A Path-based Relational RDF Database

Akiyoshi Matono† Toshiyuki Amagasa† Masatoshi Yoshikawa‡ Shunsuke Uemura†

†Graduate School of Information Science
Nara Institute of Science and Technology
Nara 630–0192, Japan
{akiyo-ma,amagasa,uemura}@is.naist.jp

‡Information Technology Center
Nagoya University
Nagoya 464–8601, Japan
yosikawa@itc.nagoya-u.ac.jp

Abstract

We propose a path-based scheme for storage and retrieval of RDF data using a relational database. The Semantic Web is much anticipated as the next-generation web where high-level processing of web resources are enabled by underlying metadata described in RDF format. A typical application of RDF is to describe ontologies or dictionaries, but in such applications, the size of RDF data is large. As large-size RDF data are emerging and their number is increasing, RDF databases that can manage large-size RDF data are becoming ever more important. To date, some RDF databases have already been proposed; however, they have critical problems: the performance of path queries is insufficient and they cannot discriminate between schema data and instance data. In this paper, as a solution to these problems, we propose a path-based relation RDF database. In our approach, we first divide the RDF graph into subgraphs, and then store each subgraph by applicable techniques into distinct relational tables. More precisely, all classes and properties are extracted from RDF schema data, and all resources are also extracted from RDF data. Each is assigned an identifier and a path expression, and stored in corresponding relational table. Because our proposed scheme retains schema information and path expressions of each resource, unlike most conventional RDF databases, it is possible to process path-based queries efficiently and store RDF instance data without schema information. The effectiveness of this approach is demonstrated through several experiments.

Keywords: RDF, RDF database, path expressions, numbering scheme, relational database

1 Introduction

The Semantic Web has emerged as a vision of 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 because of the wealth of metadata on Web resources. One of the key differences between the current Web and the Semantic Web is the quality and quantity of metadata: the quality and quantity of the currently available metadata are insufficient for advanced processing. The Semantic Web, on the other hand, makes it possible to perform such high-level processes as reasoning, deduction, and semantic searches to maximize metadata associated with Web resources.

In the Semantic Web, metadata are described by Resource Description Framework (RDF) [1] (World Wide Web Consortium 2004*b*), which describes data and their semantics. The specification defines an RDF model and RDF syntax. In an RDF model, *statements* describe a relationship between a pair of terms. By using a set of statements, we can represent metadata whose structure is a directed graph.

Today, it is becoming increasingly common to use RDF as a format to describe various types of metadata. One typical usage is to describe large-scale metadata, such as ontologies and dictionaries. Some examples of these are: Wordnet (Miller, Beckwith, Fellbaum, Gross & Miller 1993), an online dictionary, whose total size is 35 MB; Gene Ontology (GENE ONTOLOGY CONSORTIUM n.d.) , a controlled vocabulary for functional annotation of gene products, whose size is 365 MB in total; and Open Directory Project (ODP) (Netscape n.d.), which provides a human-edited Web directory in RDF format, and whose size is over 2 GB in total. In the near future, such large RDF-based metadata is considered to increase rapidly as RDF comes into wide spread use. In order to handle such data efficiently, RDF databases that can manage massive RDF data are essential.

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 serialized as XML data. However, this approach is impractical because the structure of semantics as RDF data is different to the structure of syntax as XML data, and the semantics cannot be stored into XML databases.

Another way to implement RDF databases is to utilize relational databases or Berkeley DB. So far, several RDF databases have been proposed (Alexaki, Christophides, Karvounarakis, Plexousakis & Tolle 2000) (Beckett 2001) (Broekstra, Kampman & van Harmelen 2002) (Wilkinson, Sayers, Kuno & Reynolds 2003) (Miller n.d.) (Reggiori n.d.) (R.V. Guha n.d.). Such conventional RDF databases can be classified into two groups by how the RDF schema is handled. In the first group, schema data for underlying data storage are designed based on RDF schema. From this, when processing schema queries, they do not need to search instance data and thus are capable of processing efficiently. However, they cannot handle such RDF data that do not have accompanying RDF schema. On the other hand, RDF databases in the second group store RDF data in terms of triples.

Processing large RDF data using such conventional RDF databases could cause several problems in aspects of both capability and performance. The first problem concerns the ability to handle RDF schema. Obviously, an RDF query using information about

[1]XML Topic Maps (XTM) (Members of the TopicMaps.Org Authoring Group 2001) are also used to describe metadata in the Semantic Web. However, we focus on RDF in this paper.

RDF schema is one of the most important classes of RDF queries. "Check if Painter class inherits Artist class" is such an example. In fact, it is difficult for RDF databases in the second group to process schema queries. Because they do not make any distinction between schema information and instance data, it is necessary to search the superfluous area, where both data are stored, for processing of schema queries. On the other hand, RDF databases in the first group can process such queries, although they cannot store RDF data that do not have RDF schema data.

The second problem is poor performance in processing path queries. A path query is a class of RDF query; a path query is composed of a list of statements, and the answer is a set of resources that are reachable via statements specified in the given query. The reason for its poor performance is that most of the conventional RDF databases utilize a relational database or Berkeley DB[2] as the underlying data storage, where RDF data are decomposed into statements. Therefore, when processing a path query, we need to perform a join operation per each path step. This results in performance degradation as RDF data grows and/or query length becomes longer.

In this paper, as a solution for the problems, we propose a path-based relational RDF database. Our proposed relational schema is designed to be independent of RDF schema information, and designed to make the distinction between schema information and instance data. For this reason, our RDF database can handle schemaless RDF data as well as RDF data with schema. Moreover, to improve performance for path queries, we extract all reachable path expressions for each resource, and store them. We thus do not need to perform join operations unlike the abovementioned existing RDF databases.

We first classify every statement as a triple in the form (*subject*, *predicate*, *object*) into categories according to the type of predicate. We then construct subgraphs for each category. Finally, we store the subgraphs into distinct relational tables by applying appropriate techniques for representing the semantics of each subgraph. In particular, we extract all reachable path expressions from a subgraph that represents instance data, and store them in a relational table. In addition, we apply an extended version of the interval numbering scheme to subgraphs which represent schema information, enabling us to efficiently detect ancestor-descendant or parent-child relationships between two classes (or properties). Note that we limit the structure of a subgraph of them is directed acyclic graph. As with RDF classification indicated above, RDF data in our target group have a structure that hardly contains any cycle; as far as we know, a large majority of RDF data in real applications is expressible as directed acyclic graphs. Thus, we also claim that our scheme can be applied to many applications.

We evaluate its performance in this paper. In our experiments, we use the Gene Ontology as the experimental data set, and compare our RDF database with the RDF database Jena2 (Wilkinson et al. 2003) in processing times of path queries. We demonstrate that our approach can achieve effectiveness of long path expression queries through several experiments.

The rest of the paper is organized as follows. Section 2 briefly summarizes the RDF data and RDF Schema to define the schema of RDF using examples. Related work is discussed in Section 3. Section 4 describes the storage of RDF data using relational tables. Implementation and evaluation of our scheme are described in Section 5. Section 6 summarizes our study.

[2]Berkley DB is an open-source database designed based on hash function and/or B-tree. `http://www.sleepycat.com/`

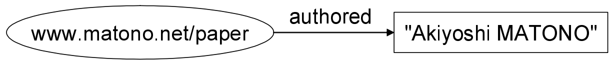

Figure 1: A statement from an RDF data model.

2 An Overview of RDF

In this section, we outline the RDF specifications and a schema definition language called RDF Schema in Section 2.1. In Section 2.2, we discuss the characteristics of conventional RDF data on the web and then classify RDF data by considering its characteristics.

2.1 RDF Specification

The Resource Description Framework (RDF) (World Wide Web Consortium 2004*b*), a foundation for representing and manipulating metadata on Web resources, is emerging for describing metadata of Web resources. It is usable as long as the location of a Web resource is identifiable in terms of a Uniform Resource Identifier (URI). In RDF, *statements* represent binary relationships between two distinct (or maybe identical) resources. Since complicated information can be represented by a set of statements, Thus, RDF data are modeled as a directed graph where nodes and arcs represent resources and relationships, respectively. For example, let us take a look at the statement "This paper is authored by Akiyoshi MATONO." in Figure 1. The statement consists of three parts; a *subject* ("This paper"), a *predicate* ("is authored by") and an *object* ("Akiyoshi MATONO"). For this reason, the statement is also called a triple.

RDF Schema (World Wide Web Consortium 2004*a*) is a specification for defining schematic information of RDF data. Specifically, we can define: 1) classes (*rdfs:Class*) as types of resources; 2) properties of a class (*rdf:Property*); 3) domains (*rdfs:domain*) and ranges (*rdf:range*) of the properties; and 4) inheritance relationships (*rdfs:subClassOf* and *rdfs:subPropertyOf*) among classes or properties. Classes and properties defined using RDF Schema are used as types (*rdf:type*) of resources in RDF data. In other words, RDF data is an instance of RDF Schema data, and the implicit vocabularies used in RDF Schema data definition are meta schematic information. Moreover, not only classes and properties but also the implicit vocabularies in the RDF specification inherit resources (*rdfs:Resource*).

Using RDF and RDF Schema, we can represent complex information. Figure 2 shows an example of an RDF graph with RDF schema. The upper part displays a meta schema, that is, the implicit information to define the schematic information detailed in the specification of RDF Schema. Such schematic information, such as classes and properties, are shown in the middle part. The property "creates" takes an "Artist" and an "Artifact" as its domain and range, respectively. "Painter" and "Sculptor" are subclasses of "Artist." Likewise, "sculpts" and "paints" are subproperties of "creates," and so on. The resource descriptions can be found in the lower part. The resources, such as "r1" and "r2," are defined as instances of "Painter" and "Painting" classes, respectively. Character strings, such as "Pablo" and "Picasso," are defined as literals.

2.2 A Classification of RDF Data

In this section, we discuss the characteristics of size and structure of widely-used RDF data such as RSS

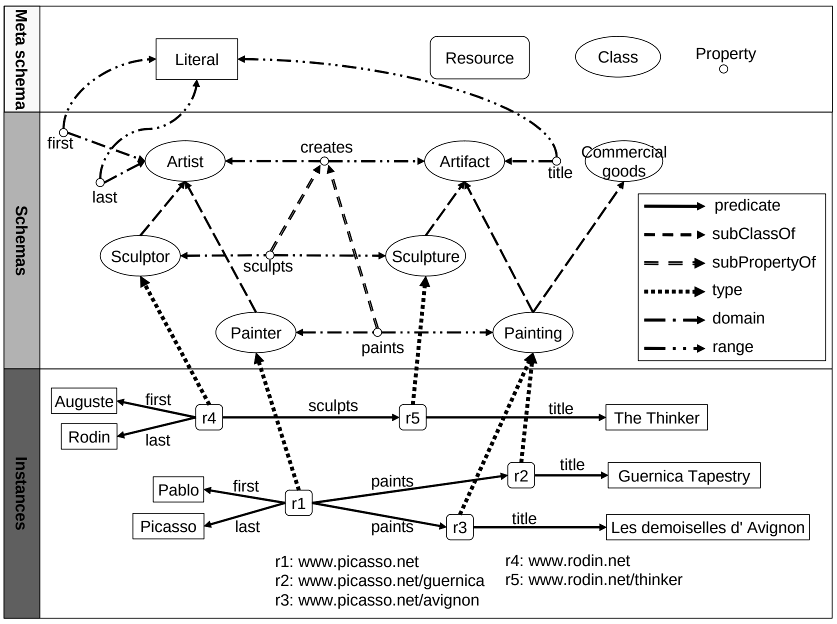

Figure 2: Example of RDF graph with RDF Schema.

(RSS-DEV Working Group 2000), FOAF (Dan Brickley and Libby Miller 2003), Dublin Core (Dublin Core Metadata Element Set, Version 1.1 2003), Wordnet (Miller et al. 1993), Open Directory Project (ODP) (Netscape n.d.), and Gene Ontology (GENE ONTOLOGY CONSORTIUM n.d.).

When examining at RDF data carefully, we can classify to two kinds of predicate: *terminal predicates* and *divergence predicates.* A node as an object through a *terminal predicate* does not have outgoing links, or the number of the path converges from the statement. Generally, a *terminal predicate* has a literal as the object, and there is an inclusive relationship between its subject and object. In other words, a *terminal predicate* is the function as a supplementary explanation of the resource as subject. "Name" and "belongings" are such examples. All the predicates defined in the middle in Figure 2 are *terminal predicates.* On the other hand, a *divergence predicate* means that the number of the path diverges from the statement including the predicate, and the number of branches increase enormously. Generally, in a statement including a *divergence predicate*, the subject and the object are instances of a similar level class. Thus, a *divergence predicate* is used to represent an equivalent relationship between the subject and the object, such as "friend" and "related work."

We can classify RDF data currently in use into two groups by size. RDF data of large size such as Wordnet, ODP and Gene Ontology belong to the first group. RDF data in this category is created mainly for systematical organization of data resources such as vocabularies and genomic data. To the best of our knowledge, this kind of RDF data do not contain cycles, and the structure is simple. Actually, the structures of the above six RDF formats are trees or directed acyclic graphs, because most predicates in a RDF data are *terminal predicates*; *divergence predicates* are rarely included in RDF data. It is difficult to compose an RDF data with a large number of *divergence predicates.*

The second group contains RDF data of small size such as RSS, FOAF, and Dublin Core. These RDF data are usually used as metadata of images or Web pages. The latter is more useful when distributed over a network and connected by hyperlinks, rather than used stand-alone.

In this paper, we focus on RDF data in the first category; our scheme deals with RDF data that do not contain cycles. We insist that this assumption does not affect the applicability of the proposed scheme, because our target RDF data seldom contain cycles.

3 Related Work and the Differences with Our Work

Several RDF databases have already been proposed (Alexaki et al. 2000) (Beckett 2001) (Broekstra et al. 2002) (Wilkinson et al. 2003) (Miller n.d.) (Reggiori n.d.) (R.V. Guha n.d.), most of which use relational databases or Berkeley DB as their underlying data storage. For relational databases, we can make further categorization of the ways relational schemas are designed. One approach flatly stores statements into a single relational table. The other creates relational tables for classes and properties that are defined in the RDF schema information, storing resources according to their classes (or properties). On the other hand, the approaches using Berkeley DB create three hash tables whose keys are subjects, predicates, and objects; we can retrieve corresponding statements using these keys.

Table 1 summarizes the approaches: *flat* represents the approaches that store statements flatly into a single relation; *schema* represents the approaches that design relational tables for classes and properties based on RDF schema; and *hash* represents the approaches that use the three hash tables implemented by Berkeley DB.

The problems of the conventional approaches are: 1) Using the *flat* and *hash* approaches, it is difficult to perform schema queries because they do not make any distinction between schema information and resource descriptions. Thus, it is necessary to repeat queries composed from the previous answers; 2) The *schema* approach is able to process queries about RDF schema. However, the approach cannot handle RDF data without RDF schema information because the relational schema is designed based on RDF schema information. Additionally, it is costly to maintain schema evolution because we have to change the relational schema depending on the changes of

Table 1: Storage methods of RDF databases.

	Storage method	Database
RDFSuite (Alexaki et al. 2000)	flat, schema	RDBMS
Redland (Beckett 2001)	hash	Berkeley DB
Sesame (Broekstra et al. 2002)	schema	RDBMS
Jena2 (Wilkinson et al. 2003)	flat	RDBMS
Inkling (Miller n.d.)	flat	RDBMS
RDFStore (Reggiori n.d.)	hash	Berkeley DB
rdfDB (R.V. Guha n.d.)	hash	Berkeley DB

RDF schema; and 3) The capabilities of the three approaches for processing path-based queries are not sufficient.

We should make some remarks about the final issue. In conventional RDF databases, statement-based queries can be processed efficiently because RDF data is decomposed into a large number of statements that are directly stored in relational (or hash) tables; a statement-based query lacks one or two parts of a triple, and the answer is a set of resources that complement the lacked statements. However, we assert that path-based queries, whose path expressions are given as conditions to filter parts of interest, are more important than statement-based queries (Matono, Amagasa, Yoshikawa & Uemura 2003). When processing a path-based query, conventional approaches require a number of join operations according to the steps in the path expression.

Based on the above observations, we propose a path-based approach for storing RDF data in relational databases. For given RDF data, we first extract five subgraphs according to the types of predicates: class-inheritance (CI) (*rdfs:subClassOf*), property-inheritance (PI) (*rdfs:subPropertyOf*), type (T) (*rdf:type*), domain-range (DR) (*rdfs:domain* and *rdfs:range*), and any others (G). In fact, our relational schema is based on the subgraphs. As we can see, schema information is managed separately from resource information, and we can process queries about schema information. In addition, to improve the speed of query processing of classes and properties, we apply an extended interval numbering scheme to subgraphs for classes and properties to efficiently detect ancestor-descendant or parent-child relationships between two classes (or properties).

In addition, we have a supplemental relation *path* for storing path expressions of every resource. From a graph G, which will be explained later, we extract all possible path expressions and store them in a *path* table. It is therefore easy to find resources that match given path expressions.

Our relational schema is fixed, since it does not depend on the RDF schema, although we can process queries about the schema information. This is completely different from the schema approach described above.

4 Path-based Approach for Storing RDF Data in Relational Databases

4.1 Subgraph extraction from RDF graph

RDF data are composed of the following components: RDF-meta schema data consisting of axiomatic statements defined in the RDF Schema specification; RDF schema data as instances of the RDF-meta schema data; and RDF data as instances of the RDF schema data. When storing RDF data, our proposed scheme parses the RDF data and generates its own RDF graph. It then decomposes the graph into five subgraphs according to the type of predicate. The subgraphs are as follows:

- **Class Inheritance (CI) graphs** are composed of statements whose predicates represent the inheritance relationships between classes (*rdfs:subClassOf*). A graph CI is thus a single-labeled directed acyclic graph (DAG) where the nodes and arcs correspond to the classes and their class-subclass relationships, respectively.

- **Property Inheritance (PI) graphs** are composed of statements whose predicates represent inheritance relationships between properties (*rdfs:subPropertyOf*). A graph PI is thus a single-labeled directed acyclic graph, where the nodes and arcs correspond to the properties and their property-subproperty relationships, respectively.

- **Type (T) graphs** are composed of statements whose predicates represent resources' type information (*rdf:type*). A graph T is a single-labeled directed acyclic graph where each arc *rdf:type* has a resource as its source, and a class as its destination.

- **Domain-Range (DR) graphs** are composed of statements whose predicates represent the domain (*rdfs:domain*) or the range (*rdfs:range*) of each property. This graph is thus a directed acyclic graph where each arc, whose label is *rdfs:domain* or *rdfs:range*, has a property as its source and a class as its destination.

- **Generic (G) graphs** consist of all the remaining statements not included in the above subgraphs. User-defined properties and other properties, such as *rdfs:seeAlso* and *rdfs:isDefinedBy*, are included in this subgraph. In general, a graph G is a labeled graph that may contain cycles. However, for simplicity, we assume that graph G does not contain cycles.

Dividing an RDF graph into the above subgraphs brings some advantages: 1) We can store RDF data into distinct relational tables. In other words, we can make distinctions between schema information and instance data. We can thus design relational schema to be independent of RDF schema information. Our relational schema, which appears in the following section, is based on these subgraphs. We basically store the subgraphs into six specially designed relational tables (in Section 4.4) by considering their characteristics; 2) The structures of the resulting subgraphs are less complex than the original RDF graphs, so we have opportunities to apply several techniques for representing each subgraph by considering graph structure. Specifically, we extract and retain all possible path expressions from graph G for efficient processing of path-based queries (in Section 4.2). In addition, we apply an extended version of an interval numbering scheme, a well-known technique in studies of XML databases, to graphs CI and PI for efficient detection of ancestor-descendant (or parent-child) relationships between two distinct resources (in Section 4.3).

This facilitates fast retrieval of superclasses (or subclasses) of given classes, as well as superproperties (or subproperties) of given properties.

4.2 Path expressions

Since the structure of RDF data is directed graph, most the queries for RDF data are generally queries to detect subgraphs matching a given graph and/or queries to detect a set of nodes which can be reached via given path expressions. These queries are represented in path expressions. Since the conventional RDF databases store RDF data based on statements, many join operations are needed to process path-based queries. For this reason, we can say that the processing of path-based queries efficiently leads directly to a decrease in the number of join operations. We can thus achieve the speed of query processing of RDF data. Therefore, our approach is a scheme for storage based on path expressions.

We store not the entire RDF graph based on path expressions, but only graph G to which path-based queries are frequently posed. Graphs CI and PI should be stored by a scheme that can detect ancestor-descendant (or parent-child) relationships between classes or properties, since they represent inheritance hierarchies. Because the structures of graphs DR and T are flat, directed graphs whose depth is one, there is no advantage in their path-based storage. The usefulness cannot be demonstrated without a graph whose depth is great because the path-based approach reduces the number of joins.

Because RDF graphs have a structure in which not only nodes but also arcs have labels, a path expression of the RDF graph consists of both of nodes and arcs alternately. However, the queries for RDF data do not usually use such path expressions, but path expressions consisting of arcs. Therefore, our scheme based on path expressions is one that stores *arc paths*, where only arcs are components, into a relational table. We begin with the definition of path expressions.

Definition 1 (Arc path) *Given a directed acyclic graph g, the node set is denoted by $V(g)$, and the arc set by $E(g)$. Let $(v_m, v_n) \in E(g)$ be an arc that connects two distinct nodes, $v_m \in V(g)$ and $v_n \in V(g)$. An arc path from the source v_0 to the destination v_k is defined as a finite sequence of arcs:*

$$(v_0, v_1), (v_1, v_2), \ldots, (v_{k-2}, v_{k-1}), (v_{k-1}, v_k).$$

The path expression of the arc path is defined as the series of labels:

$$l(v_0, v_1), l(v_1, v_2), \ldots, l(v_{k-2}, v_{k-1}), l(v_{k-1}, v_k),$$

where l is the label function that maps an arc to a label. □

For any directed acyclic graph, there exists at least one node whose in-degree is equal to zero. We call such nodes *roots*, and an arc path whose source node is a root is called *absolute arc path*.

Path expressions can be extracted from graph G by depth-first traversal on every node starting from the roots. For each step, the absolute arc-path expressions from all roots to the current node and the node itself are output and stored in relational tables *path* and *resource*, respectively.

Most path-based queries detect resources that are reached by a given path expression. In such queries, it is necessary to compare a character string, as a given path expression, which starts with a wild card, and stored path expressions. However, if a string that starts with a wild card is used as a query key, we cannot use indexing schemes in relational databases such as B-tree, and thus we must compare them in sequence. As a solution, we introduce reverse-path expressions. We store the reverse absolute arc-path expressions from the current node to all roots into a relational table. We first create a character string that ends with a wild card as a reverse path from a given query. We then compare the created string and stored reverse-path expressions. Because a query key is a string that ends with a wild card, we can use the indexing scheme in a relational database, and achieve the speed of path-based query processing.

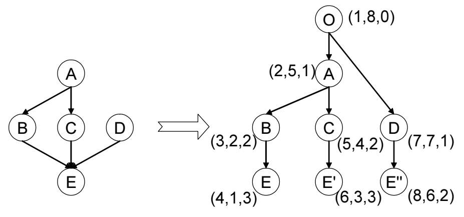

Figure 3: Interval numbering scheme for DAG.

4.3 Extended interval numbering scheme for DAGs

The interval numbering scheme (Li & Moon 2001) can detect ancestor-descendant (or parent-child) relationships between two nodes in a tree. Each node in a tree is given a pair of integers, preorder and postorder, and the relationships between two nodes can be detected by checking if the number of a node is subsumed by another node number. Because of its simplicity, it been typically used in literature related to XML databases.

Here, we use the interval numbering scheme to detect inheritance relationships between classes or properties. However, it is obvious that we cannot apply the scheme to graphs CI and PI in its original form, because classes or properties may have multiple inheritances, thus the graph structures are directed acyclic graphs.

We extend the interval numbering scheme to apply it to directed acyclic graphs. First, we place the virtual root as a parent of the roots (O in Figure 3) and connect it to each node whose in-degree is equal to zero. There are two reasons for this: graphs CI and PI may have several nodes of in-degree zero; and, a graph CI (or PI) may not be a connected graph; that is, the graph may be a forest consisting of several components. To produce consistent node numbers, we force the graph to be a rooted directed acyclic graph. Next, we expand the multiple paths to single paths (right side of Figure 3). In this example, node E appears three times because there are three paths from O to E. Note that we can still keep the identity of a node with an URI. Finally, we assign a node number (preorder, postorder, depth) to every node. Thus, some nodes that have multiple paths from the root may be assigned two or more node numbers. In the above example, E is assigned three node numbers. Algorithm 1 shows the algorithm.

The relationship between two nodes, v and u, can be verified by a subsumption: v is an ancestor of u iff $pre(v) < pre(u) \wedge post(u) < post(v)$ holds, where $pre(v)$ and $post(v)$ represent the preorder and postorder of v, respectively. We extend the original numbering scheme to adjust it to directed acyclic graphs.

Section 1 (Subsumption theorem for DAGs) *Suppose v and u are nodes, and $(pre(v_i), post(v_i), depth(v_i))$, $i = \{1, 2, \ldots, m\}$*

Algorithm 1: `AssignNumbers`

```
Data: A directed acyclic graph $G$
Result: $G'$ including node numbers
$V \leftarrow$ the set of nodes in $G$
$pre \leftarrow 1$
$post \leftarrow 1$
$depth \leftarrow 0$
foreach $v \in V$ do
    if the in-degree of $v = 0$ then
        reAssignNumbers($v$)
    end
end
return $G'$
function reAssignNumbers($v$) begin
    $pre \leftarrow pre + 1$
    $depth \leftarrow depth + 1$
    Append the number $pre$ to node $v$ in $G$.
    Append the number $depth$ to node $v$ in $G$.
    $U$ = a set of nodes adjoined from $v$.
    foreach $u \in U$ do
        reAssignNumbers($u$)
    end
    Append the number $post$ to node $v$ in $G$.
    $post \leftarrow post + 1$
    $depth \leftarrow depth - 1$
end
```

and $(pre(u_j), post(u_j), depth(u_j))$, $j = \{1, 2, \ldots, n\}$ *are the node numbers of* v *and* u*, respectively. Node* v *is an ancestor of* u *iff there exists at least the pair of* (i, j) *such that*

$$pre(v_i) < pre(u_j) \land post(u_j) < post(v_i)$$

holds. In addition, v *is a parent of* u *if* $depth(u_j) - depth(v_i) = 1$. □

Proof 1 (omitted)

Let us look at the example in Figure 3, which shows that A's node number is (2,5,1) and that E's node numbers are (4,1,3), (6,3,3), and (8,6,2). One of E's node numbers satisfies $2 < 4 \land 1 < 5$, and so we know that A is an ancestor of E. In fact, another E node number also satisfies the subsumption theorem. We know that D is a parent of E because $7 < 8 \land 6 < 7$ and $2 - 1 = 1$.

4.4 Proposed relational schema

Based on the subgraphs introduced in Section 4.1, we designed our relational schema for storing RDF data (Figure 4).

The relations *class* and *property* are for storing classes and properties in an RDF schema. In the tables, the attributes *pre*, *post*, and *depth* represent node numbers generated by the extended interval numbering scheme, and the values are computed from the graphs CI and PI. As mentioned in Section 4.3, for classes (or properties) that can be reached via two or more paths from the (virtual) root, several node numbers are assigned; therefore the numbers will be stored using multiple tuples. For *property* tables, the attributes *domain* and *range* represent the domain and range of a property, and the values are computed from the graph DR.

The RDF instance data that appears in graph G are represented by the three tables *resource*, *path*, and *triple*. Every node in graph G is stored in the *resource* table. At the same time, all absolute arc-path expressions for each node are also extracted and stored in

```
CREATE TABLE class ( className varchar,
                     pre       int,
                     post      int,
                     depth     int )
CREATE TABLE property ( propertyName varchar,
                        domain       varchar,
                        range        varchar,
                        pre          int,
                        post         int,
                        depth        int )
CREATE TABLE resource ( resourceName text,
                        ppathID      int ,
                        datatype     int )
CREATE TABLE triple ( subject   varchar,
                      predicate varchar,
                      object    text )
CREATE TABLE path ( pathID  int,
                    pathexp text )
CREATE TABLE type ( resourceName varchar,
                    className    varchar )
```

Figure 4: Proposed relational schema.

the *path* table. The attribute *pathID* is used to correlate the two tables. As mentioned in Section 4.2, a resource can be reached from two or more roots. (Recall that the nodes in graph G, whose in-degree is 0, are defined as the roots of G.) In such a case, the resource is stored in the relation using several tuples. The *triple* table is used to store all statements of the graph G.

Finally, graph T, which correlates the RDF schema with RDF instances, is represented by the *type* table.

Figure 5 shows storage example of the RDF data in Figure 2 [3].

4.5 Query Processing

Our RDF databases can provide efficient support for queries about RDF schemas and path-based queries as well as conventional statement-based queries. We demonstrate how path-based queries and queries on schema information are processed in the proposed relational schema.

Example 1 *Find the title of something painted by someone.*

```
SELECT r.resourceName
FROM   path AS p, resource AS r
WHERE  p.pathID = r.pathID
AND    p.pathexp = '#title<#paints'
```

Example 2 *Find the names of the classes that are* `http://www.w3.org/2000/01/rdf-schema#Resource`*'s direct superclass.*

```
SELECT c1.className
FROM   class AS c, class AS c1
WHERE  c.pre < c1.pre
AND    c.post > c1.post
AND    c.depth = c1.depth - 1
AND    c.className =
'http://www.w3.org/2000/01/rdf-schema#Resource'
```

5 Performance Evaluation

This section evaluates the proposed scheme's performance in a series of experiments. We compared the processing time between our approach and Jena2 (Wilkinson et al. 2003), which is a relational RDF

[3] We omit the information of namespace and meta schema from Figure 5.

class

className	*pre*	*post*	*depth*
'Literal'	2	1	1
'Artist'	3	4	1
'Sculptor'	4	2	2
'Painter'	5	3	2
'Artifact'	6	7	1
'Sculpture'	7	5	2
'Painting'	8	6	2
'CommercialGoods'	9	9	1
'Painting'	10	8	2

property

propertyName	*domain*	*range*	*pre*	*post*	*depth*
'first'	'Artist'	'Literal'	2	1	1
'last'	'Painter'	'Literal'	3	2	1
'creates'	'Artist'	'Artifact'	4	5	1
'sculpts'	'Sculptor'	'Sculpture'	5	3	2
'paints'	'Painter'	'Painting'	6	4	2
'title'	'Artifact'	'Literal'	7	6	1

type

resourceName	*className*
'r1'	'Painter'
'r2'	'Painting'
'r3'	'Painting'
'r4'	'Sculptor'
'r5'	'Sculpture'

resource

resourceName	*pathID*	*datatype*
'r1'	1	0
'r2'	4	0
'r3'	4	0
'r4'	1	0
'r5'	6	0
'Picasso'	2	1
'Pablo'	3	1
'August'	2	1
'Rodin'	3	1
'Guernica ..'	5	1
'Les demoi..'	5	1
'The Thinker'	7	1

path

pathID	*pathexp*
1	''
2	'#first'
3	'#last'
4	'#paints'
5	'#title<#paints'
6	'#sculpts'
7	'#title<#sculpts'

triple

subject	*predicate*	*object*
'r1'	'first'	'Picasso'
'r1'	'last'	'Pablo'
'r1'	'paints'	'r2'
'r1'	'paints'	'r3'
'r2'	'title'	'Guernica ..'
'r3'	'title'	'Les dem..'
'r4'	'first'	'August'
'r4'	'last'	'Rodin'
'r4'	'sculpts'	'r5'
'r5'	'title'	'The Thinker'

Figure 5: A storage example of RDF data.

database based on the flat approach. This selection is based on the fact that the flat and hash approaches have similar characteristics due to their underlying storage structure, and Jena2 is one of the major implementations of the flat approach. In other words, these experiments we attempt to observe any behavioral difference of our approach against the flat and hash approaches. In addition, we do not take the schema approach, because our experimental data, Gene Ontology, do not contain RDF Schema data, which means that the schema approach cannot deal with the data.

Although we should evaluate the performance of schema-based queries and path-based queries if possible, as far as we know, there exist no RDF data with schema information whose size is large enough to be used in our experiments on the Web. For this reason, we cannot evaluate the performance of schema-based queries. Instead, we show whether schema-based queries can be processed in the approaches.

We used a PC with an Athlon 1.4 GHz CPU and 1 GB of memory running Gentoo Linux 1.4, and used PostgreSQL 7.4.3 as a relational database.

5.1 Schema-based Queries

In this section, we investigate whether schema-based queries can be processed in each approach.

Basic schema queries

1. Find immediate children (or parents) of a given class (or property).
2. Find descendants (or ancestors) of a give class (or property).
3. Find descendants (or ancestors) of a give class (or property) whose distance is n generations.
4. Find inheritance relationships between given two classes (or properties).
5. Find classes as a domain (or range) of a given property.
6. Find properties whose domain (or range) is a given class.

Querying the meta-schema

7. Find all resources, that is, instances of "*rdfs:Resource*."
8. Find all classes (or properties), that is, instances of "rdfs:Class (rdf:Property)."
9. Find all literals.

Querying type information

10. Find a set of instances of a given class.
11. Find a set of instances of a given class and its descendant classes.
12. Find a set of statements using a given property.
13. Find a set of statements using a given property and its descendant properties.

For each approach, when the above queries are processed, there are two cases: the answer is obtained by a single access to data storage, or multiple accesses. Thus, we focus not only on whether the queries can be processed, but also on whether repeat accesses are needed. Table 2 shows the ability of each approach for schema-based queries.

From Table 2, we can see that our approach is more efficient than other approaches for queries concerning inheritance relationships. because we use the interval numbering scheme. However, in meta-schema queries, if the RDF graph includes many multiple paths, the redundancy is increased. In the feature work, we have to evaluate the performance of schema-based queries using large RDF schema information.

5.2 Path-based Queries

In this section, we evaluate the performance of path-based queries through a series of experiments.

5.2.1 Datasets

A number of RDF data, such as ODP (Netscape n.d.), Wordnet (Miller et al. 1993), and Gene Ontology (GENE ONTOLOGY CONSORTIUM n.d.), are available on the Net. From those candidates, we made our selection taking the following points into account: 1) sufficient size to see scalability; 2) the G

Table 2: The ability of schema-based queries

	our approach	flat/hash	schema
#1.	Single	Single	Single
#2.	Single	Multiple (by navigation)	Multiple (by navigation)
#3.	Single	Single (by n joins)	Single (by n joins)
#4.	Single	Multiple (by navigation)	Multiple (by navigation)
#5.	Single	Single	Single
#6.	Single	Single	Single
#7.	Single (including redundancy)	Impossible	Single
#8.	Single (including redundancy)	Single	Single
#9.	Single (including redundancy)	Impossible	Single
#10.	Single	Single	Multiple (by difference)
#11.	Multiple (by union)	Multiple (by navigation)	Single
#12.	Single	Single	Multiple (by difference)
#13.	Multiple (by union)	Multiple (by navigation)	Single

Table 3: The statistics of Gene Ontology RDF data

Data size (KB)	18,532
# of statement	258,288
# of statement in graph G	240,687
# of resources in G	147,140
# of kinds of predicates in G	11
Max # of out-degree in G	1,467
Average # of out-degree in G	3.733
Max # of in-degree in G	17,601
Average # of in-degree in G	1.771
# of path in G	10,000
Max length of path in G	17

graph of the data does not contain any cycles; and 3) the G graph of the data contain long absolute path expressions, enabling us to test the ability of processing path-based queries. Unfortunately, ODP is not appropriate, because it does not conform to the latest RDF specifications; therefore, it cannot be processed with current RDF databases/processors. Wordnet is also inappropriate because its path expressions are too short. Finally, we decided to use the Gene Ontology as the experimental dataset. In fact, the size of the *resource* table becomes much larger than the size of the resources due to the existence of multiple paths. However, if we can prove that our approach is more effective than other related approaches, we can say that it is also effective even for other types of RDF data.

There are two types of RDF data in the Gene Ontology; one is coupled with gene associations and the other is not. We use the latter one, whose size is 19 MB. Table 3 shows the dataset statistics.

We prepared different path expressions whose length varied from 1 to 17; it is rare that long path queries of clear length such as 17 are used in reality. However, we should consider recursive queries and/or regular expressions, such as "//" and "*" of XPath. In these situations, we cannot prespecify the length of path expressions in the answer set. As a result, we think that using long path expressions is a realistic assumption.

5.2.2 Experimental Results

Table 4 shows the number of rows in each table. As mentioned above, since the Gene Ontology RDF data do not have RDF schema, *class* and *property* tables store no data. From this table, we can see that the size of the *resource* table is extremely large because as can be expected from Table 3, it includes a lot of multiple paths.

Figure 6 depicts the processing times of path-based queries using our approach and Jena2. The x-axis shows the length of path expressions, and the y-axis shows the processing times (ms). Note that the processing of queries, whose lengths are more than 11, are blank for Jena2, because Jena2 could not accomplish those queries due to errors. Interestingly, the times of Jena2 increase as the path lengths grow, while the times of our approach decrease. For path expressions of a length less than 3, Jena2 outperforms the proposed scheme. However, our approach becomes faster than Jena2 for path queries longer than 4. The reason for this is that Jena2 requires more join operations as path expressions lengthen. On the other hand, the number of join operations is fixed for our approach. Furthermore, the size of answers decreases. As a consequence, our approach significantly outperforms Jena2 for queries with long path expressions.

Table 4: Table sizes of RDF data.

class	0
property	0
type	17,601
resource	38,255,366
path	10,000
triple	240,687

6 Conclusions

In this paper, we proposed a path-based relational RDF database. In our scheme, because RDF schema information and RDF instance data are stored into distinct relational tables separatly, we can handle schemaless RDF data, and in the case of RDF data including schema information, we can process schema-based queries using the interval numbering scheme. Additionally, for path-based queries, the performance of the conventional RDF databases is poor, because the RDF databases need a large number of join operations according to path steps. To reduce the number of join operations, we stored RDF data based on path expressions in our database, and as a result, we achieved high performance on query retrieval based on path expressions.

In the future, we plan to evaluate the performance of our scheme using a larger amount of RDF data. In addition, we will investigate query-processing techniques including query language, query transformation, and query optimization for RDF data.

Acknowledgments

This work is partially supported by the Ministry of Education, Culture, Sports, Science and Technology

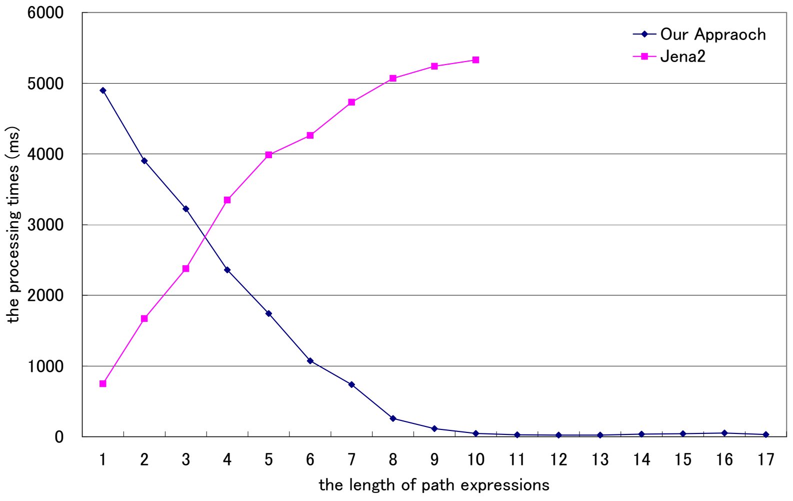

Figure 6: The processing times our approach and Jena2.

(MEXT) Grant-in-Aid for Scientific Research for Priority Areas No. 15017243, and by Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research No. 15200010 and 15700097.

References

Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D. & Tolle, K. (2000), The RDFSuite: Managing Voluminous RDF Description Bases, *in* 'Proceedings of the Second International Workshop on the Semantic Web (SemWeb'2001)'.

Beckett, D. J. (2001), The design and implementation of the redland RDF application framework, *in* 'Proceedings of the Tenth International World Wide Web Conference (WWW10)', pp. 449–456. **URL:** *citeseer.nj.nec.com/beckett01design.html*

Broekstra, J., Kampman, A. & van Harmelen, F. (2002), Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema, *in* I. Horrocks & J. Hendler, eds, 'Proceedings of the First Internation Semantic Web Conference', number 2342 *in* 'Lecture Notes in Computer Science', Springer Verlag, pp. 54–68.

Dan Brickley and Libby Miller (2003), 'FOAF Vocabulary Specification', `http://xmlns.com/foaf/0.1/`. RDFWeb Namespace Document 16 August 2003.

Dublin Core Metadata Element Set, Version 1.1 (2003), `http://dublincore.org/documents/dces/`.

GENE ONTOLOGY CONSORTIUM (n.d.), 'Gene Ontology', `http://www.geneontology.org/`.

Li, Q. & Moon, B. (2001), Indexing and Querying XML Data for Regular Path Expressions, *in* 'Proceedings of the 27th International Conference on Very Large Data Bases', Morgan Kaufmann Publishers Inc., pp. 361–370.

Matono, A., Amagasa, T., Yoshikawa, M. & Uemura, S. (2003), An Indexing Scheme for RDF and RDF Schema based on Suffix Arrays, *in* 'Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Berlin, Germany, September 7-8, 2003', pp. 151–168.

Members of the TopicMaps.Org Authoring Group (2001), 'Xml topic maps (xtm) 1.0', `http://www.topicmaps.org/xtm/1.0/xtm1-20010806.html`.

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D. & Miller, K. (1993), 'Introduction to WordNet: An On-Line Lexical Database', `http://www.cogsci.princeton.edu/~wn/`.

Miller, L. (n.d.), 'Inkling: RDF query using SquishQL', `http://swordfish.rdfweb.org/rdfquery`.

Netscape (n.d.), 'Open Directory Project', `http://dmoz.org/`.

Reggiori, A. (n.d.), 'RDFStore: Perl API for RDF Storage', `http://rdfstore.sourceforge.net/`.

RSS-DEV Working Group (2000), 'RDF Site Summary (RSS) 1.0', `http://web.resource.org/rss/1.0/`.

R.V. Guha (n.d.), 'rdfDB : An RDF Database', `http://guha.com/rdfdb/`.

Wilkinson, K., Sayers, C., Kuno, H. A. & Reynolds, D. (2003), Efficient RDF Storage and Retrieval in Jena2, *in* 'Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Berlin, Germany, September 7-8, 2003', pp. 131–150.

World Wide Web Consortium (2004*a*), 'RDF Vocabulary Description Language 1.0: RDF Schema', `http://www.w3.org/TR/rdf-schema/`. W3C Recommendation 10 February 2004.

World Wide Web Consortium (2004*b*), 'Resource Description Framework (RDF)', `http://www.w3.org/RDF/`. W3C Recommendation 10 February 2004.