Extracting Indexing Information from XML DTDs †

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, KOREA
{*tschung, hjk*}*@papaya.snu.ac.kr*

Keywords: databases, semistructured data, XML, query optimization

1 Introduction

Recently, 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, as XML is self-describing, we can issue many kinds of queries against XML documents in heterogeneous sources and get the necessary information.

As data in XML is an instance of a semistructured data model based on labeled-edge graph, it can be mapped to a semistructured data model and queries can be processed against it. Though the semistructured data model is flexible in data modeling, it requires a large search space in query processing since there is no schema fixed in advance. So, much work has addressed the problem of reducing the search space for evaluating semistructured queries[1, 2, 3, 4, 5, 6, 7]. Most of these techniques extract schema information from a semistructured data instance and use it as an index.

In this paper, we propose query optimization techniques for XML queries using DTDs. Our technique extract information from DTDs statically and provide a query processor with it in run time. Compared to the previous work, our technique can be applied to arbitrary queries and doesn't require much additional storage for indexes themselves.

2 Overview of Our Approach

2.1 Data Model

We assume that data in XML is mapped to an OEM(Object Exchange Model)[9] 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.

†This work was supported by the Brain Korea 21 Project.

```
<AGroup>
  <person id="&1" company="&2">
    <name> park </name>
    <e-mail> park@papaya </e-mail>
  </person>
  <company id="&2" person="&1">
   <name> cnn </name>
   <url> http://www.cnn.com </url>
  </company>
  <person id="&3" school="&4">
   <name> kim </name>
  </person>
  <school id="&4" person="&3">
   <name> snu </name>
   <baseball-team> lions </baseball-team>
  </school>
</AGroup>
```

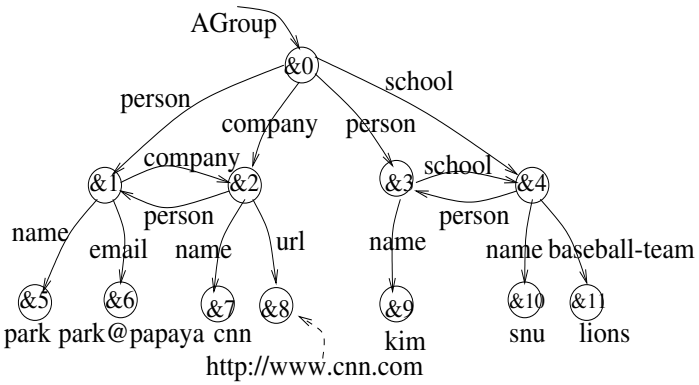


Figure 1: An Example of XML Data and OEM Graph

- Complex objects: The value of the complex objects is a set of <label, id> pairs.

Figure 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.

2.2 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 DTD declaration for the person element in Figure 1 is as follows.

```
<!ELEMENT person (name, e-mail*, (school|company))>                    (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 3), and query optimization techniques using this information(section 4).

3 Classification of DTD elements

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 [8], i.e., XML documents always have DTDs,

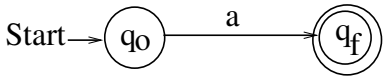


Figure 2: $r = a$

and do not have attributes other than the ID attribute, and so on. Let N be a set of element names, we abstract 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.

3.1 DTD automata

We construct DTD automata from each regular expression r for corresponding element n to classify elements in DTDs. When a regular expression has the form of r^+, this means that a particular attribute or a composition of attributes exists more than once. This kind of information is not necessary in reducing the search space because we should process all the attribute values when an attribute exists more than once. So, we define the following relaxed regular expression to extract only the necessary information during the query processing.

Definition 1 (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|⊥), (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 $n_i r_i^{'}$[1]. 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 deterministic finite automaton. On the other hand, in the traditional technique, first, an NFA with ϵ-transitions is constructed. For a given regular expression, Algorithm 1 makes an automaton for each label(in line 4) and merges automata for each operator(in line 7 and 10). So, the running time of Algorithm 1 is in $O(n-1)+O(n)$, where n is the number of operators.

Theorem 1 *There always exists an automaton $M_k (for\ 1 \leq k \leq n)$ constructed by Algorithm 1 for the input regular expression $n_k r_k^{'} (1 \leq k \leq n)$, and if $L(M_k)$ is the language accepted by M_k, and $L(n_k r_k^{'})$ is the language which is describable by the regular expression $n_k r_k^{'}$, then $L(M_k) = L(n_k r_k^{'})$.*

[1]In this paper, we occasionally omit concatenation operator, that is, $n_i r_i^{'}$=$n_i, r_i^{'}$.

Algorithm 1 The construction of DTD automata

1: **Input:** A relaxed regular expression $r = n_k r'_k (for\ 1 \leq k \leq n)$
2: **Output:** An automaton $M_k (for\ 1 \leq k \leq n)$
3: **if** $r = a\ (a \in \sum)$ **then**
4: Construct an automaton as shown in Figure 2;
5: **else if** $r = r_1 | r_2$ **then**
6: Construct the automata $M_1 = (Q_1, \sum_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \sum_2, \delta_2, q_2, F_2)$ by Algorithm 1 with input regular expressions r_1 and r_2 respectively;
7: Construct the new automaton $M = (Q_1 - \{q_1\} \cup Q_2 - \{q_2\}, \sum_1 \cup \sum_2, \delta, [q_1, q_2], F_1 \cup F_2)$ 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 $a \in \sum_1$,
2. $\delta(q, a) = \delta_2(q, a)$ for $q \in Q_2 - \{q_2\}$ and $a \in \sum_2$,
3. $\delta([q_1, q_2], a) = \delta_1(q_1, a)$ where $a \in \sum_1$,
4. $\delta([q_1, q_2], a) = \delta_2(q_2, a)$ where $a \in \sum_2$;

8: **else** { $r = r_1, r_2$ }
9: Construct the automata $M_1 = (Q_1, \sum_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \sum_2, \delta_2, q_2, F_2)$ by Algorithm 1 with input regular expressions r_1 and r_2 respectively;
10: Let the final states F_1 of M_1 be states $f_1, f_2, ..., f_m$ $(m \geq 1)$. Construct the new automaton $M = (Q_1 - F_1 \cup Q_2 - \{q_2\} \cup \{[f_1, q_2], [f_2, q_2], ..., [f_m, q_2]\}, \sum_1 \cup \sum_2, \delta, q_1, F_2)$ from the automata M_1 and M_2, where δ is defined by

1. $\delta(q, a) = \delta_1(q, a)$ for $q \in Q_1 - F_1$, $\delta_1(q, a) \neq f_k\ (1 \leq k \leq m)$, and $a \in \sum_1$,
2. $\delta(q, a) = \delta_2(q, a)$ for $q \in Q_2 - q_2$ and $a \in \sum_2$,
3. $\delta([f_k, q_2], a) = \delta_2(q_2, a)$ for all k$(k = 1, 2, ..., m\ )$ and $a \in \sum_2$,
4. $\delta(q_f, a) = [f_k, q_2]$ for all q_f which satisfies $\delta_1(q_f, a) = f_k (1 \leq k \leq m)$ and $a \in \sum_1$;

11: **end if**

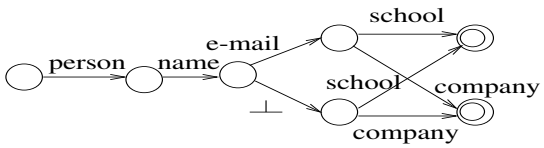
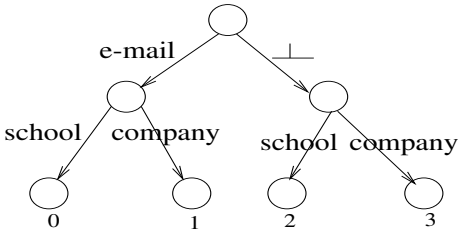


Figure 3: A DTD automaton



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

Figure 4: A classification tree and a classification table

We omit the proof for lack of space.

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

3.2 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. Algorithm 2 shows the construction of classification trees having labels at the diverging points of the DTD automata. 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}.

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

4 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

Algorithm 2 The construction of classification trees from DTD automata

```
1: Input: M_k = (Q_k, Σ_k, δ_k, q_k, F_k) (for k = 1, 2, ..., n)
2: Output: The classification tree T_k (for k = 1, 2, ..., n)
3: procedure Make_classification_tree(state q, automaton M_k)
4: if q ∈ F_k then
5:    make a vertex q' corresponding to a state q;
6:    return q';
7: else
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 ∈
         transition(q) with edges labeled a in the transition;
11:      return T;
12:   else
13:      q ← δ_k(q, a);
14:      No_effect_label_k = No_effect_label_k ∪ a;
15:      Make_classification_tree(q,M_k);
16:   end if
17: end if
```

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 2 *(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, $P = label|(P|P)|(P.P)|P*$.*

Definition 3 *(Simple Regular Path Expression) A simple regular path expression is a sequence $H.p_1.p_2.....p_n$ where*

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

4.1 NodeInfo

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 Figure 4, and the object &1 in Figure 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[5, 6] of exponential cost in the worst case. Moreover, in average case, it is usually superior to 1-index[7] 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 4 *Let $H.p_1.p_2.....p_n$ be a simple regular path expression. For each $p_i (i = 1, 2, ..., n-1)$, the value of p_i's classification_info is*

1. $\{\}$ *where* $p_{i+1} = *$, *or* $p_{i+1} \neq *$ *and* $p_{i+1} \in No\_effect\_label$[2], *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 $\{e\text{-}mail\}$.*

Query processing in the NodeInfo technique is performed as follows. To find all objects reachable by a simple regular path expression $H.p_1.p_2.....p_n$ 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 $p_i (1 \leq i \leq n-1)$, 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.

Example 5 *Consider the simple regular path expression 'AGroup.person.company.url' against the data graph in Figure 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 $classification\_info\ of\ p_i - classification\_table[person][w.node\_info] = \{company\} - \{school\} \neq \phi$. So it doesn't traverse the object &3.*

4.2 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 5 *Let $H.p_1.p_2.....p_n$ be a simple regular path expression that exists in a query. The merge_classification_info of $p_i (i = 1, 2, ..., n-1)$ is defined by*

[2] No_effect_label is an union set of $No\_effect\_label_k$ for all k.

$\cup_{k=i}^{n-1}$ *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 $H.p_1.p_2.....p_n$, 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 $p_i (1 \leq i \leq n-1)$, the search should be expanded to object w when the condition $merge\_classification\_info\ of\ p_i - w.merge\_node\_info = \phi$ 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*$\} \neq \phi$. *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[3]. In this case, query processing is carried out as follows. First, we define the following variables.

Definition 6 *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 all the sub-objects of it have no figure. 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,* $P - (Q \cup R) = \{figure\} \neq \phi$, *after which it stops searching.*

4.3 An experiment

We have implemented our techniques described in this paper with about 3000 lines of Java code to illustrate their enhancement in query processing. Our techniques are applied to a MLB database[4] that is composed of 14646 objects including 60 teams and 2400 players. Figure 5 shows queries used in the experiment and the number of

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

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

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

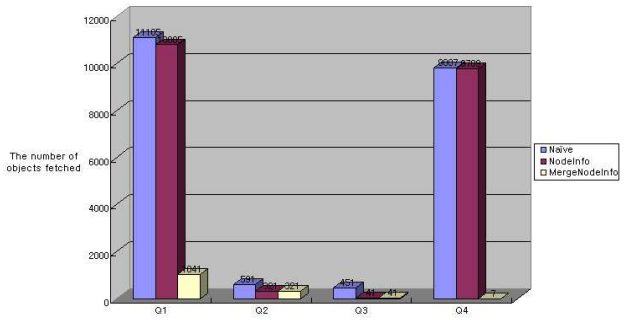


Figure 5: Queries and the number of objects searched

objects searched to evaluate the queries with three methods. We can see that the NodeInfo and MergeNodeInfo technique reduce the search space significantly. Additionally, when queries having '$*$' expression are processed, the MergeNodeInfo technique outperforms the NodeInfo technique.

5 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.

References

[1] Dan Suciu, Mary Fernandez, Susan Davidson, and Peter Buneman. Adding structure to unstructured data . In *Proceedings of the International Conference on Database Theory*, 1997.

[2] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proceedings of the Conference on Very Large Data Bases*, 1999.

[3] Mary Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of International Conference on Data Engineering*, 1998.

[4] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1996.

[5] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of the Conference on Very Large Data Bases*, 1997.

[6] Svetlozar Nestorov, Jeffrey Ullman, Janet Wiener, and Sudarshan Chawathe. Representative objects: concise representations of semistructured, hierarchical data. In *Proceedings of International Conference on Data Engineering*, 1997.

[7] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, 1999.

[8] Yannis Papakonstantinou and Pavel Velikhov. Enhancing semistructured data mediators with document type definitions. In *Proceedings of International Conference on Data Engineering*, 1999.

[9] Yannis Papakonstantinou and Serge Abiteboul. Object fusion in mediator systems. In *Proceedings of the Conference on Very Large Data Bases*, 1996.