A two phase optimization technique for XML queries with multiple regular path expressions

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 21 May 2001; received in revised form 21 September 2001; accepted 12 November 2001

Abstract

As XML (eXtensible Markup Language) has emerged as a 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. XML queries are based on regular path queries, which find objects reachable by given regular expressions. To answer many kinds of user queries, it is necessary to evaluate queries that have multiple regular path expressions. However, previous work on subjects such as query rewriting and query optimization in the framework of semistructured data has usually dealt with a single regular path expression. For queries that have multiple regular path expressions we suggest a two phase optimizing technique: query rewriting using views by finding the mappings from the view’s body to the query’s body and for rewritten queries, evaluating each query conjunct and combining them. We show that our rewriting algorithm is sound and our query evaluation technique is more efficient than that of previous work on optimizing semistructured queries.

2002 Elsevier Science Inc. All rights reserved.

1. Introduction

Recently, as XML (Bray et al., 1998) has emerged as a 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. As XML data is self-describing we can issue queries over XML documents distributed in heterogeneous sources and get the necessary information.

Since XML data is an instance of a semistructured data model, semistructured query languages can be used to process it. Here, query languages such as XQuery (Chamberlin et al., 2001), Lorel (Abiteboul et al., 1996), XML-QL (Deutsch et al., 1999), etc. are based on regular path expressions. For example, the regular path expression \((\. *movie)\. (\. *actor \{Tom Cruise|Brad Pitt\})\) denotes all of the paths that first have the edge ‘movie’ at some point, next, arbitrary edges and then the edge ‘actor’, and finally arbitrary edges followed by the ‘Tom Cruise’ or ‘Brad Pitt’ edge.

Many researchers have addressed the problem of processing the single regular path query that is defined as follows: 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 \(r\).

However, user’s queries are usually composed of several regular path expressions. For example, one can issue the following query that asks for movies which have actors ‘Tom Cruise’ or ‘Brad Pitt’ and were produced in 2000. (The clear semantics of the query language is defined in Section 3.)

\[q(p_2): -p_1(\. *movie)p_2, \quad p_2(\. *year:2000)p_3, \quad p_2(\. *actor\. *\. (Tom Cruise)|Brad Pitt))p_4,\]

where \(p_i\) are variables binding to nodes in the semistructured database, and the variables are connected by regular path expressions. Here, the usual semantics holds, i.e. \(p,rp_1\) is the set of all nodes \((u,v)\) such that there is a path satisfying \(r\) from \(u\) to \(v\).

This kind of query that has more than one regular expression is frequently used in XML queries. However, previous work such as query rewriting using views (Grahne and Thomo, 2000), rewriting regular path queries (Calvanese et al., 1999) and query optimization (Goldman and Widom, 1997) have dealt with the single
regular path query. So, if we apply the proposed techniques in processing queries having multiple regular path expressions, they cannot achieve greater efficiency.

We propose a two-phase algorithm for processing semistructured queries having multiple regular path expressions. Our technique has the following contributions.

- We propose a unified two-phase optimization solution to process queries having multiple regular path expressions.
- In the query rewriting phase, we show that our mapping algorithm is sound and more efficient than the previous work.
- We show that our query evaluation technique is more efficient than that of the previous work in the query processing phase.

The paper is organized as follows. Section 2 mentions related work, and Section 3 defines the data model and the query language. Section 4 describes our query rewriting algorithm using views. In Section 5, we propose our query optimization technique, and finally, we conclude in Section 6.

2. Related work

Many researchers have discussed the problems of materialized views that we used in step 1 of our algorithm in the relational model (Chaudhuri et al., 1995; Levy et al., 1995; Larson and Yang, 1985) and the object model (Abiteboul and Bonner, 1991; Rundensteiner, 1992). That is, issues including the definition of views and incremental maintenance of views, query rewriting and complexity problems have been studied. In particular, in Chaudhuri et al. (1995), the authors propose an optimization algorithm that generalizes the traditional join enumeration algorithm (Selinger et al., 1979). It adds new query evaluation plans that use materialized views to the plan space and finds the optimal plan in the whole plan space. This paper addresses the same problem in the context of semistructured data.

In the semistructured data model, there has been much work rewriting queries using views. Authors in Papakonstantinou and Vassalos (1999) presented an algorithm of query rewriting for TSL, a semistructured query language and showed the soundness and completeness of it. TSL has restructuring capabilities and is composed of multiple path expressions. However, it does not support regular path expressions, whereas our work supports regular path expressions but does not support restructuring capabilities. The complexity problem of query containment in \( \text{STRUQL} \) is described in Levy and Suciu (1998). In Calvanese et al. (1999) and Grahne and Thomo (2000), the problem of finding queries \( q' \) that access the views, given a semistructured query \( q \) and a set of semistructured views \( V = \{v_1, \ldots, v_n\} \), is studied. However, they dealt with queries and views having a single regular path expression.

The second phase of our algorithm is related to the problem of query optimization for semistructured data. 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 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. These techniques have characteristics whereby they define graph schemas statically and process 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 of the paths in a database dynamically, and use this 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 only to queries with a single regular expression. That is, it cannot be directly applied to complex queries with several regular expressions and variables.

Three kinds of index structures are proposed in Milo and Suciu (1999). That is, 1-indexes, 2-indexes, and T-indexes. 1-indexes, like DataGuides, find all of the objects reachable by a regular path expression from the root node. However, unlike DataGuides, if we view a data graph as an automaton, a 1-index is a non-deterministic automaton. So, the storage for the 1-index is at most linear. The 2-index is an index structure for efficiently finding all pairs of nodes that are connected by a regular path expression. We use it as a base scheme for query evaluation. T-indexes provide index structures to evaluate multiple regular path expressions. However, the target query should conform to the corresponding template.

The cost based query 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 to its plan space as another efficient plan.

3. Data model and query language

Data in XML can be mapped to a semistructured data model, that is, a rooted labeled graph. XML ele-
ments are represented by nodes of the 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 edges which are connected to leaves in the graph. Fig. 1 shows an example of an XML data and the corresponding data graph. Formally, we define a data graph $DB$ as follows. First, we assume an infinite set $O$ of oids, which is disjoint from an infinite set $C$ of constants.

**Definition 1.** A data graph $DB = (V, E, R)$ is a labeled rooted graph, where $V \subseteq O$ 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.

For a given data graph $DB$, a query is based on the following regular path expression.

**Definition 2.** A regular path expression is defined recursively by the grammar $r = e | a \bigl[\bigl[|\langle r_1 \rangle\bigl|\langle r_2 \rangle\bigl| \langle r_1 \rangle^* \bigr\rangle\bigr| \langle r_1 \rangle^* \bigr\rangle$, where $r, r_1$ and $r_2$ are regular path expressions, $a \in C$ is a label constant, $\bigl|$ denotes any label, and $e$ denotes an empty string.

Many semistructured query languages based on the regular path expression have been proposed. For example, XQuery (Chamberlin et al., 2001); Quil (Chamberlin et al., 2000); UnQL (Buneman et al., 1996); Lorel (Abiteboul et al., 1996); XML-QL (Deutsch et al., 1999), and so on. We use a Datalog-like notation as follows. However, our technique can be applied to other query languages based on the regular path expression.

**Definition 3.** A query is an expression of the form $q(x) : \neg \bigwedge \bigwedge_{0 \leq i \leq n-1} t_i r_i z_i = \bigwedge \bigwedge_{0 \leq i \leq n-1} t_i r_i z_i$, where $v_a r(q) = \{ y_0, z_0, y_1, z_1, \ldots, z_{n-1} \}$ are the query’s node variables (they need not be distinct), $x \subseteq v_a r(q)$ are the query’s head variables, and the $r_i, 0 \leq i \leq n-1$, are regular path expressions. Here, the query has the following properties of branching regular path expressions. For each query conjunct $y_i r_i z_i (0 \leq i \leq n-1)$, let $y_i$ be the source variable and $z_i$ be the destination variable.

1. Each source variable except the first one appears as a destination variable in an earlier step.
2. A variable may appear as a source variable in more than one step.
3. A variable may not appear as a destination variable in more than one step.
Again, for each query conjunct \( y_r z_i \), letting \( \delta \) be a function which maps node variables to \( O \), i.e., \( \delta(U) = o \) where \( U \in \text{near}(q) \) and \( o \in O \), there is a path which satisfies the regular path expression \( r \), between \( \delta(y_r) \) and \( \delta(z_i) \) in the data graph \( DB \). Each substitution \( \delta \) defines a tuple in a relation \( R_v \), whose attributes are variables in \( q \). The result of the query \( q \) is the projection of \( R_q \) on the variables in \( x \).

**Example 1.** When the query \( q(b) : \neg a(\text{movie}) \land b(\text{actor.name} \land \text{"Brad Pitt"}) \land c \) which asks for movies having the actor “Brad Pitt” is applied to the database in Fig. 1, the relation \( R_q(a, b, c) = \{(1, 2, 26), (1, 3, 26)\} \). So the result of \( q \) is \( \pi_b(R_q) = \{2, 3\} \).

Like queries views are defined as follows.

**Definition 4.** A view is an expression of the form \( v : y_1p_1z_0, \ldots, y_{n-1}r_{n-1}z_{n-1} \). Unlike the query definition, the head variables are not specified and the regular expressions may have variables. Let \( \delta \) be a function which maps node variables to \( O \) and variables in regular path expressions to label constant. Each substitution \( \delta \) defines a tuple in a relation \( R_v \), whose attributes are variables in \( v \). The result of the view \( v \) is the relation \( R_v \).

**Example 2.** Consider the following view:

\[
v : \neg p_1(\text{movie}) \land p_2(\text{year} \land L) \land p_3(\text{actor.name} \land \text{"Brad Pitt"}) \land p_4.
\]  

When the view \( v \) is applied to the database in Fig. 1, the result of the view is as follows.

<table>
<thead>
<tr>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
<th>( L )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>15</td>
<td>26</td>
<td>1994</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>19</td>
<td>26</td>
<td>1995</td>
</tr>
</tbody>
</table>

4. Query rewriting

4.1. Problem definition

We address the following query rewriting problem: given the following form of query \( q \), and a set of views \( v \), finding the query \( q' \) which accesses at least one view of \( v \) and returns the same result as \( q \).

\[
q(u) : \neg p_1r_1p_1 \land p_1r_2p_2 \land p_1r_3p_3 \land p_1r_4p_4 \land p_1r_5p_5 \land \neg q_0r_6q_1 \land q_0r_6q_2
\]  

(2)

That is, given a view \( v \) and a query \( q \), let \( c_1^1, \ldots, c_m^1 \) be the conjuncts in the body of the view, and \( c_1^2, \ldots, c_5^2 \) be conjuncts in the body of the query. For example, the query conjuncts \( \langle p_1r_1p_1 \rangle \) and \( \langle p_1r_2p_2 \rangle \) in Formula (2) correspond to \( c_1^1 \) and \( c_1^2 \), respectively. The problem is finding a set of symbol mappings \( \Pi \) which, for \( \pi \in \Pi \), satisfies \( \pi(c_1^j) = c_1^j(1 \leq j \leq n) \) for all \( c_1^i(1 \leq i \leq m) \), and then obtaining \( v' \) by applying \( \pi \) to \( v \), and finally constructing the rewritten query \( q' \) with \( v' \).

If a query \( q \) and a view \( v \) are composed of \( n \) and \( m \) query conjuncts respectively, there are \( n^m \) possible mappings. Actually, the mapping algorithm of Papakonstantinou and Vassalos (1999) considers all \( n^m \) mappings. Additionally, in order to show that the equation \( \pi(c_1^j) = c_1^j(1 \leq j \leq n) \) is satisfied, we should check whether or not \( L(r_1^j) = L(r_2^j) \), where \( r_1^j \) and \( r_2^j \) are regular expressions in \( c_1^j \) and \( c_2^j \), respectively, and \( L(r) \) represents the language denoted by \( r \). This is PSPACE-complete (Stockmeyer and Meyer, 1973). We propose an algorithm that improves this potential complexity.

4.2. Our solution

4.2.1. Symbol mapping

Our mapping algorithm composed of two steps. In step 1, it finds candidate mappings and tests the correctness of candidate mappings in step 2.

**Step 1:** Finding candidate mappings: First, given a query \( q \) and a view \( v \) we define a query graph \( G_q \) and a view graph \( G_v \) as follows. Let \( D \) be a set of symbols that denote regular expressions.

**Definition 5.** A query graph \( G_q = (V, E, R) \) is a labeled rooted graph, where \( V \in \text{near}(q) \) is a set of nodes, \( E \subseteq V \times D \times V \) is a set of directed edges, and there exists an edge \( a \rightarrow b \) for each query conjunct \( a \) \( r \) \( b \) where \( d \in D \) denotes the regular expression \( r \). \( R \in V \) is a root node. The view graph \( G_v \) is defined in the same way.

**Example 3.** Fig. 2 shows the graph \( G_q \) and \( G_v \) for Formula (2).

As the query \( q \) and the view \( v \) have the properties of branching regular path expressions, the graph \( G_q \) and \( G_v \) are trees without cycles. From this property, we can reduce the number of candidate mappings. For example, there are \( 5^2 \) mappings from the view’s body to the query’s body in Formula (2). However, Fig. 2 shows that there are only four possible mappings.\(^2\)

Our mapping algorithm finds candidate mappings from the input graphs \( G_q \) and \( G_v \) as follows. First, it finds a node in \( G_q \) which can be mapped to the root node of \( G_v \) using a breadth first search, and for each such a mapping, finds the mappings of the child node recursively. For a node \( v \) in \( G_v \) and a node \( w \) in \( G_q \), if the number of \( v \)’s child nodes and \( w \)’s child nodes are \( n \) and

\(^{2}\)Fig. 2 shows only two mappings but there are four possible mappings as we can change the order of child mappings.
As it is expensive to check whether or not unifications (Luger and Stubblefield, 1993) are needed, views can have variables, variable substitutions, that is, expressions. Moreover, as regular path expressions in languages described by the corresponding regular expressions and views, so we should check the equality of mappings constructed from structural information of the four candidate mappings, that is, \[ \text{find-candidate-mappings}(G_q, G_v) \]

The candidate mappings of step 1 are only symbol mappings constructed from structural information of queries and views, so we should check the equality of mappings with root mappings. In this way, the number of candidate mappings is reduced significantly. Algorithm 1 shows our mapping algorithm.

Example 4. For the graph in Fig. 2, Algorithm 1 first finds the nodes \( p_1 \) and \( p_3 \) that can be mapped to the root node \( q_0 \). Next, it finds submappings recursively. So the four four candidate mappings, that is, \{\{q_0 \rightarrow p_1, q_2 \rightarrow p_3\}, \{q_0 \rightarrow p_3, q_2 \rightarrow p_3\}, \{q_0 \rightarrow p_3, q_1 \rightarrow p_1, q_2 \rightarrow p_3\}, \{q_0 \rightarrow p_3, q_1 \rightarrow p_1, q_2 \rightarrow p_3\}\} are found.

Step 2: Testing Correctness of Candidate Mappings

The candidate mappings of step 1 are only symbol mappings constructed from structural information of queries and views, so we should check the equality of languages described by the corresponding regular expressions. Moreover, as regular path expressions in views can have variables, variable substitutions, that is, unifications (Luger and Stubblefield, 1993) are needed. As it is expensive to check whether or not \( L(r_1) = L(r_2) \) we filter the mappings again in step 2A.

Algorithm 1. Finding Candidate Mappings

1: **INPUT:** Graphs \( G_q \) and \( G_v \)
2: **OUTPUT:** Candidate mappings
3: **Procedure** find-candidate-mappings(\( G_q, G_v \))
4: initialize a queue \( Q \) to be empty;
5: \( rq \leftarrow \) root of \( G_q \);
6: \( rv \leftarrow \) root of \( G_v \);
7: visit \( rq \) with \( rv \) and mark \( rq \); insert \( rq \) in \( Q \);
8: while \( Q \) is nonempty do
9: \( x \leftarrow Q . \)getFront();
10: for each unmarked vertex \( w \) adjacent to \( x \) do
11: if \( w \)’s depth < \( rv \)’s depth then
12: continue;
13: else if number of \( w \)’s children < number of \( rv \)’s children then
14: mark \( w \) and insert \( w \) into \( Q \);
15: else
16: mark \( w \) and insert \( w \) in \( Q \);
17: build new mapping \( c \) fixed with root mapping;
18: find-sub-mappings(\( w, rv, c \))
19: end if
20: end for
21: end while
22: 
23: **INPUT:** vertex \( q, \) vertex \( v, \) and mapping \( c \)
24: **OUTPUT:** Candidate mappings
25: **Procedure** find-sub-mappings(\( q, v, c \))
26: if \( v \) has no child then
27: return;
28: end if
29: \( nq \leftarrow \) number of \( q \)’s children;
30: \( nv \leftarrow \) number of \( v \)’s children;
31: for each Permutation(\( nq, nv \)) child mappings do
32: if mapping condition is hold then
33: build new mapping table \( d \) initialized to \( c \);
34: add current mapping to \( d \);
35: if \( d \) is filled then
36: insert \( d \) into candidate mappings;
37: else
38: for each child mapping (\( q_c, v_c \)) pair
39: find-sub-mappings(\( q_c, v_c, d \));
40: end for
41: end if
42: end if
43: end for

Step 2A: Filtering candidate mappings: From the graphs \( G_q \) and \( G_v \) we define the graphs \( G_{2q} \) and \( G_{2v} \) as follows.

Definition 6. A graph \( G_{2q} \) is constructed from a graph \( G_q \) by replacing each edge labeled \( r_i \) by an automaton \( A_i \) such that \( L(A_i) = L(re(r_i)) \). Without loss of generality, we assume that \( A_i \) has a unique start state and a final state, which are identified with the source and target of the edge, respectively.

Definition 7. A graph \( G_{2v} \) is constructed from a graph \( G_v \) by replacing each edge labeled by \( r_i \) by an arbitrary expression in \( L(re(r_i)) \).

---

\( L(re) \) refers to the regular expression denoted by \( r \).
Example 5. Fig. 3 shows graphs G2_y and G2_c from the graphs G_y and G_c when r_0 = (a, b), r_1 = c(d(e)), r_2 = c(d(e)*), r_3 = g(h), r_4 = g, r_5 = Ld|Le, and r_6 = (Ld) * L. Here, the uppercase letter L is a label variable.

Algorithm 2 shows how to find filtered candidate mappings.

Example 6. For the graphs in Fig. 3 and candidate mappings constructed in Example 4, Algorithm 2 finds one filtered candidate mapping: \{((q_0 \rightarrow p_1, q_1 \rightarrow p_2, q_2 \rightarrow p_3), c/L)\}. Here, the notation c/L indicates that c is substituted for the variable L.

**Algorithm 2. Finding Filtered Candidate Mappings**

1: **INPUT:** Graphs G2_y, G2_c, and the candidate mappings c
2: **OUTPUT:** Filtered candidate mappings c
3: initialize filtered candidate mappings to c;
4: for each candidate mapping in c do
5:    for each edge e in G2_y do
6:        Let e' be an expression obtained by applying unification to e;
7:        if e’s label is not accepted by the corresponding automaton in G2_y then
8:            drop the mapping from the filtered candidate mappings;
9:        break;
10:    end if
11: end for

**Step 2B:** Finding final mappings: The filtered candidate mappings obtained in step 2A are a necessary condition to the final mappings. So, we get the final mappings by inserting an existing L(r_1) = L(r_2) checking algorithm to Algorithm 2. As in step 2A, if variables exist in r_2, variable substitutions are required.

Example 7. For the query and the view in example 5, the final mapping is \{((q_0 \rightarrow p_1, q_1 \rightarrow p_2, q_2 \rightarrow p_3), c/L)\}.

If there are multiple views, the part of the query graph which is mapped to one view is condensed to a single node and our mapping algorithm can be applied again. In this case, the condensed node cannot be mapped to any nodes in the view graph.

4.2.2. Query rewriting

Let \( \Pi \) be the final mappings in the previous section. For a given view \( v : -p(u, w) \) and a query \( q(u) : -p'(x, y) \), if we apply a mapping \( \pi \in \Pi \) to the view, that is, apply variable mappings and variable substitutions, we obtain \( \pi': -p(x, y) \). Here, \( p \equiv p' \) by our mapping algorithm. Finally, we rewrite the query \( q \) and obtain \( q'(u) : -\pi', s(y, z) \). For example, if we apply the mapping \{((p_1 \rightarrow q_1, p_2 \rightarrow q_2, p_3 \rightarrow q_3, p_4 \rightarrow q_4), 1994/L)\} to the view \( v \) in Formula (1), the view \( v' \) is obtained as follows.

<table>
<thead>
<tr>
<th>q_1</th>
<th>q_2</th>
<th>q_3</th>
<th>q_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>15</td>
<td>26</td>
</tr>
</tbody>
</table>

The following theorem shows the soundness of our rewriting algorithm.

**Theorem 1.** The query result of \( q' \) which is obtained by the rewriting algorithm is the same as that of the given query \( q \).

**Proof.** We regard each query conjunct as a predicate and assume that \( p'(x, y) = q'_1(x_1', \ldots, x_i'), \ldots, q'_n(y_1', \ldots, y_j') \), \( s(y, z) = r_1(v_1, \ldots, v_k), \ldots, r_n(w_1, \ldots, w_l) \) in our query rewriting algorithm, and let \( Q_1, \ldots, Q_m \) be relations corresponding to the predicates \( q'_1, \ldots, q'_n, \) and \( R_1, \ldots, R_n \) to \( r_1, \ldots, r_n \). Assume that, in \( \pi' \), \( p(u, w) = q_1(x_1, \ldots, x_i), \ldots, q_m(y_1, \ldots, y_j) \) and let \( Q_1, \ldots, Q_m \) be corresponding relations. Then, the result of the query \( q(u) : -p'(x, y), s(y, z) \) is \( \pi_n((Q'_1 \bowtie \ldots \bowtie Q'_n) \bowtie (R_1 \bowtie \ldots \bowtie R_n)) \). On the other hand, the result of the query \( q'(u) : -\pi', s(y, z) \) is \( \pi_n((Q'_1 \bowtie \ldots \bowtie Q'_n) \bowtie (R_1 \bowtie \ldots \bowtie R_n)) \). Here, as \( p \equiv p' \) by our mapping algorithm, \( Q_1 \bowtie \ldots \bowtie Q_m = Q'_1 \bowtie \ldots \bowtie Q'_m \). Thus, the result of \( q' \) is the same as that of \( q \). □

5. Query optimization

The evaluation of the following query that has two regular expressions connected linearly becomes a basis for evaluating queries with multiple regular expressions.

\( q(u) : -p_0p_0p_1, p_1p_1p_2 \) (3)

If we regard a regular expression as a path label, we can consider three kinds of query evaluation methods as follows (McHugh and Widom, 1999b).
1. Forward scan: From the objects bounded to the variable $p_0$, finds all objects that are reachable by the regular expression $r_0$, and binds them to the variable $p_1$. Next, from the objects bound to $p_1$, finds all objects reachable by the regular expression $r_1$, and binds them to the variable $p_2$.

2. Backward scan: It is opposite to the forward scan and can be implemented by value indexes (McHugh and Widom, 1999b) that find all objects with a given incoming edge label and satisfying a given predicate.

3. Hybrid: It is a combination of the forward scan and the backward scan.

However, these methods are expensive because they are based on the source database, and as our rewritten queries may have views, they cannot be adapted directly. So, we propose another technique named $2^*$-index-join.

The key idea of $2^*$-index-join is that each query conjunct is evaluated independently by an index technique or by using materialized views, and then small cost joins are performed to them.

5.1. Review: 2-index

The 2-index (Milo and Suciu, 1999) is an index structure for answering queries of the form \( \text{select } x_1, x_2 \text{ from } * \times_1 \mathcal{R} x_2 \), where $\mathcal{R}$ is a regular path expression. First, $L_{e,u}(DB)$ is defined as follows.

**Definition 8.** $L_{e,u}(DB) = \{w | w = a_1 \ldots a_n \text{, and there exists a path } v \rightarrow \ldots \rightarrow u \text{ in } DB\}$

Then, two pairs are defined to be equivalent, as follows.

**Definition 9.** $(v, u) \equiv (v', u') \iff L_{e,u} = L_{e,u'}$

For example, in Fig. 4, the node pairs $(1, 5) \equiv (1, 7)$ since $L_{(1,5)} = L_{(1,7)} = a.c$. From this language equivalent relation, we can find pairs of objects connected by a regular path expression efficiently. Although computing $\equiv$ is expensive, there is a refinement based on simulation and bisimulation, $\approx$, satisfying:

$(v, u) \approx (v', u') \Rightarrow (v, u) \equiv (v', u')$

The 2-index $I^2(DB)$ of DB is defined as follows. Let $[(v, u)]$ denote the equivalent class of $(v, u)$.

**Definition 10.** The 2-index $I^2(DB)$ of DB is the following rooted graph. Its nodes are equivalence classes, $[(v, u)]$, of $\approx$; the roots are all equivalence classes of the form $[(x, x)]$; finally, there is an edge $s \rightarrow u'$ iff there exist $v, u, u'$ s.t. $(v, u) \in s, (v, u') \in s'$ and DB contains an edge $u \rightarrow u'$.

Here, as $L_{e,u}(DB) = L_{e,u}(I^2(DB))$, we compute the query $Ry$ on $I^2$ and take the union of the extents to evaluate select $x, y$ from $* xRy$.

**Example 8.** Fig. 4 shows a source database and the corresponding $I^2$ graph with extents. The query select $x, y$ from $* x(a.d)y$ is evaluated by traversing the path $a.d$ from the root and the extent of the node $s_8 = \{(1, 6), (1, 8)\}$ is returned.

5.2. 2$^*$-index

The 2-index technique finds all pairs of objects connected by a regular path expression with an $I^2$ graph that usually has a small size compared to the source data graph. However, when traversing the $I^2$ graph, if a query has the `$*$` expression, the query processor should traverse the entire $I^2$ graph.

The 2$^*$-index technique removes the `$*$` expression by considering all possible instantiations against an XML document type definition (DTD) (Bray et al., 1998). DTDs describe the XML document’s structure by means of regular expressions. Although the DTD is an optional feature of XML, the DTD can be inferred from XML data by the technique proposed in (Garofalakis et al., 2000). Additionally, our technique can be aided when XML Schemas (Brown et al., 2001) that are extensions to DTDs are available.

We assume that DTDs do not have the entity and notation declaration and mixed contents elements whose contents mixes strings with elements. Then, we can abstract a DTD as a set of $(k, n, p)$ triples, where $k$ describes the kind of DTD declaration: an element (e) or an attribute (a), and let $L$ be a set of element names, $n \in L$, and finally, when $k = e$, $p$ is either a regular expression over $L$ or PCDATA which denotes a character.

---

Fig. 4. Example: 2-index.

---

\[ \begin{array}{c}
\text{Fig. 4} \quad \text{Example: 2-index.} \\
\end{array} \]

---

\[ \begin{array}{c}
\text{Fig. 4} \quad \text{Example: 2-index.} \\
\end{array} \]
string, and when $k = a$, $p$ is the name of an attribute. We define a DTD graph $G_D$ as follows. It is similar to that of Shanmugasundaram et al. (1999) and our $G_D$ has no operators in the DTD. Let $N$ be a set of elements and attributes names.

Definition 11. A DTD graph $G_D = (V, E)$ is a graph where $V \subseteq N$ is a set of nodes and $E \subseteq V \times V$ is a set of edges. For the set of nodes $N$, each element appears exactly once, while attributes appear as many times as they appear in the DTD. For each DTD declaration $(k, n, p)$, when $k = e$, there is an edge $n \rightarrow m$ from an element $n$ to each element in $p$, and when $k = a$ there is an edge $n \rightarrow a$ from the node $n$ to the attribute name $a$.

Example 9. Fig. 5 shows a DTD graph $G_D$ when $DTD = \{(e, person, (name, position?, company)), (e, company, (address, person+)), (e, name, (firstname?, lastname)), (e, position, PCDATA), (e, address, PCDATA), (e, firstname, PCDATA), (a, person, age), (a, company, name).\}

Given a graph $G_D$, and a query having the `.r` expression, Algorithm 3 produces new queries without the `.r` expression. When $R_1 = \phi$, it finds all possible paths from each element name in the DTD. Otherwise, it finds all possible paths from the tail of $R_1$ to the head of $R_2$. When the $G_D$ has cycles, the algorithm finds the shortest possible path. In line 15, Algorithm 3 traverses the marked vertices that do not make a cycle with the current path for the case of diamond hierarchy in the DTD graph.

Algorithm 3. Star Flattening
1: **Input:** A graph $G_D$ and a query $R = R_1...R_2$
2: **Output:** A set of queries without the `.r` expression
3: **Procedure** Star-Flattening($G_D$, $q$)
4: if $R_1 = \phi$ then
5: Star-Flattening($G_D, e...R_2$)
6: else
7: for each element $e$ in $R_1$ do
8: endfor
9: else
10: head $\leftarrow$ the tail label of $R_1$;
11: tail $\leftarrow$ the head label of $R_2$;
12: initialize a query $Q$ to be empty;
13: mark head and insert it in $Q$;
14: while $Q$ is nonempty do
15: $x \leftarrow$ GetFront($Q$);
16: for each unmarked and marked vertex that doesn’t make a cycle $w$ adjacent to $x$ do
17: if $w = tail$ then
18: print the current path from head to tail;
19: mark $w$;
20: endfor
21: endfor
22: endwhile
23: endfor
24: endif

Example 10. Given the DTD graph in Fig. 5 and the query $person_. * .name$, Algorithm 3 produces the query ($person.name|person.company.name$).

5.3. 2*-index-join

Algorithm 4 shows our query processing algorithm named 2*-index-join when given a query $q$ and a set of views $v$. It is composed of two steps: query rewriting and query processing. In the query rewriting phase, the given query is rewritten using views by our mapping algorithm. In the second phase, each unchanged query conjunct is evaluated by the 2*-index technique and the join of the result of views and unchanged query conjunct is returned.

Algorithm 4. 2*-index-join algorithm

**INPUT:** The query $q(u) = \ldots y_1 r_{n-1} z_{n-1}$, the set of views $v = \{v_1, \ldots, v_m\}$

**OUTPUT:** A set of oid tuples which are mapped to head variables $u$ of $q$

**Step 1:** Query rewriting

- Find mappings $\Pi_i$ from $v_i \in v$ to $q$ (Section 4.2.1);
- For each $v_i \in v$, let $v'_i = p(x, y)$ be the view after applying $\pi \in \Pi_i$ to the view $v_i$, and $q'(u) = p'(x, y, s(y, z))$, rewrite the query as $q'(u) = v'_i s(y, z)$ (Section 4.2.2);

**Step 2:** Query processing

- For the final $s(y, z)$ in step 1, if $s(y, z) = y_0 r_{0,0}^0 \ldots y_{n-1} r_{n-1} z_{n-1}$, and $Q_0, \ldots, Q_{k-1}$ are relations corresponding to each query conjunct, find tuples of $Q_0, \ldots, Q_{k-1}$ by the 2*-index technique (Section 5.2);
- Letting $v_0, \ldots, v_i$ be views which participate in query rewriting and $W_1, \ldots, W_k$ be corresponding relations, return $\pi_0 W_1 \bowtie \ldots \bowtie W_k \bowtie Q_0 \bowtie \ldots \bowtie Q_{k-1}$.

Example 11. Consider the following query and view.

$q(p_3) = \neg p_1(movie|drama)p_3$, $p_2(year.1995)p_3$, $p_2(actor.name."BradPit")p_4$, $v : q_1(movie|drama)q_2$, $q_2(year.L)q_3$.
The query asks for movies and dramas that have an actor named ‘Brad Pitt’ and were produced in 1995. The 2*-index-join algorithm finds a mapping \( \Pi = \{(q_1 \rightarrow p_1, q_2 \rightarrow p_2, q_3 \rightarrow p_3), (1995/\ell)\} \) in step 1. Let \( v' \) be the view obtained by applying this mapping to the view \( v \). The query is rewritten to \( q'(p_2) : -v', p_2 \) \( (\text{name} \ldots \text{Brad Pitt}) \) \( p_4 \). If the rewritten query is applied to the data graph in Fig. 1, the result of \( v' \) is \( R_v(p_1, p_2, p_3) = \{(1, 3, 19), (1, 4, 23)\} \) and the result of \( p_2 \) \( (\text{name} \ldots \text{Brad Pitt}) \) \( p_4 \) is \( R(p_2, p_3) = \{(2, 26), (3, 26)\} \). So, finally, the result of the query is \( R_q(p_2) = \{3\} \).

5.4. Evaluation

As stated earlier, we can consider three kinds of query evaluation methods: forward scan, backward scan, and hybrid scan. Among them, we compared the 2*-index-join technique with the forward scan by a cost model. The backward scan and hybrid scan can outperform the forward scan in certain database statistics. For example, if only a few leaf nodes in the data graph satisfy a given predicate, the backward scan outperforms the forward scan. However, we assume that the three methods usually have similar performance.

We compared query processing techniques based on the number of objects fetched. This is reasonable since it is difficult to enforce object clustering in semistructured data models. So, we assume each object fetch has uniform cost.

One can predict that the 2*-index-join technique has the following disadvantages compared to the forward scan.

1. Join cost: Additional joins are required in the second phase of step 2 in Algorithm 4.
2. Loss of binding information: Assume that a query having two regular path expressions connected linearly is being evaluated. In the forward scan, when the second query conjunct is processed, the source graph is traversed for the second regular path expression from the objects reachable by the first regular path expression. However, in the 2*-index-join technique, each query conjunct is evaluated independently and joins are performed to them.

First, join operations in 2*-index-join have low cost, since relations participating in joins have tuples of oids that are the result of query conjuncts. For example, let us estimate the size of joins when a query has five query conjuncts. We use the cost formula in (Korth and Silberschatz, 1991), and estimate the size of \( r_1 \bowtie r_2 \ldots \bowtie r_5 \), and assume the following variables:

- \( R_1, R_2, \ldots, R_5 \): the set of attributes corresponding to \( r_1, r_2, \ldots, r_5 \)
- \( n_r \): the number of tuples in the relation \( r_i \)
- \( s_r \): the size of a tuple of the relation \( r_i \) in bytes, \( s_A \): the size of attribute \( A \)

\( V(A, r_i) \): the number of distinct values that appear in the relation \( r_i \) for attribute \( A \)

- \( R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 = A_5 \)
- \( R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 = A_4 \)
- \( R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 = A_3 \)
- \( R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 = A_2 \)
- \( R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 = A_1 \)
- \( R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 = A_0 \)

If we assume uniform distribution of values, then the query \( \sigma_{\text{dist}}(r) \) is estimated to have \( n_r/V(A, r) \) tuples. So, as a tuple of \( r_1 \) produces \( n_{r_2}/V(A_1, r_2) \) tuples in \( r_1 \bowtie r_2 \), all of the tuples in \( r_1 \) produces \( n_{r_1}n_{r_2}/V(A_1, r_2) \) tuples in \( r_1 \bowtie r_2 \). In the same manner, there are \( n_{r_1}n_{r_2} \ldots n_{r_5}/V(A_1, r_2) \ldots V(A_5, r_5) \) tuples in \( r_1 \bowtie r_2 \ldots \bowtie r_5 \). Finally, the size of \( r_1 \bowtie r_2 \ldots \bowtie r_5 \) is \( n_r n_{r_2} \ldots n_{r_5}/V(A_1, r_2) \ldots V(A_5, r_5) \) \( + \sum_{i=1}^{5} s_{r_i} - \sum_{i=1}^{5} s_{A_i} \). In 2*-index-join, the value of \( V(A_i, r_{i+1}) \) is the number of distinct objects in the data graph. Let us assume that \( a \) denotes this value. \( n_r \) is the number of pairs that satisfy a regular path expression for a query conjunct. The upper bound of this value is \( O(a^d) \), but in practice, this value is smaller than \( a \). So, we assume that \( n_r = a \). In 2*-index-join, each relation corresponding to each query conjunct has two attributes, that is, the relation has tuples of (source, target) oid pairs which are connected by the corresponding regular expressions. If we assume 4 bytes for each attribute, \( \sum_{i=1}^{5} s_{r_i} = 40 \) and \( \sum_{i=1}^{5} s_{A_i} = 16 \). So, the size of joins is at most 24 * \( a \) bytes. For example, if there are 1 million distinct objects in a data graph, the size of joins is at most 24,000,000 bytes. This value is less than 24M bytes, so the join cost in 2*-index-join is quite low.

Next, to evaluate the loss of binding information, we compare our technique to forward scan using an analytic model as in Abiteboul et al. (1998) and McHugh and Widom (1999b). Assume that the following quantities are statistics kept by the system.

- Fanout\((x, l)\): the estimated average number of children with the label \( l \) that are descendants of some object in the set of objects bound to variable \( x \). For example, if the objects having oids 2 and 3 are bound to the variable \( z \) in Fig. 1, then \( \text{Fanout}(z, \text{name}) = 1 \).
- \( |x| \): the estimated number of objects bound to \( x \). In the above example, \( |x| = 2 \).
- \( \text{cost}(x, l, y) \): the estimated number of objects fetched in order to get all of the sub-objects with edges labeled \( l \) originating from any object in \( x \), where each resulting object is placed into \( y \). This cost is computed by \( |x| \times \text{Fanout}(x, l) \). For example, given a simple path expression \( z(\text{name}, w) \), the cost is \( \text{cost}(z, \text{name}, w) = |z| \times \text{Fanout}(z, \text{name}) = 2 \times 1 = 2 \).
- OIDJoin: join cost of the result sets in 2*-index-join.

---

\(^5\) This is defined in McHugh and Widom (1999b), and means single step navigation in the database.
2*-index-join evaluates the following form of queries.
\[ q(u) : w(x, y), c(y, z) \]  
(4)

Here, \( w(x, y) \) denotes rewritten queries using views among query conjuncts and \( c(y, z) \) denotes unchanged query conjuncts. When \( c(y, z) = \phi \), complete rewritings (Levy et al., 1995) exist and when \( w(x, y) = \phi \), there is no rewriting using views. In both cases, our technique can be applied to answer the queries. We compute the cost of the forward scan and 2*-index-join when \( w(x, y) = \phi \). \(^6\) and \( c(y, z) \) has the form of a query in Formula (3). Additionally, assume that \( r_0 = a \) and \( r_1 = b \). In this case, the forward scan has an advantage over 2*-index-join since the regular path expressions are the simplest forms.

This is because that if regular path expressions are complex, 2*-index-join has an advantage over the forward scan. Then, the cost of the forward scan and 2*-index-join are calculated as follows. Here, let \( |p_i| \) be the number of objects bound to variable \( p_i \) in the source database and \( |p_i^2| \) be those in the \( I^2 \) graph. Similarly, \( Fanout(p_i, l) \) denotes the average number of objects that are traversed by label \( l \) from the set of objects bound to variable \( p_i \) in the source database and \( Fanout^2(p_i, l) \) denotes those in the \( I^2 \) graph.

\[
Cost_{\text{forward scan}} = |p_i| Fanout(p_i, a) + |p_i^2| Fanout^2(p_i, b)
\]
\[
= |p_i| Fanout(p_i, a) + |p_i^2| Fanout^2(p_i, a) Fanout^2(p_i, b)
\]

\[\text{Cost}_{2*-\text{index-join}} = |p_i^2| Fanout^2(p_i, a) + |p_i^2| Fanout^2(p_i, b) + \text{OIDJoin}\]

In the \( I^2 \) graph, \( |p_i^2| = |p_i^2| \) = the number of roots, and these are all of the equivalence classes of the form \([x, x]\). Often, these are only a few. In particular, \( I^2 \) has a single root in acyclic databases. Also, \( Fanout^2(p_i, l) \gg Fanout^2(p_i, l) \). Finally, the cost of OIDJoin is very low as it is joining between oids. So, \( Cost(\text{forward scan}) \gg Cost(2*-\text{index-join}) \).

5.5. An experiment

We have implemented our technique described in this paper with about 3500 lines of Java code to illustrate its enhancement in query processing. Our technique is applied to a MLB database \(^7\) that is composed of 14,646 objects including 60 teams and 2400 players.

Table 1 shows queries used in the experiment. Here, the query \( Q_1 \) is composed of a single regular path expression and the others are composed of multiple regular path expressions.

The number of objects searched to evaluate the queries is presented in Table 2. We compared our technique with naive evaluation of the query. The results show that our technique reduces the search space significantly.

Next, Table 3 shows the total elapsed time. In this case, we can see that the small cost OIDJoin is performed in our technique. Compared to the cases \( Q_1 \), \( Q_2 \), and \( Q_3 \), our technique shows a poor performance result when the query \( Q_4 \) is evaluated. This is because that the query \( Q_4 \) requires almost entire graph traversal to answer the query. However, our technique outperforms naive evaluation of the query.

6. Conclusion

We propose an efficient technique to process semi-structured queries having multiple regular path expressions. Our algorithm is composed of two steps: query rewriting using views and query processing. In step 1, our algorithm finds symbol mappings from the given query and the views. In this case, by using structural information of the query and the views, we reduce the complexity of the mapping algorithm. In step 2, we suggest a query evaluation technique that processes each query conjunct independently by an index technique named 2*-index, and then combines the query result by low cost joins. We show that our technique outperforms the general query processing technique.

### Table 1

<table>
<thead>
<tr>
<th>Queries used in the experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 ( q_1(b) : \neg a ) (National.West.player.name) b</td>
</tr>
<tr>
<td>Q2 ( q_2(b) : \neg a ) (National.West) b (player.nickname) c</td>
</tr>
<tr>
<td>Q3 ( q_3(c) : \neg a ) (American.East) b (stadium) c.b (player.nickname) d</td>
</tr>
<tr>
<td>Q4 ( q_4(c) : \neg a ) (National.Central) b (name) c.b (player.name) d</td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th>The number of objects searched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive evaluation: 841 451 452 851</td>
</tr>
<tr>
<td>2*-index-join: 6 8 11 11</td>
</tr>
</tbody>
</table>

### Table 3

<table>
<thead>
<tr>
<th>The total elapsed time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive evaluation: 2123 1105 1479 3287</td>
</tr>
<tr>
<td>2*-index-join: 20 24 45 1093</td>
</tr>
</tbody>
</table>

---

\(^6\) In this case, the forward scan has an advantage over 2*-index-join.

\(^7\) This is constructed synthetically by programming techniques but it is similar to the real MLB database.

### References

Abiteboul, S., McHugh, J., Rys, M., Vassalos, V., Weiner, J., 1998. Incremental maintenance for materialized views over semistruc-