Transforming XPath Expressions into Relational Algebra Expressions With Kleene Closure

Yangjun Chen

Dept. Applied Computer Science, University of Winnipeg
515 Portage Ave., Winnipeg, Manitoba, Canada, R3B 2E9
y.chen@uwinnipeg.ca

***Abstract*—In the problem of translating XPath expressions into SQL queries, the most challenging part is to find a way to minimize the use of least fixpoint (*LFP*) operators when a *DTD* graph contains cycles. In this paper, we address this issue and present a new algorithm to do the task based on the recognition of a kind of DTD graphs, which can be reduced to a single node by contracting nodes into their parents one by one. For this kind of DTD graphs, not only the corresponding relational algebra expressions can be efficiently generated, but the use of LFP operators can also be minimized. For those DTD graphs that are not reducible, we devise a different algorithm which is less efficient than the algorithm for reducible graphs, but more efficient than any existing method.**

Keywords: *XML, XPath, Query Processing, XPath transformation*

I. INTRODUCTION

With the widespread of XML both as a document format and as a data exchange format, the interest in querying XML data stored in relational databases has increased. With this comes the need for answering XML queries in a relational database system, by translating XML queries to SQL statements [11, 19, 21]. It is quite different from the prevailing methods for evaluating *twig joins* [7, 8, 9, 10, 11].

Let D be a DTD (Document Type Definition). Let R be a relational schema defined for D by using the shared-inlining technique [24], denoted as a mapping f: $D \rightarrow R$. Denote by $\mathscr{D}$ all the XML documents conforming to D. Denote by $\mathscr{R}$ all the possible relational states of R. Then, the storage of a set of documents conforming to D in $DB(R)$ (a database with the relational schema R) can be considered as a mapping derived from f, denoted as f_s: $2^{\mathscr{D}} \rightarrow \mathscr{R}$.

Given an XPath expression Q, what we want is to find an equivalent relational algebra expression Q', which can be evaluated against $DB(R)$, such that for any document $d \in \mathscr{D}$, Q on d can be answered by Q' on $f_s(\{d\})$. That is, the set of nodes selected by Q on T equals the set of tuples selected by Q' on $f_s(\{d\})$. We denote this by

$$Q(T) = Q'(f_s(\{d\})).$$

When a DTD is simply a tree or a *DAG* (*directed acyclic graph*), a simple translation can be conducted by enumerating all matching paths of the input XPath expression in the DTD, sharing common subpaths, rewriting the paths as relational algebra expressions, and taking a union of all of them [12]. However, when a DTD contains recursive element type definition, the problem becomes challenging [4, 5]. In this case, the interaction between recursion in the DTD and recursion (*descendant-or-self* axis, represented by '//') in an XPath expression significantly complicates the translation.

In the past decade, a lot of work has been done on querying XML data stored in relational databases such as those discussed in [7, 9, 10, 12, 14, 23]. However, as surveyed in [15], in all these methods, except the strategies proposed in [14, 25], the problem of translating recursive XML queries over recursive DTD is not addressed.

The method discussed in [14] is capable of translating path queries with '//' to a sequence of SQL queries using the SQL'99 recursion operator. However, the SQL queries produced by [14] tend to be large and complicated and cannot be effectively optimized. Also, as pointed out by Fan *et al.* [25], the method is applicable only to a very limited class of path expressions.

In [25], Fan *et al*. proposed a different method. The main idea of this method is to transform an XPath expression to an *extended* XPath expression, in which some variables may be used to represent sub-expressions. In addition, any '//' is replaced with a *Kleene* closure. Given an extended XPath expression, a sequence of relational algebra expressions can be easily created. The time complexity of this process is bounded by $O(|D|^3|Q|\log|D|)$. When translated to an extended XPath expressions, a Kleene closure of the form: E^* corresponds to a sub-expression of the form: $A//B$ in an XPath expression, and E represents all the paths from A to B in the corresponding DTD graph.

However, the generated expressions are also very large with many unnecessary joins involved. For example, for the graph shown in Fig. 1, the regular expression generated by Fan's algorithm [25] for the path from v_1 to v_1 would be

$$e_0 \cup e_0^* \cup ((e_1 \cup e_0^*\cdot e_1)/(e_4\cdot e_0^*\cdot e_1)^* \cdot (e_4 \cup e_4\cdot e_0^*)).$$

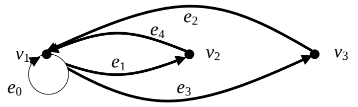


Fig. 1. A directed graph

But the minimized regular expression for this is $(e_0 \cup e_4 \cup e_2)^*$. In the Appendix, we will make a sample trace to show how Fan's algorithm [25] works in generating such a complicated expression over the above graph.

It is obvious that the Kleene closure is a very costly operation. It dominates the whole query evaluation time. So, it is necessary to reduce the size of E such that as few joins as possible are involved.

In this paper, we propose a new algorithm to mitigate the problem to some extent. We will recognize a class of DTD graphs G, for which a reduction sequence of nodes: v_1, v_2, …, $v_n = r$ can be found such that G can be reduced to a single node r, where n is the number of G's nodes. For this kind of DTD graphs, we can create a relational algebra expression in O($m$log$n$) time, where m is the edges of G. More importantly, we can always find a way to generate minimized relational algebra expressions for them. For those non-reducible graphs, we propose a different algorithm. Although it is less efficient than the algorithm for the reducible graphs, it is more efficient than any existing strategies.

The paper contains five sections. In Section 2, we review DTDs, XPath expressions, and schema-based mapping from XML to relations. In Section 3, we concentrate on the recursion in XPath expressions. Section 4 is devoted to the general process for transforming XPath queries to relational algebra expressions. Finally, the paper concludes in Section 5.

II. BASIC CONCEPTS

In this section, we review DTDs and XPath expressions, as well as the XML data storage in relational databases to provide a discussion background.

- *DTD*

Abstractly, an XML DTD can be considered as a triple <H, S, r>, where H is a set of element types (corresponding to element tag names); r is the root type; and S is a set of rules defining the types in H. That is, for any type A in H, $S(A)$ (the definition of A) is an expression:

$$\beta ::= \varepsilon \mid B \mid \beta, \beta \mid (\beta \mid \beta) \mid \beta^*,$$

where ε is the empty word, B represents a type in H (referred to as a subelement or child type of A), and '|', ',' and '*' denote disjunction, concatenation, and the Kleene star, respectively. We refer to $A \rightarrow S(A)$ as the production of A.

We will represent a DTD D as a graph, called the DTD graph of D and denoted by G_D, as done in [24]. In G_D, each node stands for a distinct element type and each edge for a parent/child relationship. In addition, an edge (A, B) is marked with '*' if B is enclosed in a definition of A with the form: β*.

As an example, see the DTD graph shown in Fig. 2, representing a DTD: <H, S, dept> with

H = {dept, course, cno, title, time, prereq, takenBy, taughtBy, professor, pno, pname, teaching, student, sno, sname, qualified}, and

S defined as follows:

dept → course*
course → cno, title, time, prereq, takenBy, taughtBy
prereq → course*
takenBy → course*
taughtBy → professor*
student → sno, sname, qualified
qualified → course*
professor → pno, pname, teaching
teaching → course*

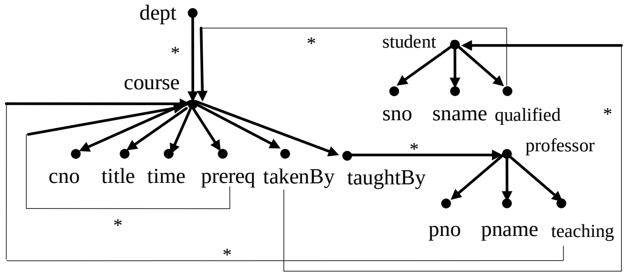


Fig. 2. A DTD graph

In the above DTD graph, we handle each attribute as a primitive element type for simplicity. But it obviously does not lose any generality.

A DTD is recursive if it has an element type that is defined (directly or indirectly) in terms of itself. When represented as a graph, it will contain a few nested and overlapping cycles. So the DTD shown in Fig. 2 is recursive.

- *XPath expressions*

XPath [6] is a popular language for querying XML data. It has been used in many XML applications and in some other languages for querying and transforming XML data, such as *XQuery* and *XSLT*. In this paper, we address a practical fragment of XPath, in which each *path* in a *predicate* can be compared with a constant, but not with another *path*, given as below:

$$p ::= . \mid A \mid * \mid p/p \mid p//p \mid p[q] \mid$$
$$q ::= p \mid p\ \delta\ c \mid \neg q \mid q \vee q \mid q \wedge q$$
$$\delta ::= \text{'='} \mid \text{'!='} \mid \text{'>'} \mid \text{'>='} \mid \text{'<'} \mid \text{'<='}$$

where '.', A, and * denote the *self*-axis, a type (element tag name) and a wild card, respectively. '/' and '//' are *child*-axis and *descendant-or-self*-axis, respectively; and [q] is a predicate (also referred to as a *qualifier*), in which c is a constant and δ represents a comparison relation. For example, the following XPath expression

/dept/course[title = 'XML' or

(¬(time = 2008) and prereq = 'CS2201')]//professor

selects *the professor who taught a course either with title 'XML' or with the prerequisite 'CS2201' but not in 2008*. Such an XPath expression can be represented as a tree with five kinds of nodes: axis-tag nodes (*at*-node), logical-AND nodes (∧-node), logical-OR nodes (∨-node), logic-negation node, and constant node:

- *at*-node: An axis-tag node in the tree stands for one location step. It has the content /*tag* or //*tag*.
- ∧-node: A logical-AND node connects two or more child subtrees with AND logic.
- ∨-node: A logical-OR node connects two or more child subtrees with OR logic.
- ¬-node: A logical-negation node negates the result of its unique subtree.

- *C*-node: A constant node is a value of the form: = *c*, != *c*, < *c*, > *c*, <= *c*, or >= *c*.

For example, the above XPath expression can be represented as a tree shown in Fig. 3.

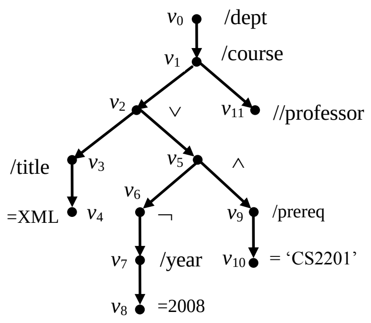


Fig. 3. A tree representing an XPath expression

For any *at*-node *v*, we use *v*.*axis* and *v*.*tag* to refer to the axis ('/' or '//') and the tag appearing in *v*, respectively. In addition, we define some operations on query tree nodes:

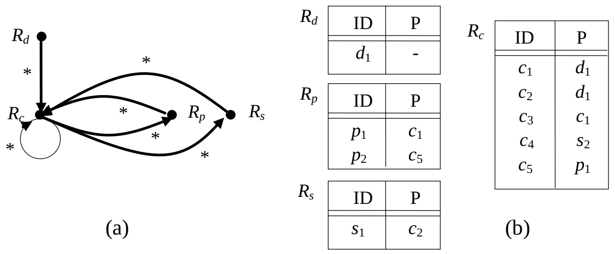
children(*v*) – returns all child nodes of *v*;
parent(*v*) – returns the parent of *v*;
atChildren(*v*) – returns a set of at-nodes in the subtree rooted at *v*, which are reachable without traversing through other *at*-nodes.
atParent(*v*) – returns the nearest ancestor *at*-node of *v*.

For example, given the query tree in Fig. 3, we have *children*(v_1) = {v_2, v_{11}}, *parent*(v_5) = v_2, *atChildren*(v_1) = {v_3, v_7, v_9, v_{11}}, *atParent*(v_9) = v_1.

- *Mapping a DTD into a relation schema*

In order to store a set of XML documents (conforming to a certain DTD *D*) in a relational database, we will first establish a map *f*: $D \rightarrow R$ from *D* to a relational schema *R*. To this end, we will first remove all the edges marked with '*' in G_D, dividing it into several node-disjoint components: G_1, …, G_k. Each G_j is then mapped to a relation schema R_j in *R*, which has three attributes: *ID* (identifier of elements), *P* (parent of the current element) and *V* (for the values of all the other attributes). If G_j has more than one incoming edges, we will use a *parentCode* attribute [24] to distinguish among different parents.

From *f*, one can easily derive a data mapping f_s: $2^{\mathcal{D}} \rightarrow \mathcal{R}$, representing the storage of a set of documents in *DB*(*R*). Let *d* be a document conforming to *D*. Then, in $f_s(\{d\})$ (the database storing *d*), a tuple (*id*, *p*, *v*) in a relation with a certain schema R_A represents an element in *d* with its identifier equal to *id*, its parent element identifier to *p*, and all its attribute values represented by *v*. For example, the DTD shown in Fig. 1 can be mapped to four relation schemas: R_d, R_c, R_p, and R_s, representing *dept*, *course*, *professor*, and *student*, respectively. These four relation schemas are connected as shown in Fig. 3(a). In Fig. 3(b), we show a sample database, in which for each relation only values for *ID* and *P* are displayed. (See [24] for a detailed discussion.)



R_d

ID	P
d_1	-

R_p

ID	P
p_1	c_1
p_2	c_5

R_s

ID	P
s_1	c_2

R_c

ID	P
c_1	d_1
c_2	d_1
c_3	c_1
c_4	s_2
c_5	p_1

Fig. 4. A DTD graph and a set of relations

III. On the Recursion in XPath

The most difficult issue in translation of XPath expressions is the treatment of //-axis. In this section, we mainly address this problem. The discussion of a general process for the transformation is shifted to the next section.

A. Reducible subgraphs

Consider an XPath query *A*//*B* over a DTD *D*. This query, when evaluated at an *A*-element in a document *T* conforming to *D*, is to find all *B*-elements which are the descendants of the *A*-element. To do this, Fan *et al.* [25] proposed an algorithm to create a sequence of extended XPath expressions to represent all the paths connecting *A* to *B*. (An extended XPath is a regular XPath expression [18] with variables being used.) As shown in the introduction, an extended XPath expression generated by Fan *et al.* [25] can be very large.

To mitigate this problem, we recognize a class of graphs, for which not only the corresponding relational algebra expressions can be efficiently created, but the use of LFP operations is minimized.

First, we notice that what we want is to produce an expression for a graph containing all the paths from a node α to another nodeβ in a certain graph *G*. We call such a graph an $\alpha\beta$-graph, denoted as $G[\alpha, \beta]$. Obviously, every node in $G[\alpha, \beta]$ is reachable from α.

Definition 1 An $\alpha\beta$-graph $G[\alpha,\beta]$ is reducible if it can be reduced to a graph consisting of a single node by means of the following transformations:

O_1 (Remove a loop): If *e* is an edge such that *head*(*e*) = *tail*(*e*), delete *e*. (Note that for an edge *e* = (*a*, *b*), *head*(*e*) = *a* and *tail*(*e*) = *b*.)

O_2 (Remove a node): If $u \neq \alpha$ is a node such that all edges *e* with *tail*(*e*) = *u* have *head*(*e*) being a same node *v*, contract *u* into *v* by deleting *u* and all edges from *v* to *u*, and converting any edge *e* with *head*(*e*) = *u* into an edge *e'* with *head*(*e'*) = *v* and *tail*(*e'*) = *tail*(*e*). (We remark that *v* may be connected to *u* by multiple edges.) □

As an example, consider the subgraph $G[R_c, R_p]$ of the graph shown in Fig. 4(a). It can be reduced as shown in Fig. 5.

From Fig. 5, we can see that in each step we remove some loops and then contract a node *u* ($\neq \alpha$) into another node *v* if *v* is the unique parent of *u*. This process continues until only one node is left.

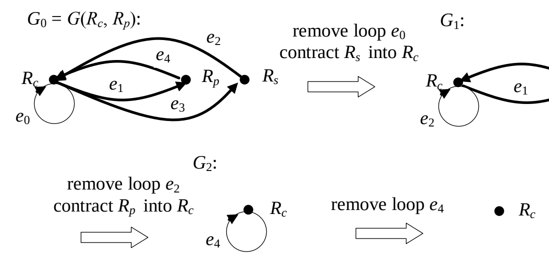


Fig. 5. Illustration of graph reduction

If in a certain step we cannot find a node ($\neq \alpha$) which has only one parent, the process gets stuck. Then the corresponding $\alpha\beta$-graph is *non-reducible*. Since for the $\alpha\beta$-graph in the graph shown in Fig. 4(a) the reduction can always be conducted, it is reducible.

To check whether an $\alpha\beta$-graph is reducible, we do the following operation repeatedly.

1. Remove all the loops.
2. Check each u in the $\alpha\beta$-graph to see whether O_2 can be applied to it. If it is the case, contract it.

Obviously, this process requires $O(n^2)$ time, where n is the numbers of the nodes of the $\alpha\beta$-graph.

From the above discussion, we can see that for a reducible $\alpha\beta$-graph a reduction sequence of nodes: $v_1, v_2, \ldots, v_n = \alpha$ can be found such that the $\alpha\beta$-graph can be reduced to α by removing v_i in the sequence. For convenience, we call α the root of the $\alpha\beta$-graph. For example, for the $G[R_c, R_p]$ shown in Fig. 5, the reduction sequence of the nodes is: R_s, R_p, R_c. Its root is R_c.

Accordingly, we also get a sequence of graphs: $G_0, G_1, \ldots, G_{n-1}$ such that G_0 is the original $\alpha\beta$-graph and $G_i = G_{i-1}/\{v_i\}$ for $i > 0$ (see Fig. 5 for illustration). For an edge $e \in G_i$, we use $head_i(e)$ and $tail_i(e)$ to represent its head and tail in G_i, respectively. We notice that for the same edge e appearing in G_i and G_j with $i \neq j$, it is possible that $head_i(e) \neq head_j(e)$. For instance, in G_0 (see Fig. 5), $head_0(e_2) = R_s$. But in G_1, $head_1(e_2) = R_c$. However, for any e, if it appears in $G_0, G_1, \ldots, G_i$ for some i, we must have $tail_0(e) = tail_1(e) = \ldots = tail_i(e)$.

In the graph reduction process, we can associate each node v with three data structures to facilitate the creation of relational algebra expressions:

loop(v): a set of edges such that for each e in the set there exists G_i for some i such that we have $head_i(v) = tail_i(v)$.

non-loop(v): a set of edges, along which v is contracted into another node. (Remember that we may have multiple edges in a graph.)

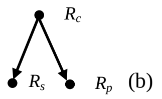
contractor(v): a node, into which v is contracted.

Since each node has only one contractor, the contraction process can be represented by a tree (called a *contraction spanning tree* and denoted as *CST*), in which there is an edge from v to u if u is contracted into v.

Example 1 In the graph reduction process shown in Fig. 5, a set of data structures will be constructed, as shown in Fig. 6(a). Fig. 6(b) shows the corresponding CST tree.

	loop(v)	*non-loop*(v)	*contractor*(v)
R_s		e_3	R_c
R_p		e_1	R_c
R_c	e_0, e_2, e_4		

(a)



(b)

Fig. 6. Data structures and contraction tree

In order to generate an expression for a reducible $\alpha\beta$-graph, we explore the correspnding CST tree bottom-up. During the traversal of the CST, for each encountered node v, the associated data structures are used to generate an expression for it, which is then utilized to create an expression representing all the paths from the root to v.

```
Algorithm gen-expression(T) (*T is a CST.*)
begin
1.  search T bottom-up;
2.  for each encountered node v do
3.  {   $E_v \leftarrow \phi$; $Q_v \leftarrow \phi$;
4.      for each $e \in$ non-loop(v) do
5.          {$E_v \leftarrow E_v \cup R_{head_0(e)} \bowtie R_{tail_0(e)}$ ;}
6.      for each $e \in$ loop(v) do
7.      {let $v_1 \rightarrow \ldots \rightarrow v_k$ be the path from v to $head_0(e)$ in T;
8.          $Q \leftarrow E_{v_1} \bowtie \ldots \bowtie E_{v_k}$ ;
9.          $Q_v \leftarrow Q_v \cup Q \bowtie R_{head_0(e)} \bowtie R_{tail_0(e)}$ ;
10.     }
11.     $E_v \leftarrow E_v \bowtie Q_v$*;
12. }
13. for each node v do
14. {   let $u_1 \rightarrow \ldots \rightarrow u_j$ be the path from $\alpha$ to v in T;
15.     $E_{\alpha\text{-}v} \leftarrow E_{u_1} \bowtie \ldots \bowtie E_{u_j}$ ;
16. }
End
```

The algorithm works in two phases. The first phase consists of lines 1 - 12. The second phase consists of lines 13 - 16. In the first phase, we create an expression for each node v. In the second phase, for each node v, the expression representing all paths from the root to it is generated by using the expressions generated in the first phase.

Example 2 Applying the above algorithm to the CST tree and the data structures shown in Fig. 6, we will generate the following expressions step by step.

first phase:

Step 1: access R_s. $loop(R_s) = \phi$. $non\text{-}loop(R_s) = \{e_3\}$.
$E_s = R_c \bowtie R_s$.

Step 2: access R_p. $loop(R_p) = \phi$. $non\text{-}loop(R_p) = \{e_1\}$.
$E_p = R_c \bowtie R_p$.

Step 3: access R_c. $loop(R_p) = \{e_0, e_2, e_4\}$.
$head_0(e_0) = R_c$. $head_0(e_2) = R_s$. $head_0(e_4) = R_p$.
$head_0(e_3) = R_{pub.}$

$E_c = (R_p \bowtie R_p \cup$

$(R_c \bowtie R_s) \bowtie (R_s \bowtie R_c) \cup$

$$(R_c \bowtie R_p) \bowtie (R_p \bowtie R_c))^*$$
$$= (R_p \cup R_c \bowtie R_s \cup R_c \bowtie R_p)^*$$

second phase:

Step 4: $E_{c\text{-}s} = E_c \bowtie E_s$.

Step 5: $E_{c\text{-}p} = E_c \bowtie E_p$.

Step 6: $E_{c\text{-}c} = E_c \bowtie E_c = E_c$. □

Proposition 1 Let v be a node in a reducible $\alpha\beta$-graph. Then the expression $E_{\alpha\text{-}v}$ produced by *gen-expression*( ) exactly represents all paths from α to v.

Proof. We use $\chi(E_{\alpha\text{-}v})$ to represent a set containing all the paths represented by $E_{\alpha\text{-}v}$. We prove the proposition by induction on the length of the path p from α to v in the corresponding CST tree T.

Basis. When $|p| = 1$, α is the contractor of v. $E_{\alpha\text{-}v} = E_\alpha \bowtie E_v$. The proposition holds.

Induction step. Assume that when $|p| = k$ the proposition holds. We consider the case that $|p| = k + 1$. Let v' be the contractor of v. Then the length of the path from α to v' is k. According to the induction assumption, $E_{\alpha\text{-}v'}$ exactly represents all paths from α to v'. Since the $\alpha\beta$-graph is reducible, all paths reaching v must go through v'. So the expression should be $E_{v'} \bowtie E_v$. It is exactly done by the algorithm. □

B. Non-reducible subgraphs

If an $\alpha\beta$-graph $G[\alpha,\beta]$ is not reducible, then in the reduction process we will reach a graph G', in which we are not able to find a node ($\neq \alpha$) that has only one parent.

Let $v_1, v_2, \ldots, v_j$ ($j < n$) be the node sequence removed from $G[\alpha,\beta]$ in the incomplete reduction process. Let $G_0, G_1, \ldots, G_{j+1}$ be the corresponding graph sequence such that $G_0 = G[\alpha,\beta]$, $G_{i+1} = G_i/\{v_i\}$ ($i = 0, \ldots, j$), and G_{j+1} cannot be reduced any more. We call G_{j+1} a remaining graph. Let $r_1, \ldots, r_l$ be those nodes in G_{j+1} such that into each of them some v_i ($1 \leq i \leq j$) is contracted. Then, we will construct l CST trees: $T_1, \ldots, T_l$ in the same way as discussed in the previous subsection. Each T_i is rooted at r_i ($1 \leq i \leq l$). Assume that β is a node in some T_k ($1 \leq k \leq l$). Then, we can construct an expression $E_{r_k-\beta}$ representing all paths from r_k to β, as discussed in 3.1.

As an example, consider an $G[\alpha,\beta]$ shown in Fig. 7(a) with $\alpha = v_2$ and $\beta = v_7$.

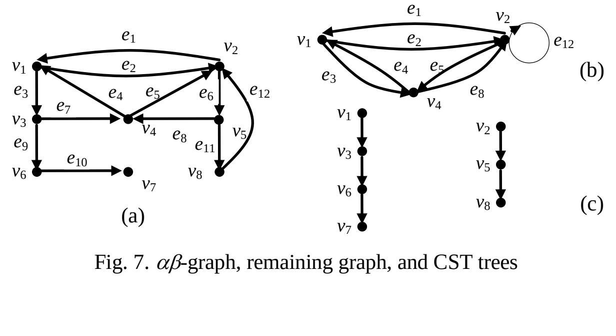


Fig. 7. $\alpha\beta$-graph, remaining graph, and CST trees

For this graph, we can find a contraction sequence of nodes: v_7, v_6, v_3, v_8, v_5, and a series of contracted graphs: $G_0 = G[\alpha,\beta]$, $G_1 = G_0/\{v_7\}$, $G_2 = G_1/\{v_6\}$, $G_3 = G_2/\{v_3\}$, $G_4 = G_3/\{v_8\}$, $G_5 = G_4/\{v_5\}$. We show G_5 in Fig. 7(b). It is a remaining graph and cannot be reduced any more. In G_5, special attention should be paid to v_1 and v_2. They are the contractors of v_3 and v_5, respectively. So, two CST trees will be constructed for the removed nodes as shown in Fig. 7(c). The expression representing all the paths from v_1 to v_7 is easy to compute: $E_{v_1-v_7} = v_7 \bowtie v_6 \bowtie v_3 \bowtie v_1$.

In a next step, we need to calculate $E_{\alpha-r_k}$ over the remaining graph, and produce an expression $E_{\alpha-r_k} \bowtie E_{r_k-\beta}$ as the final result.

However, $E_{\alpha-r_k}$ cannot be calculated in the same way as an expression over a reducible graph. A different algorithm has to be devised. In the following, we discuss this algorithm in detail. Its time complexity is the same as Fan's algorithm [25]. But much less computation will be conducted.

Let G' be a remaining graph. We number its nodes from 1 to n', where n' is the number of the nodes in G'. Our purpose is to produce a set of expressions of the form $E_{i\text{-}j}$ with each representing all paths from i to j, from which $E_{\alpha\text{-}r}$ can be created, where r is the root of some CST tree, in which β appears.

So the algorithm works in two phases. In the first phase, we create necessary $E_{i\text{-}j}$'s. In the second phase, we generate $E_{\alpha\text{-}r}$.

```
Procedure phase-1(G')
begin
1.    for $i$ = 1 to $n$ do
2.    {  for $j$ = 1 to $n$ do
3.       {  if $e = (i, j)$ is an edge in $E$ then $E_{i\text{-}j} \leftarrow i \bowtie j$;
4.          else $E_{i\text{-}j} \leftarrow \phi$;
5.       }
6.    }
7.    for $k$ = 1 to $n$ do
8.    {  $E_{k\text{-}k} \leftarrow E_{k\text{-}k}^*$;
9.       for $i = k + 1$ to $n$ do
10.      {  if $E_{i\text{-}k} \neq \phi$ then $E_{i\text{-}k} \leftarrow E_{i\text{-}k} \bowtie E_{k\text{-}k}$;
11.         for $j = k + 1$ to $n$ do
12.         {  if $E_{i\text{-}j} \neq \phi$ then $E_{i\text{-}j} \leftarrow E_{i\text{-}j} \cup E_{i\text{-}k} \bowtie E_{k\text{-}j}$;
13.         }
14.      }
15.   }
end
```

Example 3 In this example, we trace the computation when the above algorithm is applied to the graph shown in Fig. 7(b). In the process, the three nodes v_1, v_2, and v_4 in the graph are numbered with 1, 2, and 3, respectively. Besides, we use '·' and e_i to represent respectively $\bowtie$ and $head(e_i) \bowtie tail(e_i)$ for simplicity. We also use I to represent an *identity relation* such that for any relation R we have $I \bowtie R = R \bowtie I =$

R. Finally, for a Kleene operation R^*, if R is ϕ or I, R^* is defined to be I.

Initialization (lines 1 – 5):

$$E^{(0)} = \begin{bmatrix} \phi & e_1 & e_4 \\ e_2 & e_{12} & e_5 \\ e_3 & e_8 & \phi \end{bmatrix}$$

First iteration (k = 1):

$$E^{(1)} = \begin{bmatrix} I & e_1 & e_4 \\ e_2 & e_{12} \cup e_2 \cdot e_1 & e_5 \cup e_2 \cdot e_4 \\ e_3 & e_8 \cup e_3 \cdot e_1 & e_3 \cdot e_4 \end{bmatrix}$$

Second iteration (k = 2):

$$E^{(2)} = \begin{bmatrix} I & e_1 & e_4 \\ e_2 & (e_{12} \cup e_2 \cdot e_1)* & e_5 \cup e_2 \cdot e_4 \\ e_3 & (e_8 \cup e_3 \cdot e_1)\cdot(e_{12} \cup e_2 \cdot e_1)* & e_3 \cdot e_4 \cup (e_8 \cup e_3 \cdot e_1)\cdot(e_{12} \cup e_2 \cdot e_1)* e_5 \cup e_2 \cdot e_4 \end{bmatrix}$$

Third iteration (k = 3):

$$E^{(3)} = \begin{bmatrix} I & e_1 & e_4 \\ e_2 & (e_{12} \cup e_2 \cdot e_1)* & e_5 \cup e_2 \cdot e_4 \\ e_3 & (e_8 \cup e_3 \cdot e_1)\cdot(e_{12} \cup e_2 \cdot e_1)* & (e_3 \cdot e_4 \cup (e_8 \cup e_3 \cdot e_1)\cdot(e_{12} \cup e_2 \cdot e_1)* e_5 \cup e_2 \cdot e_4)* \end{bmatrix}$$

□

Concerning the above algorithm, we have the following proposition.

Proposition 2 After the execution of *phase*-1, the following statements are true:

i) $E_{i\text{-}j}$ for $i \geq j$ is an expression representing exactly the paths from i to j which contain no intermediate node larger than j.

ii) $E_{i\text{-}j}$ for $i < j$ is an expression representing exactly the paths from i to j all of whose intermediate nodes are smaller than i.

Proof. Straightforward by induction on k. □

In terms of this proposition, we design the second phase to generate $E_{\alpha\text{-}r}$.

If $\alpha \geq r$, for all those joins of the form $E_{\alpha\text{-}j} \bowtie E_{j\text{-}r}$, which should be included in $E_{\alpha\text{-}r}$, j should be larger than r. If $\alpha < r$, j should be larger than or equal to α.

According to the above analysis, we give the following procedure.

```
Procedure phase-2(E) (*E contains all the expressions
                       produced in phase-1.*)
Begin
1.    If $\alpha \geq r$ then $j \leftarrow r + 1$;
2.    else $j \leftarrow \alpha$;
3.    $E_{\alpha-r} \leftarrow E_{\alpha-r_k} \cup (E_{\alpha-j} \bowtie E_{j-r}) \cup \dots \cup (E_{\alpha-n'} \bowtie E_{n'-r})$;
end
```

For example, the expression representing all paths from v_2 to v_1 in the graph shown in Fig. 7(b) is

$$E_{2\text{-}1} \cup (E_{2\text{-}2} \bowtie E_{2\text{-}1}) \cup (E_{2\text{-}3} \bowtie E_{3\text{-}1})$$
$$= e_2 \cup (e_{12} \cup e_2 \cdot e_1)* \bowtie e_2 \cup (e_5 \cup e_2 \cdot e_4) \bowtie e_3.$$

Proposition 3 After the execution of *phase*-2, $E_{\alpha\text{-}r}$ is an expression representing exactly the paths from α to r.

Proof. In terms of Proposition 2, $E_{\alpha\text{-}r}$ generated by *phase*-1 represents only those paths from α to r, which contain no node larger than α or larger than r, depending on whether $\alpha \geq r$ or $\alpha < r$. By *phase*-2, the missing sub-expressions are calculated and included in the final expression. □

IV. General Process

In this section, we describe a general process to transform an XPath expression to a relational algebra expression.

As shown in Section 2, an XPath expression Q, can always be represented as a tree T_Q. Then, given a query tree, what we need to do is to construct an expression from T_Q. Our method works in three steps.

In the first step, we transform each node v of the form //B to a node of the form /E such that E may contain Kleene closures. It is done as follows.

- If B is an attribute name, we will first find the relation name C, in which B appears. Then, find *atParent*(v).*tag* and construct an *αβ*-graph G[*atParent*(v).*tag*, C]. Assume that the expression created for the graph is E. We will replace v with an edge (u, u') such that u = '/E' and u' = '/B' and the children of v become the children of u'.
- If B is a relation name, we will construct an expression E for G[*atParent*(v).*tag*, B] and replace v with '/E'. In this way, we transform T_Q to T_Q' which does not contain any '//'.

In the second step, we mainly handle attributes. For any *at*-node v of the form */tag* with *tag* being an attribute in some relation R, we will find its child u if it exists. We distinguish two cases of u.

- u is a logic node '$\wedge$' or '$\vee$'. In this case, we will find all children of u. Each of them must be a constant node of the form δc, where c is a constant and δ is =, !=, >, >=, < or <=. Let $\delta_1 c_1$, …, $\delta_k c_k$ be the children of u. If u = '$\wedge$', construct a selection operation
  $\sigma_{tag\delta_1 c_1 \wedge \cdots \wedge tag\delta_k c_k}(R)$.
  Otherwise, construct
  $\sigma_{tag\delta_1 c_1 \vee \cdots \vee tag\delta_k c_k}(R)$.
  Substitute it for v.
- u is a constant node δc. Replace v with $\sigma_{tag\delta c}(R)$.

In this way, we transform T_Q' to T_Q'' which does not contain any attribute name.

Now we have only logic nodes and *at*-nodes with relation names in T_Q''. (See Fig. 8 for illustration.)

In the third step, we will search T_Q'' bottom-up. Let v be the node currently encountered. The following operations will be performed.

1. If v is a leaf node, return v.
2. If v is a non-leaf node, it will be checked as follows.

v = '$\vee$': Let v_1, …, v_k be the children of v. Let F_i be the relational algebra expression F_i for $Q[v_i]$ (subtree rooted at v_i) (i = 1, …, k). Return $F_1 \cup \ldots \cup F_k$.

v = '$\wedge$': Let v_1, …, v_k be the children of v. Let F_i be the relational algebra expression F_i for $Q[v_i]$ (i = 1, …, k). Return $F_1 \cap \ldots \cap F_k$.

v = '¬': Return $R - F$, where R is the relation name in its *atParent*, and F is the expression created for its unique child.
v = '/*tag*': Let $v_1, \ldots, v_k$ be the children of v. Let F_i be the relational algebra expression F_i for $Q[v_i]$ ($i = 1, \ldots, k$). If '*tag*' appears in at least one F_i, return $F_1 \bowtie \ldots \bowtie F_k$. Otherwise, return $tag \bowtie F_1 \bowtie \ldots \bowtie F_k$.

Example 4 Applying the above process to the query tree shown in Fig. 3, we will get a tree shown in Fig. 8 after the first two steps.

After the third step, we will get the following expression:

$$\text{dept} \bowtie (\sigma_{title=\text{XML}}(\text{course}) \cup ((\text{course} - \sigma_{year=2008}(\text{course})) \cap \sigma_{prereq=\text{CS2001}}(\text{course})) \bowtie E. \ \square$$

The above expression can be evaluated in any relational database system which supports the LFP operation. But such an expression should be optimized. This can be done by using the standard techniques [26].

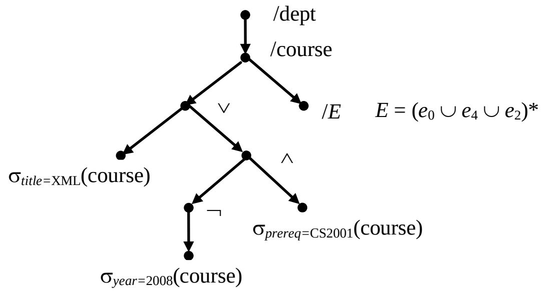


Fig. 8. Illustration for tree transformation

V. Conclusion

In this paper, a new method is proposed for transforming unique parents. For this kind of graphs, not only the expressions can be efficiently generated, but the use of the LFP operators can be minimized. The main idea behind it is to recognize a class of DTD graphs, which can be reduced by contracting nodes into their respective roots. For a non-reducible graph, we divide it into two parts: a reducible part and a non-reducible part, and create expressions for them separately. In this way, the use of the LFP operators can also be dramatically decreased. In addition, a theoretical comparison of our method with Fan's algorithm is conducted, showing that Fan's algorithm is in essence a brute-force algorithm, by which no attention is paid to the structure of DTD graphs. So it cannot be efficient, especially for the reducible graphs.

References

[1] R. Agrawal and P. Devanbu. Moving selections into linear least fixpoint queries. In *ICDE*, 1988.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, 1986.

[3] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Log. Program*, 10, 1991.

[4] BIOML. BIOpolymer Markup Language http://xml.coverpages.org/BIOML-XML-DTD.txt.

[5] Y. Chen, Magic Sets and Stratified Databases, *Int. Journal of Intelligent Systems*, John Wiley & Sons, Ltd., Vol. 12, No. 3, March 1997, pp. 203-231.

[6] Y. Chen, On the Bottom-up Evaluation of Recursive Queries, *Int. Journal of Intelligent Systems*, John Wiley & Sons, Ltd., Vol. 11, No. 10, Oct. 1996, pp. 807-832.

[7] Y. Chen, An Efficient Streaming Algorithm for Evaluating XPath Queries, in: *Proc. of Int. Conf. on Web Information Systems (WEBIST 2008)*, Lisabo, Portual, May 2008, 190-196.

[8] Y. Chen, Tree Embedding and XML Query Evaluation, *Int. Conf. on Enterprise Information Systems (ICEIS-2008)*, IEEE, Funchal-madeira, Barcelona, Spain, June 12-16, 2008, pp. 173-178.

[9] Y. Chen, Document Tree Reconstruction and Fast Twig Pattern Matching, in *Proc: International Conf. on Information and Knowledge Engineering (IKE'09)*, Monte Carlo Resort, Las Vegas, USA, July 13-16, 2009, pp. 393-399.

[10] Y. Chen, Unordered Tree Matching and Tree Pattern Queries in XML Databases, in *Proc: I4th nternational Conf. on Software and Data Technology (ICSOFT'09)*, Sofia, Bulgaria, July 26-29, 2009, pp. 191-198.

[11] Yangjun Chen: A time optimal algorithm for evaluating tree pattern queries. *SAC 2010*: 1638-1642.

[12] B. Choi. What are real DTDs like. In *WebDB*, 2002.

[13] J. Clark and S. DeRose. XML path language (XPath). W3C Working Draft, Nov. 1999.

[14] D. DeHaan, D. Toman, M. Consens, and T. Ozsu. Comprehensive XQuery to SQL translation using dynamic interval encoding. In *SIGMOD*, 2003.

[15] M. Fernandez and D. Suciu. Optimizing regular path expression using graph schemas. In *ICDE*, 1998.

[16] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middleware queries. In *SIGMOD*, 2001.

[17] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull*, 22(3), 1999.

[18] IBM. DB2 XML Extender. http://www 3.ibm.com/software/data/db2/extended/xmlext/index.html.

[19] S. Jain, R. Mahajan, and D. Suciu. Translating XSLT programs to efficient SQL querie. In *WWW*, 2002.

[20] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *SODA*, 2002.

[21] R. Krishnamurthy, V. Chakaravarthy, R. Kaushik, and J. Naughton. Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In *ICDE*, 2004.

[22] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Xsym*, 2003.

[23] R. Krishnamurthy, R. Kaushik, and J. Naughton. Efficient XML-to-SQL query translation: Where to add the intelligence. In *VLDB*, 2004.

[24] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, 2001.

[25] M. Marx. XPath with conditional axis relations. In *EDBT*, 2004.

[26] Microsoft SQLXML and XML mapping technologies. http://msdn.microsoft.com/sqlxml/default.asp.

[27] M. Nunn. An overview of SQL server 2005 for the database developer, 2004. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql ovyukondev.asp.

[28] Oracle. Oracle9i XML Database Developer's Guide – Oracle XML DB Release 2. http://otn.oracle.com/tech/xmldb/content.html.

[29] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient algorithms for multi query optimization. In *SIGMOD*, 2000.

[30] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, 2001.

[31] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. De-Witt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.

[32] W. Fan, J.X. Yu, J. Li, B. Ding and L. Qin, Query Translation from XPath to SQL in the Presence of Recursive DTDs. *The VLDB Journal* (2009) 18:857-883.

[33] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems*, **3rd edition**, Addison-Wesley, 5th edition, 2007.

APPENDIX

In the Appendix, we describe Fan's algorithm [25] and apply it to a simple graph to see how it works. Especially, showing that even for a simple graph the created regular expression can be very large.

In Fan's algorithm, the nodes in a graph are numbered, and a variable $M[i, j, k]$ is used to store the expression representing all paths from node i to node j via nodes whose numbers are less than or equal to k.

Through a nested loop, the algorithm checks all possible values for i, j, and k; and for each combination the value of the corresponding $M[i, j, k]$ is established. Therefore, it is a brute-force algorithm.

```
Algorithm CycleE(G, A, B)
Input: a graph G with n nodes, and two nodes A and B in G.
Output: a regular expression representing all paths from A to B in G.
begin
1.   for $i$ = 1 to $n$ do {
2.     for $j$ = 1 to $n$ do {
3.       if $i = j$
4.       then $M[i, j, 0] \leftarrow \phi$;
5.       else if $i \neq j$ and $(i, j)$ is an edge $e$ in $G$
6.           then $M[i, j, 0] \leftarrow e$;
7.           else $M[i, j, 0] \leftarrow \phi$; }}
8.   for $k$ = 1 to $n$ do {
9.     for = 1 to $n$ do {
10.      for $j$ = 1 to $n$ do {
11.        if $M[i, k, k - 1] \neq \phi$ and $M[k, j, k -1] \neq \phi$
12.        then $M[i, j, k] \leftarrow M[i, j, k - 1] \cup$
                   $M[i, k, k - 1]\cdot M[k, k, k - 1]^*\cdot M[k, j, k - 1]$;
13.        else $M[i, j, k] \leftarrow M[i, j, k - 1]$; }}}
14.  return $M[A, B, n]$;
end
```

In the algorithm, all $M[i, j, 0]$'s are first initialized (lines 1 – 7). Then $M[i, j, k]$'s with $k \geq 1$ are calculated, by inspecting $M[i, j, k - 1]$, $M[i, k, k - 1]$, and $M[k, j, k -1]$, including all possible cycles, i.e., $M[k, k, k - 1]^*$ (lines 8 - 13).

The following example helps for illustration.

Example 5 In this example, we apply the algorithm to the graph shown in Fig. 9, and trace the computation process.

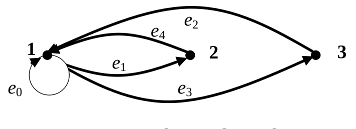


Fig. 9. A directed graph

$\underline{k = 1, i = 1, j = 1.}$
$M[i, k, k\text{-}1] = M[1, 1, 0] = e_0$
$M[k, j, k\text{-}1] = M[1, 1, 0] = e_0$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[1, 1, 1] = M[1, 1, 0] \cup (M[1, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 1, 0])$
$= e_0 \cup (e_0\cdot e_0^*\cdot e_0) = e_0 \cup e_0^*.$
$\underline{k = 1, i = 1, j = 2.}$
$M[i, k, k\text{-}1] = M[1, 1, 0] = e_0$
$M[k, j, k\text{-}1] = M[1, 2, 0] = e_1$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[1, 2, 1] = M[1, 2, 0] \cup (M[1, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 2, 0])$
$= e_1 \cup (e_0\cdot e_0^*\cdot e_1) = e_1 \cup e_0^*\cdot e_1.$
$\underline{k = 1, i = 1, j = 3.}$
$M[i, k, k\text{-}1] = M[1, 1, 0] = e_0$
$M[k, j, k\text{-}1] = M[1, 3, 0] = e_3$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[1, 3, 1] = M[1, 3, 0] \cup (M[1, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 3, 0])$
$= e_3 \cup (e_0\cdot e_0^*\cdot e_3) = e_3 \cup e_0^*\cdot e_3.$
$\underline{k = 1, i = 2, j = 1.}$
$M[i, k, k\text{-}1] = M[2, 1, 0] = e_4$
$M[k, j, k\text{-}1] = M[1, 1, 0] = e_0$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[2, 1, 1] = M[2, 1, 0] \cup (M[2, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 1, 0])$
$= e_4 \cup (e_4\cdot e_0^*\cdot e_0) = e_4 \cup e_4\cdot e_0^*.$
$\underline{k = 1, i = 2, j = 2.}$
$M[i, k, k\text{-}1] = M[2, 1, 0] = e_4$
$M[k, j, k\text{-}1] = M[1, 2, 0] = e_1$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[2, 2, 1] = M[2, 2, 0] \cup (M[2, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 2, 0])$
$= \phi \cup (e_4\cdot e_0^*\cdot e_1) = e_4\cdot e_0^*\cdot e_1.$
$\underline{k = 1, i = 2, j = 3.}$
$M[i, k, k\text{-}1] = M[2, 1, 0] = e_4$
$M[k, j, k\text{-}1] = M[1, 3, 0] = e_3$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[2, 3, 1] = M[2, 3, 0] \cup (M[2, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 3, 0])$
$= \phi \cup (e_4\cdot e_0^*\cdot e_3) = e_4\cdot e_0^*\cdot e_3$
$\underline{k = 1, i = 3, j = 1.}$
$M[i, k, k\text{-}1] = M[2, 1, 0] = e_4$
$M[k, j, k\text{-}1] = M[1, 1, 0] = e_0$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[3, 1, 1] = M[3, 1, 0] \cup (M[3, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 1, 0])$
$= e_4 \cup (e_4\cdot e_0^*\cdot e_0) = e_1 \cup e_4\cdot e_0^*$
$\underline{k = 1, i = 3, j = 2.}$
$M[i, k, k\text{-}1] = M[3, 1, 0] = e_2$
$M[k, j, k\text{-}1] = M[1, 2, 0] = e_1$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[3, 2, 1] = M[3, 2, 0] \cup (M[3, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 2, 0])$
$= \phi \cup (e_2\cdot e_0^*\cdot e_1) = e_2\cdot e_0^*\cdot e_1$
$\underline{k = 1, i = 3, j = 3.}$
$M[i, k, k\text{-}1] = M[3, 1, 0] = e_2$
$M[k, j, k\text{-}1] = M[1, 3, 0] = e_3$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[3, 3, 1] = M[3, 3, 0] \cup (M[3, 1, 0]\cdot M[1, 1, 0]^*\cdot M[1, 3, 0])$
$= \phi \cup (e_2\cdot e_0^*\cdot e_3) = e_2\cdot e_0^*\cdot e_3$
$\underline{k = 2, i = 1, j = 1.}$
$M[i, k, k\text{-}1] = M[1, 2, 1] = e_1 \cup e_0^*\cdot e_1$
$M[k, j, k\text{-}1] = M[2, 1, 1] = e_1 \cup e_4\cdot e_0^*$
$M[i, j, k] = M[i, j, k\text{-}1] \cup (M[i, k, k\text{-}1]\cdot M[k, k, k\text{-}1]^*\cdot M[k, j, k\text{-}1])$
$M[1, 1, 2] = M[1, 1, 1] \cup (M[1, 2, 1]\cdot M[2, 2, 1]^*\cdot M[2, 1, 1])$
$= e_0 \cup e_0^* \cup ((e_1 \cup e_0^*\cdot e_1) \cdot (e_4\cdot e_0^*\cdot e_1)^* \cdot (e_4 \cup e_4\cdot e_0^*)).$ □

From the sample trace, we can see that the expressions produced by Fan's algorithm tend to be large. For instance, the expression representing all paths from node 1 to node 1 is

$$e_0 \cup e_0^* \cup ((e_1 \cup e_0^*\cdot e_1) \cdot (e_4\cdot e_0^*\cdot e_1)^* \cdot (e_4 \cup e_4\cdot e_0^*)).$$

But the minimized regular expression for this is $(e_0 \cup e_4 \cup e_2)^*$, which can be obtained by doing a computation similar to Example 1 since the graph is reducible.