Query Workload-based RDF Graph Fragmentation and Allocation

Peng Peng[1], Lei Zou[1,3]*, Lei Chen[2], Dongyan Zhao[1]
[1]*Peking University, China;*
[2] *Hong Kong University of Science and Technology, China;*
[3] *Key Laboratory of Computational Linguistics (PKU), Ministry of Education, China*
{ pku09pp,zoulei,zhaodongyan}@pku.edu.cn, leichen@cse.ust.hk

ABSTRACT

As the volume of the RDF data becomes increasingly large, it is essential for us to design a distributed database system to manage it. For distributed RDF data design, it is quite common to partition the RDF data into some parts, called *fragments*, which are then distributed. Thus, the distribution design consists of two steps: fragmentation and allocation. In this paper, we propose a method to explore the intrinsic similarities among the structures of queries in a workload for fragmentation and allocation, which aims to reduce the communication cost during SPARQL query processing. Specifically, we mine and select some *frequent access patterns* to reflect the characteristics of the workload. Based on the selected frequent access patterns, we propose two fragmentation strategies, vertical and horizontal fragmentation strategies, to divide RDF graphs while meeting different kinds of query processing objectives. Vertical fragmentation is for better throughput and horizontal fragmentation is for better performance. After fragmentation, we discuss how to allocate these fragments to various sites. Finally, we discuss how to process a query based on the results of fragmentation and allocation. Extensive experiments confirm the superior performance of our proposed solutions.

1. INTRODUCTION

As a standard model for publishing and exchanging data on the Web, *R*esource *D*escription *F*ramework (RDF) has been widely used in various applications to expose, share, and connect pieces of data on the Web. In RDF, data is represented as triples of the form ⟨subject, property, object⟩. An RDF dataset can be naturally seen as a graph, where subjects and objects are vertices connected by named relationships (i.e., properties). SPARQL is a structured query language proposed by W3C to access RDF repository. As we know, answering a SPARQL query Q is equivalent to finding subgraph matches of query graph Q over an RDF graph G [31]. Figures 1 and 2 show an RDF graph and a set of SPARQL query graphs used as the running example in this paper.

As RDF repositories increase in size, evaluating SPARQL queries is beyond the capacity of a single machine. For example, DBpedia, a project aiming to extract structured content from Wikipedia, consists of 2.46 billion RDF triples [4]; according to the W3C, the numbers of triples in some commercial RDF datasets have been more than 1 trillion [6]. The large-scale of RDF data volume increases the demand of designing the high performance distributed RDF database system.

In distributed database design, the first issue is "data fragmentation and allocation" [18]. We need to divide an RDF graph into several parts, called *fragments*, and then distribute them among sites. One important issue during data fragmentation and allocation in a distributed system is how to reduce the communication cost between different fragments during distributed query evaluation (assuming different fragments are resident at different sites). To minimize the communication cost, many existing graph fragmentation strategies maximize the global goal (such as min-cut [12]). However, evaluating a SPARQL query is a subgraph (homomorphism) match problem. The subgraph match computation often does not involve all vertices in graph G, and the communication cost of subgraph match computation depends on not only the RDF graph but also the query graph. In other words, subgraph match computation exhibits strong locality. There is no direct relation between minimizing the communication cost (in subgraph match computation) and maximizing the global goal. Hence, we propose a *local pattern-based fragmentation* strategy in this paper, which can reduce the communication cost of subgraph match computation.

The intuition behind the local pattern-based fragmentation is as follows: if a query "satisfies" a local pattern and all its matches are in a single fragment, then the query can be evaluated on the single fragment and no communication cost is needed to answering the query. The key issue in local pattern-based fragmentation is how to define the "local patterns". Different from the existing methods, we consider the query workload-driven "local pattern" definition.

1.1 Why Query Workload Matters ?

The workload-driven distributed data fragmentation has been well studied in relational databases [18]. However, few RDF data fragmentation proposals consider the query workload except for [8, 6]. We will review these related papers in Section 9. Here, we discuss why the query workloads is important for RDF data fragmentation.

We study one real SPAQRL query workload, the DBpedia query workload, which records 8,151,238 SPARQL queries issued in 14 days of 2012[1]. For this workload, if we set the minimum support threshold as 0.1% of the total number of queries, we mine 163 frequent subgraph patterns. The most surprising is that 97% query graphs are isomorphic to one of the 163 frequent subgraph pat-

*corresponding author: zoulei@pku.edu.cn

[1]http://aksw.org/Projects/DBPSB.html

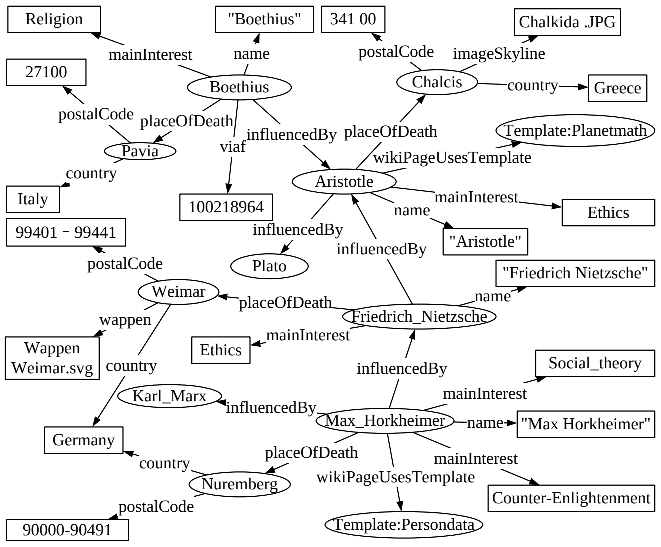


Figure 1: Example RDF Graph

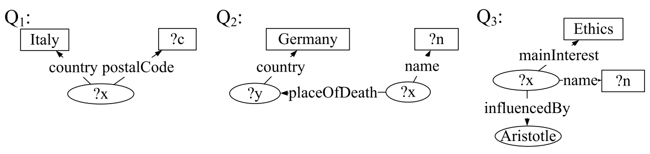


Figure 2: Example SPARQL Query Graphs

terns. Thus, if we use these frequent subgraph patterns as the basic fragmentation units, 97% SPARQL queries do not lead to communication cost, since their matches are resident at one fragment.

1.2 Our Solution

According to the above motivation, we propose a workload-driven data fragmentation for distributed RDF graph systems. Specifically, we first mine frequent subgraph patterns, named *frequent access patterns*, in the query workload. We treat these frequent access patterns as the *implicit* schemas for the underlying RDF data. Then, we propose two fragmentation strategies based on these implicit schemas. We study the following technical issues in this paper.

Frequent Access Pattern Selection. Given a frequent access pattern, we build a fragment by collecting all its matches in the RDF graph. In this way, we can reduce the communication cost (i.e., improve query performance) if a SPARQL query satisfies the frequent access pattern. However, if we simply select all frequent access patterns as the implicit schemas, it may lead to expensive space cost due to the data replication, since different frequent access patterns may involve share the same edges. In other words, we have a tradeoff between performance gain and space cost during selecting frequent access patterns. We formalize the frequent access pattern selection problem (Section 4.1) and prove that it is a NP-hard problem (Theorem 1). Thus, we propose a heuristic algorithm which can guarantee the data integrity and the approximation ratio (Theorem 2). This algorithm also achieves the good performance (See experiments in Section 8).

Vertical and Horizontal Fragmentation. Based on the selected frequent access patterns (i.e., implicit schemas), we design two fragmentation strategies, i.e, vertical and horizontal fragmentation. These two fragmentation strategies are adaptive to different query processing objectives. The objective of vertical fragmentation strategy is to improve the query throughout, and requires that all structures involved by one frequent access pattern should be placed to the same fragment. Instead, the horizontal fragmentation strategy distributes the structures involved by one frequent access pattern among different fragments to maximize the parallelism of query evaluation, namely, reducing the query response time for a single query. To perform the horizontal fragmentation over RDF graphs, we extend the concept of "minterm predicate" in [18] to "structural minterm predicate" (see Section 5.2), which consider the structures of both RDF graphs and workloads. Different applications have different requirements, so we provide customizable options that can be used for different RDF graphs and SPARQL query workloads.

Query Decomposition. As we know, the query decomposition always depends on the fragmentation. In traditional vertical and horizontal fragmentation in RDBMS and XML, the query decomposition is unique, since there is no overlap between different fragments. As mentioned before, there are some data replications in our fragmentation strategies for RDF graphs. Thus, we may have multiple decomposition results for a query. A cost model driven selection is proposed in this paper.

The contributions of this paper can be summarized as follows:

- We analyze the characteristics of the real SPARQL query workload and use the intrinsic similarities of queries in the workload to mine and select some frequent access patterns for distributed RDF data design. Although we prove that the problem of frequent access pattern selection is NP-hard, we propose a heuristic method to achieve the good performance.
- Based on the above scheme, we propose two fragmentation strategies, vertical and horizontal fragmentation, to divide the RDF graph into many fragments and a cost-aware allocation algorithm to distribute fragments among sites. The two fragmentation strategies provide customizable options that are adaptive to different applications.
- We propose a cost-aware query optimization method to decompose a SPARQL query and generate a distributed execution plan. With the decomposition results and execution plan, we can efficiently evaluate the SPARQL query.
- We do experiments over both real and synthetic RDF datasets and SPARQL query workloads to verify our methods.

2. PRELIMINARIES

In this section, we review the terminologies used in this paper and formally define the problem to be addressed.

2.1 RDF and SPARQL

RDF data can be represented as a graph according to the following definition.

DEFINITION 1. ***(RDF Graph)*** *An RDF graph is denoted as $G = \{V(G), E(G), L\}$, where (1) $V(G)$ is a set of vertices that correspond to all subjects and objects in RDF data; (2) $E(G) \subseteq V(G) \times V(G)$ is a set of directed edges that correspond to all triples in RDF data; and (3) L is a set of edge labels. For each edge $e \in E(G)$, its edge label is its corresponding property.*

Similarly, a SPARQL query can also be represented as a query graph Q. For simplicity, we ignore FILTER statements in SPARQL syntax in this paper.

Definition 2. ***(SPARQL Query)*** *A* SPARQL query *is denoted as $Q = \{V(Q), E(Q), L'\}$, where (1) $V(Q) \subseteq V(G) \cup V_{Var}$ is a set of vertices, where $V(G)$ denotes vertices in RDF graph G and V_{Var} is a set of variables; (2) $E(Q) \subseteq V(Q) \times V(Q)$ is a set of edges in Q; and (3) L' is also a set of edge labels, and each edge e in $E(Q)$ either has an edge label in L (i.e., property) or the edge label is a variable.*

In this paper, we assume that Q is a connected graph; otherwise, all connected components of Q are considered separately. Given a SPARQL query Q over RDF graph G, a SPARQL match is a subgraph of G that is homomorphic to Q [31]. Thus, answering a SPARQL query is equivalent to finding all subgraph matches of Q over RDF graph G. The set of all matches for Q over G is denoted as $[\![Q]\!]_G$

In this work, we study a query workload-driven fragmentation. A query workload $\mathcal{Q} = \{Q_1, Q_2, ..., Q_q\}$ is a set of queries that users input in a given period.

2.2 Fragmentation & Allocation

In this paper, we study an efficient distributed SPARQL query engine. There are many issues related to distributed database system design, but, the focus of this work is "data fragmentation and allocation" for RDF repository. We formalize two important problems as follows.

Definition 3. ***(Fragmentation)*** *Given an RDF graph G, a* fragmentation *$\mathcal{F}$ of G is a set of graphs $\mathcal{F} = \{F_1, ..., F_n\}$ such that: (1) each F_i is a subgraph of G and called as a* fragment *of RDF graph G; (2) $E(F_1) \cup ... \cup E(F_n) = E(G)$; and (3) $V(F_1) \cup ... \cup V(F_n) = V(G)$, where $E(F_i)$ and $V(F_i)$ denote the edges and vertices in F_i ($i = 1, .., n$).*

In our work, we allow the overlaps between different fragments. Given a fragmentation $\mathcal{F}$, the next issue is how to distribute these fragments among different sites (i.e., computing nodes). This is called *allocation*.

Definition 4. ***(Allocation)*** *Given a fragmentation $\mathcal{F} = \{F_1, ..., F_n\}$ over an RDF graph G and a set of sites $\mathcal{S} = \{S_1, S_2, ..., S_m\}$ (usually $m < n$), an* allocation *$\mathcal{A} = \{A_1, ..., A_m\}$ of fragments in $\mathcal{F}$ to $\mathcal{S}$ is a partitioning of $\mathcal{F}$ such that (1) $A_j \subseteq \mathcal{F}$, where $1 \leq j \leq m$; (2) $A_{j_1} \cap A_{j_2} = \emptyset$, where $1 \leq j_1 \neq j_2 \leq m$; (3) $A_1 \cup ... \cup A_m = \mathcal{F}$; and (4) All fragments in A_j are stored at site S_j, where $1 \leq j \leq m$.*

Given an RDF graph G, a query workload $\mathcal{Q}$ and a distributed system consisting of sites $\mathcal{S}$, the goal of this paper is to first decompose G into a fragmentation $\mathcal{F}$ and then finding the allocation $\mathcal{A}$ of $\mathcal{F}$ to $\mathcal{S}$.

3. OVERVIEW

This paper studies a SPARQL query workload-driven data fragmentation and allocation problem. Some observations on the real query workload tell us that some RDF properties have few access frequencies. For example, few users input queries contain the properties like *imageSkyline* and *wappen* in Figure 1. As well, the classical distributed database design suggests a "80/20" rule, meaning the active "20%" of query patterns account for "80%" of the total query input [24]. Therefore, we divide the whole RDF repository into two parts: "hot graph" and "cold graph" as follows.

Definition 5. ***(Infrequent and Frequent Property)*** *Given a query workload $\mathcal{Q} = \{Q_1, ...Q_n\}$, if a property p occurs in less than θ queries in $\mathcal{Q}$, where θ is an user specified parameter, p is an* infrequent property*; otherwise, p is a* frequent property.

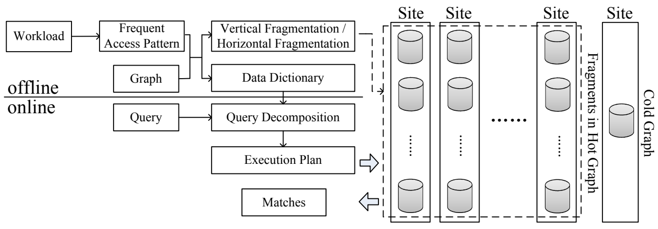


Figure 3: System Architecture

Definition 6. ***(Hot and Cold Graphs)*** *Given an edge $e = \overrightarrow{u_iu_j} \in E(G)$ with property p, if property p is a frequent property, e is a hot edge; otherwise, e is a cold edge.*

Given an RDF graph G, it is divided into two parts: hot graph *H and* cold graph *C, where H consists of all hot edges and C consists of all cold edges.*

The goal of this work is how to partition "hot graph" to achieve performance improvement. We regard the cold graph as a "black block". The cold graph does not overlap to the hot graph, since the cold graph contains different edges with different kinds of properties from the hot graph. Any existing approach can be utilized for the cold graph. We only consider the cold graph in the SPARQL query processing (Section 7), since some queries may involve "infrequent" properties. Moreover, both the cold graph and the hot graph may be disconnected.

Figure 3 illustrates our system architecture. In the offline phase, we mine the *frequent access patterns* (see Section 4) in the workload. Each frequent access pattern can correspond to one or more fragments. Generating a fragment from all matches of a frequent access pattern make many queries be answered efficiently without cross-fragments joins, while it may also replicate some hot edges and increase the space cost. Thus, we should select an appropriate subset of frequent access patterns to balance the efficiency and the space cost. Since we find out that selecting an appropriate set of patterns is a NP-hard problem (Section 4.1), we propose a heuristic pattern selection solution while guaranteeing both the data integrity and the approximation ratio. Based on these selected frequent access patterns, we study two different data fragmentation strategies, i.e., vertical and horizontal fragmentation (Section 5). The vertical fragmentation is to improve the query throughput, and the horizontal fragmentation is to reduce a single query's response time. Fragments are distributed among different sites. Meanwhile, we maintain the metadata in a data dictionary.

In the online phase, we study how to decompose a query into several subqueries on different fragments and generate an efficient execution plan. A cost model for guiding decomposition is proposed (Section 7.2). Finally, we execute the plan and return the matches of the query (Section 7.3).

4. FREQUENT ACCESS PATTERNS

As mentioned before, we believe that a query often contains some patterns in the previously issued queries, so we mine some patterns with high access frequencies and use these patterns as the fragmentation units. Then, if a query Q can be decomposed to some subgraphs isomorphic to the frequent access patterns, Q can be answered while avoiding some joins across multiple fragments.

Before we mine frequent access patterns, we first normalize the query graphs in the workload to avoid overfitting. For each SPARQL query, we remove all constants (strings and URIs) at subjects and

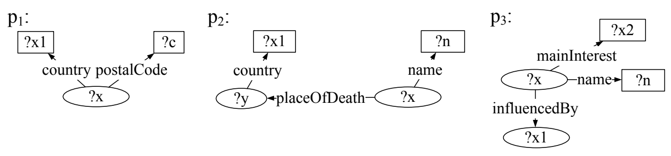


Figure 4: Example Frequent Access Patterns

objects and replace them with variables. The FILTER expressions are also removed. By doing this, we extract a general representation of a SPARQL query from the workload. Figure 4 shows the generalized query graphs of query graphs in Figure 2. We assume that the generalized query in Figure 4 graphs are also frequent access patterns.

To mine patterns with high access frequencies, we need to first count the number of queries in the workload where a pattern p is a subgraph. We define the *frequent access pattern usage value* to record the access frequencies of the frequent access patterns.

DEFINITION 7. ***(Frequent Access Pattern Usage Value)*** *Given a SPARQL query Q and a frequent access pattern p, we associate a* frequent access pattern usage value*, denoted as $use(Q,p)$, and defined as follows:*

$$use(Q,p) = \begin{cases} 1 & if\ pattern\ p\ is\ a\ subgraph\ of\ Q \\ 0 & otherwise \end{cases}$$

Then, given a workload $\mathbf{Q} = \{Q_1, Q_2, ..., Q_q\}$ and a pattern p, we define the *access frequency*, $acc(p)$, as the number of queries in $\mathbf{Q}$ where a pattern p is a subgraph.

$$acc(p) = \sum_{k=1}^{q} use(Q_k, p)$$

A pattern p is *frequent access pattern* if its access frequency is no less than a threshold, $minSup$.

The frequent access patterns can be easily generated by existing frequent graph mining algorithms [17]. Given a workload of SPARQL queries $\mathbf{Q} = \{Q_1, Q_2, ..., Q_q\}$ in a given period, we denote the set of frequent access patterns that we find as $P = \{p_1, p_2, ..., p_x\}$. In practice, the size of P is often limited. For example, if we set $minSup$ as 0.1% of the total access frequency, there are only 163 frequent access patterns for DBPedia.

4.1 Frequent Access Pattern Selection

Obviously, it is not necessary to generate fragments from all frequent access patterns due to high space cost. For two similar frequent access patterns p and p', if they are contained by similar queries of the workload, then selecting both p and p' for building fragments will not be able to provide more information than selecting one of p and p'. Hence, it is often sufficient to only select a subset of all frequent access patterns to generate fragments.

To select a subset of all frequent access patterns, there are two factors that we should consider.

1. (*Hitting the Whole Workload*) We should select frequent access patterns to hit the query workload as much as possible. This is because that when we select a frequent access pattern to generate a fragment, all queries isomorphic to this pattern can be answered directly, which improve the efficiency.

2. (*Satisfying the Storage Constraint*) The total storage of the system in real applications is limited, so selecting too many frequent access patterns is not desirable.

The above two factors contradict each other. Hitting the whole workload requires to select as many frequent access patterns as possible, while the storage constraint requires to select not too many frequent access patterns. There should be a tradeoff between the two factors. In the following, we propose a cost model to combine these two factors for selecting a set of frequent access patterns.

4.1.1 Hitting the Whole Workload

If a fragment is generated from the graph induced by matches of a frequent access pattern, then evaluating all queries containing the pattern can be speeded up by using this fragment. The more queries a frequent access pattern hits, the more gains we obtain during query processing. Therefore, the *benefit* of selecting a frequent access pattern to generate its corresponding fragment should be defined based on the number of queries that the frequent access pattern hits.

In addition, if two similar frequent access patterns are contained by the same set of queries in the workload, it is probably wise to include only one of them. Generally speaking, among similar frequent access patterns contained by the same number of queries, it is often sufficient to materialize only the largest frequent access pattern. That is to say, if p', a subgraph of p, is contained by the same set of queries as p, p is more beneficial than p' to be selected as building fragments. This is because that if we select the larger pattern, a query is more probable to be decomposed to fewer number of subqueries during query processing. Fewer subqueries can avoid some distributed joins, which can improve the efficiency of query processing.

The above observation implies that larger frequent access patterns are more beneficial to be selected as building fragments. This above criterion on the selection of frequent access patterns is formally defined as *size-increasing benefit.*

DEFINITION 8. ***(Size-increasing Benefit)*** *Given a frequent access pattern p, the benefit of selecting p for hitting the query Q, $Benefit(p,Q)$, is denoted as follows.*

$$Benefit(p, Q) = |E(p)| \times use(Q, p)$$

Furthermore, a query in the workload may contain multiple selected frequent access patterns. This means that the query can be decomposed into multiple sets of subqueries if we evaluate the query. Each set of subqueries can map to an execution plan. Since only one execution plan is finally selected to evaluate the query, a query in the workload should only be limited to contribute to the benefits of some particular frequent access patterns once. Based on this observation, we limit a query to only contribute the largest frequent access pattern that the query contains.

DEFINITION 9. ***(Benefit of a Frequent Access Pattern Set)*** *Given a set of frequent access patterns $P' \subseteq P$, the benefit of selection of P' over the workload $\mathbf{Q}$ is the sum of the maximum benefit of its frequent access patterns over $\mathbf{Q}$.*

$$Benefit(P', \mathbf{Q}) = \sum_{Q \in \mathbf{Q}} \max_{p \in P'} \{Benefit(p, Q)\}$$

4.1.2 Satisfying the Storage Constraint

Furthermore, the total storage of the system in real applications is limited, so selecting too many frequent access patterns is not desirable. The selection of frequent access patterns should meet some constraints. When the size of all fragments is larger than the storage constraint, we cannot further select any more frequent access patterns. We normalize the storage capacity of the system to a value SC. Then, we have the constraint as:

$$\sum_{p \in P'} |E([\![p]\!]_G)| \leq SC$$

Here, we assume that SC is larger than the number of edges in the hot graph, so each hot edge can have at least one copy. This assumption guarantees the completeness of the RDF graph.

4.1.3 Combining the Two Factors

Then, our optimization objective is to maximize the benefit subject to the storage constraint. We can prove that this benefit function (Definition 9) is submodular as follows, so this problem is NP-hard.

THEOREM 1. *Finding a set of frequent access patterns with the largest benefit while subject to the storage constraint is NP-hard.*

PROOF. Here, we prove that the benefit function $Benefit(P', Q) = \sum_{Q \in Q} \max_{p \in P'}\{|E(p)| \times use(Q, p)\}$ is submodular. In other words, for every $P_1 \subseteq P_2$ and a frequent access pattern $p \notin P_2$, we need to prove that $\triangle_{Benefit}(p|P_1) \geq \triangle_{Benefit}(p|P_2)$.

For pattern p, we assume that Q' is the set of queries containing p in the workload. There are three kinds of queries in Q': the set Q_1 of queries not containing any patterns in P_2, the set Q_2 of queries containing patterns in $(P_2 - P_1)$, and the set Q_3 of queries only containing patterns in P_1.

Since any query in Q_1 and Q_3 does not concern patterns in $(P_2 - P_1)$, $Benefit(\{p\} \cup P_1, Q_1 \cup Q_3) = Benefit(\{p\} \cup P_2, Q_1 \cup Q_3)$. Hence, the marginal gains of p for P_1 and P_2 over Q_1 and Q_3 are the same.

For Q_2, $\triangle_{Benefit}(p|P_1) > \triangle_{Benefit}(p|P_2)$, if there exist at least one query Q^* meeting all the two following conditions: 1) the largest pattern contained by Q^* over P_2 is in $(P_2 - P_1)$ and has larger size than p; 2) the largest pattern contained by Q^* over P_1 has smaller size than p. The above two conditions mean that p can only increase the benefit of P_1 over Q_2 but not the benefit of P_2 over Q_2. Otherwise, for Q_2, $\triangle_{Benefit}(p|P_1) = \triangle_{Benefit}(p|P_2)$.

In conclusion, $\triangle_{Benefit}(p|P_1) \geq \triangle_{Benefit}(p|P_2)$ and the function $Benefit(P', Q)$ is submodular. Since the problem of maximizing submodular functions is NP-hard [3], the problem is NP-hard. □

4.1.4 Our Solution

As proved in Theorem 1, frequent access pattern selection is NP-complete problem. We propose a greedy algorithm as outlined in Algorithm 1. Note that, to guarantee data integrity of distributed RDF data fragmentation, each hot edge should be contained in at least one fragment. Hence, we initialize a pattern of one edge for each frequent property and compute out its corresponding fragment (Line 3-6).

After we select all patterns with one edge, we enumerate all feasible frequent access pattern sets containing one pattern of more than one edge. Let P_1 be a feasible set of cardinality one that has the largest benefit (Line 7). Then, we iteratively select one of the remaining frequent access patterns p' to maximize the value of $\frac{Benefit(\{p'\} \cup P', Q) - Benefit(P', Q)}{|E([\![p']\!]_G)|}$ until we meet the storage constraint or cannot find a frequent access pattern to increase the benefit (Line 8-14). Let P_2 be the solution obtained in the iterative phase. Finally, the algorithm outputs $P' \cup P_1$ if $Benefit(P' \cup P_1, Q) \geq Benefit(P' \cup P_2, Q)$ and $P' \cup P_2$ otherwise (Line 15-17).

THEOREM 2. *Algorithm 1 obtains a set of frequent access patterns of benefit at least* $\min\{\frac{1}{(\max_{p \in P} |E(p)|)}, \frac{1}{2}(1 - \frac{1}{e})\}$ *times the value of an optimal solution.*

PROOF. There are two parts in Algorithm 1: initialization and greedy selection of frequent access patterns.

Algorithm 1: Frequent Access Pattern Selection Algorithm

```
   Input: A set of frequent access patterns $P = \{p_1, p_2, ..., p_x\}$
   Output: A set $P' \subseteq P$ to generate fragments
1  $P' \leftarrow \emptyset$;
2  $TotalSize \leftarrow 0$;
3  for each $p \in P$ and $p$ has only one edge do
4      $P' \leftarrow P' \cup \{p\}$;
5      $P \leftarrow P - \{p\}$;
6      $TotalSize \leftarrow TotalSize + |E([\![p]\!]_G)|$;
7  $P_1 \leftarrow argmax\{\frac{Benefit(\{p_i\}, Q)}{|E([\![p_i]\!]_G)|} : p_i \in P, |E([\![p_i]\!]_G)| + TotalSize \leq SC \wedge |E(p_i)| > 1\}$;
8  $P_2 \leftarrow \emptyset$;
9  $TotalSize' \leftarrow 0$;
10 while $TotalSize' \leq SC - TotalSize$ do
11     Find the frequent access pattern $p' \in P - P'$ with the largest
       additional value of $\frac{Benefit(\{p'\} \cup P', Q) - Benefit(P', Q)}{|E([\![p']\!]_G)|}$;
12     $P_2 \leftarrow P_2 \cup \{p'\}$;
13     $P \leftarrow P - \{p'\}$;
14     $TotalSize' \leftarrow TotalSize' + |E([\![p']\!]_G)|$;
15 if $Benefit(P' \cup P_1, Q) \geq Benefit(P' \cup P_2, Q)$ then
16     Return $P' \cup P_1$;
17 Return $P' \cup P_2$;
```

For initialization (Line 3-6 in Algorithm 1), all selected patterns only contain one edge, so $|E(p)| = 1$. Therefore, the benefit of patterns only having one edge of a frequent property is $\sum_{Q \in Q} \max_{p \in P'}\{1 \times use(Q, p)\}$. Since the hot edges hit almost all queries in the workload, $\sum_{Q \in Q} \max_{p \in P'}\{1 \times use(Q, p)\}$ is approximately equal to the size of the workload, $|Q|$. On the other hand, in the worst case, the optimal solution is that all queries in the workload contain the largest frequent access pattern. Then, the benefit of the optimal solution is $\sum_{Q \in Q}\{|E(p_{max})| \times use(Q, p)\}$, where p_{max} is the frequent pattern with the largest size. Hence, the benefit of the selected patterns in the initial phase is at least $\frac{1}{(\max_{p \in P} |E(p)|)}$ of the optimal benefit.

For the phase of greedily selecting frequent access patterns (Line 7-14 in Algorithm 1), since the problem of selecting the optimal set of frequent access patterns is a problem of maximizing a submodular set function subject to a knapsack constraint as discussed in Theorem 1, we directly apply the greedy algorithm in [11] to iteratively select frequent access patterns. [11] proves that the worst-case performance guarantee of the greedy algorithm is $\frac{1}{2}(1 - \frac{1}{e})$, so the benefit of the selected patterns in this phase is at least $\frac{1}{2}(1 - \frac{1}{e})$ of the optimal benefit.

In summary, the final performance guarantee of our algorithm is $\min\{\frac{1}{(\max_{p \in P} |E(p)|)}, \frac{1}{2}(1 - \frac{1}{e})\}$. □

5. FRAGMENTATION

In this section, we present two fragmentation strategies: vertical and horizontal.

5.1 Vertical Fragmentation

For vertical fragmentation, we put matches homomorphic to the same frequent access pattern into the same fragment. Because a query graph often only contains a few frequent access patterns and matches of one frequent access pattern are put together, other irrelevant fragments can be filtered out during query evaluation and only sites stored relevant fragments need to be accessed to find matches. Filtering out irrelevant fragments can improve the query performance. Furthermore, sites not storing relevant fragments can be used to evaluate other queries in parallel, which improves the total throughput of the system. In summary, the vertical fragmentation strategy utilizes the locality of SPARQL queries to improve both query response time and throughput. Experimental results in Section 8 also confirm the above argument.

Given a frequent access pattern p, it can then be transformed into a SPARQL query, resulting in a vertical fragment of the RDF graph. We use the results $[\![p]\!]_G$ of a selection operation based on p to generate a vertical fragment. All vertical fragments generated from our selected frequent access patterns construct a vertical fragmentation. Given a set of frequent access patterns P, we formally define its corresponding vertical fragmentation over an RDF graph G as follows.

DEFINITION 10. ***(Vertical Fragmentation)*** *Given an RDF graph G and a frequent access pattern p, a* vertical fragment *F generated from p is defined as $F = \{V(F), E(F), L''\}$, where (1) $V(F) \subseteq V(G)$ is the set of vertices occurring in $[\![p]\!]_G$; (2) $E(F) \subseteq E(G)$ is the set of edges occurring in $[\![p]\!]_G$; and (3) $L'' \subseteq L$ is the set of edge labels occurring in $[\![p]\!]_G$.*

Then, given a set of frequent access patterns $P = \{p_1, p_2, ..., p_x\}$, the corresponding vertical fragmentation *is $\mathcal{F} = \{F_i | 0 \leq i \leq x$ and F_i is the vertical fragment generated from p_i.$\}$*

EXAMPLE 1. *Given the frequent access pattern p_3 in Figure 4, Figure 5 shows the corresponding vertical fragment.*

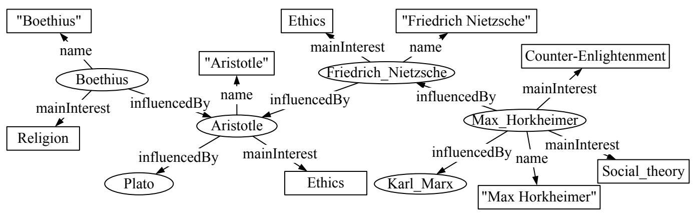


Figure 5: Example Vertical Fragment

5.2 Horizontal Fragmentation

For horizontal fragmentation, we put matches of one frequent access pattern into the different fragments and distribute them among different sites. Then, a query may involve many fragments and each fragment has a few matches. The size of a fragment is often much smaller than the size of the whole data, so finding matches of a query over a fragment explores smaller search space than finding matches over the whole data. If the fragments involved by a query are allocated to different sites, then each site finds a few matches over some fragments with the smaller size than the whole data. This strategy is to utilize the parallelism of clusters of sites to reduce the query response time. The above argument is also confirmed by the experimental results in Section 8.

In this section, we extend the concepts of *simple predicate* and *minterm predicate* originally developed for relational systems [18] to divide the RDF graph horizontally.

5.2.1 Structural Minterm Predicate

First, we define the structural simple predicate. Each structural simple predicate corresponds to a frequent access pattern with a single (in)equality. Given a frequent access pattern p with variables set $\{var_1, var_2, ..., var_n\}$, a structural simple predicate sp defined on D has the following form.

$$sp : p(var_i)\ \theta\ Value$$

where $\theta \in \{=, \neq\}$ and $Value$ is a constant constraint for var_i chosen from a query containing p in $\boldsymbol{Q}$.

EXAMPLE 2. *Let us consider the query graph Q_3 in Figure 2 and its corresponding frequent access pattern p_3 in Figure 4. We can generate four structural simple predicates: (1). sp_1 : $p_3(?x1) = Aristotle$; (2). sp_2 : $p_3(?x1) \neq Aristotle$; (3). sp_3 : $p_3(?x2) = Ethics$; (4). sp_4 : $p_3(?x2) \neq Ethics$.*

Then, we define the structural minterm predicate as the conjunction of structural simple predicates of the same frequent access pattern. We can obtain all structural minterm predicates by enumerating all possible combinations of structural simple predicates. Given a set of structural simple predicates $SP = \{sp_1, sp_2, ...,, sp_y\}$ for frequent access pattern p, the set of structural minterm predicates $M = \{mp_1, mp_2, ..., mp_z\}$ for p is defined as follows.

$$M = \{mp_i | \bigwedge_{sp_k \in SP} sp_k^*, 1 \leq k \leq y\}$$

where $sp_k^* = sp_k$ or $sp_k^* = \neg sp_k$. So each structural simple predicate can occur in a structural minterm predicate either in its natural form or its negated form.

Similar to the frequent access pattern, we can also define the *structural minterm predicate usage value* and *access frequency* to record the access frequency of a structural minterm predicate. We can prune the minterm predicates with small access frequencies.

DEFINITION 11. ***(Structural Minterm Predicate Usage Value)*** *Given a SPARQL query Q and a structural minterm predicate mp, we associate a* structural minterm predicate usage value*, denoted as $use(Q, mp)$, and defined as follows:*

$$use(Q, mp) = \begin{cases} 1 & if\ predicate\ mp\ is\ a\ subgraph\ of\ Q \\ 0 & otherwise \end{cases}$$

Then, given a set of SPARQL queries $\boldsymbol{Q} = \{Q_1, Q_2, ..., Q_q\}$, we define the *access frequency* of a structural minterm predicate mp as follows.

$$acc(mp) = \sum_{k=1}^{k=q} use(Q_k, mp)$$

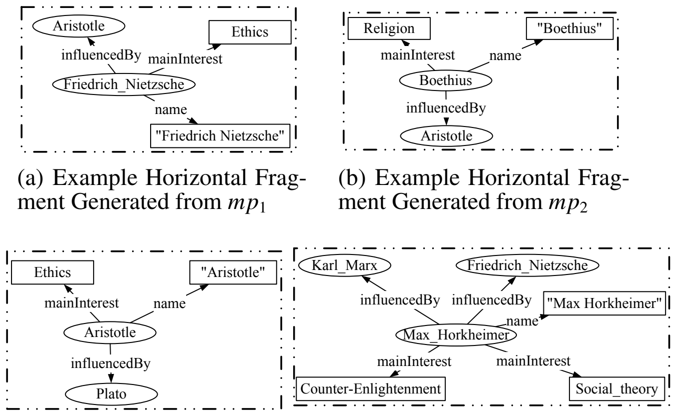
In practice, there may exist many minterm predicates. It is too expensive to enumerate all minterm predicates. Therefore, we prune some minterm predicates with too small access frequencies.

Given a structural minterm predicate mp, it can then be transformed into SPARQL queries, resulting in a horizontal fragment of the RDF graph. We use the results $[\![mp]\!]_G$ of a selection operation based on mp to generate a horizontal fragment. All horizontal fragments generated from the structural minterm predicates that we obtain construct a horizontal fragmentation. Given a set of minterm predicates M, we formally define its corresponding horizontal fragmentation over an RDF graph G as follows.

DEFINITION 12. ***(Horizontal Fragmentation)*** *Given an RDF graph G and a structural minterm predicate mp, a* horizontal fragment *F generated from mp is defined as $F = \{V(F), E(F), L''\}$, where (1) $V(F) \subseteq V(G)$ is the set of vertices occurring in $[\![mp]\!]_G$; (2) $E(F) \subseteq E(G)$ is the set of edges occurring in $[\![mp]\!]_G$; and (3) $L'' \subseteq L$ is the set of edge labels occurring in $[\![mp]\!]_G$.*

Then, given a set of structural minterm predicates $M = \{mp_1, mp_2, ..., mp_y\}$, the corresponding horizontal fragmentation *is $\mathcal{F} = \{F_i | 0 \leq i \leq y$ and F_i is the vertical horizontal generated from mp_i.$\}$*

EXAMPLE 3. *Given the structural simple predicates in Example 2, we can get all structural minterm predicates from frequent access pattern p_3 as follows: (1). mp_1 : $p_3(?x0) = Aristotle \wedge p_3(?x1) = Ethics$; (2) mp_2 : $p_3(?x0) = Aristotle \wedge p_3(?x1) \neq$*



(a) Example Horizontal Fragment Generated from mp_1

(b) Example Horizontal Fragment Generated from mp_2

(c) Example Horizontal Fragment Generated from mp_3

(d) Example Horizontal Fragment Generated from mp_4

Figure 6: Example Horizontal Fragments

Ethics; (3). $mp_3 : p_3(?x0) \neq Aristotle \wedge p_3(?x1) = Ethics$*; (4).* $mp_4 : p_3(?x0) \neq Aristotle \wedge p_3(?x1) \neq Ethics$*.*

Figure 6 shows all horizontal fragments generated from the above structural minterm predicates.

6. ALLOCATION

After fragmenting the RDF graph, the next step is to allocate all fragments on several sites. In real applications, some frequent access patterns or structural minterm predicates are usually accessed together, so their corresponding fragments should be placed in one site to further avoid the cross-fragments joins. There is a need for some measures evaluating precisely the notion of "togetherness". This measure is the affinity of fragments, which indicates how closely related the fragments are.

We define *fragment affinity metric* to measure the togetherness between two frequent access patterns or structural minterm predicates as follows:

DEFINITION 13. ***(Fragment Affinity Metric)*** *The* fragment affinity metric *between two fragments F and F' with respect to the workload $\mathbf{Q} = \{Q_1, Q_2, ..., Q_q\}$ is defined as follows*

- $aff(F, F') = \sum_{k=1}^{q} use(Q_k, p) \times use(Q_k, p')$*, if F and F' are vertical fragments generated from frequent access patterns p and p';*
- $aff(F, F') = \sum_{k=1}^{q} use(Q_k, mp) \times use(Q_k, mp')$*, if F and F' are horizontal fragments generated from structural minterm predicates mp and mp';*

Based on the fragment affinity metric, we can show how closely related the fragments are. If the affinity metric of two fragments is large, it means that these two fragments are often involved by the same query. Some fragments are so related that they should be placed together to reduce the number of cross-sites joins. Here, we group all fragments into some clusters. The result of clustering corresponds to an allocation $\mathcal{A}$, and each cluster corresponds to an element of $\mathcal{A}$, which means that all fragments in the cluster are placed into the same site.

There are many clustering algorithms to cluster all fragments and we need to select one of them. In this paper, we extend a graph clustering algorithm, PNN [5], to cluster all fragments into an allocation $\mathcal{A} = \{A_1, A_2, ..., A_m\}$. All fragments of the same cluster are put into one site.

First, we build the *allocation graph* as follows.

DEFINITION 14. ***(Allocation Graph)*** *Given a fragmentation $\mathcal{F} = \{F_1, F_2, ..., F_n\}$, the corresponding* allocation graph $AG = \{V(AG), E(AG), f_W\}$ *is defined as follows:*

- *$V(AG)$ is a set of vertices that map to all fragments;*
- *$E(AG)$ is a set of undirected edges that $\overline{vv'} \in E(VG)$ if and only if the fragment affinity metric between the corresponding fragments of v and v' is larger than 0;*
- *f_W is a weight function $f_W : E(AG) \rightarrow N^+$. If v and v' correspond to fragments F and F', $f_W(\overline{vv'}) = aff(F, F')$.*

Then, the allocation problem is equivalent to cluster all fragments in m clusters, and all fragments in a cluster are connected in AG. We define the *density* of a cluster A_i in AG to rate the quality of A_i as follows.

$$\delta(A_i) = \frac{\sum\limits_{v_i \in A_i \wedge v_j \in A_i \wedge \overline{v_i v_j} \in E(AG)} f_W(\overline{v_i v_j})}{\binom{|A_i|}{2}}$$

where $\sum\limits_{v_i \in A_i \wedge v_j \in A_i \wedge \overline{v_i v_j} \in E(AG)} f_W(\overline{v_i v_j})$ is the sum of weights of all edges in A_i and $\binom{|A_i|}{2}$ is the maximum possible number of edges.

The objective of our allocation algorithm is to search for m subgraphs of AG that have the highest densities. Unfortunately, this problem is NP-complete [20], so we propose a heuristic solution as Algorithm 2. Algorithm 2 is a variant of PNN and picks the locally optimal choice of merging two vertices in AG at each step. Because our objective function can guarantee the locally optimal choice is also the optimal choice for the overall solution, Algorithm 2 can find out the optimal clustering result of AG.

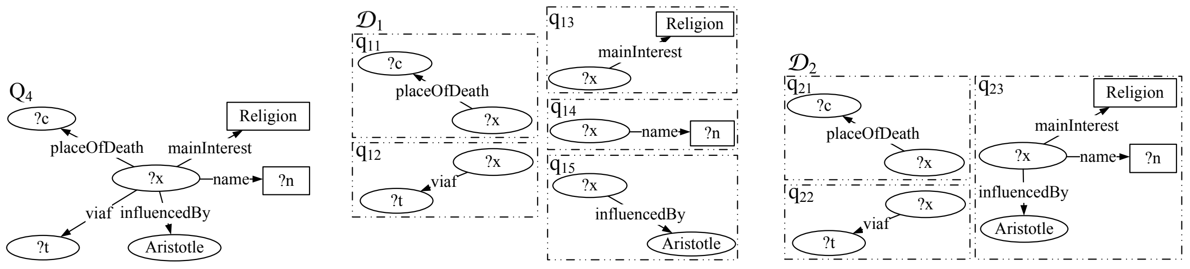
Generally speaking, we initialize a cluster for each fragment. Then, we repeatedly picks the two clusters (singletons or larger) that have the highest weight value to be merged. The weight between two clusters are the density value of merging them. Such merging is iterated until the size of the allocation graph has been reduced to m.

Algorithm 2: Allocation Algorithm

```
  Input: The allocation graph $AG$ and the preset threshold $\theta$
  Output: An allocation $\mathcal{A} = \{A_1, A_2, ..., A_m\}$
1 for each vertex $v_i$ in $V(VG)$ do
2     $A_i \leftarrow \{v_i\}$;
3 Find the edge $e_{max}$ with the highest weight in $E(AG)$;
4 Initialize $AG'$ that is the same to $AG$;
5 while $|V(AG')| \neq m$ do
6     Generating $AG'$ from $AG$ by merging $e_{max} = \overline{A_i A_j}$ to $A_{ij}$;
7     for each $A_k$ adjacent to $A_{ij}$ in $E(AG')$ do
8         $f_W(\overline{A_k A_{ij}}) \leftarrow \frac{\sum_{v_i \in A_k \wedge (v_j \in A_i \vee v_j \in A_j) \wedge \overline{v_i v_j} \in E(AG)} f_W(\overline{v_i v_j})}{\binom{|A'_k|}{2}}$
9     Find the edge $e_{max}$ with the highest weight in $E(AG')$;
```

7. DISTRIBUTED QUERY PROCESSING

In this section, we discuss how to process a SPARQL query. For query processing, the metadata is necessary and we introduce how to maintain the metadata in a data dictionary in Section 7.1. Then, we discuss how to decompose a query into some subqueries in Section 7.2. Last, we discuss how to produce a distributed execution plan and execute all subqueries based on the plan in Section 7.3.



(a) A New Input Query Q_4 (b) Valid Decomposition $\mathcal{D}_1$ (c) Valid Decomposition $\mathcal{D}_2$

Figure 7: A New Input Query and Its Example Valid Decompositions

7.1 Data Dictionary

After fragmentation and allocation, the results of fragmentation and allocation need to be stored and maintained by the system. This information is necessary during distributed query processing. This information is stored in a data dictionary. The data dictionary stores a global statistics file generated at fragmentation and allocation time. It contains the following information: fragment definitions, their sizes, site mappings, access frequencies and so on.

Since each fragment corresponds to a frequent access pattern or a structural minterm predicate, the data dictionary uses the frequent access pattern with/without constraints as the representative of a fragment. Each frequent access pattern with/without constraints corresponds to a fragment and is associated with all statistics of the fragment. The data dictionary need to fast retrieve all frequent access patterns with/without constraints to determine the relevant frequent access pattern for a query.

We build a hash table to achieve the above objective. We first use the DFS coding [26] to translates frequent access patterns into sequences. With the DFS code of a frequent access pattern, we can map any frequent access pattern to an integer by hashing its canonical label. Then, we use the hash table to locate frequent access patterns and retrieve the statistics of their corresponding fragments.

7.2 Query Decomposition

When users input a query Q, the system first uses the data dictionary to determine which fragments are involved in the query and decomposes the query into some subqueries on fragments.

Given a query Q, a decomposition of Q is a set of subqueries $\mathcal{D} = \{q_1, q_2, ..., q_t\}$ such that (1) each q_i is a subgraph of Q and q_i maps to a frequent access pattern or structural minterm predicate; (2) $V(q_1) \cup ... \cup V(q_t) = V(Q)$; and (3) $E(q_1) \cup ... \cup E(q_t) = E(Q) \wedge \forall i \neq j, E(q_i) \cap E(q_j) = \emptyset$.

Since we partition the RDF graph based on the frequent access patterns, we also decompose the query based on the frequent access patterns. In other words, we decompose the query into subqueries that are homomorphic to frequent access patterns. If a query involves infrequent properties that cannot be decomposed into subqueries homomorphic to any frequent access patterns, then each connected subgraph of the query that only contains infrequent properties corresponds to a subquery. We define the *valid* decomposition as follows.

DEFINITION 15. ***(Valid Decomposition)*** *Given a SPARQL query Q, a* valid *decomposition $\mathcal{D} = \{q_1, q_2, ..., q_t\}$ of Q should meet the following constraint: if q_i ($1 \leq i \leq t$) is not homomorphic to any frequent access patterns, all edges in q_i should be cold edges.*

There exist at least one valid decompositions. A possible decomposition is the decomposition of all subqueries of a single edge. Because we select all frequent access patterns of one edge, the decomposition of all subqueries of a single edge is valid. Besides the valid decomposition, there may also exist some other valid decompositions. Hence, we propose a cost-model driven selection and the best valid decomposition is the valid decomposition with the smallest cost.

Here, we assume that the cost of a decomposition is the cost of joining all matches of the subqueries in $\mathcal{D}$ and each pair of subqueries' matches can join together. The assumption is the worst case, so that we can quantify the worst-case performance. Then, we define the cost of a decomposition as follows.

$$cost(\mathcal{D}) = \prod_{q_i \in \mathcal{D}} card(q_i)$$

where $card(q_i)$ is the number of matches for q_i, which can be estimated by looking up the data dictionary.

EXAMPLE 4. *Assume that an user inputs a new query Q_4 as shown in Figure 7(a). Given frequent access patterns in Figure 4, there can be two valid decompositions $\mathcal{D}_1$ and $\mathcal{D}_2$ as shown in Figures 7(b) and 7(c). For vertical fragmentation, q_{23} in $\mathcal{D}_2$ is evaluated on the vertical fragment of p_3 (Figure 5); for horizontal fragmentation, q_{23} is evaluated on the horizontal fragment of mp_2 (Figure 6(b)).*

Whether in vertical or in horizontal fragmentation, it is obvious that $\mathcal{D}_2$ has fewer subqueries than $\mathcal{D}_1$ and $card(q_{23}) < card(q_{13}) \times card(q_{14}) \times card(q_{15})$. Hence, $cost(\mathcal{D}_2)$ is smaller than $cost(\mathcal{D}_1)$, and $\mathcal{D}_2$ is more of a priority as the final decomposition.

Based on the above definitions, we propose the query decomposition algorithm as Algorithm 3. Because the SPARQL query graphs in real applications usually contain 10 or fewer edges, we can use a brute-force implementation to enumerate all possible decompositions and find the decomposition with the smallest cost.

Algorithm 3: Query Decomposition Algorithm

```
Input: A query $Q$
Output: A valid decomposition $\mathcal{D} = \{q_1, q_2, ..., q_t\}$ of query $Q$
1  $MinCost \leftarrow +\infty$;
2  Initialize $\mathcal{D}$ as the decomposition of all subqueries of a single edge;
3  for each possible valid decomposition $\mathcal{D}' = \{q_1, ..., q_t\}$ do
4      $CurrentCost \leftarrow 1$;
5      for each query $q_i$ in $\mathcal{D}'$ do
6          Estimate the number of results for $q_i$ as $card(q_i)$ based on the
           data dictionary;
7          $CurrentCost \leftarrow CurrentCost \times card(q_i)$
8      if $MinCost > CurrentCost$ then
9          $\mathcal{D} \leftarrow \mathcal{D}'$;
10         $MinCost \leftarrow CurrentCost$;
11 Return $\mathcal{D}$;
```

7.3 Query Optimization and Execution

After decomposing the query, the next step is to find an execution plan for the query which is close to optimal. In this section, we discuss the major optimization issue of finding execution plan, which deals with the join ordering of subqueries. We extend the algorithm of System-R [2] to find the optimal execution plan for distributed SPARQL queries. The algorithm is described in Algorithm 4.

Generally speaking, Algorithm 4 is a variant of System-R style dynamic programming algorithm. It firstly generates the best execution plan of $n-1$ subqueries, and then join the matches of $n-1$ subqueries with the matches of n-th subquery. The cost of an execution plan can also be estimated based on the number of subqueries' results, which is stored in the data dictionary.

Finally, each subquery is executed in the corresponding sites in parallel. The optimization of each subquery uses the existing methods in centralized RDF database systems. After the matches of all subqueries are generated, we join them together according to the optimal execution plan.

Algorithm 4: Query Optimization Algorithm

```
Input: A decomposition $\mathcal{D} = \{q_1, q_2, ..., q_t\}$ of query $Q$
Output: An execution plan $(...((q_{i1} \bowtie q_{i2}) \bowtie q_{i3}) \bowtie ... \bowtie q_{it})$
1  for each two subqueries $(q_i)$ and $(q_j)$ where $1 \leq i \neq j \leq t$ do
2      Initialize an execution plan $q_i \bowtie q_j$ and estimate its cost;
3      Store all execution plans and their costs in a table $T_2$;
4  for $i = 3$ to $t$ do
5      for each execution plan $pl_j$ in $T_{i-1}$ do
6          for each subquery $q_k$ that is not contained by $pl_j$ do
7              Build execution plan $pl_j \bowtie q_k$ and estimate its cost;
8              Store this execution plan and its costs in a table $T_i$;
9          for each two plans $pl_j$ and $pl_k$ in $T_i$ do
10             if $pl_j$ and $pl_k$ map to the same set of subqueries then
11                 Eliminate one of $pl_j$ and $pl_k$ that has the larger cost;
12 Return the execution plan with the minimum cost;
```

8. EXPERIMENTAL EVALUATION

We conducted extensive experiments to test the effectiveness of our proposed techniques on a real dataset, DBPedia, and a synthetic dataset, WatDiv. In this section, we report the setting of test data and various performance results.

8.1 Setting

DBPedia. DBPedia[2] is an RDF dataset extracted from Wikipedia. The DBPedia contains 163, 977, 110 triples. We use the DBpedia SPARQL query-log as the workload. This workload contains queries posed to the official DBpedia SPARQL endpoint in 14 days of 2012. After removing some queries that cannot be handled, there are 8, 151, 238 queries in the workload.

WatDiv. WatDiv [1] is a benchmark that enable diversified stress testing of RDF data management systems. In WatDiv, instances of the same type can have the different sets of attributes. For testing our methods, we generate five datasets varying sizes from 50 million to 250 million triples. By default, we use the RDF dataset with 100 million triples. In addition, WatDiv can generate a workload by instantiating some templates with actual RDF terms from the dataset. WatDiv provides 20 templates to generate test queries. We use these benchmark templates to generate a workload with 2000 test queries.

We conduct all experiments on a cluster of 10 machines running Linux, each of which has one CPU with four cores of 3.06GHz. Each site has 16GB memory and 150GB disk storage. We select one of these sites as a control site. At each site, we install gStore [31] to find matches. We use MPICH-3.0.4 running on C++ to join the results generated by subqueries.

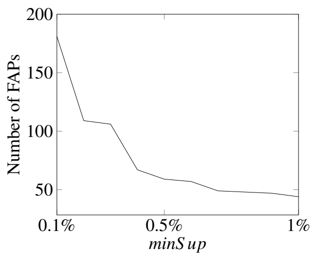
[2]http://km.aifb.kit.edu/projects/btc-2012/dbpedia/

For fair performance comparison, we use gStore and MPICH-3.0.4 to re-implement two recent distributed RDF fragmentation strategies. The first one is SHAPE [14], which defines a vertex and its neighbors as a triple group and assigns the triple groups according to the value of its center vertices. There are many different kinds of triple groups in [14] and we use the subject-object-based triple groups in this paper. The second one is WARP [8]. WARP first uses METIS [12] to divide the RDF graph into fragments. Then, it replicates all matches of a query pattern that cross two fragments in one fragment. We use all frequent access patterns to extend the fragments in WARP.

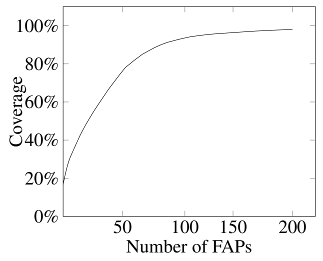
8.2 Parameter Setting

Our frequent access patterns selection method uses a parameter: $minSup$. In this subsection, we discuss how to set up $minSup$ to optimize query processing. Note that, since the numbers of query templates and queries per query template in WatDiv are specified by users, the parameters can also be determined beforehand. Thus, we only discuss how to set the parameters for DBPedia.

Given a workload Q, we set the support threshold, $minSup$, to find patterns whose access frequencies are larger than $minSup$. It is clear that the smaller $minSup$ is, the larger number of frequent access patterns there are. More frequent access patterns mean that a query in the workload may have a higher possibility to contain some frequent access patterns.



(a) $minSup$



(b) Workload Hitting Ratio

Figure 8: Effect of Frequent Access Patterns

Figure 8(a) shows the impact of $minSup$. As $minSup$ increases, the number of frequent access patterns (FAPs) decreases. Hence, when we set $minSup$ as 0.1% of the total number of queries in the workload, there are 163 frequent access patterns for DBPedia. When $minSup$ is 1% of the total number of queries, the number of frequent access patterns is reduced to 44 for DBPedia. Furthermore, fewer frequent access patterns means that fewer queries in the workload are hit, as shown in Figure 8(b).

Even if we set $minSup$ as 0.1% of the total number of queries, the number of frequent access patterns is not large. Hence, in the following, we set $minSup$ as 0.1% of the total number of queries for DBPedia by default.

8.3 Throughput

In this experiment, we test the throughput of different fragmentation strategies. We sample 1% of all queries in the workload and measure the throughput in queries per minute. Figure 9 shows the number of queries answered in one minute of different fragmentation strategies.

For SHAPE and WARP, each query concerns all fragments, so queries are still processed sequentially. Since WARP is more balanced than SHAPE, the throughput of WARP is a little better than SHAPE. WARP can handle about 32 and 82 queries in one minute

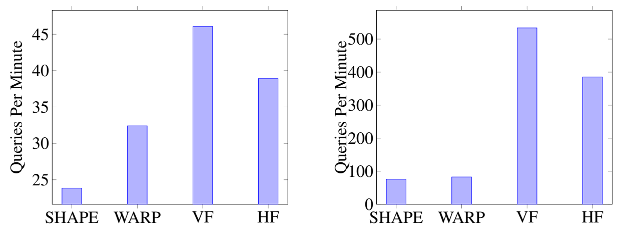


Figure 9: Throughput Comparison

for DBPedia and WatDiv, while SHAPE can handle 24 and 75 queries.

For the vertical fragmentation strategy (VF), since a query often only contains a few frequent access patterns, it only involves a few fragments. Two queries involving different fragments can be evaluated in parallel. Hence, about 46 queries and 533 queries can be answered in one minute for DBPedia and WatDiv, respectively. For the horizontal fragmentation strategy (HF), each frequent access pattern specified by the query may map to many structural minterm predicates and the corresponding fragments of these structural minterm predicates may be allocated to different sites. Hence, the throughput of the horizontal fragmentation strategy is a little worse than the vertical fragmentation strategy, and 38 and 385 queries can be answered in one minute for DBPedia and WatDiv.

8.4 Response Time

In this experiment, we test the query performance of different fragmentation strategies. We also sample 1% of all queries in the workload and compute the average query response time of a query. Figure 10 shows the performance results.

SHAPE and WARP partition the RDF graph into some subgraphs, and distributes these subgraphs among different sites. The query should be processed in many sites in parallel. Hence, SHAPE is less balanced and sometime need cross-fragment joins, so SHAPE needs about 2.5 and 0.79 seconds to answer a query for DBPedia and WatDiv, while WARP takes 1.8 and 0.72 seconds.

For the vertical fragmentation strategy, only relevant fragments are searched for matches and the search space is reduced. Therefore, a query can be answered in about 0.8 seconds for DBPedia and 0.3 seconds for WatDiv. For the horizontal fragmentation strategy, we can filter out all irrelevant fragments mapping to the structural minterm predicates not specified by the query, which can further reduce the search space. Hence, a query can be answered with about 0.6 seconds for DBPedia and 0.15 seconds for WatDiv.

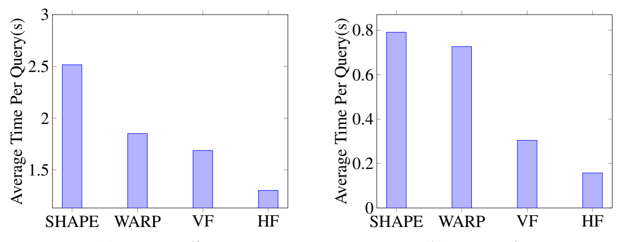


Figure 10: Performance Comparison

8.5 Scalability Test

In this experiment, we investigate the impact of dataset size on our fragmentation strategies. We generate five WatDiv datasets varying the from 50 million to 250 million triples to test our strategies. Figure 11 shows the results. Generally speaking, as the size of RDF datasets gets larger, the average response times of one query increase and the numbers of queries answered in one minute decrease accordingly. However, the rates of increase and decrease are slow, and we can say that the query performance and throughput are scalable with RDF graph size on the datasets.

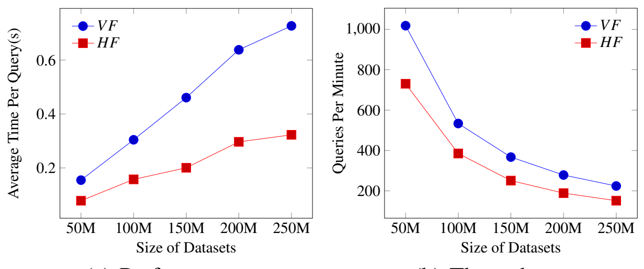


Figure 11: Varying Size of Datasets

8.6 Redundancy

Table 1 shows the redundancy ratio of the number of edges in all generated fragments to the total number of edges in the original RDF graph for each fragmentation strategy. For SHAPE, if a fragment contains a vertex with high degree, all adjacent edges of the high degree vertex are introduced. Most of these introduced edges are redundant, and cause the redundancy ratios of SHAPE nearly 3 for DBPedia and 1.74 for WatDiv. WARP divides the RDF graph while minimizing the edge cut, so the number of edges crossing two fragments for WARP is smaller than the number for SHAPE. Therefore, the redundancy ratio of WARP is smaller. Note that, WatDiv is much denser than DBPedia, so the minimum cut-set for WatDiv contains a higher proportion of edges. Hence, the redundancy ratio of WatDiv is 1.54, but the ratio of DBPedia is only 1.01.

	DBPedia	WatDiv
SHAPE	2.99	1.74
WARP	1.01	1.54
VF	1.38	1.04
HF	1.42	1.06

Table 1: Redundancy (Ratio to original dataset)

Our fragmentation strategies find and materialize some frequent access patterns (or structural minterm predicates). As discussed in Section 8.2, the number of frequent access patterns is limited. Hence, the redundancy ratios of our fragmentation strategies are limited. Note that, the horizontal strategy has a little larger redundancy ratio than the vertical fragmentation strategy. This is because that different structural minterm predicates derived from the same frequent access patterns share some common triple patterns. These common triple patterns may cause more redundant edges.

8.7 Offline Performance

Table 2 shows the data partitioning and loading time of the datasets for different fragmentation strategies. Although SHAPE has an almost perfect uniform distribution, its redundancy ratio is too large and each fragment contains too many redundant edges. Hence, loading fragments in SHAPE also takes much time. WARP uses METIS [12]. Since DBPedia is sparse (i.e. $|(E(G)|/|V(G)| \approx 1)$, METIS can guarantee that there are a few redundant edges and all fragments have a nearly uniform distribution. Then, WARP has less loading time than SHAPE. However, for WatDiv, the data graph is dense (i.e. $|(E(G)|/|V(G)| \gg 1)$, so the fragmentation result of

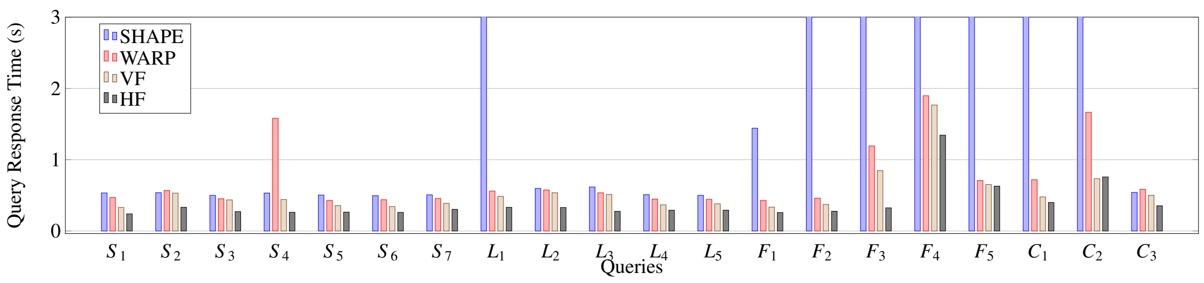


Figure 12: Query Performance of Benchmark Queries

METIS is unbalanced. Then, WARP takes more loading time than SHAPE to load the largest fragments.

Since nearly half of all edges for DBPedia are infrequent edges, loading the cold graph of DBPedia is the bottleneck in our fragmentation strategies. However, in WatDiv, there are not so many infrequent edges. Then, the loading time of our fragmentation strategies for WatDiv is more acceptable. Note that, because the structural minterm predicates are derived from the frequent access patterns, the cold graphs for the vertical and horizontal fragmentation strategies are the same. Thus, the loading times for the vertical and horizontal fragmentation strategies are the same.

	DBPedia			WatDiv		
Strategies	Partitioning	Loading	Total	Partitioning	Loading	Total
SHAPE	41	30	71	20	19	49
WARP	43	28	71	33	46	79
VF	50	97	147	31	28	59
HF	58	97	139	34	28	62

Table 2: Partitioning and Loading Time (in min)

8.8 Experiments for Benchmark Queries

In this experiment, we compare our methods with other fragmentation strategies on benchmark queries provided by WatDiv. There are 20 benchmark queries in WatDiv, and these queries can be classified into 4 structural categories: linear (L), star (S), snowflake (F) and complex (C). Figure 12 shows the performance of different approaches. Generally speaking, we find out that our methods outperforms other two methods in most cases. This is because that each benchmark query can be decomposed into some frequent access patterns or structural minterm predicates. Hence, our fragmentation strategies can filter out many irrelevant fragments. In contrast, SHAPE and WARP always concern all fragments, and SHAPE further needs some cross-fragment joins for complex queries.

Let us look deeper into Figure 12 and analyze each individual fragmentation strategy. SHAPE has to involve all fragments for any queries, so its performance is always worse than our fragmentation strategies. In particular, for star queries (S_1 to S_7), the difference between the query response times of SHAPE and our fragmentation strategies is not very large, because the subject-object-based triple groups that we use can guarantee that there is no intermediate result and all star queries can be answered at each fragment locally. However, for other shapes of queries, SHAPE has to decompose the queries and do cross-fragment joins to merge the intermediate results. Then, the performance of SHAPE decreases greatly. Especially for the unselective queries (L_1, F_1, F_2, F_3, F_4, F_5, C_1 and C_2), the performance of SHAPE is an order of magnitude worse than our fragmentation strategies.

Since WARP also use patterns to replicate triples for avoiding cross-fragment joins in complex queries, WARP has better performance that SHAPE in most case. However, WARP still always concerns all fragments in all sites for any kind of queries. The search space of WARP for a query is higher than our fragmentation strategies. Thus, our fragmentation strategies always result in better performance. Especially for the query of very complex structure (C_2), our fragmentation strategies can filter out many irrelevant fragments, which can result in much smaller search space than WARP. Hence, for C_2, our strategies is twice as fast as WARP.

Since all benchmark queries are generated from instantiating benchmark templates with actual RDF terms, these benchmark queries always correspond to a limited number of minterm predicates. Hence, the horizontal fragmentation is always faster than the vertical fragmentation.

9. RELATED WORK

For both the general graph and the RDF graph, as the graph size grows beyond the capability of a single machine, many works [6, 8, 9, 10, 29, 14, 15, 7, 23, 12, 30, 22, 25] have been proposed about graph fragmentation and allocation. We can divide all these methods into two categories: global goal-oriented graph fragmentation methods and local pattern-based graph fragmentation methods.

Global Goal-Oriented Graph Fragmentation. For this kind of methods [12, 9, 30, 22, 16], they divide G into several fragments while maximizing some goal function. They first transform a large graph into a small graph; then, apply some graph partitioning algorithms on the small graph; finally, the partitions on the small graph are projected back to the original graph. These methods often apply some existing methods (such as KL [13]) directly on the transformed graph in the second step. If we track the transforming step, the partitions on the small graph can be easily projected back to the original graphs in the third step. Hence, the largest difference among different graph coarsening-based methods is how to coarsen the original graph into a small graph.

In particular, METIS [12] uses the maximal matching to coarsen the graph. A matching of a graph is a set of edges that no two edges share an endpoint. A maximal matching of a graph is a matching to which no more edges can be added and remain a matching. GraphPartition [9] directly uses METIS in the RDF graph. WARP [8] uses some frequent structures in workload to further extend the results of GraphPartition. EAGRE [30] coarsens the RDF graph by using the entity concept in RDF data. It considers an entity to be a subject and its complete description. By grouping the entities of the same class, an RDF graph can be compressed as a compressed RDF entity graph. MLP [22] designs a method to coarsen the graph by label propagation. Vertices with the same label after the label propagation are coarsened to a vertex in the coarsened graph. Sheep [16] transform the graph into a elimination tree via a distributed map-reduce operation, and then partition this tree while reducing communication volume. Tomaszuk et. al. [21] briefly survey how to apply existing graph fragmentaion solutions from the theory of graphs to RDF graphs.

Global goal-oriented graph fragmentation methods assume that if there are few edges crossing different fragments, the communication cost is little. If an application involves nearly all vertices in the graph, few cross-fragments edges indeed result in little communication. A typical application suitable for graph coarsening-based methods is PageRank.

In some applications, one static fragmentation cannot fit all. Hence, Sedge [28] maintains many fragmentations with different crossing edges, while Shang et. al. [19] move some vertices of one fragment to another fragment during graph computing according to the workload. Yan et. al. [27] propose a indexing scheme based on fragmentation to help query engine fast locate the instances.

Local Pattern-based Graph Fragmentation. For this kind of methods [10, 29, 14, 15, 7, 23, 25] , they first find certain patterns as the fragmentation units to cover the whole graph; then, they distribute these patterns into sites. The local pattern-based methods mainly differ in their definitions of the fragmentation unit.

HadoopRDF [10] groups triples with the same property together and each group corresponds to a fragmentation unit. Then, they store all fragmentation units over HDFS. Yang et. al.[29] define some special query patterns, and subgraphs of a pattern are considered as a fragmentation unit. Lee et. al. [14, 15] define the fragmentation unit as a vertex and its neighbors, which they call a triple group. The triple groups are distributed based on some heuristic rules. For each vertex, SketchCluster [23] identifies the set of labeled vertices reachable within its one-hop neighborhood as its features and employs the KModes algorithm to group related vertices based on the features. Partout [6] extends the concepts of minterm predicates in relational database systems, and uses the results of minterm predicates as the fragmentation units. TriAD [7] uses METIS [12] to divide the RDF graph into many partitions. Then, each result partition is considered as a unit and distributed among different sites based on a hash function. PathPartitioning [25] uses paths in RDF graphs as fragmentation units.

Local pattern-based graph fragmentation methods assume that some real applications only concerns a part of the whole graph. If an application only concerns the vertices of some certain patterns, these methods only access the relevant fragments and reduce the communication cost across fragments. A typical example application is subgraph homomorphism checking.

10. CONCLUSION

In this paper, we discuss how to manage the large RDF graph in a distributed environment. First, we mine and select some frequent access patterns to partition the RDF graph into many smaller fragments. Then, we propose an allocation algorithm to distribute all fragments over different sites. Last, we discuss how process the query based on the results of fragmentation and allocation. Extensive experiments verify our approaches.

Acknowledgement. This was supported by 863 project under Grant No. 2015AA015402, NSFC under Grant No. 61532010, 61370055, 61272344 and 61303073. Lei Chen's work is supported in part by the Hong Kong RGC Project N HKUST637/13, National Grand Fundamental Research 973 Program of China under Grant 2014CB340303 , NSFC Grant No. 61328202, NSFC Guang Dong Grant No. U1301253, Microsoft Research Asia Gift Grant and Google Faculty Award 2013

11. REFERENCES

[1] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212, 2014.

[2] M. M. Astrahan, H. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1:97–137, 1976.

[3] L. Bordeaux, Y. Hamadi, and P. Kohli. *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 2014.

[4] DBpedia. http://dbpedia.org/about.

[5] P. Fränti, O. Virmajoki, and V. Hautamäki. Fast PNN-based Clustering Using K-nearest Neighbor Graph. In *ICDM*, pages 525–528, 2003.

[6] L. Galarraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. In *WWW (Companion Volume)*, pages 267–268, 2014.

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

[8] K. Hose and R. Schenkel. WARP: Workload-aware Replication and Partitioning for RDF. In *ICDE Workshops*, pages 1–6, 2013.

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

[10] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. Knowl. Data Eng.*, 23(9):1312–1327, 2011.

[11] R. K. Iyer and J. A. Bilmes. Submodular Optimization with Submodular Cover and Submodular Knapsack Constraints. *CoRR*, abs/1311.2106, 2013.

[12] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. In *SC*, 1995.

[13] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2), 1970.

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

[15] K. Lee, L. Liu, Y. Tang, Q. Zhang, and Y. Zhou. Efficient and customizable data partitioning framework for distributed big RDF data processing in the cloud. In *IEEE CLOUD*, pages 327–334, 2013.

[16] D. W. Margo and M. I. Seltzer. A Scalable Distributed Graph Partitioner. *PVLDB*, 8(12):1478–1489, 2015.

[17] S. Nijssen and J. N. Kok. The Gaston Tool for Frequent Subgraph Mining. *Electr. Notes Theor. Comput. Sci.*, 127(1):77–87, 2005.

[18] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[19] Z. Shang and J. X. Yu. Catch the Wind: Graph Workload Balancing on Cloud. In *ICDE*, pages 553–564, 2013.

[20] J. Síma and S. E. Schaeffer. On the NP-Completeness of Some Graph Cluster Measures. *CoRR*, abs/cs/0506100, 2005.

[21] D. Tomaszuk, L. Skonieczny, and D. Wood. RDF Graph Partitions: A Brief Survey. In *BDAS*, pages 256–264, 2015.

[22] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to Partition a Billion-node Graph. In *ICDE*, pages 568–579, 2014.

[23] Y. Wang, S. Parthasarathy, and P. Sadayappan. Stratification Driven Placement of Complex Data: A Framework for Distributed Data Analytics. In *ICDE*, pages 709–720, 2013.

[24] G. Wiederhold. *Database Design, Second Edition*. McGraw-Hill, 1983.

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

[26] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD Conference*, pages 335–346, 2004.

[27] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficient Indices Using Graph Partitioning in RDF Triple Stores. In *ICDE*, pages 1263–1266, 2009.

[28] S. Yang, X. Yan, B. Zong, and A. Khan. Towards Effective Partition Management for Large Graphs. In *SIGMOD Conference*, pages 517–528, 2012.

[29] T. Yang, J. Chen, X. Wang, Y. Chen, and X. Du. Efficient SPARQL Query Evaluation via Automatic Data Partitioning. In *DASFAA (2)*, pages 244–258, 2013.

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

[31] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gStore: A Graph-based SPARQL Query Engine. *VLDB J.*, 23(4):565–590, 2014.