ELSEVIER

Information and Software Technology 45 (2003) 241–251

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Efficient processing of regular path joins using PID

Jongik Kim*, Hyoung-Joo Kim

School of Computer Science and Engineering, Seoul National University, San 56-1, Shillim-dong, Gwanak-gu, Seoul 151-742, South Korea

Received 3 May 2002; revised 7 October 2002; accepted 27 November 2002

Abstract

XML is data that has no fixed structure. So it is hard to design a schema for storing and querying an XML data. Instead of a fixed schema, graph-based data models are widely adopted for querying XML. Queries on XML are based on paths in a data graph.

A meaningful query usually has several paths in it, but much of recent research is more concerned with optimizing a single path in a query. In this paper, we present an efficient technique for processing multiple path expressions in a query. We implemented our technique and present preliminary performance results.

© 2003 Elsevier Science B.V. All rights reserved.

Keywords: XML query; Multiple regular paths; Path identifier

1. Introduction

As XML [6] has become a new standard for data representation and exchange on the Web, storing and querying XML data is an active research area in the database community. Unlike HTML, an XML document has the structure information in itself (*self-describing*). Though one can retrieve the structure information from an XML document, it is hard to design a fixed schema to store and query the document for the irregularity of the structure. These features show that XML is an instance of semi-structured data. The graph based data model is widely used to represent the structure of XML and semi-structured data. The graph based data model is classified into the edge-labeled graph model and the node-labeled graph model. These two models have the same semantics. Though almost XML documents are represented in node-labeled graph, we use edge-labeled graph as the data model for descriptive purpose. Conversion one into the other is relatively simple. We describe this conversion in Section 3. Fig. 1 shows movie XML data represented by a graph model.

Query languages for XML [10,12] use graph models to query XML data, and queries on XML are expressed as paths in a data graph. Because of the irregular structure of XML, it is hard to know whole structure of an XML document. To query an XML data with the knowledge of partial structure of it, query languages support regular expressions on the paths. Regular expressions are also useful for representing arbitrary paths (a path can be infinite due to a cycle) in a data graph. The following example is a query that finds all movies and dramas in which Brad Pitt acted.

```
⟨result⟩
{
   for $m in //(movie|drama)
   where $m//actor//name = "Brad Pitt"
   return $m
}
⟨/result⟩
```

We use the query language, XQuery [10], for the example (the query language used in this paper is presented in Section 3). This query contains two regular path expressions; //(movie|drama) and //actor//name. The first path indicates the result nodes and the second path plays a role of a constraint for the first path. Unlike queries on value-based traditional data, queries on XML data frequently require intermediate nodes as its results. When a query requires intermediate nodes as its results, the query tends to use another path as a constraint, like the one above.

A query which has multiple paths can be processed as following strategy. Starting at the root of the data graph,

* Corresponding author. Tel.: +82-2-888-7527; fax: +82-2-882-0269.
E-mail addresses: jikim@oopsla.snu.ac.kr (J. Kim), hjk@oopsla.snu.ac.kr (H.-J. Kim).

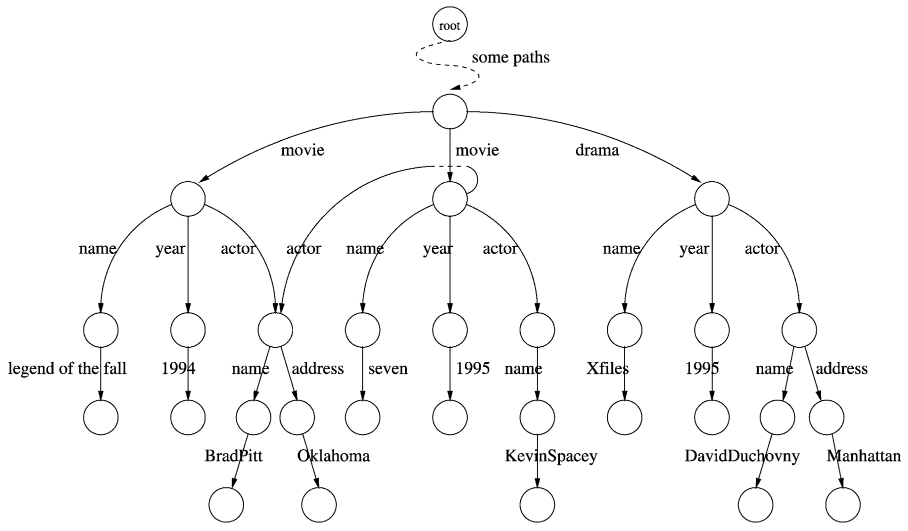

Fig. 1. Movie database.

the query searches for paths satisfying the regular expression //(movie|drama), and from the retrieved nodes, searches paths which satisfy the other regular expression //actor//name. This is because not every node that satisfies the regular path //(movie|drama) has the paths satisfying //actor//name as its descendant, and that is the reason the path //actor//name can be a constraint for the path //(movie|drama). A naive evaluation that traverses all possible paths in the data graph is obviously inefficient.

Optimizing regular path expressions is the main research issue in query processing for XML and semi-structured data. However, most of the work is more concerned with processing a single regular path expression and is inadequate to a query that has multiple path expressions.

Current state of the art suggests three plans to process a query that has multiple regular path expressions.

Plan 1: *using an index for a single path.* Find the extents that satisfy the regular path //(movie|drama) using or an index for a single paths (e.g. Dataguide, 1-index, Index Fabric). From each node in the retrieved extents, search the data graph for the regular path expression //actor/-name (= "Brad Pitt"). To evaluate the second path expression, this plan traverses the data graph, so it does not work efficiently.

Plan 2: *using the 2-index.* Using the 2-index, we can find the pairs (a, b)s that satisfy the regular path expression, //(movie|drama). Again, we can find the pairs (c, d)s that satisfy //actor/name using the 2-index. Project (a, b)s with the condition, a = *root*, and project (c, d)s with the condition, d = "Brad Pitt". And then, we can obtain (a, m, b)s by (a,b)s $\bowtie_{b=c}$ (c, d)s. Finally the m's are the result. This plan is simple and easy to use, but this plan is not preferable because the size of the 2-index can grow quadratic to the size of the data graph in the worst case, and always grow much larger than 1-index. Actually, for the same data graph, the 2-index has the same structure of the 1-index as its subgraph.

Plan 3: *using the T-index.* The T-index is the alternative of the 2-index for the index size of the 2-index. Using the T-index, we can get the result of a query directly, but the T-index cannot be used in the case where appropriate templates have not been built.

We propose a new query processing scheme that can handle multiple regular path expressions in a query efficiently. The proposed scheme uses an index structure for a single regular path, and does not traverse the data graph at all. The following roughly describes the strategy of our techniques.

1. Find the union of extents e_1 satisfying the regular path //(movie|drama) using any index structure for a single path.
2. Find the union of extents e_2 satisfying the regular path //(movie|drama)//actor.name (= "Brad Pitt") using the index structure for a single path.
3. If there is a path between an object in e_1 and an object in e_2, add the pair of these two objects into the result set.

To figure out whether two given nodes lie on the same path in data graph, we define the path identifier (PID) and assign the PID to each node in the data graph. In addition to path identification, we can use the PID for supporting order-based query. We discuss the PID in Section 4.

The followings describes main contribution of this paper:

- The PID described in Section 4 can determine the path relationship (ancestor–descendant relationship) between two nodes in a data graph, where the data graph can be not only a tree but also an arbitrary directed graph which can contain cycles.

- The proposed PID can be used for order-based queries.
- We proposed an efficient algorithm which can evaluate a query with multiple path expressions in it. Our algorithm uses an index graph for a single path with the PID.

The paper is organized as follows. Section 2 lists related work. In Section 3, we describe data model and query language used in this paper and review an index structure for a single regular path (Dataguide) which is used in this paper for the descriptive purpose. In Section 4, we define the PID and present the algorithm for PID construction. A processing algorithm for multiple path expressions is presented in Section 5. Section 6 shows preliminary results and we conclude the paper and give an outlook to future work in Section 7.

2. Related work

The similarities and differences between XML data and semi-structured data are described in Ref. [23]. In spite of some differences, XML data can be viewed as an instance of semi-structured data.

The most significant difference between XML data and semi-structured data is that XML data has the order information in it. The PID proposed in this paper can be used for a query which uses order information in XML data.

Query languages for XML can be found in Refs. [10, 12]. Those languages use regular path expressions to express a query like the query languages used in semi-structured data [3,7]. In particular, the XQuery [10] uses XPath [25] as its path expression. Our techniques can be applied to all of the query languages which use the regular path expressions.

The structural summary (or data guide) [2] plays the role of a schema of semistructured data. Refs. [15,21,24] present techniques to extract the schema graph from the data source. Specially Ref. [15] describes the Dataguide, a structural summary, as an index of paths in a data graph.

Query processing for semi-structured data is focused on optimizing the regular path expressions. Optimizing techniques for the regular path expressions are presented in Refs. [1,8,13,14,18,19,22]. In Ref. [13], query pruning and query rewriting techniques using *graph schemas* [7] are introduced and it is shown that graph schemas can be used as an index for paths. The 1-index, 2-index, and T-index are proposed in Ref. [19]. This index family uses language equivalent relation for paths to construct indexes. In Ref. [4], the Index Fabric is presented which uses the patricia trie for indexing paths of XML and semi-structured data.

The 1-index is somewhat like the Dataguide without the nondeterministic feature. Any two out-going edges in the Dataguide which point to different nodes have different labels. The size of the Dataguide can grow exponential order of the size of a data graph due to the deterministic feature of it. But the index graph of the 1-index is nondeterministic. That is, more than two edges which point to different nodes can have the same label. So the size of the 1-index grows linear to a data graph. Because of the nondeterministic feature, the 1-index cannot play a role of the data guide.

However, these techniques are more concerned with a single path expression, and cannot be used for optimization of multiple regular path expressions in a query.

To process multiple path expressions efficiently, the 2-index and T-index are proposed, but the 2-index can grow quadratic to the size of a data graph in the worst case and the T-index can only be used in the case where appropriate templates have been already built. Fernandez and Suciu [13] presented a query pruning technique for multiple regular paths but they only focused on query pruning so the inefficiency still remains to traverse a data graph.

The numbering scheme proposed in Ref. [17] is a method for detecting the ancestor–descendent relationship between two nodes in a tree. They assign numbers at each node in the data tree and store them in relational index for processing single regular path expression. The numbering scheme, PID, proposed in this paper can detect ancestor–descendent relationship not only in a tree but also in an arbitrary (directed) graph. Moreover, we can use the PID for supporting order-based queries. An apparent difference between this paper and Ref. [17] is that we use PID to process multiple regular path expressions using an existing index for a single path.

3. Background

3.1. Data model

XML can be modeled by a graph. Let O be an infinite set of object identifiers and C be an infinite set of constants. O and C are disjoint sets. We define our data model as follows.

Definition 3.1. Data graph $\mathrm{DB} = (V, E, R)$ is a rooted graph such that $V \subseteq O$ is a set of nodes, $E \subseteq V \times C \times V$ is a set of directed edges and $R \in V$ is a root node.

Each element in XML can be mapped to V. The relationships, element to element, element to sub-element, and element to attribute can be mapped to E. References between elements can also be mapped to E.

There are many sorts of nodes in XML. For example, there can be an element node, an attribute node, a text node, and so on. But we do not distinguish the type of a node, because our technique is neutral to node types.

In this paper, we use the edge-labeled graph, such that $E \subseteq V \times C \times V$, as the data model, because it is convenient for describing an index graph such as the Dataguide and the 1-index. It is easy to convert an edge-labeled graph into a node-labeled graph. Without concerning references, the conversion can be done by labeling a node with the label of its incoming edge. If there exist some references, then the incoming edges of a node can have different labels. In such a case, we can redirect a reference edge by inserting a node. For example, a node labeled by A has a reference labeled by B. The conversion can be done as follows. First we insert a node labeled by B, and then make the reference edge which point to the node A to point the node B. Finally we make an edge between N and A.

In XML, a reference between nodes can be represent as an attribute. In the edge labeled graph model, we can omit such an attribute node in a data graph. When we convert an edge labeled graph to a node labeled graph, a node inserted for a reference is actually an attribute node which represents reference relationship.

3.2. Query language

Queries for XML are based on paths in a data graph, which are expressed using regular expressions. To describe our optimization technique, we define a simple query language used in this paper as follows.

Definition 3.2. Query language

Query :: = **bind** BindList **return** VarList
Bind :: = *var* **in** Path Predicate
BindList :: = Bind|Bind, BindList
Predicate :: = ∅| [*predicate*]
VarList :: = *var*|*var*, VarList
Path :: = /|/RPath|//RPath|*var*/RPath
RPath :: = *label*|RPath/RPath|RPath//RPath|(RPath|R-Path)|RPath*

All variables denote nodes in a data graph. We can express a path expression with RPath. RPath is recursively constructed by the concatenation (/) and the alternative (|) of two RPaths, and Kleene star (*) of an RPath. Actually, it is a regular language with labels in a data graph as its alphabet. The target application of our technique is a path join query (Definition 3.3), and a path join query can appear in many different forms in the XQuery [10]. We make above query language by simplifying and modifying the XQuery to represent a join query in a single form. The following is the definition of a path join query.

Definition 3.3. With two variable bindings, 'v_1 *in* P_1, v_2 *in* v_1/P_2', we define the *regular path join* as the procedure of finding the pairs, (v_1, v_2)s, such that there exsits a path from the node, v_1, to the node, v_2, which satisfies the regular path expression, P_1, and there exists a path which satisfies the regular path expression, P_2, as a descendant of v_2. We define the *n-way path join* as the query in the form of '*bind* v_1 *in* P_1, v_2 *in* $v_1/P_2,\ldots,v_n$ *in* v_{n-1}/P_n *return…*'.

3.3. Dataguide as an index for a regular path

We develop the PID and the join algorithm to process a path join query. Our algorithm uses an index structure for processing a single path. From the results of processing each single path, our algorithm generates final results using PID. We did not develop an index for a single path. Instead, we use an external index structure for a single path. By using an external index structure, our algorithm can be applied to many systems which use different index structures, and if a more efficient index structure is proposed, our algorithm can cooperate with that index.

We use *dataguide* [15] as our index for a single path for convenience, and with the term, index graph, we assume the dataguide hereafter. However, we present a join algorithm which is independent of a specific index graph, and any index structure for a single path can works with our algorithm without performance degradation.

Creating a dataguide over a source database is equivalent to the conversion of a non-deterministic finite automation (NFA) to a deterministic finite automation (DFA) [15,16,21]. Fig. 2 shows a data graph and its corresponding dataguide. Each node in a dataguide has an

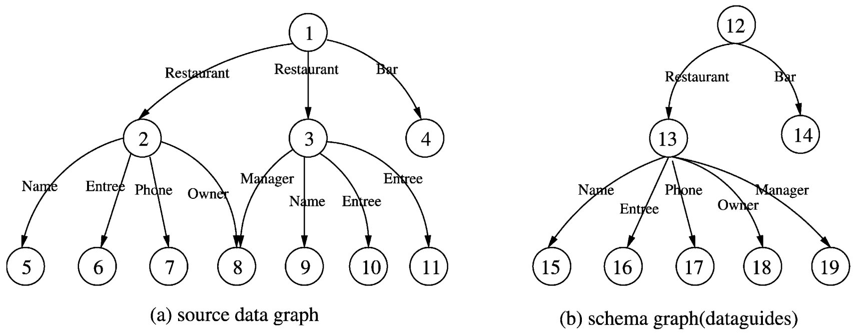

Fig. 2. Index for a single path (the Dataguide).

extent for the corresponding nodes in the data graph. For example, the extent in the node 16 is the set of {6, 10, 11}, where each element can be reached by the path Restaurant. Entree in Fig. 2.

4. Path identifier

PID is information stored in each node of the data graph to identify whether two given nodes in the data graph lie on the same path or not. In Section 4.1, we describe PID and present a PID construction algorithm. We describe the incremental maintenance of PID in Section 4.2.

4.1. Definition and construction

We give several simple definitions useful for describing the PID.

Definition 4.1. *Range* consists of two integers. The first integer in a range is called the *start point* and the other integer the *end point*. The relationship such that 'start point $\leq$ end point' must be satisfied in a range.

Definition 4.2. With the two ranges $r_1 = (s_1, e_1)$ and $r_2 = (s_2, e_2)$,

- r_1 and r_2 are *continuous* if $s_2 - e_1 = 1$.
- r_1 and r_2 are *overlapped* if $s_1 \leq s_2$ and $e_1 \geq s_2$ and $e_2 \geq e_1$.
- r_1 *includes* r_2 (or r_1 and r_2 are *inclusive*) if $s_1 \leq s_2$ and $e_1 \geq e_2$.

For example, (2,5) and (6,8) are continuous ranges, (2,6) and (4,7) are overlapped ranges, and the range (1,5) includes the range (2,4).

Definition 4.3. With the two sets, R_1 and R_2 whose elements are ranges, if every element in R_2 can find an element in R_1 which includes that element, then R_2 is a *path subset* of R_1, and likewise R_1 is a *path superset* of R_2. We call such a relationship a *path relationship*.

For example, $R_2 = \{(2,4),(7,9)\}$ is a path subset of $R_1 = \{(1,4),(7,9)\}$, because the range (1,4) in R_1 includes the range (2,4) in R_2 and the range (7,9) in R_1 includes the range (7,9) in R_2.

Definition 4.4. PID is a set of range, which satisfies following PID constraint.

PID constraint. The two PIDs of any two given nodes have the path relationship if and only if the two nodes have the 'ancestor–descendant' relationship such that the PID of the ancestor node is a path superset of the PID of the descendant node.

Using the PID defined above, we can see that if the PIDs of any two given nodes have the path relationship, then the two nodes lie on the same path.

Algorithm 1. *PID construction*

Algorithm 1 PID Construction
```
// input: target, the root oid of a source data graph
// effect: construct path id at each node in source data graph
// initially, PID at each node in data graph is empty

sf() : skolem function

PIDConst(target)
{
  if(target.PID is not empty)
    return target.PID

  startpoint = endpoint = sf()
  insert (startpoint, endpoint) into target.PID

  if(target is non-leaf node){
    foreach(child node c of target)
      target.PID = target.PID ∪ PIDConst(c)
    Merge elements in the target.PID
  }
  return target.PID
}
```

Algorithm 1 is a PID construction algorithm. This algorithm constructs PID in each node of the data graph, which satisfy the PID constraint. The skolem function f used in this algorithm is the function such that $f_{n+1} = f_n + 1$ and $f_0 = 0$. We assign a range (such that the start point equals to the end point) to each node using the *skolem function* by traversing the data graph in depth first manner. So each node has a PID which contains only one element created by the skolem function. We call this PID in each node as *initial* PID. A PID of a leaf node is not change, that is, a PID of a leaf node has only one element which is created by the skolem function. A PID of a non-leaf node is created by the following strategy. First, unite the PID of each child node of a non-leaf node. And then unite the initial PID of that non-leaf node and union of the PID of each child node. Finally merge elements which are continuous or overlapped or inclusive.

With the PID construction algorithm, we only consider the case where a data graph is a DAG. If a data graph has cycles, we make the condensation [5] of the data graph, and we construct the PID of each node in the condensation. And then every node in a strongly connected component has the same PID. It is easy to see that this technique constructs the PID which satisfies the PID constraints in any cyclic graph.

Fig. 3 shows the PIDs in a data graph when a data graph is a tree, a DAG, and a cyclic graph. In case that a data graph has the cycle as (c) of Fig. 3, every node in a strongly connected component has the same PID. In Fig. 3(a), the union of the PID of n_4 and the PID of n_5

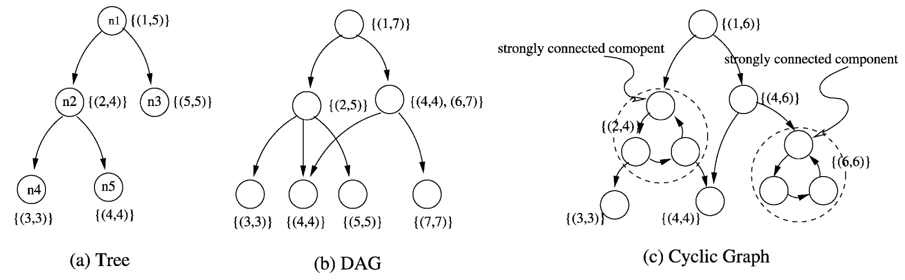

Fig. 3. PID in data graphs.

is{(3,3), (4,4)}, and the two ranges (3,3) and (4,4) are continuous, so we can merge the two ranges into (3,4), and then we unite{(3,4)} and{(2,2)} which is initial PID of n_2, and merge the set {(2,2), (3,4)}, into{(2,4)} as the same way, which is the PID of n_2. A PID of an non-leaf node can only be created by the PIDs of its children and its own PID, so this allows Algorithm 1 to satisfy PID constraint.

If a data graph has heavy reference structure, PIDs of some node can grow large. For a data which has heavy reference structure, we assume that references tend to either make many cycles or be destined for some specific nodes. In the former case, the condensation of the data graph tends to have light reference structure when there are many cycles (strongly connected components) in the data graph. So the PID in each node of the data graph can be created with a few elements. In the latter case, the size of the PID of a node is not affected by its descendant because most of references in the descendant nodes tend to destined for the same nodes. So the size of the PID can be bounded on a constant. Much work for XML data assumes that XML data is a tree structure. But such an assumption is not true. Many XML data cannot be a tree because of its reference structures. Though XML data can be an arbitrary directed graph, it still has tree-like structure in many cases. So our PID keeps reasonable size. In particular, every PID in a data graph has one element when the data graph is a tree.

To construct the PID in each node of the data graph, Algorithm 1 visits each node once by traversing the data graph in depth first search. Let the maximum number of children of a node be m and the size of data graph be n. To merge elements in the PID construction strategy, each intermediate node has the cost of m^2 under the assumption that the size of the PID of a node is a constant. So the construction algorithm has the time complexity, $O(m^2n)$. Because m is much smaller than n, we consider it is a constant, so the time complexity of PID construction algorithm is linear to the size of the data graph.

A PID in each object (node) of the data graph can be stored in the extent of an index graph with an object identifier. If the size of a PID is too big to be stored in an extent, the PID can be retrieved from the data graph dynamically when the PID is required.

4.2. Order support

Order information is one of the unique features of XML data. XQuery has **before**($\ll$) and **after**($\gg$) clause to represent order-aware queries. To use before and after clause, XQuery defines *document order* [10], a total ordering among all the nodes in an XML document. Briefly speaking, given two distinct nodes, A and B in an XML document, A is before B if and only if A occurs before B in the textual representation of the XML document. In graph representation of XML, we can get the same ordering as the document order by traversing the graph in depth first manner. We can use the PID for supporting order-aware queries, cause the initial PID of a node is assigned by depth first traversal of the data graph. When we make initial PIDs on a data graph, we should not traverse reference edges, because references can make invalid document order. Fig. 4 shows valid and invalid initial PID. In Fig. 4, the dashed edge is a reference edge.

Initial PID can be changed by merging it with the ranges from the PID of children. Recall that the start point equals to the end point of the range in an initial PID. We call the value (either start point or end point) of the range in an initial PID, the *anchor value* of the node. To support order-aware queries, we can keep the anchor value of a node in its PID. So A PID consists of the pair {anchor, a set of range}. By comparing anchor values, we can compare the order of two nodes. If two nodes are in the same connected component, the two nodes have the same anchor value and it is impossible to compare the order between the two nodes. We

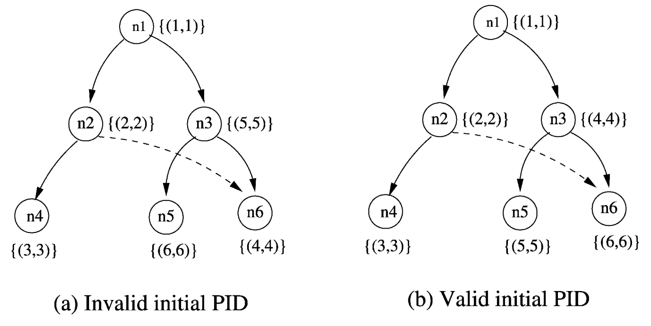

Fig. 4. Initial PID assignment.

must assign additional value to each node in the same strongly connected component. We can assign this additional value by traversing each strongly connected component by depth first manner. Therefore, the final format of a PID is the triple {anchor, ssc, a set of range}. The ssc field in a PID is meaningless in a node which does not participate in a strongly connected component.

4.3. Incremental maintenance

It is not necessary to update the PID in the data graph when a node is deleted from the data graph, but when a new node is inserted to the data graph, we should assign a new PID to the node and sometimes PIDs in other nodes of the data graph need to be updated. A naive updating scheme is to reconstruct the PIDs in the data graph using Algorithm 1, but this is obviously very inefficient. In this section, we present a simple scheme to maintain the PID incrementally.

For a newly inserted node, a room for the PID is required. We can get the room for the PID by modifying the skolem function *f* in Algorithm 1 such that $f_{n+1} = f_n + m$ and $f_0 = 0$. Now, we can get the room for the PID of size m between two incident nodes in document order. The definition of *continuous* in Section 4.1 should be changed such that if $s_2 - e_1 \leq m$, then the two nodes are continuous.

Insertion takes place in the condensation of a data graph, and we assume that a newly inserted node is a leaf node of the data graph. So the inserted node has the PID which has one range with the same value for the start point and the end point. We can assign a PID to the newly inserted node as following strategy.

1. Find the node, N_p, which is the very previous to the inserted node in document order. Let the anchor value of N_p be A_p.
2. Find the node, N_n, which is the very next to the inserted node in document order. Let the anchor value of N_n be A_n.
3. If the difference between A_n and A_p equals to 1, then reconstruct whole the PID in the data graph.

Else, insert a range $(A_p + 1, A_p + 1)$, to the PID of the newly inserted node.

Note that we can always find the node N_p, unless the inserted node is the only node of the data graph. When a node inserted as the first child of a node or the last child of a node, ancestors of the newly inserted node are updated to preserve the path relationship.

When a sequence of nodes is inserted, we can insert the nodes one by one using above procedure. But if a sequence of nodes is about to be inserted at the position of the first or the last child of a node, it is inefficient to insert the nodes one by one, because the PIDs of ancestors can be updated repeatedly. In such a case, we can process the insertion as follows. Actually, the sequence of nodes being inserted forms a graph structure. First, we make the PID in each node of the graph to be inserted. And then add up $(A_p + 1)$ to every value in all PIDs of the graph to be inserted. Finally, insert the graph to the target node of the data graph. Now, update for ancestors occurs only once. Of course, the number of nodes in the sequence is less than the difference between A_n and A_p. Otherwise, whole PIDs in the data graph must be reconstructed. We omit the anchor value and ssc value in a PID in the further discussion, because it is not used in regular path join procedure.

5. Regular path join

As illustrated in Section 3.2, we define a *regular path join query* used in this paper as follows.

bind v_1 **in** P_1, v_2 **in** $v_1/P_2,\ldots,v_n$ **in** v_{n-1}/P_n **return** $x_1,\ldots,x_k$ $(x_i \in \{v_1,\ldots,v_n\})$

To illustrate the join procedure, we explain how to process a query which has two regular paths, and then present a generalized processing algorithm. We denote the path expression in the form of 'x_n in x_{n-1}/P_1, x_{n+1} in x_n/P_2' as '$P_1x_nP_2x_{n+1}$' for simplicity. And with the term, *index graph*, we mean a index graph for a single path. Suppose P_1 and P_2 are paths which do not use regular expressions. There is only one node for P_1 in the index graph, and also, there is only one node for $P_1 \cdot P_2$ in the index graph.[1] So, we find one extent, e_{p_1}, for P_1 and one extent, e_{p_2}, for $P_1 \cdot P_2$ in the index graph. With these two extents, we perform join processing as follows. If the PID of an object o_2 in e_{p_2} is the path subset of the PID of an object o_1 in e_{p_1}, add the pair (o_1, o_2) into the join result. It is easily seen that the join condition is the path relationship between the PID of an object in e_{p_1} and the PID of an object in e_{p_2}. We evaluate this procedure efficiently using the sort-merge style processing scheme as follows.

First, sort the two extents of e_{p_1} and e_{p_2}. To sort an extent, we divide each element, $e = (\text{oid}, \{r_1,\ldots,r_n\})$ in the extent into (oid, r_1),…,(oid, r_n). For example, extent $\{(o_1,\{(2,4),(7,8)\}), (o_2,\{(2,6),(9,10)\})\}$ will be divided into $\{(o_1,(2,4)), (o_1,(7,8)), (o_2,(2,6)), (o_2,(9,10))\}$. Then, we sort the divided extent by the start point of each element in increasing order (note that each divided element has only one range). When the start points of two elements have the same value, sort the elements using the end points. The above example can be sorted to be $\{(o_1,(2,4)), (o_2,(2,6)), (o_1,(7,8)), (o_2,(9,10))\}$. The following is the join strategy using the sorted extent e_{p_1} and e_{p_2}.

1. Suppose r = (sp, ep) is the range of the first element, o_i, in the sorted extent e_{p_1}, remove every element of which

[1] 1-Index may have several nodes in this case because of its nondeterministic characteristics. Here, we assume the dataguide as the index graph for the purpose of explanation, but we present a generalized algorithm for regular path joins which is not dependent of a specific index graph, later in this section.

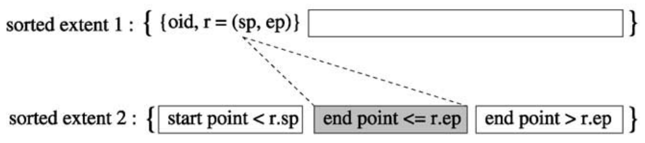
sorted extent 1 : { {oid, r = (sp, ep)} }
sorted extent 2 : { start point < r.sp | end point <= r.ep | end point > r.ep }

Fig. 5. Extent join procedure.

the start point is less than r.sp from the sorted extent e_{p_2}.

2. For each element o_j of which the end point is less than or equal to r.ep in the extent e_{p_2}, if the PID of o_j is a path subset of the PID of o_i then add (o_i, o_j) into the result of the join.
3. Remove the first element, o_i, in e_{p_1}, go to step 1.

When we check the path relationship between two objects in step 2, we use the original PID of each object, not divided one. The above procedure halts when the sorted extent e_{p_1} or e_{p_2} is empty. Fig. 5 shows the join procedure of the two sorted extents.

Algorithm 2. *Regular path join algorithm*

Algorithm 2 Regular path join algorithm

```
1:  // input: root node of the index graph and 0
2:  // output: result set of regular path join query
3:  // there are k Paths to be joined : $P_0, ..., P_{k-1}$
4:
5:  processingJoin(Node $N$, $i$)
6:  {
7:      result is an empty set //a set of the object set
8:
9:      //if node $N$ is the node which satisfies the last path, $P_{k-1}$
10:     if($i$ == k){
11:         foreach(element e in the extent of Node $N$)
12:             add 1-ary tuple, [e], into result
13:         return result
14:     }
15:
16:     //process subgraph($P_{i+1}, ..., P_{k-1}$) first, and store the result into result
17:     foreach(node $n$ that satisfies the regular path $P_i$ from the node $N$)
18:         result = result ∪ processingJoin($n$, $i$+1)
19:
20:     //if there is no path which satisfies $P_{i+1}$ from the node $N$
21:     if(result is empty) return an empty set
22:
23:     //Join the result of subgraph with the extent in the node N
24:     return extentJoin(extent in the node $N$, result)
25: }
26:
27: extentJoin(ext, tuple_set)
28: {
29:     result is an empty set //a set of the object set
30:     s_ext = sort the ext
31:
32:     while(tuple_set != empty && s_ext != empty){
33:         t = first tuple in tuple_set
34:         $e_1$ = first element of s_ext
35:         $e_2$ = first element of t
36:         while($e_1$.sp > $e_2$.sp){
37:             delete t from tuple_set
38:             t = first tuple in tuple_set
39:             $e_2$ = first element of t
40:         }
41:         while($e_1$.ep ≥ $e_2$.ep){
42:             if(PID of $e_1$ is path superset of $e_2$){
43:                 add $e_1$ in front of t
44:                 append t at the end of result
45:             }
46:             t = next tuple in tuple_set
47:             $e_2$ = first element of t
48:         }
49:         delete $e_1$ from s_ext
50:     }
51:     return result
52: }
```

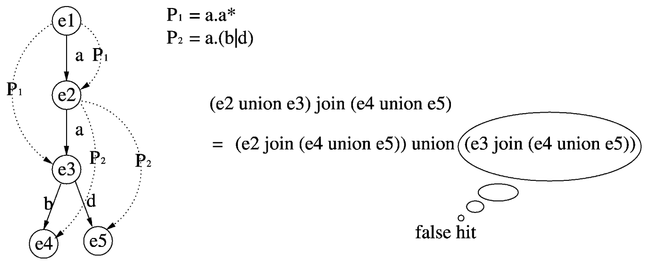
e1
e2
e3
e4
e5
a
a
b
d
P₁
P₁
P₂
P₂
P₁ = a.a*
P₂ = a.(b|d)
(e2 union e3) join (e4 union e5)
= (e2 join (e4 union e5)) union (e3 join (e4 union e5))
false hit

Fig. 6. False hit.

There can be several nodes for P_1 in the index graph when P_1 uses regular expressions. Also, there can be several nodes for $P_1 \cdot P_2$ in the index graph when P_2 uses regular expressions. That is, there can be several extents for each regular path. We can perform (the union of the extents for P_1) $\bowtie$ (union of the extents for $P_1 \cdot P_2$), using the procedure explained above, but this causes *false hits*. For example, in Fig. 6, a query with the path expression $a \cdot a^* xa \cdot (b|d)y$ produces false hits. The graph in Fig. 6 is a index graph and $e_1, \ldots, e_5$ is the extent in each node (for convenience, we also use them as node identifiers). There is no node which satisfies $a \cdot (b|d)$ from e_3, but this query produces false hits of $e_3 \bowtie (e_4 \cup e_5)$, because e_4 and e_5 which satisfy $a \cdot (b|d)$ from e_2 happen to lie on the same path as with e_3.

To prevent false hits, we evaluate the query as follows. Find each extent e_i which satisfies the path P_1. Perform $e_i \bowtie$ (the union of the extents which satisfies P_2 from e_i). Merge the results for each e_i. For example, in Fig. 6, there are two extents, e_2 and e_3 which satisfy the path $a \cdot a^*$. There are two extents, e_4 and e_5 which satisfies the path $a \cdot (b|d)$ from e_2, so perform $e_2 \bowtie e_4 \cup e_5$. There is no extent that satisfies the path $a \cdot (b|d)$ from e_3, so evaluate $e_3 \bowtie \emptyset$ to produce $\emptyset$. Merge the two results to yield $(e_2 \bowtie e_4 \cup e_5) \cup \emptyset$, which is the correct result.

Algorithm 2 is a *multi-way* regular path join algorithm. Multi-way means that there exists more than two relevant path expressions in a query. The function *processingJoin* evaluates each path in a query recursively to join k paths. At line 18, Algorithm 2 performs merge sort by the first element of each tuple in the operands of the union operator. The function *extentJoin* join the two extents which are given as arguments. The first argument of *extentJoin* is the extent of the current node to process (e_i of the above explanation) and the second argument is the result of processing the subtree (subgraph) of the current node. The second argument has the form of

$$\{[(oid_{11}, range_{11}), \ldots, (oid_{1n}, range_{1n})], \ldots, \\ [(oid_{m1}, range_{m1}), \ldots, (oid_{mn}, range_{mn})]\}$$

The function, *extentJoin*, joins the first argument with first element of each tuple in the second argument, which is $\{(oid_{i1}, range_{i1}) | 1 \le i \le m\}$. This function evaluates the very procedure which is shown in Fig. 5. It is easy to see that paths are joined in the order of $P_{k-1}, \ldots, P_0$.

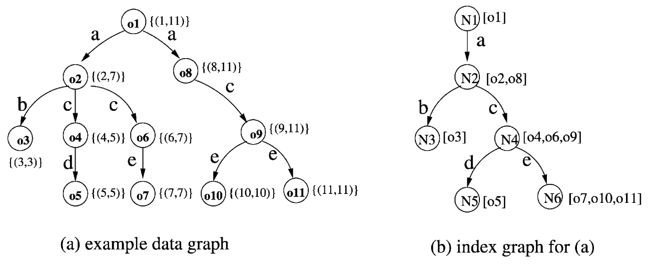

Fig. 7. Example data and index.

To describe how the algorithm works, we use the simple data graph and the corresponding index graph depicted in Fig. 7. For the data graph in Fig. 7, we want to evaluate the following regular path join query.

bind v_1 **in** //a, v_2 **in** v_1/(b|c), v_3 **in** v_2//e **return** v_1, v_2, v_3

Let the path, //a, be P_0,(b|c) be P_1, and //e be P_2. First we call *processJoin*(N1, 0). Subsequent calls for *processJoin* are as follows.

1. processJoin(N1, 0) = N1 ⋈ processJoin(N2, 1)
2. processJoin(N2, 1) = N2 ⋈ (processJoin(N3, 2) ∪ processJoin(N4, 2))
3. processJoin(N3, 2) returns an empty set at line 21 in Algorithm 2
4. processJoin(N4, 2) = N4 ⋈ (processJoin(N5, 3) ∪ processJoin(N6, 3))
5. processJoin(N5, 3) returns{[o5]} at line 13 in Algorithm 2.
6. processJoin(N6, 3) returns{[o7], [o10], [o11]} at line 13 in Algorithm 2.

Here, A ⋈ B stands for the function call, *extentJoin*(A, B). After all possible *processJoin* are called, *extentJoin* are called in the order of (step 4, step 2, step 1). In step 4, *extentJoin* joins the two extents, [o4, o6, o9] and [o5, o7, o10, o11], and the result is{[o6, o7], [o9, o10], [o9, o11]}. In step 2, *extentJoin* joins the two extents, [o2, o8] and [o6, o9, o9], and the result is{[o2, o6, o7], [o8, o9, o10], [o8, o9, o11]}. Note that [o6, o9, o9] is the set of the first element of each tuple in {[o6, o7], [o9, o10], [o9, o11]}. Finally we can get the result of the query {[o1, o2, o6, o7], [o1, o8, o9, o10], [o1, o8, o9, o11]} in step 1. Actually it is unnecessary to evaluate *extentJoin* in step 1, because the root element is joined with every element in the data graph. The second element in each tuple of the result is bound to the variable v_1, the third element in each tuple of the result is bound to the variable v_2, and the fourth element in each tuple of the result is bounded to the variable v_3.

Now, we analyze the time complexity of the function, *extentJoin*. We consider the size of the PID for each node as a constant, c. Let the size of the first argument be m, and the size of the second argument be n. By sorting the first argument at line 30, the size of *s_ext* becomes cm. Let each element in *s_ext* be joined with k elements. The time complexity of the function is $m\log(m) + kcm + n$, where $m\log(m)$ is the cost for sorting the first argument, and kcm is the cost for the comparison to join each element in *s_ext* with k elements, and n is the cost for the comparison to remove each element in the second argument. By storing extents in the sorted form, we can make the time complexity of *extentJoin* to be $O(km + n)$.

Relation to descendant-or-self axis in XPath. The descendant-or-self axis (//) searches through all the descendants of the context node, starting with the context node itself. For the path expression, $/P_1//P_2$, we can use our technique to support descendant-or-self axis if we can find all node which satisfy the path, P_1, from the root node, and all nodes which satisfy the path, P_2, from any node of the data graph. For each node, n_1, which satisfy P_1, and for each node, n_2, which satisfy P_2, if n_2 is a descendant of n_1 then n_2 is the result of the above query. Here descendant means that there exists a path from n_1 to n_2, and checking the path relationship between two nodes is what the *extentJoin* do.

6. Preliminary result

In our experiments, we compared the number of fetched objects (nodes) between the technique using PID and the Plan 1 described in Section 1. We did not consider I/O optimization techniques such as object clustering and prefetching into consideration. We only use the number of fetched objects from the disk to the memory as the performance measure. We applied our technique to the following data set.

- *Astronomical data of NASA* [20]. This data set consists of many small XML files which contain astronomical information, publications, etc.
- *Hamlet*. These data are the Hamlet of the Shakespeare's Plays [9] in XML format.
- *Internet movie data* [11]. This data set consists of many small XML files containing information about movies and actors.
- *Synthetic data for Major League Baseball*. These data are randomly generated data about Major League Baseball.

We merge small XML files into one XML data for the astronomical data set and the Internet movie data set.

Table 1
Experimental results—query 6.3, 6.4, 6.5

	Number of objects in the data	With PID	Without PID
Query 6.3	12,158	34	6611
Query 6.4	70,521	373	40,971
Query 6.5	3843	56	1284

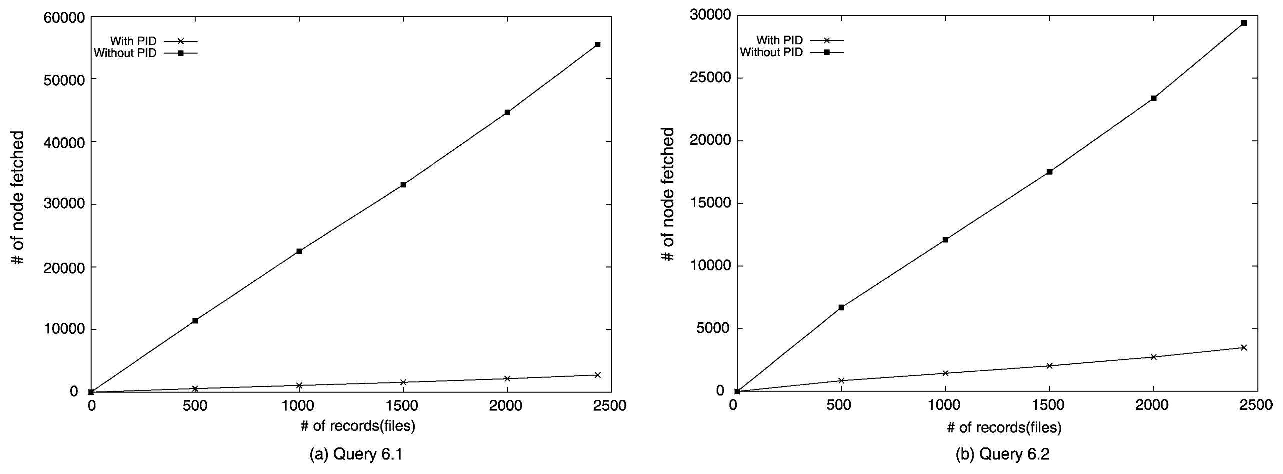

Fig. 8. Experimental results—query 6.1, 6.2.

Specially, we ran experiments consisting of two queries varying the number of files for the astronomical data set.

We use the following queries in our experiments.

Query 6.1
bind x **in** //reference//(journal|other), y **in** x//year **return** x, y
Query 6.2
bind x **in** //history, y **in** x//creator **return** x, y
Query 6.3
bind x **in** /play//scene, y **in** x//speaker[text( ) = " Francisco"] **return** x
Query 6.4
bind x **in** //Movie, y **in** x//Year[text( ) = "1975") **return** x, y
Query 6.5
bind x **in** /MLB//EAST/player, y **in** x//nickname y **return** x, y

Query 6.1 and 6.2 are for the astronomical data set, query 6.3 is for the Hamlet data, query 6.4 is for the Internet movie data set, and query 6.5 is for the synthetic data for the Major League Baseball. Table 1 shows the result of query 6.3, 6.4, and 6.5. And Fig. 8 shows the result of query 6.1 and 6.2. As depicted in Fig. 8, the technique without using the PID searches the data graph to find the answer to the second path expression, so the number of fetched objects increases rapidly as the size of data grows, but our algorithm does not depend on the size of the data graph, and the number of fetched objects is proportional to the number of result objects as the size of the data graph is increased.

The performance of our algorithm depends on only the size of schema graph, and our algorithm always fetches less than (the size of schema graph + the number of result) objects regardless of the size of the data graph and the number of regular paths in a query. Although the size of a schema graph can grows greater than or equal to the size of the data graph, in many cases the size of schema graph is much smaller than the size of source data graph as our experiments.

7. Conclusions and future work

Regular path expressions are a useful and necessary feature of query languages for XML and semi-structured data. As a query tends to have more than one regular path in it, the *regular path join queries* are as important as the join queries in relational databases.

We define the PID and present a method to identify whether or not two given nodes lie on the same path using the PID. Also, an efficient regular path join algorithm is presented which uses the PID. We implement our algorithm and show the superiority of our algorithm from the preliminary results. Our technique performs better with other research results. For example, pruning the query [13,15] before the query processing will apparently decrease the number of fetched objects.

We present a simple scheme to maintain the PID incrementally in Section 2, but in the case that the input data is skewed, PIDs in the data graph must be frequently reconstructed. We plan to modify the method of incremental maintenance for the PID to be more suitable in the environment where the data is dynamically updated.

Acknowledgements

This work was supported by the Brian Korea 21 Project, the Ministry of Information and Communication, Korea through the ITRC, and the Ministry of Commerce, Industry and Energy, korea through XDOM project.

References

[1] S. Abiteboul, V. Vianu, Regular path queries with constraints, Proceedings of the ACM Symposium on Principles of Database Systems (1997).

[2] S. Abiteboul, Querying semi-structured data, Proceedings of the International Conference on Database Theory (1997).

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, The lorel query language for semistructured data, International Journal on Digital Libraries (1996).

[4] B. Cooper, N. Sample, J.J. Franlin, G.R. Hjaltason, M. Shadmon, A fast index for semistructured data, Proceedings of the Conference on Very Large Data Bases (2001).

[5] S. Baase, Computer Algorithms. Introduction to Design and Analysis, Addison-Wesley, Reading, MA, 1988.

[6] T. Bray, J. Paoli, C. Sperberg-McQueen, Extensible markup language (XML) 1.0, Technical report, W3C Recommendation, 1998.

[7] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, A query language and optimization techniques for unstructured data, Proceedings of the ACM SIGMOD International Conference on the Management of Data (1996).

[8] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi, Rewriting of regular expressions and regular path queries, Proceedings of the ACM Symposium on Principles of Database Systems (1999).

[9] R. Cover, The XML Cover Pages, http://xml.coverpages.org.

[10] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, M. Stefanescu, XQuery: a query language for XML, Technical report, W3C Working Draft, February 2001.

[11] The Internet Movie Database, http://www.imdb.com.

[12] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, Query language for XML, Proceedings of the Eighth International World Wide Web Conference (1999).

[13] M. Fernandez, D. Suciu, Optimizing regular path expressions using graph schemas, IEEE International Conference on Data Engineering (1998).

[14] G. Grahne, A. Thomo, An optimization technique for answering regular path queries, International Workshop on the Web and Databases (2000).

[15] R. Goldman, J. Widom, DataGuides: enabling query formulation and optimization in semistructured databases, Proceedings of the Conference on Very Large Data Bases (1997).

[16] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.

[17] Q. Li, B. Moon, Indexing and querying XML data for regular path expressions, Proceedings of the Conference on Very Large Data Bases (2001).

[18] J. McHugh, J. Widom, Compile-time path expansion in lore, Proceedings of the Workshop on Query Processing for Semistructured Data and Non-standard Data Formats (1999).

[19] T. Milo, D. Suciu, Index structures for path expressions, Proceedings of the International Conference on Database Theory (1999).

[20] NASA, Astronomical Data Center, http://tarantella.gsfc.nasa.gov/xml/.

[21] S. Nestorov, J. Ullman, J. Wiener, S. Chawathe, Representative objects: concise representations of semistructured, hierarchical data, IEEE International Conference on Data Engineering (1997).

[22] Y. Papakonstantinou, V.A. Vassalos, Query rewriting using semi-structured views, Proceedings of the ACM SIGMOD International Conference on the Management of Data (1999).

[23] D. Suciu, Semistructured data and XML, Proceedings of the International Conference on Foundations of Data Organization (1998).

[24] D. Suciu, M. Fernandez, S. Davidson, P. Buneman, Adding structure to unstructured data, Proceedings of the International Conference on Database Theory (1997).

[25] W3C, XML Path Language (XPath) 1.0, W3C Recommendation, 1999.