XML-Based RDF Data Management for Efficient Query Processing

Mo Zhou
Indiana University, USA
mozhou@cs.indiana.edu

Yuqing Wu
Indiana University, USA
yuqwu@cs.indiana.edu

Abstract

The Semantic Web, which represents a web of knowledge, offers us new opportunities to search for knowledge and information. To harvest such search power requires robust and scalable data repositories that can store RDF data and support efficient evaluation of SPARQL queries. Most of the existing RDF storage techniques rely on relation model and relational database technologies for these tasks. They either keep the RDF data as triples, or decompose it into multiple relations. The mis-match between the graph model of the RDF data and the rigid 2D tables of relational model jeopardizes the scalability of such repositories and frequently renders a repository inefficient for some types of data and queries. We propose to decompose RDF graph into a forest of XML trees, store them in an XML repository and rewrite SPARQL queries into XPath/X-Query queries to be evaluated in the XML repository. In this paper, we discuss the basic idea of RDF-to-XML decomposition, the criteria of such decomposition in term of correctness, redundancy and query efficiency. Then, based on these criteria, we propose two algorithms for decomposing RDF data into XML trees. Our experimental evaluation results illustrate that our approach is capable of improving both the storage efficiency and query processing efficiency compared to existing RDF techniques.

1. INTRODUCTION

RDF (Resource Description Framework) [12] is a W3C recommended language for describing linked data of the Semantic Web in the form of triples. *RDFS (RDF schema)* [6] extends RDF vocabularies to define the structure of underlying RDF data, as well as the taxonomic hierarchies of concepts and relations. The flexibility, simplicity and expressiveness of the Semantic Web evoke extensive exploitation of RDF/RDFS in many areas such as Bio-informatics and Social Networks. SPARQL[15] is a W3C recommended query language for querying RDF data and it features graph patterns for describing query requirements.

The needs to develop applications on the Semantic Web and support search in the RDF graph call for RDF repositories that are reliable, robust and efficient in answering SPARQL queries. As in the context of RDB and XML, the selection of the storage model is critical to a data repository as it is the dominating factor to determine how queries can be evaluated and how the system behaves when scales up.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PHD Workshop 2010 Indianapolis, Indiana USA
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

1.1 Related Works

Most of the existing RDF data repositories [1, 7, 8, 17] rely on the relational models for data storage and evaluates SPARQL queries by rewriting them into SQL queries and then executing them in the RDB engine. Among them, there are two major directions: one is to keep the simple triple data model [7, 17] of the RDF data and store it in one three column table; the other is to decompose RDF triples into multiple relations [1, 8].

In the *Triple Store* [7] approach, all triples in the RDF data are stored together in one table with three columns corresponding to (Subject, Predicate, Object). The triple store does not scale well as the evaluation of a complex SPARQL query invokes many self-joins. Various indexing techniques [11, 13, 14, 18] were proposed as remedy, at the cost of huge increase in storage space and decrease in the scalability and update efficiency.

In the *Vertical Partition* [1] approach, a two-column table is created for every distinct predicate in the RDF data. This approach works well for SPARQL queries when all predicates in the WHERE clause are known. Unfortunately, in case there is even one predicate variable in the query, all tables have to be accessed and results assembled by union operations.

In the *Property Table* [8] approach, RDF data is decomposed into relational tables by clustering RDF resources with common predicates into a property table. Its biggest advantage is that the query evaluation has small number of joins because a selection in one property table can match multiple simple access patterns. However property tables suffer many disadvantages as pointed out by [1]: (1) storage redundancy introduced by the mismatch between the flexibility of RDF data and the rigid RDB model; and (2) query overhead introduced by UNIONs and JOINs to connect the tables.

Both directions in storing RDF data in relational databases encounter the scalability problem in terms of the size of storage and the complexity of queries: the former suffers low query processing efficiency because of large number of self-joins; the latter suffers high overhead in storage and query evaluation and limited flexibility due to the mismatched data models. In addition, such storage models are not suitable for providing support to SPARQL query extensions with regular path variables which is critical to many applications that are core to the Semantic Web, such as semantic association discovery.

An alternative approach [3] is to preserve the graph nature of the RDF data by storing the RDF graph in an ORDB. However, this separates the RDF schema and RDF primary data, which brings difficulties in evaluating queries containing both schema and data instances.

1.2 Our Proposal

We summarize the desired properties of an RDF storage model to be as follows:

- Preserve semantics. The semantical relationships should be preserved and retrieved efficiently.
- High performance. It should be efficient to evaluate *all* SPARQL queries rather than only some types of SPARQL queries.
- Scalable. The storage should have no or small redundancy hence scale well.
- Small overhead. There should be very small overhead in query rewrite and query optimization.

Semi-structured data model organizes data entries in a tree structure and represents the semantic relationships among them via containment relationships. Tree pattern matching is at the core of the query languages on XML, e.g. XPath[10] and XQuery[9]. Over the last ten years, sophisticated techniques have been proposed for storing and querying XML data efficiently [2, 5].

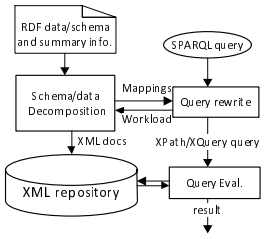

Figure 1: System Architecture

Observing the similarity between RDF and XML, in terms of data representation (e.g. using links in a tree or graph to represent the relationship among data instances) and query (e.g. tree pattern matching in XML and graph pattern matching in RDF), we propose to leverage the storage management and query evaluation techniques of XML data repositories to store and query RDF data. In other words, we propose to strategically decompose an RDF data into XML documents, which preserve the semantics, store the XML documents in an XML repository and then rewrite SPARQL queries into XPath/XQuery queries to be evaluated against the XML data using the latest XML query evaluation techniques that are capable of capturing and taking advantage of the structural relationship among the data. The system architecture that reflects our proposal is shown in Fig. 1.

Our contribution can be summarized as follows:

- We propose an XML-based RDF storage and query evaluation approach which is scalable and is capable of supporting various types of SPARQL queries.
- We define the RDF-to-XML decomposition and identify the criteria for measuring the goodness of a decomposition.
- We propose two RDF-to-XML algorithms for decomposing RDF data into XML documents, with the second algorithm taking workload into consideration to further improve the query performance.
- We discusse the query rewrite strategies for rewriting SPARQL queries into XPath/XQuery queries.
- We report on experiments to better understand the storage footprint of our approach, as well as the query performance for various types of SPARQL queries.

2. RDF-TO-XML DECOMPOSITION

2.1 RDF data and RDF queries

We represent RDF schema and RDF data in the form of a node and edge labeled graph: $R = (V_R, Ed_R, \lambda_R)$ where λ_R is a labeling function that maps items in $V_R \cup Ed_R$ into a finite set of labels and literals. We further distinguish RDF schema R_s from RDF data R_d, with $R_s = (V_{R_s}, Ed_{R_s}, \lambda_R)$ representing the class nodes and properties between classes, and $R_d = (V_{R_d}, Ed_{R_d}, \lambda_R)$ representing the data instances and properties between instances. In particular, in R_s, we call the set of labels associated with nodes the *class labels*, and the set of labels associated with the edges the *predicate labels*. Please note that $V_{R_s} \cup V_{R_d} = V_R$, but $Ed_{R_s} \cup Ed_{R_d} \neq Ed_R$ as there exist edges between class nodes and instance nodes in R indicating the *types* of the instances.

SPARQL [15] queries on RDF data are frequently represented as graph patterns that are formed by connecting a collection of *simple access patterns* (RDF triples with variables) in the query's WHERE clause. The connection between two simple access patterns with the same variable on their subjects are usually referred to SS joins. Similarlly, there are other types of joins, such as SO join, OO join, PP join, etc. Evaluating a SPARQL query can be regarded as mapping the graph patterns against the RDF graph to locate all matches.

2.2 Decomposing RDF data into XML documents

We propose to decompose RDF graph into a set of XML trees. We distinguish the decomposition in two steps: (1) the *schema-level decomposition* which maps RDF schema to a set of XML schemas; and (2) the *data-level decomposition* which maps an RDF data conforming to the RDF schema to a set of XML documents conforming to the XML schemas obtained from step 1.

We use a directed labeled tree $(V \cup \{root\}, Ed, \lambda_X)$, where λ_X is a labeling function that maps nodes in V to a finite set of labels, to represent an XML schema, and define the mapping from RDF schema R_s to a set of XML schemas X_s as follows:

DEFINITION 2.1. *Given an RDF schema R_s and a set of XML schemas $X_s = \{X_{s,1}, \ldots, X_{s,n}\}$, the mapping from R_s to X_s is a function $\mathcal{M}_s : R_s \rightarrow X_s$ such that*

- *for a node $v \in V_{R_s}$, $\mathcal{M}_s(v) = \{u | u \in \bigcup_{i=1\ldots n} V_{X_{s,i}} \wedge \lambda_X(u) = \lambda_R(v)\}$.*
- *for an edge $e = (v_1, v_2) \in Ed_{R_s}$, $\mathcal{M}_s(e) = \{((u_1, u), (u, u_2)) | (u_1, u), (u, u_2) \in \bigcup_{i=1\ldots n} Ed_{X_{s,i}} \wedge (\lambda_X(u_1) = \lambda_R(v_1) \wedge \lambda_X(u_2) = \lambda_R(v_2) \wedge \lambda_X(u) = \lambda_R(e))\}$.*

Based on the RDF schema decomposition, the RDF data R_d can be transformed into a set of XML documents conforming to X_s. To help define the RDF data decomposition, we first introduce the *label path* of a node v in an XML schema tree as the label sequence on the path from the child of the root to the node. Please note that this path always start with a class label and features predicate labels and class labels in turn. Assuming that $lp(v) = (c_1, p_1, c_2, p_2, \ldots, c_n)$ is a label path for v, we can construct a corresponding graph pattern $P_g(lp(v))$ that represents a query that looks for all data instances of class c_n that satisfy this pattern. We say that an XML schema class node v *hosts* data instances in $P_g(lp(v))(R_d)$.

DEFINITION 2.2. *Given an RDF data R_d that conforms to schema R_s, and a set of XML schema $X_s = \{X_{s,1}, \ldots, X_{s,n}\}$, together with the mapping function $\mathcal{M}_s : R_s \rightarrow X_s$, we say that a set of XML documents $X_d = \{X_{d,1}, \ldots, X_{d,n}\}$ is the translation of R_d under the mapping $\mathcal{M}_s$ if $X_{d,i}$ conforms to schema $X_{s,i}$ for $i = 1..n$ and there exists a mapping $\mathcal{M}_d : R_d \rightarrow X_d$ such that*

- *for any node $v_d \in V_{R_d}$ of type $v_s \in V_{R_s}$ and any $v_{X_s} \in \mathcal{M}_s(v_s)$, $\mathcal{M}_d(v_d, v_{X_s}) = \{u_d | u_d$ conforms to v_{X_s} and $v_d \in P_g(lp(v_{Xs}))(R_d)\}$.*
- *for any edge $e = (v_1, v_2) \in Ed_{R_s}$ of type $e_s \in Ed_{R_s}$ and any XML schema fragment $f = ((u_{s,1}, u_s), (u_s, u_{s,2})) \in \mathcal{M}_s(e_s)$, $\mathcal{M}_d(e_d, e_{X_s}) = \{((u_{d,1}, u_d), (u_d, u_{d,2})) | ((u_{d,1}, u_d), (u_d, u_{d,2}))$ conforms to f and $e_d \in P_g(lp(u_s))(R_d)\}$.*

Given the mapping $\mathcal{M}_s : R_s \rightarrow X_s$, we distinguish *class node* and *predicate node* in X_s based on what RDF schema element (class or predicate) is mapped into it. For a node $v \in V_{R_s}$, if a class node $u \in \mathcal{M}_s(v)$ can host all instances of the RDF class $\lambda_R(v)$ in R_d, we call it a *full node*. Further if u can preserve the immediate local structure, e.g, v's all outgoing edges corresponding to all predicates of the class $\lambda_R(v)$, we call u an *expanded full node* of v.

Given any class node u in X_s, it is desirable to find out whether u is a full node without computing the mapping $\mathcal{M}_d$. In fact, such knowledge can be obtained by analyzing R_d and summarizing the participation constraint among the classes in R_s.

DEFINITION 2.3. *Given an RDF data R_d conforming to an RDF schema R_s and given two classes c_1, c_2 connected by a predicate p in R_s, we say c_1 (c_2) is totally participated in a relation (c_1, p, c_2) (denoted as $c_1 \vdash_p c_2$ ($c_2 \vdash_p c_1$)) , if in R_d any instance of type c_1(c_2) connects to at least one instance of type c_2(c_1) by edge labeled p.*

We observe that u must be a full node if u is a child of a root in X_s. In addition, if there are two edges $(u', e), (e, u) \in \bigcup_{i=1\ldots n} Ed_{X_{s,i}}$ and u' is a full node, u is a full node only if $\lambda_X(u) \vdash_{\lambda_X(e)} \lambda_X(u')$.

3. DECOMPOSITION CRITERIA

The notions of X_s, X_d and the mappings $\mathcal{M}_s$ and $\mathcal{M}_d$ are very loosely defined. There are multiple ways to decompose an RDF schema into XML schemas, and therefore multiple ways to decompose an RDF data into XML documents. Therefore we propose the pre-requisite for the data transformation from RDF to XML to be correctness, low redundancy and high semantic clustering.

While the first criteria must be satisfied, the second and third indeed reflect the trade-off between space efficiency and query efficiency that are traditional to database system design.

Most RDF schemas have hierarchies where a class has all instances in its subclass and inherits all predicates of its superclasses. However, by analyzing the data, it is always possible to find the finest level of classes whose instances together with relations among them form the entire dataset but are not overlapped. Therefore, without loss of generality, we only consider such classes and relations among them in the rest of the paper.

Correctness

The correctness of the decomposition is determined in the schema decomposition. We say that a set of XML schemas X_s *structurally covers* a given RDF schema R_s if any node or edge in R_s has at least one mapping in X_s. We say X_s can *fully cover* R_d if there exists an X_d conforming to X_s such that any node or edge in R_d has at least one mapping in X_d.

Instead of taking all possible X_s fully covering R_d into consideration, we are interested in a subset of them that is a straightforward translation of R_s.

DEFINITION 3.1. *Given R_s, R_d conforming to schema R_s, X_s, R_d and the mappings $\mathcal{M}_s$ and $\mathcal{M}_d$ we say that X_s is a* full *XML schema if for any node $v_{R_s} \in V_{R_s}$ there exists at least one entry in $\mathcal{M}_s(v_{R_s})$ that is an expanded full node.*

Redundancy

We summarize two types of redundancy: *Schema Level Redundancy (SLR)* when an RDF schema node is mapped to more than one XML schema node; and *Instance Level Redundancy (ILR)* when an RDF data node is mapped to more than one XML data node that conforms to the same XML schema node.

SLR can be reduced by removing redundant nodes from X_s under the condition that X_s still fully covers R_s. However SLR cannot be fully avoided if R_s contains circles. It is possible to map any edge in R_s to exactly one fragment in X_s but at least one node in the cycle has to be repeated in X_s.

The existence of ILR is tied to the cardinality relationship between classes and the predicate connecting them.

DEFINITION 3.2. *Given an RDF data R_d conforming to an RDF schema R_s and given two classes c_1, c_2 connected by a predicate p in R_s, we say (c_1, p, c_2) is a 1-1 relation ($c_1 \leftrightarrow_p c_2$), if in R_d any instance of c_1 is connected to at most one instance of c_2 by p and vice versa. Similarly, we define 1-n relation ($c_1 \rightarrow_p c_2$) and m-n relation ($c_1 -_p c_2$).*

ILR exists if (1) there exists at least one m-n relation in R_s; or (2) two nodes c_1 and c_2 in R_s with $c_1 \rightarrow_p c_2$ maps to two class nodes c_1' and c_2' in X_s where c_2' is an ancestor of c_1'. ILR in case (1) cannot be eliminated. However ILR caused by case (2) can be eliminated by avoiding such schema mappings.

We say that a decomposition has *minimum redundancy* if no node in X_s can be removed without losing its full coverage over R_d and no two XML schema trees in X_s can be combined into one schema tree without introducing ILR.

Query efficiency

We propose to rewrite SPARQL queries into XPath/XQuery queries that are evaluated on XML documents in XML engines. Well-designed RDF-to-XML decomposition strategy can improve the efficiency of query evaluation by increasing data locality and minimizing the number of joins.

With respect to the data locality we would expect that minimum amount of data needs to be accessed to evaluate a query and the accessed data should be physically stored together. With respect to the joins, value joins can not be avoided when nodes in different XML documents have to be accessed. However, structural joins are much more efficient than value joins [2] and multiple structural joins along a linear pattern can be replaced by a single index access when structural indices are available [5]. Therefore, we strongly favor structural joins over value joins, e.g. favor deeper XML documents than shallower ones, even though it may sacrifice data locality to a certain degree.

4. DECOMPOSITION ALGORITHMS

In this section we describe algorithms for decomposing RDF schema and data into XML schemas and documents that satisfy the criteria in Sec. 3.

4.1 Schema Decomposition

We propose two algorithms, greedy decomposition (gR2X) and workload-based decomposition (wR2X), for decomposing an RDF schema into XML schemas. Both algorithms take as inputs an RDF schema R_s and the summary information reflecting the total participation relations and cardinality relations as discussed above and return XML schemas X_s as output. Both are carried out in the following steps:

Step 1. Initialization. We create node lists V and F. V contains the class nodes to be assigned to XML schema trees and is initialized to the set of nodes in R_s. For each class node c in V, we create a list L_c that contains relations (c, p, c') where $c \vdash_p c' \wedge c' \rightarrow_p c$. F contains the root nodes of the XML schema trees and is initialized to be an empty set.

Step 2. Tree root selection. For all nodes c in V where L_c is empty, we create a dummy root node r in F, make c the child of r, and remove it from V.

Step 3. Class node placement. We maintain a prioritized list of the class nodes in V. Each time, we pick the node c at the head of the list, call *chooseRelation*(L_c) function to pick a relation in L_c that

is to be mapped in the XML schema. If there is no such relation, we create a dummy root node r in F, and make c the child of r. Otherwise for each relation (c, p, c'), we create a predicate node u_p with label p, make it the child of c' and make c the child of u_p. In both case we remove c from V.

Step 4. Node expansion. We expand the class nodes to expanded full nodes so that all predicates that have not been mapped into the XML schema are mapped now and the resultant XML schemas structurally cover R_s.

Please note that the design of the key of the prioritized list and *chooseRelation* function is at the core of the decomposition, as it reflects the strategy on when/where/how a class node should be placed.

Greedy Decomposition (gR2X)

The greedy decomposition uses a greedy approach in *chooseRelation* to construct *full* XML schemas (Def 3.1) with *minimum redundancy* and deep tree structures. The class nodes with smallest L_c have higher priority in the prioritized list. Given a class node, the *chooseRelation* function chooses a relation from L_c, such that the relation (c, p, c') satisfies the following conditions: 1) c' is not a decendant of c; and 2) c' is in a tree that is deepest among all the class nodes that have a relation with c in L_c. The widest tree is chosen in need of breaking a tie.

Workload-based Decomposition (wR2X)

It is frequently the case that logical and physical storage models are designed to support efficient evaluation of frequent (sub)queries, hence improving the overall throughput. Since every path in the query graph pattern can be mapped to a path in the RDF schema by replacing the nodes along the path by their classes, we use a set of weighted paths in the RDF schema to represent the frequent paths in a workload where weights reflect frequencies. Our workload-based schema decomposition algorithm (wR2X) is to ensure that every frequent path in the workload is represented by a path in the resultant XML schemas.

The wR2X algorithm utilizes the general framework discussed above and takes one extra parameter, a set of weighted paths L_e, as input. For the prioritized list of class nodes, the key of a class node is the sum of the weights associated with the paths in L_e containing the node. The *chooseRelation* function chooses relations that are heavily covered by the paths in L_e to be placed first. More specifically, the *chooseRelation* function chooses a relation from L_c , so that the relation (c, p, c') satisfies the following conditions: 1) c' is not a descendant of c; and 2) the sum of the weights associated with paths containing the relation in L_e is the highest. The deepest tree is chosen in need of breaking a tie.

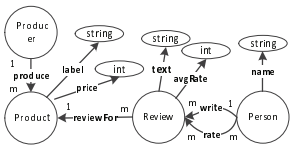

Figure 2: Example RDF schema

After we generates the minimum full XML schemas in step 4, we expand the XML schemas to include all frequent paths in the step as describe below:

Step 5 Path expansion We *remove* paths in L_e that are already represented in the XML schemas. We *pick* the longest path lp with the highest weight from the remaining paths in L_e. We then identify the candidate paths in the XML schemas that are rooted at a full node and contain some fragments of lp. For each candidate path, we estimate the redundancy that may be introduced if we expand the path to represent lp and pick the one with the minimum estimated redundancy. We *expand* the selected path to represent lp and remove lp from L_e. We keep the *remove-pick-expand* process until L_e is empty so that all paths are represented.

EXAMPLE 4.1. *Consider the example RDF schema shown in Fig. 2. The numbers on the edges are the cardinality information between the corresponding classes and we know that $Person$ is the only class that does not total participate in any relation, while the other classes total participate in all relations associated with them. Fig. 3 illustrates how the gR2X and wR2X algorithms decompose this RDF schema into corresponding XML schemas [1]. We use the shades of the nodes (light to dark) to indicate in which steps they are added to the resultant schemas.*

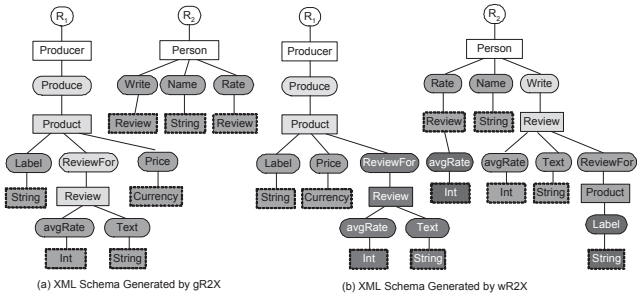

Figure 3: Schema Decomposition

4.2 Data Decomposition

Our data decomposition algorithm (R2D) transforms an RDF data R_d conforming to schema R_s into XML documents conforming to X_s generated using gR2X or wR2X. Intuitively the transformation process is based on a depth-first traversal of the R_d graph guided by a depth-first traversal of each XML schema in X_s. Details are omitted due to space limitation.

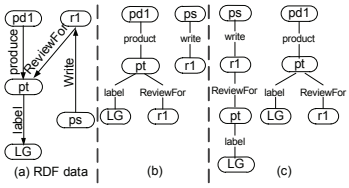

Figure 4: Data Decomposition

EXAMPLE 4.2. *Please consider a small fragment of RDF data shown in Fig. 4(a) that conforms to the schema shown in Fig 2. Fig. 4(b) and (c) shows the XML data generated based on the resultant XML schemas in Fig. 3 (a) and (b), respectively.*

5. QUERY REWRITE

Query Rewrite Algorithm

We propose to rewrite SPARQL queries into XPath/XQuery queries based on the XML schemas generated and the mapping from the RDF schema to XML schemas. Given a SPARQL query, the rewrite is carried out in three steps.

Step 1: Class/predicate identification We derive the possible classes of each node and predicate variable in the query using the information available in the RDF schema, such as the possible predicates of a known class and the range and domain of a known predicate.

Step 2: Query decomposition We decompose the SPARQL query graph into pattern trees where each pattern tree can be matched to a subtree rooted at a full node in the XML schema. There could be many ways of such decomposition, among which we choose one that satisfies the following criteria: 1) the number of pattern trees are small; 2) the pattern trees feature longer paths; and 3) the roots of the pattern trees match to XML schema nodes that are closer to

[1]The workload consists of three paths: a. (Producer $product$ Product $reviewFor$ Review $avgRate$ Int) with w=30; b. (Person $rate$ Review $avgRate$ Int) with w=10; and c. (Person $write$ Review $ReviewFor$ Product $label$ String) with w=15

the root. We annotate each node in the pattern tree with following information: 1) the role of the node in the query (e.g. variable, optional variable, return variable, URI, or literal), and 2) filtering conditions. In addition, we keep track of the overlapping nodes in these pattern trees as they are the *point of join* to reconstruct the graph pattern from the tree patterns.

Step 3: XML query construction We construct an XPath expression for each pattern tree and assign a variable to it in a *for* clause; we use the *where* to take care of the joins among the pattern trees and the filtering conditions; we then put all return variables into the *return* clause. Hence an XQuery query is constructed.

The rewrite process of the following query is illustrated in Fig. 5. Stratigical construction methods can facilitate query evaluation. For example, XPath2 likely leads to more efficient evaluation plan than XPath1 does by replacing **/reviewFor/*/text* to **//text*.

Q_1 : Select ?l From ...
Where {?r text "nice" . pd produce ?pt .
?r ?p ?pt . ?pt label ?l}

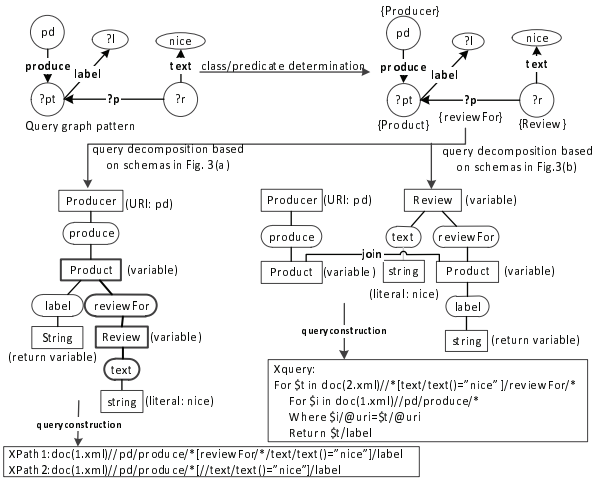

Figure 5: Query rewrite

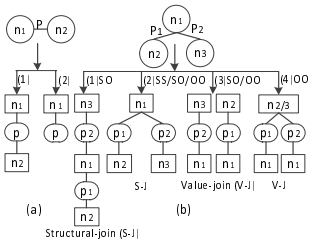

Figure 6: Example Pattern Matching

Matched Patterns

A simple access pattern in a query is matched to a path in the XML document as in Fig. 6(a) and evaluated by structural joins. When a variable is neither to be joined nor returned, such a pattern can be matched to an edge like Fig. 6(a)(2).

Two simple access patterns connected by SS/SO/OO join are matched to either a path or twig(s) in the XML document as in Fig. 6(b). Simple access patterns connected by an SS join are always matched to a twig as Fig. 6(b)(2) and evaluated by structural joins, while those connected via SO/OO joins are matched to path or twig(s) as in Fig. 6(b) and may get value-join involved in the evaluation.

A graph pattern consists of many simple access patterns is matched to tree(s) in the XML documents as a natural extension to the above situations.

6. EXPERIMENTAL EVALUATION

Experimental Setup We conduct extensive experiments to compare the XML-based approach we proposed, with the well-know existing approaches, including triple store (TS) [11], vertical partition (VP) [1] and property class table (PT) [8]. We implement (g/w)R2X, e.g. gR2X and wR2X, on MonetDB/XQuery v4.20.0 and the RDB-based approaches on MonetDB server v4.34.2. We rewrote SPARQL queries into SQL queries for TS, VP and PT following the examples in the Berlin benchmark[4] and the discussion in [16]. We generated XPath/XQuery queries based on the approach in Sec. 5. We time the hot run [17] of each query. The experiments were carried out on a desktop PC running Ubuntu v8.10 linux (32-bit) with Intel Pentium4 2.80GHz CPU, 3GB memory, and SAMSUNG SP2004C hard drive with 8MB Cache.

Dataset and Queries We used the Berlin Benchmark to generate RDF datasets in Turtle format that used compact *in-scope Base URI*. For each dataset, We used $gR2X$ and $R2D$ algorithms to generate a set of XML documents based on the greedy approach and used $wR2X$ and $R2D$ with workload that consists of four frequent paths of lengths between 1 and 5 to generate another set of XML documents. In triple store, we sorted triples by PSO and created indexes on all permutations of S,P,O; in vertical partition, we sorted tuples by SO and created an index on OS in each table. We report results on seven datasets with 100K, 400K, 1M, 5M, 10M, 15M and 20M[2] triples.

We classified queries in two sets, unbound-variable (UV) queries and extensive-join (EJ) queries. The queries in the UV query set have simple access patterns with variables on nodes or predicates. We report results on four UV queries: UV_{s_1/s_2} only have subject variables and UV_{p_1/p_2} contain predicate variables. The predicate in UV_{s_1} is associated to only one class while the one in UV_{s_2} is associated to many. UV_{p_1} returns the predicate variable while UV_{p_2} returns the object variable. Their results are all less then 12 triples.

Query	#SS/OO/SO	#DP	#UC	OP	DC	NR
EJ_{B_1}	4/0/0	4	0	N	N	187
EJ_{B_2}	0/0/3	14	2	Y	N	20
EJ_{U_1}	4/0/0	5	0	N	Y	44
EJ_{U_2}	2/0/3	6	1	N	N	134
EJ_{U_3}	3/1/1	5	0	N	Y	13875

Table 1: EJ Query Characteristics

The EJ queries feature various number and types of joins. We included various benchmark queries as well as queries we generated that reflect some special features not available in the benchmark queries. We identify the following characteristics of SPARQL queries as factors that have significant impact on the query evaluation: (1) the number of SS/OO/SO joins; (2) the number of distinct properties in the query (#DP); (3) the number of subjects whose classes are neither explicitly indicated by *RDF:type* nor determined by predicates directly associated with them (UC); (4) whether the query contains OPTIONAL operators (OP); (5) whether the query contains variables neither to be joined nor returned (DC); and (6) the cardinality of results (RN). We report results on two benchmark queries (EJ_{B_1}, EJ_{B_2}) and three queries (EJ_{U_1},EJ_{U_2},EJ_{U_3}) of our own. Their characteristics are summarized in Tab. 6. The values in the **NR** column reflect the cardinalities of the query results when the queries are evaluated against the dataset with 5M triples.

Storage Size We examine the size of physical storage of data with indexes and show the results in Fig. 7. The PT is most compact since predicates are stored only once in the database catalog. Com-

[2]MonetDB failed to load 20M dataset into a single table for TS approach. When an XML file exceeds the capacity of the MonetDB/XQuery, we break it into smaller files. We issue queries against them and union the results.

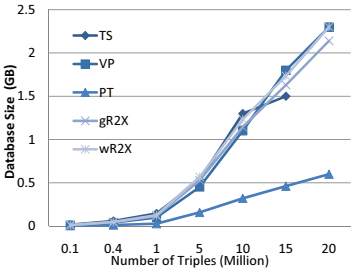

Figure 7: Storage Size

paring to PT and TS with indexes, our (g/w)R2X approaches require slightly less storage and scale in the similar manner.

Query Evaluation We conducted experiments on various queries on all data sets we created. Due to space limitation, we will only present the query time on the dataset with 5M triples in Fig. 8. We summarize our observations in the following aspects:

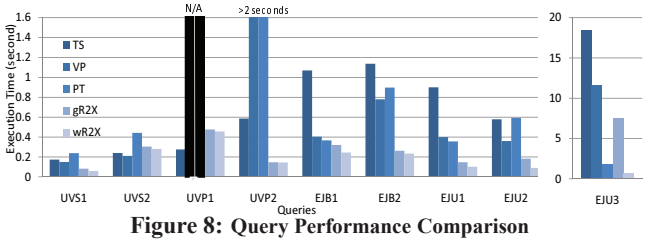

Figure 8: Query Performance Comparison

1. (g/x)R2X and TS are capable of evaluating all types of queries purely based on primary data, while PT and VP have to rely on database catalog (as in UV_{P_2}) or cannot handle such queries without further information (as in UV_{P_1})[3].

2. (g/w)R2X delivers good performance in evaluating various queries from simple selection to ones with many joins, which TS, VP and PT all favor certain types of queries while having difficulty with others: PT has significant performance retrogression caused by uncertain subject class (UV_{S_2}, EJ_{B_2}) and predicate variables (UV_{P_1}), VP is sensitive to the number of predicates (EJ_{B_2}) and predicate variables (UV_{P_2}), and TS does not favor complex queries (EJ queries).

3. (g/w)R2X perform very well on path matching queries (EJ_{B_2/U_2}) and outperform RDB-based approaches because of the efficient structural joins and high data locality.

4. (g/w)R2X access less data than RDB-based approaches, because our query rewrite algorithm restricts the data to be touched in the evaluation process to a small fragment that corresponds to only few classes as discussed in Sec. 5. Only PT has similar feature, but less precise than our approach in terms of restricting classes (EJ_{B_2/U_2}). Our approach is also capable of skipping certain matches as the information about the structures is preserved in the mapping and is available to the rewrite algorithm (EJ_{U_1}).

5. (g/w)R2X can efficiently support queries with OPTIONAL operators because XQuery naturally supports the operator while PT and VP have to involve additional left-outer-join (EJ_{B_2}).

Frequently, SPARQL queries feature many simple patterns and mixed types of joins. It is very challenging to evaluate such queries (e.g. EJ_{U_3}) against massive data on the Semantic Web. The evaluation of EJ_{U_3} is very costly for TS and VP due to the loss of the advantage of sort-merge join. Comparing to them, PT performs better because of the transformation from SS joins to selections. Structural joins are used in our approaches to evaluate SS and SO joins, while value-join is used here to evaluate OO joins. wR2X performs best among all because of its high data locality. gR2X does not perform well since the deep tree structure chosen by the greedy approach results in lower data locality, but it still outperforms TS and PV significantly.

Scalability We study the scalability of our approaches as well as the RDB-based approaches by comparing the evaluation time of the same query on data of various sizes. We observe that TS scales poorly which is consistent with what was pointed out in the literature. VP scales badly when the query is complex and contains many SO joins. PT scales the best among the RDB-based approaches but its high parsing and optimization time is too big an overhead for it to be suitable for small datasets. Comparing to them, gR2X and wR2X are able to resist the impacts from the complexity of the data. wR2X scales better than gR2X because wR2X preserves data locality in the XML documents even when the dataset scales up. The test result of query EJ_{U_2} is shown in Fig. 9 to illustrate our observations.

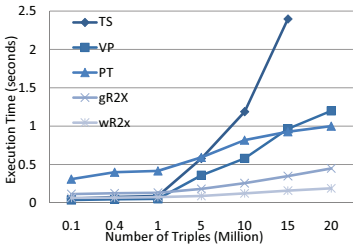

Figure 9: Scalability Study

7. SUMMARY

To answer the increasing demands on RDF repository, especially high query processing efficiency and scalability, we carefully studied the existing RDF data management systems, identified the preferred properties of an RDF repository and proposed to take advantage of the latest XML data storage and query evaluation techniques and to decompose an RDF graph into XML trees for storage and query evaluation. We identified correctness, low redundancy and high query efficiency as the criteria for a good RDF-to-XML decomposition and proposed two algorithms for decomposing an RDF graph into XML trees. Our experiments demonstrated that compared to other existing RDF storage and query evaluation techniques, our proposed approach requires small storage space, is capable of evaluating SPARQL queries more efficiently and scales better. In addition, our approach is indifferent to the complexity of data and type of queries, making it suitable for supporting Semantic Web applications in various domains.

8. REFERENCES

[1] D. J. Abadi, *et al.* Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, 2007.
[2] S. Al-Khalifa, *et al.* Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, 2002.
[3] S. Alexaki, *et al.* The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *SemWeb*, 2001.
[4] C. Bizer, *et al.* The Berlin SPARQL Benchmark. In *Int. J. Semantic Web Inf. Syst.*, 5(2): 1-24 (2009).
[5] S. Brenes, *et al.* Trie Indexes for Efficient XML Query Evaluation. In *WebDB*, 2008.
[6] D. Brickley, *et al.* RDF Vocabulary Description Language 1.0: RDF Schema, Feb 2004.
[7] J. Broekstra, *et al.* Sesame: An Architecture for Storin gand Querying RDF Data and Schema Information. In *Spinning the Semantic Web*, 2003.
[8] J. J. Carroll, *et al.* Jena: implementing the semantic web recommendations. In *WWW*, 2004.
[9] D. Chamberlin, *et al.* XQuery 1.0: An XML query language, May 2003.
[10] J. Clark, *et al.* XML path language(XPath), 1999.
[11] G. H. L. Fletcher, *et al.* Scalable indexing of RDF graphs for efficient join processing. In *CIKM*, 2009.
[12] RDF Working Group. Resource Description Framework (RDF), Feb 2004.
[13] A. Harth, *et al.* YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC/ASWC*, 2007.
[14] T. Neumann, *et al.* The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
[15] E. Prud'hommeaux, *et al.* SPARQL Query Language for RDF.
[16] M. Schmidt, *et al.* An experimental comparison of rdf data management approaches in a sparql benchmark scenario. In *ISWC*, 2008.
[17] L. Sidirourgos, *et al.* Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.
[18] C. Weiss, *et al.* Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[3] [16] relies on an additional table that stores all predicates to handle such queries.