BGP query with UNION, OPTIONAL and FILTER statements. The UNION and OPTIONAL may recursively contain multiple pattern groups. It is easy to show that each leaf node (in the parser tree) is a BGP query whose evaluation was discussed earlier. We design a recursive algorithm (Algorithm 8) to find answers to handle UNION, OPTIONAL and FILTER. Specifically, we perform left-outer join between BGP and OPTIONAL query results (Lines 4-5 in Function RecursiveEvaluation). Then, we join the answer set with UNION query results (Line 9 in Function RecursiveEvaluation). Finally, we evaluate FILTER operator (Line 13).

Further optimizing general SPARQL evaluation is also possible (e.g., [4]). However, this issue is independent on our studied problem in this paper.

[9] We use ANTRL v3's grammar which is an implementation of the SPARQL grammar's specifications. It is available at http://www.antlr3.org/grammar/1200929755392/

Algorithm 6: Handling General SPARQLs

Input: A SPARQL Q
Output: The result set RS of Q
1 Parse Q into a parser tree T
2 RS =**RecursiveEvaluation**(T) // Call Function

Function RecursiveEvaluation(T)

1 Evaluate BGP in T and put all its results into RS
2 **for** *each subtree T' in OPTIONAL statement of T* **do**
3 | // *Handling OPTIONAL stamtent.*
4 | RS' =**RecursiveEvaluation**(T')
5 | $RS = RS \rtimes RS'$
6 $RS' = \emptyset$
7 **for** *each subtree T' of pattern group in UNIONs of T* **do**
8 | // *Handling UNION stamtent.*
9 | $RS' = RS'\ UNION$ **RecursiveEvaluation**(T')
10 $RS = RS \bowtie RS'$
11 **for** *each expression F in FILTER operators* **do**
12 | // *Handling FILTER operator.*
13 | Select RS by using expression F
14 Return RS

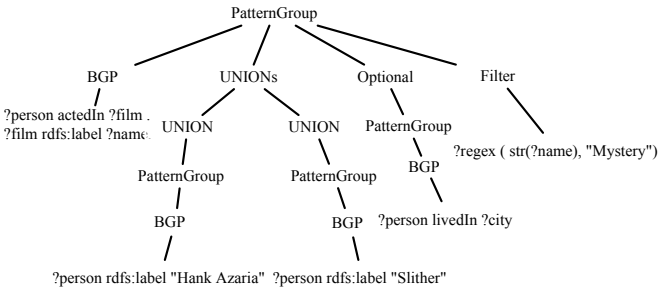


Fig. 13 Parser Tree of Example General SPARQL Query

7 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 a cloud-based approach (EAGRE [48]), two partition-based approaches (GraphPartition [22] and TripleGroup [28]), two memory-based systems (TriAD [18] and Trinity.RDF [47]) and two federated SPARQL query systems (FedX [42] and SPLENDID [16]). The results of of federated system comparisons are given in Appendix E since, as argued earlier, the environment targeted by these systems is different than ours.

Setting. We use two benchmark datasets with different sizes and one real dataset in our experiments, in addition to FedBench used in federated system experiments. Table 1 summarizes the statistics of these datasets. All sample queries are shown in Appendix B.

1) WatDiv [2] is a benchmark that enables diversified stress testing of RDF data management systems. In WatDiv, instances of the same type can have different attribute sets.

Dataset	Number of Triples	RDF N3 File Size(KB)	Number of Entities
WatDiv 100M	109,806,750	15,386,213	5,212,745
WatDiv 300M	329,539,576	46,552,961	15,636,385
WatDiv 500M	549,597,531	79,705,831	26,060,385
WatDiv 700M	769,065,496	110,343,152	36,486,007
WatDiv 1B	1,098,732,423	159,625,433	52,120,385
LUBM 1000	133,553,834	15,136,798	21,715,108
LUBM 10000	1,334,481,197	153,256,699	217,006,852
BTC	1,056,184,911	238,970,296	183,835,054

Table 1 Datasets

We generate three datasets varying sizes from 100 million to 1 billion triples. We use 20 queries of the basic testing templates provided by WatDiv [2] to evaluate our method. We randomly partition the WatDiv datasets into several fragments (except in Exp. 6 where we test different partitioning strategies). We assign each vertex v in RDF graph to the i-th fragment if $H(v)MOD\ N = i$, where $H(v)$ is a hash function and N is the number of fragments. By default, we use the uniform hash function and $N = 10$. Each machine stores a single fragment.

2) LUBM [17] is a benchmark that adopts an ontology for the university domain, and can generate synthetic OWL data scalable to an arbitrary size. We assign the university number to 10000. The number of triples is about 1.33 billion. We partition the LUBM datasets according to the university identifiers. Although LUBM defines 14 queries, some of these are similar; therefore we use the 7 benchmark queries that have been used in some recent studies [5,50]. We report the results over all 14 queries in Appendix B for completeness. As expected ,the results over 14 benchmark queries are similar to the results over 7 queries.

3) BTC 2012 (http://km.aifb.kit.edu/projects/btc-2012/) is a real dataset that serves as the basis of submissions to the Billion Triples Track of the Semantic Web Challenge. After eliminating all redundant triples, this dataset contains about 1 billion triples. We use METIS to partition the RDF graph, and use the 7 queries in [48].

4) FedBench [41] is used for testing against federated systems; it is described in Appendix E along with the results.

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. Each site holds one fragment of the dataset. At each site, we install gStore [50] 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 at each stage (i.e., partial evaluation and assembly process) with regard to different queries in WatDiv 1B and LUBM 1000. We report the running time of each stage (i.e., partial evaluation and assembly) and the number of local partial matches, inner matches, and crossing matches, with regard to different query types in Tables 2 and 3. We also compare the centralized and distributed assembly strategies. The time for assembly includes the time for computing the optimal join order. Note that we classify SPARQL queries into four categories according to query graphs' structures: star, linear, snowflake (several stars linked by a path) and complex (a combination of the above with complex structure).

Partial Evaluation: Tables 2 and 3 show that if there are some selective triple patterns[10] 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 selective triple patterns can reduce the search space. Furthermore, the running time also depends on the number of inner matches and local partial matches, as shown in Tables 2 and 3. 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 performances of linear, 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 parallelization to speed up both the network communication and assembly. For example, in F_3, there are 4065632 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 alternative. 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 and centralized assembly approaches 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.

In Tables 2 and 3, we also show the number of fragments involved in each test query. For most queries, their local partial matches and crossing matches involve all fragments. Queries containing selective triple patterns (L_1 in WatDiv) may only involve a part of the fragmentation.

Exp 2: Evaluating Optimizations in Assembly. In this experiment, we use WatDiv 1B to evaluate two different

[10] A triple pattern t is a "selective triple pattern" if it has no more than 100 matches in RDF graph G

				Partial Evaluation			Assembly			Total			# of LPMFs[8]	# of CMFs[9]
							Time(in ms)			Time(in ms)				
				Time(in ms)	# of LPMs[2]	# of IMs[3]	Centralized	Distributed	# of CMs[4]	PECA[5]	PEDA[6]	# of Matches[7]		
Star		S_1	√[1]	43803	0	1	0	0	0	43803	43803	1	0	0
		S_2	√	74479	0	13432	0	0	0	74479	74479	13432	0	0
		S_3	√	8087	0	13335	0	0	0	8087	8087	13335	0	0
		S_4	√	16520	0	2	0	0	0	16520	16520	1	0	0
		S_5	√	1861	0	112	0	0	0	1861	1861	940	0	0
		S_6	√	50865	0	14	0	0	0	50865	50865	14	0	0
		S_7	√	56784	0	1	0	0	0	56784	56784	1	0	0
Linear		L_1	√	15340	2	0	1	16	1	15341	15356	1	2	2
		L_2	√	1492	794	88	18	130	793	1510	1622	881	10	10
		L_3	√	16889	0	5	0	0	0	16889	16889	5	0	0
		L_4	√	261	0	6005	0	0	0	261	261	6005	0	0
		L_5	√	48055	1274	141	572	1484	1273	48627	49539	1414	10	10
Snowflake		F_1	√	64699	29	1	9	49	14	64708	64748	15	10	10
		F_2	√	203968	2184	99	1598	3757	1092	205566	207725	1191	10	10
		F_3	√	2341932	4065632	58	3673409	2489325	6200	6015341	4831257	6258	10	10
		F_4	√	251546	6909	0	13693	8864	1808	265239	260410	1808	10	10
		F_5	√	25180	92	3	58	1028	46	25238	26208	49	10	10
Complex		C_1		206864	161803	4	9195	5265	356	216059	212129	360	10	10
		C_2		1613525	937198	0	229381	174167	155	1842906	1787692	155	10	10
		C_3		123349	0	80997	0	0	0	123349	123349	80997	0	0

[1] √ means that the query involves some selective triple patterns.
[2] "# of LPMs" means the number of local partial matches.
[3] "# of IMs" means the number of inner matches.
[4] "# of CMs" means the number of crossing matches.
[5] "PECA" is the abbreviation of *P*artial *E*valuation & *C*entralized *A*ssembly.
[6] "PEDA" is the abbreviation of *P*artial *E*valuation & *D*istributed *A*ssembly.
[7] "# of Matches" means the number of matches.
[8] "# of LPMFs" means the number of fragments containing local partial matches.
[9] "# of CMFs" means the number of fragments containing crossing matches.

Table 2 Evaluation of Each Stage on WatDiv 1B

				Partial Evaluation			Assembly			Total			# of LPMFs	# of CMFs
							Time(in ms)			Time(in ms)				
				Time(in ms)	# of LPMs	# of IMs	Centralized	Distributed	# of CMs	PECA	PEDA	# of Matches		
Star		Q_2		1818	0	1081187	0	0	0	1818	1818	1081187	0	0
		Q_4	√	82	0	10	0	0	0	82	82	10	0	0
		Q_5	√	8	0	10	0	0	0	8	8	10	0	0
Snowflake		Q_6	√	158	6707	110	164	125	15	322	283	125	10	10
Complex		Q_1		52548	3033	2524	53	60	4	52601	52608	2528	10	10
		Q_3		920	3358	0	36	48	0	956	968	0	10	0
		Q_7		3945	167621	42479	211670	35856	1709	215615	39801	44190	10	10

Table 3 Evaluation of Each Stage on LUBM 1000

optimization techniques in the assembly: partitioning-based join strategy (Section 5.1) and the divide-and-conquer approach in the distributed assembly (Section 5.3). If a query does not have any local partial matches in RDF graph G, it does not need the assembly process. Therefore, we only use the benchmark queries that need assembly (L_1, L_2, L_5, F_1, F_2, F_3, F_3, F_4, F_5, C_1 and C_2) 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 4, 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 5.2.2. 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 4. We find that the assembly with the optimal partitioning is faster than that with random partitioning, which confirms the effectiveness of our cost model. Especially for C_2, the assembly with the optimal partitioning is an order of magnitude faster than the assembly with random partitioning.

Divide-and-Conquer in Distributed Assembly. Table 5 shows that dividing the search space will speed up distributed assembly. Otherwise some duplicate results can be generated, as discussed in Section 5.3. Elimination of duplicates and parallelization speeds up distributed assembly. For example, for C_1, dividing search space lowers the time of assembly more than twice as much as no dividing search space.

Exp 3: Scalability Test. In this experiment, we vary the RDF dataset size from 100 million triples (WatDiv 100M) to 1 billion triples (WatDiv 1B) to study the scalability of our methods. Figures 14 and 15 show the performance of different queries using centralized and distributed assembly.

Query response time is affected by both the increase in data size (which is $1x \rightarrow 10x$ in these experiemnts) and the query type. For star queries, the query response time increases proportional to the data size, as shown in Figures 14(b) and 15(b). For other query types, the query response times may grow faster than the data size. Especially for F_3, the query response time increases 30 times as the datasize

	Partitioning-based Join Based on the Optimal Partitioning	Partitioning-based Join Based on the Random Partitioning	Naive Join
L_1	1	1	1
L_2	18	23	139
L_5	572	622	3419
F_1	1	1	1
F_2	1598	2286	48096
F_3	3673409	4005409	timeout[1]
F_4	13693	13972	timeout
F_5	58	80	8383
C_1	9195	10582	timeout
C_2	229381	4083181	timeout

[1] timeout is issued if query evaluation does not terminate in 10 hour

Table 4 Running Time of Partitioning-based Join vs. Naive Join (in ms)

	Distributed Assembly Time (in ms)	
	Dividing	No Dividing
L_1	16	19
L_2	130	151
L_5	1484	1684
F_1	49	55
F_2	3757	5481
F_3	2489325	4439430
F_4	8864	19759
F_5	1028	1267
C_1	5265	12194
C_2	174167	225062

Table 5 Dividing vs. No Dividing (in ms)

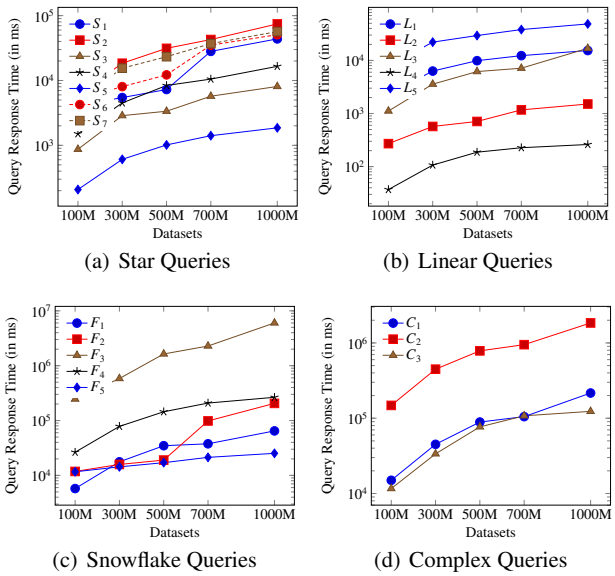


Fig. 14 Scalability Test of PECA

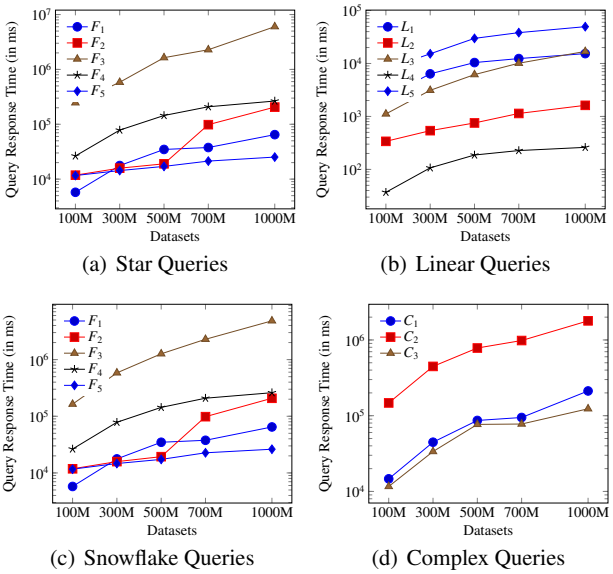


Fig. 15 Scalability Test of PEDA

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 query performance is scalable with RDF graph size on the benchmark datasets.

Note that, as mentioned in Exp. 1, there is no assembly process for star queries, since matches of a star query cannot cross two fragments. Therefore, the query response times for star queries in centralized and distributed assembly are the same. In contrast, for other query types, some local partial matches and crossing matches result in differences between the performances of centralized and distributed assembly. Here, L_3, L_4 and C_3 is a special case. Although they are not star queries, there are few local partial matches for L_3, L_4 and C_3. Furthermore, the crossing match number is 0 in L_3, L_4 and C_3 (in Table 2). Therefore, the assembly times for L_3, L_4 and C_3 are so small that the query response times in both centralized and distributed assembly are the almost same.

Exp 4: Intermediate Result Size and Query Performance VS. Query Decomposition Approaches. Table 6 compares the number of intermediate results in our method with two typical query decomposition approaches, i.e., GraphPartition and TripleGroup. We use undirected 1-hop guarantee for GraphPartition and 1-hop bidirection semantic hash partition for TripleGroup. The dataset is still WatDiv 1B.

A star query has no intermediate results, so each method can be answered at each fragment locally. Thus, all methods have the same response time, as shown in Table 7 (S_1 to S_6).

For other query types, both GraphPartition and TripleGroup need to decompose them into several star subqueries, and find these subquery matches (in each fragment) as intermediate results. Neither GraphPartition nor TripleGroup distinguish the star subquery matches that contribute to crossing matches from those that contribute to inner matches – all star subquery matches are involved in the assembly process. However, in our method, only local partial matches are involved in the assembly process, leading to lower communication cost and the assembly computation cost. Therefore,

	PECA & PEDA	GraphPartition	TripleGroup
$S_1 - S_7$	0	0	0
L_1	2	249571	249598
L_2	794	73307	79630
$L_3 - L_4$	0	0	0
L_5	1274	99363	99363
F_1	29	76228	15702
F_2	2184	501146	1119881
F_3	4065632	4515731	4515752
F_4	6909	132193	329426
F_5	92	2500773	9000762
C_1	161803	4551562	4451693
C_2	937198	1457156	2368405
C_3	0	0	0

Table 6 Number of Intermediate Results of Different Approaches on Different Partitioning Strategies

	PECA	PEDA	GraphPartition	TripleGroup
S_1	43803	43803	43803	43803
S_2	74479	74479	74479	74479
S_3	8087	8087	8087	8087
S_4	16520	16520	16520	16520
S_5	1861	1861	1861	1861
S_6	50865	50865	50865	50865
S_7	56784	56784	56784	56784
L_1	15341	15776	40840	39570
L_2	1510	1622	36150	36420
L_3	16889	16889	16889	16889
L_4	261	261	261	261
L_5	48627	49539	57550	57480
F_1	64708	64748	66230	66200
F_2	205566	207725	240700	248180
F_3	6015341	4831257	6244000	6142800
F_4	265239	260410	340540	340600
F_5	25238	29208	52180	91110
C_1	216059	212129	216720	223670
C_2	1842906	1787692	1954800	2168300
C_3	123349	123349	123349	123349

Table 7 Query Response Time of Different Approaches (in milliseconds)

the intermediate results that need to be assembled with others is smaller in our approach.

More intermediate results typically lead to more assembly time. Furthermore, both GraphPartition and TripleGroup employ MapReduce jobs for assembly, which takes much more time than our method. Table 7 shows that our query response time is faster than others.

Existing partition-based solutions, such as GraphPartition and TripleGroup, use MapReduce jobs to join intermediate results to find SPARQL matches. In order to evaluate the cost of MapReduce-jobs, we perform the following experiments over WatDiv 100M. We revise join processing in both GraphPartition and TripleGroup, by applying joins where intermediate results are sent to a central server using MPI. We use WatDiv 100M and only consider the benchmark queries that need join processing (L_1, L_2, L_5, F_1, F_2, F_3, F_3, F_4, F_5, C_1 and C_2) in our experiments. Moreover, all partition-based methods generate intermediate results and merge them at a central sever that shares the same framework with PECA, so we only compare them with PECA. The detailed results are given in Appendix C. Our technique is always faster regardless of the use of MPI or MapReduce-based join. This is because our method produces smaller intermediate result sets; MapReduce-based join dominates the query cost. our partial evaluation process is more expensive in evaluating local queries than GraphPartition and TripleGroup in many cases. This is easy to understand – since the subquery structures in GraphPartition and TripleGroup are fixed, such as stars, it is cheaper to find these local query results than finding local partial matches. Our system generally outperforms GraphPartition and TripleGroup significantly if they use MapReduce-based join. Even when GraphPartition and TripleGroup use distributed joins, our system is still faster than them in most cases (8 out of 10 queries used in this experiment, see Appendix C for details).

Exp 5: Performance on RDF Datasets with One Billion Triples. This experiment is a comparative evaluation of our method against GraphPartition, TripleGroup and EAGRE on three very large RDF datasets with more than one billion triples, WatDiv 1B, LUBM 10000 and BTC. Figure 16 shows the performance of different approaches.

Note that, almost half of the queries (S_1, S_2, S_3, S_4, S_5, S_6, S_7, L_3, L_4 and C_3 in WatDiv, Q_2, Q_4 and Q_5 in LUBM, Q_1, Q_2 and Q_3 in BTC) have no intermediate results generated in any of the approaches. For these queries, the response times of our approaches and partition-based approaches are the same. However, for other queries, the gap between our approach and others is significant. For example, L_2 in WatDiv, for Q_3, Q_6 and Q_7 in LUBM and Q_3, Q_4, Q_6 and Q_5 in BTC, our approach outperforms others one or more orders of magnitudes. We already explained the reasons for GraphPartition and TripleGroup in Exp 4; reasons for EAGRE performance follows.

EAGRE stores all triples as flat files in HDFS and answers SPARQL queries by scanning the files. Because HDFS does not provide fine-grained data access, a query can only be evaluated by a full scan of the files followed by a MapReduce job to join the intermediate results. Although EAGRE proposes some techniques to reduce I/O and data processing, it is still very costly. In contrast, we use graph matching to answer queries, which avoids scanning the whole dataset.

Exp 6: Impact of Different Partitioning Strategies. In this experiment, we test the performance under three different partitioning strategies over WatDiv 100M. The impact of different partitioning strategies is shown in Table 8. We implement three partitioning strategies: uniformly distributed hash partitioning, exponentially distributed hash partitioning, and minimum-cut graph partitioning.

The first partitioning strategy uniformly hashes a vertex v in RDF graph G to a fragment (machine). Thus, fragments on different machines have approximately the same size. The second strategy uses an exponentially distributed hash function with a rate parameter pf 0.5. Each vertex v has a probability of 0.5^k to be assigned to fragment (machine) k. This partitioning strategy results in skewed frag-

		Uniform	Exponential	Min-cut
S_1	PECA	4095	7472	**3210**
	PEDA	4095	7472	**3210**
S_2	PECA	5910	5830	**5053**
	PEDA	5910	5830	**5053**
S_3	PECA	**869**	2003	1098
	PEDA	**869**	2003	1098
S_4	PECA	**1506**	1532	1525
	PEDA	**1506**	1532	1525
S_5	PECA	**208**	384	255
	PEDA	**208**	384	255
S_6	PECA	5153	5642	**4145**
	PEDA	5153	5642	**4145**
S_7	PECA	5047	5720	**4085**
	PEDA	5047	5720	**4085**
L_1	PECA	**2301**	4271	3162
	PEDA	**2325**	4296	3168
L_2	PECA	271	502	**261**
	PEDA	339	505	**297**
L_3	PECA	**1115**	2122	1334
	PEDA	**1115**	2122	1334
L_4	PECA	37	54	**27**
	PEDA	37	54	**27**
L_5	PECA	7741	6736	**4984**
	PEDA	7863	6946	**5163**
F_1	PECA	5754	7889	**4386**
	PEDA	5768	7943	**4415**
F_2	PECA	11809	16461	**10209**
	PEDA	11832	16598	**10539**
F_3	PECA	246277	155064	**122539**
	PEDA	163642	115214	**103618**
F_4	PECA	26439	37608	**21979**
	PEDA	26421	36817	**22030**
F_5	PECA	11630	16433	**8735**
	PEDA	11654	16501	**8262**
C_1	PECA	14980	30271	**14131**
	PEDA	14667	29861	**13807**
C_2	PECA	147962	105926	**36038**
	PEDA	147406	104084	**35220**
C_3	PECA	**11631**	16368	13959
	PEDA	**11631**	16368	13959

Table 8 Query Response Time under Different Partitioning Strategies(in milliseconds)

ment sizes. Finally, we use min-cut based partitioning strategy (i.e., METIS algorithm) to partition graph G.

Minimum-cut partitioning strategy generally leads to fewer crossing edges than the other two. Thus, it beats the other two approaches in most cases, especially in complex queries (such as F and C category queries). For example, in C_2, the minimum-cut is faster than the uniform partitioning by more than four times. For star queries (i.e., S category queries), since there exist no crossing matches, the uniform partitioning and minimum-cut partitioning have the similar performance. Sometimes, the uniform partitioning is better, but the performance gap is very small. Due to the skew in fragment sizes, exponentially distributed hashing has worse performance, in most cases, than uniformly distributed hashing.

Although our partial evaluation-and-assembly framework is agnostic to the particular partitioning strategy, it is clear that it works better when fragment sizes are balanced, and the crossing edges are minimized. Many heuristic minimum-cut graph partitioning algorithms (a typical one is METIS [31]) satisfy the requirements.

Exp 7: Comparing with Memory-based Distributed RDF Systems. We compare our approach (which is disk-based) against TriAD [18] and Trinity.RDF [47] that are memory-based distributed systems. To enable fair comparison, we cache the whole RDF graph together with the corresponding index into memory. Experiments show that our system is faster than Trinity.RDF and TriAD in these benchmark queries. Results are given in Appendix D.

	RDF-3X	PECA	PEDA
Q_1	1084047	326167	309361
Q_2	81373	23685	23685
Q_3	72257	10239	10368
Q_4	7	753	753
Q_5	6	125	125
Q_6	355	3388	1914
Q_7	146325	143779	46123

Table 9 Comparison with Centralized System (in milliseconds)

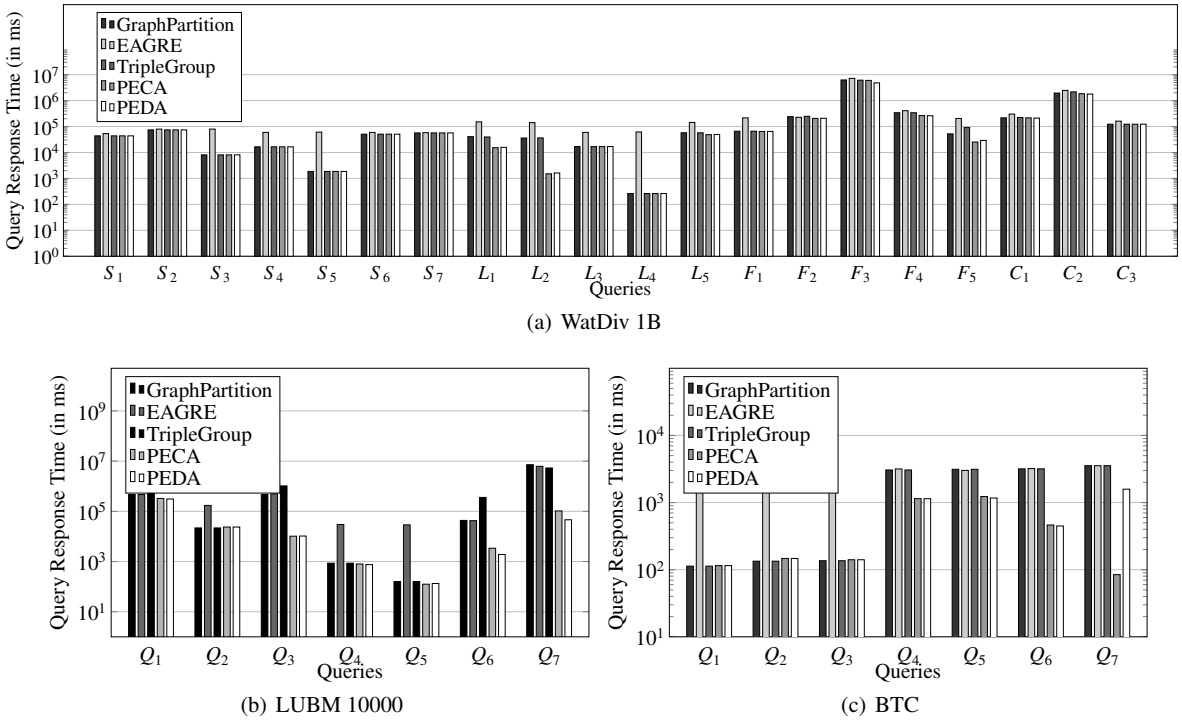
Exp 8: Comparing with Federated SPARQL Systems. In this experiment, we compare our methods with some federated SPARQL query systems including (FedX [42] and SPLENDID [16]). We evaluate our methods on the standardized benchmark for federated SPARQL query processing, FedBench [41]. Results are given in Appendix E.

Exp 9: Comparing with Centralized RDF Systems. In this experiment, we compare our method with RDF-3X in LUBM 10000. Table 9 shows the results.

Our method is generally faster than RDF-3X when a query graph is complex, such as Q_1, Q_2, Q_3 and Q_7. Since these queries do not contain selective triple patterns and the query graph structure is complex, the search space for these queries is very large. Our method can take advantage of parallel processing and reduce query response time significantly relative to a centralized system. If the queries (Q_4, Q_5 and Q_6) contain selective triple patterns, the search space is small. The centralized system (RDF-3X) is faster than our method in these queries, since our approach spends more communication cost between different machines. These queries only spend less than 1-3 seconds in both RDF-3X and our distributed system. However, for some challenging queries (such as Q_1, Q_2, Q_3 and Q_7), our method outperforms RDF-3X significantly. For example, RDF-3X spends about 1000 seconds in Q_1, while our approach only spends about 300 seconds. The performance advantage of our distributed system is more clear in these challenging queries.

8 Conclusion

In this paper, we propose a graph-based approach to distributed SPARQL query processing that adopts the partial evaluation and assembly approach. This is a two-step process. In the first step, we evaluate a query Q on each graph fragment in parallel to find *local partial matches*, which, intuitively, is the overlapping part between a crossing match and a fragment. The second step is to assemble these local



(a) WatDiv 1B

(b) LUBM 10000

(c) BTC

Fig. 16 Online Performance Comparison

partial matches to compute crossing matches. Two different assembly strategies are proposed in this work: *centralized assembly*, where all local partial matches are sent to a single site; and *distributed assembly*, where the local partial matches are assembled at a number of sites in parallel.

The main benefits of our method are twofold: First, our solution is partition-agnostic as opposed to existing partition-based methods each of which depends on a particular RDF graph partition strategy, which may be infeasible to enforce in certain circumstances. Our method is, therefore, much more flexible. Second, compared with other partition-based methods, the number of involved vertices and edges in the intermediate results are minimized in our method, which are proven theoretically and demonstrated experimentally.

There are a number of extensions we are currently working on. An important one is handling SPARQL queries over linked open data (LOD). We can treat the interconnected RDF repositories (in LOD) as a virtually integrated distributed database. Some RDF repositories provide SPARQL endpoints and others may not have query capability. Therefore, data at these sites need to be moved for processing that will affect the algorithm and cost functions. Furthermore, multiple SPARQL query optimization in the context of distributed RDF graphs is also an ongoing work. In real applications, queries in the same time are commonly overlapped. Thus, there is much room for sharing computation when executing these queries. This observation motivates us to revisit the classical problem of multi-query optimization in the context of distributed RDF graphs.

References

1. D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *VLDB J.*, 18(2):385–406, 2009.
2. G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *Proc. 13th Int. Semantic Web Conf.*, pages 197–212, 2014.
3. M. M. Astrahan, H. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1:97–137, 1976.
4. M. Atre. *Left Bit Right*: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1793–1808, 2015.
5. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In *Proc. 19th Int. World Wide Web Conf.*, pages 41–50, 2010.
6. P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proc. 32nd Int. Conf. on Very Large Data Bases*, pages 211–222, 2006.
7. G. Cong, W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 509–520, 2007.
8. G. Cong, W. Fan, A. Kementsietsidis, J. Li, and X. Liu. Partial evaluation for distributed XPath query processing and beyond. *ACM Trans. Database Syst.*, 37(4):Article No. 32, 2012.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.

10. R. G. Downey, M. R. Fellows, A. Vardy, and G. Whittle. The parametrized complexity of some fundamental problems in coding theory. *SIAM J. on Comput.*, 29(2):545–570, 1999.
11. M. E. Dyer and C. S. Greenhill. The complexity of counting graph homomorphisms. *Random Struct. Algorithms*, 17(3-4):260–289, 2000.
12. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *Proc. VLDB Endowment*, 3(1):264–275, 2010.
13. W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *Proc. VLDB Endowment*, 5(11):1304–1315, 2012.
14. W. Fan, X. Wang, Y. Wu, and D. Deng. Distributed graph simulation: Impossibility and possibility. *Proc. VLDB Endowment*, 7(12):1083–1094, 2014.
15. L. Galarraga, K. Hose, and R. Schenkel. Partout: a distributed engine for efficient RDF processing. In *Proc. 23rd Int. World Wide Web Conf. (Companion Volume)*, pages 267–268, 2014.
16. O. Görlitz and S. Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proc. ISWC 2011 Workshop on Consuming Linked Data*, 2011.
17. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semantics*, 3(2-3):158–182, 2005.
18. S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 289–300, 2014.
19. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *Proc. 19th Int. World Wide Web Conf.*, pages 411–420, 2010.
20. O. Hartig and M. T. Özsu. Linked data query processing (Tutorial). In *Proc. 30th Int. Conf. on Data Engineering*, pages 1286–1289, 2014.
21. K. Hose and R. Schenkel. WARP: Workload-aware replication and partitioning for RDF. In *Proc. Workshops of 29th Int. Conf. on Data Engineering*, pages 1–6, 2013.
22. J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endowment*, 4(11):1123–1134, 2011.
23. M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. and Data Eng.*, 23(9):1312–1327, 2011.
24. N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.
25. Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1):67–91, 2015.
26. G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Proc. ACM/IEEE Conf. on Supercomputing*, 1995. Article No. 29.
27. V. Khadilkar, M. Kantarcioglu, B. M. Thuraisingham, and P. Castagna. Jena-HBase: A distributed, scalable and efficient RDF triple store. In *Proc. International Semantic Web Conference Posters & Demos Track*, 2012.
28. K. Lee and L. Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *Proc. VLDB Endowment*, 6(14):1894–1905, 2013.
29. K. Lee, L. Liu, Y. Tang, Q. Zhang, and Y. Zhou. Efficient and customizable data partitioning framework for distributed big RDF data processing in the cloud. In *Proc. IEEE 6th Int. Conf. on Cloud Computing*, pages 327–334, 2013.
30. F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using MapReduce. *ACM Comput. Surv.*, 46(3):Article No. 31, 2014.
31. S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *Proc. 21st Int. World Wide Web Conf.*, pages 949–958, 2012.
32. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endowment*, 1(1):647–659, 2008.
33. N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H_2RDF: adaptive query processing on RDF data in the cloud. In *Proc. 21st Int. World Wide Web Conf. (Companion Volume)*, pages 397–400, 2012.
34. N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H_2RDF+: an efficient data management system for big RDF graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 909–912, 2014.
35. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(Article No. 3), 2009.
36. B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *Proc. 5th European Semantic Web Conf.*, pages 524–538, 2008.
37. K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *Proc. Int. Workshop on Programming Support Innovations for Emerging Distributed Applications*, 2010. Article No. 4.
38. M. Saleem and A. N. Ngomo. HiBISCuS: Hypergraph-based source selection for sparql endpoint federation. In *Proc. 11th Extended Semantic Web Conf.*, pages 176–191, 2014.
39. M. Saleem, S. S. Padmanabhuni, A. N. Ngomo, A. Iqbal, J. S. Almeida, S. Decker, and H. F. Deus. TopFed: TCGA tailored federated query processing and linking to LOD. *J. Biomedical Semantics*, 5:47, 2014.
40. M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of best data practices in different topical domains. In *Proc. 13th Int. Semantic Web Conf.*, pages 245–260, 2014.
41. M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A benchmark suite for federated semantic data query processing. In *Proc. 10th Int. Semantic Web Conf.*, pages 585–600, 2011.
42. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *Proc. 10th Int. Semantic Web Conf.*, pages 601–616, 2011.
43. H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endowment*, 1(1):364–375, 2008.
44. B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 505–516, 2013.
45. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
46. L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *Proc. 30th Int. Conf. on Data Engineering*, pages 568–579, 2014.
47. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *Proc. VLDB Endowment*, 6(4):265–276, 2013.
48. X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: towards scalable I/O efficient SPARQL query evaluation on the cloud. In *Proc. 29th Int. Conf. on Data Engineering*, pages 565–576, 2013.
49. X. Zhang, L. Chen, and M. Wang. Towards efficient join processing over large RDF graph using mapreduce. In *Proc. 24th Int. Conf. on Scientific and Statistical Database Management*, pages 250–259, 2012.
50. 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.

Online Supplements

A Queries in Experiments

Table 10, 11, 12, 13 and 15 show all queries used in the paper.

B Results of 14 Benchmark Queries over LUBM 1000

Table 16 shows the experimental results of each stage for original 14 LUBM benchmark queries listed in Table 14. Note that, since the current version of gStore does not currently support type reasoning, we revised the original 14 to remove type reasoning. The resulting queries return larger result sets since there is no filtering as a result of type reasoning.

Generally speaking, many of these queries are simple pattern queries or they are quite similar to each other, and the 7 queries chosen by [5, 50] are representative queries out of this list. Hence, the results of the 14 benchmark queries are also similar to the results of our 7 representative queries in Table 3.

C Exp 4 – Expanded Results on MapReduce Effect

Existing partition-based solutions, such as GraphPartition and TripleGroup, use MapReduce jobs to join intermediate results to find SPARQL matches. In order to evaluate the cost of MapReduce-jobs, we perform the following experiments over WatDiv 100M. We revise join processing in both GraphPartition and TripleGroup, by applying joins where intermediate results are sent to a central server using MPI. We use WatDiv 100M and only consider the benchmark queries that need join processing (L_1, L_2, L_5, F_1, F_2, F_3, F_3, F_4, F_5, C_1 and C_2) in our experiments. Moreover, all partition-based methods generate intermediate results and merge them at a central sever that shares the same framework with PECA, so we only compare them with PECA.

Tables 17 and 18 show the performance of the three approaches. Our technique is always faster regardless of the use of MPI or MapReduce-based join. This is because our method produces smaller intermediate result sets; MapReduce-based join dominates the query cost.

Tables 17 and 18 demonstrate that our partial evaluation process is more expensive in evaluating local queries than GraphPartition and TripleGroup in many cases. This is easy to understand – since the subquery structures in GraphPartition and TripleGroup are fixed, such as stars, it is cheaper to find these local query results than finding local partial matches.

Our system generally outperforms GraphPartition and TripleGroup significantly if they use MapReduce-based join. Even when GraphPartition and TripleGroup use distributed joins, our system is still faster than them in most cases (8 out of 10 queries in this experiment).

D Comparisons with Memory Systems

The comparison results between our system with two typical distributed memory systems are given in Table 19.

E Comparisons with Federated Systems

Comparisons with federated systems is done using the FedBench benchmark [41]. The partitioning of this benchmark is pre-defined and fixed. FedBench includes 6 real cross domain RDF datasets and 4 real life science domain RDF datasets, and each dataset maps to one fragment. In this benchmark, 7 federated queries are defined for cross domain

L_1	#mapping v1 wsdbm:Website uniform SELECT ?v0 ?v2 ?v3 WHERE { ?v0 wsdbm:subscribes %v1% . ?v2 sorg:caption ?v3 . ?v0 wsdbm:likes ?v2 . }
L_2	#mapping v0 wsdbm:City uniform SELECT ?v1 ?v2 WHERE { %v0% gn:parentCountry ?v1 . ?v2 wsdbm:likes wsdbm:Product0 . ?v2 sorg:nationality ?v1 . }
L_3	#mapping v2 wsdbm:Website uniform SELECT ?v0 ?v1 WHERE { ?v0 wsdbm:likes ?v1 . ?v0 wsdbm:subscribes %v2% . }
L_4	#mapping v1 wsdbm:Topic uniform SELECT ?v0 ?v2 WHERE { ?v0 og:tag %v1% . ?v0 sorg:caption ?v2 . }
L_5	#mapping v2 wsdbm:City uniform SELECT ?v0 ?v1 ?v3 WHERE { ?v0 sorg:jobTitle ?v1 . %v2% gn:parentCountry ?v3 . ?v0 sorg:nationality ?v3 . }
S_1	#mapping v2 wsdbm:Retailer uniform SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 WHERE { ?v0 gr:includes ?v1 . %v2% gr:offers ?v0 . ?v0 gr:price ?v3 . ?v0 gr:serial- Number ?v4 . ?v0 gr:validFrom ?v5 . ?v0 gr:validThrough ?v6 . v0 sorg:eligibleQuantity ?v7 . ?v0 sorg:eligibleRegion ?v8 . ?v0 sorg:priceValidUntil ?v9 . }
S_2	#mapping v2 wsdbm:Country uniform SELECT ?v0 ?v1 ?v3 WHERE { ?v0 dc:Location ?v1 . ?v0 sorg:nationality %v2% . ?v0 wsdbm:gender ?v3 . ?v0 rdf:type wsdbm:Role2 . }
S_3	#mapping v1 wsdbm:ProductCategory uniform SELECT ?v0 ?v2 ?v3 ?v4 WHERE { ?v0 rdf:type %v1% . ?v0 sorg:caption ?v2 . ?v0 wsdbm:hasGenre ?v3 . ?v0 sorg:publisher ?v4 . }
S_4	#mapping v1 wsdbm:AgeGroup uniform SELECT ?v0 ?v2 ?v3 WHERE { ?v0 foaf:age %v1% . ?v0 foaf:familyName ?v2 . ?v3 mo:artist ?v0 . ?v0 sorg:nationality wsdbm:Country1 . }
S_5	#mapping v1 wsdbm:ProductCategory uniform SELECT ?v0 ?v2 ?v3 WHERE { ?v0 rdf:type %v1% . ?v0 sorg:description ?v2 . ?v0 sorg:keywords ?v3 . ?v0 sorg:language wsdbm:Language0 . }
S_6	#mapping v3 wsdbm:SubGenre uniform SELECT ?v0 ?v1 ?v2 WHERE { ?v0 mo:conductor ?v1 . ?v0 rdf:type ?v2 . ?v0 wsdbm:hasGenre %v3% . }
S_7	#mapping v3 wsdbm:User uniform SELECT ?v0 ?v1 ?v2 WHERE { ?v0 rdf:type ?v1 . ?v0 sorg:text ?v2 . %v3% wsdbm:likes ?v0 . }
F_1	#mapping v1 wsdbm:Topic uniform SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE { ?v0 og:tag %v1% . ?v0 rdf:type ?v2 . ?v3 sorg:trailer ?v4 . ?v3 sorg:keywords ?v5 . ?v3 wsdbm:hasGenre ?v0 . ?v3 rdf:type wsdbm:ProductCategory2 . }
F_2	#mapping v8 wsdbm:SubGenre uniform SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 WHERE { ?v0 foaf:homepage ?v1 . ?v0 og:title ?v2 . ?v0 rdf:type ?v3 . ?v0 sorg:caption ?v4 . ?v0 sorg:description ?v5 . ?v1 sorg:url ?v6 . ?v1 wsdbm:hits ?v7 . ?v0 wsdbm:hasGenre %v8% . }
F_3	#mapping v3 wsdbm:SubGenre uniform SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE { ?v0 sorg:contentRating ?v1 . ?v0 sorg:contentSize ?v2 . ?v0 wsdbm:hasGenre %v3% . ?v4 wsdbm:makesPurchase ?v5 . ?v5 wsdbm:purchaseDate ?v6 . ?v5 wsdbm:purchaseFor ?v0 . }
F_4	#mapping v3 wsdbm:Topic uniform SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE { ?v0 foaf:home- page ?v1 . ?v2 gr:includes ?v0 . ?v0 og:tag %v3% . ?v0 sorg:description ?v4 . ?v0 sorg:contentSize ?v8 . ?v1 sorg:url ?v5 . ?v1 wsdbm:hits ?v6 . ?v1 sorg:language wsdbm:Language0 . ?v7 wsdbm:likes ?v0 . }
F_5	#mapping v2 wsdbm:Retailer uniform SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE { ?v0 gr:includes ?v1 . %v2% gr:offers ?v0 . ?v0 gr:price ?v3 . ?v0 gr:validThrough ?v4 . ?v1 og:title ?v5 . ?v1 rdf:type ?v6 . }
C_1	SELECT ?v0 ?v4 ?v6 ?v7 WHERE { ?v0 sorg:caption ?v1 . ?v0 sorg:text ?v2 . ?v0 sorg:contentRating ?v3 . ?v0 rev:hasReview ?v4 . ?v4 rev:title ?v5 . ?v4 rev:reviewer ?v6 . ?v7 sorg:actor ?v6 . ?v7 sorg:language ?v8 . }
C_2	SELECT ?v0 ?v3 ?v4 ?v8 WHERE { ?v0 sorg:legalName ?v1 . ?v0 gr:offers ?v2 . ?v2 sorg:eligibleRegion wsbm:Country5 . ?v2 gr:includes ?v3 . ?v4 sorg:jobTitle ?v5 . ?v4 foaf:homepage ?v6 . ?v4 wsdbm:makesPurchase ?v7 . ?v7 wsdbm:purchaseFor ?v3 . ?v3 rev:has Review ?v8 . ?v8 rev:totalVotes ?v9 . }
C_3	SELECT ?v0 WHERE { ?v0 wsdbm:likes ?v1 . ?v0 wsdbm:friendOf ?v2 . ?v0 dc:Location ?v3 . ?v0 foaf:age ?v4 . ?v0 wsdbm:gender ?v5 . ?v0 foaf:givenName ?v6 . }

Table 10 WatDiv Queries

		Query		Partial Evaluation			Assembly			Total			# of LPMFs	# of CMFs
				Time(in ms)	# of LPMs[2]	# of IMs[3]	Time(in ms) Centralized	Time(in ms) Distributed	# of CMs[4]	Time(in ms) PECA[5]	Time(in ms) PEDA[6]	# of Matches[7]		
Edge		Q5	√[1]	191	0	678	0	0	0	191	191	678	0	0
		Q6		13648	0	7924765	0	0	0	13648	13648	7924765	0	0
		Q13	√	231	0	3295	0	0	0	231	231	3295	0	0
		Q14		13646	0	7924765	0	0	0	13646	13646	7924765	0	0
Star		Q1	√	191	0	1081187	0	0	0	191	191	1081187	0	0
		Q3	√	120	0	4	0	0	0	120	120	4	0	0
		Q4	√	85	0	10	0	0	0	85	85	10	0	0
		Q10	√	190	0	4	0	0	0	190	190	4	0	0
		Q12	√	127	0	540	0	0	0	127	127	540	0	0
Linear		Q11	√	42	0	4	0	0	0	42	42	4	0	0
Snowflake		Q7	√	1660	60172	58	1338	1415	1	2998	3075	59	10	10
		Q8	√	1132	20050	5901	654	728	15	1786	1860	5916	10	10
Complex		Q2		52712	18035	2506	639	439	22	53351	53151	2528	10	10
		Q9		3328	12049	44086	763	362	104	4091	3690	44190	10	10

Table 16 Evaluation of Each Stage for 14 Benchmark Queries over LUBM 1000

	PECA			GraphPartition			MPI-revised GraphPartition	
	Partial Evaluation	Assembly	Total Time	Finding Partial Matches	MapReduce-based Join	MapReduce-based Total Time	MPI-based Join	MPI-based Total Time
L_1	2350	1	2351	1423	19570	20993	183	**1606**
L_2	557	1	**558**	386	16420	16806	204	590
L_5	524	2	**526**	479	27480	27959	76	555
F_1	3906	1	**3907**	4011	36200	40211	35	4046
F_2	2659	31	**2690**	2466	58180	60646	1277	3743
F_3	16077	1945	**18022**	14136	61400	75536	4191	18327
F_4	21446	47	21493	15535	34060	49595	165	**15700**
F_5	9043	43	**9086**	9910	51110	61020	1900	11810
C_1	12969	52	**13021**	9799	223670	233469	18522	28321
C_2	37850	1454	**39304**	44998	2168300	2213298	19494	64492

Table 17 Query Response Time over Partitioning Strategy of GraphPartition (in milliseconds)

	PECA			TripleGroup			MPI-revised TripleGroup	
	Partial Evaluation	Assembly	Total Time	Finding Partial Matches	MapReduce-based Join	MapReduce-based Total Time	MPI-based Join	MPI-based Total Time
L_1	2250	1	2251	1122	20840	21962	452	**1574**
L_2	249	1	**250**	204	16150	16354	50	254
L_5	737	2	739	304	27550	27854	70	**374**
F_1	5753	1	**5753**	4413	36230	40643	1538	5951
F_2	4771	21	**4792**	3909	40700	44609	911	4820
F_3	10425	3174	**12599**	10517	62440	72957	5346	15863
F_4	16373	66	**16439**	15403	54054	69457	1212	16615
F_5	11611	22	**11633**	13039	22180	35219	4923	17962
C_1	12794	2265	**15059**	6057	216720	222777	12194	18251
C_2	44272	8870	**53142**	48204	1954800	2003004	15062	63266

Table 18 Query Response Time over Partitioning Strategy of TripleGroup (in milliseconds)

Q_1	SELECT ?x,?y,?z WHERE { ?x rdf:type ub:GraduateStudent.?y rdf:type ub:University.?z rdf:type ub:Department.?x ub:memberOf ?z.?z ub:subOrganizationOf ?y.?x ub:undergraduateDegreeFrom ?y}
Q_2	SELECT ?x WHERE { ?x rdf:type ub:Course.?x ub:name ?y. }
Q_3	SELECT ?x,?y,?z WHERE { ?x rdf:type ub:UndergraduateStudent. ?y rdf:type ub:University. ?z rdf:type ub:Department. ?x ub:memberOf ?z. ?z ub:subOrganizationOf ?y. ?x ub:undergraduateDegreeFrom ?y}
Q_4	SELECT ?x WHERE { ?x ub:worksFor http://www.Department0. University0.edu. ?x rdf:type ub:FullProfessor. ?x ub:name ?y1. ?x ub: emailAddress ?y2. ?x ub:telephone ?y3. }
Q_5	SELECT ?x WHERE { ?x ub:subOrganizationOf http://www. Department0.University0.edu. ?x rdf:type ub:ResearchGroup. }
Q_6	SELECT ?x,?y WHERE { ?y rdf:type ub:Department. ?y ub:subOrganizationOf http://www.University0.edu. ?x ub:worksFor ?y. ?x rdf:type ub:FullProfessor. }
Q_7	SELECT ?x,?y,?z WHERE { ?x rdf:type ub:UndergraduateStudent. ?y rdf:type ub:FullProfessor. ?z rdf:type ub:Course. ?x ub:advisor ?y. ?x ub:takesCourse ?z. ?y ub:teacherOf ?z. }

Table 11 LUBM Queries

RDF datasets, and 7 federated queries are defined for life science RDF datasets. The characteristics of FedEx dataset are given in Table 20.

FedBench also provides 7 benchmark queries for Cross Domain and 7 queries for Life Science. However, these queries are highly homogeneous. They are all snowflake queries (several stars linked by a path) and they all contain selective triple patterns [11]. To test the performance against different query structures, we introduce five other queries for each domain. Two of them (ECD_1 and ECD_2 for Cross Domain and ELS_1 and ELS_2 for Life Science) are stars; one (ECD_3 for Cross Domain and ELS_3 for Life Science) is snowflake, and two (ECD_4 and ECD_5 for Cross Domain and ELS_4 and ELS_5 for Life Science) are complex queries. Furthermore, some queries (ECD_1, ECD_4, ELS_1 and ELS_4) contain selective triple patterns, while the others (ECD_2, ECD_3, ECD_5, ELS_2, ELS_3 and ELS_5) do not. We report the experimental results in the Figure 17.

[11] As defined before, a triple pattern is "selective" if it has no more than 100 matches in RDF graph G.

CD_1	SELECT ?predicate ?object WHERE { { <http://dbpedia.org/resource/Barack_Obama> ?predicate ?object } UNION { ?subject <http://www.w3.org/2002/07/owl#sameAs> <http://dbpedia.org/resource/Barack_Obama> . ?subject ?predicate ?object } }
CD_2	SELECT ?party ?page WHERE { <http://dbpedia.org/resource/Barack_Obama> <http://dbpedia.org/ontology/party> ?party . ?x <http:// data.nytimes.com/elements/topicPage> ?page . ?x <http://www.w3.org/2002/07/owl# sameAs> <http://dbpedia.org/resource/Barack_Obama> . }
CD_3	SELECT ?president ?party ?page WHERE { ?president <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/President> . ?president <http://dbpedia.org/ontology/nationality> <http://dbpedia.org/resource/United_States> . ?president <http://dbpedia.org/ontology/party> ?party . ?x <http://data.nytimes.com/elements/topicPage> ?page . ?x <http://www. w3.org/2002/07/owl#sameAs> ?president . }
CD_4	SELECT ?actor ?news WHERE { ?film <http://purl.org/dc/terms/title> “Tarzan” . ?film <http://data.linkedmdb.org/resource/movie/actor> ?actor . ?actor <http://www.w3.org/2002/07/owl#sameAs> ?x. ?y <http://www.w3.org/2002/07/owl#sameAs> ?x . ?y <http://data.nytimes.com/elements/topicPage> ?news }
CD_5	SELECT ?film ?director ?genre WHERE { ?film <http://dbpedia.org/ontology/director> ?director . ?director <http://dbpedia.org/ontology/nationality> <http://dbpedia.org/resource/Italy> . ?x <http://www.w3.org/2002/07/owl#sameAs> ?film . ?x <http://data.linkedmdb.org/resource/movie/genre> ?genre . }
CD_6	SELECT ?name ?location ?news WHERE { ?artist <http://xmlns.com/foaf/0.1/name> ?name . ?artist <http://xmlns.com/foaf/0.1/based_near> ?location . ?location <http://www.geonames.org/ontology#parentFeature> ?germany . ?germany <http://www.geonames.org/ontology#name> “Federal Republic of Germany” }
CD_7	SELECT ?location ?news WHERE { ?location <http://www. geonames.org/ontology#parentFeature> ?parent . ?parent <http://www. geonames.org/ontology#name> “California” . ?y <http://www.w3.org/ 2002/07/owl#sameAs> ?location . ?y <http://data.nytimes.com/elements/ topicPage> ?news }
ECD_1	SELECT ?name WHERE { <http://data.semanticweb.org/conference/www/2008> <http://xmlns.com/foaf/0.1/based_near> ?location . ?location <http://www.geonames.org/ontology#name> ?name .}
ECD_2	SELECT ?actor WHERE { ?actor <http://www.w3.org/2002/07/owl#sameAs> ?x . ?actor <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://data.linkedmdb.org/resource/movie/actor> . }
ECD_3	SELECT ?m ?c WHERE { ?a <http://dbpedia.org/ontology/residence> ?c . ?c <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Place> . ?a1 <http://www.w3.org/2002/07/owl#sameAs> ?a . ?m <http://data.linkedmdb.org/resource/movie/actor> ?a1 .}
ECD_4	SELECT ?x ?y WHERE { ?gplace <http://www.geonames.org/ontology#alternateName> "Philadelphia"@en . ?gplace <http://www.w3.org/2002/07/owl#sameAs> ?place . ?x <http://dbpedia.org/ontology/birthPlace> ?place . ?x <http://dbpedia.org/ontology/spouse> ?y . ?z <http://dbpedia.org/ontology/starring> ?x . ?z <http://dbpedia.org/ontology/starring> ?y . ?y1 <http://www.w3.org/2002/07/owl#sameAs> ?y . ?y1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://data.linkedmdb.org/resource/movie/actor> .}
ECD_5	Select ?n1 ?n2 ?gn where { ?a1 <http://xmlns.com/foaf/0.1/name> ?n1 . ?p2 <http://www.w3.org/2002/07/owl#sameAs> ?a2 . ?p2 <http://data.linkedmdb.org/resource/movie/actor_name> ?n2 . ?a1 <http://dbpedia.org/ontology/award> ?award . ?a2 <http://dbpedia.org/ontology/award> ?award . ?a1 <http://dbpedia.org/ontology/birthPlace> ?city . ?a2 <http://dbpedia.org/ontology/birthPlace> ?city . ?gc <http://www.w3.org/2002/07/owl#sameAs> ?city . ?gc <http://www.geonames.org/ontology#name> ?gn .}

Table 12 FedBench Queries in Cross Domain

LS_1	SELECT $drug $melt WHERE { { $drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/meltingPoint> $melt. } UNION { $drug <http://dbpedia.org/ontology/Drug/meltingPoint> $melt . } }
LS_2	SELECT ?predicate ?object WHERE { { <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00201> ?predicate ?object . } UNION { <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00201> <http://www.w3.org/2002/07/owl#sameAs> ?caff . ?caff ?predicate ?object . } }
LS_3	SELECT ?Drug ?IntDrug ?IntEffect WHERE { ?Drug <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Drug> . ?y <http://www.w3.org/2002/07/owl#sameAs> ?Drug . ?Int <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/ interactionDrug1> ?y . ?Int <http://www4.wiwiss.fu-berlin.de/drugbank/ resource/drugbank/interactionDrug2> ?IntDrug . ?Int <http://www4.wi wiss.fu-berlin.de/drugbank/resource/drugbank/text> ?IntEffect . }
LS_4	SELECT ?drugDesc ?cpd ?equation WHERE { ?drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/drugCategory> <http://www4. wiwiss.fu-berlin.de/drugbank/resource/drugcategory/cathartics> . ?drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/keggCompoundId> ?cpd . ?drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/description> ?drugDesc . ?enzyme <http://bio2rdf.org/ns/kegg#xSubstrate> ?cpd . ?enzyme <http://www.w3.org/ 1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/ns/kegg#Enzyme> . ?reaction <http://bio2rdf.org/ns/kegg#xEnzyme> ?enzyme . ?reaction <http://bio2rdf.org/ns/kegg#equation> ?equation . }
LS_5	SELECT ?drug ?keggUrl ?chebiImage WHERE { ?drug <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/drugs> . ?drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/keggCompoundId> ?keggDrug . ?keggDrug <http://bio2rdf.org/ns/bio2rdf#url> ?keggUrl . ?drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/genericName> ?drugBankName . ?chebiDrug <http://purl.org/dc/ elements/1.1/title> ?drugBankName . ?chebiDrug <http://bio2rdf.org/ns/bio2rdf#image> ?chebiImage . }
LS_6	SELECT ?drug ?title WHERE { ?drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/drugCategory> <http://www4.wi wiss.fu-berlin.de/drugbank/resource/drugcategory/micronutrient> . ?drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/casRegistryNumber> ?id . ?keggDrug <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/ns/kegg#Drug> . ?keggDrug <http://bio2rdf.org/ns/bio2rdf#xRef> ?id . ?keggDrug <http://purl.org/dc /elements/1.1/ title> ?title . }
LS_7	SELECT $drug $transform $mass WHERE { { $drug <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/affectedOrganism> ’Humans and other mammals’. $drug <http://www4.wiwiss.fu-berlin.de/drug bank/resource/drugbank/casRegistryNumber> $cas . $kegg Drug <http://bio2rdf.org/ns/bio2rdf#xRef> $cas . $keggDrug <http://bio2rdf.org/ns/bio2rdf #mass> $mass FILTER ($mass > ’5’) } OPTIONAL { $drug <http://www4.wiwiss.fu-berlin.de/drug bank/resource/drugbank/biotransformation> $trans form . } }
ELS_1	SELECT ?num WHERE { <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00918> <http://www.w3.org/2002/07/owl#sameAs> ?x . ?x <http://dbpedia.org/ontology/casNumber> ?num .}
ELS_2	SELECT ?y WHERE { ?x <http://bio2rdf.org/ns/bio2rdf#xRef> ?y . ?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/ns/kegg#Compound> .}
ELS_3	SELECT ?n ?t WHERE { ?Drug <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Drug> . ?y <http://www.w3.org/2002/ 07/owl#sameAs> ?Drug . ?y <http://www4.wiwiss.fu-berlin.de/ drugbank/resource/drugbank/brandName> ?n . ?y <http://www4.wiwiss.fu-berlin.de/drugbank/ resource/drugbank/drugType> ?t .}
ELS_4	SELECT ?parent WHERE { ?y <http://bio2rdf.org/ns/bio2rdf#formula> "C8H10NO6P" . ?z http://bio2rdf.org/ns/chebi#has_functional_parent> ?parent . ?x <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/keggCompoundId> ?y . ?x <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/chebiId> ?z . ?y <http://bio2rdf.org/ns/bio2rdf#xRef> ?z .}
ELS_5	SELECT ?y1 ?label WHERE { ?x1 <http://bio2rdf.org/ns/chebi#has_role> ?x2 . ?x1 <http://bio2rdf.org/ns/chebi#has_role> ?x3 . ?x2 <http://bio2rdf.org/ns/chebi#is_a> ?x4 . ?x3 <http://bio2rdf.org/ns/chebi#is_a> ?x4 . ?y1 <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/chebiId> ?x1 . ?y1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/drugs> . ?x4 <http://www.w3.org/2000/01/rdf-schema#label> ?label . }

Table 13 FedBench Queries in Life Science Domain

Q_1	SELECT ?x WHERE { ?x rdf:type ub:GraduateStudent. ?x ub:takesCourse http://www.Department0.University0.edu/GraduateCourse0}
Q_2	SELECT ?x,?y where { ?x rdf:type ub:GraduateStudent. ?y rdf:type ub:University. ?z rdf:type ub:Department. ?x ub:memberOf ?z. ?z ub:subOrganizationOf ?y. ?x ub:undergraduateDegreeFrom ?y}
Q_3	SELECT ?x WHERE { ?x rdf:type ub:Publication. ?x ub:publicationAuthor http://www.Department0.University0.edu/AssistantProfessor0}
Q_4	SELECT ?x,?y1,?y3 WHERE { ?x ub:worksFor http://www.Department0.University0.edu. ?x ub:name ?y1. ?x ub:emailAddress ?y2. ?x ub:telephone ?y3.}
Q_5*	SELECT ?x WHERE { ?x ub:memberOf http://www.Department0.University0.edu.}
	(removing the triple pattern "?x rdf:type ub:Person" with type reasoning)
Q_6*	SELECT ?x WHERE { ?x rdf:type ub:UndergraduateStudent.}
	(modifying the triple pattern "?x rdf:type ub:Student" with type reasoning to "?x rdf:type ub:UndergraduateStudent")
Q_7*	SELECT ?x,?y WHERE { ?x rdf:type ub:UndergraduateStudent. ?x ub:takesCourse ?y. ?y rdf:type ub:Course. http://www.Department0.University0.edu//AssociateProfessor0 ub:teacherOf ?y.}
	(modifying the triple pattern "?x rdf:type ub:Student" with type reasoning to "?x rdf:type ub:UndergraduateStudent")
Q_8*	SELECT ?x,?y WHERE { ?x rdf:type ub:UndergraduateStudent. ?x ub:memberOf ?y. ?x ub:emailAddress ?z. ?y rdf:type ub:Department. ?y ub:subOrganizationOf http://www.University0.edu}
	(modifying the triple pattern "?x rdf:type ub:Student" with type reasoning to "?x rdf:type ub:UndergraduateStudent")
Q_9*	SELECT ?x,?y WHERE { ?x rdf:type ub:UndergraduateStudent. ?y rdf:type ub:FullProfessor. ?z rdf:type ub:Course. ?x ub:advisor ?y. ?x ub:takesCourse ?z. ?y ub:teacherOf ?z. }
	(modifying the triple pattern "?x rdf:type ub:Student" with type reasoning to "?x rdf:type ub:UndergraduateStudent" and the triple pattern "?x rdf:type ub:Faculty" with type reasoning to "?x rdf:type ub:FullProfessor")
Q_{10}*	SELECT ?x WHERE { ?x rdf:type ub:GraduateStudent. ?x ub:takesCourse http://www.Department0.University0.edu/GraduateCourse0.}
	(modifying the triple pattern "?x rdf:type ub:Student" with type reasoning to "?x rdf:type ub:UndergraduateStudent")
Q_{11}	SELECT ?x,?y WHERE { ?x rdf:type ub:ResearchGroup. ?x ub:subOrganizationOf ?y. ?y ub:subOrganizationOf http://www.University0.edu.}
Q_{12}*	SELECT ?x,?y WHERE { ?x ub:headOf ?y. ?y rdf:type ub:Department. ?y ub:subOrganizationOf http://www.University0.edu.}
	(removing the triple pattern "?x rdf:type ub:Chair" with type reasoning)
Q_{13}*	SELECT ?x WHERE { ?x ub:undergraduateDegreeFrom http://www.University0.edu.}
	(removing the triple pattern "?x rdf:type ub:Person" with type reasoning and modifying the triple pattern "http://www.University0.edu ub:hasAlumnus ?x" with property reasoning to "?x ub:undergraduateDegreeFrom http://www.University0.edu")
Q_{14}	SELECT ?x WHERE { ?x rdf:type ub:UndergraduateStudent.}

* means that the query removes or modifys some triple patterns with reasoning.

Table 14 Original SPARQL Queries in LUBM

We compare our approach against two systems: FedX [42] and SPLENDID [16]. Our approach has a superior performance in most cases, especially when the queries do not contain selective triple patterns. FedX and SPLENDID decompose a SPARQL query 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 employs the bound join and SPLENDID uses hash join. This means that these systems first use the intermediate results to rewrite the subquery with bound join variables. Then, the rewritten queries are evaluated at the corresponding sites. Therefore, when the SPARQL queries do not contain any selective triple pattern, the size of intermediate results is so large that evaluation of bound joins takes significant time. In this case, our system outperforms them by orders of magnitude. For example, in ECD_2, ECD_3, ECD_5, ELS_2, ELS_3 and

Q_1	SELECT ?lat,?long WHERE { ?a [] "Eiffel Tower"@en. ?a <http://www .w3.org/2003/01/geo/wgs84_pos#lat> ?lat. ?a <http://www.w3.org/2003 /01/geo/wgs84_pos#long> ?long. ?a <http://dbpedia.org/ontology/location> <http://dbpedia.org/resource/France>.}
Q_2	SELECT ?b,?p,?bn WHERE { ?a [] "Tim Berners-Lee"@en. ?a <http:// dbpedia.org/property/dateOfBirth> ?b. ?a <http://dbpedia.org/ property/placeOfBirth> ?p. ?a <http://dbpedia.org/property/name> ?bn.}
Q_3	SELECT ?t,?lat,?long WHERE { ?a <http://dbpedia.org/property/wikiLinks> <http://dbpedia.org/resource/List_of_World_Heritage_Sites _in_Europe>. ?a <http://dbpedia.org/property/title> ?t. ?a <http://www .w3.org/2003/01/geo/wgs84_pos#lat> ?lat. ?a <http://www.w3.org/2003/ 01/geo/wgs84_pos#long> ?long. ?a <http://dbpedia.org/property/wikiLinks> <http://dbpedia.org/resource/Middle_Ages>.}
Q_4	SELECT ?l,?long,?lat WHERE { ?p <http://dbpedia.org/property/ name> "Krebs, Emil"@en. ?p <http://dbpedia.org/property/deathPlace> ?l. ?c [] ?l. ?c <http://www.geonames.org/ontology#featureClass> <http:// www.geonames.org/ontology#P>. ?c <http://www.geonames.org/onto- logy#inCountry> <http://www.geonames.org/countries/#DE>. ?c <http:// www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat. ?c <http://www.w3.org/ 2003/01/geo/wgs84_pos#long> ?long.}
Q_5	SELECT distinct ?l,?long,?lat WHERE { ?a [] "Barack Obama"@en. ?a <http://dbpedia.org/property/placeOfBirth> ?l. ?l <http:// www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat. ?l <http://www.w3.org/ 2003/01/geo/wgs84_pos#long> ?long.}
Q_6	SELECT distinct ?d WHERE { ?a <http://dbpedia.org/property/ senators> ?c. ?a <http://dbpedia.org/property/name> ?d. ?c <http://dbpedia.org/property/profession> <http://dbpedia.org/resource/Veterinarian>. ?a <http://www.w3.org/2002/07/owl#sameAs> ?b. ?b <http://www.geonames.org/ontology#inCountry> <http://www.geonames.org/countries/#US>.}
Q_7	SELECT distinct ?a,?b,?lat,?long WHERE { ?a <http://dbpedia.org/property/spouse> ?b. ?a <http://dbpedia.org/property/wikiLinks> <http://dbpedia.org/property/actor>. ?b <http://dbpedia.org/property/wikiLinks> <http://dbpedia.org/property/actor>. ?a <http://dbpedia.org/property/placeofbirth> ?c. ?b <http://dbpedia.org/property/placeofbirth> ?c. ?c <http://www.w3.org/2002/07/owl#sameAs> ?c2. ?c2 <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat. ?c2 < http://www.w3.org /2003/01/geo/wgs84 _pos#long> ?long.}

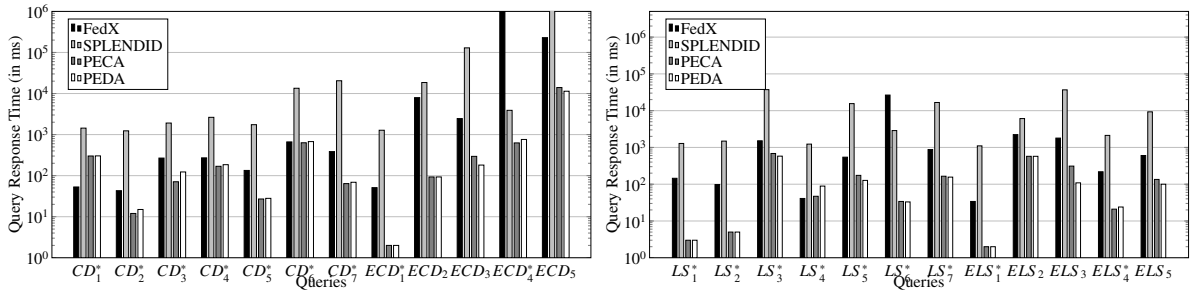
Table 15 BTC Queries

	Trinity.RDF	TriAD	PECA	PEDA
Q_1	281	97	53	51
Q_2	132	140	73	73
Q_3	110	31	27	28
Q_4	5	1	1	1
Q_5	4	0.2	0.18	0.18
Q_6	9	1.8	1.2	1.7
Q_7	630	711	123	122

Table 19 Comparison with Memory-based Distributed RDF Systems in LUBM 1000

Dataset		Number of Triples	RDF N3 File Size(KB)	Number of Entities
FedBench (Cross Domain)	DBPedia subset	42,855,253	6,267,080	8,027,158
	NY Times	337,563	103,788	21,667
	LinkedMDB	6,147,997	1,745,790	665,441
	Jamendo	1,049,647	147,280	290,292
	GeoNames	107,950,085	12,112,090	7,479,715
	SW Dog Food	103,595	16,858	10,459
FedBench (Life Science)	DBPedia subset	42,855,253	6,267,080	8,027,158
	KEGG	1,090,830	120,115	34,261
	Drugbank	766,920	146,906	19,694
	ChEBI	7,325,744	847,936	50,478

Table 20 Datasets



(a) Cross Domain (b) Life Science Domain

Fig. 17 Online Performance Comparison with Federated RDF systems over FedBench($*$ means that the query involves some selective triple patterns.)

ELS_5, our approach is faster than FedX and SPLENDID by an order of magnitude.