ELSEVIER

The Journal of Systems and Software 64 (2002) 195–205

The Journal of Systems and Software

www.elsevier.com/locate/jss

XML query processing using document type definitions ☆

Tae-Sun Chung *, 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 9 May 2001; received in revised form 21 September 2001; accepted 21 December 2001

Abstract

As eXtensible Markup Language (XML) has become an emerging standard for information exchange on the World Wide Web, it has gained attention in database communities to extract information from XML seen as a database model. Data in XML can be mapped to a semistructured data model based on edge-labeled graph, and queries can be processed against it. Here, we propose new query optimization techniques using document type definitions which have the schema information about XML data. Our techniques reduce the large search space significantly while at the same time requiring less memory compared to the traditional index techniques. Also, as they preserve source database's structure, they can process many kinds of complex queries. We implemented our techniques and provided preliminary performance results.

1. Introduction

1.1. Motivation and problem definition

Recently, as eXtensible Markup Language (XML) has become an emerging standard for information exchange on the World Wide Web, it has gained attention in database communities to extract information from XML seen as a database model. That is, it is clear that an enormous amount of data in the Internet will be encoded in XML in the near future. So, the technique of querying XML documents will play a key role in the applications such as electronic commerce, e-newspaper, information retrieval, and so on.

There are two kinds of approaches to querying XML documents. One is using traditional databases such as relational databases or object-oriented databases for storing and querying XML documents. The other is using special purpose query engines for semistructured data since an XML document can be regarded as an instance of a semistructured data set. We assume the latter in this paper. However, our technique can also be applied in the former if the graph based schema is constructed to store XML documents.

If we assume that XML data is mapped to a data graph, the basic problem of XML query processing is how to find the result of the single regular path query which is defined as follows.

Definition 1 (*Single regular path query*). Given a regular path expression r and a data graph D, the result of r on D is the set of objects on D that are reachable by the regular path expression r.

The result of the single regular path query is computed by traversing the data graph with the regular path expression. This query evaluation technique based on graph traversal is usually inefficient compared to that of traditional database systems since there is no schema fixed in advance. That is, to process given queries targeted to specific schemas in traditional database systems, a query processor can only process the schemas targeted. On the other hand, in semistructured data models, the entire data graph should be processed by a query processor.

However, in XML there are document type definitions (DTDs) (Bray et al., 1998) which provide schema information, so we can reduce the search space of the graph by using DTDs. Although the DTD is an optional feature of XML, the DTD can be inferred from XML

☆ This work was supported by the Brain Korea 21 Project.
* Corresponding author.
E-mail addresses: tschung@papya.snu.ac.kr (T.-S. Chung), hjk@papya.snu.ac.kr (H.-J. Kim).

0164-1212/02/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
PII: S0164-1212(02)00042-0

data by the technique proposed in Garofalakis et al. (2000). Additionally, XML schemas (Fallside, 2001) that are extensions to DTDs are being developed by the Web community. XML schemas extend DTDs by adding typing information. However, we concentrate on DTDs in this paper since DTDs are simpler than XML schemas and XML schemas are still evolving. Our technique can be aided when XML schemas are available.

For example, we process a query Q1, given the following DTD. [1]

```
<!ELEMENT MLB (National|American)+>
<!ELEMENT National (East|Central|West)>
<!ELEMENT American (East|Central|West)>
<!ELEMENT East (stadium?, name, player+)>
<!ELEMENT Central (stadium?, name,
player+)>
<!ELEMENT West (stadium?, name,
player+)>
<!ELEMENT Player (nickname?, name, ERA?,
AVG?, win?)>

Q1: select x
    where (MLB.National.Central.Player.
nickname) x
```

This query asks for the nicknames of the players in the National League Central Division teams. Assuming that there are no indexes, the query processor finds all the objects reachable by the path, `MLB.National.Central.Player.nickname`, and then returns the objects. However, dealing with data in XML, DTDs provide the following information about the query Q1.

1. The MLB consists of National League and American League teams.
2. A National League team belongs to one of East, West, or Central Divisions.
3. A player has or doesn't have a nickname.

Using these information, the query search space can be reduced by searching for only the players who belong to the National League Central Division teams and have nicknames. In this paper, we propose the query optimization techniques using DTDs.

1.2. Contributions and organization

Our techniques improve the previous work as follows. First, our techniques can be applied to queries having several regular path expressions. For example, one can issue the following query that asks for the National League Central Division teams that have the player with the nickname "Big Mac".

```
select x
where (MLB.National.Central) x (Player.
nickname."Big Mac") y
```

Many researchers have proposed enhanced query evaluation techniques by schema extraction (Goldman and Widom, 1997; Nestorov et al., 1997; Milo and Suciu, 1999). That is, given a particular data instance of large size, the technique finds the schema for it and traverses the schema graph of small size instead of the data graph. However, the techniques are not adequate for queries having multiple regular path expressions. This is because that the queries like the one above cannot be evaluated by only traversing the schema graph. On the other hand, since our techniques preserve source database's structure, they can efficiently process queries having multiple path expressions. Second, our techniques construct index information efficiently and do not require much additional storage for indexes.

The paper is organized as follows. Section 2 mentions related work, Section 3 presents the overview of our approach. Section 4 describes the classification of the DTD elements which is used in query optimization. In Section 5, we propose our query optimization techniques. Section 6 presents the preliminary results. And finally, we conclude in Section 7.

2. Related work

First, the overviews of semistructured data models are summarized in Abiteboul (1997) and Buneman (1997). Here, semistructured data is represented in edge-labeled graph, in which the nodes correspond to the objects, and the edges to their attributes.

In semistructured data represented in edge-labeled graph, query languages are derived from those of object-oriented DBMSs such as OQL (Cattell, 1994) and XSQL (Kifer et al., 1992). In these query languages, the expressive power is improved by using path expressions that may have variables that range over classes and attributes rather than data values in RDBMSs.

However, these query languages designed for well structured data are inappropriate in semistructured data whose structure may be irregular or incomplete. So, many semistructured query languages based on the regular path expression that allows many kinds of regular expressions in queries have been proposed. These include XQuery (Chamberlin et al., 2001), UnQL (Buneman et al., 1996), Lorel (Abiteboul et al., 1996), XML-QL (Deutsch et al., 1999), and so on. Our techniques proposed in this paper can be applied to these kinds of queries.

[1] We omitted the form of <!ELEMENT element_label (#PCDATA)>.

The query optimization techniques for semistructured data are derived from path indexes (Bertino and Kim, 1989) in OODBMSs. Path indexes introduced in Bertino and Kim (1989), index on particular paths rather than attributes of relations.

Theoretical foundations to query processing for semistructured data are studied in Abiteboul and Vianu (1997) and Mendelzon and Wood (1995). In Abiteboul and Vianu (1997), the authors define regular path queries that find all objects reachable by paths whose labels form a word in a regular expression over an alphabet of labels, and propose the query optimization techniques which use information about path constraints. The complexities of query processing in a graph database is studied in Mendelzon and Wood (1995). The authors only treat simple paths, that is, those that have no cycles. We apply the theoretical foundations in our framework and design practical algorithms for real applications.

The query optimization techniques using graph schemas are proposed in Fernandez and Suciu (1998) and Suciu et al. (1997). By using graph schemas which have partial information about a graph's structure, they reduce the large search space by query pruning and query rewriting. However, graph schemas cannot be constructed automatically, while we provide automatic construction algorithms. The technique using graph schemas has characteristics whereby it defines graph schemas statically and processes queries for data that conforms to them.

On the other hand, DataGuides (Goldman and Widom, 1997; Nestorov et al., 1997) focus on data and record information about all the paths in a database dynamically, and use them as indexes. DataGuides can be used efficiently in the environment where there is no schema information provided in advance. However, this technique can be applied to only queries with a single regular expression. That is, it cannot be directly applied to complex queries with several regular expressions and variables. T-index (Milo and Suciu, 1999) provides a general indexing mechanism in evaluating queries in semistructured data. And it is constructed by the computation of a simulation or a bisimulation, for which there exist efficient algorithms. However, the target query should conform to the corresponding template.

The cost based optimization technique is addressed in McHugh and Widom (1999a,b). It generates optimal plans based on new kinds of indexing for semistructured data and database statistics. Our result can be added its plan space as another efficient plan.

3. Overview of our approach

3.1. Data model

We assume that data in XML is mapped to an object exchange model (OEM) (Papakonstantinou and Abiteboul, 1996) graph that is the de facto model for semistructured data. Every object in OEM consists of an identifier and a value, and the nodes in the graph are objects and the edges are labeled with attribute names. The OEM objects are classified as the following two kinds of objects, according to their values.

- *Atomic objects*: The value of the atomic objects is an atomic quantity, such as an integer, a string, an image, a sound, and so on.
- *Complex objects*: The value of the complex objects is a set of <label, id> pairs.

XML corresponds closely to semistructured data based on edge-labeled graph, so it is possible to map data in XML to a semistructured data model such as OEM. Before mapping data in XML to an OEM graph, we assume that no element have attributes other than the attribute `ID` and the attribute `IDREF` (Bray et al., 1998). The XML elements which have attributes other than those mentioned above can be redefined as ones that do not have them in the following manner (Suciu, 1998).

```
<Paper format="ps">
  <author> Serge Abiteboul </author>
</Paper>
```

This is converted to the following XML data.

```
<Paper>
  <format> ps </format>
  <value>
    <author> Serge Abiteboul </author>
  </value>
</Paper>
```

Data in XML can be represented by the OEM model. [2] That is, XML elements are represented by nodes of an OEM graph and element–subelement, element-attribute, and reference relationships are represented by edges labeled by the corresponding names. Values of XML data are represented by leaves in the OEM graph.

Fig. 1 shows an XML data and a corresponding OEM graph. Here, &0, &1, etc. are object identifiers. Objects such as &5 and &6 are atomic objects and those such as &1 and &2 are complex objects.

3.2. Key idea

From the flexibility of XML data, we can classify each element using DTDs and give a hint to a query processor in run time. For example, let's assume that a

[2] In this case some information can be lost.

```
<AGroup>
  <person id="id1">
    <name> park </name>
    <e-mail> park@papaya </e-mail>
    <company-link company="id2"/>
  </person>
  <company id="id2">
    <name> cnn </name>
    <person-link person="id1"/>
    <url> http://www.cnn.com </url>
  </company>
  <person id="id3">
    <name> kim  </name>
    <school-link school="id4"/>
  </person>
  <school id="id4">
    <name> snu </name>
    <baseball-team> lions </baseball-team>
    <person-link person="id3"/>
  </school>
</AGroup>
```

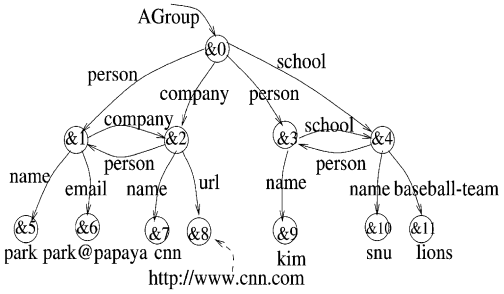


Fig. 1. An example of XML data and OEM graph.

DTD declaration for the person element in Fig. 1 is as follows:

$$< !\texttt{ELEMENT person (name, e-mail}^{*}\texttt{, (school-link|company-link))} > \quad (1)$$

From the DTD, we can classify the person element into four groups: (1) ones who have one or more e-mail addresses and work for companies, (2) ones who have no e-mail address and work for companies, (3) ones who have one or more e-mail addresses and are students, and (4) ones who have no e-mail address and are students. When each element is classified in this way, the search space can be reduced. For example, when the query that is related to students who have e-mails is processed, the nodes denoting persons who have no e-mail and work for companies need not be traversed.

In this paper, we present a method of classification of DTD elements (Section 4), and query optimization techniques using this information (Section 5).

4. Classification of DTD elements

Data in XML can be represented as an OEM graph using the method in the previous section, and queries can be processed over it. At this point, the problem is that the graph search space is too large.

Our techniques make use of the DTD information to reduce the search space. DTDs provide structural information about elements by regular expressions. So, we can classify DTD elements from DTDs. First, we make some assumptions about DTDs as in Papakonstantinou and Velikhov (1999), i.e., XML documents always have DTDs, and do not have attributes other than the ID and IDREF attributes as mentioned in Section 3.1. Additionally, DTDs do not have mixed contents elements whose contents mixes strings with elements. If this is the case, our technique bypasses the declaration since they do not give information to the query processor. Then, let N be a set of element names, we can abstract the element declaration in a DTD as a set of $(n : r)$ pairs, where $n \in N$, r is either a regular expression over N or PCDATA which denotes a character string or EMPTY which denotes an empty element.

4.1. DTD automata

The regular expressions appearing in DTDs can be divided into five categories, as follows. If r, r_1, and r_2 are regular expressions that DTDs represent, $L(r)$, $L(r_1)$, and $L(r_2)$ are the languages that can be described by the regular expressions.

1. *Case* $r = r_1, r_2$: The languages that r denotes are the concatenation of $L(r_1)$ and $L(r_2)$.
2. *Case* $r = r_1|r_2$: $L(r)$ is the union of $L(r_1)$ and $L(r_2)$. This kind of DTDs becomes important hints in query processing. For instance, in the MLB DTD, the element of 'National' is the union of 'East', 'Central', and 'West'. This information can be used when the query which has a label of 'National' is processed. That is, the query search space can be divided into three categories, according to the query's having a label of 'East', 'Central', or 'West'.
3. *Case* $r = r_1+$: This represents one or more occurrences of the same structure. As these regular expressions mean that particular attributes exist more than once, this kind of information is not necessary in reducing the search space. That is, we should process all the attribute values when an attribute exists more than once.

4. *Case* $r = r_1*$: This represents zero or more than one occurrence of the same structure. This can be a hint to the query processor since the elements of this form can be divided into two types according to having one or more than one corresponding structure or not.
5. *Case* $r = r_1?$: This kind of regular expressions can be a hint as in case 2. In the previous MLB DTD, a player can have an attribute `nickname` or not. This can be a piece of information to reduce the query search space. That is, if a query is related to players having a nickname, the query processor need not traverse the player who doesn't have a nickname.

So, we define the following relaxed regular expression to extract only the necessary information during the query processing.

Definition 2 (*Relaxed regular expression*). A relaxed regular expression is constructed from a given regular expression as follows.

1. $r_1, r_2 \Rightarrow r_1, r_2$
2. $r_1|r_2 \Rightarrow r_1|r_2$
3. $r+ \Rightarrow r$
4. $r* \Rightarrow r + | \perp \Rightarrow r| \perp$ (by rule 3)
5. $r? \Rightarrow r| \perp$

Example 1. The DTD declaration in Formula (1) is abstracted to (person: (name, e-mail*, (school| company))), and the corresponding relaxed regular expression is (person: (name, (e-mail| $\perp$), (school| company))).

DTD automata are constructed in the following ways. Let $(n_i : r'_i)$ be an expression which is obtained by applying relaxed regular expressions to each DTD declaration $(n_i : r_i)$. We construct automation A_i by Algorithm 1 with a new regular expression $n_i r'_i$. [3] These automata which have the DTD information classify the each DTD element, and they are used in pruning the search space. Here, we use the notation about automata in Hopcroft and Ullman (1979). An automaton is denoted by a five-tuples $(Q, \Sigma, \delta, q_0, F)$, and the meanings of each tuple are as follows.

- Q is a finite set of states. The notation $[q_1, q_2]$ describes a new state that is related to the states q_1 and q_2.
- Σ is a finite input alphabet, namely labels in DTDs and a special symbol $\perp$.
- q_0 is the start state with $q_0 \in Q$.
- F is the set of all final states with $F \subseteq Q$.
- δ is the transition function which maps $Q \times \Sigma$ to Q.

Algorithm 1 is similar to the standard automata construction for a regular expression. However, in our technique, since the input regular expression is a relaxed regular expression, it directly derive a nondeterministic finite automaton. On the other hand, in the traditional technique, first, an NFA with ϵ-transitions is constructed. The label $\perp$ can be regarded as engendering an epsilon transition in the DTD automata. However, since the label is not treated as an epsilon transition but as an input symbol in our algorithm, it doesn't require further nondeterminism. The following theorem shows the soundness of Algorithm 1.

Theorem 1. *There always exists an automaton M constructed by Algorithm* 1 *for the input regular expression r, and if $L(M)$ is the language accepted by M, and $L(r)$ is the language which is describable by the regular expression r, then $L(M) = L(r)$.*

The proof is shown in the Appendix A.

Example 2. Fig. 3 shows an automaton constructed by Algorithm 1 for the person element in Example 1.

Algorithm 1 (*The construction of DTD automata*).

1: **Input:** A relaxed regular expression r
2: **Output:** An automaton M
3: **procedure** Make_DTD_Automata (regular expression r)
4: **if** $r = a$ $(a \in \Sigma)$ **then**
5: Construct an automaton M as shown in Fig. 2;
6: return M;
7: **else if** $r = r_1|r_2$ **then**
8: $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1) \leftarrow$ Make_DTD_Automata(r_1);
9: $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2) \leftarrow$ Make_DTD_Automata(r_2);
10: Construct the new automaton $M = (Q_1 - \{q_1\} \cup Q_2 - \{q_2\} \cup [q_1, q_2], \Sigma_1 \cup \Sigma_2, \delta, [q_1, q_2], F_1 \cup F_2)$ from the automata M_1 and M_2, where δ is defined by
1. $\delta(q, a) = \delta_1(q, a)$ for $q \in Q_1 - \{q_1\}$ and $a \in \Sigma_1$,
2. $\delta(q, a) = \delta_2(q, a)$ for $q \in Q_2 - \{q_2\}$ and $a \in \Sigma_2$,
3. $\delta([q_1, q_2], a) = \delta_1(q_1, a)$ where $a \in \Sigma_1$,
4. $\delta([q_1, q_2], a) = \delta_2(q_2, a)$ where $a \in \Sigma_2$;
11: **else** $\{r = r_1, r_2\}$

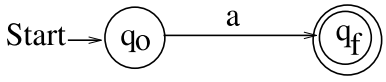


Fig. 2. $r = a$.

[3] In this paper, we occasionally omit concatenation operator, that is, $n_i r'_i = n_i, r'_i$.

12: $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1) \leftarrow$ Make_DTD_Automata(r_1);
13: $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2) \leftarrow$ Make_DTD_Automata(r_2);
14: Let the final states F_1 of M_1 be states $f_1, f_2, \ldots, f_m$ ($m \geqslant 1$). Construct the new automaton $M = (Q_1 - F_1 \cup Q_2 - \{q_2\} \cup \{[f_1, q_2], [f_2, q_2], \ldots, [f_m, q_2]\}, \Sigma_1 \cup \Sigma_2, \delta, q_1, F_2)$ from the automata M_1 and M_2, where δ is defined by

1. $\delta(q, a) = \delta_1(q, a)$ for $q \in Q_1 - F_1$, $\delta_1(q, a) \neq f_k$ (where $1 \leqslant k \leqslant m$), and $a \in \Sigma_1$,
2. $\delta(q, a) = \delta_2(q, a)$ for $q \in Q_2 - q_2$ and $a \in \Sigma_2$,
3. $\delta([f_k, q_2], a) = \delta_2(q_2, a)$ for all k (where $k = 1, 2, \ldots, m$) and $a \in \Sigma_2$,
4. $\delta(q_f, a) = [f_k, q_2]$ for all q_f which satisfies $\delta_1(q_f, a) = f_k$ (where $1 \leqslant k \leqslant m$) and $a \in \Sigma_1$;

15: **end if**
16: return M

4.2. Classification of DTD elements

In this section, using the DTD automata constructed by Algorithm 1, we classify each element of DTDs. As the DTD automata are constructed from relaxed regular expressions, they contain information only about concatenations and unions. Here, the diverging points in automata become those of the query search. So, by recording the labels at diverging points we can classify the DTD elements. We define the classification tree as the tree having labels at the diverging points of the DTD automata. Algorithm 2 shows the construction of classification trees. It traverses from the start state to the final states of an automaton M_k, and constructs the classification tree recursively. In Algorithm 2, as each transition is processed exactly once, the complexity of it is in $O(m)$, where m is the number of transitions in the automaton.

In Algorithm 2, the function transition(q) returns p when there is a transition function $\delta_k(q, a) = p$. No_effect_label$_k$ stores the set of labels which do not affect the classification of the DTD elements, and it is used when we restructure the OEM graph in the next section. For instance, the No_effect_label for the DTD element of the person is {person, name}.

Algorithm 2 (*The construction of classification trees from DTD automata*).

1: **Input:** $M_k = (Q_k, \Sigma_k, \delta_k, q_k, F_k)$ (for $k = 1, 2, \ldots, n$)
2: **Output:** The classification tree T_k (for $k = 1, 2, \ldots, n$)
3: **procedure** Make_classification_tree (state q, automaton M_k)
4: **if** $q \in F_k$ **then**
5: make a vertex q' corresponding to a state q;
6: return q';
7: **else**

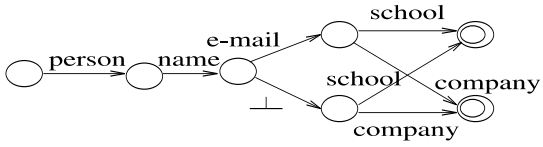


Fig. 3. A DTD automaton.

8: **if** transition(q) has more than two states **then**
9: make a vertex q' corresponding to a state q;
10: let T be a tree rooted q' and having children of Make_classification_tree(w,M_k) for all w where $w \in$ transition(q) with edges labeled a in the transition;
11: return T;
12: **else**
13: $q \leftarrow \delta_k(q, a)$;
14: No_effect_label$_k$ = No_effect_label$_k \cup a$;
15: Make_classification_tree(q,M_k);
16: **end if**
17: **end if**

Example 3. Fig. 4 shows a classification tree and a corresponding classification table constructed from the DTD automaton of the person element in Fig. 3 using Algorithm 2.

5. Query optimization

Using the classification tables, we restructure an input OEM graph, and provide a query processor with information about reducing the search space. Here we propose two techniques. One is that the query processor keeps classification information only about each object and the other is about all objects under the target object.

Before we describe our techniques, we define path expressions that occur in queries.

Definition 3 (*Regular path expression*). A regular path expression is a form of $H.P$ where

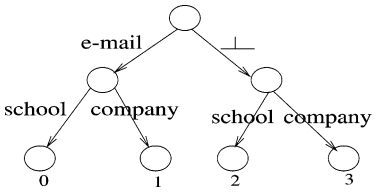
1. H is an object name or a variable denoting an object,
2. P is a regular expression over labels in an OEM graph. Namely, $P = \text{label}|(P|P)|(P.P)|P*$.

Definition 4 (*Simple regular path expression*). A simple regular path expression is a sequence $H.p_1.p_2 \ldots p_n$ where

1. H is an object name or a variable denoting an object,
2. p_i ($1 \leqslant i \leqslant n$) is a label in an OEM graph or wild-card "$*$" which denotes any sequence of labels.

5.1. NodeInfo

Algorithm 3 shows the construction of NodeInfo.



0	{e-mail, school}
1	{e-mail, company}
2	{school}
3	{company}

Fig. 4. A classification tree and a classification table.

The inputs of Algorithm 3 are an OEM graph which specifies data in XML, and the classification table constructed in the previous section. The classification table T is a two-dimensional array and the value of T[element_label][i] is a label set of element_label's ith group. For instance, $T[\text{person}][1] = \{\text{e-mail}, \text{company}\}$ in Fig. 4. Algorithm 3 traverses the OEM graph in a post order way, and determines the value of the variable `node_info`, which stores the index of the label set that corresponding objects belongs to in the classification table.

Algorithm 3 (*The construction of NodeInfo*).

Input: An OEM graph, and a classification table T[element_label][index]
Output: An OEM graph with `node_info`
procedure PostOrderNodeInfo(vertex v)
for all w which has an connected edge from v and it is unmarked **do**
  PostOrderNodeInfo(w);
end for
if v is a complex object
  $L \leftarrow$ element_label(v);
  $S \leftarrow$ child_label_set(v) $\cup L$;
  $S \leftarrow S -$ No_effect_label$_k$; {assume L is kth element in a DTD}
  for all $T[L]$[index] where $0 \leqslant$ index $<$ number of groups corresponding to L **do**
    if $T[L]$[index] $- S = \phi$ **then**
      v.node_info = index;
      break;
    end if
  end for
end if
mark v;

In Algorithm 3, the function child_label_set(v) returns the set of child labels which the node v has. For example, in Fig. 1, child_label_set(&2) = {person, name, url}. The function element_label(v) returns the element label corresponding to the node v. For example, in Fig. 1, element_label(&1) = person.

Each node v in an OEM graph is processed as follows. When the node v is an atomic object, it has no meaning in constructing NodeInfo since it has no child label. When the node v is a complex object, Algorithm 3 finds the group which the node belongs to from its child labels by referring to the classification table, and keeps the index of the label set in the table to the variable `node_info`.

The NodeInfo technique gives classification information about each object to a query processor. For example, the person element is divided into four groups as in Fig. 4, and the object &1 in Fig. 1 belongs to 1:{e-mail, company} and the object &3 to 2:{school}. The variable `node_info` in the NodeInfo technique stores the index of the label set to which the corresponding object belongs in the classification table. For example, the object &1 has `node_info` of 1 which is an index of a label set {e-mail, company}.

The construction cost of NodeInfo in the worst case is in $O(kn)$ where k is the maximum number among the number of groups for each DTD element, and n is the number of nodes. So, it is superior to DataGuides (Goldman and Widom, 1997; Nestorov et al., 1997) of exponential cost in the worst case. Moreover, in average case, it is usually superior to 1-index (Milo and Suciu, 1999) of $O(m \log n)$ construction cost under a graph with m edges and n nodes.

The NodeInfo technique can process the queries that only have simple regular path expressions. First, we define a variable classification_info for simple regular path expressions.

Definition 5. Let $H.p_1.p_2 \ldots p_n$ be a simple regular path expression. For each p_i ($i = 1, 2, \ldots, n-1$), the value of p_i's classification_info is

1. {} where $p_{i+1} = *$, or $p_{i+1} \neq *$ and $p_{i+1} \in$ No_effect_label, [4] and
2. $\{p_{i+1}\}$ where $p_{i+1} \neq *$ and $p_{i+1} \notin$ No_effect_label.

Example 4. Consider the simple regular path expression 'AGroup.person.*.e-mail'. Here, $p_1 =$ person and $p_2 = *$. So, the classification_info of p_1 *is* {} and that of p_2 is {e-mail}.

Query processing in the NodeInfo technique is performed as follows. To find all objects reachable by a simple regular path expression $H.p_1.p_2 \ldots p_n$ a query processor begins searching the graph. When the query processor searches an object v which has a directed edge to an object w and the corresponding label is p_i ($1 \leqslant i \leqslant n-1$), the search should be expanded to the object w when the condition classification_info of $p_i -$ classification_table[element_label of w][w.node_info] $= \phi$ holds.

[4] No_effect_label is an union set of No_effect_label$_k$ for all k.

Example 5. Consider the simple regular path expression 'AGroup.person.company.url' against the data graph in Fig. 1. In a naive method, the objects &1 and &3 should be traversed. However, in the NodeInfo technique, when the query processor searches the object &0 and reads the values for the object &3, p_i = person, and classification_info of p_i − classification_table [person] × [w.node_info] = {company} − {school} ≠ ϕ. So it doesn't traverse the object &3.

5.2. MergeNodeInfo

The MergeNodeInfo technique provides classification information about all the objects that are reachable from the target object. That is, in the MergeNodeInfo technique, the variable `merge_node_info` has the union of all `node_info` of its descendants. For example, `merge_node_info` of the object &1 is {e-mail, company, url}.

The size of MergeNodeInfo is in O(tn) where t is the cardinality of a difference in set between the set of labels which exist in a DTD and the set of labels in No_effect_label, and n is the number of objects in the OEM graph. In DataGuides, the size may be as large as exponential in that of database.

The MergeNodeInfo technique can process the queries that have simple regular path expressions and regular path expressions. First, we define a variable merge_classification_info of p_i for simple regular path expressions in queries.

Definition 6. Let $H.p_1.p_2 \ldots p_n$ be a simple regular path expression that exists in a query. The merge_classification_info of p_i $(i = 1, 2, \ldots, n-1)$ is defined by

$\cup_{k=i}^{n-1}$ classification_info of p_i.

Query processing in the MergeNodeInfo technique is as follows. To find all objects reachable by a simple regular path expression $H.p_1.p_2 \ldots p_n$, a query processor begins searching the graph. When the query processor searches an object v which has a directed edge to an object w and the corresponding label is p_i $(1 \leqslant i \leqslant n-1)$, the search should be expanded to object w when the condition merge_classification_info of $p_i - w$.merge_node_info = ϕ holds.

Example 6. Consider the simple regular path expression 'AGroup.person.*.baseball-team'. To find all objects reachable by this path in a naive method, the whole graph should be traversed to find the label 'baseball-team' which follows the '*' symbol. However, in the MergeNodeInfo technique, when the query processor searches the object &0 and reads its values for the object &1, p_i = {person} and merge_classification_info of $p_i - w$.merge_node_info = {baseball-team} − {e-mail, company, url} ≠ ϕ. So, the objects under the object &1 is not traversed.

The MergeNodeInfo technique can be used for queries that have regular path expressions without the '|' operator. [5] In this case, query processing is carried out as follows. First, we define the following variables.

Definition 7. When a query processor searches an object v which has a directed edge to an object w, the variables P, Q, and R are defined as follows.

- P: A difference in a set between the set of labels which exist in the query and the set of labels in No_effect_label,
- Q: A set of labels in `merge_node_info` for the node w,
- R: A set of labels in the path from the root to the node w.

Here, when the condition $P - (Q \cup R) = \phi$ is satisfied, the search is expanded from node v to node w.

Example 7. Consider the regular path expression 'Bib.paper(.section)*.figure' with a new bibliography database. We assume that an object v which denotes a section in the database and there are no figures in all the subsections of it. Then, when the query processor searches the object v, it is a possible condition that P = {paper, section, figure}, Q = {section}, and R = {paper, section}. So, $P - (Q \cup R)$ = {figure} ≠ ϕ, after which it stops searching.

6. Preliminary results

We have implemented our techniques described in this paper with about 3000 lines of Java code to illustrate their enhancement in query processing. Additionally, we implemented the well known dataguide algorithm and compared it with our algorithms. Our techniques are applied to the MLB database [6] that is composed of 14 646 objects including 60 teams and 2400 players.

The dataguide algorithm resembles the technique to transform a nondeterministic finite state automaton into a deterministic one. Here, the nondeterministic finite state automaton corresponds to source data graph and the deterministic one to an index graph. The technique finds all of the objects reachable by a regular path ex-

[5] The regular path expression which has a '|' operator can be divided into more than one regular path expressions without a '|' operator.

[6] This is constructed synthetically by programming techniques but it is similar to the real MLB database.

pression from the root efficiently. However, the technique is restricted to a single regular path expression. That is, it cannot be directly applied to complex queries with several regular expressions and variables. Moreover,in the worst case, the index size and the construction cost may be exponential.

In our techniques, the variables `node_info` and `merge_node_info` play an index role and are added in the OEM graph. The query processor prunes the search space by reading the variables during query processing. Our experiment presented in Table 1 shows that the NodeInfo and MergeNodeInfo technique have small index size and construction cost compared to the dataguide algorithm, although small dataguide graph is constructed in our MLB database.

Next, we consider the performance results of our techniques for simple regular path expressions. Table 2 shows queries used in the experiment. Table 3 shows the number of objects that satisfy the queries and the number of objects that are searched to evaluate the queries with four methods: naive, NodeInfo, MergeNodeInfo, and dataguide. The number of objects in the result set shows the minimum number of objects that the query processor should traverse to answer the queries.

Table 1
Construction cost and index size

	Construction cost (s)	Index size (byte)
DataGuide	116.397	63 132
NodeInfo	37.168	38 912
MergeNodeInfo	36.703	43 308

Table 2
Queries used in the experiment

Q1	MLB.*.Central.player.RBI
Q2	MLB.American.East.player.win
Q3	MLB.National.West.player.nickname
Q4	MLB.*.West.stadium

Table 3
The number of objects searched

	Q1	Q2	Q3	Q4
The number of objects in the result set	500	100	10	2
Naive	11 105	591	451	9807
NodeInfo	10 805	321	41	9789
MergeNodeInfo	1041	321	41	7
DataGuide	188	106	16	90

We can see that the NodeInfo and MergeNodeInfo technique reduce the search space significantly compared to naive evaluation of the queries. Since dataguide is an optimal solution for processing the single regular path queries without wildcards in average case, dataguide wins our technique for the queries Q_1, Q_2, and Q_3. However, dataguide has a critical problem that it cannot be applied to queries having multiple regular path expressions as mentioned earlier, whereas our technique can be applied to them. Additionally, when queries having '$*$' expression are processed, the MergeNodeInfo technique outperforms the NodeInfo and the dataguide techniques. This is because the entire dataguide graph should be traversed to process '$*$' expression. So, when the size of dataguide is large, the MergeNodeInfo technique outperforms the dataguide technique in processing a single regular path expression. Finally, since data queries are I/O bound, we measured the number of page I/O and the total execution time. The result is shown in Fig. 5.

If queries are mainly composed of labels that do not classify elements, our approach will show a rather poor performance. However, we think that this is a rare case and if we store the result of that kind of query as materialized views the weakness can be overcome. The issues on materialized views are our future work.

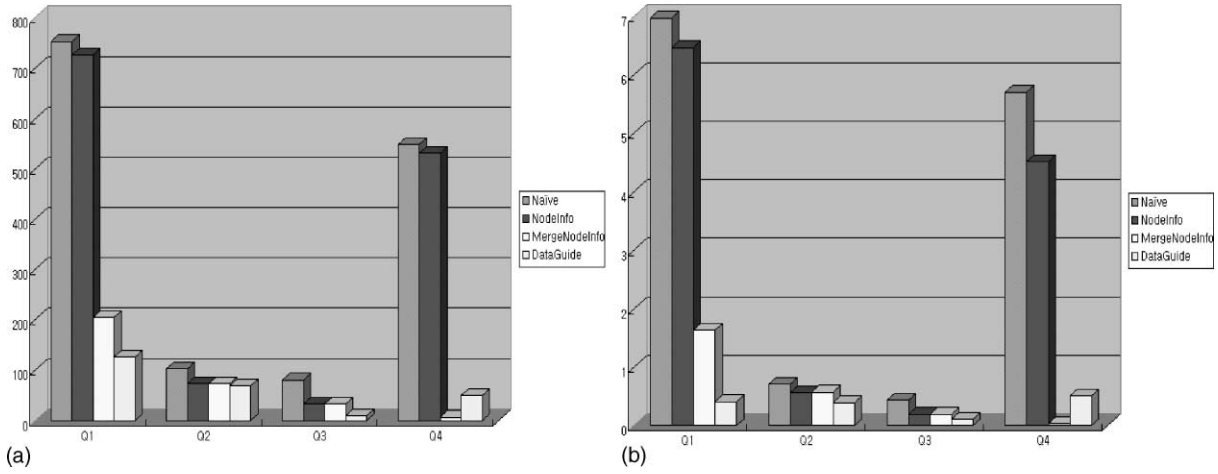


Fig. 5. Page I/O and execution time: (a) the number of page I/O and (b) the total elapsed time (s).

7. Conclusion

In this paper, we proposed two query optimization techniques named NodeInfo and MergeNodeInfo. From the DTD automata constructed from given relaxed regular expressions, the techniques captures information about the structures of data and about the classification of queries, and uses it in pruning the graph traversal. Our technique doesn't require much additional storage for indexes, and since the structure of the source database to which queries are processed is preserved, they can process complex queries such as those that have more than one regular expressions.

Appendix A. Proof of Theorem 1

We can prove Theorem 1 in similar ways to Theorem 2.3 of (Hopcroft and Ullman, 1979) by induction on the number of the operators in the regular expression r.

1. If there are no operators in the regular expression r, it corresponds to the form of $r = a$. The constructed automaton is just like Fig. 2. It clearly satisfies the conditions (induction basis).
2. We assume that the theorem is true for the regular expressions which have fewer than i operators (induction hypothesis).
3. When r has i operators, r is in the form of $r = r_1|r_2$ or $r = r_1r_2$ since r is a relaxed regular expression.

(a) *Case* $r = r_1|r_2$. As the number of operators in r_1 and r_2 are surely less than i, there are automata M_1 and M_2 with $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$ based on the induction hypothesis. In the transition function δ for case $r = r_1|r_2$ in Algorithm 1, according to equations 1, 3 all the moves of M_1 are present in M, and according to equations 2, 4 all the moves of M_2 are present in M. And for Q_1 and Q_2, as states can be renamed, we can assume $Q_1 \cap Q_2 = \phi$. The automaton M constructed from Algorithm 1 is shown in Fig. 6. In Fig. 6, any path from the initial state to the final states in M has two kinds of moves. One is that it starts from a state $[q_1, q_2]$ and ends in a state of F_1 by following any path in M_1, and the other is that it begins from the state $[q_1, q_2]$ and ends in a state of F_2 by following any path of M_2. This means that there is a path labeled x from the start state to the final states in M if and only if there is a path labeled x from the start state to the final states in M_1 or a path labeled x from the start state to the final states in M_2. Hence $L(M) = L(M_1) \cup L(M_2) = L(r_1) \cup L(r_2) = L(r)$ as required.

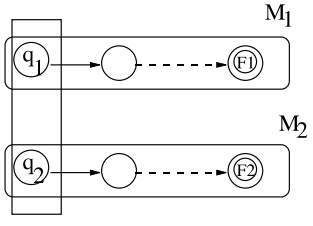


Fig. 6. $r = r_1|r_2$.

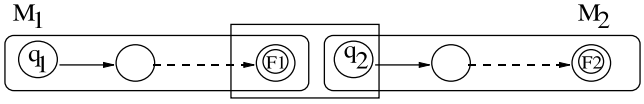


Fig. 7. $r = r_1r_2$.

(b) *Case* $r = r_1r_2$. Based on the induction hypothesis, as the number of operators in r_1 and r_2 are less than i, there are automata M_1 and M_2 with $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. Fig. 7 shows the automaton constructed. Every path from the start state to the final states in M is a path labeled by some string x which is a string from q_1 to $[f_k, q_2]$ $(1 \leqslant k \leqslant m)$, followed by a path labeled by some string y which is a string from $[f_k, q_2]$ to F_2. Namely, the language that automaton M accepts satisfies the following equation, $L(M) = \{xy | x \text{ is in } L(M_1) \text{ and } y \text{ is in } L(M_2)\}$. So, the equation $L(M) = L(M_1)L(M_2) = L(r_1)L(r_2) = L(r)$ is satisfied.

So, the Theorem 1 is true for any number of operators.

References

Abiteboul, S., 1997. Querying semi-structured data. In: Proceedings of the International Conference on Database Theory.

Abiteboul, S., Vianu, V., 1997. Regular path queries with constraints. In: Proceedings of ACM Symposium on Principles of Database Systems.

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

Bertino, E., Kim, W., 1989. Indexing techniques for queries on nested objects. IEEE Transaction on Knowledge and Data Engineering. 1 (2).

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

Buneman, P., 1997. Semistructured data. In: Proceedings of ACM Symposium on Principles of Database Systems.

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

Cattell, R., 1994. The object database standard: ODMG-93. Morgan Kaufmann Publishers.

Chamberlin, D., Florescu, D., Robie, J., Simeon, J., Stefanescu, M., 2001. XQuery: A query language for XML. Technical report, W3C Working Draft.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D., 1999. Query language for XML. In: Proceedings of Eighth International World Wide Web Conference.

Fallside, D.C., 2001. Xml schema part 0: Primer. Technical report, W3C Recommendation.

Fernandez, M., Suciu, D., 1998. Optimizing regular path expressions using graph schemas. In: IEEE International Conference on Data Engineering.

Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., Shim, K., 2000. XTRACT: A system for extracting document type descriptors from XML documents. In: Proceedings of the ACM SIGMOD International Conference on the Management of Data.

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

Hopcroft, J.E., Ullman, J.D., 1979. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company.

Kifer, M., Kim, W., Sagiv, Y., 1992. Querying object-oriented databases. In: Proceedings of the ACM SIGMOD International Conference on the Management of Data.

McHugh, J., Widom, J., 1999a. Optimizing branching path expressions. Technical report, Stanford University Database Group.

McHugh, J., Widom, J., 1999b. Query optimization for XML. In: Proceedings of the Conference on Very Large Data Bases.

Mendelzon, A., Wood, P., 1995. Finding regular simple paths in graph databases. SIAM Journal of Computing 24 (6).

Milo, T., Suciu, D., 1999. Index structures for path expressions. In: Proceedings of the International Conference on Database Theory.

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

Papakonstantinou, Y., Abiteboul, S., 1996. Object fusion in mediator systems. In: Proceedings of the Conference on Very Large Data Bases.

Papakonstantinou, Y., Velikhov, P., 1999. Enhancing semistructured data mediators with document type definitions. IEEE International Conference on Data Engineering.

Suciu, D., 1998. Semistructured data and XML. In: Proceedings of International Conference on Foundations of Data Organization.

Suciu, D., Fernandez, M., Davidson, S., Buneman, P., 1997. Adding structure to unstructured data. In: Proceedings of the International Conference on Database Theory.