An Efficient Summary Graph Driven Method for RDF Query Processing

Lei Gai, Wei Chen and Tengjiao Wang

*School of Electrical Engineering and Computer Science
Peking University. Beijing, P.R.China
Email: {lei.gai, pekingchenwei, tjwang}@pku.edu.cn

Abstract—RDF query optimization is a challenging problem. Although considerable factors and their impacts on query efficiency have been investigated, this problem still needs further investigation. We identify that decomposing query into a series of light-weight operations is also effective in boosting query processing. Considering the linked nature of RDF data, the correlations among operations should be carefully handled. In this paper, we present SGDQ, a novel framework that features a partition-based Summary Graph Driven Query for efficient query processing. Basically, SGDQ partitions data and models partitions as a summary graph. A query is decomposed into subqueries that can be answered without inter-partition processing. The final results are derived by perform summary graph matching and join the results generated by all matched subqueries. In essence, SGDQ combines the merits of graph match processing and relational join-based query implementation. It intentionally avoids maintain huge intermediate results by organizing sub-query processing in a summary graph driven fashion. Our extensive evaluations show that SGDQ is an effective framework for efficient RDF query processing. Its query performance consistently outperforms the representative state-of-the-art systems.

I. INTRODUCTION

The Resource Description Framework (RDF) have been pervasively adopted in many fields, such as knowledge bases(DBpedia [1], YAGO [2], Freebase [3]), semantic social networks (FOAF [4], Open Graph [5]) and bioinformatics (Bio2RDF [6], UniProt [7]), etc. The available RDF data keep increasing in both size and quality, this challenges for a framework for efficient query processing. For schema-flexible RDF data, query is naturally a graph pattern matching process. Generating results in a refined data space can greatly boost the RDF query performance. To this end, a considerable exploratory methods have been proposed in the recent decade.

One class of solutions utilize the graph nature of RDF data and implement query as subgraph matching [8]–[10]. Subgraph matching only needs to explore the neighborhood data, and generates results in an incremental fashion. This avoids maintaining large intermediate results [9]. Nonetheless, the scale of RDF graph have a profound impact on query performance. The time complexity of subgraph matching increase exponentially as the graph size increase. Furthermore, as graph exploration is based on random data access, the efficiency of exploration-intensive subgraph match is guaranteed by holding the whole graph topology in memory, which is infeasible for large-scale graph. Recent researches [11], [12] introduced graph summarization in RDF query. The condense summary graph are solely used in filtering out irrelevant data in query.

The other class, which are the mainstream solutions, are based on 40+ years of relational database researches [13], [14]. In principle, RDF data is managed as relational tables, and joins are excessively utilized in query implementation. The scan and join on large table are prone to introduce large intermediate results, which can easily overwhelm the system memory and cause memory thrashing that greatly degrade query performance. Existing query optimization methods mainly focus on triple indexing techniques (e.g. [15], [16]) and query optimization methods (e.g. [17], [18]) that utilizing the typical characteristics of RDF. Using relational solutions, we argue that splitting RDF data into smaller parts that can be processed independently is also effective in optimizing query processing. On the flip side, there is no one-size-fit-all partition method, some results may span across partitions. The existence of inter-partition processing complicated the query implementation in this manner. Considering that the extensive connections among partitions can be naturally modeled using graph structure, this envisions us to introduce a summary graph and use subgraph match to drive the operations on partitions.

In this paper, we present a new RDF processing framework named SGDQ (Summary Graph Directed Query). SGDQ partitions data and decomposes a query into sub-queries, such that each sub-query can be answered independently in partitions. All sub-query processing are driven by a summary graph matching process, and the final results is generated as joins between sub-queries in a match. The rational behind SGDQ is two folds. First, SGDQ alleviates the need of maintaining massive intermediate results by decomposing query process into a series of light-weight sub-queries. Second, subgraph match is effective in representing the inter-partition query processing, and it is can be efficiently implemented on a summarized graph of moderate scale.

To achieve this, two aspects in SGDQ need to be carefully considered. The first aspect is the design of physical data layout that support decomposed sub-query processing. We distinguish triples that do not contribute to graph partition and manage them in a specific manner. To facilitate summary match, we adopt a graph partition based strategy for data partition. Each partition is managed individually and the inter-connection among partitions is modeled as a summary graph. The second aspect is the efficient processing of sub-queries. We propose a process that use auxiliary patterns to filter out irrelevant data in sub-query processing, and adopt a mechanism to prevent duplications in final results.

We summarize our major contributions in this paper as follows:
We present SGDQ (Summary Graph Directed Query), a novel and effective framework for RDF query processing. SGDQ decomposes a query into a series of light-weight sub-queries, and leverages summary graph matching paradigm to drive the processing of sub-queries. SGDQ can greatly reduce the amount of data that need to maintain during query, and boost query performance by effective summary graph matching and efficient sub-query processing.

We propose a specialized physical data layout that support SGDQ, and provide efficient operations based on this physical data layout. These operations are the building blocks of SGDQ.

We perform extensive evaluations using two types of benchmarks. The evaluation results show that the query performance of SGDQ consistently outperforms three existing representative RDF systems.

The rest of the paper is organized as follows. Section II describes the principle concepts. Section III gives an overview of SGDQ framework. Section IV presents the physical data layout and implementation of sub-query processing. Section V details the paradigm of summary graph driven processing. The most related works is reviewed in Section VI, and Section VII presents the evaluation results. We conclude in Section VIII.

II. PRELIMINARIES

RDF is a W3C recommended standard for representing and publishing linked web resources. Due to its simple and schema-free nature, RDF facilitates the representation and integration of data from different domains. In RDF, A real world resource is identified by an Uniform Resource Identifier (URI) string. Its properties and linkages to other resources are described by a set of triples. Each triple is uniformly represented as three elements (Subject, Predicate, Object), or (s,p,o) for brevity. We call p a relation if o is a resource URI, or an attribute if o is a literal (string, number, or date, etc.). A collection of such triples forms an RDF dataset \( D = \bigcup_i D_i \). \( D \) can be naturally modeled as a directed, edge-labeled RDF graph \( G = (V, E) \). Each triple \( t_i \in D \) corresponds to a directed labeled edge \( e \in E \), which connects vertex \( s \) to vertex \( o \) with an edge value \( p \).

Like SQL in relational database, users commonly resorts to declarative query for RDF data access. Adopting query language such as SPARQL [19], a conjunctive RDF query \( Q \) can be expressed as: \( \text{SELECT } ?v_1, \ldots, ?v_n \text{ FROM } TP \), where \( TP \) is a combination of triples, each one of which has a least one of \( s, o \) or \( p \) replaced by variables \( ?v \). We refers to such triples as triple patterns, and denotes a query as \( Q = \{ tp_1, \ldots, tp_k \} \). Two triple patterns are joinable if they implicitly share a common elements. In this paper, we assume that all predicates of triple patterns may not be variables. This assumption is reasonable for that Predicate join is infrequent in practice, as shown in a previous study [20]. This leads to 3 types of joins, named according to the position of common elements:

- \( s-s \) join, \( o-o \) join, \( s-o \) join. By considering each \( tp_i \) as an edge and follows same way of mapping \( D \) to \( G \), \( Q \) can also be viewed as a query graph \( G^Q \). Accordingly, a query can be implemented as the process of match \( G^Q \) on \( G \).

The notations used in this paper are listed in Table II. Next, we further introduce some principle concepts.

| Notation | Description |
|----------|-------------|
| \( D \) | RDF dataset, \( D = \bigcup_i D_i \) in N-Triple (s,p,o) |
| \( s, p, o \) | denotes the value of Subject, Predicate and Object in a triple. |
| \( D_A, D_R \) | The set of A-Triples and R-Triples. |
| \( M_A, M_R \) | The global dictionary of \( D_A \) and \( D_R \). |
| \( T \) | Query, as a set of \( k \) triple patterns, \( Q = \{ tp_1, \ldots, tp_k \} \). |
| \( \mathcal{G}_R \) | The Resource-Linkage graph of \( D, \mathcal{G}_R = (V_R, E_R) \). |
| \( \mathcal{G}_P \) | The Resource-Linkage graph of \( Q, \mathcal{G}_P = (V_P, E_P) \). |
| \( P \) | A partition set of \( D, P = \{ P_1, \ldots, P_m \}, \bigcup_i P_i = D \). |
| \( \mathcal{S}_P \) | The summary data graph that models \( P, \mathcal{S}_P = (V_S, E_S) \). |
| \( \mathcal{G}_Q \) | The summary query graph, \( \mathcal{G}_Q = (V_Q, E_Q) \). |
| \( \mathcal{G}_{Q^R} \) | The graph partition strategy. |
| \( \mathcal{D}_{op} \) | The inter-parition join processing. |

A. Triple Classification

Considering that the relation and attribute Predicate show different characteristics, \( D \) and \( Q \) fall into two categories define as follow:

Definition 1 (R/A-Triple, R/A-Pattern): A triple \( t_i \in D \) is called a R-Triple(an A-Triple) if its \( p \) is a relation( an attribute). The set of all R-Triples(or A-Triples) in \( D \) is denoted as \( D_R \) or \( D_A \). Analogously, A triple pattern \( tp_i \in Q \) is called a R-Pattern(an A-Pattern) if its \( p \) is a relation( an attribute), and the set of R-Pattern(A-Pattern) in \( Q \) is denoted as \( Q_R \) or \( Q_A \).

Intuitively, a \( tp_i \in Q_R \in (Q_A) \) can only be match to \( t_j \in D_R \in (D_A) \). Using Definition 1, we can get a compact graph model defines as follow:

Definition 2 (Resource-Linkage(RL) Graph): The graph representation of \( D_R \) or \( Q_R \) is named as the Resource-Linkage graph of \( D \) or \( Q \), or RL graph for brevity, denoted as \( \mathcal{G}_R \) or \( \mathcal{G}_{Q^R} \).

According to Definition 2, \( \mathcal{G}_R \) or \( \mathcal{G}_{Q^R} \) only models the linkage information of resources. That provides a compact representation of \( G \) or \( G^Q \). Intuitively, Property 1 holds.

Property 1 (RL Graph Match): Following the subgraph match defined in [21], if there exists a subgraph \( g \) in \( G \) that match \( G^Q \), consequently, the RL graph of \( g \) is a match of \( G^Q \).

Example 1: All triples of an RDF dataset \( D \) are listed in Fig 1(a), \( D_A \) in the left and \( D_R \) in the right. \( D_R \) represents part of a twitter-like semantic social network, which has 3 social accounts, 2 posted contents and 4 replies. Fig 1(b) shows the equivalent RDF graph \( G \) and Fig 1(c) is the RL-graph of \( G \). Given a query \( Q \) with 12 triple patterns shown in Fig 2(a), its query graph \( G^Q \) is represented as Fig 2(b). For \( D \), there are two matched triple sets of \( Q \), lists in Fig 2(c). Therefore, the result of \( Q \) are \( \{u1, u3, p1, r3, r4\} \), and \( \{u1, u3, p1, r2, r1\} \).
between two partitions; e = (V, V) where graph keeps the schematic topological information among partitions.

Fig. 1: An exemplar RDF dataset D, its data graph G and RL-graph G_R.

(b) G, the RDF graph of D.

Fig. 2: An exemplar query and matched triples.

(a) An exemplar RDF dataset, D.

B. Summary Graph and Query

For an RDF graph G, graph partition allows triples that are close to each other in G to be managed in the same partition. Thus, a large G can be summarized into a small graph that keeps the schematic topological information among partitions. We introduce a kind of summary graph defines as follow:

**Definition 3 (Summary RDF Graph):** Given an RDF dataset D, a set P = {P_1, ..., P_n} is the partition set of its corresponding RDF graph G = (V, E). A summary RDF graph SG = (S_V, S_E, P) is a labeled, undirected multi-graph, where S_V = {P_1, ..., P_n}, and S_E is consist of: i) edges that connects two vertices in S_V if there exists a connecting edge between two partitions; ii)loop edges for each vertexes in S_V, e = (v, v), v ∈ S_V. f_p maps all distinct p in P_i as vertex label.

Query decomposition is extensively used in converting a complex query into a set of simple sub-queries. A query Q_R can be decomposed into a set of sub-queries, and each sub-query is a subgraph of the original query graph G^Q. We use the term subgraph and sub-query interchangeably in this paper. Based on query decomposition, we introduce the definition of transformed query graph as follow:

**Definition 4 (Transformed Query Graph):** Given a query Q and its corresponding query graph G^Q = (V^Q, E^Q). Let one of its decomposition as Q^P = \{Q^P_1, ..., Q^P_m\}, where Q^P_k = \{V^P_k, E^P_k\}. A transformed query graph \(TQ^G = \{V^T, E^T, f_q\}\) for Q is a vertex-labeled, undirected graph, where \(V^T = \{Q^P_1, ..., Q^P_m\}\), and edge \((Q^P_i, Q^P_j) \in E^T\) if there exists common vertices between \(Q^P_i\) and \(Q^P_j\). \(f_q\) maps all distinct p in \(Q^P\) as vertex label.

**Example 2:** We partition the G_R in Fig. 1(c) into two partitions, \(P^* = \{P^*_1, P^*_2\}\), show as Fig. 4(a). Using \(P^*\) we can model a Summary RDF Graph SG as Fig. 4(c). For the query in Fig. 1(a) its query RL-Graph Q_R can be decomposed into four subgraphs \(Q^P\), show as Fig. 4(b). Using \(Q^P\), we model a transformed query graph \(TQ^G\) as Fig. 4(d).

C. Summary Graph Match

Let \(OP_{sp}\) refers to the operation of a sub-query on a data partition. More precisely, the processing of a \(Q^P_i\) on a \(P_a \in \mathcal{P}\) is denoted as \(Q^P_i(P_a)\), and its result set is denoted as \(R(Q^P_i(P_a))\). To deal with the connection between sub-queries, we introduce the concept of intra-partition join, expressed as \(OP_{ij}\). For two \(OP_{sp}\) namely \(Q^P_i(P_a)\) and \(Q^P_j(P_b)\), \(1 \leq i, j \leq m, 1 \leq a, b \leq n\), a \(OP_{ij}\) is the join processing between \(R(Q^P_i(P_a))\) and \(R(Q^P_j(P_b))\), denoted as \(OP_{ij}(Q^P_i(P_a), Q^P_j(P_b))\), and the result set is denoted as \(R(Q^P_i(P_a), Q^P_j(P_b))\). We define the summary graph match of \(TQ^G\) on \(SG\) as follow:

**Definition 5 (Summary Graph Match):** Given a \(TQ^G = \{V^T, E^T, f_q\}\) and a \(SG = \{S_V, S_E, f_v\}\), where \(V^T = \{V^P_1, ..., V^P_m\}\), and \(S_V = \{V_1, ..., V_n\}\), and \(E^T = \{E^T_1, ..., E^T_m\}\), and \(f_q\) maps all distinct \(p\) in \(P_i\) as vertex label.
{Q_1^p, ..., Q_n^p}, V_S = \{P_1, ..., P_n\}, a summary graph match is an injective function \( M : V^S_T \rightarrow V_S \), such that: i) vertex match: \( \forall Q_i^p \in V^T_S, f_i^p(Q_i^p) \subseteq f_i(M(Q_i^p)), \) and \( R(Q_i^p(M(Q_i^p))) \neq \emptyset \), and ii) edge injection: \( \forall (Q_i^p, Q_j^p) \in E^S, (M(Q_i^p), M(Q_j^p)) \in E_S, \) and \( R(Q_i^p(M(Q_i^p)), Q_j^p(M(Q_j^p))) \neq \emptyset \).

III. OVERVIEW OF SGDQ

Fig. 3 gives an overview of SGDQ framework. A dataset \( D \) is offline indexed to facilitate online query processing.

A. Offline Processing

Given an RDF dataset \( D \), according to the \( p \) of each triple, \( D \) is divided into two parts, \( D_A \) and \( D_R \). \( D_A \) is managed on disk using a set of bitmap-based indexes (Section IV-A). Besides, \( D_R \) is modeled as a data RL-graph \( G_R \), and partitioned into \( n \) set of subgraphs \( P^o = \{P_1^o, ..., P_n^o\} \) using METIS algorithm (denoted as \( P^o_{\text{METIS}} \)). We call \( P^o \) as an original partitions. \( P^o_{\text{METIS}} \) is an edge-cut graph partition method, it divides a graph into a given number (there it is \( n \)) of roughly vertices-balanced connected subgraphs \( P_j = (V_j, E_j) \), \( 1 \leq j \leq n \), with minimum number of cutting edges. Therefore, we can model \( P^o \) as a summary data graph \( G^S \) according to Definition 5. \( G^S \) is maintained as an in-memory adjacency list to facilitate graph exploration. Each partition in \( P^o \) is further expanded using a partition strategy which duplicates boundary triples, and final partitions \( P = \{P_1, ..., P_n\} \) are generated. Each \( P_i \in P \) is managed as a self-governed triple store. All triples in \( D_R \) are encoded globally using a global dictionary. This enables inter-partition processing. (Section IV-B).

B. Runtime Query Processing

Once a user interactively submit a query \( Q \), the query processing is implemented in three steps.

Step 1. Query preprocessing. Under the premise of no Predicate join in \( Q \), \( Q \) is divided into \( Q_A \) and \( Q_R \), judged by whether \( p \) of each triple pattern is a relation or an attribute. \( Q_R \) is modeled as a \( G^R_R \), while \( Q_A \) are grouped by Subject. By decomposing \( G^Q_R \) into set of sub-queries \( Q^p = \{Q_1^p, ..., Q_n^p\} \), we model a \( TG^Q \) (Section IV-C).

Step 2. A-Pattern processing. Foremost, each group of \( Q_A \) is processed using bitmap indexes, and the candidate set of the corresponding vertex in \( G^Q_R \) are generated for further used in \( OP_{sp} \) (Section IV-A).

Step 3. Summary graph driven processing. Based on data partition and query decomposition, the query processing is decomposed as a set of light-weight \( OP_{sp} \). Follow the paradigm of match \( TG^Q \) on \( S_G \). We organize the execution of \( OP_{sp} \) and perform \( OP_{ip} \) between two \( OP_{sp} \). The final results is generated in an incremental fashion, and stops until no more matches could be found. (Section IV-A).

IV. DATA ORGANIZATION AND OPERATIONS

In this section, we explain how to decompose a costly query processing a series of light-weight \( Op_{ip} \) operations. A common practice is to introduce data partition and query reformulation. Our implementation is efficient in two aspects. On one side, we identify two types of query pattern to simplify a query. Utilize the characteristics of A-Pattern, we introduce bitmap index for A-Triples and implement efficient bitwise operation for A-Pattern processing (Section IV-A). On the other side, we introduce a data partition strategy (Section IV-B) and corresponding query decomposition method, and implement an efficient \( Op_{ip} \) (Section IV-C).

A. Bitmap Indexes for A-Triples and A-Pattern processing

Managing and querying A-Triples in a specific manner brings two main benefits. First, it avoids unnecessary duplication of A-Triples in data partitions. This simplifies data partition. Second, it employs efficient bitmap-based bitwise operations that take advantage of the characteristics of s-s join for A-Patterns.
Bitmaps are commonly used in databases as they can be efficiently compressed and processed. Using bitmap in RDF data, s-s joins on a common Subject can be translated to simple bitwise AND operations. This saves a potentially large number of joins compared with canonical relational query processing. For this reason, some RDF researches have also considered the extensive adoption of bitmap indexes.

To encode data as bitmap, we build two types of global dictionary, one for $D_R$ and the other for $D_A$, denoted as $M_R$ and $M_A$ respectively. $M_R$ holds the mapping of an unique ID, namely VertexID, to a s or an o in $D_R$(i.e., $v \in V_R$). $M_R$ is built according to the original partitions. We consecutively allocate a VertexID to each vertex grouped by partitions, and store all VertexIDs in an sorted order in $M_R$. Similarly, $M_A$ is the mapping of an AttributeID to an o in $D_A$. All AttributeIDs are managed as a sorted list in $M_A$. The partial order lists of VertexID and AttributeID in $M_R$ and $M_A$ are further used in bitmap indexes construction.

Based on encoding data globally using $M_R$ and $M_A$, we design and implement a set of compressed column-oriented indexes that dedicated for A-Triples management and A-Patterns processing. Consider there are two types of A-Patterns,

**Pattern-I** ($\forall s,p,o$), where $o$ is a designated URI or Literal. **Pattern-II** ($\forall s,p,o$) and ($s,p,o$), where $o$ is a variable.

For **Pattern-I**, we design a bitmap of size $|V_R|$, referred to as BitVector, to represent the existence of all VertexID for a given combination of $o$ and $p$. In a BitVector, a bit at position $i$ corresponds to the $i$-th vertex in $D_R$, and is set to 1 if triple (VertexID, $p$, o) is retrieved, or 0 otherwise. There are totally $|D_A|$ such BitVectors. They are grouped by $p$, and each group is sorted and indexed on $o$ using B-tree to facilitate index lookup. Consider that the size of BitVectors (i.e. $|V_R|$) can be quite large, a common way in practice is using compression to take advantage of the sparse nature of the vectors. Similar to [15], we adopt D-Gap compression. This allows bitmap operations to be performed against compressed data, which eliminates the cost of AttributeVector decompression. Such index can be viewed as a special implementation of POS permutation index. We name it as ABIdx$_{POS}$.

For **Pattern-II**, we design two kinds of index, namely, ABIdx$_{comp}$ and ABIdx$_{PSO}$. In ABIdx$_{comp}$, each $p$ maintains a bitmap which stores the bitwise AND results of all BitVectors of this $p$ in ABIdx$_{POS}$. This bitmap captures all A-Triples that have this $p$. ABIdx$_{PSO}$ can be viewed as a special implementation of PSO index, and implemented the same way as ABIdx$_{POS}$, where a BitVector of size $|M_A|$ is arranged according to the sorted list of AttributeID in $M_A$.

At query time, for a $v \in V_R$ of a given $Q$, let $T_P$ be the set of A-Patterns that have $v$ as Subject. We introduce an operation, named CandidateRLVertex, that inputs $T_P$ and returns the existence of candidate VertexIDs for $v$ as a BitVector of size $|V_R|$. For each pattern $tp_i \in T_P$, CandidateRLVertex first check the type of $tp_i$. If $tp_i$ is a **Pattern-I**, it retrieves ABIdx$_{POS}$ according to $p$ and $o$ of $tp_i$ and get a BitVector. Otherwise, if $tp_i$ is a **Pattern-II**, it gets a BitVector in ABIdx$_{comp}$ corresponding to the $p$ of $tp_i$. The final result is a BitVector that is derived from the bitwise AND of all retrieved BitVectors.

This BitVector represents the existence of all VertexIDs. To get the corresponding AttributeID for the variable $o$ in **Pattern-II**, we implement an extra operation, named RetrieveAttribute, at the final stage of $Q$ processing. RetrieveAttribute retrieves the BitVector in ABIdx$_{POS}$ according to the VertexID in previous results and $p$ in **Pattern-II**, then retrieves $M_A$ to get the corresponding AttributeID, and maps to its real value.

**Example 3:** For the original partition (Fig.4(a)) of $D$ (Fig.1(a)), we arrange an ordered vertex list as $\{u_1.u_3,p_1,r_1.r_2\}$ for $P^1_1$ and $\{u_2,p_2,r_3,r_4\}$ for $P^1_2$. In $M_R$, we consecutively allocate VertexID 1 to 5 for the former list, and 6 to 9 for the later one. Given a query with 2 patterns, $Q_{12}$ ?s rdf:type sioc:Reply, ?sioc:content $o$, we first convert all strings into ID using $M_A$ and $M_R$, then retrieve ABIdx$_{PSO}$ according to Predicate rdf:type and Object sioc:Reply, and get a BitVector of [000110011]. Meanwhile, ABIdx$_{comp}$ is retrieved using a key of Predicates sioc:content, and [001110111] is got. By performing bitwise AND between two retrieved BitVector, we get a result BitVector [000110011]. To get the final results, we get all 1-bit corresponding VertexIDs using $M_R$, and retrieve ABIdx$_{POS}$ for each VertexID and Predicates sioc:content, and get a result BitVector each time. The final result can be derived by checking the 1-bit corresponding AttributeID in $M_A$.

### B. R-Triples Partition and Index

Next, we introduce how $D_R$ is partitioned. In order to explain what type of query $Q$ can be implemented without inter-partition operations, we introduce the concept of query coverage. Given a set of partitions $P = \{P_1, \ldots, P_n\}$ generated by a partition strategy $P$ over $D$. If a query $Q$ can be completely answered in $P$ without inter-partition operations, we call that partition $P$ can cover $Q$. If $P$ can completely cover all query graphs in which the shortest path between two arbitrary vertices $< \phi$, we call $\phi$ as the query coverage of $P$. To increase $\phi$, a common way is to introduce triple replication in each $P_i$, $1 \leq i \leq n$. We defines replication factor $\alpha = \frac{|D_A| + \sum_{i=1}^{n} |P_i|}{|D_A|}$ to quantitatively measure such replication. There is a tradeoff between $\phi$ and $\alpha$. For example, $\alpha_{\text{METIS}}$ have $\phi = 0$, $\alpha = 1$. This means although $\alpha_{\text{METIS}}$ does not introduces any replicated triples, it must deal with inter-partition query processing for an arbitrary query. Increase $\phi$ can results in an exponential increase of replicated triples, which led to an extra storage overhead.

To achieve a reasonable $\phi$ with acceptable $\alpha$, in our work, we utilize a $P$ named Undirected One Hop Cover(1-UHC) partition strategy, denoted as $P_{1-UHC}$. $P_{1-UHC}$ follows the idea of undirected 1-hop guarantee introduced by [23]. To elaborate, based an original partition $P^0 = (V^0, E^0) \in P^n$ for $G_R$, let $V_i = \{v \mid (v, v_i) \in E^0 \vee (v, v_i) \in E^0, v_i \in V_i, v \notin V_i\}$ denotes the set of cutting edges in $P^0$. $P_{1-UHC}$ generates expanded partitions $P_i = (V_i, E_i)$ following these steps:

1. Adds all cutting edges $\forall e \in \{e = (v_o, v_b) \in G_R\} \vee (v_a, v_b) \in E_j\}$ to $E_j$;
2. Adds all edges that connect cutting edges $\forall e \in \{e = (v_o, v_b) \in G_R\} \vee (v_a, v_b) \in E_j\}$ to $E_j$;
3. Assigns $\forall v \in V_i \rightarrow V_j$.

$P_{1-UHC}$ differs from [23] in that it solely applied on $G_R$. As $G_R$ is more compact than $G$, this leads to a smaller $\alpha$.
traverse through all implemented in a greedy fashion. We start with selection of a \( Q = \{x_i\} \) corresponding subgraphs in \( T G^Q \) is constructed by transform each \( \bar{Q} \rightarrow \bar{Q}^Q \) and all edges of all complete graph from \( E_R^Q \). We iterate the above process until there are no edges in \( E_R^Q \). Obviously in worst case, \( T G^Q \) can have \( |E_R^Q| \) vertices.

As the basic building block of SGDQ, an \( OP_{sp} \), \( Q_i^o(P_j) \), is the sub-query \( Q_i^m \in Q^p, 1 \leq i \leq m \) on partition \( P_j \in \mathcal{P}_i, 1 \leq j \leq n \). Optimization of \( OP_{sp} \) can greatly enhance the overall query performance. Furthermore, as \( \mathcal{P}_i \) introduce overlapping among partitions, in \( OP_{sp} \) a mechanism is needed to tackle the problem of latent duplications in final results.

Before explain the implementation of \( OP_{sp} \), we first introduce a process, named as filter. The filter uses A-Patterns to prune the matched triple set of a R-Pattern in a \( P_j \). Given a R-Pattern, let \( v_o \) be its Subject vertex, \( v_s \) be its Object vertex. Each vertex may have A-Patterns connect to it, and the candidate set of \( v_o \) and \( v_s \), denoted as \( CR(v_o) \) and \( CR(v_s) \), can be got using CandidateRLVertex operation. For all triples in \( P_j \) that match a R-Pattern, filter eliminates triples that theirs Subject/Object are not contained in \( CR(v_o) \) and \( CR(v_s) \). The set containment check is also based on BitVectors, and can be efficiently implemented in \( O(1) \) time.

For Type-I sub-query, which is a match of simplex triple pattern, \( OP_{sp} \) is implemented in a scan-filter fashion. The scan is a scan of a pattern-specific permutation index. In filter, \( CR(v_s) \) is derived by a bit-wise AND between the CandidateRLVertex result BitVector of \( v_s \) and the partition’s pVector. For \( CR(v_o) \), we distinguish between two cases. If current \( OP_{sp} \) corresponds to the last matched vertex in the match order(discuss in detail in Section [V,A]), denoted as the last \( OP_{sp} \), for brevity, \( CR(v_o) \) is derived by a range selection of the CandidateRLVertex result BitVector of \( v_o \). If \( OP_{sp} \) is not the last \( OP_{sp} \), \( CR(v_o) \) is derived the same way as \( CR(v_o) \). Both range data and pVector is stored as metadata in a partition. By only keep the triples that have Subject in the overlapping-free original partition, duplications in the final results can be avoided.

For Type-II sub-query, we implement \( OP_{sp} \) in a scan-filter-join-verify fashion. In essence, as each \( P_j \) is managed by a standalone RDF-3X triple store, we can utilize the query mechanism provided by RDF-3X for \( OP_{sp} \). More precisely, RDF-3X uses a scan-join mechanism for query processing. A query is decomposed into a set of triple patterns. The candidate set of each triple pattern is generated by a scan of a corresponding permutation index. All candidate sets are joined following an optimized order and get the final query results. We further add a filter operation before join really happens. At filter stage, \( CR(v_o) \) and \( CR(v_s) \) is derived by bit-wise AND between the CandidateRLVertex result BitVector of \( v_o \) and \( v_s \) and the partition’s pVector. The filter introduces A-Patterns to filter irrelevant triple from participate in join operation, this design optimize the original scan-join processing. To avoid final results duplication, we introduce a verify stage right after join process to prune the results of scan-filter-join processing if current \( OP_{sp} \) is the last \( OP_{sp} \). The basic idea behind verify is that it keeps the sub-query results that tend to be generated by this \( OP_{sp} \). This tendency is measured by that the result
graph in $G_R$ have more vertices in the original partition generated by $\text{METIS}$. It is rational on that $\text{METIS}$ follows an iterative coarsen-partition processing scheme, complete graph $K_n,n \geq 4$ are more likely to be treated as whole in each coarsen stage. This means that $K_n, n \geq 4$ are always allocated in a partition. To deal with exceptions, we assume that at most one vertex in a $K_n$ is involved in partition. To illustrate, for $K_3$, which is a triangle, we keep the results of $OP_{sp}$ that have two vertices in its original partition. And for $K_4$, we keep the results that have three vertices in its original partition. This is implemented as an range check of all $\text{VertexID}$ of a result using the range information in the metadata of this partition.

As each Type-II sub-query needs to determine its own query plan, the total time of query plan selection can sum up to a high cost. In RDF query processing, cardinality estimation is difficult for the lack of schema. A heuristic-based query plan selection method for each $OP_{sp}$ follows [24]. Considering the size of data partition, a sub-optimal plan is still acceptable for an $OP_{sp}$.

V. SUMMARY GRAPH DRIVEN QUERY(SGDQ) PROCESSING

In this section, we explain how SGDQ use summary graph match process to direct $OP_{sp}$ processing.

A. Summary Graph Match

According to definition 5 the summary graph match is a special case of subgraph isomorphism problem. This problem can be solved using a generic backtracking method [21]. Algorithm 1 outlines an implementation of backtracking-based summary graph match. It determines a match order of vertices in $TQ^S(L)$, then recursively calls a match procedure to find matched $TQ^S$ in $SG$. The match process directs the processing of $OP_{sp}(Line 6)$, and outputs query results of $Q$ on $G$. As SGDQ follows the general backtracking paradigm, we only explain the specific features used in SGDQ.

GetCandidateRLVertex(Line 2) process all A-Patterns in $Q$ grouped by common Subject. For each query RL-graph vertex $v \in V_Q^R$, GetCandidateRLVertex gets the the candidate vertices in $G_R$ as a BitVector. If $v$ is a variable with no connected A-Patterns, all bits in BitVector(v) are assigned to 1. If $v$ is an URI of a VertexID, the corresponding bit in BitVector(v) are assigned to 1, and 0 elsewhere. Otherwise, a CandidateRLVertex operation is called to process all A-Patterns that connect to $v$.

DetermineMatchOrder(Line 3). In principle, the order is determined as an order of DFS graph traversal in $TQ^S$. Thus the selection of the starting vertex in $TQ^S$ is important. We adopt heuristics that considers the output of GetCandidateRLVertex and the the statistics of $D$. These heuristics are listed as follows, and can be used in combination or separately.

Heuristic 1. Choose the order according to the selectivity of all $Q_i^P \in V_Q^P, 1 \leq i$. The selectivity of $Q_i^P$ is roughly estimated as the smallest candidate set size of $v \in Q_i^P$ that has the smallest candidate set in $V_R$, or the smallest selectivity of Predicates of all R-Patterns in $Q_i^P$. The former can be derived using GetCandidateRLVertex function, and the candidate set of $Q_i^P$ in $SG$ is determined by a range check of all 1-bits in BitVector(v). The later is based on the statistics of $D$, which are pre-computed as metadata. The $Q_i^P \in V_Q^P$ with smallest selectivity is chosen as the starting vertex. After starting vertex is selected, in the DFS traversal that follows, high selective vertex goes first.

Heuristic 2. Given $Q_i^P \in V_Q^P$ that have been arranged in order, if $Q_i^P \in V_Q^P$ is a neighbour of $Q_j^P$ in $TQ^S$, and merge-join can be applied between them, then arrange $Q_j^P$ as the next in order.

Heuristic 3. If Heuristic 1 and 2 can not apply, select Type-II $Q_i^P$ into order with priority.

SubgraphMatch(Line 8-20) is a recursive function. It starts as gets a current query vertex $v$ from order(Line 8), and obtains a set of candidate vertices of $M(v) \in SG$, denoted as $C(v, M(v_{prev}))$, which is the set of neighbour vertices of the previous matched vertices $M(v_{prev})$. The vertex match in $SG$ is judged by vertex label match, and corresponding bit of $M(v) \in C(v, M(v_{prev}))$ in $BitVector(M(v))$ is 1-bit(Line 9).

SubgraphMatch iterates over all candidate vertices $M(v) \in C(v, M(v_{prev}))$ (Line 10-19). In each iteration, SubgraphMatch calls an $Op_{sp}$, i.e. $v(M(v))$, and gets its results $R(v(M(v)))(Line 11)$. If $R(v(M(v))) = \varnothing$, $v$ is removed by setting the corresponding bit in BitVector(v) to 0(Line 13). This avoids unnecessary repeated execution of $Op_{sp}$ that do not satisfy vertex math in Definition 5. For each edge between $v$ and all $v_{prev}$ in $TQ^S$, an $Op_{sp}(v(M(v)), v_{prev}(M(v_{prev})))$ is called to check for edge injection(Line 14). As the search order follows a $TQ^S$ graph traversal order, in most cases $Op_{sp}$ are implemented using hash join. We use order-preserving merge-joins for $Op_{sp}$ if possible, considering Heuristic 2 in DetermineMatchOrder. If the result set

---

3Considering the existence loop edges for each vertex $SG$, the neighbours contains the vertex itself.
Algorithm \( R(v(M(v), v_{prev}(M(v_{prev})))) \) is not \( \emptyset \), the join results are cached in a structure \( S \). Line 17. \( S \) enables backtracking to previous matched vertex, and the cached intermediate results in \( S \) can be reused for all vertices in the candidate region \( C(v, M(v_{prev})) \).

Next, SubgraphMatch recursively calls another SubgraphMatch with an incremental in depth \( d \) (Line 18). If \( d \) equals to the number of vertex in \( T^Q \), this means a match of \( T^Q \) is generated, and currently cached \( R \) can be output as part of final results (Line 7). If \( Op_{pj} \) returns \( \emptyset \), SubgraphMatch iterate to the next candidate if exists, otherwise backtrack to the next candidate vertex in \( S \) for \( v_{prev} \in T^Q \) in order.

Property 2 (Query Completeness and Non-Redundancy): Let \( R_{SGDQ} \) and \( R \) be the result set of generated by \( Q \) on \( D \) and SGDQ respectively. If \( r \in R \), then \( r \in R_{SGDQ} \) holds.

This property can be proved by contradiction. We omit the details in this paper.

B. Cost Analysis

Given \( D_A \) and \( D_R \) as \( P \) of \( n \) partitions. For a query \( Q \), assume that in \( Q \) there are totally \( t_1 \) Pattern-I A-Patterns, \( t_{II} \) Pattern-II A-Patterns, and \( Q \) is decomposed into \( m \) sub-queries, of which \( m_1 \) Type-I, and \( m_1 \) Type-II. The average cost of Type-I and Type-II sub-query are \( \text{Cost}(Op_{p}) \) and \( \text{Cost}(Op_{pj}) \) respectively.

We first give the cost analysis of A-Patterns. As there are total \( |D_A| \) BitVectors, the cost of CandidateRLVertex operation is the time complexity of index lookup using a composite key of \( p \) and \( o \), which is \( O(\log|D_A|) \). Similarly, the cost of RetrieveAttribute operation is \( O(\log|V_R|) \). Consequently the aggregated cost of all A-Patterns processing sum up to \( O(t_1 \cdot (|D_A|) + t_{II} \cdot \log|V_R|) \).

The R-Patterns processing follows summary graph matching paradigm. Conventional optimization methods of subgraph matching are focused on establishing graph topological indexes to effectively prune candidate vertices. Such optimizations can not work for Algorithm [1] as we still needs to actually take joins to determine whether two partitions are joinable. Thus exhaustive exploration of all feasible neighbours is adopted in Algorithm [1]. This is acceptable as \( S \) is small. To this end, In worst case SGDQ needs \( \sum_{i=1}^{m-1} |V_S|^i \) times of \( Op_{p} \) and \( |E_S| \cdot \sum_{i=1}^{m-1} |V_S|^i \) times of \( Op_{pj} \). The cost of R-Patterns processing can sum up to \( \sum_{i=1}^{m-1} |V_S|^i \cdot \text{Cost}(Op_{p}) + (|E_S| \cdot \sum_{i=1}^{m-1} |V_S|^i) \cdot \text{Cost}(Op_{pj}) \).

SGDQ only needs to maintain intermediate result at each recursive call, which is generated on \( m \) partitions. This significantly reduces the cost to keep vast amount of immediate results generated by scans and joins on large dataset.

VI. RELATED WORKS

RDF query optimization has attracted interests from both industrial and research domain, and considerable methods, either relational or graph-based, centralized or distributed, have been proposed. In this section, we only list the most related to our approach and compare them with ours.

Graph-based approaches. RDF query in native graph storage is implemented as subgraph matching. Graph pattern matching has been extensively studied [21], [25]. Following these, [9], [10] use fine-grained graph exploration for RDF query. This need to maintain the entire graph in memory. [8], [11] introduce structural summaries of graph as disk-based structural index. Matched results are generated using a filter-and-refinement strategy. SGDQ constructs a summary graph based on graph partitions, which is most similar to [12]. SGDQ differs from [12] in that the summary graph is used in SGDQ for arrange sub-query processings, while it is used in [12] for pruning unnecessary data before join processing.

Relational approaches. A naïve way to manage RDF data is adopting relational database to store all triples in a single 3 column table. Query is implemented in a scan-join fashion [12], [13], [26]. Optimizations of naïve approach are mainly from three aspects. i) Using permutation index to make join and scan themselves efficient. [15] first introduce the concept of permutation index. Hexastore [26] exhaustively indexes all SPO permutations and stores them directly in sorted B+-tree. RDF-3X [13] further introduces extra indexes of all SPO attributes projections. Such permutation indexes enables efficient range scan and merge-join at the cost of data redundancy. SGDQ extensively use permutation indexes in manage each partitions. ii) Reducing the number of joins needed. Property table [27] clustering relevant predicates together in a table schema. This helps in eliminating s-s joins, but requires a previous knowledge about query. BitMat [16] represents property tables as 3 dimension bitmaps, and replace joins with bitwise operations which share common elements at a dimension. SGDQ is different in that it classifies triples and adopts bitmaps only for A-Triples. This eliminates all s-s joins related to A-Patterns, and avoids complex considerations of other types of joins when using bitmap index. iii) Filtering out irrelevant inputs of join. RDF-3X adopted a technique named SIP(Sideway Information Passing) to dynamically determine the unnecessary data blocks that can be skipped during scan [28]. RG-index [29] introduces a relational operator filter that using frequent graph pattern to prune triples that can not match. SGDQ uses filter in each \( Op_{p} \) and shares the same objective as [28], [29], but it uses the results of A-Patterns processing in filtering.

Data partition are widely used in shared-nothing distribute RDF systems [23], [12]. They follow the principles of evenly distribute data to each computing node so that each partitions can be processed in parallel to boost the overall query performance. The number of partitions are generally equal to the number of computing nodes, and the goal of optimization is to design effective partition strategy to minimize the cost of distribute inter-partition joins. To this end, the most common adopted partition strategies are edge-based hash [30] and graph partition [23]. SGDQ adopts partition in a totally different way. It partition data in order to decompose original query into a series of light-weight processing, the number of partitions depends on the dataset size.

VII. EVALUATION

In this section, we empirically evaluate the RDF query processing performance on two kinds of large-scale benchmark data. The goals of our evaluations are i) We show the correctness and the competitive performance of SGDQ on basic benchmark queries (contain at most 6 triple patterns) over
the state-of-the-art triple stores (Section VII-C); ii) We show the superior performance of SGDQ on complex queries that involve up to 16 triple patterns, and analyze the effect of our methods (Section VII-D VII-E).

A. Environment, Competitors and Datasets

All experiments were performed on a server with Debian 7.4 in 64-bit Linux kernel, two Intel Xeon E5-2640 2.0GHz processors and 64GB RAM. The query response time was measured in milliseconds. SGDQ is developed in C++, and is compiled using GCC-4.7.1 with -O3 optimization flag. We used METIS 5.1.4 to get the original partition of RDF graph, and RDF-3X 0.3.8 for partition management and support of sub-query processing. For fair comparison, we also used the dictionary to convert between the string value of s,o,p and the identifiers used in processing, and presented results in its original string value form.

Competitors. There exists a wide choice on the state-of-the-art triple store. We chosen three publicly available and representative ones. We list these competitors as:

RDF-3X maintains all possible permutation of s,o,p as indexes, and use B+-tree to facilitate index lookup. Query of a single pattern is implemented as an index scan operation, and merge join are extensively leveraged to join between two patterns. Together with index-specific query optimization techniques [13], [28], RDF-3X remains a competitive state-of-the-art triple stores.

TripleBit organizes data in a compact bit matrix, and vertically partition the matrix into sub-matrices following predicates dimension and sorted subject/object dimension. Such compact data representation and partition can minimizes the size of intermediate results, thus theoretically results in a query performance promotion [14].

Virtuoso is a commercial-of-the-shelf column store that support RDF management. It exploit existing relational RDBMS techniques and add functionality to deal with features specific to RDF data. It also adopted bitmap index to take advantage of the property that many triple share the same predicate and object [11].

Dataset. We adopted two different kinds of benchmarks.

LUBM is a widely used benchmark in both academical researches and industrial applications. It provides a university database where the components, such as university, departments, professors, students etc, can be polynomially generated. The size of dataset scales according to the number of universities. We generated two dataset, they are identified as LUBM-8000 and LUBM-20480 according to the number of universities. As all components in the data are generated in a proportional fashion, the RDF graph of a LUBM dataset can be regarded as a random graph.

SNIB provides the semantic dataset of a twitter-like social network, which includes resources like users, post, reply, tags and comments, etc. The dataset scales according to the number of users. We generate a SNIB dataset of 15000 users using S3G2 [32], identified as SNIB-15000. As it models a social network, the RDF graph of SNIB-15000 is a power-law graph.

Remind that to support the original LUBM queries, the standard way is to inference on RDF Schema subclasses and subproperties. We used the inference engine in Virtuoso 7 in our experiment, and the inferred triples were also managed by each triple store. The statistics of dataset are listed in Table II.

| Dataset     | # Triples | # Inferred | RL-Graph | #p |
|-------------|-----------|------------|----------|----|
| LUBM-8000   | 1,970,942,330 | 186,030,725 | 173,808,051 | 529,614,087 | 18 |
| LUBM-20480  | 4,844,909,072 | 2,024,718,666 | 444,851,202 | 1,355,446,183 | 48 |
| SNIB-15000  | 1,503,766,205 | 27,618,900 | 125,779,052 | 875,721,431 | 444,851,202 |

Queris. For LUBM dataset, we used the 14 queries provided by LUBM benchmark. Furthermore, we manually constructs 6 complex queries on LUBM and and 7 queries on SNIB dataset. These complex queries contains more triple patterns than the original ones. All Queries are listed in the Appendix.

Consider that the cache mechanism of operating system have a profound impact in query execution, we measure the performance using both cold cache mode and warm cache mode. In cold cache mode, we purged the OS file system cache each time before running a query[10]. Each query was executed in cold cache mode and warm cache mode for 10 times in a successive manner. The result in each mode was reported as the arithmetic mean and was round to millisecond[11].

B. Effects of Partitions

We adopted n = 500, i.e. the RL-graph of each dataset was partitioned into 500 subgraphs using METIS. The effects of P_METIS and P_1-UHC is given in Table III manifested as the number of cutting edges and replication factor α respectively. By managing A-Triples individually, P_1-UHC have far smaller duplicated triples compared with that of the un-two strategy show in [23].

We list the physical storage characteristics in Table III. Basically, RDF-3X costs the most space, as it needs to maintain all 6 SPO permutation indexes and 9 aggregated indexes of binary and unary projection. Even though SGDQ use RDF-3X to manage all partitions of R-Triples, but the bitmap index of A-Triples contributes to a more compact storage. Consider the proportion of A-Triples in a dataset(as shown in Table III), SGDQ shows a more efficient storage compared with RDF-3X.

C. LUBM benchmark queries

We report the response time of query on LUBM-8000 and LUBM-20480 dataset in Table |IIIa and Table |IIIb respectively. Generally speaking, SGDQ achieve a comparable query performance using both cold cache mode and warm cache mode. In cold cache mode, we purged the OS file system cache each time before running a query[10]. Each query was executed in cold cache mode and warm cache mode for 10 times in a successive manner. The result in each mode was reported as the arithmetic mean and was round to millisecond[11].

[10]In Debian OS, we purged its file system cache using commands: /bin/sync; echo 3 > /proc/sys/vm/drop_caches

[11]Actually, in warm cache mode, a query was executed 11 times consecutively. The first execution was for cache warm-up, and its result was removed from averaging.

[12]Virtuoso 7 manage all dataset in a single database instance. The storage characteristic of Virtuoso 7 is measured by the increased database file size after each dataset is loaded.
TABLE III: Benchmark query response time on LUBM dataset (in milliseconds).

| Dataset   | L1  | L2  | L3  | L4  | L5  | L6  | L7  | L8  | L9  | L10 | L11 | L12 | L13 | L14 |
|-----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| RDF-3X    | 377 | 1,034,395 | 372 | 456 | 377 | 2,308,477 | 581 | 1,040 | 517,775 | 385 | 497 | 440 | 24,179 | 1,828,117 |
| Virtuoso7 | 7   | 25,193 | 3   | 25 | 5 | 2,450,552 | 4 | 628 | 225,441 | 2 | 5 | 35 | 4,381,397 |
| TripletBit | 115 | 91,065 | 2 | 625 | 51 | 3,891,194 | 31 | 109* | 483,726 | 0.3 | 3 | 5 | 620 | 3,122,775 |
| SGDQ      | 92  | 11,790 | 42 | 56 | 49 | 924,555 | 102 | 325 | 94,702 | 22 | 63 | 92 | 271 | 592,374 |

Warm Cache.

| Dataset   | L1  | L2  | L3  | L4  | L5  | L6  | L7  | L8  | L9  | L10 | L11 | L12 | L13 | L14 |
|-----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| RDF-3X    | 2   | 910,820 | 2   | 5 | 7 | 2,196,855 | 7 | 439 | 109,162 | 2 | 186 | 19 | 6,174 | 1,553,394 |
| Virtuoso7 | 2   | 24,033 | 1 | 3 | 4 | 2,381,086 | 2 | 427 | 223,551 | 1 | 21 | 2 | 1,592 | 1,568,623 |
| TripletBit | 0.2 | 39,137 | 0.3 | 3 | 3 | 3,823,302 | 2 | 102* | 178,381 | 0.2 | 1 | 0.4 | 582 | 3,045,274 |
| SGDQ      | 2   | 0.815 | 1 | 2 | 3 | 892,722 | 2 | 221 | 76,118 | 2 | 22 | 64 | 573,189 |

#Results

| L1  | L2  | L3  |
|-----|-----|-----|
| 4   | 2528 | 6   |

(a) LUBM-8000 dataset.

(b) LUBM-20480 dataset.

TABLE IV: Partition characteristics

| Dataset | # cutting edges |
|---------|----------------|
| RDF-3X  | 2,131,709 |
| Virtuoso7 | 41,199 |
| TripletBit | 124 |
| SGDQ    | 76 |

TABLE V: Physical storage characteristics (in GB).

| Dataset | Raw N3 file | RDF-3X | Virtuoso7 | TripletBit | SGDQ |
|---------|-------------|--------|-----------|------------|------|
| LUBM-8000 | 340 | 23,624,351 |
| LUBM-20480 | 61,518,672 | 1.21 |
| SNIB-15000 | 58,823,336 | 1.32 |

SGDQ outperforms all competitors by 1.7(L2 for Virtuoso 7 on LUBM-20480) to 98(L2 for RDF-3X on LUBM-20480) times for cold cache, and by 2.3(L9 for TripleBit on LUBM-8000) to 193(L2 for RDF-3X on LUBM-20480) times for warm cache. For L6 and L14, SGDQ is efficient in that they only need to look up the bitmap indexes of A-Triples, the cost is due to the dictionary operation for string and ID conversation. For L2 and L9, theirs RL-graph is a triangle. This means that they can be answered without inter-partition processing, and theirs $T^Q_{G}$ have only one vertex. Intuitively, the sum of all $OP_{i}$ are far better than execute the query as a whole.

D. Complex queries on LUBM

As standard LUBM benchmark queries can not fully evaluate the performance of SGDQ, we further design 3 more complex queries on LUBM dataset. These queries show a more complex structural information. The $G^{Q}_{R}$ of L15-L17 are shown as Figure 6. We make additional 3 queries by adding a highly selective A-Pattern to each of them, and get $L15_{sel}$-$L17_{sel}$ accordingly. These queries are listed in Appendix A. Intuitively, the $T^Q_{G}$ of L15-L17 have 3,4,4 vertices and 2,3,4 edges individually.

We report the response time of query on LUBM-8000 and LUBM-20480 dataset in Table [VI] in both cold cache and warm cache mode. In fact, for complex query on large-scale dataset, this evaluation shows that only Virtuoso 7, which is a commercial-off-the-shelf system, is qualified. RDF-3X fails performance in comparison to its competitors using standard benchmark queries.

Query L1, L3, L4, L5, L7, L8, L10, L11 exhibit the same characteristics. They all have a high-selective A-Pattern, and their result sets are quite small and independent of the dataset size. Among them, L4 is a bit more complex, it contains 3 Pattern-II A-Patterns. SGDQ needs an extra bitmap lookup to retrieve the variable of these A-Pattern. Due to their simple nature, they cannot benefit from SGDQ optimizations. In fact, the response time of all these queries for both SGDQ and all competitors are within 1 second for both cold cache and warm cache. It is hard for a user to feel the difference. Although Query L13 also have a high-selective A-Pattern, the result set of L13 increase as the dataset size increase. For cold cache query, RDF-3X needs to scan the large indexes and cache the data at each query time. SGDQ outperforms RDF-3X by 4.1(L1 on LUBM-8000) to 168(L13 on LUBM-20480) times. This proves the effectiveness of SGDQ in avoiding the cost of scan large indexes by manage and query data in partitions. As for warm cache query, the candidate triples of high selective patterns can be held in cache. As SGDQ introduce extra cost of bitmap search and partition selection, SGDQ has no advantages over competitors.

For the time consuming queries, i.e. L2, L6, L9 and L14, SGDQ outperforms all competitors by 1.7(L2 for Virtuoso 7 on LUBM-20480) to 98(L2 for RDF-3X on LUBM-20480) times for cold cache, and by 2.3(L9 for TripleBit on LUBM-8000) to 193(L2 for RDF-3X on LUBM-20480) times for warm cache. For L6 and L14, SGDQ is efficient in that they only need to look up the bitmap indexes of A-Triples, the cost is due to the dictionary operation for string and ID conversation. For L2 and L9, theirs RL-graph is a triangle. This means that they can be answered without inter-partition processing, and theirs $T^Q_{G}$ have only one vertex. Intuitively, the sum of all $OP_{i}$ are far better than execute the query as a whole.

D. Complex queries on LUBM

As standard LUBM benchmark queries can not fully evaluate the performance of SGDQ, we further design 3 more complex queries on LUBM dataset. These queries show a more complex structural information. The $G^{Q}_{R}$ of L15-L17 are shown as Figure 6. We make additional 3 queries by adding a highly selective A-Pattern to each of them, and get $L15_{sel}$-$L17_{sel}$ accordingly. These queries are listed in Appendix A. Intuitively, the $T^Q_{G}$ of L15-L17 have 3,4,4 vertices and 2,3,4 edges individually.

We report the response time of query on LUBM-8000 and LUBM-20480 dataset in Table [VI] in both cold cache and warm cache mode. In fact, for complex query on large-scale dataset, this evaluation shows that only Virtuoso 7, which is a commercial-off-the-shelf system, is qualified. RDF-3X fails
to return for L17 on LUBM-8000 and L16, L17 on LUBM-20480. TripleBit returns with wrong number of results for all queries except L17下调. For non-selective queries L15-17, SGDQ consistently outperform others in both warm cache and cold cache mode.

To further investigate this problem, we record the total counts of $OP_{sp}$ operations for each query processing. We can see that L15 generates more $OP_{sp}$, as each sub-queries of L15 is a simple pattern query. SGDQ implements such $OP_{sp}$ and $OP_{ij}$ at the magnitude of approximately 1 ms, this contributes to the overall query performance promotion. Compare the result of L15 on LUBM-8000 and LUBM-20480, we can see that as the data size increase, the average cost of $OP_{sp}$ increase accordingly. This is due to expansion of partition size. Although L17 generates far more less $OP_{sp}$, but most of its $OP_{sp}$ are triangles, and the average execution time is at a magnitude of 10 to 20 ms. This evaluation shows the efficiency of $OP_{sp}$ in LUBM dataset.

E. Complex queries on SNIB

We also evaluated the query performance on a SNIB benchmark dataset, which is generated as a social network of 15000 users. SNIB dataset presents a more connected graph structure compared with LUBM dataset, thus we can introduce more complex queries to evaluate the effect of SGDQ. We define 7 SNIB queries as listed in Appendix B. Their $G_{R}$ features are shown as Figure 6. We report the response time of query on SNIB-15000 dataset in Table VII. All queries were executed in warm cache mode. As TripleBit is inclined to to return wrong number of results, we do not select it in this evaluation. In essence, SGDQ consistently outperforms all competitors as the queries are more complex.

To elaborate, query $S1$ and $S4$ shows a similar characteristic. All the $OP_{sp}$ are Type-I sub-query. This cause the number of $OP_{sp}$ grows exponentially with the number of edges in $G_{R}^Q$. SGDQ shows a superior performance due to the efficiency of $OP_{sp}$. Query $S3$ and $S7$ have a triangle $G_{R}^Q$, and their $TG_{R}^Q$ contains only one vertex. They differ in that $S7$ does not have a selective A-Pattern as $S3$. $S7$ needs to explore all partitions. Like in LUBM evaluation, SGDQ also performs well on such queries. Query $S2$, $S5$ and $S6$ feature the processing of complex $OP_{ij}$ between Type-II sub-queries. RDF-3X performs the worst due to excessive number of joins, even fails for $S5$. The superior performance of SGDQ on these queries shows that all $OP_{ij}$ are also efficiently processed in SGDQ. These complex queries benefit from the simplification of query processing using query decomposition and data partition.

VIII. Conclusion

We have presented SGDQ, a Summary Graph Driven Query framework for RDF query processing. SGDQ is based on efficient execution of sub-queries on partitioned data. It utilizes the effectiveness of subgraph matching in representing the inter-partition query processing, and avoids the problem of maintaining huge intermediate results in canonical relational query methods. To enable SGDQ, we introduced specialized physical data layout and implemented efficient operations in SGDQ. We have performed an extensive evaluation on the query performance of SGDQ compared with three competitive RDF systems. We also noticed that in current work, the exhaustive exploration of neighbourhood in summary graph match introduces many useless inter-partition joins. In the future, we plan to investigate the mechanism to prune unnecessary partitions by maintaining fine-grained inter-partition data in
As you can see, the text is a mixture of code snippets and natural language content. To make it more readable, I will extract the natural language text and provide a plain text representation of it:

### REFERENCES

[1] “DBpedia,” http://wiki.dbpedia.org/  
[2] “YAGO,” www.mpi-inf.mpg.de/yago/  
[3] “Freebase,” https://www.freebase.com/  
[4] “The Friend of a Friend (FOAF) project,” http://www.foaf-project.org/  
[5] “Fackbook Open Graph,” http://ogp.me/  
[6] “Bio2RDF, linked data for the life sciences,” http://bio2rdf.org/  
[7] “The Friend of a Friend (FOAF) project,” http://www.foaf-project.org/  
[8] “Jena Property Table Implementation,” in T. Neumann and G. Weikum, “Scalable join processing on very large RDF graphs,” in PVLDB, vol. 6, no. 8, pp. 265–276, 2013.  
[9] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, “Taming subgraph isomorphism for RDF query processing,” PVLDB, vol. 8, no. 11, pp. 1238–1249, 2015.  
[10] T. Tran, G. Gadewadikar, and S. Rudolph, “Managing structured and semistructured RDF data using structure indexes,” IEEE Trans. Knowl. Data Eng., vol. 25, no. 9, pp. 2076–2089, 2013.  
[11] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, “TriAD: A fast and compact system for large scale RDF data,” in PVLDB, vol. 6, no. 7, pp. 517–528, 2013.  
[12] A. Lathar and S. Decker, “Optimized index structures for querying RDF from the web,” in LA-WEB, 2005, pp. 10 pp.  
[13] M. Stocker, A. Seaborn, A. Bernstein, C. Kiefer, and D. Reynolds, “Sparql basic graph pattern optimization using selectivity estimation,” in WWW, 2008, pp. 595–604.  
[14] A. Gubichev and T. Neumann, “Exploiting the query structure for efficient join ordering in SPARQL queries,” in EDBT, 2014, pp. 439–450.  
[15] “SPARQL,” http://www.w3.org/TR/rdf-sparql-query/  
[16] M. Aria, J. D. Fernandez, and M. A. Martinez-Prieto, “An empirical study of real-world sparql queries,” in USEWOD@WWW, 2011.  
[17] J. Lee, W.-S. Han, R. Kasprowski, and J.-H. Lee, “An in-depth comparison of subgraph isomorphism algorithms in graph databases,” in PVLDB, vol. 6, 2013, pp. 133–144.  
[18] G. Karypis and V. Kumar, “METIS – unstructured graph partitioning and sparse matrix ordering system, version 2.0,” Tech. Rep., 1995.  
[19] J. Huang, D. J. Abadi, and K. Ren, “Scalable SPARQL querying of large RDF graphs,” in PVLDB, vol. 4, no. 11, 2011, pp. 1123–1134.  
[20] P. Tsaiyalamanis, L. Sidirouglos, I. Fundulaki, V. Christophides, and P. Boncz, “Heuristics-based query optimisation for sparql,” in EDBT, 2012, pp. 324–335.  
[21] J. Cheng, J. Y. Yu, and P. S. Yu, “Graph pattern matching: A join/semi-join approach,” IEEE Trans. on Knowl. and Data Eng., vol. 23, no. 7, pp. 1006–1021, 2011.  
[22] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: sextuple indexing for a Structured Web data management,” PVLDB, vol. 1, no. 1, pp. 1008–1019, 2008.  
[23] K. Wilkinson, “Jena Property Table Implementation,” in SSWS, 2006, pp. 35–46.  
[24] T. Neumann and G. Weikum, “Scalable join processing on very large RDF graphs,” in SIGMOD, 2009, pp. 627–640.  
[25] K. Kim, B. Moon, and H. Kim, “R3F: RDF triple filtering method for efficient SPARQL query processing,” World Wide Web, vol. 18, no. 2, pp. 317–357, 2015.  
[26] A. Gubichev and T. Neumann, “Exploiting the query structure for efficient join ordering in SPARQL queries,” in EDBT, 2014, pp. 439–450.  
[27] T. Tran, G. Gadewadikar, and S. Rudolph, “Managing structured and semistructured RDF data using structure indexes,” IEEE Trans. Knowl. Data Eng., vol. 25, no. 9, pp. 2076–2089, 2013.  
[28] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, “TriAD: A fast and compact system for large scale RDF data,” in PVLDB, vol. 6, no. 7, pp. 517–528, 2013.  
[29] A. Lathar and S. Decker, “Optimized index structures for querying RDF from the web,” in LA-WEB, 2005, pp. 10 pp.  
[30] M. Stocker, A. Seaborn, A. Bernstein, C. Kiefer, and D. Reynolds, “Sparql basic graph pattern optimization using selectivity estimation,” in WWW, 2008, pp. 595–604.  
[31] A. Gubichev and T. Neumann, “Exploiting the query structure for efficient join ordering in SPARQL queries,” in EDBT, 2014, pp. 439–450.  
[32] “SPARQL,” http://www.w3.org/TR/rdf-sparql-query/  
[33] M. Aria, J. D. Fernandez, and M. A. Martinez-Prieto, “An empirical study of real-world sparql queries,” in USEWOD@WWW, 2011.  
[34] J. Lee, W.-S. Han, R. Kasprowski, and J.-H. Lee, “An in-depth comparison of subgraph isomorphism algorithms in graph databases,” in PVLDB, vol. 6, 2013, pp. 133–144.  
[35] G. Karypis and V. Kumar, “METIS – unstructured graph partitioning and sparse matrix ordering system, version 2.0,” Tech. Rep., 1995.  
[36] J. Huang, D. J. Abadi, and K. Ren, “Scalable SPARQL querying of large RDF graphs,” in PVLDB, vol. 4, no. 11, 2011, pp. 1123–1134.  
[37] P. Tsaiyalamanis, L. Sidirouglos, I. Fundulaki, V. Christophides, and P. Boncz, “Heuristics-based query optimisation for sparql,” in EDBT, 2012, pp. 324–335.  
[38] J. Cheng, J. Y. Yu, and P. S. Yu, “Graph pattern matching: A join/semi-join approach,” IEEE Trans. on Knowl. and Data Eng., vol. 23, no. 7, pp. 1006–1021, 2011.  
[39] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: sextuple indexing for a Structured Web data management,” PVLDB, vol. 1, no. 1, pp. 1008–1019, 2008.  
[40] K. Wilkinson, “Jena Property Table Implementation,” in SSWS, 2006, pp. 35–46.  
[41] T. Neumann and G. Weikum, “Scalable join processing on very large RDF graphs,” in SIGMOD, 2009, pp. 627–640.  
[42] K. Kim, B. Moon, and H. Kim, “R3F: RDF triple filtering method for efficient SPARQL query processing,” World Wide Web, vol. 18, no. 2, pp. 317–357, 2015.