Efficient Indices using Graph Partitioning in RDF Triple Stores

Ying Yan [†♯], Chen Wang [‡], Aoying Zhou [†§], Weining Qian [§], Li Ma [‡], Yue Pan [‡]

[†]*Fudan University*, [♯]*SAP Research Center China* ying.yan@sap.com

[‡]*IBM China Research Laboratory* {chwang, malli, panyue}@cn.ibm.com

[§]*East China Normal University* {ayzhou, wnqian}@sei.ecnu.edu.cn

Abstract— **With the advance of the Semantic Web, varying RDF data were increasingly generated, published, queried, and reused via the Web. For example, the DBpedia, a community effort to extract structured data from Wikipedia articles, broke 100 million RDF triples in its latest release. Initiated by Tim Berners-Lee, likewise, the Linking Open Data (LOD) project has published and interlinked many open licence datasets which consisted of over 2 billion RDF triples so far. In this context, fast query response over such large scaled data would be one of the challenges to existing RDF data stores. In this paper, we propose a novel triple indexing scheme to help RDF query engine fast locate the instances within a small scope. By considering the RDF data as a graph, we would partition the graph into multiple subgraph pieces and store them individually, over which a signature tree would be built up to index the URIs. When a query arrives, the signature tree index is used to fast locate the partitions that might include the matches of the query by its constant URIs. Our experiments indicate that the indexing scheme dramatically reduces the query processing time in most cases because many partitions would be early filtered out and the expensive exact matching is only performed over a quite small scope against the original dataset.**

I. Introduction

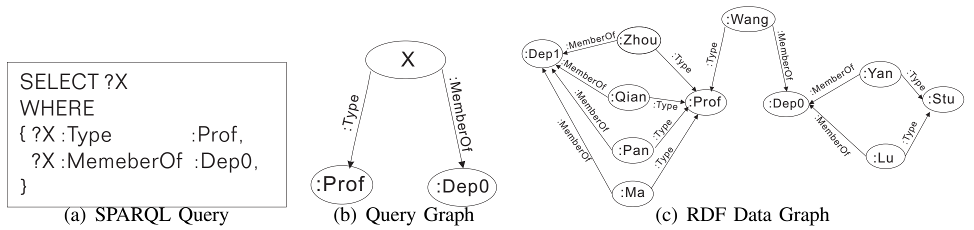
The Semantic Web was designed to enable data integration and sharing across different applications by the World Wide Web Consortium (W3C). With the advance of the Semantic Web, varying RDF data were increasingly generated, published, queried, and reused via the Web. For example, the DBpedia, a community effort to extract structured data from Wikipedia articles, broke 100 million RDF triples in its latest release. Initiated by Tim Berners-Lee, likewise, the Linking Open Data (LOD) project has published and interlinked many open licence datasets which consisted of over 2 billion RDF triples so far. In this context, fast query response over such large scale of data would be one of the challenges to existing RDF data stores.

Mature relational database was regarded as a good basis for RDF store to manage large scale of triples. In general, those RDF stores included a triple table where a RDF triple would be persisted as a database row. To reduce the cost in space, the URIs and literals of the triples might not be stored in the triple table but could be identified by the links to URI/literal-mapping tables. At present, Jena2 [1], Sesame [2], SOR [3] and Oracle-RDF [4] were typical RDB based RDF stores. In those systems, they contained a few unique features to improve the query performance respectively. For example, property table was proposed to store patterns of RDF statements in Jena2. A n-column property table stored $n-1$ statements (1 column per property), which was efficient in terms of storage and access. SOR utilized a DB2 feature, MDC (Multi Dimensional Clustering) table, to relocate instances (except for typeOf assertions) in physical layer. MDC mimics a multi-dimensional cube by using a physical region for each unique combination of dimension attribute values. A physical block could be addressed by block indices, a higher granularity indexing scheme. Oracle-RDF was a RDF store embedded in Oracle RDBMS, which enabled users to embed a RDF query into another SQL query retrieving non-RDF data and optimized the query in the round.

In this paper, we proposed a novel triple indexing scheme to help RDF query engine fast locate the instances within a small scope. By considering the RDF data as a graph, we partition the graph into many subgraphs and store them individually, over which a signature tree is built up to index the URIs. When a query passes, the signature tree index is used to fast locate the partitions that might include the matches of the query by its constant URIs. Our experiments indicated that the indexing scheme dramatically reduced the query time in most cases because many partitions would be early filtered out and the costly exact matching was only performed over a quite small scope against the original dataset. An intuitive example is given as follows to illustrate the idea and our motivation.

Given a SPARQL [5] query of Figure 1(a) which logically equals to the query graph as shown in Figure 1(b), we issue it over the RDF data whose data graph is shown in Figure 1(c). Suppose that the triples stored in the table are not well sorted intentionally in advance and no index is created as well. After the SPARQL query is translated and submitted to relational database, query engine might determine a nested-loop self-join on the column of subject accordingly. Roughly estimated as shown in Figure 2(a), the join will cost $5 \times 3 = 15$ times of string comparison and $5 + 3 = 8$ I/O times. Once the query and data are much more complex, the cost will increase dramatically. Observing the same example from perspective of graph model as shown in Figure 1(c), even though the triples (or edges bridging two vertices) like <`:Zhou :Type :Prof`> are totally disconnected with those including <`:Yan :MemberOf :Dep0`>, and <`:Lu :MemberOf :Dep0`>,

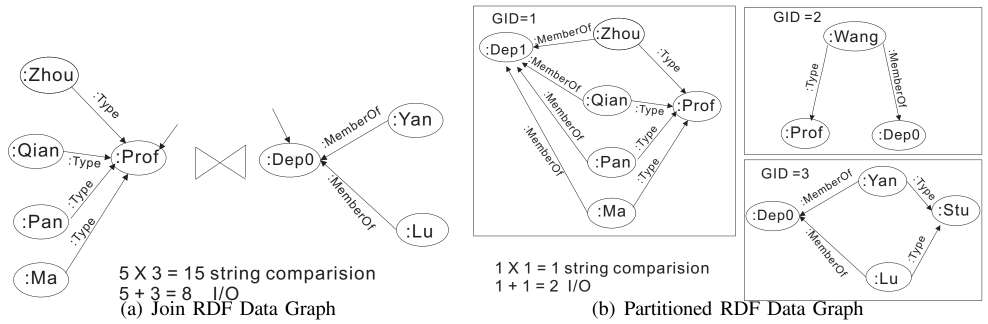
1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.216



```
SELECT ?X
WHERE
{ ?X :Type          :Prof,
  ?X :MemeberOf  :Dep0,
}
```

(a) SPARQL Query (b) Query Graph (c) RDF Data Graph

Fig. 1. A Motivating Example



(a) Join RDF Data Graph (b) Partitioned RDF Data Graph

Fig. 2. Our Proposed Idea

they are still selected to compare with each other by string. Here, we are motivated to decrease the join cost by considering the fact that the query results must be connected subgraphs embedded in the RDF data graph and able to match query graph. Therefore, we first partition the data graph into three groups and add an additional column for the triple table to store the group (subgraph) identities illustrated in Figure 2(b). Thus, we only perform the join operations within each group, and finally merge the results together and eliminate redundancy. For this case, only two triples of `<:Wang :Type :Prof>` and `<:Wang :MemberOf :Dep0>` are selected as candidates in group 2. The join cost is reduced to $1 \times 1 = 1$ times of string comparison and $1 + 1 = 2$ I/O times. Furthermore, to fast locate the candidate groups probably containing the query graph, we build up signatures for all partitions and filter them using the query graph signature.

The remaining of the paper is organized as follows. Graph partition and overlapped storage techniques are presented in Section II. Section III introduces our signature index structure. Query processing procedure is shown in Section IV. Finally, Section V concludes our techniques.

II. Graph Partitioning and Storage

The first intention of our proposed method is to reduce the cost of self-join on vertical database structure via replacing with sub-self-joins within the independent narrow scopes of triples (or graphs). Intuitively, graph partitioning technologies like Metis [6] can help divide the large scale graph of RDF data into multiple small pieces. Many of these efficient algorithms and approaches have been contributed in the areas of parallel computing, VLSI design, social network analysis and community discovery. The graph partitioning method itself is not the focus of our paper. We just exploit it to prepare data over which our following techniques can exert their maximum functions to improve RDF query performance.

General graph partitioning methods always divide a graph into multiple non-overlapped subgraphs by cutting their connecting edges. To secure the query completeness and soundness, however, we have to keep the information of cutting edges and partition vertices in a way. Therefore, we would partition the original RDF data graph into overlapped subgraphs, and add one column GID, i.e., subgraph id, into the triple table. Let us begin with the definitions:

A RDF graph is a directed labeled graph $G = \{(v_i, e_j, v_k)| v_i, v_k \in V, e_j \in E\}$, where V denotes the vertex set and E represents edge set. We can also denote it as $G = (V, E)$ for simplicity. Assume that G is divided into n partitions $P_1 = (V_1, E_1), P_2 = (V_2, E_2), ..., P_n = (V_n, E_n)$ by partitioning the vertices into n groups, then:

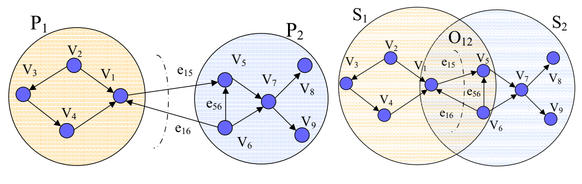
- *Borderline vertex set*: $V_b = \{v_i| < v_i, e, v_j >, v_i \in P_i, v_j \in P_j, i \neq j\}$.
- *Related edge set*: $E_r = \{e| < v_i, e, v_j >, v_i, v_j \in V_b\}$.
- *Overlapped graph O_{ij} between P_i and P_j*: $O_{ij} = (V_b, E_r)$.

Definition 1: RDF subgraph. Given $i, j \in [1, n], i \neq j$, a RDF subgraph corresponding to partition P_i and P_j is defined as

$$S_i = \{(v_x, e_y, v_k)|v_x, v_k \in V_i \cup V_{O_{ij}}, e_y \in E_i \cup E_{O_{ij}}\} \quad (1)$$

As demonstrated in Figure 3, the original graph is first divided into two non-overlapped partitions P_1 and P_2 by cutting the edges e_{15} and e_{16}. v_1, v_5 and v_6 belong to *Borderline vertices*. e_{15}, e_{16} and e_{56} are *Related edges*. According to definition 1, after overlapping, S_1 and S_2 are two subgraphs. Through our partitioning and overlapping processes, the original RDF graph is divided into n overlapped subgraphs over which the query will be processed.

Definition 2: d-reachable vertex pair. Given two vertices



(a) Graph Partition (b) Subgraph

Fig. 3. Graph Partitioning and Overlapping

v_i and v_j, the distance $D(v_i, v_j) = d$, we call v_i and v_j the d-reachable vertex pair.

The distance here is defined as the minimum number of edges that connecting two vertices.

Definition 3: 2-reachable Graph. Given a graph $G = (V, E)$, if $\forall(v_i, v_j) \in V$ is a 1 or 2-reachable vertex pair, G is called a 2-reachable graph.

The definitions lead to the following property:

Property 1: Given a 2-reachable graph g and a set of subgraphs $S = \{S_1, S_2...S_n\}$ obtained from partitioning and overlapping a RDF graph G according to definition 1. (1) If g is contained in G, g must be contained in at least one of the subgraphs $S_i...S_k \in S, 1 \leq i \leq k \leq n$ as well. (2) If g is not contained in any subgraph, then g must not be contained in G.

Proof: (1) Given that g is a subgraph of G, there are only two cases: (a) g is contained in a particular partition P_i. In this case, straightforwardly, g would be contained in the G's subgraphs that contain P_i. (b) Some vertices of g are borderline vertices, which means there does not exist a single non-overlapped partition containing g. However, according to definition 1, these vertices would be redundantly stored in their overlapped partitions. Therefore, g would still be contained in some subgraphs of G. (2) Suppose that g is not contained by any subgraphs of G, and g is a subgraph of G. According to (1), The conflict happens. Therefore, the assumption is false. We get property 1. ■

According to the property, given a 2-reachable RDF query graph over RDF graph G, the query results can be obtained soundly and completely from searching the subgraphs set S of G instead of G itself. The technique of indexing the subgraphs and decomposing a RDF query graph into 2-reachable subgraphs will be introduced in section III and IV.

III. Signature Tree based Index Structure

Property 1 has given a hint that a 2-reachable query graph can be answered completely and efficiently over a set of overlapped RDF subgraphs by comparing the query over the original graph. Supposing that a RDF query graph and a RDF data graph have been decomposed and divided into a set of 2-reachable graph subqueries and multiple overlapped partitions respectively in a certain way, we thus need to fast locate the embedded subgraphs in partitions for each 2-reachable graph subquery. However, subgraph isomorphism checking has been proven to be a NP-Complete problem. Therefore, we propose a two-phase strategy to address the problem: filter first and refine next. Through probing an index structure, all partitions that may potentially answer a particular 2-reachable graph subquery are returned. Then, an exact SQL query corresponding to the 2-reachable graph is executed over these candidate partitions to get the final results.

Considering that all vertices of an RDF graph identified by URIs are unique, we can provide a strong punning power in term of the URIs while processing a RDF subquery.

Property 2: Given two graphs g and s , the URI sets of g and s are U_g and U_s respectively. If and only if $U_g \bigcap U_s = U_g$, g might be potentially contained in s.

Property 2 indicates that those candidate RDF data partitions (or subgraphs) could be pruned safely during the query processing, while there exists at least a URI appearing in the query graph but not in the candidates. Taking the query example of Figure 1(a) again, we can find that the URI set of the query includes `:memberOf`, `:typeOf`, `:Dep0`, and `:Prof`. Comparing this URI set with the counterpart of each candidate partition as shown in Figure 2(b), only one partition whose GID value equals to 2 can satisfy the above requirement. The other two candidate partitions are pruned safely. At last, we just need to perform the costly subgraph isomorphism checking once over the filtered candidate partitions.

An URI is a string in essence, large scale of string matching is not trivial. In the following subsections, a URI signature and a signature tree index structure are proposed to speed up the filtering process. (In this paper, we do not take literals as pruning condition because some of RDF queries may involve range comparisons over literals.)

A. Graph Signature

The signature is a bit vector to represent a set of objects and often used as an approximate filter for supporting membership query. Its advantages like very quick comparison, easy maintenance and none false negatives lead to wide adoption in the applications [7], [8]. Many approaches such as [7] can be used to construct a signature. In this paper, we utilize bloom filter method [9] to generate signatures. A bloom filter-based signature consists of a vector of m bits and k independent hash functions ranging from 1 to m. According to the formula $p = (1 - e^{-kn/m})^k$ [9], n is the distinct number of elements. We can observe that the size m of the vector is varied with false positive rate.

The signature of a RDF triple is built through: (1) Hashing each URI U_i in a triple to k values $f_{i1}, f_{i2}...f_{ik}$, by k hash functions $h_1, h_2...h_k$. (2) Setting the corresponding positions of those hash values for URIs to 1 on the m bits vector. (For example, if a particular hash value equals to 5, we should set the 5_{th} position of the vector to 1.) Based on the RDF triple signatures, a RDF graph signature can be computed over them with an "OR" bit operation. As soon as a 2-reachable query graph q comes, its graph signature s_q is constructed at first, and compared with s_g. If $s_q \wedge s_g = s_q$, q may be a potential subgraph of g. Otherwise, any match of q must not be contained in g, and g is pruned safely. Benefited from bit operation, all computations among signatures are extremely efficient.

B. Index Structure

1) Index Construction: We structure the subgraph signatures to facilitate the filtering process using a tree. The leaf nodes in this tree are the signatures themselves, while the internal nodes are the combinations (bit "or") of their children nodes. The principle of organizing the tree structure is to push the internal nodes having higher selectivity up to the root as much as possible for early pruning. There were some existing methods for building a signature tree [7], [10], [11]. These techniques were mainly applied in the applications where the objects were independent with each other. Their optimizations are merely based on "0" and "1" positions of the signatures. For our problem, however, the objects are subgraphs, and they are generated by the graph partitioning process illustrated in Section II, where some vertices and edges on their borderlines are overlapped each other. The existing methods obviously do not perform very well against our problem. Our proposals are based on two different definitions of subgraph distances [12]: (1) A statistical approach which uses the size of overlapping region among a set of graphs to measure the distance and (2) Using hierarchical information which is obtained graph partition phase. We can conclude that the bigger the distances among the subgraphs are, the higher the selectivity of the internal node indexing them is.

2) Index Probing: When a query graph q comes, its signature s_q is used to probe index structure. When comparing with each internal node p with signature s_p, we need to test whether $s_q \wedge s_p = s_q$. If yes, we continue the comparisons with p's children. Once the searching reaches the leaf nodes, we can get a set of subgraph IDs. The answer of q if not NULL must be included in these subgraphs. While the test stops in the internal nodes or even the root, definitely, no results would be returned for q. Fast response to NULL-result query by probing an external index structure instead of accessing the databases is very useful for most of practical query applications. It is also one of the important features of our technique.

IV. Query Processing

As discussed in property 1, only a 2-reachable query graph can be searched in the subgraph set without loss of any accuracy. However, the query graphs are not always 2-reachable. Thus, We have to decompose a complex query graph into several 2-reachable subgraphs, and apply these subgraphs to probe index structure individually. These subgraphs are called *sub-queries*. Finally, we combine the related subgraph ids together for all the sub-queries and perform the query in database. To decompose a query graph, a two-phase method is proposed as follows:

1) Enumerate 2-reachable *sub-queries* centered by each vertex in the query graph.
2) Find out the *sub-queries* with highest selectivity in the signature tree.

For each sub-query q_i, the index probing phase returns an IDList which is a group of subgraph IDs. As long as the IDList is not empty, a SQL statement will be generated accordingly. Otherwise, the query processing is early terminated with NULL result. The size of IDList potentially affects the performance of SQL executions. Here, different SQL optimization plans are adopted in terms of the size of IDList. The SQL plans are produced as follows:

1) **Direct**: If the size of IDList is very small, IDList.size()$\leq n\alpha$, SQL statement is rewritten by adding subgraph ID constraints into WHERE clause directly.
2) **Temp Table**: If the size of IDList is not very large compared to the whole table, $n\alpha <$IDList.size()$< n\beta$, we insert all the subgraph IDs into a temp table and rewrite the SQL by an additional join with the temp table. (If we adopt the **Direct** method in this case, the SQL would be possible too long to be accepted by RDBMS for the length restriction.)
3) **Full Table**: When the size of IDList is very very large, IDList.size()$\geq n\beta$, we will determine to use the original SQL without additional constraints. The reason is that the cost of using the temp table increases beyond the querying over the whole table.

The parameter of α and β were tuned in our experiments [12].

V. Conclusions

We propose a novel scheme to store, index, and query RDF data in triple stores. Graph feature of RDF data is taken into considerations which help reduce the join costs on the vertical database structure. We first partition RDF triples into overlapped groups and store them in a triple table with group identity. Second, we build a signature tree to index them. Third, a complex RDF query is decomposed into multiple sub-queries to be evaluated with optimized SQL. The experimental results [12] confirm that for some extreme cases, it can promote 3 to 4 orders of magnitude.

References

[1] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient RDF storage and retrieval in Jena2." in *SWDB*, 2003, pp. 131–150.
[2] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A generic architecture for storing and querying RDF and RDF schema." in *ISWC*, 2002, pp. 54–68.
[3] J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, and Y. Yu, "Sor: A practical system for ontology storage, reasoning and search," in *VLDB*, 2007, pp. 1402–1405.
[4] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan, "An efficient sql-based rdf querying scheme," in *VLDB*, 2005, pp. 1216–1227.
[5] E. Prud'hommeaux and A. Seaborne, "SPARQL query language for RDF, W3C Candidate Recommendation," April 2006. [Online]. Available: http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/
[6] "http://glaros.dtc.umn.edu/gkhome/metis/metis/."
[7] Y. Chen, "On the signature trees and balanced signature trees." in *ICDE*, 2005, pp. 742–753.
[8] X. Gong, Y. Yan, W. Qian, and A. Zhou, "Bloom filter-based xml packets filtering for millions of path queries." in *ICDE*, 2005, pp. 890–901.
[9] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970. [Online]. Available: citeseer.ist.psu.edu/bloom70spacetime.html
[10] U. Deppisch, "S-tree: A dynamic balanced signature index for office retrieval," in *SIGIR'86*. ACM, 1986, pp. 77–87.
[11] E. Tousidou, A. Nanopoulos, and Y. Manolopoulos, "Improved methods for signature-tree construction," *Comput. J.*, vol. 43, no. 4, pp. 301–314, 2000.
[12] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan, "Efficiently querying rdf data in triple stores." Technique Report, 2008.