Parallel SPARQL query optimization

Wu, Buwen; Zhou, Yongluan; Jin, Hai; Deshpande, Amol

Published in:
Proceedings of the 33rd IEEE International Conference on Data Engineering (ICDE)

DOI:
10.1109/ICDE.2017.110

Publication date:
2017

Parallel SPARQL Query Optimization

Buwen Wu[†‡], Yongluan Zhou[‡], Hai Jin[†], Amol Deshpande[§]
[†]Huazhong University of Science and Technology, [‡]University of Southern Denmark, [§]University of Maryland at College Park
wubuwen@hust.edu.cn, y.zhou@acm.org, hjin@hust.edu.cn, amol@cs.umd.edu

***Abstract*—Existing parallel SPARQL query optimizers assume hash-based data partitioning and adopt plan enumeration algorithms with unnecessarily high complexity. Therefore, they cannot easily accommodate other partitioning methods and only consider an unnecessarily limited plan space. To address these problems, we first define a generic RDF data partitioning model to capture the common structure of various state-of-the-art RDF data partitioning methods. Then we propose a query plan enumeration algorithm that not only has an optimal efficiency, but also accommodates different data partitioning methods. Furthermore, based on a solid analysis of the complexity of the plan enumeration algorithm, we propose two new heuristic methods that can consider a much larger plan space than the existing methods, and at the same time can still confine the search space of the algorithm. An autonomous approach is proposed to choose one of the two methods by considering the structure and the size of a complex SPARQL query. We conduct extensive experiments using synthetic and a real-world dataset, which show the superiority of our algorithms in comparing to existing ones.**

I. Introduction

The simplicity and flexibility of RDF (Resource Description Framework) model have attracted an increasing number of organizations storing their data in the RDF format, i.e. triple statements as ⟨ subject, predicate, object ⟩. For example, the DBpedia community extracts structured information from Wikipedia and makes it available in the RDF format, whose size has reached 3 billion triples. The statistics from Linked Open Data Project show that more than 52 billion triples had been published [1]. With the rapidly increasing sizes of RDF data repositories, using MapReduce-like parallel data analytic platforms to process queries over big RDF datasets have recently attracted much attention [2]–[12].

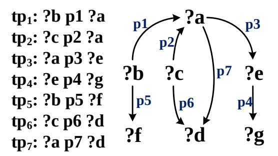
In a parallel engine, RDF datasets would be partitioned onto a cluster of computing nodes. Then a query would be first decomposed into several *local subqueries* to be executed in parallel without cross-node interactions. Given the outputs of the local subqueries, distributed joins are used to produce the final output. Optimizing query plans to minimize the cost of distributed joins is crucial to query performance [2]–[4], [6]. This is challenging because RDF queries tend to involve a large number of joins and complex join graphs simply because the RDF schema only contains three columns. We observe that existing parallel SPARQL optimizers [6], [7] adopt inefficient plan enumeration algorithms which have exponential amortized complexity. The inefficiency of plan enumeration limits the plan space that can be considered. In this paper, we attempt to answer the question of *"can we design a plan enumeration algorithm with optimal efficiency, which allows us to explore a larger plan space leading to better query plans?"*. We propose an efficient query plan enumeration algorithm that has linear amortized complexity per enumerated join operator. According to the previous studies on relational query optimizations [13]–[16], such an algorithm is of optimal efficiency, which means no other algorithms can enumerate the same number of query plans with lower complexity. To the best of our knowledge, this is the first work to study efficient algorithms to enumerate k-ary ($k \geqslant 2$) bushy plans.

In many RDF analytical applications [17], the sizes and the complexities of the join graphs could be very high, which result in a large plan space. For these cases, due to the NP-hardness of the query optimization problem, we cannot enumerate all the plans within a reasonable time even with our efficient enumeration algorithm. Similar to the state-of-the-art optimizers [6]–[8], we need heuristics to reduce the plan space. Therefore, we need to answer the second important question: *"can we design heuristics that can confine the complexity of the plan enumeration algorithm, and at the same time consider as many potentially good query plans as possible?"*. To avoid over-pruning the plan space, we carefully analyze the factors (such as the size and structure of the query graph) that determines the complexity of the plan enumeration algorithm, and then carefully prune the plan space to simplify the algorithm while keeping the potentially good query plans. We propose two heuristic-based approaches that are suitable for different types of queries as well as an algorithm to autonomously choose the appropriate heuristic according to the size and the structure of the join graph. In comparing to the existing optimizers [2]–[4], [6], our approach considers a much larger solution space and produces significantly better query plans, meanwhile, thanks to our efficient query plan enumeration algorithm, it runs much faster in most tested cases.

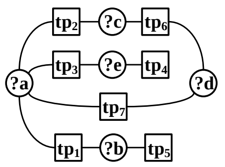
Last but not least, to maximize the use of local subqueries and to reduce the cost of distributed joins, many RDF data partitioning methods [2]–[5] have been proposed for different types of workloads [9]. However, many existing optimizers [6]–[8] that consider static data partitioning assume data is hash partitioned and hence it remains unclear how to extend them to other partitioning methods. We argue that a parallel RDF engine should be able to choose its data partitioning method according to the actual application. Hence the last question we attempt to answer is *"can a query optimizer accommodate different data partitioning methods?"*. We design our query optimizer as partition-aware but not tightly coupled with a particular data partitioning method. This is achieved by employing a simple yet effective generic RDF data partitioning model, which can provide the query optimizer with necessary information of the actual data partitioning. In summary, our contributions include:

- We propose a generic model to capture the structure of existing static RDF data partitioning methods, which

The work was done while Buwen Wu was visiting the University of Southern Denmark. Corresponding author: Yongluan Zhou.



(a) SPARQL Query



(b) Join Graph

Fig. 1. SPARQL Query and Join Graph

enables an efficient algorithm to detect local subqueries.

- We propose an efficient top-down join enumeration algorithm that enumerates query plans with k-way joins, which has a linear amortized complexity per enumerated join operator. This algorithm can also be applied to enumerating multi-way bushy plans for complex join queries over relational data.
- To optimize large SPARQL queries, we introduce two new heuristic algorithms to limit the search space of our enumeration algorithm, and a novel autonomous algorithm to choose the most suitable algorithms to be applied according to the query structure.
- To compare with the state-of-the-art approaches, we conduct extensive experiments using a synthetic workload, a benchmark and a real-world dataset. The results show that our algorithms significantly outperform the state-of-the-art both in optimization time and query performance.

Our optimization algorithms are generic enough to be applied to relational query optimization. However, they are only tested with RDF queries and data partitioning methods that are particularly designed for RDF data.

II. Problem Formulation

A. RDF and SPARQL Query

An RDF dataset, i.e. a set of triples $\langle$subject, predicate, object$\rangle$, can be represented as an RDF graph. An RDF graph is a directed labeled graph, denoted as $G_R = (V_R, E_R)$. V_R is a set of vertices, corresponding to all the subjects and objects of the RDF triples. $E_R \subseteq V_R \times V_R$ is a set of directed edges from the subjects to the objects. Each edge is attached with a label, referring to the predicate associated with it.

SPARQL is a W3C standard query language for RDF datasets. The main mechanism of SPARQL is specifying subgraph matching queries. A subgraph matching query Q is a set of triple patterns $Q = \{tp_1, tp_2, ..., tp_n\}$. A triple pattern tp is a triple whose subject, predicate or object can be either a constant or a variable. Similar to RDF triples, a SPARQL query can also be represented as a directed labeled graph, denoted by $G_Q = (V_Q, E_Q)$. For instance, Figure 1a shows a query with seven triple patterns and its corresponding query graph.

Given an RDF graph G_R and a query Q, a match $\mu(G_Q)$ of G_Q with respect to G_R is a subgraph of G_R that satisfies the following conditions: (1) $\mu(G_Q)$ and G_Q are isomorphic, (2) the constant vertices and the constant edge labels in G_Q should be identical to those in $\mu(G_Q)$. Essentially, to evaluate a query on an RDF graph is to find all the matches of this query in the RDF graph.

B. Query Model

A query is evaluated by scanning the bindings for each triple pattern and joining them on their shared variables. Our algorithm uses a join graph to capture the join relationships among a query's triple patterns.

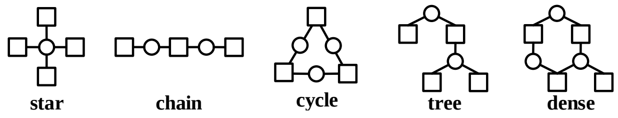


Fig. 2. Types of Join Graph. The squares and circulars indicate the triple patterns and join variables respectively.

Definition 1: (Join Graph) The join graph $J(Q)$ of a query Q is a bipartite graph $J(Q) = (V_T, V_J, E_J)$, where V_T is a set of vertices, each corresponding to one triple pattern in Q, V_J is another set of vertices, each corresponding to a shared variable among the triple patterns of Q, and $E_J \subseteq V_T \times V_J$, is a set of edges, each connecting one vertex v_t from V_T with another one v_j from V_J and denoting that the triple pattern v_t contains the join variable v_j.

In the join graph, the *neighborhood* of a join variable v_j is a set of triple patterns that contain v_j, denoted by $N_{tp}(v_j)$. The *degree* of v_j is denoted by $|N_{tp}(v_j)|$.

Example 1: Figure 1b is the join graph of Figure 1a. A rectangle vertex denotes a triple pattern, while a circular one is a join variable shared by multiple triple patterns. The neighborhood of the join variable $?c$ is denoted by $N_{tp}(?c) = \{tp_2, tp_6\}$ and the degree of $?c$, $|N_{tp}(?c)| = 2$. □

A subquery SQ of Q is a subset of Q's triple patterns. Similarly, a join graph with respect to the subquery SQ can be denoted as $J(SQ)$. We call a query or a subquery *connected*, if its query graph is connected. As a side note, evaluating a non-connected subquery involves cross-products.

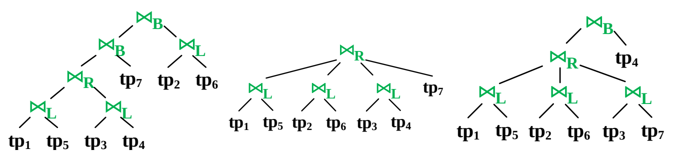
Based on the join graph, we can categorize a query as: a star query whose triple patterns share a single join variable, a chain query whose join graph is a chain, a cycle query whose join graph is a cycle, a tree query whose join graph is an acyclic graph or a dense query whose join graph contains cycles. Figure 2 gives an example for each query type.

C. Data Partitioning Model

In a parallel RDF processing engine, a dataset is usually partitioned and allocated to a set of computing nodes by a data partitioning algorithm. In this paper, we mainly focus on static partitioning algorithms and we discuss about the integration with dynamic partitioning in the extended version [18]. Although the different data partitioning algorithms proposed in the literature [2]–[5] differ in the implementation and performance, they share a common overall structure. We introduce a generic model to capture this structure as follows.

This model consists of two conceptual phases: combining and distributing. In the combining phase, the triples correlated to a vertex are assembled into an indivisible partitioning element by a generalized *combine* function. Specifically, given an RDF graph $G_R = (V_R, E_R)$, in the combining phase, for each vertex $v \in V_R$, $combine(v, G_R)$ assembles a set of triples related to v into an indivisible partitioning element anchored at v, denoted as e_v. The *combine* function is used to choose the triples to be included in the partitioning element e_v:

$$\forall v \in V_R: \quad e_v \leftarrow combine(v, G_R) \tag{1}$$



(a) TriAD Plan (b) MSC Plan (c) DP-Bushy Plan

Fig. 3. The query plans generated by the state-of-the-art SPARQL query optimization algorithms.

Note that, for two different vertices v and v', it is possible that $e_v = e_{v'}$.

After the combining phase, there are at most $|V_R|$ partitioning elements, each consisting of a set of triples. A triple can appear in multiple partitioning elements anchored at different vertices. Then, in the distributing phase, another function $distribute$ distributes each partitioning element to a computing node. This phase runs as:

$$\forall e_v : \qquad \mathscr{P}_i \leftarrow distribute(e_v) \tag{2}$$

where $\mathscr{P}_i$ denotes a computing node. The main purpose of the function $distribute$ is to reduce the duplicate triples (a triple can appear in several partitioning elements) and balance the load of computing nodes [2]–[4].

Example 2: We show the examples of three different data partitioning algorithms proposed in [2]–[4] respectively. For the un-one hop algorithm in [4], the $combine(v, G_R)$ function assemble all the triples in G_R, whose subject or object is v, into the partitioning element e_v. Then the $distribute$ function is a graph partitioner, e.g. METIS [19], to place the vertices with the partitioning elements into different computing nodes.

For the 2f algorithm in [3], the $combine(v, G_R)$ function assembles all the edges within 2-hop distance from v. Then the $distribute$ function assigns the partitioning elements to the computing servers by the hash value of v.

In the Path-BM algorithm in [2], all the end-to-end paths starting from a start vertex v would be merged into one element (definition can be found in [2]). In our model, the $combine$ function is applied on each start vertex v and assembles all the triples that are reachable from v. Then the $distribute$ function is the bottom-up path merging algorithm in [2]. □

D. Query Plan and Join Algorithms

A query plan of a query Q is a labeled bushy tree, denoted by $p(Q)$. It is a physical query plan that indicates how to evaluate Q in the system. In $p(Q)$, each leaf node denotes the bindings of a triple pattern. An inner node denotes a k-way join operator ($k \geqslant 2$), which has k inputs. The k-way join operator joins all the intermediate results from its children using the labeled join algorithm and output the join intermediate results to its parent node (if it is not the root node). Note that, an inner node represents the join results of the bindings of all its descendant triple patterns.

The k-way join operator can be evaluated by a number of different algorithms, which have different cost in different situations. We discuss different types of algorithms below.

Given the data partitions, if a join can be processed in parallel on each computing node without any cross-node data communication, then it is called a *local join*. A query that can be evaluated by local joins is called a *local query*. Whether a query is a local query mainly relies on the data partitioning algorithm. According to our generic data partitioning model, we have the following definition.

TABLE I. JOIN COST

op	C_{io}	C_{trans}	C_{join}
Local	$\alpha\cdot\sum\lvert SQ_i\rvert$	0	$\gamma_L\cdot\lvert\bowtie_{i=1}^{k} SQ_i\rvert$
Broadcast	$\alpha\cdot\sum\lvert SQ_i\rvert$	$\beta_B\cdot(\sum\lvert SQ_i\rvert - \max\lvert SQ_i\rvert)\cdot n$	$\gamma_B\cdot\lvert\bowtie_{i=1}^{k} SQ_i\rvert$
Repartition	$\alpha\cdot\sum\lvert SQ_i\rvert$	$\beta_R\cdot\sum\lvert SQ_i\rvert$	$\gamma_R\cdot\lvert\bowtie_{i=1}^{k} SQ_i\rvert$

Definition 2: (Local Query) Given an RDF graph $G_R = (V_R, E_R)$, a query Q and a data partitioning algorithm (i.e. a $combine$ function and a $distribute$ function), if, for each subgraph of G_R that matches G_Q, denoted as $\mu(G_Q)$, there exists a vertex $v \in V_R$ such that $\mu(G_Q)$ is a subgraph of $combine(v, G_R)$, then Q is a local query.

Since for each v, $combine(v, G_R)$ is an indivisible partitioning element, therefore, all the matches of a query defined above can be found using local joins. If a join is not a local query, then it has to be processed by a distributed join algorithm. We consider not only binary but also multi-way distributed join algorithms, which join multiple inputs at the same time. Two distributed join algorithms are considered:

1) The *k-way broadcast join* algorithm first broadcasts k-1 smaller inputs to all the computing nodes holding the partitions of the largest input. Then the join can be executed in parallel.

2) The *k-way repartition join* algorithm repartitions all the inputs by the shared join variable, and then performs the join in parallel.

These two algorithms can be implemented in MapReduce [20] and in Spark [21].

Example 3: Figure 3 shows several query plans of the query shown in Figure 1. $\bowtie_L$, $\bowtie_B$ and $\bowtie_R$ denote local join, broadcast join and repartition join respectively. □

E. Cost Model and Problem Statement

While our optimization algorithm is independent on the cost model, we provide its details here for completeness. Given a query Q, the cost of a parallel query plan $p(Q)$, denoted as $C(p(Q))$, can be estimated as as follows:

$$C(p(Q)) = \max\{C(p(SQ_1)), ..., C(p(SQ_k))\} + C(op_{join}) \tag{3}$$

Each SQ_i ($1 \leqslant i \leqslant k$) is a subquery of Q, such that all of them can be joined by a k-way join operator op_{join} to produce the complete results of Q. Since each input of a k-way join is the intermediate results of a subquery, we define the cost of $p(Q)$ recursively by using the cost of the plans of the subqueries and the k-way join operator. We only consider the cost of the subquery with the maximal cost to account for the concurrent execution of the subqueries.

The cost of a k-way join operator consists of three components: I/O cost, network cost and join computation cost, denoted by C_{io}, C_{trans} and C_{join} respectively.

$$C(op_{join}) = C_{io} + C_{trans} + C_{join} \tag{4}$$

Let $|SQ_i|$ and $|\bowtie_{i=1}^{k} SQ_i|$ denote the cardinality of a subquery and the cardinality of the join of a set of subqueries respectively. We summarize the cost of three join algorithms in Table I, where n denotes the number of computing nodes in the cluster, α, β_{op} and γ_{op} are the normalization factors for I/O, data transfer and join computation.

The above cost model is simple but we find it consistent with the query running time in our experiments. Developing a sophisticated and more accurate cost model is out of the scope of this paper. Furthermore, since our algorithm is loosely coupled with the cost model and explores a larger space than existing ones, a better cost model can certainly be used to improve our optimization results.

Problem Statement. Given a query Q, the *query optimization problem* is to find a k-way bushy plan without Cartesian-product that has minimum cost estimated by the cost model.

III. Efficient Query Plan Enumeration

Query plan enumeration algorithm is an important component of a query optimizer. Its complexity significantly affects the number of possible query plans that we can afford to explore. However, we observe that the state-of-the-art optimizers for parallel SPARQL engines do not employ efficient algorithms to enumerate bushy plans involving multi-way joins, and therefore they have to opt for heuristics that only consider a limited plan space. For example, both the MSC algorithm [6] and the DP-Bushy algorithm [7] consider multi-way joins. However, MSC [6] needs to run an expensive minimum set cover algorithm to construct each level of the query plan. As minimum set cover is an NP-hard problem, the complexity of enumerating the join operators at each level is exponential. DP-Bushy [7] does not check connectivity in the join graph in enumerating join operators. It can only check and eliminate Cartesian products afterwards. Enumerating plans that eventually lead to Cartesian products is inefficient. As proved in [14], for chain and cycle queries, such an approach has an exponential amortized complexity per join operator. TriAD [8] adopts a bottom-up dynamic programming algorithm to enumerate binary bushy plans, which is similar to [13]. As proved in [13], its amortized complexity per enumerated join operator is linear to the number of triple patterns, hence it is of optimal efficiency. However, it cannot enumerate multi-way bushy plans. We will return to the limitation of the heuristics adopted by the existing optimizers in Section IV.

To address the inefficiency and limitations of the existing plan enumeration algorithms, we propose a new algorithm to enumerate k-ary bushy plans with optimal efficiency.

A. Connected Multi-Division

Given a query Q with triple patterns $\{tp_1, ..., tp_n\}$. A k-way join ($k \geqslant 2$) of Q is a join of k inputs $SQ_1, ..., SQ_k$ on a common join variable v_j that produces the complete results of Q. Each input SQ_i is a subquery of Q. We call $(SQ_1, ..., SQ_k, v_j)$ a *connected multi-division* (or ***cmd*** for short) of Q, if it meets the following definition:

Definition 3: (Connected Multi-Division) Given a connected query Q and its join graph $J(Q) = (V_T, V_J, E_J)$, $(SQ_1, ..., SQ_k, v_j)$ is a connected multi-division of Q on the join variable $v_j \in V_J$, if all the following conditions hold: (1) $\forall SQ_i, SQ_j$ ($1 \leqslant i, j \leqslant k$ and $i \neq j$), $SQ_i \cap SQ_j = \emptyset$, (2) $SQ_1 \cup ... \cup SQ_k = Q$, (3) $\forall SQ_i$, SQ_i is connected and $\exists tp \in SQ_i$ s.t. $tp \in N_{tp}(v_j)$.

Algorithm 1: Top-Down Join Enumeration

```
Function: GetBestPlan($Q, isLocal$)
   Input: Query $Q = \{tp_1, ..., tp_n\}$
   Output: The best query plan of $Q$
 1 if $BestPlan[Q] = Null$ then
 2     if $isLocal \neq true$ then
 3         $isLocal \leftarrow$ IsLocalQuery($Q$);
 4     $BestPlan[Q] \leftarrow$ BestPlanGen($Q, isLocal$);
 5 return $BestPlan[Q]$;
Function: BestPlanGen($Q, isLocal$)
   Output: The best plan $bPlan$
 6 $bPlan \leftarrow$ NULL ;                          // Let $C(NULL) = \infty$
 7 if $|Q| = 1$ then
 8     $bPlan \leftarrow$ best scanning plan of $Q$;
 9 else
10     if $isLocal = true$ then $bPlan \leftarrow$ local join plan of $Q$;
11     ;
12     foreach $cmd$ in ConnMultiDivision($Q$) do
         /* ConnMultiDivision($Q$) iterates $D_{cmd}(Q)$ */
13         foreach subquery $SQ_i \in cmd$ do
14             $p(SQ_i) \leftarrow$ GetBestPlan($SQ_i, isLocal$);
15         $OP \leftarrow \{B, R\}$;        // broadcast and repartition join
16         foreach join operator $op \in OP$ do
17             $cPlan \leftarrow$ BuildPlan($op, p(SQ_1), ..., p(SQ_{|cmd|})$);
18             if $C(cPlan) < C(bPlan)$ then
19                 $bPlan \leftarrow cPlan$;
20 return $bPlan$;
Function: IsLocalQuery($Q$)
   Output: whether $Q$ can be processed locally
21 if first time then generate all maximal local queries of $Q$;
22 ;
23 foreach $v \in V_Q$ do
24     if $Q$ is the subquery of the maximal local query at $v$ then
25         return true;
26 return false
```

Note that, each cmd represents a k-way join. Condition (3) is to avoid Cartesian products. The set of all cmds of the query Q is denoted by $D_{cmd}(Q)$. We do not consider the order of the subqueries in a cmd, e.g., $(SQ_1, SQ_2, ..., SQ_k, v_j)$ is equivalent to $(SQ_2, SQ_1, ..., SQ_k, v_j)$.

Example 4: For the query in Figure 1, $(\{tp_1, tp_5\}, \{tp_7\}, \{tp_2, tp_6\}, \{tp_3, tp_4\}, ?a)$ is a cmd and so is $(\{tp_1, tp_5, tp_7\}, \{tp_2, tp_6\}, \{tp_3, tp_4\}, ?a)$. □

B. Top-down Join Enumeration

The top-down join enumeration algorithm generates all the query plans of the given query. It adopts the memoization algorithm [14], [15], which recursively enumerate the query plans and choose the best one. There are two important functions in the algorithm: `ConnMultiDivision` and `IsLocalQuery`. `ConnMultiDivision` is to enumerate $D_{cmd}(Q)$, i.e. the cmds of query Q. Each cmd corresponds to one multi-way join operator. `IsLocalQuery` is used to examine whether an input query is a local query, and makes the algorithm partition-aware.

The pseudo-code is shown in Algorithm 1. The input is a query, i.e. a set of triple patterns, and the output is the optimized query plan. In our implementation, a query or a subquery is encoded into a bitset. Each bit indicates if a triple pattern is contained in a subquery. The algorithm maintains a global data structure, called $BestPlan$, to store the best query plan generated for each subquery. Hence, the procedure first checks if the best plan of the input query has already been generated. If not, it needs to generate the best plan. The idea is to try every possible k-way join of the input query, i.e. the cmd of the input query (recall Definition 3), and find the best plan for each subquery in the cmd by recursively calling function `GetBestPlan`. Then it enumerates the feasible join algorithms to construct all the possible join plans, and computes the cost for each of them. Finally it stores the plan with the lowest cost into $BestPlan$.

Function `isLocalQuery` uses the generic partitioning model to check if a subquery is local. Due to the space limit, we only briefly explain the algorithm and refer the readers to the extended version [18] for the detail analysis. First, given a query Q and a vertex $v \in G_Q$, we define the *maximal local query* at v as a local query that has the largest number of triple patterns in Q containing v, denoted as $MLQ_v(Q)$, which can be obtained by calling $combine(v, G_Q)$ (Section II-C). The reason that $combine(v, G_Q)$ is a local query is that if there is a match of $combine(v, G_Q)$ in G_R, then there must exist a match of v in G_R, denoted as $\mu(v)$, such that the match of $combine(v, G_Q)$ is a subgraph of $combine(\mu(v), G_R)$. Furthermore, $combine(v, G_Q)$ is the maximal local query at v that we can deduce from the partitioning model and $combine(v, G_Q)$ covers all the local queries that can be detected in the existing static partitioning methods [2]–[5]. Accordingly, to check whether a subquery SQ is a local query, we first generate the maximal local queries at all the vertices in V_Q and then check if SQ is a subquery of one of them. The worst-case complexity of this approach is $\Theta(|V_Q|)$.

Example 5: Suppose the data are partitioned using path partitoning [2]. For the query in Figure 1a, by applying $combine(?b, G_Q)$, we can get the maximal local query at $?b$ as $\{tp_1, tp_3, tp_4, tp_5, tp_7\}$. This is because $combine(v, G_R)$ in the path partitioning algorithm returns all the triple patterns that are reachable from vertex v. One can also see that all the subqueries of $\{tp_1, tp_3, tp_4, tp_5, tp_7\}$ are local queries. □

C. Connected Multi-Division Enumeration

To achieve optimal efficiency, we should only enumerate multi-divisions that are connected. The naive strategy of enumerating *cmd* of a query Q is a generate-and-check method, which generates all multi-divisions and then check if they fulfill the definition of *cmd*s. This method suffers from two drawbacks. Firstly, the number of multi-division is huge, which is equal to $B_{|V_T|}$, the *Bell Number*, where $|V_T|$ is the number of triple patterns in Q. Secondly, the cost of checking whether a multi-division is connected or not is high, which is $\Theta(|V_T|)$. Thus, we aim at designing an efficient algorithm that only enumerates the *cmd*s, and only enumerates each *cmd* once.

To achieve the above goal, we begin with solving the special case, enumerating connected binary-divisions. A connected binary-division (or ***cbd*** for short) is a special case of *cmd*, which has only two subqueries. The set of all *cbd*s of a query Q is denoted by $D_{cbd}(Q)$. Our idea of using $D_{cbd}(Q)$ to enumerate *cmds* is as follows. Given a query Q, each of its *cbd*s is also a *cmd*. In addition, suppose (SQ_1, SQ_2, v_j) is a *cbd* of Q, if SQ_2 has a *cbd* on the join variable v_j, denoted as (SQ_{21}, SQ_{22}, v_j), then $(SQ_1, SQ_{21}, SQ_{22}, v_j)$ is a *cmd* of Q. Again, if SQ_{22} has a *cbd* on v_j, which is $(SQ_{221}, SQ_{222}, v_j)$, then $(SQ_1, SQ_{21}, SQ_{221}, SQ_{222}, v_j)$ is a *cmd* of Q, and so on. In this way, we can enumerate all the possible *cmd*s.

Enumeration of connected binary-division. We first present the algorithm to find all the *cbd*s on a particular join variable of a query, which is presented in Algorithm 2. Given a query Q and a join variable v_j in $J(Q)$, the basic idea of this algorithm is to recursively extend a connected subquery SQ.

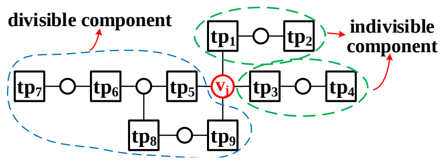


Fig. 4. Indivisible and Divisible Components

Algorithm 2: Connected Binary-Division Enumeration

```
Function: ConnBinDivision($Q, v_j$)
    Input: A query $Q$ and a join variable $v_j$
    Output: $D_{cbd}(Q)$ on the join variable $v_j$
 1  $\mathbb{C}_{v_j}$ ←GetComponents($Q, v_j$);
 2  CBDRec($\emptyset, \emptyset, Q$);
Function: CBDRec($SQ, X, Q$)
    Output: emits connected binary-divisions
 3  if $SQ = Q$ and $SQ \cap X \neq \emptyset$ then return;
 4  ;
 5  if $SQ \neq \emptyset$ then
 6      Emit(($SQ, Q \backslash SQ, v_j$));          // emit a $cbd$ $(SQ, Q \backslash SQ, v_j)$
 7  $X' \leftarrow X$;
 8  if $SQ = \emptyset$ then
 9      $A \leftarrow$ {an arbitrary triple pattern in $N_{tp}(v_j)$};
10  else
11      $A \leftarrow ((Adj(SQ) \cap Q) \backslash SQ) \backslash X$;
12  foreach $tp \in A$ do
13      find the $component \in \mathbb{C}_{v_j}$ that contains $tp$;
14      if $component$ is indivisible then
15          CBDRec($SQ \cup component, X', Q$);
16      else
17          $S \leftarrow$ GetPartInComponent($component$);
18          CBDRec($SQ \cup \{tp\} \cup S, X', Q$);
19      $X' \leftarrow X' \cup \{tp\}$;
```

Initially, SQ contains an arbitrary triple pattern in $N_{tp}(v_j)$, which contains all v_j's neighboring triple patterns in the query graph. If the extension of SQ does not make $Q \backslash SQ$ mismatch the *cbd* conditions, then it outputs one *cbd*.

Note that, the size of $D_{cbd}(Q)$ might be huge. Therefore, to avoid storing all of them, we implement Algorithm 2 as an Iterator, which generates and returns one *cbd* on each invocation. The `Emit` function (line 5) is similar to the `yield return` statement in C#. Concretely, it returns one *cbd* at a time and maintains the current running state. When the iterator function is called again, the execution will be restarted from the maintained state to generate the next *cbd*.

In line 1 of Algorithm 2, we first divide the query into a set of connected components by removing v_j from the join graph $J(Q)$, and save them in $\mathbb{C}_{v_j}$. Each of the resulting connected components has at least one triple pattern in $N_{tp}(v_j)$. The connected components can be classified into two categories: *indivisible component*, which has only one triple pattern adjacent to v_j, and *divisible component*, which has more than one triple pattern adjacent to v_j.

Example 6: In the following, we use the join graph in Figure 4 as the running example to illustrate our algorithm. Figure 4 shows the indivisible and divisible components in the example join graph after removing v_j. Here $\mathbb{C}_{v_j} = \{\{tp_1, tp_2\}, \{tp_3, tp_4\}, \{tp_5, tp_6, tp_7, tp_8, tp_9\}\}$, where $\{tp_1, tp_2\}$ and $\{tp_3, tp_4\}$ are indivisible components and $\{tp_5, \cdots, tp_9\}$ is a divisible component. Each component contains at least one triple pattern in $N_{tp}(v_j)$. □

Then Algorithm 2 calls the recursive function `CBDRec` (line 2) to produce the *cbd*s on the join variable v_j. In this function, SQ is extended incrementally on each recursive invocation. SQ is initialized as $\emptyset$ and A is initialized with an arbitrary triple pattern in $N_{tp}(v_j)$ (line 7–8) in the first invocation. In the subsequent invocations, A contains all the neighboring triple patterns of SQ (line 10). For a triple pattern $tp \in A$, if tp is contained in an indivisible component $\mathbb{C}_{v_j}[i] \in \mathbb{C}_{v_j}$, then we have to use the whole $\mathbb{C}_{v_j}[i]$ to extend SQ (line 13–14) according to the following lemma.

Lemma 1: If $(SQ, Q\backslash SQ, v_j)$ is a connected binary-division and there exists an indivisible component $\mathbb{C}_{v_j}[i] \subsetneq (Q\backslash SQ)$, then, *(1)* $(SQ \cup \mathbb{C}_{v_j}[i], Q\backslash(SQ \cup \mathbb{C}_{v_j}[i]), v_j)$ is also a *cbd*, *(2)* for any subset $\mathscr{S} \subsetneq \mathbb{C}_{v_j}[i]$, $(SQ \cup \mathscr{S}, Q\backslash(SQ \cup \mathscr{S}), v_j)$ is not a *cbd*. □

Due to the space limit, the proofs of all the lemmas and theorems in this paper are omitted but can be found in [18].

Example 6: ***(continued)*** Suppose A is set to $\{tp_1\}$ in the first run of CBDRec. Since tp_1 belongs to an indivisible component $\mathbb{C}_{v_j}[1] = \{tp_1, tp_2\}$, we should extend SQ with the whole component $\mathbb{C}_{v_j}[1]$. Therefore, SQ is set to $\emptyset \cup \{tp_1, tp_2\}$ (line 13-14). Then the recursive call of CBDRec produces a *cbd*, which is $(\{tp_1, tp_2\}, \{tp_3, tp_4, \cdots, tp_9\}, v_j)$. □

However, if $tp \in A$ is contained in a divisible component, then we do not need to use the whole component containing tp to extend SQ. Instead, we use the following lemma to extend SQ to generate a *cbd*.

Lemma 2: Assume $(SQ, Q\backslash SQ, v_j)$ is a connected binary-division. If a triple pattern $tp \in A$ is contained in a divisible component $\mathbb{C}_{v_j}[k] \in \mathbb{C}_{v_j}$, then $\mathbb{C}_{v_j}[k]\backslash(SQ \cup \{tp\})$ consists of a set of connected components $S_1, ..., S_i$. Then

(1) if $i = 1$, $(SQ \cup \{tp\}, Q\backslash(SQ \cup \{tp\}), v_j)$ is a cbd.

(2) if $i > 1$, in $S_1, ..., S_i$, there must be only one connected component that contains the triple patterns in $N_{tp}(v_j)$. Assume S_1 contains the triple patterns in $N_{tp}(v_j)$ and $S = S_2 \cup ... \cup S_i$, then $(SQ \cup \{tp\} \cup S, Q\backslash(SQ \cup \{tp\} \cup S), v_j)$ is a cbd, and for any subset $\mathscr{S} \subset \{tp\} \cup S$, $(SQ \cup \mathscr{S}, Q\backslash(SQ \cup \mathscr{S}), v_j)$ is not a *cbd*. □

If we add tp contained in a divisible component $\mathbb{C}_{v_j}[k]$ into SQ, then $\mathbb{C}_{v_j}[k]\backslash(SQ \cup \{tp\})$ would be divided into a set of connected components $S_1, ..., S_i$. Assume S_1 contains the triple patterns in $N_{tp}(v_j)$, function GetPartInComponent (line 16) returns S, which is the union of the remaining components, i.e. $S = S_2 \cup ... \cup S_i$. Then we extend SQ with tp as well as all the triple patterns in S. Then the next recursive invocation (line 17) generates a new *cbd* at line 5.

Example 6: ***(continued)*** Recall that the initial invocation of CBDRec sets SQ as $\{tp_1, tp_2\}$, therefore, in the next recursive invocation, line 10 sets $A = \{tp_5, tp_3, tp_9\}$, which are the neighbors of SQ. Suppose lines 11 picks tp_5, which belongs to the divisible component $\mathbb{C}_{v_j}[3]$. Since $\mathbb{C}_{v_j}[3]\backslash(SQ \cup tp_5)$, which is equal to $\{tp_3, tp_4, tp_6, tp_7, tp_8, tp_9\}$, contains only one connected component, according to Lemma 2-(1), SQ can be extended with tp_5, i.e. $SQ = \{tp_1, tp_2, tp_5\}$. Then the next recursive invocation of CBDRec produces a new *cbd* (lines 4–5), which is $(\{tp_1, tp_2, tp_5\}, \{tp_3, tp_4, tp_6, tp_7, tp_8, tp_9\}, v_j)$.

Now in line 10, A is set to $\{tp_6, tp_8, tp_9, tp_3\}$. Suppose line 11 picks tp_6, which again belongs to the divisible component $\mathbb{C}_{v_j}[3]$. $\mathbb{C}_{v_j}[3]\backslash(SQ \cup \{tp_6\}) = \{tp_7, tp_8, tp_9\}$ contains two connected components, which are $\{tp_7\}$ and $\{tp_8, tp_9\}$ repsectively. Based on Lemma 2-(2), $\{tp_7\}$ does not contain any triple patters in $N_{tp}(v_j)$, so SQ should be extended with tp_6 and the component $\{tp_7\}$, i.e. $SQ = \{tp_1, tp_2, tp_5, tp_6, tp_7\}$ (line 17). Then the next recursive invocation of CBDRec produces a new *cbd*, which is $(\{tp_1, tp_2, tp_5, tp_6, tp_7\}, \{tp_3, tp_4, tp_8, tp_9\}, v_j)$. □

Algorithm 3: Connected Multi-Division Enumeration

```
Function: ConnMultiDivision($Q$)
    Input: A query $Q$ and $J(Q) = (V_J, V_T, E_J)$
    Output: $D_{cmd}(Q)$
1   foreach $v_j \in V_J$ do
2   |   $stack$.Init(); $SQ_L \leftarrow Q$;
3   |   CMDRec($v_j, SQ_L, stack$);
Function: CMDRec($v_j, SQ_L, stack$)
    Output: emits connected multi-divisions
4   if $stack \neq \emptyset$ then
5   |   $cmd \leftarrow \emptyset$;
6   |   foreach $SQ \in stack$ do $cmd$.Add($SQ$);
7   |   ;
8   |   $cmd$.Add($SQ_L$); $cmd$.Add($v_j$);
9   |   Emit($cmd$);
10  $R \leftarrow N_{tp}(v_j) \cap SQ_L$;
11  if $|R| = 1$ then
12  |   if $|stack| > 0$ then $stack$.Pop();
13  |   ;
14  |   return;
15  foreach $(SQ'_L, SQ''_L, v_j)$ in ConnBinDivision($SQ_L, v_j$) do
    /* ConnBinDivision($SQ_L, v_j$) iterates $D_{cbd}(SQ_L)$ on $v_j$ */
16  |   $stack$.Push($SQ'_L$);
17  |   CMDRec($v_j, SQ''_L, stack$);
18  |   $stack$.Pop();
```

The recursion will return when SQ contains all the elements in Q or SQ contains elements in X (line 3), where X denotes all the triple patterns used for extending SQ (line 18).

Theorem 1: Algorithm 2 generates all *cbd*s of a query on a particular join variable once and only once. □

Enumeration of connected multi-division. Algorithm 3 is a recursive algorithm that enumerates all *cmd*s. Similar to Algorithm 2, the Emit function (line 7) suspends the execution and maintains the current running state. The execution will be restarted from the maintained state at the next invocation. Given a query Q, the algorithm enumerates *cmd*s for each join variable in $J(Q)$. Initially, subquery SQ_L contains all the triple patterns in Q. The core idea of this algorithm is to recursively generate a connected binary-division of SQ_L denoted as (SQ'_L, SQ''_L, v_j). SQ'_L is pushed into a *stack* and all its triple patterns are removed from SQ_L. After that, all the subqueries in *stack* and SQ''_L and v_j form a *cmd*. By doing this recursively, we can produce all the *cmd*s.

Precisely, for each join variable v_j in $J(Q)$, we initialize *stack* as empty and SQ_L as containing all the triple patterns in Q (line 2). Then the function CMDRec is invoked recursively. At each recursive invocation, if *stack* is not empty, the algorithm emits a *cmd* consisting of SQ_L and all the subqueries in *stack* (line 4–6). Then the algorithm checks if subquery SQ_L has *cbd*s. Since SQ_L is connected, it only needs to check if SQ_L contains more than one triple pattern that is the neighbor of v_j (line 8–9). If SQ_L contains only one such triple pattern, then the recursion will stop. Otherwise, the algorithm uses the function ConnBinDivision to generate the *cbd*s for subquery SQ_L. For each *cbd*, denoted by (SQ'_L, SQ''_L, v_j), the algorithm pushes SQ'_L into the stack (line 12) and recursively invoke itself with suqbuery SQ''_L (line 13).

Theorem 2: Algorithm 3 generates all *cmd*s once and only once. □

D. Complexity Analysis

To analyze the complexity of Algorithm 1, we first introduce an important notation. Let Q be the input query, $T(Q)$ denotes the number of *cmd*s of all the subqueries of Q. It is formally defined as follows,

$$T(Q) = \sum\nolimits_{SQ_i \in \mathbb{S}} |D_{cmd}(SQ_i)| \tag{5}$$

where $\mathbb{S}$ denotes the set of all the connected subqueries of Q.

In Algorithm 1, function `ConnMultiDivision` is invoked as many times as the number of all the connected subqueries of the input query (recall Theorem 2) . Hence, the complexity of Algorithm 1 is equal to the number of *cmd*s of all the subqueries (i.e. $T(Q)$) multiplied by the amortized complexity of generating each *cmd* and the complexity of local query checking. The complexity of local query checking is discussed in Section III-B and [18]. We focus on the other two parts here.

The Amortized Complexity. In fact, the amortized complexity per enumerated join operator of Algorithm 1 is the cost of enumerating each *cmd* in Algorithm 3. Hence, we have:

Lemma 3: Given a join graph $J(Q) = (V_T, V_J, E_J)$, the worst-case amortized complexity per enumerated *cmd* of Algorithm 3 is $\Theta(|V_T|)$. □

In other words, Algorithm 3 has a linear amortized complexity per enumerated *cmd*.

Measurement of $T(Q)$. Different query structures might lead to very different values of $T(Q)$. So in the following we discuss a number of cases individually.

The *worst case for* $T(Q)$ depends on the number of connected subqueries and the number of connected multi-divisions of each connected subquery. The worst case w.r.t. the number of connected subqueries is $2^n - 1$, which occurs when each subset of the input query is a connected subquery. The worst cast w.r.t. the number of connected multi-division of each subquery happens when every multi-division is a connected multi-division. The number of the multi-divisions of a set can be counted by the *Bell Number*. The Bell Number of a set with k elements is denoted by B_k. Thus in the worst cast, the number of connected multi-division is $B_k - 1$.

These two worst cases might appear simultaneously with a star query. For a star query Q with n triple patterns, since each subquery of a star query is also a star query, Equation (5) can be transformed into:

$$T(Q_{star}) = \sum_{k=2}^{n} D(k) \cdot N(k) \tag{6}$$

where $D(k)$ denotes the number of subgraphs containing k triple patterns, and $N(k)$ denotes the number of connected multi-divisions of the subgraphs containing k triples patterns. Thus, $T(Q_{star})$ can be calculated by :

$$T(Q_{star}) = \sum_{k=2}^{n} (B_k - 1)C_n^k \tag{7}$$

We also analyze some *special cases for* $T(Q)$. All the connected subqueries of a chain query are chain queries as well. Let Q_{chain} be a chain query with n triple patterns. For a subquery with k triple patterns, $D(k) = n - k + 1$ and $N(k) = k - 1$. Then, $T(Q_{chain})$ can be calculated by :

$$T(Q_{chain}) = \sum_{k=2}^{n} (n - k + 1) \cdot (k - 1) = \frac{n^3 - n}{6} \tag{8}$$

For a cycle query with n triple patterns, denoted by Q_{cycle}, if a subquery has $k \neq n$ triple patterns, then it is a chain query. So we have $D(k) = n$, $N(k) = k - 1$. Otherwise, it is a cycle query, thus $D(k) = 1$ and $N(k) = k(k - 1)$. Therefore,

$$T(Q_{cycle}) = \sum_{k=2}^{n-1} n \cdot (k - 1) + n(n - 1) = \frac{n^3 - n^2}{2} \tag{9}$$

Summary. As aforementioned, putting all the three parts together, including the complexity of local query checking, the amortized complexity and the measurement of $T(Q)$, we have:

Theorem 3: Given a query Q and its query graph $G_Q = (V_Q, E_Q)$ and join graph $J(Q) = (V_T, V_J, E_J)$, the worst-case complexity of Algorithm 1 is $\Theta(|V_Q| \cdot |V_T| \cdot T(Q))$. □

Accordingly, the complexity of Algorithm 1 mainly depends on $T(Q)$. Based on Equation (7) and Equation (5), we find that the complexity of Algorithm 1 increases along with two factors: the number of triple patterns in the query and the degree of the join variables. If a join variable has a high degree or the average degree of all the join variables is high, then the number of *cmd*s on these join variables are large and hence the value of $T(Q)$ would be large.

IV. Heuristics

Since the query optimization problem is NP-hard, it is too costly to enumerate all the possible plans of a complex query. Therefore, for complicated queries, the state-of-the-art optimizers adopt heuristics to limit the number of plans explored by the enumeration algorithm. For example, TriAD [8] adopt heuristics that only consider binary join operators. However, as shown in previous work, such as [6], [10], [22], multi-way joins may be significantly more efficient than multiple binary joins in modern massively parallel computation systems, such as MapReduce-like systems, where intermediate data have to be reshuffled over the cluster. MSC [6] intends to find the so-called flat plan, which has the minimum number of levels, to reduce the cost of data reshuffling. However, as shown in our experiments in Section V, the flattest plan is not always the best plan. DP-Bushy [7] adopts a recursive top-down dynamic programming algorithm to enumerate bushy plans. On each recursive call, it considers all the possible binary joins and the multi-way join that can join the maximal number of inputs. In comparing to MSC, DP-Bushy can find good query plans that are not the flattest. But it only considers very limited possible multi-way joins, i.e. those that can join the most inputs at each plan level. Figure 3 illustrates typical plans that could be generated by these optimizers for the query in Figure 1.

We argue that due to the inefficiency of the plan enumeration algorithms adopted in the aforementioned optimizers, they consider unnecessarily limited plan spaces that may miss many good query plans. Given our efficient plan enumeration algorithm presented in the previous section, we intend to develop heuristics that consider a plan space as large as possible. Unlike the previous work, which designs heuristics without correlating them to the complexity of the enumeration algorithms, we carefully analyze the factors that affect the complexity of the enumeration algorithm and avoid over-pruning as much as possible. This is based on our extensive complexity analysis presented above.

As analyzed in Section III-D, there are two main factors that determine the complexity of the algorithm: (1) the degree of a join variable and (2) the total number of triple patterns. We present two new heuristic algorithms that focus on these factors, respectively, to limit the plans considered by the enumeration algorithm. Finally, as these two algorithms are suitable for different types of queries, we introduce an autonomous algorithm, which can analyze the structure of the input query and choose the suitable approach to optimize it.

A. ***TD-CMDP***: *Connected Multi-Division with Pruning*

As shown in Section III-D, a join variable with a high degree would dramatically increase the number of connected multi-divisions, and consequently increase the complexity. To reduce the algorithm running time, we should prune the less promising portions of the search space.

Initially, we introduce a variant of connected multi-division, namely connected complete-multi-division ($ccmd$ for short). Suppose $(SQ_1, ..., SQ_k, v_j)$ is a cmd, if $\forall SQ_i$, there exists only one $tp \in SQ_i$ s.t. $tp \in N_{tp}(v_j)$, then it is a $ccmd$.

Rule 1: For a k-way join ($k > 2$), we only take the plans constructed by $ccmd$ into account.

This heuristic is based on the observation that for all k-way join plans ($k > 2$) using repartition join operators, using $ccmd$s can construct a plan with fewer levels, which may incur less reshuffling of intermediate data. This heuristic might dramatically reduce the search space for queries that has a join variable with a very high degree in the join graph.

Rule 2: For all broadcast join plans, we only consider binary broadcast joins.

Among all the k-way broadcast joins, binary broadcast join is a good choice in many cases, since it only needs to broadcast one input, instead of $k-1$ inputs.

Rule 3: If a subquery is a local query, then the best plan for this subquery is to join all of the triple patterns using the local join algorithm.

Based on this heuristic, in Algorithm 1 (line 10), if Q is a local query, the best plan is to use a local join, i.e., $BestPlan[Q]$ is the local join plan and the function `BestPlanGen` will directly return the local join plan. So the procedure from line 11 to line 19 will be skipped, including function `ConnMultiDivision`.

Eventually, we combine these pruning rules into function `ConnMultiDivitionPruning`, which only outputs the cbds and $ccmd$s. In Algorithm 1, we use this function to replace `ConnMultiDivision`. We refer to the original Algorithm 1 and this variant as **TD-CMD** and **TD-CMDP** respectively.

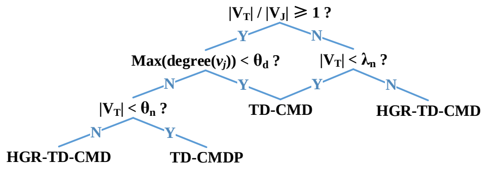


Fig. 5. Decision Tree for Autonomous Algorithm

Note that TD-CMDP is very different from the heuristic of choosing the flattest plan as adopted in MSC [6]. For each enumerated subset of triple patterns, say with size $m > 2$, if they cannot be joined by a local query, TD-CMDP considers an m-way join (if possible) as well as all the possible binary broadcast joins, and then it continues to enumerate the join plans for the remaining $m-1$ triple patterns. On the contrary, MSC would consider a k-way join with the maximum possible k at each step. Therefore, TD-CMDP considers a lot of binary join and k-way join operators that are not considered by MSC.

B. ***HGR-TD-CMD***: *Heuristic-based Join Graph Reduction*

As discussed in the previous section, the pruning algorithm can decrease the complexity for queries that have a join variable with a high degree in the join graph. However, according to Section III-D, if the total number of triple patterns is large, it will become the key factor of the complexity of the optimization algorithm. Hence, in this section, we propose a new algorithm aiming at large SPARQL queries, i.e. queries containing a large number of triple patterns. To reduce the complexity for generating the optimal query plan, the first intuition is to reduce the size of the query. If the input query can be reduced to a moderate size, then we can use the aforementioned enumeration algorithm to generate the best plan on the reduced query.

To realize this idea, we need to design an algorithm to reduce the input join graph. Here, we use the following heuristic. We find the triple patterns that can be processed by a local join and collapse them into one vertex in the join graph. Thus, we need to solve a join graph reduction problem defined as follows.

Definition 4: (Join Graph Reduction (JGR) Problem) Let $J(Q)$ be a join graph, the join graph reduction problem is to construct a new graph $J'(Q)$ that satisfies the following conditions: (1) each vertex in V'_T in $J'(Q)$ must be a local query, and (2) the sum of the cardinalities of each $v \in V'_T$ in $J'(Q)$ is minimized.

Theorem 4: The JGR problem is NP-Hard. $\square$

To solve the JGR problem efficiently, we use a greedy weighted set cover algorithm, which can achieve an approximation ratio of $\ln n$, where n is the number of all candidate sets. Specifically, given a query Q, we first compute all its maximal local queries and then put all the subqueries of each maximal local query, i.e., all the local queries of Q, into a set C. Each element SQ in C is weighted by its cardinality. Then the algorithm runs as follows. Initially, let E be an empty set. If Q has uncovered elements, the algorithm picks an element SQ in C with the minimal value of $\frac{|SQ \cap Q|}{card(SQ)}$, and adds it to E. This continues until E is a cover of Q.

After reducing the join graph, we run the top-down join enumeration algorithm on the reduced join graph to generate

TABLE II. PARAMETERS OF THE COST MODEL IN TABLE I

Parameters	α	β_B	β_R	λ_L	λ_B	λ_R
Values	0.02	0.05	0.1	0.004	0.008	0.005

the optimized query plan. Note that, the degrees of join variables will also be reduced along with join graph reduction. We call this variant as **HGR-TD-CMD**.

C. **TD-Auto:** *Autonomous Algorithm*

Putting the above three variants together, we present the autonomous algorithm, called **TD-Auto**, which is to choose the right algorithm for a given query. Based on the two factors that have great impacts on the algorithm complexity, we summarize the characteristics of these algorithms into a decision tree to construct the autonomous algorithm, as shown in Figure 5.

Concretely, for a join graph $J(Q)$, if the value of $\frac{|V_T|}{|V_J|}$ is greater than or equal to 1, then the join graph is an acyclic graph or a graph with exactly one cycle. Among these queries, if the join variables have low degrees, such as chain and cycle queries, TD-CMD is very efficient. If the maximum degree of the join variables is larger than a threshold θ_d, then we have to choose the algorithm HGR-TD-CMD or TD-CMDP based on the number of triple patterns. For queries with more than one cycles, TD-CMD is only a good choice if the number of triple patterns is less than λ_n. Otherwise HGR-TD-CMD is chosen for larger queries. In practice, based on our experiments, we set $\theta_d = 5$, $\theta_n = 30$ and $\lambda_n = 14$.

V. EXPERIMENTAL EVALUATION

A. Experimental Setup

System. We use a cluster of 10 computing nodes for the experiments. Each node in the cluster has 64GB RAM, two 2.4GHz Intel Xeon E5-2670 CPUs, each with 8 cores and 16 threads, and 1Gb network adapter. The OS is RHEL 6.2. All nodes are used for query processing while only one node is used for query optimization. We make use of RDF-3X [23] and Hadoop to build the prototype system. Each node is powered by RDF-3X version 0.3.5 to perform matchings of individual triple patterns as well as local joins. The distributed joins (broadcast join and repartition join) are implemented on Hadoop 1.2 running on Java 1.6, following the implementation of broadcast join and improved repartition join proposed in [20].

Query optimization algorithms. We compare our algorithms with the state-of-the-art algorithms of optimizing multi-way bushy plans proposed in [6] and [7]. They are denoted by **MSC** and **DP-Bushy** respectively. We use the implementations of **MSC** and **DP-Bushy** algorithms provided by the authors, which are written in Java. To make a fair comparison, we also implement all our algorithms in Java without using any special external library. We do not compare with the optimizer of TriAD because TriAD only considers binary join query plans and [6] has shown that the **MSC** plans outperform binary join query plans. All the algorithms use the same cost model. The parameters of the cost model listed in Table I are given in Table II. They are obtained by our experiments.

Workloads. We use two datasets, LUBM and UniProt. LUBM is a benchmark generator. We generate an LUBM dataset, LUBM-10000, with 1.38 billion of triples. UniProt

TABLE III. QUERIES

Query Type	Star		Chain		Tree							Dense			
Queries	L1	U1	L2	U2	L3	L4	L5	L6	U3	U4	U5	L7	L8	L9	L10
#Triple Patterns	2	5	2	5	4	4	8	8	11	6	5	6	6	11	12

is a real-world protein dataset with 2 billion of triples. The benchmark queries, L1 to L10 for LUBM and U1 to U5 for UniProt (see the [18] for details), come from [2], [23]–[26], covering most types of queries mentioned earlier in the paper, as shown in Table III.

To perform further analysis of the algorithms described in Section V-C, we use the stress testing in Waterloo SPARQL Diversity Test Suite (WatDiv) [27], which contains 124 structurally diverse query templates, each created by a random walk over the graph representation of the data schema and instantiated with 100 queries. The test case contains in total $12,400$ random queries. Furthermore, we have implemented a query generator that can randomly generate chain, cycle, tree and dense queries (recall Section II-B), which are not sufficiently represented in the aforementioned benchmark. The workload contains 116 queries, each with 3 different cardinalities and bindings. So in total, 348 input queries are generated for the optimization algorithms. The query size, i.e., the number of triple patterns, ranges from 2 to 30. For both the query generators, the cardinalities and the number of bindings are randomly attached with the triple patterns and variables respectively. More precisely, the cardinality of each triple pattern tp_i is a positive integer x_i randomly chosen from 1 to $1,000$ (we have also used the range between 1 to $100,000$, which does not affect any of our conclusions). The number of bindings of each variable in tp_i is a random integer from 1 to x_i. The cardinality of joins is computed based on the cost model (see the [18] for details).

Data partitioning. We consider three state-of-the-art data partitioning methods. (1) Hash partitioning with a hash function on both the subject and the object of an RDF triple. This can be used for all query optimization algorithms. (2) The semantic hash partitioning algorithm, 2-hop forward (2f for short), proposed in [3]. 2f extends each vertex in RDF graph with 2-hop forward expansion to form the partitioning element, then hashes the partitioning elements to the computing nodes. (3) The path partitioning approach in [2], called Path-BMC. This algorithm firstly decomposes the RDF graph into end-to-end paths, and then merges them in a greedy manner based on the weights of the vertices. Note that only our methods can make use of the last two data partitioning methods.

B. Query Performance Comparison

The first part of our experiments is to analyze the query performance over two datasets: LUBM-10000 and UniProt. The queries (as shown in Table III) are optimized by three query optimizers and all query plans are running on our prototype system. As shown in our previous analysis and the experimental analysis in the next section, TD-Auto is the most practical method among those we proposed. Therefore we only use TD-Auto in this section.

All numbers in Table IV and Table V are averages over at least three runs. Table IV shows the query optimization time of the algorithms. We observe that the MSC algorithm always spends more time to generate the query plans, especially for the complex queries. Table V shows the query processing

TABLE IV. QUERY OPTIMIZATION TIME (LUBM AND UNIPROT QUERIES)

Algorithm	Star		Chain		Tree							Dense			
	L1	U1	L2	U2	L3	L4	L5	L6	U3	U4	U5	L7	L8	L9	L10
TD-Auto	0.001s	0.027s	0.004s	0.010s	0.009s	0.009s	0.054s	0.066s	0.121s	0.018s	0.014s	0.043s	0.041s	0.525s	1.569s
MSC	0.005s	0.008s	0.005s	0.067s	0.013s	0.014s	0.125s	0.150s	21.628s	0.089s	0.020s	0.124s	0.135s	432.429s	>36000s
DP-Bushy	0.004s	0.002s	0.005s	0.013s	0.010s	0.005s	0.048s	0.056s	0.026s	0.009s	0.004s	0.013s	0.011s	0.147s	0.331s

TABLE V. QUERY PROCESSING TIME (LUBM AND UNIPROT QUERIES)

Data Partitioning	Algorithm	Star		Chain		Tree							Dense			
		L1	U1	L2	U2	L3	L4	L5	L6	U3	U4	U5	L7	L8	L9	L10
Hash-SO	TD-Auto	1.25s	3.17s	1.04s	176.23s	131.77s	14.65s	34.32s	305.19s	325.81s	73.93s	60.91s	76.63s	172.34s	372.42s	389.15s
	MSC	1.33s	3.21s	1.11s	409.18s	176.53s	38.17s	55.18s	471.51s	523.09s	197.66s	71.59s	104.37s	1647.24s	731.09s	N/A
	DP-Bushy	1.67s	3.16s	1.20s	287.75s	272.52s	29.94s	110.50s	397.76s	405.57s	215.42s	58.77s	125.33s	201.73s	756.16s	492.36s
2f	TD-Auto	1.26s	3.20s	1.10s	405.26s	134.65s	2.99s	119.63s	241.21s	289.11s	97.29s	7.29s	30.78s	29.09s	149.69s	170.31s
Path-BMC	TD-Auto	1.25s	3.11s	1.05s	5.09s	1.01s	1.13s	1.14s	1.23s	19.03s	14.38s	7.11s	23.54s	20.33s	3.54s	4.48s

TABLE VI. THE ESTIMATED COST OF THE QUERY PLANS GENERATED BY THE QUERY OPTIMIZATION ALGORITHMS

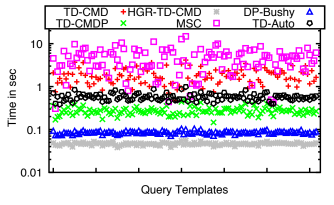
Algorithm	Star		Chain		Tree							Dense			
	L1	U1	L2	U2	L3	L4	L5	L6	U3	U4	U5	L7	L8	L9	L10
TD-Auto	3.12E4	3.57E5	7.52E4	1.98E6	3.39E6	9.78E4	3.03E6	4.56E6	9.28E5	7.18E5	8.26E5	1.84E6	1.71E7	8.67E6	9.79E6
MSC	3.12E4	3.57E5	7.52E4	9.35E6	6.02E6	2.76E5	3.81E6	6.15E6	8.14E6	3.54E6	8.89E5	4.42E6	2.39E7	3.18E7	N/A
DP-Bushy	3.12E4	3.57E5	7.52E4	3.91E6	3.45E6	2.75E5	3.06E6	6.69E6	1.13E6	1.17E6	1.45E6	4.41E6	2.11E7	1.29E7	9.81E6

time. The general observation is that the plans of TD-Auto outperform the best plans generated by MSC and DP-Bushy on the chain, tree and dense benchmark queries. Moreover, TD-Auto can leverage more data partitioning algorithms, such as 2f and Path-BMC, and hence the performance improvement can be more significant. For example, the TD-Auto plans using Path-BMC partitioning can outperform the MSC and DP-Bushy plans by an order of magnitude, since all of the queries are local queries with the Path-BMC partitioning. While Path-BMC has a superior query performance, it is less flexible in dealing with dynamic RDF datasets than simple hash-based methods. TD-Auto can be employed in a general RDF engine that can adapt its partitioning methods according to the application's requirements.

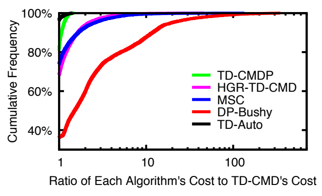
For L1, U1 and L2, because hash partitioning will collocate all the triples sharing the same subjects and objects, all the queries can be evaluated with local joins. So the query performance are similar for all the algorithms. For the long chain query U2, the TD-Auto plan is faster than DP-bushy plan and MSC plan.

The search space of the DP-Bushy algorithm for the complex queries, such as tree queries and dense queries, is much smaller than that of TD-Auto (see Section V-C for details). The limited search space causes DP-Bushy to miss the optimal plans. Therefore, for tree queries L3, L4, L5 and U4, the TD-Auto plans are faster than the best DP-Bushy plans by a factor of about 2. For L5, TD-Auto with 2f achieves a poor performance. This is because, based on 2f, the query decomposition result is not efficient, which leads to too much network cost. For L6, U3 and U4, repartition joins have very high cost due to the massive reshuffling of intermediate results. However, MSC generates flat plans, which cannot take advantage of broadcast joins, hence it has low performance.

For the dense queries, the MSC plans are not efficient. The MSC plan for L8 generates a huge amount of intermediate results, which incurs a large reshuffling cost. MSC runs 432 seconds to generate the plan for L9, which is more than the total execution time of TD-Auto. For L10, MSC cannot even finish within 10 hours. In addition, the DP-Bushy plans are significantly slower than the TD-Auto plans, due to the limited plan space of DP-Bushy.



(a) Optimization Time



(b) Cumulative Frequency

Fig. 6. Waterloo SPARQL Diversity Test Suite (WatDiv)

C. Query Optimization Algorithms Analysis

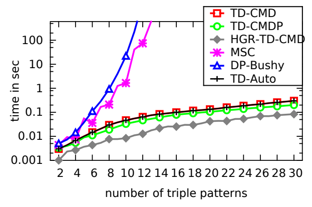
In this section, we present a thorough study of the algorithms. We compare the algorithms on three parameters, the optimization time, the size of the search space (i.e. the number of enumerated plans) and the cost of the best plan generated. For this purpose, we use hash partitioning, WatDiv as well as our own query generator, which can produce a large number of random queries for us to collect meaningful statistics.

Here we use the cost model proposed in Section II-E to compare the plans generated by different algorithms. To verify the validity of such comparisons, in Table VI, we present the cost of the plans generated for the benchmark queries used in the previous subsection. As shown in Table IV and Table VI, most of the plans with the minimal estimated cost also have the lowest query processing time. That implies our cost model provides a good indication of the general quality of the plans.

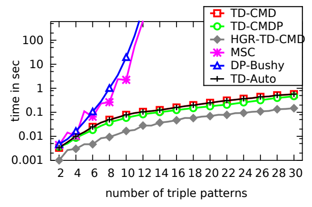
Figure 6a and 7 report the optimization time. In Figure 6a, the x-axis is the 124 random query template in WatDiv. For each template, the average optimization time of the 100 randomly generated queries is reported. Most query templates in WatDiv are star queries or joins of a few star queries, so this workload contains very few chain queries or no cycle queries. Therefore, we use our generator to produce these types of queries, whose results are presented in Figure 7a– 7d. Here, all the reported figures are the averages over three runs of the same queries. Table VII shows the number of plans enumerated by the different algorithms.

TABLE VII. SIZE OF SEARCH SPACE

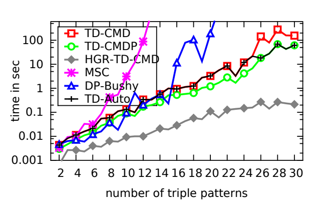
	Chain			Cycle			Tree			Dense		
#Triple Patterns	8	16	30	8	16	30	8	16	30	8	16	30
MSC	1	N/A	N/A	4	N/A	N/A	3	N/A	N/A	18	N/A	N/A
DP-Bushy	80	N/A	N/A	192	N/A	N/A	20	199	N/A	32	329	1,266
TD-CMD	84	680	4,495	224	1,920	13,050	342	182,490	75,256,333	6,715	N/A	N/A
TD-CMDP	84	680	4,495	224	1,920	13,050	269	99,823	46,828,168	4,577	66,639,152	N/A
HGR-TD-CMD	20	120	969	24	605	4,410	10	241	8,010	20	1,384	22,344
TD-Auto	84	680	4,495	224	1,920	13,050	342	182,490	46,828,168	6,715	1,384	22,344



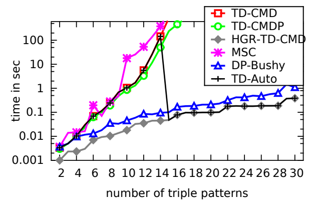
(a) Chain



(b) Cycle

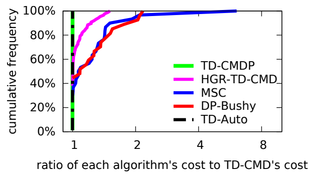


(c) Tree

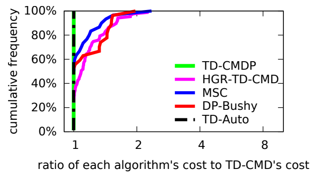


(d) Dense

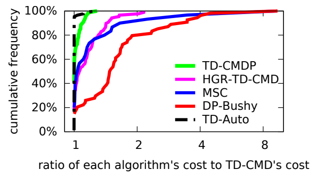
Fig. 7. Optimization time of queries from our query generator.



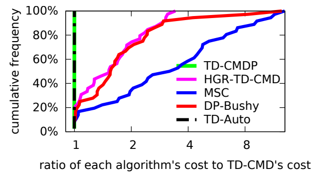
(a) Chain



(b) Cycle



(c) Tree



(d) Dense

Fig. 8. Cumulative frequency distribution of the cost of the queries from our query generator.

In Figure 6b and Figure 8, we present the costs of the query plans produced by the algorithms using a cumulative frequency analysis. Since TD-CMD always select the optimal plan, for the ease of comparison, we use the cost of the query plan generated by TD-CMD to normalize the cost of the plans generated by the other algorithms for the same query. We only include the plans that can be generated within 600 seconds by the algorithms. Then we draw the cumulative frequency distribution of these normalized values. The x-axis indicates the ratio of the cost of the query plan generated by an algorithm to the cost of the plan generated by TD-CMD.

TD-CMD is very efficient for chain, cycle and tree queries, whose join variables have moderate degrees in the join graph. On the contrary, for the queries, whose join variables have high degrees, such as star and dense queries, TD-CMD has a high cost and it can complete within 600 seconds only for queries with a dozen triple patterns. However, one can see from Table VII that its search space is much larger than all the other algorithms.

TD-CMDP prunes a large number of cmds, especially for star, tree and dense queries, resulting in a great performance improvement. As shown in Figure 6a and 7, it is usually faster than TD-CMD by a factor of 2-5 for large tree and dense queries. Moreover, due to the carefully selected pruning strategies, the costs of the plans produced by TD-CMDP are very close to those produced by TD-CMD, as shown in Figure 6b and 8.

HGR-TD-CMD significantly reduces the optimization time especially for large queries and as shown in Figure 6b and 8, the costs of the plans it produced are not too far-off from those produced by TD-CMD. Thus, it is very effective when the query size is large. Finally, by autonomously choosing the suitable algorithms for different queries, TD-Auto achieves a good trade-off between optimization time and plan cost.

MSC runs an expensive minimum set cover algorithm for each level. Therefore, its running time increases exponentially with the number of triple patterns for all types of queries. Moreover, since it focuses on finding flat plans, its search space is limited. As shown in Figure 8, only less than 50% of the plans generated by MSC have costs comparable to those produced by TD-CMD.

DP-Bushy has a high efficiency with star queries but not with many other queries. This is because it cannot prevent enumerating plans with Cartesian products but rather eliminating them when they are formed. It is not a problem for star queries because any subquery of a star query is connected. DP-Bushy runs fast with dense queries, because it explores a limited portion of the plan space. However, as shown in Figure 8d, 90% of its plans are more expensive than the plans generated by TD-CMD.

VI. Related Work

SPARQL query optimization is one of the most challenging problems in SPARQL query processing, which can have a dramatic impact on query performance [23]. Query optimization has been well studied in a centralized context [17], [23], [25], [26], [28], [29].However, the centralized query optimization algorithms do not take into account the RDF data partitioning and the communication overhead.

Early work on query optimization in traditional distributed and parallel databases mainly focuses on generating binary bushy plans [30]–[33]. Recently, researchers attempt to leverage MapReduce systems to process queries over massive datasets [34]. Many recent efforts are devoted to designing efficient join algorithms in MapReduce [20], [22], [35]. A few works investigate query optimization in MapReduce-like frameworks [36]–[38], which consider the left-deep plans and binary bushy plans. However, as shown in various studies [6], [10], [22], multi-way joins are significantly more efficient than multiple binary joins in MapReduce-like systems.

There are more query optimization algorithms proposed for distributed and parallel RDF engines. In [10], [39], the authors try to build flat query plans in MapReduce. However, in comparing to MSC, they cannot guarantee to find the flattest plan. DREAM [9] considers multi-way joins at the first plan level but uses only binary joins at the other levels. In [5], [40], a SPARQL query is evaluated in a graph processing system, such as Pregel [41] and Trinity [42]. Therefore, their query optimization problem is fundamentally different from ours, which assumes queries are evaluated by join operators.

To minimize communication overhead, [2]–[5], [43] focus on designing data partitioning methods to maximize the possibility of using local joins. The query optimization algorithms in these papers focus on minimizing the number of distributed joins based on their data partitioning algorithms. Our generic data partitioning model can accommodate all these methods. Another research direction of RDF data partitioning is dynamic run-time data partitioning, which redistributes the RDF data according to the run-time changes of system workload or the hot queries in the workload [5], [44], [45]. We discuss how to how to accommodate these dynamic methods in [18].

Previous researches [14]–[16] have studied dividing a join graph into two connected components. However, these algorithms are not efficient for enumerating connected binary divisions. The reasons are two-fold. First, not all divisions with two connected components satisfy the definition of connected binary-division, hence these algorithms would enumerate useless divisions. Second, checking if a division satisfies the conditions of connected binary-division increases the complexity.

VII. Conclusion

In this paper, we analyze the pros and cons of the state-of-the-art SPARQL query optimization algorithms. We show that a generic data partitioning model and an efficient algorithm of checking local queries allows an optimizer to easily adapt to different state-of-the-art data partitioning methods. Furthermore, a carefully designed top-down join enumeration algorithm can generate multi-way bushy plans with linear amortized complexity. A more efficient enumeration algorithm allows the optimizer to explore a larger plan space within the same time and hence can achieve a better optimization result. Finally, heuristics to reduce the degrees of join variables and the number of triple patterns in join graphs are effective to reduce the problem complexity while maintaining good optimization results.

VIII. Acknowledgment

The authors from HUST are supported by NSFC under grant No. 61433019 and No. U1435217.

References

[1] *Linking Open Data*, http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData.

[2] B. Wu, Y. Zhou, P. Yuan, L. Liu, and H. Jin, "Scalable sparql querying using path partitioning," in *ICDE*, 2015.

[3] K. Lee and L. Liu, "Scaling queries over big rdf graphs with semantic hash partitioning," in *VLDB*, 2014.

[4] J. Huang, D. Abadi, and K. Ren, "Scalable SPARQL querying of large RDF graphs," *PVLDB*, vol. 4, no. 11, pp. 1123–1134.

[5] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *SIGMOD*, 2012, pp. 517–528.

[6] F. Goasdoué, Kaoudi *et al.*, "Cliquesquare: Flat plans for massively parallel rdf queries," in *ICDE*, 2015.

[7] J. Huang, K. Venkatraman, and D. J. Abadi, "Query optimization of distributed pattern matching," in *ICDE*, 2014, pp. 64–75.

[8] S. Gurajada, S. Seufert *et al.*, "Triad: A distributed shared-nothing rdf engine based on asynchronous message passing," in *SIGMOD*, 2014.

[9] M. Hammoud, Rabbou *et al.*, "Dream: Distributed rdf engine with adaptive query planner and minimal communication," in *VLDB*, 2015.

[10] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham, "Heuristics based query processing for large RDF graphs using cloud computing," *IEEE TKDE*, vol. 23, no. 9, pp. 1312–1327, 2011.

[11] X. Zhang, L. Chen *et al.*, "Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud," in *ICDE*, 2013.

[12] Z. Kaoudi and I. Manolescu, "Rdf in the clouds: A survey," *The VLDB Journal*, vol. 24, no. 1, pp. 67–91, 2015.

[13] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products," in *VLDB*, 2006, pp. 930–941.

[14] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration," in *SIGMOD*, 2007.

[15] P. Fender and G. Moerkotte, "A new, highly efficient, and easy to implement top-down join enumeration algorithm," in *ICDE*, 2011.

[16] P. Fender, G. Moerkotte *et al.*, "Effective and robust pruning for top-down join enumeration algorithms," in *ICDE*, 2012.

[17] A. Gubichev and T. Neumann, "Exploiting the query structure for efficient join ordering in sparql queries." in *EDBT*, 2014.

[18] B. Wu, Y. Zhou, H. Jin, and A. Deshpande, "Sparql query optimization in mapreduce," Tech. Rep. [Online]. Available: http://imada.sdu.dk/~zhou/papers/sparql.pdf

[19] *METIS*, http://www.cs.umn.edu/~metis/.

[20] S. Blanas, J. M. Patel *et al.*, "A comparison of join algorithms for log processing in mapreduce," in *SIGMOD*, 2010, pp. 975–986.

[21] R. S. Xin, J. Rosen *et al.*, "Shark: Sql and rich analytics at scale," in *SIGMOD*, 2013, pp. 13–24.

[22] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *EDBT*, 2010, pp. 99–110.

[23] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *The VLDB Journal*, 2010.

[24] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics*, 2005.

[25] P. Yuan, P. Liu *et al.*, "TripleBit: a fast and compact system for large scale RDF data," *PVLDB*, vol. 6, no. 7, 2013.

[26] M. Atre, V. Chaoji *et al.*, "Matrix bit loaded: a scalable lightweight join query processor for RDF data," in *WWW*, 2010, pp. 41–50.

[27] G. Aluç, O. Hartig *et al.*, "Diversified stress testing of RDF data management systems," in *ISWC*, 2014, pp. 197–212.

[28] P. Tsialiamanis, L. Sidirourgos *et al.*, "Heuristics-based query optimisation for sparql," in *EDBT*, 2012.

[29] J. Kim, H. Shin *et al.*, "Taming subgraph isomorphism for RDF query processing," *PVLDB*, vol. 8, no. 11, pp. 1238–1249, 2015.

[30] M. Spiliopoulou, M. Hatzopoulos *et al.*, "Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline," *TKDE*, vol. 8, no. 3, pp. 429–445, 1996.

[31] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl, "Parallelizing query optimization," *PVLDB*, vol. 1, no. 1, pp. 188–200, 2008.

[32] D. Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys*, vol. 32, no. 4, pp. 422–469, 2000.

[33] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query optimization for parallel execution," in *SIGMOD*, 1992.

[34] F. Li, B. C. Ooi *et al.*, "Distributed data management using mapreduce," *ACM Computing Surveys*, vol. 46, no. 3, p. 31, 2014.

[35] S. Chu, M. Balazinska, and D. Suciu, "From theory to practice: Efficient join query evaluation in a parallel database system," in *SIGMOD*, 2015.

[36] A. Thusoo, J. Sarma *et al.*, "Hive-a petabyte scale data warehouse using hadoop," in *ICDE*, 2010, pp. 996–1005.

[37] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, "Query optimization for massively parallel data processing," in *SoCC*, 2011.

[38] A. Schätzle, M. Przyjaciel-Zablocki *et al.*, "S2RDF: RDF querying with SPARQL on spark," *PVLDB*, vol. 9, no. 10, pp. 804–815, 2016.

[39] P. Ravindra, S. Hong *et al.*, "Efficient processing of RDF graph pattern matching on MapReduce platforms," in *DataCloud-SC*, 2011.

[40] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," in *VLDB*, 2013, pp. 265–276.

[41] G. Malewicz, M. H. Austern *et al.*, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[42] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *SIGMOD*, 2013.

[43] B. Wu, Y. Zhou, P. Yuan, H. Jin, and L. Liu, "Semstore: A semantic-preserving distributed rdf triple store," in *CIKM*, 2014.

[44] T. Yang, J. Chen *et al.*, “Efficient sparql query evaluation via automatic data partitioning,” in *DASFAA*, 2013.

[45] R. Al-Harbi, I. Abdelaziz *et al.*, “Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning,” *VLDB J.*, vol. 25, no. 3, pp. 355–380, 2016.

APPENDIX

A. Checking Local Queries

In the RDF data partitioning literature [2]–[4], various algorithms are proposed to decompose a query into a set of local queries. The idea is to check the query based on the structures of their partitioning elements. For example, both MSC [6] and DP-Bushy [7] inherently assume data are hash partitioned. To check if a subquery SQ is a local query, one can see if there exists a vertex $v_i \in SQ$ such that v_i exists in all the triple patterns in SQ. In other words, if all the triple patterns in SQ share a common vertex, then it is a local query, because triples with the same values on the join variable would be hashed to the same node.

Basically, the algorithm has to enumerate every vertex in SQ and see if it is shared by all the triple patterns. The complexity of such an algorithm is $\Theta(|V_Q|\cdot|E_Q|)$. The function `isLocalQuery` has to be invoked as many time as the number of connected subquery of Q in Algorithm 1, which could be very large. Furthermore, the existing algorithms can only be designed for a particular data partitioning method and cannot be readily applied to our generic partitioning model.

We aim to design an algorithm that is not only more efficient but also can accommodate different data partitioning algorithms. Our approach has a complexity as $\Theta(|V_Q|)$ and uses the following concept:

Definition 5: (Maximal Local Query) Given a query Q and a vertex v in the query graph G_Q, the maximal local query at v is a local query that has the most number of triple patterns in Q containing v, denoted by $MLQ_v(Q)$.

Lemma 4: If Q is a local query, then all subqueries of Q are local queries. ☐

Theorem 5: For a query Q, a subquery SQ of Q is a local query, if and only if there exists a vertex $v \in V_Q$ such that SQ is a subquery of the maximal local query $MLQ_v(Q)$. ☐

The idea of our approach is to apply the *combine* function in our generic data partitioning model in Section II-C on the query graph G_Q. For each $v \in V_Q$, we set the maximal local query at v as $MLQ_v(Q) \leftarrow combine(v, G_Q)$. The reason that $combine(v, G_Q)$ is a local query is that if there is a match of $combine(v, G_Q)$ in G_R, then there must exist a match of v in G_R, denoted as $\mu(v)$, such that the match of $combine(v, G_Q)$ is a subgraph of $combine(\mu(v), G_R)$. Note that $combine(\mu(v), G_R)$ is the partition element $e_{\mu(v)}$ anchored at $\mu(v)$, therefore any subgraph of $combine(\mu(v),\ G_R)$ must be accessible on a single node and $combine(v, G_Q)$ can be evaluated locally on each node. Furthermore, $combine(v, G_Q)$ is the maximal local query at v that we can deduce from the partitioning model and $combine(v, G_Q)$ covers all the local queries that can be detected in all the existing static partitioning methods [2]–[5].

Accordingly, to check whether a subquery SQ can be processed locally, we should first generate a maximal local query for each vertex and then check if SQ is a subquery of one of the maximal local queries MLQ_v. While the general containment problem of conjunctive queries is NP-complete, here we can simply check if SQ is a subquery of MLQ_v by checking if MLQ_v contains all the triple patterns of SQ because both SQ and MLQ_v are subqueries of Q and are connected (by Definition 3 and 5). Recall that each subquery is encoded as a bitset. Therefore we can easily check if SQ is a subquery of MLQ_v using bit operations. Let b_{MLQ_v} and b_{SQ} be the bitsets of MLQ_v and SQ respectively. If the equation $b_{MLQ_v}\ \&\ b_{SQ} = b_{SQ}$ is true, then SQ is a local query.

Example 7: Assume that the data partitioning algorithm is hash partitioning on both the subject and object of an RDF triple. For the query in Figure 1, by applying $combine(?a, G_Q)$, we can get the maximal local query at $?a$ as $\{tp_1, tp_2, tp_3, tp_7\}$. This is because $combine(v, G_R)$ in this partitioning algorithm returns all the triple patterns containing v. Then one can see that all the subqueries of $\{tp_1, tp_2, tp_3, tp_7\}$ are local queries, e.g. $\{tp_1, tp_2, tp_3\}$. ☐

The main advantages of this algorithm are twofold. First, the space complexity for storing the maximal local queries is quite low. This is because the number of maximal local query is less than or equal to the number of vertices in the query, denoted by $|V_Q|$. In addition, the space required for storing each maximal local query is only a bitset. Furthermore, every local join detection is actually a bitset operation, which has a low complexity as $\Theta(1)$. Since there are $|V_Q|$ maximal local queries, the worst case complexity is $\Theta(|V_Q|)$. In practice, it is even more efficient. As shown in Algorithm 1 (line 2-3), if a query is a local query, function `IsLocalQuery` does not need to be invoked for all its subqueries.

B. Cardinality Estimation

Given a triple pattern tp, its cardinality is denoted by $|tp|$. The $B(tp, v)$ indicates the number of distinct bindings of variable v in the matchings of tp. Let $var(tp_1, tp_2) = var(tp_1) \cap var(tp_2)$ denotes the shared variables between tp_1 and tp_2. Then, we have

$$|tp_1 \bowtie tp_2| = \frac{|tp_1| \cdot |tp_2|}{\prod_{\forall v_j \in var(tp_1,tp_2)} \max B(tp_i, v_j)} \tag{10}$$

According to the above equation, we can estimate the cardinality of a subquery that is the join of n triple patterns by

$$|tp_1 \bowtie \cdots \bowtie tp_n| = |(tp_1 \bowtie \cdots \bowtie tp_{n-1}) \bowtie tp_n| \tag{11}$$

The generic RDF data partitioning model can be extended to capture the dynamic RDF data partitioning methods [5], [45]. In general, the dynamic RDF data partitioning algorithms pre-partition the data by a static partitioning algorithm, such as hash partitioning. Then at run-time, they redistribute the data based on some “hot queries” such that the hot queries can be evaluated locally.

Basically, We can extend our static data partitioning model with the list of the hot queries. Since the generic data partitioning model is mainly used to generate all the maximal local queries, therefore we focus on how to capture the dynamic RDF data partitioning methods to generate the maximal local queries. When computing the maximal local queries, we can leverage both the hot-query list and the $combine$ function to generate the maximal local queries. For each vertex v in a query G_Q, we first use $combine(v, G_Q)$ function to generate a local query that anchored at v. Then we can check if there is an intersection, as a set of triple patterns, between a hot query and the query G_Q. If such an intersection exists and both the following conditions are met: (1) the triple patterns contained in the intersection is connected; (2) one of the triple patterns contains the vertex v, then the set of triple patterns in the intersection forms another local query that anchored at v. Finally, MLQ_v is set as the one of the local queries anchored at v with the maximum number of triple patterns.

C. Proof of Theorem 5

Proof: (1) "$\Rightarrow$". If SQ is a subquery of a maximal local query, then according to Lemma 4, SQ must be a local query.

(2) "$\Leftarrow$". We proof this by contradiction. Suppose there exists a subquery SQ that is a local query, but it is not a subquery of any maximal local query. Then according to the definition of maximal local query, SQ must be a maximal local query, which is a contradiction. ■

D. Proof of Lemma 1

Proof: (1) Since $\mathbb{C}_{v_j}[i] \subsetneq (Q \backslash SQ)$, there must be at least one triple pattern tp_i such that $tp_i \in N_{tp}(v_j)$ and $tp_i \in (Q \backslash SQ) \backslash \mathbb{C}_{v_j}[i]$. Otherwise, according to the definition of indivisible component, all the triple patterns in $(Q \backslash SQ) \backslash \mathbb{C}_{v_j}[i]$ should be in $\mathbb{C}_{v_j}[i]$. In addition, removing/adding a $\mathbb{C}_{v_j}[i]$ from/to a connected subquery that contains triple patterns in $N_{tp}(v_j)$ dose not affect its connectivity. Thus, $(SQ \cup \mathbb{C}_{v_j}[i], Q \backslash SQ \backslash \mathbb{C}_{v_j}[i], v_j)$ is a *cbd*.

(2) Suppose that there exists a subset $\mathscr{S}' \subsetneq \mathbb{C}_{v_j}[i]$, and $(SQ \cup \mathscr{S}', Q \backslash SQ \backslash \mathscr{S}', v_j)$ is a cbd. Then either $\mathscr{S}'$ or $\mathbb{C}_{v_j}[i] \backslash \mathscr{S}'$ contains a triple pattern tp_i in $N_{tp}(v_j)$. Suppose $\mathscr{S}'$ contains tp_i. Then $\mathbb{C}_{v_j}[i] \backslash \mathscr{S}'$ is disconnected from $Q \backslash SQ$, which is a contradiction. A contradiction can also be derived in a similar way for the case that $\mathbb{C}_{v_j}[i] \backslash \mathscr{S}'$ contains tp_i. ■

E. Proof of Lemma 2

Proof: (1) Removing $\{tp\}$ dose not make $Q \backslash SQ \backslash \{tp\}$ disconnected. Thus $(SQ \cup \{tp\}, Q \backslash SQ \backslash \{tp\}, v_j)$ is a cbd.

(2) We proof the first conclusion by contradiction. Suppose there are more than one connected components that contains the triple patterns in $N_{tp}(v_j)$. Since all of them contain the triple patterns in $N_{tp}(v_j)$, they must be connected by v_j, which is a contradiction. Then we have that $(SQ \cup \{tp\} \cup S, Q \backslash SQ \backslash \{tp\} \backslash S, v_j)$ must be a *cbd*. We proof the next conclusion by contradiction. Suppose that there exists a subset $\mathscr{S}' \subsetneq \{tp\} \cup S$, such that $(SQ \cup \mathscr{S}', Q \backslash SQ \backslash \mathscr{S}', v_j)$ is a cbd. Since $S_2, ..., S_i$ is not connected with SQ or $Q \backslash SQ \backslash \{tp\} \backslash S$, but only is connected with $\{tp\}$. Assume $\mathscr{S}'$ contains tp, then $Q \backslash SQ \backslash \mathscr{S}'$ must be disconnected, and vice versa. ■

F. Proof of Theorem 1

Before giving the proof of this theorem, we first prove the following lemma.

Lemma 5: Given a connected binary-division (SQ', SQ'', v_j), if either SQ' or SQ'' contains one and only one triple pattern in $N_{tp}(v_j)$, Algorithm 2 can enumerates this *cbd*.

Proof: If SQ' contains only one triple pattern in $N_{tp}(v_j)$, then this triple pattern must be the triple pattern used to initialize the set A in Algorithm 2 (line 8). Thus this *cbd* can be enumerated. Otherwise, since Algorithm 2 is recursively extending SQ until $Q \backslash SQ$ has only one triple pattern in $N_{tp}(v_j)$, all *cbd* (SQ', SQ'', v_j) whose SQ'' contains only one triple pattern in $N_{tp}(v_j)$ will be enumerated. ■

Then, Theorem 1 is proved as follows.

Proof: (1) We first proof that Algorithm 2 can generate all *cbd*s of a query Q on a join variable v_j by contradiction. Assume that a set of *cbd*s cannot be enumerated, denoted by $\mathbb{N}_{cbd}$. For each $cbd \in \mathbb{N}_{cbd}$, denoted by (SQ', SQ'', v_j), we denote the number of triple patterns neighboring join variable v_j in SQ' and SQ'' by $T_{v_j}(SQ')$ and $T_{v_j}(SQ'')$ respectively. Then for all $T_{v_j}(SQ')$ and $T_{v_j}(SQ'')$ of all *cbd*s in $\mathbb{N}_{cbd}$, there must be one, denoted by SQ_{min}, with the minimal $|T_{v_j}(SQ_{min})|$. We choose the $cbd \in \mathbb{N}_{cbd}$ that contains this minimal one, denoted by $(SQ_{min}, Q \backslash SQ_{min}, v_j)$ (or $(Q \backslash SQ_{min}, SQ_{min}, v_j)$). Moreover, Lemma 5 implies that $|T_{v_j}(SQ_{min})| > 1$.

Case 1: If a triple pattern $tp_{v_j} \in T_{v_j}(SQ_{min})$ is contained in an indivisible component $\mathbb{C}_{v_j}[i]$, then $SQ_{min} \backslash \mathbb{C}_{v_j}[i]$, $Q \backslash SQ_{min} \cup \mathbb{C}_{v_j}[i]$ and v_j construct a *cbd*, and $|T_{v_j}(SQ_{min} \backslash \mathbb{C}_{v_j}[i])| < |T_{v_j}(SQ_{min})|$, which is a contradiction.

Case 2: If none of the triple patterns $tp_{v_j} \in T_{v_j}(SQ_{min})$ is contained in a indivisible component, then for a triple pattern $tp_{v_j} \in T_{v_j}(SQ_{min})$, $(SQ_{min} \backslash \{tp_{v_j}\}, Q \backslash SQ_{min} \cup \{tp_{v_j}\}, v_j)$ is a *cbd* and $|T_{v_j}(SQ_{min} \backslash \{tp_{v_j}\})| < |T_{v_j}(SQ_{min})|$, which is a contradiction.

(2) Algorithm 2 is to extend the initial SQ (line 8) recursively, thus it generate all *cbd*s on the join variable v_j only once. ■

G. Proof of Theorem 2

Proof: (1) We prove this theorem by contradiction. Assume there exists one *cmd* $(S_1, S_2, ..., S_n, v_j)$ joined on join variable v_j, which cannot be found by Algorithm 3. Furthermore, we assume that at the first time of invoking the function `ConnBinDivision`, the triple pattern used to initialize A (line 8 in Algorithm 2) is in S_1. According to Theorem 1, the *cbd* $(S_1, S_2 \cup S_3 \cup ... \cup S_n, v_j)$ can be generated by Algorithm 2. Similarly, the *cbd* $(S_2, S_3 \cup ... \cup S_n, v_j)$ can be generated when processing connected binary-division on $S_2 \cup S_3 \cup ... \cup S_n$. Consequently, one can see that the connected multi-division $(S_1, S_2, ..., S_n, v_j)$ can be generated by Algorithm 3. Then a contradiction is derived.

(2) According to Theorem 1, Algorithm 3 generates all *cmd*s only once. ■

H. Proof of Lemma 3

Before giving the proof of this lemma, we first prove the following lemma.

Lemma 6: Given a join graph $J(Q) = (V_T, V_J, E_J)$, the worst-case complexity of Algorithm 2 for generating each connected binary-division is $\Theta(|V_T|)$.

Proof: In Algorithm 2, the function `Emit` outputs a *cbd* on each invocation of function `GetPartInComponent`. The worst-case complexity of `GetPartInComponent` is $\Theta(|V_T|)$. Thus, the worst-case complexity of Algorithm 2 for generating each connected binary-division is $\Theta(|V_T|)$. ■

Then, Lemma 3 is proved as follows.

Proof: In Algorithm 3, the function `Emit` outputs a *cmd* on each generated *cbd*. According to Lemma 6, the function `ConnBinDivision` has a complexity of $\Theta(|V_T|)$ per *cbd* in the worst case. Therefore, Algorithm 3 costs $\Theta(|V_T|)$ per connected multi-division in the worst case. ■

I. Proof of Theorem 4

Proof: To prove that this problem is NP-Hard, we show a reduction from the weighted set cover problem. Consider a universe set U with n element, and a set S whose elements are the subsets of U with weights. The weighted set cover problem is to find a subset of S that can cover U with the minimum weight. Then we construct a query Q satisfying the following conditions. Let $Q = \{tp_1, ..., tp_n\}$. $|S|$ can be represented by $|S| = a_k{\cdot}2^k{+}a_{k-1}{\cdot}2^{k-1}{+}...{+}a_0{\cdot}2^0$. For a term $a_i{\cdot}2^i$ in this polynomial expression that $a_i > 0$, we construct a maximal local query with i triple patterns in Q. Moreover, all the constructed maximal local queries should cover Q. The subqueries of all maximal local queries are put in C, thus $|C| = |S|$. For each subquery in C, it is assigned a cardinality identical to the weight of the corresponding element in S. Thus, selecting a set E with m elements in C (i.e., m local queries of Q), such that every triple pattern in Q is contained by at least one element in E and the sum of the cardinality of each element in E is minimized if and only if the weighted set cover problem is solved. ■

LUBM:
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/ zhp2/2004/0401/univ-bench.owl#>

L1: SELECT ?x WHERE {
?x rdf:type ub:ResearchGroup.
?x ub:subOrganizationOf <Department0.University0.edu>. }

L2: SELECT ?x ?y WHERE {
?x ub:worksFor ?y.
?y ub:subOrganizationOf <University0.edu>. }

L3: SELECT ?x ?y WHERE {
?x rdf:type ub:GraduateStudent.
<Department0.University0.edu/AssociateProfessor0>
ub:teacherOf ?y.
?y rdf:type ub:GraduateCourse.
?x ub:takesCourse ?y. }

L4: SELECT ?x ?y WHERE {
?x ub:worksFor ?y.
?y rdf:type ub:Department.
?x rdf:type ub:FullProfessor.
?y ub:subOrganizationOf <University0.edu>. }

L5: SELECT ?x ?w WHERE {
?x ub:advisor ?y.
?y ub:worksFor ?z.
?x rdf:type ub:GraduateStudent.
?z ub:subOrganizationOf ?w.
?w ub:name ?u.
?z rdf:type ub:Department.
?w rdf:type ub:University.
<Department12.University0.edu/FullProfessor0/Publication0>
ub:publicationAuthor ?x. }

L6: SELECT ?x ?p WHERE {
?x ub:advisor ?y.
?y ub:worksFor ?z.
?x rdf:type ub:GraduateStudent.
<Department0.University0.edu/FullProfessor0/Publication0>
ub:publicationAuthor ?x. ?p ub:name ?n.
?z rdf:type ub:Department.
?z ub:subOrganizationOf ?w.
?p ub:publicationAuthor ?x. }

L7: SELECT ?x ?y ?z WHERE {
?z ub:subOrganizationOf ?y.
?y rdf:type ub:University.
?z rdf:type ub:Department.
?x rdf:type ub:GraduateStudent.
?x ub:memberOf ?z.
?x ub:undergraduateDegreeFrom ?y. }

L8: SELECT ?x ?y ?z WHERE {
?y ub:teacherOf ?z.
?y rdf:type ub:FullProfessor.
?z rdf:type ub:Course.
?x ub:takesCourse ?z.
?x rdf:type ub:UndergraduateStudent.
?x ub:advisor ?y. }

L9: SELECT ?x ?y ?f ?c ?p ?n WHERE {
?y rdf:type ub:University .
?x rdf:type ub:GraduateStudent .
?x ub:undergraduateDegreeFrom ?y .
?f rdf:type ub:FullProfessor .
?x ub:advisor ?f .
?x ub:takesCourse ?c .
?f ub:teacherOf ?c .
?c rdf:type ub:GraduateCourse .
<Department2.University6.edu/FullProfessor1/Publication1>
ub:publicationAuthor ?f .
?p ub:publicationAuthor ?f .
?p ub:name ?n . }

L10: SELECT ?x ?y ?z ?f ?c ?p ?n WHERE {
?z ub:subOrganizationOf ?y .
?y rdf:type ub:University .
?z rdf:type ub:Department .
?x ub:memberOf ?z .
?x rdf:type ub:GraduateStudent .
?x ub:undergraduateDegreeFrom ?y .
?f rdf:type ub:FullProfessor .
?x ub:advisor ?f .
?x ub:takesCourse ?c .
?f ub:teacherOf ?c .
?c rdf:type ub:GraduateCourse .
<Department2.University6.edu/FullProfessor1/Publication1>
ub:publicationAuthor ?f .
?p ub:publicationAuthor ?f .
?p ub:name ?n . }

UniProt:
PREFIX uni: <http://purl.uniprot.org/core/>
PREFIX uniprot: <http://purl.uniprot.org/>
PREFIX schema: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX file: <file://uniprot.rdf#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX taxon: <http://purl.uniprot.org/taxonomy/>

U1: SELECT ?a ?vo WHERE {
?a uni:encodedBy ?vo.
?a schema:seeAlso <http://purl.uniprot.org/refseq/NP_346136.1>.
?a schema:seeAlso <http://purl.uniprot.org/tigr/SP_1698>.
?a schema:seeAlso <http://purl.uniprot.org/pfam/PF00842>.
?a schema:seeAlso <http://purl.uniprot.org/prints/PR00992>. }

U2: SELECT ?a ?ab ?b ?link ?db WHERE {
<http://purl.uniprot.org/uniprot/Q4N2B5> uni:replacedBy ?a .
?a uni:replaces ?ab .
?ab uni:replacedBy ?b .
?b rdfs:seeAlso ?link .
?link uni:database ?db . }

U3: SELECT ?p2 ?interaction ?p1 ?annotation ?text ?en WHERE {
?p1 uni:enzyme <http://purl.uniprot.org/enzyme/2.7.7.-> .
?p1 rdf:type uni:Protein .
?interaction uni:participant ?p1 .
?interaction rdf:type uni:Interaction .
?interaction uni:participant ?p2 .
?p2 rdf:type uni:Protein .
?p2 uni:enzyme <http://purl.uniprot.org/enzyme/3.1.3.16> .
?p1 uni:annotation ?annotation .
?p1 uni:replaces ?p3.
?p1 uni:encodedBy ?en.
?annotation rdfs:comment ?text . }

U4: SELECT ?a ?ab ?b ?annotation ?range WHERE {
?a uni:classifiedWith <http://purl.uniprot.org/keywords/67> .
?a schema:seeAlso <http://purl.uniprot.org/embl-cds/AAN81952.1>.
?a uni:replaces ?ab .
?ab uni:replacedBy ?b .
?b uni:annotation ?annotation .
?annotation uni:range ?range . }

U5: SELECT ?protein ?annotation WHERE {
?protein uni:annotation ?annotation .
?protein rdf:type uni:Protein .
?protein uni:organism taxon:9606 .
?annotation rdf:type <http://purl.uniprot.org/core/Disease_Annotation>.
?annotation rdfs:comment ?text . }