An Indexing Scheme for RDF and RDF Schema based on Suffix Arrays

Akiyoshi MATONO[1], Toshiyuki AMAGASA[1], Masatoshi YOSHIKAWA[2], and Shunsuke UEMURA[1]

[1] Graduate School of Information Science, Nara Institute of Science and Technology
8916–5 Takayama-cho, Ikoma-shi, Nara 630–0192, Japan
{akiyo-ma,amagasa,uemura}@is.aist-nara.ac.jp
[2] Information Technology Center, Nagoya University
Furo-cho, Chikusa-ku, Nagoya-shi 464–8601, Japan
yosikawa@itc.nagoya-u.ac.jp

Abstract. The Semantic Web is a candidate for the next generation of the World Wide Web. It is anticipated that the number of metadata written in RDF (Resource Description Framework) and RDF Schema will increase as the Semantic Web becomes popular. In such a situation, demand for querying metadata described with RDF and RDF Schema will also increase, and therefore effective query retrieval of RDF data is important. To this end, we propose an indexing scheme for RDF and RDF Schema. In our (proposed) scheme, we first extract four kinds of DAGs (Directed Acyclic Graphs) from an RDF data, and extract all path expressions from the DAGs. Then, we generate four kinds of suffix arrays based on the path expressions. Using the indices, we can achieve efficient processing of query retrievals on RDF data including schematic information defined by RDF Schema (for example, classes and/or properties).

1 Introduction

The Semantic Web [1, 2] has emerged as the next generation of the World Wide Web. In the Semantic Web, human-to-machine and machine-to-machine interactions are expected to become more intelligent from the wealth of metadata associations between resources on the Internet. The key difference between the current Web and the Semantic Web is the quality and quantity of metadata. Currently available metadata are insufficient, in terms of quality and quantity, for the purposes of advanced processings. The Semantic Web, on the other hand, makes it possible to perform high-level processes, such as reasoning, deduction, and semantic searches, to make the best use of metadata associated with web resources.

In the Semantic Web, RDF (Resource Description Framework) [3] and RDF Schema [4] are commonly used to describe metadata. RDF is a framework to describe data and their semantics, and is composed of the RDF model and RDF syntax. In the RDF model, *statements* are used to describe relationships between pairs of terms. A *statement* is called a triple, because a statement is comprised of three elements: a resource, a property and a value. The value can be either literal or resource, and thus complex information can be represented as a set of statements, such as a form of directed graphs. RDF syntax is a specification to serialize RDF statements as XML (Extensible Markup Language) data. RDF Schema is the schema language for RDF used to specify schematic information, such as definitions of resources, properties and classes.

In the near future, the quantity of metadata represented by RDF is expected to increase significantly as the Semantic Web comes into wide use. We expect that RDF databases will become important as an efficient means of access to massive metadata bases written in RDF and RDF Schema. One naive approach to constructing RDF databases is to use XML databases to store and retrieve RDF data simply because any RDF data can be represented in XML. However, this approach is not practical because the structure of RDF data is different from the structure of XML data, and there are many ways to serialize RDF data in XML form. Thus, queries to retrieve RDF data cannot be implemented as queries of their XML representations.

Another way to implement RDF databases is to utilize relational databases. In this approach, a piece of RDF data is decomposed and stored into relational tables. Several methods have been proposed already [5]. RDFSuite [6] is an implementation of RQL (RDF Query Language) [7], a query language for RDF. To store RDF data, RDFSuite uses tailor-made relational schema specially designed for the RDF Schema that we would like to explore. Jena [8] is an RDF database that implements RDQL (RDF Data Query Language) [9] using MySQL. However, a few of the previously mentioned works has investigated the performance of RDF databases.

We propose an indexing scheme for RDF and RDF Schema to achieve efficient query retrieval. Specifically, we focus on path expressions extracted from RDF and RDF Schema. Our first step is to extract four types of partial graphs from RDF and RDF Schema, because RDF and RDF Schema data have four distinct relationships. The graphs represent relationships among instances, classes and properties. Then, we extract all possible path expressions from the graphs, and construct suffix arrays on the path expressions. As a result, for a given query as partial path expression, we can efficiently detect the result.

The basic concept underlying our proposal is similar to that of Yamamoto et al. [10]. The main difference is that this approach is used for XML data, whereas we propose applying it to RDF and RDF Schema. Since XML data is a tree structure, enumeration of all possible path expressions in XML is an easy task. However, path expression cannot be as straightforward with RDF and RDF Schema, because they may contain multiple paths and/or cycles. For this reason, we will limit our first targets to cases where RDF and RDF Schema do not contain cycles.

However, even if we limit our target to DAGs, we should claim that our scheme can be applicable to many applications due to the fact that a large majority of RDF data in real applications is expressible as DAGs. For instance, WordNet [11], a famous on-line lexical database written in RDF, does not contain cycles based on our investigation. Following this step we will introduce a method to cope with cycles.

We have implemented our approach and evaluated its performance in a series of experiments. We used four kinds of RDF documents with different sizes using Wordnet [11], and stored each of the four RDF documents in RDFSuite. Eight queries were executed against the RDF database to compare the processing time using our index and

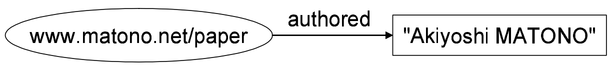

Fig. 1. An RDF data model statement.

those of non-index (or original indices of RDFSuite). Our index was more efficient than the non-index, and our approach has shown scalability.

The rest of this paper starts with an outline of RDF and RDF Schema using examples in Section 2. In Section 3, we describe our approach for efficient RDF retrieval. In particular, we defined a suffix array for DAG, explained about extracting the four DAGs from the RDF data, and described path expressions for each DAG. In addition, we describe our experimental setup and evaluate the performance using our index in Section 4. In Section 5, we describe an idea to cope with cycles. We discuss related work in Section 6, and conclude the paper in Section 7.

2 An Overview of RDF

RDF (Resource Description Framework) [3] is a foundation for representing and manipulating metadata on Web resources. RDF enables us to implement various applications, such as resource discovery, interoperation of metadata and description of machine-understandable information.

In RDF specification, the data model and its syntax are defined. In addition, RDF Schema [4] is used to describe schematic information of RDF data.

RDF can be used to describe the metadata of any resource in the Net as long as its location is identifiable using a URI (Uniform Resource Identifier) [12]. In RDF, "statements" are used to represent binary relationships between two distinct (or maybe identical) resources. Complex information can be represented by a set of statements. Thus, an RDF document is modeled as a directed graph (DG), where a resource corresponds to a vertex and a relation corresponds to an arc. For example, let us take a look at the statement "this paper is authored by Akiyoshi MATONO." The statement consists of three parts, namely, a subject ("This paper"), a predicate ("is authored by") and an object ("Akiyoshi MATONO"). For this reason, the statement is also called a triple. We call the relation represented by a statement the "predicate relation" (Figure 1).

For the purpose of exchanging metadata written in RDF, RDF syntax, by which we can serialize RDF data into XML data, is defined. Figure 2 shows an RDF document corresponding to the above example.

RDF Schema is used to give semantic information to RDF data. Specifically, RDF Schema makes it possible to specify the properties of a resource, data type of a property, class memberships of properties, and class hierarchies.

Using RDF and RDF Schema, we can represent complex information (Figure 3). Classes and properties defined by RDF Schema are shown in the upper part. For example, the property "creates" takes an "Artist" and an "Artifact" as its domain and range, respectively. "Sculptor" is a subclass of "Artist", and so on. Resource descriptions can

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:s="http://www-db.aist-nara.ac.jp/~akiyo-ma/test.rdfs#">
  <rdf:Description about="www.matono.net/paper">
    <s:authored>Akiyoshi MATONO</s:authored>
  </rdf:Description>
</rdf:RDF>
```

Fig. 2. An RDF document.

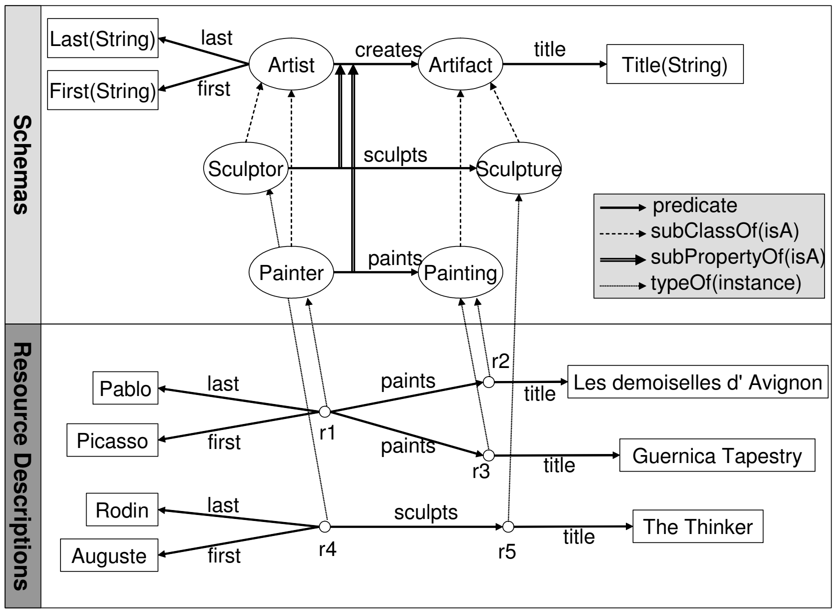

Fig. 3. A complex example using RDF and RDF Schema.

be found in the lower part. Resources, such as “r1” and “r2”, are defined as instances of classes. Consequently, resource “r1”, for example, has three properties, “last”, “first” and “paints”, which are inherited from the “Artist” class. Resources as character strings, such as “Pablo” and “Picasso”, are instances of the Literal class in RDF Schema.

3 Efficient RDF Data Retrieval using Suffix Array for DAGs

3.1 Problem description

Basically, queries on RDF data can be expressed as combinations of some path expressions based on graph structures of RDF data. For example, the query “find all resources that are created by artists” can be constructed as follows: 1) find all artists, and 2) for each artist, find all resources that are reachable by following “create” property. As we can see, both steps, 1) and 2), can be processed on the basis of path expressions of RDF graphs. We can therefore say that efficient processing of path expressions is crucial to achieve efficient RDF data retrieval.

In fact, this is similar to XML data retrieval, and many researchers are devoted to efficient XML query processing based on path expressions [13, 10, 14]. Both XRel [13] and XParent [14] propose a relational schema based on path expressions for efficient storage and retrieval of XML data into a relational database. In Yamamoto et al. [10], an indexing scheme based on path expressions is proposed. In this approach, all path expressions are extracted from XML data first. Then, a suffix array is constructed on the extracted path expressions where the occurrences of element (or attribute) names are alphabets. As a consequence, we can efficiently find any (partial) path expression using the full-text search functionality provided by the suffix array.

However, we cannot apply the above technique to RDF data, because of the differences between RDF and XML data. The differences can be summarized as follows; 1) RDF data may contain cycles, whereas XML data does not. This comes from the topology of RDF graphs, that is, an RDF data forms a directed graph and an XML data forms a tree. Extracting all possible path expressions from an RDF data is not trivial, consequently. 2) In RDF data, not only vertexes but also arcs have labels, whereas arcs are not labeled in XML data. Thus, we need to take care in path expressions. 3) We need to take schematic information provided by the RDF Schema, because we think that the query which involves schematic information is general on the Semantic Web.

3.2 Proposed method

In this paper, we propose a novel indexing technique based on suffix arrays for efficient retrieval of RDF data. The basic idea behind our approach is similar to that of Yamamoto et al. [10]. In order to cope with the above problems, we made the following modifications.

1. To cope with problem 1), we first limit out target to RDF data of DAGs (Directed Acyclic Graphs), that is, we assume that RDF graphs do not contain cycles. Thus, we can extract all possible path expressions from a DAG. Then, we construct suffix arrays on the path expressions. To this end, we newly introduce a suffix array for DAGs, which is an extension of suffix arrays for character strings. In fact, the proposed scheme can be adapted to the cases of general directed graphs. The algorithm will be shown later in Section 5.
2. To cope with problem 2), we define a path expression as an alternation of labels of vertexes and labels of arcs. In addition, we introduce special symbols to make a distinction among classes, properties and literals.
3. To cope with problem 3), we extract four kinds of subgraphs, namely, predicates in schema, predicates in resource descriptions, class inheritance and property inheritance graphs, from an original RDF graph. Then, we construct four kinds of suffix arrays for each subgraph. As a consequence, queries including schematic information can be processed by a collaboration of these suffix arrays. To answer such queries that include both schema and instance (e.g. find the titles of Paintings painted by the instances of Painter class), we first get the instances of the "Painter" class using class inheritance graph. We then get the titles of "Paintings" painted

by the instances of "Painters" using predicates in resource descriptions. Finally, we merge the answers and obtain the final result. In this way, complicated queries can be processed using our proposed indexing scheme.

3.3 Extracting DAGs from RDF data

Given an RDF data with RDF Schema, we extract four kinds of DAGs by taking vertex types, arc types and their semantics into account.

Predicates in schema This graph is obtained by extracting classes and their properties from the schema part of an RDF graph. This graph may contain cycles.

Predicates in resource descriptions This graph is obtained by extracting resources and their properties from the resource description part of an RDF graph. This graph may contain cycles.

Class inheritance This graph is obtained by extracting classes and "subClassOf" arcs connected to the classes. Note that "subClassOf" arcs do not have labels, and this graph does not contain cycles.

Property inheritance This graph is obtained by extracting properties and "subPropertyOf" arcs in the schema part of an RDF graph, and thus we let properties, which are arcs in the original graph, be vertexes in this subgraph. Note that "subPropertyOf" arcs do not have labels, and this graph does not contain cycles.

These subgraphs, except for predicates in the resource descriptions graph, cannot be obtained if RDF Schema is not provided. In those cases, we just use predicates in the resource descriptions graph. Otherwise, we can make full use of schematic information to query RDF data.

3.4 Path expressions

Figure 4 shows the syntax, represented in EBNF (Extended Backus-Naur Form), for path expressions. In the figure, schemaPath, instancePath, classPath and propertyPath correspond to path expressions extracted from predicates in schema, predicates in resource descriptions, class inheritance and property inheritance subgraphs, respectively. In the path expressions, '>' is used as a separator. Additionally, some special prefixes, '#', '+' and '$', are used to distinguish classes, properties and resources. If these special symbols are used in labels, we replace their occurrence with an entity reference of XML for encapsulation. For example, the RDF data in Figure 1 can be represented as

```
#www.matono.net/paper > +authored > "AkiyoshiMATONO"
```

based on the definition.

For a given DAG, we can extract all possible path expressions using the algorithm shown in Figure 5. This algorithm starts with the vertexes whose in-degree are zero (0), and search for traversable paths in a depth first manner.

```
paths          ::= schemaPath* | instancePath* |
                   classPath* | propertyPath*
schemaPath     ::= (classVertex '>' propertyVertex '>')*
                   classVertex
instancePath   ::= (resourceVertex '>' propertyVertex '>')*
                   literalVertex
classPath      ::= (classVertex '>')* instanceVertex
propertyPath   ::= (propertyVertex '>')* propertyVertex
classVertex    ::= '#' typeName
propertyVertex ::= '+' propName
instanceVertex ::= resourceVertex | literalVertex
resourceVertex ::= '$' URI-reference
literalVertex  ::= '"' literal '"'
typeName       ::= see [3]
propName       ::= see [3]
literal        ::= see [3]
URI-reference  ::= see [3]
```

Fig. 4. Path expression syntax (EBNF).

3.5 Suffix array for DAGs

An ordinary suffix array is a data structure for full-text search on documents constructed on one-dimensional character strings. Given a text data, all suffixes are extracted and sorted in lexicographical order. Any substring can then be detected by performing a binary search on the array of suffixes. In addition, because any suffix can be represented by an integer (an indexing point), the array of suffixes can be implemented as an array of integers whose size is equal to the length of the original document.

When applying a suffix array on path expressions, we need an extension that allows a suffix array to accommodate multiple path expressions. For this reason, we use a pair of integers as an indexing point; The first number is for representing an identifier of a path expression, and the other is for representing an indexing point within the path expression. It is defined as follows:

Definition 1 (Suffix array for DAGs) *Let G be a directed acyclic graph (DAG), $V(G)$ be the set of vertexes in G, and $E(G)$ be the set of arcs in G. Arc $e = (u, v)$ in $E(G)$ is represented by a pair of vertexes $u, v \in V(G)$, and u and v are called the "source" and "destination," respectively. In addition, let $R \subset V(G)$ be a set of vertexes whose in-degree is equal to zero (0), and $L \subset V(G)$ be a set of vertexes whose out-degree is equal to zero (0). We call R and L the "roots" and "leaves," respectively.*

Given a path on G from a root $s_{t,1} \in R$ to a leaf $s_{t,2k_t-1} \in L$, it can be represented as $p_t = s_{t,1}.s_{t,2}.\cdots.s_{t,2k_t-2}.s_{t,2k_t-1}$, where:

- *t is the identifier of the path,*
- *k_t is the length of the path,*
- *$s_{t,2h-1} \in V(G)$ $(1 \le h \le k_t)$, and*
- *$s_{t,2h} = (s_{t,2h-1}, s_{t,2h+1}) \in E(G)$ $(1 \le h \le k_t - 1)$.*

```
var $roots$ := a set of vertexes whose in-degree is 0
var $stack$ : Stack
foreach $start$ ( $roots$ ) begin
  $createPath$ ( $start$ )
end
function $createPath$ ( $start$ : vertex ) : Void
var $end$ : vertex
var $arcs$ : set of arcs
var $triple$ : tuple of (vertex, arc, vertex)
begin
  $arcs$ := a set of arcs connected from $start$ vertex
  foreach $arc$ ( $arcs$ ) begin
    $end$ := a vertex connected from $arc$
    $triple$ := ( $start$, $arc$, $end$ )
    $stack.push$ ( $triple$ )
    $createPath$ ( $end$ )
    $stack.pop$()
  end
  Creating a path expression based on $stack$
end
```

Fig. 5. An algorithm for extracting path expressions from DAGs.

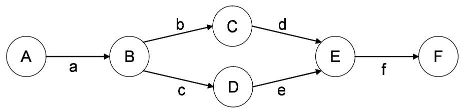

Fig. 6. A simple DAG.

A suffix of $p_j = s_{j,1}.s_{j,2}.\cdots.s_{j,2k_j-1}$ *is defined as* $S_{j,i} = s_{j,i}.s_{j,i+1}.\cdots.s_{j,2k_j-1} (i = 1, 2, \cdots, 2k_j - 1)$*, whose indexing point is* $a_{j,i} = [j, i]$.

The suffix array $S(p_j)$ *of the path* p_j *is then defined as an array of indexing points that is sorted in lexicographical order.*

The suffix array of a directed acyclic graph G is an array of indexing points, using all paths from roots $\{u|u \in R\}$ *to leaves* $\{v|v \in L\}$*, that is sorted in lexicographical order, and duplicated occurrences of the suffixes are eliminated.* □

We will demonstrate how a suffix array is constructed on a DAG using a simple example (Figure 6). From the DAG, we can extract two paths, namely, "A.a.B.b.C.d.E.f.F" and "A.a.B.c.D.e.E.f.F." Then, we assign indexing points to them (Figure 7), sort them in lexicographical order, and eliminate duplicates of identical suffixes (Figure 8). As a result, we obtain the suffix array of [1,1] [2,1] [1,3] [2,3] [1,5] [2,5] [1,7] [1,9] [1,2] [2,2] [1,4] [2,4] [1,6] [2,6] [1,8].

When processing queries, we perform binary searches on the suffix array. For this reason, $O(log_2(n + 1))$ of computational complexity is required.

	1	2	3	4	5	6	7	8	9
1	A .	a .	B .	b .	C .	d .	E .	f .	F
2	A .	a .	B .	c .	D .	e .	E .	f .	F

Fig. 7. Suffixes of paths.

A.a.B.b.C.d.E.f.F : (1, 1)		(1, 1) : A.a.B.b.C.d.E.f.F
a.B.b.C.d.E.f.F : (1, 2)		(2, 1) : A.a.B.c.D.e.E.f.F
B.b.C.d.E.f.F : (1, 3)		(1, 3) : B.b.C.d.E.f.F
b.C.d.E.f.F : (1, 4)		(2, 3) : B.c.D.e.E.f.F
C.d.E.f.F : (1, 5)		(1, 5) : C.d.E.f.F
d.E.f.F : (1, 6)		(2, 5) : D.e.E.f.F
E.f.F : (1, 7)		(1, 7) : E.f.F
f.F : (1, 8)		~~(2, 7) : E.f.F~~
F : (1, 9)	$\Rightarrow$	(1, 9) : F
A.a.B.c.D.e.E.f.F : (2, 1)		~~(2, 9) : F~~
a.B.c.D.e.E.f.F : (2, 2)		(1, 2) : a.B.b.C.d.E.f.F
B.c.D.e.E.f.F : (2, 3)		(2, 2) : a.B.c.D.e.E.f.F
c.D.e.E.f.F : (2, 4)		(1, 4) : b.C.d.E.f.F
D.e.E.f.F : (2, 5)		(2, 4) : c.D.e.E.f.F
e.E.f.F : (2, 6)		(1, 6) : d.E.f.F
E.f.F : (2, 7)		(2, 6) : e.E.f.F
f.F : (2, 8)		(1, 8) : f.F
F : (2, 9)		~~(2, 8) : f.F~~

Fig. 8. Sorting and deletion of suffixes.

4 Performance Evaluation

This section evaluates the performance of the proposed scheme in a series of experiments.

4.1 Experimental setup

Datasets We used RDF and RDF Schema documents of Wordnet [11] as the experimental data. Wordnet is an online lexical reference system whose design is inspired by current psycholinguistic theories of human lexical memory. English nouns, verbs, adjectives and adverbs are organized into synonym sets, each representing one underlying lexical concept.

As far as we have investigated, the RDF data of Wordnet does not contain any cycles, and thus we can apply our scheme directly to the datasets. We created subdocuments of them with different sizes 500 KB (Type A), 1 MB (Type B), 2 MB (Type C) and 4 MB (Type D). Table 1 shows the details of the datasets.

Table 1. Details of RDF documents of Wordnet

Type	A	B	C	D
Number of RDF Schema documents	1	1	1	1
Number of RDF documents	4	4	4	4
Total size of RDF Schema documents (KB)	4	4	4	4
Total size of RDF documents (KB)	513	999	2,073	3,982
Number of elements and attributes	15,089	29,542	62,565	119,368
Number of classes in RDF Schema documents	6	6	6	6
Number of properties in RDF Schema documents	5	5	5	5
Number of resources in RDF documents	1,555	3,100	6,571	12,380
Number of properties in RDF documents	5,647	10,851	22,773	42,878
Number of literals in RDF documents	4,553	8,645	18,107	33,473

Table 2. Performance evaluation queries

	Queries for predicates in schema	
#1	`+glossaryEntry>#`	Retrieval of classes for a given property
#2	`#LexicalConcept>+`	Retrieval of properties
#3	`#LexicalConcept>+antonymOf>#LexicalConcept>+hyponymOf>#LexicalConcept>+`	A long path expression
	Queries for predicates in resource descriptions	
#4	`+hyponymOf>#`	Retrieval of objects for a statement
#5	`#&wn;400062583>+wordForm>#`	Retrieval of statements
#6	`#&wn;100033830>+similarTo>#&wn;100033153>+wordForm>#`	A long path expression
	Queries for class inheritance	
#7	`#Adjective>$`	Retrieval of instances
#8	`#Resource#LexicalConcept>#Adjective>#AdjectiveSatellite>$`	A long path expression

Query sets The query expressions used in the experiments are shown in Table 2. In the table, " &wn;" is a character entity reference for representing the namespace of Wordnet. Using these queries, we intend to evaluate the following aspects: queries for predicate relations in schematic information (#1-#3); queries for predicate relations among instances (#4-#6); and queries for inheritance relations among classes (#7 and #8).

Methodology We used an RDF database, RDFSuite [6], as a basis for implementing our (proposed) scheme. RDFSuite is implemented on top of PostgreSQL, an open source relational database management system. Specifically, RDFSuite supports two kinds of relational schemas, *GenRepr* and *SpecRepr*, for storing RDF data. *GenRepr* has two relational tables; *Resources* is for storing resources and their identifiers, and *Triples* is for storing triples extracted from statements. On the other hand, *SpecRepr*'s relational schema is designed according to the RDF Schema of the RDF data being stored. In our experiments, we used *SpecRepr* because it is more efficient than *GenRepr* from the view point of performance.

Table 3. The number of path expressions and arrays of index-points

Description	A	B	C	D
# paths (total)	9,709	19,480	43,008	90,058
# suffixes (total)	25,977	51,060	111,108	217,549
# paths (preds in schema)	10	10	10	10
# suffixes (preds in schema)	21	21	21	21
# paths (preds in resource descs)	8,144	16,370	36,427	77,668
# suffixes (preds in resource descs)	19,409	37,999	83,459	165,512
# paths (class inheritance)	1,555	3,100	6,571	12,380
# suffixes (class inheritance)	6,547	13,040	27,628	52,016
# paths (property inheritance)	5	5	5	5
# suffixes (property inheritance)	10	10	10	10

We compared the query processing time between RDFSuite and RDFSuite powered by our indexing scheme as follows:

1. We store each dataset in RDFSuite based on SpecRepr schema. Then, we construct suffix arrays on the relational tables of RDFSuite. Specifically, a table for storing all path expressions extracted from Wordnet data, and four tables for storing indexing points are created in the relational database.
2. We then measure the query processing time of the queries in Table 2 for the two cases, pure RDFSuite and RDFSuite powered by suffix arrays.

We used a PC with an Athlon 1.1 GHz CPU and 768 MB memory running RedHat Linux 8.0, and used Java 1.4.1 for the implementation.

4.2 Experimental results

Table 3 shows the statistical data of the generated suffix arrays. From the table, we can observe that the number of path expressions and suffixes increase in proportion to the sizes of the datasets for the cases of “predicates in resource descriptions” and “class inheritance.” However, this is not the case for “predicates in schema” and “property inheritance,” because this information solely depends on RDF Schema, and RDF Schema is fixed for the experiments in this paper.

Figure 9 shows ratios (N/I) of the processing time of RDFSuite (N) to our scheme (I). That is, our approach is about four times faster than RDFSuite with respect to query #2 for dataset A. It is clear that our scheme outperforms RDFSuite.

Table 4 shows the details of the processing times. Note that for the case of #1 – #3, because the dataset is small, the absolute processing times are too short, and the results may not be reliable compared to other results.

Our scheme can process #4 – #6 in almost the same time, whereas RDFSuite does not. In particular, #4 is slower than others (#5 and #6). This is because #4 searches objects for a given predicate, while #5 and #6 search objects for a given pair of subject and predicate. For this reason, RDFSuite can make use of built-in indices to process the queries.

Table 4. Processing time

Type	A		B		C		D	
Suffix array	Yes	No	Yes	No	Yes	No	Yes	No
#1	30.7	30.0	25.6	27.0	25.9	28.2	28.4	28.2
#2	27.6	109.0	26.0	97.3	26.0	96.0	28.0	92.0
#3	23.2	164.7	24.3	162.6	24.8	180.4	25.1	158.0
#4	99.5	396.5	130.6	707.3	205.9	1274.9	337.9	2325.8
#5	82.8	166.6	113.6	217.4	182.7	357.5	312.3	541.9
#6	84.7	182.1	108.0	231.6	178.0	385.9	300.7	646.4
#7	71.4	237.6	80.3	334.2	114.1	702.7	142.3	1238.7
#8	77.0	263.0	91.3	447.7	122.0	579.6	160.7	816.9

When processing queries for inheritance between class and instance (#7 and #8), as the data size is large, the ratios of the processing time between our scheme and RDFSuite are larger. In other words, our scheme achieved scalability.

5 Coping with Cycles

In this paper, we limited our targets within directed acyclic graphs. We think that we can find many other RDF data without cycles, because even a large scale data like Wordnet does not contain cycles. Consequently, our scheme can be used for many applications. However, some RDF data with directed graph structures with cycles also exist. Thus, it is important to be able to cope with cycles in order to widen the applications of our scheme.

5.1 Path expressions extraction and index construction

When applying indices based on suffix arrays for querying graphs, we need to extract all possible path expressions beforehand. However, the previous algorithm for extracting path expressions cannot cope with graphs that include cycles, because it may not terminate due to dissatisfaction of terminal conditions. For this reason, we made some improvements on the algorithm so that it can extract all the vertexes and arcs thoroughly.

The algorithm in Figure 12 has two features as follows: 1) if a path expression contains two (or more) identical vertexes, a *loop-stamp(s)* is put on their second (and later) occurrence; and 2) we change our strategy to decide the starting positions of the path expressions. Actually, we make a list of vertexes whose in-degrees are equal to zero (0), followed by vertexes ordered by the differences between the out-degrees and in-degrees in ascending order. Starting from these vertexes, we try to enumerate path expressions until all the vertexes and arcs are included.

Let us take a look at such an example. Figure 10 illustrates a graph including a cycle, and Figure 11 shows two path expressions extracted from the graph using the algorithm in Figure 12. Note that '^' is the *loop-stamp*.

We then create suffixes with respect to those path expressions and sort them in lexicographic order. Finally, we get the following suffix array: [2,1] [1,1] [2,3] [1,3]

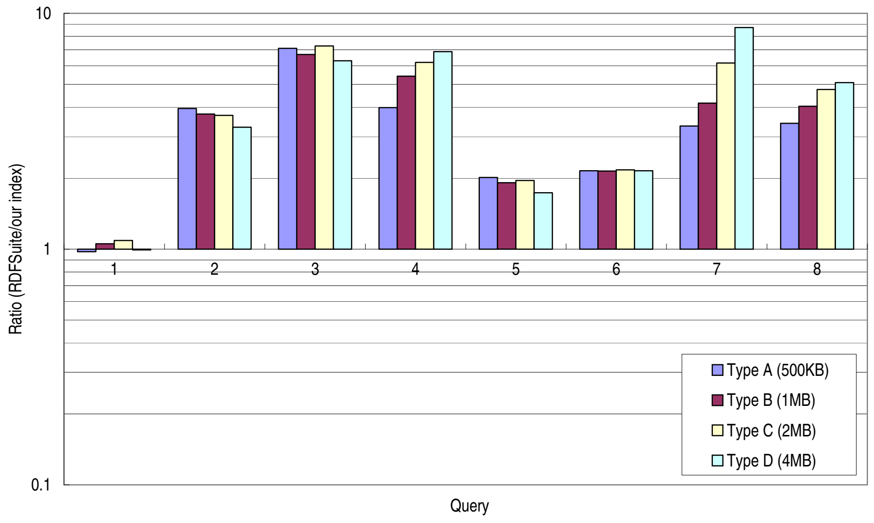

Fig. 9. Processing time (RDFSuite / Suffix array)

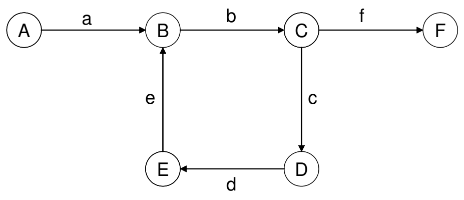

Fig. 10. Directed graph including a cycle

[2,5] [1,5] [2,7] [2,9] [1,7] [2,11] [2,2] [1,2] [2,4] [1,4] [2,6] [2,8] [2,10] [1,6], whose length is 18.

5.2 Query processing

When processing queries based on path expressions against a graph with cycles, handling unintended termination of the path expressions is crucial. That is, path expressions listed in a suffix array are not powerful enough to express cycles because of the limitation of their expressiveness. As a consequence, we may come to the end of such a termination when we are matching a query key and a path expression in the index, and may thus miss correct answers.

If a query `#E>+e>#B>+b>#C>+c>#D` is given, we cannot find the same occurrence in the suffix array, although it is a correct answer. When processing this query, we

	1	2	3	4	5	6	7	8	9	10	11
1	#A>	+a>	#B>	+b>	#C>	+f>	#F				
2	#A>	+a>	#B>	+b>	#C>	+c>	#D>	+d>	#E>	+e>	#ˆB

Fig. 11. Path expressions and indexing points of the suffixes

start from the starting element (#E) and proceed as much as possible as usual. Then, we get the indexing point [2,9] (#E>+e>#ˆB). This intermediate result partially patches until #E>+e>#B. Eventually, we come to the *loop-stamp*. Then, we decompose the query path expression here, let the following path expression (#B>+b>#C>+c>#D) be a new query, and initiate it. As a result of the brand-new query, we get a suffix [2,3] (#B>+b>#C>+c>#D>+d>#E>+e>#ˆB). Now, the initial query key is fulfilled, and we get a result of the query.

We expect that we can achieve efficient retrieval for a directed graph with cycles using the indexing scheme. However, we may have to improve the scheme, because the number of path expressions and suffixes are increasing in the case of the target data that contains many cycles.

6 Related Work

Indexing techniques for structured documents are classified into a *position-based index* and *path-based index* according to Sacks-Davis et al.w [15].

The indices proposed by Kanemoto et al. [16] and Shin et al. [17] are position-based indices. Kanemotno et al. [16] proposed an approach in which four indices are combined to achieve efficient document retrieval. The indices were a *content index* for maintaining positions of elements and contents, *local structure index* for maintaining the tree structure of document instances, *global structure index* for maintaining the tree structure of document schema, and *structure meta index* for maintaining the meta information of the other indices. Although this approach was efficient, the performance did not scale with respect to data size, because four kinds of indices must be joind. In the study by Shin et al. [17], an indexing scheme called *BUS (Bottom Up Scheme)* was proposed. In this approach, document features were maintained in a bottom-up manner.

Path-based indices were proposed by Yamamoto et al. [10], Kaushik et al. [18] and Cooper et al. [19]. The study by Yamamoto et al. [10] is a basis of our proposal. In this paper, given an XML document, all possible path expressions were extracted, and suffix arrays were constructed on path expressions and reverse path expressions, and hence efficient processing of path expressions (and reverse path expressions) was achieved. In the study by Kaushik et al. [18], they created compact models from document trees by grouping similar vertexes into one vertex. Query processing was performed using path expressions on the compact models. That is, they achieved space efficient indexing by giving up accuracy. An indexing scheme called *Index Fabric*, as an extension of *Patricia trie* [20], was proposed by Cooper et al. [19]. *Patricia trie* was an efficient and compact indexing scheme that could deal with large-size text. *Index Fabric* is an extension of *Patricia trie*, and is a height-balanced indexing structure for semi-structured data, In addition, combinations of position- and path-based indices were proposed by Sacks-Davis et al. [15] and McHugh et al. [21]. In Sacks-Davis et al.'s study [15], a position-based index was constructed as inverted lists consisting of all words and elements, and a path-based index represented the list of element names and their positions for each path. In McHugh et al.'s study [21], four indices were proposed, namely, the *Value Index*, *Text Index*, *Link Index* and *Path Index*. Value Index has pairs of values and element names. Text Index is implemented as an inverted list of text. Link Index maintains information on a list of children for each element. Finally, Path Index has path information for all elements.

Path-based indices were also used in object databases [22–24]. Similar to our approach, these approaches maintain the relationships of class hierarchies and/or object composition hierarchies. The key difference between our scheme and their approaches is that we treat path expressions as character strings, whereas the others do not.

Christophides et al. [25] have proposed a labeling scheme for efficient retrieval of RDF Schema. The study relevant to our research. They applied previously proposed labeling schemes for tree structures to RDF schema. Concretely, there approach employed the study by Agrawal et al. [26], in which an optimal spanning tree is generated from a DAG based on the number of ancestors per node, so that it can handle DAGs. The labeling schemes investigated in [25] are classified into bit vector, prefix and interval scheme. Bit vector [27] is a labeling scheme in that a node is represented by a *n* bits vector. Prefix scheme directly encodes the label of a node in an XML tree, in that the prefix is inherited by the parent's label followed by the order of the node in its siblings. Dewey scheme [28] is one of prefix scheme. Interval scheme [29, 26, 30] encodes the interval label (start, end) such that it is contained in its parent's interval label. Christophides et al. [25] limited the target data as RDF Schema for efficient retrieval. Making a comparison between their scheme and ours is an interesting topic. We plan to do it in the near future.

7 Conclusions

In this paper, we proposed an indexing scheme to enable RDF and RDF Schema to achieve efficient query retrievals on path expressions. To this end, we first proposed four types of partial graphs that can be obtained from RDF and RDF Schema. In addition, we proposed suffix arrays on DAGs. By applying this scheme to path expressions extracted from the above graphs, we achieve efficient RDF query processing. Because most of the RDF and RDF Schema in real applications are expected to be modeled as DAGs, we can make use of our proposed scheme. We conducted a performance study and the results showed that our approach outperformed an existing RDF database, RDFSuite.

In the future, we will try to deal with RDF data that include cycles and investigate query optimization techniques for RDF queries. Since our indexing scheme needs to precompute all paths as statical indexing data, we must consider an update of RDF data and schema.

References

1. World Wide Web Consortium: Semantic Web. http://www.w3c.org/2001/sw/ (2001)

2. Berners-Lee, T.: What the Semantic Web can represent. http://www.w3.org/DesignIssues/RDFnot.html (1998)
3. World Wide Web Consortium: Resource Description Framework(RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ (1999) W3C Recommendation 22 February 1999.
4. World Wide Web Consortium: Resource Description Framework(RDF) Schema Specification 1.0. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/ (2000) W3C Candidate Recommendation 27 March 2000.
5. World Wide Web Consortium: Survey of RDF/Triple Data Stores. http://www.w3.org/2001/05/rdf-ds/DataStore (2001)
6. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The RDFSuite: Managing Voluminous RDF Description Bases (2000)
7. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: Proceedings of the eleventh international conference on World Wide Web, ACM Press (2002) 592–603
8. McBride, B.: Jena: Implementing the RDF Model and Syntax Specification. In: Proceedings of the Second International Workshop on the Semantic Web - SemWeb'2001. (2001)
9. Hewlett-Packard Company: RDQL – RDF Data Query Language. (http://www.hpl.hp.com/semweb/rdql.htm)
10. Yamamoto, Y., Yoshikawa, M., Umeura, S.: On Indices for XML Documents with Namespaces. In: Conference Proceedings of Markup Technologies '99, GCA, Philadelphia, U.S.A. (1999)
11. Miller, G.A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K.: Introduction to WordNet: An On-Line Lexical Database. http://www.cogsci.princeton.edu/ wn/ (1993)
12. Berners-Lee, T., Fielding, R.T., Masinter, L.: Uniform Resource Identifiers (URI): Generic Syntax. http://www.isi.edu/in-notes/rfc2396.txt (1998) RFC2396.
13. Yoshikawa, M., Amagasa, T., Shimura, T., Uemura, S.: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Transactions on Internet Technology (TOIT) **1** (2001) 110–141
14. Jiang, H., Lu, H., Wang, W., Yu, J.X.: Path Materialization Revisited: An Efficient Storage Model for XML Data. In Zhou, X., ed.: Thirteenth Australasian Database Conference (ADC2002), Melbourne, Australia, ACS (2002)
15. Sacks-Davis, R., Dao, T., Thom, J.A., Zobel, J.: Indexing Documents for Queries on Structure, Content and Attributes. In: Proceedings of the International Symposium on Digital Media Information Base, Nara, Japan. (1997) 236–245
16. Kanemoto, H., Kato, H., Kinutani, H., Yoshikawa, M.: An Efficiently Updatable Index Scheme for Structured Documents. In: Proceedings of 9th International Workshop on Database and Expert Systems Applications (DEXA'98), IEEE Computer Society. (1998) 991–996
17. Shin, D., Jang, H., Jin, H.: BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. In: Proceedings of the Third ACM Conference on Digital Libraries. (1998) 235–243
18. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In: ICDE. (2002)
19. Cooper, B., Sample, N., Franklin, M.J., Hjaltason, G.R., Shadmon, M.: A Fast Index for Semistructured Data. In: The VLDB Conference. (2001) 341–350
20. Knuth, D.E.: The Art of Computer Programming Volume 3 Sorting and Searching, Second Edition. Addison-Wesley (1998)
21. McHugh, J., Widom, J., Abiteboul, S., Luo, Q., Rajamaran, A.: Indexing Semistructured Data. Technical report, Stanford University, Computer Science Department. (1998)

22. Bertino, E.: Index Configuration in Object-Oriented Databases. VLDB Journal **3** (1994) 355–399
23. Lee, W., Lee, D.: Path Dictionary: A New Approach to Query Processing in Object-Oriented Databases (1995)
24. Xie, Z., Han, J.: Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases. In Bocca, J.B., Jarke, M., Zaniolo, C., eds.: VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, Morgan Kaufmann (1994) 522–533
25. Christophides, V., Plexousakis, D., Scholl, M., Tourtounis, S.: On labeling schemes for the semantic web. In: Proceedings of the twelfth international conference on World Wide Web, ACM Press (2003) 544–555
26. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In Clifford, J., Lindsay, B.G., Maier, D., eds.: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989, ACM Press (1989) 253–262
27. Wirth, N.: Type Extensions. ACM Transactions on Programming Languages and Systems **10** (1988) 204–214
28. Online Computer Library Center: Dewey Decimal Classification. (http://www.oclc.org/dewey/)
29. Dietz, P., Sleator, D.: Two algorithms for maintaining order in a list. In: Proceedings of the nineteenth annual ACM conference on Theory of computing, ACM Press (1987) 365–372
30. Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Path Expressions. In: The VLDB Journal. (2001) 361–370

```
var stack : stack /* for storing triple */
var roots1 : list of vertexes /* whose in-degree = 0 */
var roots2 : list of vertexes /* sorted by values of (in-degree − out-degree)
                                 into descending order − roots1 */
var roots := append ( roots1, root2 )
foreach start ( roots ) begin
  createPath ( start )
end
function searchGraph ( start : vertex ) : Void
var end : vertex
var arcs : set of arcs
var triple : tuple of ( vertex, arc, vertex )
begin
  roots.remove ( start )
  arcs := a set of arcs connected from start
  foreach arc ( arcs ) begin
    end := a vertex connected from arc
    roots.remove ( end )
    triple := ( start, arc, end )
    /* a path expression does not include same statements. */
    if ( stack = nil or triple ∉ stack.items ) then
      stack.push ( triple )
      searchGraph ( end )
      stack.pop ()
    end
  end
  var path : path expression
  var loop_stamp : loop-stamp /* for representing a path expression containing cycles */
  path := concat ( path, stack[0].start )
  for (var i := 0; i < stack.length; i := i + 1) then
    var vertex := stack[i].end
    /* When a path expression has same vertexes */
    if ( vertex ∈ path.items ) then
      vertex := vertex + loop_stamp
    end
    path := concat( path, stack[i].arc, vertex )
  end
end
```

Fig. 12. An algorithm of creating path expressions for DG