MPC: Minimum Property-Cut RDF Graph Partitioning

Peng Peng[1], M. Tamer Özsu[2], Lei Zou[3,4], Cen Yan[1], Chengjun Liu[1]

[1]*College of Computer Science and Electronic Engineering, Hunan University, China*
[2] *University of Waterloo, Canada*
[3] *Peking University, China*
[4] *Beijing Academy of Artificial Intelligence, Beijing, China*
{hnu16pp,yan_cen,lcj2021}@hnu.edu.cn, tamer.ozsu@uwaterloo.ca, zoulei@pku.edu.cn

2022 IEEE 38th International Conference on Data Engineering (ICDE) | 978-1-6654-0883-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDE53745.2022.00019

***Abstract*—Scaling-out RDF processing to deal with graph size usually requires partitioning the RDF graph. Typical partitioning approaches minimize edge-cuts or vertex-cuts. In this paper we argue that these approaches do not avoid or reduce joins between different partitions (i.e., inter-partition join), and propose an approach based on minimizing the number of distinct crossing properties, which we call *M*inimum *P*roperty-*C*ut (MPC). This approach enables more queries to be independently evaluated without inter-partition join. However, the minimum property-cut partitioning is a NP-hard problem and we propose a heuristic greedy algorithm to address that. Extensive experiments over a variety of synthetic and real RDF graphs show that the proposed technique can significantly avoid joins and results in good performance.**

***Index Terms*—RDF graph partitioning, distributed RDF systems, SPARQL query execution**

I. Introduction

Resource description framework (RDF) is a data model proposed by the W3C, where a triple of the form ⟨subject, property, object⟩ is the basic unit to describe the properties of the resources on the web and the relationships among these resources. 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. SPARQL is the query language for RDF, where the basic graph pattern (BGP) is the fundamental building block. A BGP query can also be represented as a query graph, and answering a BGP query is equivalent to finding subgraph matches (using homomorphism) of the query graph over the RDF graph.

As RDF dataset sizes increase, the typical performance issues of managing and querying them on a single machine arise, stimulating the interest in distributed solutions. In this paper, we focus on optimizing specialized distributed RDF systems, which are built specifically for SPARQL query evaluation by utilizing custom physical layouts that integrate multiple centralized RDF systems on different sites. This layout is widely-used because of its high efficiency [2], [25]. In this layout, the RDF graph G is divided into a set of subgraphs $\{F_1, ..., F_k\}$, called *partitions*, which are then distributed over a cluster of sites. If a query matches data across multiple partitions, inter-partition joins are involved to compute the result. Our focus in this paper is a partitioning technique that eliminates or minimizes the number of inter-partition joins, thereby improving overall query execution performance.

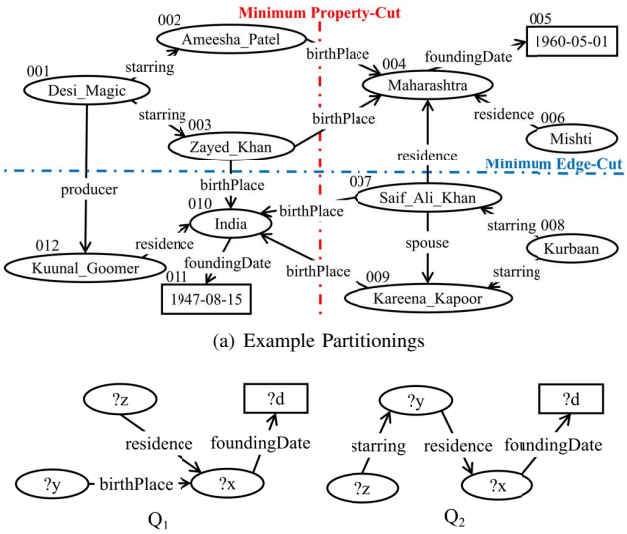
A. Background

Many popular partitioning approaches in existing distributed RDF systems are *vertex-disjoint* [16], [21], [22], [15], which assign each vertex to a single partition. In these approaches, some edges are "cut" across partitions and replicated in the two partitions of their endpoints to guarantee the completeness of the graphs of each partition. This is called 1-hop replication. An edge (triple) between two vertices in the same partition is an *internal edge* while one connecting two vertices in different partitions is called a *crossing edge*. Some works also discuss k-hop replication [16], [21], [15], allowing replication of k-hop neighbors of endpoints of crossing edges to improve localization. However, this increases the space cost and the data consistency maintenance overhead. Thus, this paper focuses on 1-hop replication.

Given a vertex-disjoint partitioning over an RDF graph G, the matches of a SPARQL query Q fall into two categories: *internal matches* and *crossing matches*. An internal match is fully contained in a single partition, while a crossing match spans multiple partitions. When a query Q is submitted, if we can determine that there are *only* internal matches, we can do *independent evaluation* at each partition. More specifically, we send Q to each site holding a partition F_i $(i = 1, ..., k)$ and evaluate Q over each F_i *independently*; we denote Q's match over partition F_i as $M(Q, F_i)$. Since there are no crossing matches, the full result is computed as the union of these matches, i.e., $M(Q, G) = \bigcup_{i=1}^{k} M(Q, F_i)$. Existing approaches [21], [22], [3], [39], [16], [13] can guarantee independent execution of *star queries* that involve a vertex and its neighbors because of 1-hop replication noted above.

If it is not possible to determine that a query Q consists only of internal matches, a common technique is to decompose Q into a set of star subqueries $\{q_1, ..., q_y\}$ and independently execute each q_j over every partition to find matches $M(q_j, G) = \bigcup_{i=1}^{k} M(q_j, F_i)$. Then, the final result is computed as $M(Q, G) = \bowtie M(q_j, G)$, $(j = 1, ..., y)$. This inter-partition join involves communication and extra computation that are costly.

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00019



(a) Example Partitionings

(b) Example Queries

Fig. 1. Comparison of Different Partitioning Approaches and Example Queries

B. Motivation & Our Approach

Existing works only consider the necessary condition for independent execution from the perspective of query structure: the query graph has to be star shaped. However, the class of independently executable queries (IEQs) can be extended beyond star queries by considering the properties (on edges). Given a vertex-disjoint partitioning, a property is called a *crossing property* if and only if there exists at least one crossing edge with that property as its label; otherwise, it is called an *internal property*. It can be proven that a query without any crossing property edges can be executed independently over each partition without inter-partition join.

The motivation of this paper is to find an RDF graph partitioning that avoids inter-partition joins in a wider set (than star queries) of SPARQL workloads. The traditional objective function of vertex-disjoint partitioning is to minimize *edge-cuts* while balancing the partition sizes (e.g., [16], [39], [13], [35], [32]); this is known as *minimum edge-cut partitioning*. The expectation is that fewer crossing edges can produce fewer crossing matches. This is a reasonable expectation, but it does not sufficiently eliminate or reduce the number of inter-partition joins. If we cannot judge that a query Q has no crossing matches before starting execution, inter-partition join cannot be avoided even if Q has no crossing matches during execution. Consider the vertex-disjoint partitioning in Fig. 1(a) where the blue dashed line indicates minimum edge-cut partitioning. In this case, there are three crossing properties:residence, birthPlace and producer. Executing query Q_2 in Fig. 1(b) would involve a crossing match, so Q_2 cannot be executed independently. Note that an edge (i.e., triple) with a crossing property does not mean that it has to be a crossing edge; there can be many edges with the same property and only some of them might be crossing edges. For example, although residence is a crossing property in the minimum edge-cut partitioning, edge $\overrightarrow{006\ 004}$ with property residence is not a crossing edge. However, if an edge has an internal property, it must be an internal edge. The important point here is the distinction between crossing edges and crossing properties.

In this paper we propose an alternative vertex-disjoint approach, called *M*inimum *P*roperty-*C*ut (MPC), where the objective function is to minimize the cuts to the number of *distinct* crossing properties rather than minimizing the edge-cuts. We prove and demonstrate that this approach avoids inter-partition joins in a wider set of SPARQL workloads. Consider the red dashed line partitioning in Fig. 1(a) that is the result of MPC partitioning. In this case there is only one crossing property, birthPlace. Query Q_2 does not involve this property, so it can be executed independently without inter-partition join although it is not a star query. Since MPC partitioning is also vertex-disjoint, it can still ensure that all star queries (e.g., Q_1) can be executed independently. Consequently, MPC can execute a larger class of queries without needing inter-partition joins, although it may have more crossing edge cuts than the minimum edge-cut partitioning.

Unfortunately, as proven in Theorem 1, the MPC problem is NP-complete. It is obvious that minimizing the number of crossing properties is equivalent to maximizing the number of internal properties. We, therefore, propose a greedy heuristic partitioning algorithm to find the partitioning with the maximum set of internal properties (Section IV). The greedy algorithm progressively coarsens internal properties, performs partitioning on this coarsened graph, and uncoarsens it to get a partitioning of the original graph. We prove that a star query is always independently evaluated in our approach (Theorem 5), so the number of decomposed subqueries in our MPC partitioning is no more than previous proposals. In addition to star queries, we demonstrate two additional query categories that can avoid inter-partition joins with MPC partitioning (Section V): *internal queries* that do not contain any crossing property edges and *extended independently executable queries* that may contain crossing properties but can be guaranteed to not involve any crossing matches. Thus, MPC partitioning can avoid more inter-partition joins than existing approaches.

In summary, we make the following contributions:

- We propose minimum property-cut (MPC) as a novel graph partitioning scheme for RDF graphs in the context of distributed SPARQL query evaluation.
- Due to the hardness of the MPC problem, we propose a greedy heuristic. The property selection order influences the partitioning quality and we propose a cost function to decide the order. We propose to use disjoint-set forest data structure to efficiently compute the cost function.
- We define the types of queries that can be independently executed and the set of such queries is larger than what is supported by existing approaches. We discuss how a query is decomposed into one of these types for independent execution, and prove the number of subqueries is guaranteed to be no more than existing approaches.
- Experiments show that MPC can significantly improve distributed SPARQL query processing by avoiding a higher number of inter-partition joins.

II. Related Work

Distributed SPARQL query evaluation has been extensively studied in the literature [18], [25], [2], [19]. One of the key issues is how to partition an RDF graph into subgraphs and distribute them among the sites. Generally, we classify the RDF graph partitioning approaches into three categories:

Vertex-disjoint partitioning approaches. Most popular partitioning techniques in existing distributed RDF systems are vertex-disjoint and assign each vertex to a single partition. The edges connecting two vertices in different partitions are replicated in the two partitions of their endpoints, ensuring that star queries can be evaluated independently. A non-star query is decomposed it into several star-shaped subqueries. In particular, SHAPE [21], [22] and AdPart [3] use hash functions to assign each vertex to a partition; while EAGRE [39], H-RDF-3X [16] and TriAD [13] use the minimum edge-cut partitioning method to partition the RDF graphs.

The proposed MPC partitioning is also vertex-disjoint, but differs from the others in its consideration of *crossing properties*. MPC enables more IEQs and reduces the number of inter-partition joins.

Edge-disjoint partitioning approaches. An alternative is edge-disjoint, which partitions an RDF graph according to edge properties. The triples with the same property go to the same partition, like vertical partitioning [1]. Edge-disjoint partitioning has been widely used in many cloud-based distributed RDF systems [17], [11], [31], [24], [34], where edges of different properties are put into different storage units in the cloud. For example, in HadoopRDF [17] and CliqueSquare [11], edges of different properties are stored in different HDFS files; while S2RDF [31], WORQ [24] and Sparklify [34] store edges of different properties in different Spark SQL tables.

In cloud-based systems, the main optimization objective is to prune irrelevant partitions and avoid too many scans in the cloud. Edge-disjoint partitioning can achieve this goal by only scanning storage units of relevant properties. However, in vertex-disjoint partitioning, although partitions can also be stored in different storage units in cloud, all these storage units need to be scanned.

Other approaches. Some other partitioning approaches consider extra information besides RDF graph itself. For example, DiploCloud [37] requires the administrator to define templates as partition units; WARP [15], Partout [9], Peng et al. [26], [27] and WASP [5] utilize query workloads to define partitioning units. In this paper, we focus on a data-driven approach without considering extra information, such as query logs. Considering the frequency of properties in query logs, a weighted MPC partitioning is also desirable, but that is beyond the scope of the paper.

The above methods focus on offline partitioning strategy to improve the system's performance. There are other approaches that optimize inter-partition join processing during query evaluation, which we call *run-time optimizations*, that are orthogonal to the RDF graph partition strategies. TriAD [13] performs distributed merge-joins over different indexes. Multiple join operators are executed in parallel. AdPart [3] employs distributed semijoin to optimize query evaluation and Wu et al. [36] propose a top-down join enumeration algorithm that enumerates query plans with multiway joins. WORQ [24] minimizes the intermediate results by precomputing join reductions through Bloom-joins and cache the join reductions that correspond to the frequent join patterns. Our earlier work, gStoreD [28], [29], proposes a partial-evaluation-and-assembly run-time query framework.

Our method involves the weakly connected components (WCCs) induced by some edge labels, which is also also used in GRASP [6]. However, GRASP is designed to handle regular path queries over property graphs, so it only considers the connectivity of WCCs induced by the edge labels. In contrast, our goal is to optimize distributed query evaluation, and we additionally consider balancing the sizes of the WCCs.

III. Preliminaries

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

Definition 3.1: **(RDF Graph)** An RDF graph is denoted as $G = \{V, E, L, f\}$, where V is a set of vertices that correspond to all subjects and objects in the RDF data; $E \subseteq V \times V$ is a multiset of directed edges that correspond to all triples in the RDF data; L is a set of edge labels; and $f : E \to L$ is a label mapping, where for each edge $e \in E$, its edge label $f(e)$ is its corresponding property.

A directed graph G is called *weakly connected* if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. A *weakly connected component* G' of an RDF graph G is a maximal weakly connected subgraph of G such that G' is not the subgraph of another weakly connected component. The set of weakly connected components of G is denoted as $WCC(G)$.

Definition 3.2: **(Property-Induced Subgraph)** Given a set of properties $L' \subseteq L$ and an RDF graph G, the *property-induced subgraph* of G by L', denoted as $G[L']$, is the subgraph formed by only edges with properties in L'.

As noted earlier, vertex-disjoint partitioning assigns each vertex to one site (i.e., the partitions are vertex-disjoint), and to guarantee data integrity and consistency, replicas of each crossing edge are stored in the partitions of its two endpoints. Formally, we define the *partitioning* of an RDF graph as follows, where u and $\overrightarrow{uu'}$ denote the vertex and edge and k is the number of graph partitions. Usually, k depends on the number of machines in the distributed system.

Definition 3.3: **(Partitioning)** Given an RDF graph G, a partitioning $\mathcal{F}$ is a set of partitions $\{F_1, F_2, ..., F_k\}$, where each $F_i = (V_i \cup V_i^e, E_i \cup E_i^c, L_i, f_i)$ $(1 \leq i \leq k)$ such that

1) $\{V_1, ..., V_k\}$ is a disjoint partitioning of V, i.e., $V_i \cap V_j = \emptyset, 1 \leq i, j \leq k, i \neq j$ and $\bigcup_{1 \leq i \leq k} V_i = V$;
2) $E_i \subseteq V_i \times V_i$, $1 \leq i \leq k$; edges in E_i are called *internal edges* of F_i;
3) E_i^c is a set of *crossing edges* between F_i and other partitions, and $E^c = \bigcup_{1 \leq i \leq n} E_i^c$ is the set of all crossing edges in $\mathcal{F}$;

4) Replicas of crossing edges are stored at two sites of its two endpoints' partitions, so some vertices of other partitions are stored at F_i and the set of these vertices are denoted as V_i^e. A vertex $u' \in V_i^e$ if and only if vertex u' resides in other partitions F_j and u' is an endpoint of a crossing edge between partitions F_i and F_j ($F_i \neq F_j$).
5) L_i is a set of edge labels in F_i.
6) $f_i : E_i \rightarrow L_i$ is a label mapping, where for each edge $e \in E_i$, its edge label $f_i(e)$ is $f(e)$.

Based on the above partitioning definition, we can formally define internal and crossing properties.

Definition 3.4: **(Internal/Crossing Property)** Given a partitioning $\mathcal{F}$ over an RDF graph G, a property that does not exist in any crossing edge (i.e., in E^c) is called an *internal property* and $L_{in} = \{p | p \in L \wedge \forall e \in E^c, f(e) \neq p\}$ is the set of all internal properties. In contrast, a property that is not in L_{in} is called a *crossing property* and $L_{cross} = L - L_{in}$ is the set of all crossing properties. A crossing property is the label of at least one crossing edge in E^c.

Fig. 2 shows a partitioning over an RDF graph consisting of two partitions F_1 & F_2. For F_2, 004, 005, 006, 007, 008 and 009 are internal vertices, and the edges between any two of them are internal edges. The edges between 002, 003 and 010 and internal vertices of F_2 are crossing edges. The internal properties are starring, residence, chronology, spouse and foundingDate, while the crossing property is birthPlace.

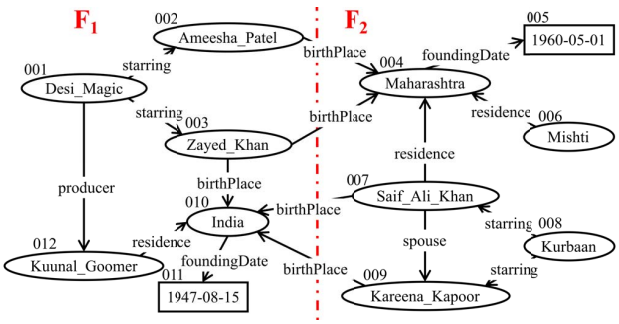


Fig. 2. Example RDF Graph and Partitioning

If a property is a crossing property, this indicates that there is at least one crossing edge with that property, but it does not mean that all edges with the property are crossing edges. For example, although edge $\overrightarrow{003\ 010}$ in Fig. 2 is an internal edge of F_1, its property is the crossing property birthPlace. However, if a property is an internal property, all edges having the property must be internal edges.

A SPARQL query can similarly be represented as a query graph Q. In this study, we focus on BGP queries as they are foundational to SPARQL, and we consider techniques for handling them.

Definition 3.5: **(SPARQL BGP Query)** A *SPARQL BGP query* is denoted as $Q = \{V^Q, E^Q, L^Q, f^Q\}$, where $V^Q \subseteq V \cup V_{Var}$ is a set of vertices, where V denotes all vertices in the 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; $L^Q \subseteq L \cup L_{Var}$ is a set of edge labels, where L denotes all properties in the RDF graph G, and L_{Var} is a set of variables for properties; and $f^Q : E^Q \rightarrow L^Q$ is a mapping, where each edge e in E^Q either has an edge label $f^Q(e)$ in L (i.e., property) or the edge label is a variable.

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

Definition 3.6: **(SPARQL BGP Match)** Consider an RDF graph G and a query graph Q with 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* μ 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, $\mu(v_i)$ and v_i have the same uniform resource identifier (URI) or literal value ($1 \leq i \leq n$); 2) if v_i is a variable, there is no constraint over $\mu(v_i)$ except that $\mu(v_i) \in \{u_1, ..., u_m\}$; 3) if there exists an edge $\overrightarrow{v_i v_j}$ in Q, there also exists an edge $\overrightarrow{\mu(v_i)\mu(v_j)}$ in G; 4) there must exist an *injective function* from edge labels in $f^Q(\overrightarrow{v_i v_j})$ to edge labels in $f(\overrightarrow{\mu(v_i)\mu(v_j)})$. Note that a variable edge label in $f^Q(\overrightarrow{v_i v_j})$ can match any edge label in $f(\overrightarrow{\mu(v_i)\mu(v_j)})$.

The motivation of our work is to find an RDF graph partitioning that enables more SPARQL queries to be executed independently. We formally define *independent evaluation* as follows:

Definition 3.7: **Independent Execution**. A SPARQL query Q is said to be *independently executable* over a partitioning $\mathcal{F} = \{F_1, F_2, ..., F_k\}$ of RDF graph G if and only if its result $M(Q, G) = \bigcup_{i=1}^{k} M(Q, F_i)$, where $M(Q, F_i)$ is the result of executing Q on partition F_i ($i = 1, ..., k$).

IV. Minimum Property-Cut Partitioning

We now define *minimum property-cut partitioning*. Our focus is on the new partitioning definition, its complexity (it is NP-complete), and the design of an approximate algorithm. We defer proof that this approach enables independent execution of a broader class of queries to Section V.

A. Problem Definition

Intuitively, to minimize inter-partition joins, it is necessary to minimize the number of distinct crossing properties ($|L_{cross}|$); we prove this formally in Section V-A in the context of query types (Theorem 3). We use this notion to formally define *minimum property-cut partitioning*.

Definition 4.1: **(Minimum Property-Cut Partitioning)** Given an RDF graph G and a positive integer k, the *minimum property-cut (MPC)* partitioning of G is a partitioning $\mathcal{F} = \{F_1, F_2, ..., F_k\}$ such that (1) the number of crossing properties $|L_{cross}|$ is minimized (i.e. the number of internal properties $|L_{in}|$ is maximized); and (2) the size of F_i (i.e. $|V_i|$) is not larger than $(1 + \epsilon) \times |V|/k$ for each F_i, where ϵ is a user-defined maximum imbalance ratio of a partitioning (i.e., how much difference can be in the relative sizes of partitions).

Note that, ϵ is a parameter widely used in previous works (e.g., [35], [32]) to specify how much of an imbalance the

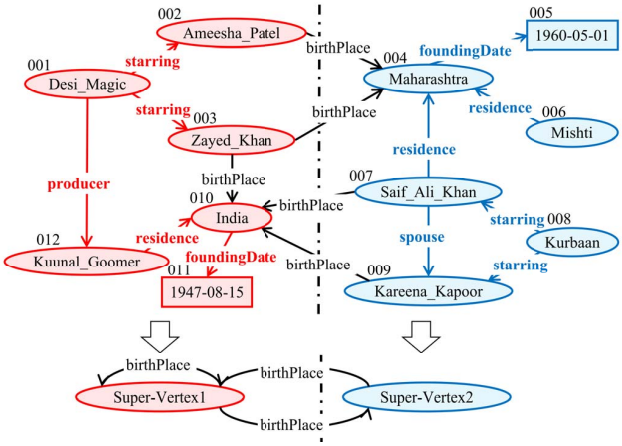


Fig. 3. Example of Coarsening

user can tolerate. This is needed because we usually cannot ensure that the partitions are completely balanced.

Theorem 1: The MPC partitioning problem is NP-complete.

Proof: We reduce the NP-complete minimum edge-cut problem to the MPC problem. Given an instance of minimum edge-cut graph partitioning over G, we assign each edge in G with a distinct property, which is denoted as $\mathcal{G}$. In this case, finding MPC graph over $\mathcal{G}$ is equivalent to finding minimum edge-cut over G. The latter is a classical NP-complete problem, thus, Theorem 1 holds. ■

Given this complexity proof, we propose, in Section IV-B, a greedy heuristic that achieves a good approximation.

B. Partitioning Technique

Following Definition 4.1, we note that $L_{cross} = L - L_{in}$, and, therefore, minimizing $|L_{cross}|$ is equivalent to maximizing the set of internal properties $|L_{in}|$. Our solution focuses on maximizing $|L_{in}|$.

First, we use a greedy algorithm to select internal properties L_{in} (Section IV-C). Let $G[L_{in}]$ be the subgraph induced by internal properties L_{in}. Then, each weakly connected component (WCC) in $G[L_{in}]$ is represented as a supervertex – we call this *coarsening*. In this way, we obtain a coarsened graph (denoted as G_c) with much smaller number of vertices, each of which represent a WCC. For example, given a graph G in Fig. 3 and internal properties $L_{in} = \{\mathsf{starring}, \mathsf{residence}, \mathsf{producer}, \mathsf{spouse}, \mathsf{foundingDate}\}$, the red and blue vertices and edges show two WCCs in $G[L_{in}]$. Each one is represented as a supervertex and G is coarsened into G_c (Fig. 3). Note that there are no internal property edges in G_c, since they have been coarsened into supervertices.

We can now use any vertex-disjoint partitioning algorithm (e.g., METIS [20]) over the coarsened graph G_c. The complexity of these algorithms is not an impediment since G_c is much smaller than G. An example is shown in Fig. 3. There are only two vertices in G_c, which are partitioned into two partitions. Finally, the partitioning over G_c is uncoarsened to obtain a partitioning of the original graph G – see the dashed line in Fig. 3.

It is obvious that no internal property edge in G can be a crossing edge in the final partitioning, since the internal property edges have been coarsened into supervertices in G_c and each supervertex is in one partition.

C. Internal Property Selection

The key issue in this framework is how to select L_{in}, i.e., how to maximize the number of internal properties under partition size constraint (see Definition 4.1). Since the problem is NP-complete, we use a greedy heuristic algorithm to select L_{in} (Algorithm 1). Since our formulation of the internal property selection algorithm is based on WCC, we first establish the relation between them.

Theorem 2: Let L_{in} be the set of internal properties and $G[L_{in}]$ be the subgraph induced by L_{in}. Any two vertices in a weakly connected component (WCC) of $G[L_{in}]$ must be in the same partition.

Proof: Assume that two vertices u and v in a WCC of $G[L_{in}]$ are not in the same partition. Since they are in the same WCC, there exists at least one weakly connected path π between them. Since they are not in the same partition, at least one edge in path π crosses two partitions, i.e., it is a crossing edge. This conflicts with the fact that all properties along the path π are internal properties. ■

According to the definition of MPC partitioning (Definition 4.1), the number of vertices in each partition should be no larger than $(1+\epsilon)\times|V|/k$. Therefore, given a set of properties L', the cost of selecting them as internal properties is defined as the size of the largest WCC in the property induced subgraph $G[L']$.

Definition 4.2: **(Selecting Internal Property Cost)** Given a set of properties $L' \subseteq L$, the cost of selecting L' as internal properties is:

$$Cost(L') = \max_{c \in WCC(G[L'])} |c|$$

where c is a weakly connected component in $WCC(G[L'])$ and $|c|$ denotes the number of vertices in c.

Using this cost function, we select as many properties to be internal properties as possible as long as the cost of the selected internal properties is no larger than $(1+\epsilon)\times|V|/k$.

The internal property selection algorithm is given in Algorithm 1. Note that the optimizations we discuss in the following section (Section IV-D) are built into this algorithm. Therefore, at this stage, we provide the highlights of its operation; more details will be filled in Section IV-D. Initially, L_{in} starts empty (Line 1) and we compute the WCCs of the property-induced subgraphs for each property (Lines 2-4). In each iteration, a property p that minimizes $Cost(L_{in} \cup \{p\})$ is selected as the next p_{opt} and inserted into L_{in} (Lines 5-14); p_{opt} is then removed from L (Lines 15-16). These steps are repeated until L is empty (Line 5) or no more property can be selected due to the cost constraint (Lines 13-14). The selected internal properties L_{in} are returned (Line 17).

D. Optimization Using Disjoint-Set Forest

The bottleneck of Algorithm 1 lies in Lines 3 and 8, namely computing the WCCs. The first step is to compute the WCCs

Algorithm 1: Internal Property Selection Algorithm

Input: An RDF graph $G = (V, E, L, f)$
Output: A set of internal properties $L_{in} \subseteq L$

1 $L_{in} \leftarrow \emptyset$;
2 **for** *each property* p *in* L **do**
3 Compute $WCC(G[\{p\}])$, i.e., weakly connected components in $G[\{p\}]$;
4 Compute the cost of $Cost(\{p\})$ according to Definition 4.2.
5 **while** $L \neq \emptyset$ **do**
6 $mincost \leftarrow \infty$; $p_{opt} \leftarrow \phi$
7 **for** *each property* p *in* L **do**
8 Compute $WCC(G[L_{in} \cup \{p\}])$
9 **if** $Cost(L_{in} \cup \{p\}) < (1+\epsilon) \times |V|/k$ **then**
10 **if** $Cost(L_{in} \cup \{p\}) < mincost$ **then**
11 $mincost \leftarrow Cost(L_{in} \cup \{p\})$;
12 $p_{opt} \leftarrow p$
13 **if** $p_{opt} == \phi$ **then**
14 BREAK
15 $L_{in} \leftarrow L_{in} \cup \{p_{opt}\}$;
16 $L \leftarrow L - \{p_{opt}\}$;
17 Return L_{in};

in $G[\{p\}]$ (Line 3). The second step is to iteratively merge WCCs in $G[L_{in}]$ and $G[\{p\}]$ (Line 8).

We propose to use the disjoint-set forest data structure [10], since it can dynamically track the WCCs of a graph as vertices and edges are added. For each property p, we initialize a disjoint-set forest $DS(\{p\})$, in which each node u corresponds to a single tree and u is associated three values $u(\{p\}).parent$, $u(\{p\}).rank$ and $u(\{p\}).size$. $u(\{p\}).parent$ is the parent of u in $DS(\{p\})$ and initialized as u itself; $u(\{p\}).rank$ is an upper bound on the height of the rooted tree and initialized as 0; and $u(\{p\}).size$ is the number of the vertices in the rooted tree and initialized as 1.

Then, for each edge $\overrightarrow{uu'}$ in RDF graph G, if its edge property is p, the rooted trees of u and u' in the corresponding disjoint-set forest $DS(\{p\})$ are combined, which means that the WCCs containing u and u' are merged. This is a UNION operation in disjoint-set forest [10]. The root of the tree with smaller rank is adjusted to point to the root of the tree with larger rank, each vertex on the paths from u and u' to their roots is made to point directly to the root of its disjoint-set forest, and the sizes of the rooted trees of u and u' are summed. After processing all the edges with property p, two vertices are in the same connected component of $G[\{p\}]$ if and only if they are in the same tree of disjoint-set forest $DS(\{p\})$. In other words, each tree in $DS(\{p\})$ corresponds to one WCC in $G[\{p\}]$. Therefore, it is straightforward to compute $Cost(\{p\})$ using Definition 4.2.

To compute $WCC(G[L_{in} \cup \{p\}])$ (Line 8 in Algorithm 1), we compute the disjoint-set forest $DS(L_{in} \cup \{p\})$ by merging $DS(L_{in})$ and $DS(\{p\})$. Initially, we set $DS(L_{in} \cup \{p\}) = DS(L_{in})$. For each vertex u in $DS(\{p\})$, we find the root of its tree in $DS(\{p\})$, denoted as $uRoot$. This is done by recursively visiting the current node's ancestors up to the root, which is the FIND operation in disjoint-set forest [10]. Since u and $uRoot$ are in the same WCC of $G[\{p\}]$, they should also be in the same WCC of $G[L_{in} \cup \{p\}]$. Thus, we first FIND two rooted trees of u and $uRoot$ in the current $DS(L_{in} \cup \{p\})$. If u and $uRoot$ are not in the same rooted tree in $DS(L_{in} \cup \{p\})$, we merge them by UNION operation.

E. Analysis

The complexity of Algorithm 1 depends on computing WCC (Lines 3 and 8) using the disjoint-set forests. The complexity of building the disjoint-set forests is $O(\alpha(|V|) \times |E|)$, where $\alpha(|V|)$ is the inverse Ackermann function and smaller than 5 for all practical purposes [8].

Merging two disjoint-set forests needs $O(\alpha(|V|) \times |E|)$ time. Considering the dual loop in Lines 5-7 of Algorithm 1, the time complexity is $O(|L|^2 \times \alpha(|V|) \times |V|)$. Therefore, the total time complexity of Algorithm 1 is $O(|L|^2 \times \alpha(|V|) \times |V|)$. Usually, the number of properties is much less than the number of vertices and edges in RDF graphs (i.e. $|L| \ll |V|$) and $\alpha(|V|)$ is a trivial value, thus, the complexity mainly depends on the number of vertices, $|V|$.

The complexity of Algorithm 1 can be further improved by some additional heuristics. First, given a property p, if the largest WCC in its property-induced subgraph $G[\{p\}]$ is larger than $(1+\epsilon) \times |V|/k$, this property can be pruned. There are some popular properties in RDF that always generate large-size WCCs. For example, rdf:type is widely used in most RDF datasets to represent the class information, and almost all entities have class owl:Thing in many real RDF graphs. Therefore, we can prune these properties and not consider them. This reduces the size of the property set on which the complexity of the algorithm is dependent.

Second, for some real RDF datasets (such as DBpedia and LGD in our experiments), most properties can be selected as internal properties. Thus, we can initially put all properties into L_{in}. In each step, we greedily remove one property p (from L_{in}) that leads to maximum cost reduction (i.e., maximizing $Cost(L_{in}) - Cost(L_{in} - \{p\})$) until the $Cost(L_{in}) \leq (1 + \epsilon) \times |V|/k$.

V. Distributed Query Execution

In this section, we address two problems. First, we identify the set of SPARQL queries that can be independently executed. Then, we study how to answer a query if it is not independently executable.

A. Independently Executable Queries

Independently executable queries (IEQs) are those that are "local" to a partition and can be executed without a join with another partition. Existing work that uses minimum edge-cut partitioning guarantee that star queries can be executed independently; this holds under Definition 3.7 as well. In this paper, we extend the class of independently executable queries. We identify two categories of such queries—internal and extended[1] .

[1]SPARQL queries may have variables in edge properties, such as edge $?p$ in Q_5 in Fig. 5. We can regard these variable-edges as crossing-property edges in the following discussion, without affecting the correctness of our method.

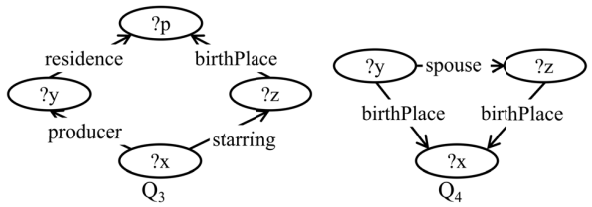


Fig. 4. Example Extended IEQs

Definition 5.1: **(Internal Independently Executable Query)** A SPARQL query Q that does not contain any crossing property edges is an *internal independently executable query (IEQ)*.

Theorem 3: An internal IEQ Q can be executed without inter-partition join.

Proof: (Proof by contradiction) If an internal IEQ Q has a crossing match m, requiring an inter-partition join, there is at least one edge e_q in Q that matches a crossing edge e_c and they have the same property, so e_q's property is a crossing property. This contradicts the definition of an internal IEQ. ■

For the partitioning in Fig. 2 where $L_{cross} = \{\mathsf{birthPlace}\}$, Q_2 in Fig. 1(b) is an internal IEQ, since it does not have an edge labelled birthPlace.

The second category of independently executable queries are what we call extended independently executable queries. There are two types: A Type-I query (Definition 5.2) contains crossing properties but is guaranteed to not involve any crossing edges. Recall that a crossing property is the label of at least one crossing edge, but this does not imply that all edges with this property are crossing edges. A Type-II query (Definition 5.3) may involve the replicas of crossing edges at each partition (as stated earlier replicas of crossing edges are stored at the two sites of its two endpoints' partitions).

Definition 5.2: **(Type-I Extended Independently Executable Query)** If a SPARQL query Q is weakly connected when all crossing property edges in Q are removed, then it is a *Type-I extended independently executable query*.

Definition 5.3: **(Type-II Extended Independently Executable Query)** Consider a SPARQL query Q that is decomposed into x WCCs $\{q_1, q_2, ..., q_x\}$ after all crossing property edges are removed. Q is a *Type-II extended independently executable query* if and only if all q_j are one-vertex WCCs except for a single WCC q_i, where $1 \leq j \neq i \leq x$, and there are no crossing property edges between any two one-vertex WCCs q_{j_1} and q_{j_2} $(1 \leq j_1 \neq j_2 \neq i \leq x)$. Formally, (1) there is a single WCC q_i, where $|q_i| \geq 1$ and $|q_i|$ denotes the number of vertices in q_i, and all other WCCs $|q_j| = 1$ $(1 \leq j \neq i \leq x)$; and (2) there are no crossing property edges between any two q_{j_1} and q_{j_2} $(1 \leq j_1 \neq j_2 \neq i \leq x)$.

For example, Q_3 and Q_4 in Fig. 4 are extended IEQs: Q_3 is Type-I, while Q_4 is Type-II. Q_3 is Type-I, because it is still weakly connected after removing crossing property edge $\overrightarrow{?z\ ?p}$. Since the other three edges in Q_3 are internal property edges, all variable bindings are internal vertices of the same partition. In other words, the vertices matching $?z$ and $?p$ are internal vertices in the same partition; thus, the edge matching $\overrightarrow{?z\ ?p}$ must be an internal edge in the same partition. Therefore, evaluating Q_3 does not need inter-partition join.

In query Q_4, removing the crossing property edges $\overrightarrow{?y\ ?x}$ and $\overrightarrow{?z\ ?x}$ leads to a size-1 subquery (i.e., containing only one vertex) $?x$ and the remaining part Q'_4. Obviously, Q'_4 is an internal IEQ. For any match of Q_4, all its vertices are internal to one partition. Due to crossing edge replication in vertex-disjoint partitioning, the matching vertex of the size-1 subquery $?x$ must be an extended vertex of the corresponding partition, since it is one-hop from the internal match. Thus, Q_4 is independently executable.

Theorem 4: Both Type-I and Type-II extended IEQs can be independently executed.

Proof: The set of crossing property edges in a query Q is denoted as E^{Qc}. Given a Type-I extended IEQ Q, after removing all crossing property edges, the remaining part (denoted as Q') is still weakly connected. Q' is an internal IEQ, because there is no crossing property in Q'. Thus, in any match of Q', all vertices must be internal vertices in the same partition, which indicates that the edges that map to edges in E^{Qc} should be internal edges in the same partition. Thus, Q can be independently executed.

Given a Type-II extended IEQ Q that is decomposed into x WCCs $\{q_1, q_2, ..., q_x\}$ after removing all crossing property edges in Q, where $|q_i| \geq 1$ and $|q_j| = 1$ $(1 \leq j \neq i \leq x)$, it is easy to show that q_i is an internal IEQ because all crossing property edges are removed. Thus, q_i only matches to internal vertices. Other size-1 subqueries q_j $(j \neq i)$ are one-hop from q_i. Due to crossing edge replication, all 1-hop neighbors of internal vertices must be in the same partition. Therefore, Q can also be independently executed. ■

As noted earlier, under MPC partitioning, star queries are still independently executable. The following theorem establishes this; thus MPC significantly extends the class of IEQs.

Theorem 5: A star query is either an internal IEQ or a Type-II extended IEQ.

Proof: Given a star query Q, if there is no crossing property edge in Q, it is obvious that it is an internal IEQ. If Q contains some crossing property edges, we prove that Q is a Type-II extended IEQ. Since Q is a star query, there should exist one vertex v_c as the central vertex and all other vertices are incident to v_c. After removing crossing property edges, we assume that Q is decomposed into $\{q_1, q_2, ..., q_y\}$ and q_1 is the subquery containing v_c. Because any other vertex is only adjacent to v_c, q_j $(1 < j \leq y)$ should only contain one vertex, which meets condition (1) of Definition 5.3. Non-central vertices in Q cannot be connected. Therefore, there is no crossing property edges between any pairs of subqueries q_i and q_j $(1 < i \neq j \leq y)$, and Q meets condition (2) of Definition 5.3. Thus, Q is Type-II extended IEQ. ■

B. Query Decomposition & Execution

Real SPARQL workloads are likely to contain queries that are not independently executable, e.g., Q_5 in Fig. 5. For these queries, to exploit the MPC partitioning and minimize the inter-partition joins, we first decompose them into a set of

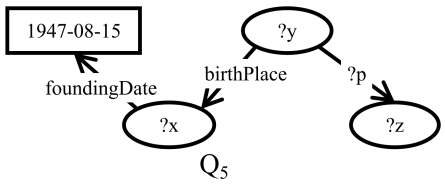


Fig. 5. Example Non-IEQ

independently executable subqueries (Section V-B1) and then join their results to obtain the final result (Section V-B2).

1) Query Decomposition: Algorithm 2 outlines query decomposition in the context of MPC partitioning. Given a non-IEQ Q and a MPC partitioning $\mathcal{F}$ with the set of crossing properties L_{cross}, we remove all crossing property edges and edges with variables in the property position from Q resulting in a set of subqueries $\{q'_1, q'_2, ..., q'_x\}$ (Line 2). Each q'_i only involves internal properties and is an internal IEQ.

Algorithm 2: Query Decomposition Algorithm

Input: A SPARQL Query Q, and a MPC partitioning $\mathcal{F}$ with the set of crossing properties L_{cross}
Output: A query decomposition result $\mathcal{Q} = \{q_1, q_2, ..., q_y\}$

1 $\mathcal{Q} \leftarrow \emptyset$;
2 Remove the crossing property edges and edges with variables in the property position in Q and get a set of WCCs, $\{q'_1, q'_2, ..., q'_x\}$;
3 **for** *each crossing property edge* $\overrightarrow{v_i v_j}$ *and edges with variables in the property position in* Q **do**
4 | /* $q(v_i)$ and $q(v_j)$ denote the WCCs containing vertices v_i and v_j, respectively */
5 | **if** $q(v_i) = q(v_j)$ **then**
6 | | Add $\overrightarrow{v_i v_j}$ into $q(v_i)$;
7 | **else**
8 | | **if** $|q(v_i)| \leq |q(v_j)|$ **then**
9 | | | /* $|q(v_i)|$ denotes the no. of vertices in $q(v_i)$ */
10 | | | Add $\overrightarrow{v_i v_j}$ into $q(v_j)$;
11 | | **else**
12 | | | Add $\overrightarrow{v_i v_j}$ into $q(v_i)$;
13 **for** *each* q'_i *in* $\{q'_1, q'_2, ..., q'_x\}$ **do**
14 | **if** $|q'_i| > 1$ **then**
15 | | Add q'_i into $\mathcal{Q}$;
16 Return $\mathcal{Q}$;

We then consider crossing property edges and edges with variables in the property position one-by-one. We add them to these subqueries. Consider a crossing property edge or edge with a variable in the property position $\overrightarrow{v_i v_j}$ and let $q(v_i)$ and $q(v_j)$ be the two subqueries containing vertices v_i and v_j, respectively. If $q(v_i) = q(v_j)$ (i.e., v_i and v_j are in the same subquery), we add the crossing property edge $\overrightarrow{v_i v_j}$ into $q(v_i)$ (Lines 5-6 in Algorithm 2), making $q(v_i)$ a type-I extended IEQ; otherwise, we add the crossing property edge $\overrightarrow{v_i v_j}$ to the subquery with more vertices (Lines 7-12 in Algorithm 2), and that subquery becomes a type-II extended IEQ. Once all the crossing property edges are handled, all the subqueries with more than one vertex are returned as the decomposition result, denoted as $\mathcal{Q} = \{q_1, q_2..., q_y\}$ (Lines 13-15 in Algorithm 2). Since Algorithm 2 needs to scan the query graph at most twice, its time complexity is $O(|E^Q|)$.

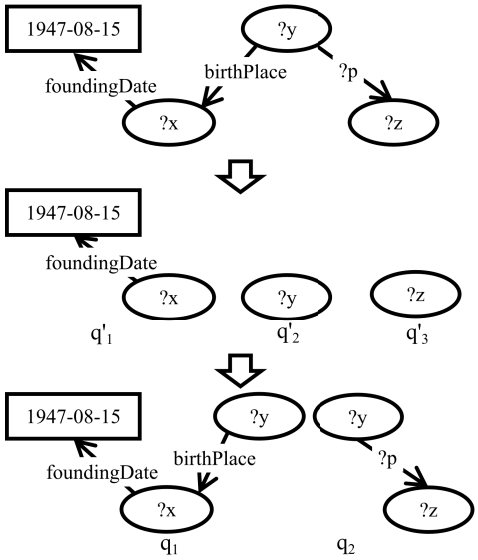


Fig. 6. Example Query Decomposition

Fig. 6 illustrates the decomposition of Q_5 using MPC partitioning with $L_{crossing}$ ={birthPlace}. We first remove the crossing property birthPlace edges $\overrightarrow{?y\ ?x}$ and edge of variable property $\overrightarrow{?y\ ?z}$. Then, the query is decomposed into three subqueries: q'_1, q'_2 and q'_3. Crossing property edge $\overrightarrow{?y\ ?x}$ is adjacent to q'_1 and q'_2 and is added to q'_1 to form q_1, since q'_1 is larger than q'_2. Edge $\overrightarrow{?y\ ?z}$ can be added to either q'_2 or q'_3 since they are of the same size; we assume that it is added to q'_2 to form q_2. In addition, q'_3 only contains one query vertex and is removed. There are too many matches mapping to the query of one vertex q'_3 but these matches are unnecessary. This is because matches of q'_3 that can contribute to a final match of Q_5 can also map to the vertex $?z$ in q_2. Consequently, $\{q_1, q_2\}$ is the query decomposition result.

2) Query Execution: When a non-IEQ Q is received, it is decomposed into a set of subqueries $\{q_1, q_2, ..., q_y\}$ as discussed in the previous section. This set of subqueries are sent to each partition for execution. We note that this approach does not localize each subquery to one partition; it executes each subquery over every partition. The localization of SPARQL queries is challenging and is left as future work. However, the approach takes advantage of intra-query parallelism; each q_i can be independently executed over each partition F_j, allowing both concurrent execution at each partition site and concurrent execution across the worker machines. The result of these executions are a set of matches of each subquery over each partition, i.e., $M(q_i, F_j)$.

Then, the subqueries' matches need to be joined to obtain the final result, i.e., $M(q_i, G) = \bowtie M(q_i, F_j)$. How this inter-partition join is executed depends on the particular system architecture: it may involve distributed joins or individual matches can be collected at the querying site and the join performed there. The possible run-time optimization strategies are well-studied and we do not discuss them further; these are orthogonal to our offline partition-oriented optimizations.

TABLE I
STATISTICS OF DATASETS

Dataset	#Entities	#Triples	#Properties
LUBM 100M	17,473,142	106,909,064	18
LUBM 1B	173,891,493	1,069,331,221	18
LUBM 10B	1,737,718,408	10,682,013,023	18
WatDiv 100M	5,212,745	108,997,714	86
WatDiv 1B	52,120,745	1,099,208,068	86
WatDiv 10B	521,200,745	10,987,996,562	86
YAGO2	21,073,153	284,417,966	98
Bio2RDF	804,671,979	4,426,591,829	1,581
DBpedia	139,493,254	1,111,481,066	124,034
LGD	311,153,753	1,292,933,812	33,348

VI. EXPERIMENTS

A. Setting

We evaluate MPC over both synthetic and real RDF datasets (Table I). All experiments are conducted on a cluster of 8 machines running Linux version 3.10.0, each of which has two CPUs with 6 cores of 2.27 GHz, 128 GB memory and 10 TB disk. At each site, a partition is stored in the gStore RDF engine [40], [38]. We select one of these machines as the coordinator to receive queries and use MPICH-3.0.4 for communication. Implementations are in C++. Our code has been released in GitHub [2].

We compare MPC with three RDF partitioning schemes that are widely used in distributed RDF systems: hashing triples based on subjects (denoted Subject_Hash) [21], [22], [3], minimum edge-cut partitioning using METIS algorithm [20] (denoted METIS) and hashing triples based on properties [17], [31], [24] (denoted VP). All methods except VP are vertex-disjoint partitioning methods, while VP is edge-disjoint and assign each edge to a single partition (allowing vertex copies between different partitions).

1) Synthetic Datasets: We use two synthetic datasets, LUBM and WatDiv. The default size is 100 million triples and we vary data sizes from 100 million to 10 billion to evaluate the scalability of our approach.

LUBM. LUBM [12] is a customizable synthetic RDF data generator that models a university. LUBM includes 18 properties and 14 benchmark queries ($LQ_1 - LQ_{14}$) [40].

WatDiv. WatDiv [4] is a benchmark that enables diversified stress testing of RDF data management systems, which includes 86 properties. WatDiv provides a generator to generate test workloads, and we generate a query log of $1,000$ queries.

2) Real Datasets: We also use four real RDF datasets, YAGO2 [14], Bio2RDF [7], DBpedia [23] and LGD [33]. For the first two datasets, we use benchmark queries proposed in [2]; and for the others we use their open real query logs.

YAGO2. YAGO2 [14] is a semantic knowledge base, derived from Wikipedia, WordNet and GeoNames. It has 284 million triples and 98 properties. We use all four benchmark queries ($YQ_1 - YQ_4$) [2].

Bio2RDF. Bio2RDF [7] is an open source project to generate and provide Linked Data for the life sciences. It has 4 billion triples and $1,714$ properties. All five benchmark queries ($BQ_1 - BQ_5$) [2] are used in our experiments.

DBpedia. DBpedia [23] is an RDF dataset extracted from Wikipedia and contains about 1 billion triples and $124,000$ properties. There is an open available query log including $8,151,238$ queries [30].

LGD. LGD [33] is a large spatial RDF graph that has been derived from Open Street Map. This dataset contains more than 1 billion triples and $33,000$ properties. We also use an open real query log [30] including $1,702,961$ queries.

B. Partitioning Quality

The partitioning quality is measured as the percentage of IEQs and the end-to-end query performance. Our goal is to minimize the number of distinct crossing properties in vertex-disjoint partitioning. Table II shows the number of crossing properties, $|L_{cross}|$, and the number of crossing edges, $|E^c|$, in different partitioning approaches. Although MPC partitioning may result in more crossing edges than minimum edge-cut partitioning like METIS, it results in fewer number of crossing properties, especially for real RDF graphs YAGO2, DBpedia and LGD. In real RDF graphs, there are many properties. The more properties an RDF graph has, the smaller the size of the maximal WCC in the property-induced subgraph of a property is. Hence, many properties can be selected as internal properties. For example, in DBpedia, there are only 64 crossing properties in MPC, while there are more than $2,000$ crossing properties in other partitioning approaches. Since VP is an edge-disjoint partitioning that distributes edges according their properties, it has vertices but not edges across different partitions. Thus, there are no crossing properties or crossing edges, so we do not include VP in Table II.

Obviously, fewer crossing properties in MPC means that more queries are IEQs. Table III shows the percentage of the IEQs over the total benchmark queries in LUBM, YAGO2 and Bio2RDF, and also reports the percentages of the IEQs in query logs in WatDiv, DBpedia and LGD. Note that the percentages of IEQs in MPC are far higher than other vertex-disjoint partitioning techniques. For comparison, we report the percentages of star queries in Table III as these are the only ones that the other techniques can independently execute.

IEQs in Subject_Hash and METIS are *star* queries. Although Subject_Hash and METIS originally do not introduce the concept of crossing properties, we extend them by finding out their crossing properties (denoted as Subject_Hash+ and METIS+ in Table III). This slightly increases the number of IEQs in these systems beyond star queries (i.e., they have a few non-star IEQs). VP follows edge-disjoint partitioning and queries in VP can be IEQs only when all properties in the query are coincidentally assigned to a single partition. Thus, the percentages of IEQs are small.

C. Query Performance

1) Evaluation of Each Stage: In this experiment, we study the performance of our approach at each stage with regard to different benchmark queries. We report the running time of each stage, including the query decomposition time, local

[2]https://github.com/bnu05pp/mpc

TABLE II
NUMBER OF CROSSING PROPERTIES AND CROSSING EDGES IN DIFFERENT VERTEX-DISJOINT PARTITIONINGS

Datasets	MPC		Subject_Hash		METIS	
	$\|L_{cross}\|$	$\|E^c\|$	$\|L_{cross}\|$	$\|E^c\|$	$\|L_{cross}\|$	$\|E^c\|$
LUBM	5	29,971,560	14	62,377,786	13	21,853,766
WatDiv	17	95,497,642	31	95,786,527	31	58,100,544
YAGO2	5	128,758,514	45	142,274,454	43	58,912,138
Bio2RDF	36	2,044,500,633	398	2,184,075,117	-*	-
DBpedia	64	504,897,118	33,966	681,734,289	17,807	171,944,344
LGD	6	518,035,967	2,012	676,620,154	2,010	296,443,817

* - means that data size is beyond the capacity of METIS.

TABLE III
PERCENTAGE OF IEQS

	MPC	VP	Subject_Hash / METIS	Subject_Hash+	METIS+
LUBM	100%	28.57%	71.43%	71.43%	71.43%
WatDiv	60%	0%	50%	50%	50%
YAGO2	100%	0%	0%	0%	0%
Bio2RDF	100%	40%	80%	80%	80%
DBpedia	75.19%	24.25%	46.87%	51.87%	51.90%
LGD	99.95%	83.51%	96.95%	96.98%	96.98%

TABLE IV
EVALUATION OF EACH STAGE ON LUBM (IN MS)

Queries	LQ_1	LQ_2	LQ_3	LQ_4	LQ_5	LQ_6	LQ_7	LQ_8	LQ_9	LQ_{10}	LQ_{11}	LQ_{12}	LQ_{13}	LQ_{14}
QDT[1]	82	81	81	80	80	80	81	81	80	80	80	79	80	80
LET[2]	39	246	102	33	12	34,759	102	123	2,580	38	44	44	47	34,512
JT[3]	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	125	12,365	185	119	124	32,267	29,166	16,419	42,564	118	88	87	100	32,650

[1] QDT means the query decomposition time; [2] LET means the local evaluation time; [3] JT is the join time.

TABLE V
EVALUATION OF EACH STAGE ON YAGO2 AND BIO2RDF (IN MS)

Queries	YAGO2				Bio2RDF				
	YQ_1	YQ_2	YQ_3	YQ_4	BQ_1	BQ_2	BQ_3	BQ_4	BQ_5
QDT	81	83	81	81	80	80	81	81	82
LET	373	191	2,967	281	192	213	479	264	248
JT	0	0	0	0	0	0	0	0	0
Total	454	274	3,048	362	272	293	560	345	330

evaluation and the join time with regard to different queries in Tables IV and V.

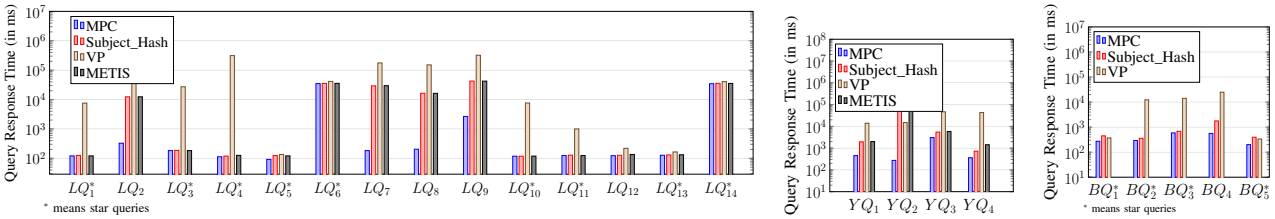
As shown in Table III, all benchmark queries in LUBM, YAGO2 and Bio2RDF are IEQs which can be executed independently while avoiding joins, so the join times for them are always 0. Meanwhile, the sizes of queries are small and few of them contain more than 10 triple patterns, so the times to parse and decompose the benchmark queries are also very small. Thus, the main differences among different queries are due to their local evaluation time. The local evaluation time mainly depends on two factors: the shape of the query graph and the existence of selective triple patterns. The complex queries (LQ_2 and LQ_9 in LUBM and YQ_1, YQ_3 and YQ_4 in YAGO2) often have higher local evaluation times, while the existence of selective triple patterns (LQ_1 in LUBM and BQ_1 and BQ_2 in Bio2RDF) significantly reduces local evaluation times. Further, LQ_6 has large input and low selectivity, and it takes much time to generate a large number of its results.

2) Query Performance Comparison: We evaluate query performance under different partitioning methods on LUBM, YAGO2 and Bio2RDF (Fig. 7). All benchmark queries are classified into two categories for reporting: star queries and other queries.

MPC, Subject_Hash and METIS partitioning lead to similar query response times for star queries, because they can be evaluated independently in any vertex-disjoint partitioning. However, for VP, few queries are IEQs, which often results in the worst performance.

MPC enables a significantly larger class of queries to be independently executable, and many IEQs in MPC require inter-partition join in other partitioning techniques: LQ_2, LQ_7, LQ_8, LQ_9 and LQ_{12} in LUBM, $YQ_1 - YQ_4$ in YAGO2 and BQ_4 in Bio2RDF. The performance of these queries under MPC is much better than under others, sometimes by orders of magnitude, due to the elimination of inter-partition joins.

We also use query logs on WatDiv, DBpedia and LGD to evaluate the effectiveness of our MPC partitioning. We randomly sample $1,000$ queries from the query log and report query response times in Fig. 8 across five values: minimum, maximum, sample median, and first and third quartiles. The minimum and first quartile query response times on different vertex-disjoint partitionings are similar, because there are nearly 25% IEQs in query logs on the vertex-disjoint partitionings, as shown in Table III. For these IEQs, independent



(a) LUBM (b) YAGO2 (c) Bio2RDF

Fig. 7. Online Performance Comparison on Benchmark Queries

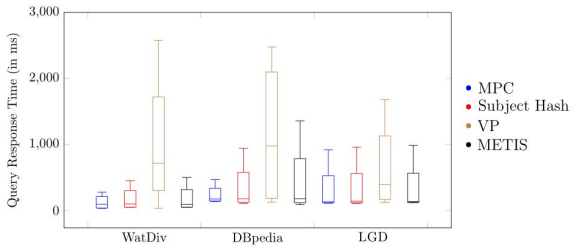


Fig. 8. Online Performance over Real Query Logs

execution can always result in high performance. Especially IEQs with high selectivity result in very low query response times in all partitioning methods. Meanwhile, the maximum and third quartile query response times differ sharply. As shown in Table III, MPC can localize more queries without inter-partition joins than other partitioning methods, so the independent execution on MPC results in the best performance, especially for DBpedia. For LGD, most queries are star queries – some are even one-triple queries (see Table III) – that are IEQs in all vertex-disjoint partitioning methods; thus, the performance improvement in MPC is not so significant. In addition, there inevitably exist some large queries containing many properties in real query logs, and their query response times in VP lead to worst performance.

Furthermore, there are also some queries that cannot be IEQs under any partitioning methods. These queries are decomposed as discussed earlier. Usually, the granularity of decomposed subqueries under MPC is larger than that in others, highlighting the fact that there are fewer subqueries leading to fewer inter-partition joins. This results in better performance for these queries under MPC.

Fig. 8 shows that the improvement in WatDiv is not as pronounced as in other datasets. Entities in WatDiv are less homogeneous and most entities share common properties. The percentage difference of IEQs between MPC and other partitioning is 10% in WatDiv (Table III), which is smaller than that in other datasets.

D. Additional Experiments

1) Offline Performance: Given an RDF graph, the entire offline process includes two steps: graph partitioning and loading each partition to the RDF engine. We report the total offline time as well as time of each step in Table VI.

The partitioning time mainly depends on the complexity of the partitioning schemes. VP and Subject_Hash only need to scan the dataset once and can directly place the triples

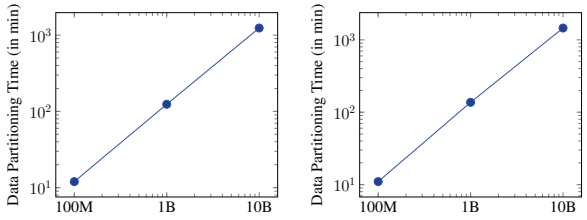
TABLE VI
PARTITIONING AND LOADING TIME (IN MINUTES)

Datasets	Strategies	Partitioning	Loading	Total
LUBM	MPC	12	15	27
	Subject_Hash	6	21	27
	VP	8	16	24
	METIS	11	18	29
WatDiv	MPC	11	46	57
	Subject_Hash	7	44	51
	VP	8	41	49
	METIS	10	89	99
YAGO2	MPC	34	215	249
	Subject_Hash	23	117	140
	VP	28	130	158
	METIS	51	137	188
Bio2RDF	MPC	790	975	1,765
	Subject_Hash	351	1,361	1,712
	VP	174	1,216	1,390
	METIS	-	-	-
DBpedia	MPC	262	745	1,007
	Subject_Hash	127	220	347
	VP	53	203	256
	METIS	171	123	294
LGD	MPC	204	389	593
	Subject_Hash	81	694	755
	VP	54	489	543
	METIS	145	581	726

according to their values of subjects or properties, so their partitioning times are less than MPC and METIS, as expected. Although the partitioning time of MPC is larger than VP and Subject_Hash, the offline performance gap is not significant, which can be tolerated considering the advantage of MPC in online query performance. Due to the introduction of disjoint-set forest, we reduce both space and time cost in MPC partitioning. However, METIS runs out of memory in large graphs such as Bio2RDF.

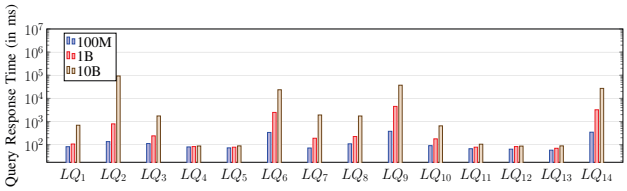
On the other hand, the loading time depends on the size balance among different partitions and the replication ratios. Since MPC also considers balancing the partiton sizes, the loading time of MPC is not much worse than others.

2) Scalability Test: We evaluate MPC's data scalability by varying the RDF graph sizes from 100 million triples to 10 billion triples in LUBM and WatDiv in Fig. 9 and Fig. 10. In Fig. 10, we report the query time of 14 LUBM queries and the average query time of 1000 sample generated queries in WatDiv. The results confirm that MPC partitioning is scalable in large graphs. Generally, as the size of RDF graph increases, the partitioning times and the response times increase as well.

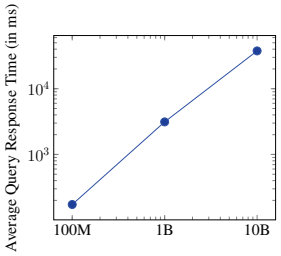


(a) LUBM (b) WatDiv

Fig. 9. Scalability Test of Offline Performance



(a) LUBM



(b) WatDiv

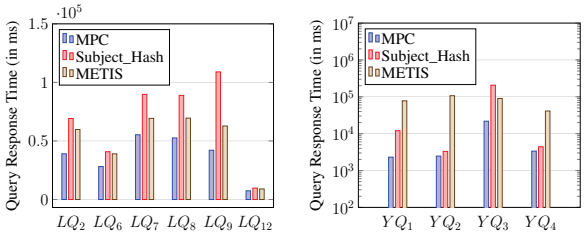
Fig. 10. Scalability Test of Online Performance

However, the rates of increase are slow, and the offline and online performance are scalable with graph size.

3) Partitioning-agnostic System Experiments: In this experiment, we LUBM and YAGO2 to evaluate MPC on our previous work gStoreD [28], [29]. gStoreD is a partitioning-agnostic system, and any vertex-disjoint partitioning method can be used. We compare the three vertex-disjoint partitioning methods, MPC, Subject_Hash and METIS, on gStoreD. Since star queries are always IEQs over different partitioning schemes and the query evaluation times of star queries are quite similar, we only compare the performance of non-star benchmark queries, i.e. LQ_2, LQ_6, LQ_7, LQ_8, LQ_9 and LQ_{12} in LUBM and all queries in YAGO2. Fig. 11 shows that fewer crossing properties in MPC often result in crossing matches, which indicates fewer local partial matches in gStoreD, so the running time over MPC is always the smallest among the three partitioning schemes.

4) Effectiveness of Our Approximate Greedy Algorithm: In this experiment, we compare our greedy algorithm (Algorithm 1) with a baseline that computes the exact optimal set of internal properties (denoted as MPC-Exact), to experimentally show the approximation ratio of our greedy algorithm. Since MPC partitioning is NP-complete, MPC-Exact can only run over LUBM, which only contains 18 properties. Table VII shows the experimental results.

The experimental results show that the greedy approximation results in the addition of only one more crossing property



(a) LUBM (b) YAGO2

Fig. 11. Online Performance Using gStoreD

on LUBM. There are many domains in LUBM and different domains are connected through some specific properties. It is not hard for the greedy algorithm to avoid selecting the properties across different domains as internal, since selecting them can connect different domains which result in a large WCC (i.e., a large selecting internal property cost in Definition 4.2). Most real RDF graphs often cover much more domains than LUBM, so we believe that the partitioning results of the greedy approximation can be sufficiently close to the optimal one across a wide range of graphs.

TABLE VII
COMPARISON OF OUR APPROXIMATE GREEDY ALGORITHM AND EXACT ALGORITHM ON LUBM

	$\lvert L_{cross}\rvert$	$\lvert E^c\rvert$	Partitioning Time (in minutes)
MPC	5	29,971,560	12
MPC-Exact	4	19,767,453	16

VII. CONCLUSIONS AND FUTURE WORK

We propose a new RDF graph partitioning scheme, called *minimum property-cut* (MPC), to minimize inter-partition joins in distributed SPARQL query execution. MPC first selects some internal properties and coarsens the RDF graph to a much smaller graph. The coarsened graph is partitioned and this partitioning is projected to the original RDF graph. The first step of the method is NP-complete; therefore, we propose a greedy heuristic using disjoint-set forest data structure to find an approximate solution. The queries that cannot completely avoid inter-partition joins are partitioned into a set of independently executable subqueries before the final result is computed. Our experiments reveal that at least 60% test queries are independently executable in MPC on both synthetic and real RDF graphs, which causes MPC to outperform existing partitioning approaches.

MPC can be further extended to property graphs, but its superiority in those graphs may not be as high as in RDF graphs. Real RDF graphs are often sparse and have a large number of properties. Furthermore, most properties cover a small number of edges. MPC is designed to exploit these characteristics. Suitability of MPC over property graphs with different characteristics require further study.

Acknowledgement. The corresponding author is Lei Zou, and the work is supported by National Key R&D Program of China (2020AAA0105200), NSFC under grants 61932001, 61961130390, U20A20174, 62172146 and 62172157. M. T. Özsu's research is supported by Natural Sciences and Engineering Research Council (NSERC) of Canada.

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

[3] R. Al-Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning. *VLDB J.*, 25(3):355–380, 2016.

[4] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *Proc. ISWC*, pages 197–212, 2014.

[5] A. Davoudian, L. Chen, H. Tu, and M. Liu. A workload-adaptive streaming partitioner for distributed graph stores. *Data Sci. Eng.*, 6(2):163–179, 2021.

[6] S. Dumbrava, A. Bonifati, A. N. R. Diaz, and R. Vuillemot. Approximate Querying on Property Graphs. In *SUM*, volume 11940 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2019.

[7] M. Dumontier, A. Callahan, J. Cruz-Toledo, P. Ansell, V. Emonet, F. Belleau, and A. Droit. Bio2RDF Release 3: A Larger Connected Network of Linked Data for the Life Sciences. In *Proc. ISWC-PD*, page 401–404, 2014.

[8] M. L. Fredman and M. E. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proc. STOC*, pages 345–354, 1989.

[9] L. Galárraga, K. Hose, and R. Schenkel. Partout: a distributed engine for efficient RDF processing. In *Proc. WWW (Companion Volume)*, pages 267–268, 2014.

[10] Z. Galil and G. F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.

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

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

[13] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. In *Proc. SIGMOD Conference*, pages 289–300, 2014.

[14] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[15] K. Hose and R. Schenkel. WARP: Workload-aware replication and partitioning for RDF. In *Proc. ICDE Workshops*, pages 1–6, 2013.

[16] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[17] 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. *TKDE*, 23(9):1312–1327, 2011.

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

[19] Z. Kaoudi, I. Manolescu, and S. Zampetakis. *Cloud-Based RDF Data Management*. Morgan & Claypool Publishers, USA, 2020.

[20] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

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

[22] 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. CLOUD*, pages 327–334, 2013.

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

[24] A. Madkour, A. M. Aly, and W. G. Aref. WORQ: Workload-Driven RDF Query Processing. In *Proc. ISWC*, pages 583–599, 2018.

[25] M. T. Özsu. A Survey of RDF Data Management Systems. *Frontiers Comput. Sci.*, 10(3):418–432, 2016.

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

[27] P. Peng, L. Zou, L. Chen, and D. Zhao. Adaptive Distributed RDF Graph Fragmentation and Allocation based on Query Workload. *TKDE*, 31(4):670–685, 2019.

[28] P. Peng, L. Zou, and R. Guan. Accelerating Partial Evaluation in Distributed SPARQL Query Evaluation. In *Proc. ICDE*, pages 112–123, Macao, China, 2019. IEEE.

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

[30] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo. LSQ: The Linked SPARQL Queries Dataset. In *Proc. ISWC*, pages 261–269, 2015.

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

[32] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri. Partitioning Trillion-Edge Graphs in Minutes. In *IPDPS*, pages 646–655. IEEE Computer Society, 2017.

[33] C. Stadler, J. Lehmann, K. Höffner, and S. Auer. LinkedGeoData: A Core for a Web of Spatial Open Data. *Semantic Web*, 3(4):333–354, 2012.

[34] C. Stadler, G. Sejdiu, D. Graux, and J. Lehmann. Sparklify: A Scalable Software Component for Efficient Evaluation of SPARQL Queries over Distributed RDF Datasets. In *Proc. ISWC*, pages 293–308, 2019.

[35] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to Partition a Billion-Node Graph. In *ICDE*, pages 568–579, Chicago, IL, USA, 2014. IEEE Computer Society.

[36] B. Wu, Y. Zhou, H. Jin, and A. Deshpande. Parallel SPARQL Query Optimization. In *Proc. ICDE*, pages 547–558, 2017.

[37] M. Wylot and P. Mauroux. DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud. *TKDE*, 28(3):659–674, 2016.

[38] L. Zeng and L. Zou. Redesign of the gStore System. *Frontiers Comput. Sci.*, 12(4):623–641, 2018.

[39] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In *Proc. ICDE*, pages 565–576, 2013.

[40] 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.