ELSEVIER

The Journal of Systems and Software 64 (2002) 219–233

The Journal of Systems and Software

www.elsevier.com/locate/jss

An effective query pruning technique for multiple regular path expressions

Chang-Won Park, Chin-Wan Chung *

Division of Computer Science, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Kusung-dong, Yusung-gu, Daejeon 305-701, South Korea

Received 22 May 2001; received in revised form 14 September 2001; accepted 3 October 2001

Abstract

Regular path expressions are essential for formulating queries over the semistructured data without specifying the exact structure. The query pruning is an important optimization technique to avoid useless traversals in evaluating regular path expressions. While the previous query pruning optimizes a single regular path expression well, it often fails to fully optimize multiple regular path expressions. Nevertheless, multiple regular path expressions are very frequently used in nontrivial queries, and so an effective optimization technique for them is required. In this paper, we present a new technique called the two-phase query pruning that consists of the preprocessing phase and the pruning phase. Our two-phase query pruning is effective in optimizing multiple regular path expressions, and is more scalable and efficient than the combination of the previous query pruning and post-processing in that it never deals with exponentially many combinations of sub-results produced from all the regular path expressions.
© 2002 Elsevier Science Inc. All rights reserved.

1. Introduction

The semistructured data (Abiteboul, 1997; Abiteboul et al., 2000; Buneman, 1997) has gained a lot of popularity recently in the light of its diverse applications such as managing many new forms of data including XML data, integrating heterogeneous data sources, and managing Web sites. The structure of the semistructured data is irregular, partially known, or subject to frequent changes. Because of such structural properties, regular path expressions are essential for formulating queries without specifying the exact structure. However, the evaluation of regular path expressions traverses many more edges than needed, and hence avoiding the useless traversals is an important optimization. In this respect, the query pruning (Fernandez and Suciu, 1998; McHugh and Widom, 1999a) has been developed to optimize a query with regular path expressions by rewriting the query into another optimized one with more exact regular path expressions using some semistructured graph schema (Abiteboul et al., 2000; Buneman et al., 1997; Goldman and Widom, 1997; Nestorov et al., 1997).

1.1. Related work

Kifer et al. (1992) have introduced the extended path expressions to retrieve information captured by the schemas of object-oriented databases. The extended path expressions support not only retrieving schemas but also retrieving data through arbitrary paths by means of new kinds of variables such as attribute variables and path variables. In addition, the generalized path expressions (Christophides et al., 1994) have been introduced to support retrieving data at various granularity levels from structured documents (e.g., SGML) stored in object-oriented databases by using attribute variables and path variables. Although the two path expressions are similar to the regular path expressions for the semistructured data, they are supplementary elements, and assume conventional rigid schemas defined a priori. In contrast, the regular path expressions are the primary elements of semistructured queries, and do not assume conventional rigid schemas (Abiteboul, 1997; Abiteboul et al., 2000; Abiteboul et al., 1997; Buneman, 1997; Quass et al., 1995).

* Corresponding author.
E-mail addresses: cwpark@islab.kaist.ac.kr (C.-W. Park), chungcw@islab.kaist.ac.kr (C.-W. Chung).

0164-1212/02/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
PII: S0164-1212(02)00046-8

The query pruning is a representative technique for optimizing regular path expressions in the context of the semistructured data. Other optimization techniques based on rewriting regular path expressions using views (Calvanese et al., 1999; Fernandez and Suciu, 1998) have been developed, which are special cases of general query rewriting techniques (Halevy, 2000) for the semistructured data. However, those techniques are applicable only to a single regular path expression (Calvanese et al., 1999; Fernandez and Suciu, 1998), and hence the query pruning is the unique technique for multiple regular path expressions. As another possible approach, we may think of adapting the technique for generalized path expressions (Christophides et al., 1996) to the cost-based optimization technique developed for the semistructured data (McHugh and Widom, 1999b; McHugh and Widom, 1999c). However, the technique assumes conventional rigid schemas, and integrates the schema lookup and the data lookup to apply cost-based optimization techniques to both lookups in a homogeneous fashion. Therefore, it is inadequate for the semistructured data.

1.2. Motivation and our approach

While the previous query pruning optimizes a single regular path expression well, it often fails to fully optimize multiple regular path expressions. This ineffectiveness of the previous query pruning is caused by ignoring correlations among multiple regular path expressions in a query, resulting in schema-level excessive paths which do not contribute to the answer of the query. Worse still, even a single schema-level excessive path may cause far more such paths to be traversed during the evaluation at the data instance level. The previous approach (Fernandez and Suciu, 1998), which augments the query pruning with post-processing, must check an exponential number of combinations of sub-results for all regular path expressions through the post-processing in order to eliminate schema-level excessive paths. Thus, the previous approach runs in exponential time with respect to the number of regular path expressions, and therefore the post-processing has not been adopted (Fernandez and Suciu, 1998).

Nevertheless, multiple regular path expressions are very frequently used in nontrivial queries such as branching path expressions (McHugh and Widom, 1999b). So an effective optimization technique for such nontrivial queries is very important because of their high evaluation cost. In this regard, the primary concern of this paper is to improve the query pruning without compromising the time complexity.

In this paper, we present a new technique called the two-phase query pruning that consists of the preprocessing phase and the pruning phase. The preprocessing phase concatenates correlated regular path expressions to preserve their correlation. A new concept called the least maximized condition set guides the preprocessing. After that, the pruning phase optimizes the preprocessed regular path expressions. Our two-phase query pruning is effective in optimizing multiple regular path expressions, and is more scalable and efficient than the combination of the previous query pruning and the post-processing in that it never deals with exponentially many combinations of sub-results produced from all the regular path expressions. In order to validate our claim, we have implemented a prototype system, and conducted several experiments. The experimental results show that the two-phase query pruning is both effective and scalable unlike the previous approach. Therefore, our contributions are as follows:

- An effective and scalable two-phase query pruning technique for multiple regular path expressions:
  - we propose a new approach to eliminate schema-level excessive paths without the post-processing;
  - we define a new concept of the least maximized condition set for concatenating correlated regular path expressions during the preprocessing;
  - we present a sound and complete preprocessing algorithm, which produces the least maximized condition set;
  - we present a correct and effective pruning algorithm, which properly optimizes the least maximized conditions;
- A set of meaningful experiments using a fully implemented prototype system.

1.3. The organization of the paper

The remainder of this paper is organized as follows. We present preliminaries and a running example to be used throughout the paper in Section 2. Section 3 describes the previous query pruning. In Section 4, we discuss the ineffectiveness of the previous query pruning in some detail, and define two new concepts to cope with the ineffectiveness. On the basis of the concepts, in Section 5, we present the two-phase query pruning. Section 6 describes experiments, and shows experimental results. Finally, we conclude the paper by briefly summarizing main results of the paper in Section 7.

2. Preliminaries and running example

Data. Fig. 1(a) shows a sample data. We can think of labels, depicted as strings without quotation marks, as structural components. Atomic values are only at leaves, and depicted as quoted strings. We omit most of data that are irrelevant to our further discussion. The top-

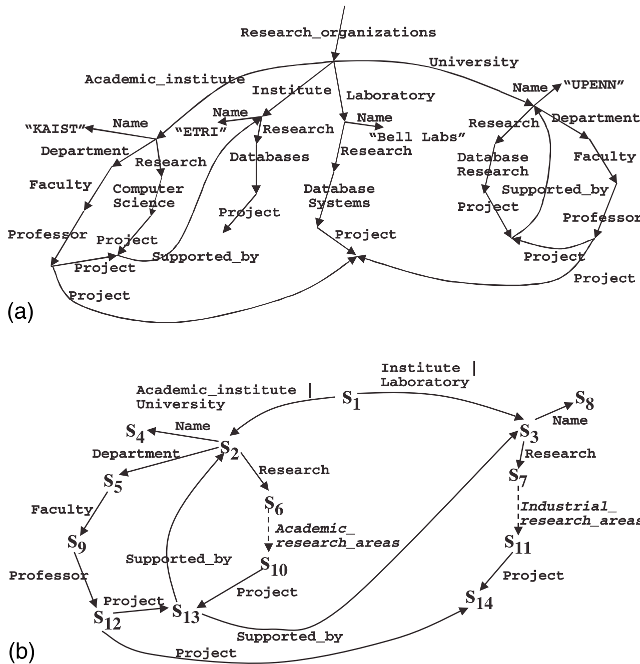

Fig. 1. A sample semistructured data and its schema about research organizations: (a) data graph, (b) schema graph.

most edge `Research_organizations` is not an ordinary edge but a globally named variable leading to the root of the data.

Schema. A schema, to which the data conforms, accompanies the data in Fig. 1(b). The schema is similar to the data except that each edge can be labeled with a set of all possible labels delimited by "|," and that no values are permitted even at leaves. We combine some portions of the original schema for brevity: As an instance, `Academic_research_areas` edge depicted as a dotted line stands for a set of edges each of which denotes a specific research area for each academic research organization. We define the extent for every schema node s, $\text{ext}(s)$, to be the set of all data nodes that are classified as s. For example, $\text{ext}(s_4)$ is {"KAIST", "UPENN"}.

Query. To formulate an example query, we have adopted the query language introduced by Fernandez and Suciu (1998) except using "_" and "$*$" to express any single edge and any sequence of edges, respectively. Consider the query below:

```
select C
 where *.Professor.Project.A in
       Research_organizations,          (1)
       _.B in Research_organizations,   (2)
       Name.C in B,                     (3)
       Research._.Project.A in B,       (4)
       Supported_by.B in A              (5)
```

The condition,[1] $R.\text{var}_1$ in var_2, means a boolean value that is true if there exists a path matched with the path expression R from a data node bound to var_2 to a data node bound to var_1. The order of the conditions in the query is not significant. Let θ be a substitution of variables with data nodes satisfying all conditions in `where` clause. Then the meaning of the query is retrieving all data nodes bound to the variable `C` by finding every θ. Thus the query retrieves names of the research organizations that appear to support their own project in which some professor participates. The answer is "UPENN".

The roles of the variables. We introduce the roles using examples. In Condition (5), `A` and `B` are the source variable and the destination variable of (5), respectively. In Condition (1) and Condition (5), `A` is an intermediate variable, for which `A` is the destination variable of (1) and the source variable of (5). This interaction from (1)–(5) via A is the input/output interaction. In addition, some conditions that can be ordered, for example from (1)–(5), by any sequence of the input/output interactions are ordinal conditions. In Condition (3) and Condition (4), `B` is a branch variable, for which `B` is the source variable of (3) and (4). In Condition (1) and Condition (4), `A` is a join variable, for which `A` is the destination variable of (1) and (4). Intermediate variables, branch variables, and join variables are to bind a unique data node to each occurrence of the variables in every substitution θ. Finally, the variable `C` to be retrieved by the query is the result variable.

3. Query pruning

With multiple conditions given, the query pruning proceeds with the following steps.

Constructing product automata. We consider constructing the product automata (Fernandez and Suciu, 1998) using, for example, "_" in Condition (2) as follows: (1) Construct a nondeterministic automaton A_2 corresponding to "_" as depicted in Fig. 2(a); (2) For the schema S in Fig. 1(b) and A_2, construct the product automaton $S \times A_2$ with 14×2 states, (s_1, a_1), $(s_1, a_2), \ldots, (s_{14}, a_2)$. Transitions are $(s, a) \xrightarrow{l} (s', a')$ for any edge $s \xrightarrow{l} s'$ in S and any transition $a \xrightarrow{l'} a'$ in A_2 if l' is either l or the wildcard. The initial states are states (s, a) such that s is a starting point of matching in S and a is the initial state of A_2: i.e. (s_1, a_1) in this case, since the starting point of matching is always the root s_1 of S. In general, starting points of matching are not fixed, nor are initial states. The terminal states are states (s, a) such that a is a terminal state in A_2. The other product

[1] In the rest of this paper, we use condition or path expression in place of regular path expression, if there is no confusion.

automata for S and the other conditions are constructed identically.

The following step constructs the AND/OR graph (Fernandez and Suciu, 1998), and produces pruned product automata $S \sqcap A_2$ in Fig. 2b, which consists of states and transitions in $S \times A_2$ that are on every valid path from an initial state to an terminal state.

Constructing the AND/OR graph. In the AND/OR graph, an OR node is a node that is accessible if some of its predecessors is accessible, while an AND node is a node that is accessible if all its predecessors are accessible. First, we take the disjoint union of the product automata $S \times A_l$, where all states of them are OR nodes. Second, implicit interactions among the conditions are expressed as follows: (1) Considering some condition $\mathtt{R}.z$ in x in isolation, z may be bound to some node in ext(s) iff (s, a) is in the pruned product automaton $S \sqcap A$ of the condition for at least some terminal state a in A. This is an OR condition, and described by an OR node (s, z, A) created for every schema node s in S and the product automaton $S \times A$ of the condition $\mathtt{R}.z$ in x. In addition, add edges $(s, a) \rightarrow (s, z, A)$ for every terminal state (s, a) in $S \times A$. (2) Considering all conditions $\mathtt{R}_1.z$ in $x_1, \ldots, \mathtt{R}_n.z$ in x_n on a variable z, z may be bound to some node in ext(s) iff it can be bound in each of these conditions. This is an AND condition, and described by an AND node (s, z) created for every variable z and every schema node s in S. Add edges $(s, z, A_i) \rightarrow (s, z)$ for every $S \times A_i$ of the condition $\mathtt{R}_i.z$ in x_i. (3) Input/output interactions of each pair of $\mathtt{R}_1.z$ in x and $\mathtt{R}_2.y$ in z via the intermediate variable z are described by adding edges $(s, z) \rightarrow (s, a)$ for the product automaton $S \times A'$ of $\mathtt{R}_2.y$ in z, where a is the initial state of A'. Finally, the query pruning computes the maximal accessibility property (Fernandez and Suciu, 1998), which is the maximal set of accessible nodes, of the AND/OR graph. The AND node (s, v), for which s is the root node of S and v is a globally named variable, is regarded as always accessible. We define the pruned product automata $S \sqcap A_l$ to consist of accessible OR nodes and transitions that are on every path from an accessible initial OR node to an accessible terminal OR node. All the other nodes in $S \times A_l$ are regarded as inaccessible. Fig. 2(c) shows a simplified version of the resulting AND/OR graph that contains only accessible nodes.

Composing the resulting optimized query. Every accessible AND node (s, z) means that the variable z can be bound to data nodes in ext(s). The query pruning composes the resulting optimized query using all paths from accessible initial OR nodes to accessible terminal OR nodes for each pruned product automaton. For example, since the AND node (s_1, Research_organizations) is accessible in Fig. 2(c), we have "`(Academic_institute | University | Institute | Laboratory)`" instead of "_" using all paths from the accessible initial OR node (s_1, a_1) to the accessible terminal OR nodes (s_2, a_2) and (s_3, a_2) in $S \sqcap A_2$ of Fig. 2(b). We can verify the resulting expression by observing all paths matched with "_" from s_1 to s_2 and s_3 in S of Fig. 1(b).

The query pruning runs in polynomial time (Fernandez and Suciu, 1998) with respect to the number of states in the regular path expressions and the number of schema nodes in the schema, and the resulting optimized query of the example query is as follows:

```
select C
 where (Academic_institute|University).Department.Faculty.Professor.Project.A
         in Research_organizations,                                              (1)
       (Academic_institute|University|Institute|Laboratory).B
          in Research_organizations,                                             (2)
       Name.C in B,                                                              (3)
       Research.(Academic_research_areas|Industrial_research_areas).Project.A in B,  (4)
       Supported_by .B in A                                                      (5)
```

Note that the resulting query is not fully optimized. For instance, sub-graphs of `Institute` and `Laboratory` need to be traversed excessively according to Condition (2). However, we should avoid traversing the sub-graphs because there exist no `Supported_by` edges. In addition, `Industrial_research_areas` edge only for `Institute` and `Laboratory` appears in Condition (4). Such conditions that are not fully optimized cause schema-level excessive paths.

4. Least maximized condition set

We have observed an important ineffectiveness of the query pruning in the previous section. Consider the

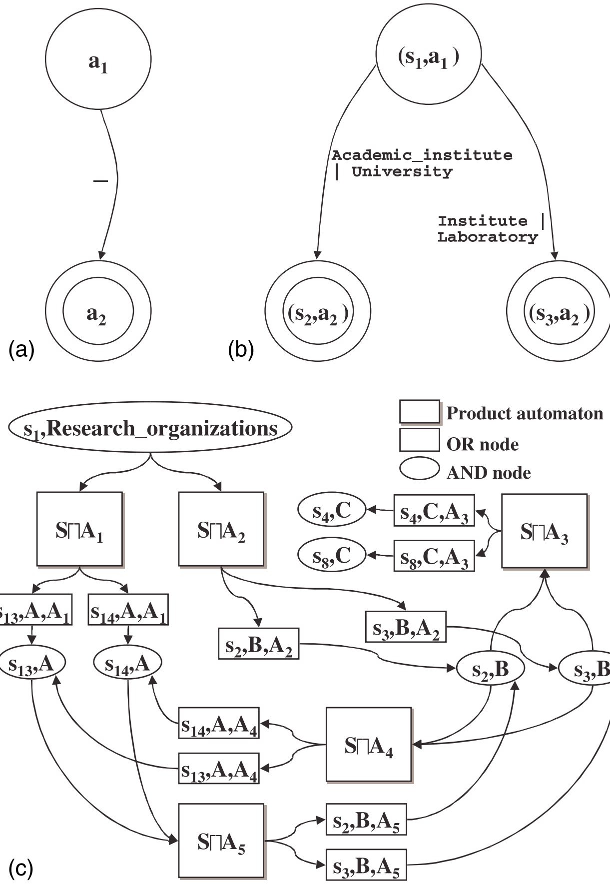

Fig. 2. The automaton A_2 (a) and the pruned product automaton $S \sqcap A_2$ for "_" (b), and the resulting AND/OR graph for the query (c).

pruned product automaton of a certain condition R.y in z in an AND/OR graph. Although multiple AND nodes (s_j, y) can be accessible from the multiple AND nodes (s_i, z) via the pruned product automaton, not all combinations of them are valid. For instance, the AND/OR graph of Fig. 2(c) indicates that (s_{13}, A) and (s_{14}, A) are accessible via $S \sqcap A_4$ of Condition (4), i.e., Research._.Project.A in B, from (s_2, B) and (s_3, B). However, in fact, if the traversal starts from s_2, then ends at s_{13} but not at s_{14} according to Condition (4) and the schema of Fig. 1(b). The ineffectiveness is caused by ignoring such correlations for each condition in some ordinal conditions, resulting in schema-level excessive paths.

The previous approach, which augments the query pruning with the post-processing, must check every combination of respective sub-results for ordinal conditions by means of their product automata in order to eliminate schema-level excessive paths (Fernandez and Suciu, 1998). For example, $S \sqcap A_2$ and $S \sqcap A_4$ are product automata for two ordinal conditions (2) and (4). The post-processing must consider 2^2 combinations of sub-results for them. Let us denote a combination as a triple $[s_i, s_j, s_k]$. The triple corresponds to a possible binding for variables Research_organizations, B, and A. From Fig. 2(c), all possible bindings are $[s_1, s_2, s_{13}]$, $[s_1, s_2, s_{14}]$, $[s_1, s_3, s_{13}]$, and $[s_1, s_3, s_{14}]$. For each possible binding triple, the post-processing checks whether the binding is valid or not by means of the two product automata $S \sqcap A_2$ and $S \sqcap A_4$. Thus, the previous approach runs in exponential time with respect to the number of regular path expressions. For this reason, the post-processing has not been adopted (Fernandez and Suciu, 1998).

In this section, we define a new concept called the least maximized condition set for an alternative approach based on preprocessing. The objective of defining the least maximized condition set is to preserve the correlations by building a single product automaton for each group of ordinal conditions instead of using individual product automata. In this respect, we present two rules to concatenate conditions in a group of ordinal conditions into a longer one. In the rules, R_i and v_i are an arbitrary path expression and a variable, respectively.

Rule 1 $R_1.v_0$ in v_1, $R_2.v_2$ in $v_0 \equiv R_1.R_2.v_2$ in v_1
Rule 2 $R_1.v_0$ in v_1, $R_2.v_2$ in $v_0 \equiv R_1 \overset{v_0}{.} R_2.v_2$ in v_1

Let $\mathscr{D}$ be a set of all sub-sets of data nodes in a given data, and P be a set of all path expressions in a given query. Suppose we have a function $t : \mathscr{D} \times P \mapsto \mathscr{D}$ such that t determines which unique set $D_d \in \mathscr{D}$ is to be produced by evaluating a path expression $p \in P$ from a set $D_s \in \mathscr{D}$. The rules are correct because $t(t(D_s, R_1), R_2) \equiv t(D_s, R_1.R_2)$. However, we must take care in removing an intermediate variable after concatenating a pair of path expressions as in Rule 1, because the intermediate variable may have other roles as a branch variable, a join variable, and a result variable. In such cases, we preserve that kind of variable bindings using a new internal construct "variable annotation" denoted by "$\overset{v_0}{.}$" in Rule 2.

We can apply the rules to arbitrary overlapping groups of ordinal conditions in a condition set. The resulting condition set is equivalent to the original condition set if the overlapping groups cover all conditions in the original set, since every substitution θ satisfying the overlapping groups also satisfy the original condition set, and vice versa. In addition, ordinal conditions in each of the overlapping groups can be concatenated in an arbitrary repeating order if they form a cycle by some sequence of input/output interactions. Thus, we have infinitely many equivalent condition sets from such cyclic conditions.

For example, consider a set of cyclic conditions {R1.x in DB, R2.y in x, R3.x in y, R4.z in y}. Four examples among infinitely many equivalent condition sets are:

(1) $\{R_1\stackrel{x}{.}R_2\stackrel{y}{.}R_3.x \text{ in } DB, R_1\stackrel{x}{.}R_2\stackrel{y}{.}R_4.z \text{ in } DB\}$
(2) $\{R_1\stackrel{x}{.}R_2\stackrel{y}{.}R_3\stackrel{x}{.}R_2\stackrel{y}{.}R_3.x \text{ in } DB, R_1\stackrel{x}{.}R_2\stackrel{y}{.}R_4.z \text{ in } DB\}$
(3) $\{R_1\stackrel{x}{.}R_2\stackrel{y}{.}R_3\stackrel{x}{.}R_2\stackrel{y}{.}R_4.z \text{ in } DB\}$
(4) $\{R_1\stackrel{x}{.}R_2\stackrel{y}{.}R_3\stackrel{x}{.}R_2\stackrel{y}{.}R_3\stackrel{x}{.}R_2\stackrel{y}{.}R_4.z \text{ in } DB\}$

Among all possible condition sets, we need to determine which is the most desirable one. In the case of (2) and (4), regarding "$\stackrel{x}{.}$" as either "x." or ".x", each of them has a redundant sub-expression ($x.R_2\stackrel{y}{.}R_3.x$). We should avoid such redundancy, in general, because the longer a path expression is in length, the costlier the path expression is in optimizing. Between (1) and (3), we prefer (1) because (3) also has a redundant sub-expression ($x.R_2.y$). The condition set like (1) is called the least maximized condition set, and defined below. Given a query, let a starting variable be a variable used only as source variables in the conditions of the query. Conversely, let a terminating variable be a variable used only as destination variables in the conditions of the query.

Definition 1 (*Least maximized condition*). A least maximized condition is a condition produced by concatenating ordinal conditions in a query using Rule 1 and Rule 2 such that

- the source variable of the condition is a starting variable;
- all variables except the destination variable of the condition are distinct;
- the destination variable of the condition is either a terminating variable if all variables in the condition are distinct, or the second occurrence of a variable otherwise.

Definition 2 (*Least maximized condition set*). The least maximized condition set for a query is the set contains every least maximized condition in the query.

The concept of the individual branch defined by McHugh and Widom (1999b) is similar to the concept of the least maximized condition, but their concept differs from ours in the following aspects: (1) The individual branch considers strictly tree-shaped path expressions only, and (2) each individual branch does not overlap with any other individual branches.

5. Two-phase query pruning

The two-phase query pruning consists of the preprocessing phase and the pruning phase. When a query is provided, the preprocessing phase produces the least maximized condition set for the query, and then the pruning phase optimizes the least maximized condition set.

5.1. preprocessing phase

The preprocessing phase converts a condition set into a graph structure, called the interaction graph. Then, it seeks every least maximized condition by performing the depth first search (DFS) over the graph.

5.1.1. Constructing the interaction graph

(1) Let the source variable s and the destination variable d of each condition be two nodes, and create a directed edge from s to d; (2) let the path expression p of the condition be the label of the directed edge; (3) let every occurrence of the same variable be a unique node. During this construction, we collect additional information about each variable to facilitate determining the roles of each variable. In the case of the result variable, we store the fact at the corresponding node. In addition, we also store the in-degree and the out-degree of each variable at the corresponding node. The in-degree of a variable is the number of conditions that use the variable as a destination variable. Similarly, the out-degree of a variable is the number of conditions that use the variable as a source variable. Using such information, we can determine the roles of a variable as in Table 1. It takes constant time.

5.1.2. The DFS-based preprocessing algorithm

The algorithm finds every simple path from each starting variable in an interaction graph by performing DFS. Here, a simple path is a path in which (1) the last variable is a terminating variable, and all variables are distinct, if there is no cycle; and (2) if there is a cycle, the second occurrence of a variable terminates the simple path. In case (2), the other variables are distinct. The following lemma states that a simple path from a starting variable corresponds to a least maximized condition.

Let $c_1, c_2, \ldots, c_n$ be an ordered sequence of ordinal conditions in a query Q. They form a concatenated condition c_{1n} by concatenating them according to Rule 1 and Rule 2. We define a variable path $v_{1s}, v_{1d}, v_{2d}, \ldots, v_{nd}$ to denote a sequence of variables appearing in the ordinal conditions according to the sequence of conditions. Here, v_{is} and v_{id} are the source and the destination variable of the ith condition, respectively. We omit every source variable except the first one since every v_{id} equals

Table 1
Determining the roles of a variable

In-degree	Out-degree	Role
=0	$\geqslant 1$	Starting variable
$\geqslant 1$	$\geqslant 1$	Intermediate variable
$\geqslant 0$	$\geqslant 2$	Branch variable
$\geqslant 2$	$\geqslant 0$	Join variable
$\geqslant 1$	=0	Terminating variable

to $v_{(i+1)s}$. A path in the interaction graph of Q is also called as a variable path, because each variable is a unique node in the interaction graph.

Lemma 1. *The concatenated condition c_{1n} is a least maximized condition iff the variable path in $c_1, c_2, \ldots, c_n$ is a simple path from a starting variable in the interaction graph of Q.*

Proof. Suppose c_{1n} is a least maximized condition, but the variable path $v_{1s}, v_{1d}, \ldots, v_{nd}$ is not a simple path from a starting variable. This means one of the following cases:

Case 1. The variable path contains either three or more identical variables or two or more pairs of identical variables;

Case 2. We can add adjacent nodes to the variable path without violating the definition of a simple path.

Case 1 contradicts that c_{1n} is a least maximized condition. Our process to produce a least maximized condition concatenates no more conditions if it sees the second occurrence of a variable. The second occurrence indicates a cycle. Hence c_{1n} never contains three or more occurrences of the same variable and two or more cycles. However, Case 1 means that c_{1n} does.

Case 2 also contradicts that c_{1n} is a least maximized condition. If c_{1n} is a least maximized condition, no incoming edges to v_{1s} exist because v_{1s} should be a starting variable. In addition, no outgoing edges from v_{nd} exist when v_{nd} is a terminating variable and every variable in the variable path is distinct. Thus we are not able to find adjacent nodes for the variable path to add in front of v_{1s} or behind v_{nd}. On the other hand, when v_{nd} is the second occurrence of a variable, no other adjacent nodes of v_{nd} can be added to the variable path without violating the definition of a simple path. Note that no other variables can be inserted in the middle of the variable path, because the conditions follow one another in the sequence. Therefore Case 2 indicates that c_{1n} cannot be a least maximized condition.

Consequently, the variable path is a simple path from a starting variable.

To prove the converse, suppose the variable path is a simple path from a starting variable, but the condition c_{1n} is not a least maximized condition. This means that v_{nd} is not a terminating variable and not the second occurrence of a variable either. The two cases contradicts that the variable path is a simple path from a starting variable. Thus the condition c_{1n} is a least maximized condition. □

The algorithm consists of two component algorithms, and is given in Fig. 3. Algorithm *Preprocess* is the main algorithm, and initiates DFS by calling Algorithm *Concatenate* to get every simple path to some terminating variable for each starting variable. The meaning of each sub-routine should be clear by its name. In particular, $Append(\emptyset, e)$ means appending a path expression e to the end of an empty path expression. $Annotate(p, d)$ creates a variable annotation d at the end of the path expression p.

Algorithm *Preprocess* is sound and complete as stated in Theorem 1.

Theorem 1. *A condition set C_q is the least maximized condition set for a given query Q iff C_q is the resulting condition set produced by Algorithm Preprocess from the query Q.*

Proof. We first show that Algorithm *Preprocess* produces every simple path from each starting variable in the interaction graph of Q. The algorithm starts with finding a starting variable. Since every data is rooted, all queries should access the root via a globally named

```
Algorithm Preprocess
Input: an interaction graph G
Output: the least maximized condition set
Begin
  clear visited flags for all variables
  for each variable v in G do
    if v is a starting variable
    then
      for each edge e of v and its destination d_e do
        Concatenate(v, Append(∅, e), d_e);
      end for
    end if
  end for
End
```

```
Algorithm Concatenate
Input: a source s, a path expression p, a destination d
Output: a least maximized condition (s, p, d)
Begin
  if d is a terminating variable or visited
  then
    Output(s, p, d);
  else
    Mark_visited(d);
    if d is a branch variable or a join variable or the result variable
    then
      Annotate(p, d); // Rule 2
    end if
    for each incident edge e and its destination d_e of d do
      Concatenate(s, Append(p, e), d_e); // Rule 1
    end for
    Clear_visited(d);
  end if
End
```

Fig. 3. The DFS-based preprocessing algorithm.

variable. This guarantees that there is at least one starting variable. Hence, every interaction graph contains one or more starting variables. After finding a starting variable, the algorithm initiates searching for every simple path by invoking Algorithm *Concatenate*. Algorithm *Concatenate* finds all possible simple paths from the starting variable by recursion. If the recursion terminates, we have a set of all simple paths that begin at the starting variable. Algorithm *Preprocess* proceeds with another starting variable. By Lemma 1, the resulting condition set C_q is the least maximized condition set for Q.

Conversely, if C_q is the least maximized condition set, each condition in C_q is a least maximized condition. By Lemma 1, the least maximized condition is a simple path in the interaction graph of Q. Therefore each least maximized condition in C_q must be generated by Algorithm *Preprocess*, because the algorithm finds every simple path from each starting variable in the interaction graph. □

Finally, it is noticeable that our algorithm can treat cyclic queries in the same way as linear queries, since it concatenates ordinal conditions until it either visits a terminating variable or revisits a variable already visited. The latter case corresponds to cyclic queries. In the sequel, the pruning phase, to be discussed later in this section, optimizes the resulting queries properly. We illustrate this with the running example later on.

5.1.3. The time complexity

Given n conditions, we need to check the other $n-1$ conditions for each condition in order to build the interaction graph. Thus, the time complexity to build the interaction graph and to store the additional information is $O(n^2)$. In addition, in order to produce the least maximized condition set, we traverse every directed edge by DFS ($O(n)$) for each starting variable. Since the number of distinct variables is proportional to n, the time complexity to produce the least maximized condition set is $O(n^2)$ as well. Consequently, the total time complexity is $O(n^2)$.

5.1.4. The example

The interaction graph for the example query is given in Fig. 4. The preprocessing phase seeks every least maximized condition in the graph, and produces four least maximized conditions as follows:

(1) `*.Professor.Project.`^{A}`Supported_by.`^{B}`Research._.Project.A in Research_organizations`
(2) `*.Professor.Project.`^{A}`Supported_by.`^{B}`Name.C in Research_organizations`
(3) `_.`^{B}`Research._.Project.`^{A}`Supported_by.B in Research_organizations`
(4) `_.`^{B}`Name.C in Research_organizations`

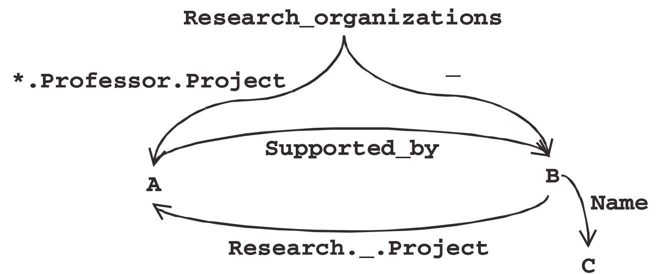

Fig. 4. The interaction graph for the example query.

5.2. Pruning phase

The pruning phase is an extended version of the previous query pruning. Since the core mechanism is not modified, we present the extensions to process the variable annotation only.

5.2.1. The extensions in constructing product automata

A dot in a condition corresponds to a state in the nondeterministic automaton for the condition. Thus, if a dot accompanies a variable annotation, the corresponding state is annotated with the variable. The variable annotation of a state is preserved in the product automaton. Formally, if the state a in the automaton A is annotated with the variable v, the corresponding states (s, a) in the product automaton $S \times A$ are annotated with v as well. We use a superscript to denote the annotated variable v as in a^v and (s, a^v).

5.2.2. The extensions in constructing the AND/OR graph

Considering all states (s_i, a_j^v) annotated with the common variable v, v may be bound to some node in $\mathrm{ext}(s_i)$ iff it can be bound to each of these states. This AND condition is processed using the AND node (s_i, v) in the AND/OR graph. In addition, consider each state (s_i, a_j^v) in a certain product automaton $S \times A_l$. All paths from (s_i, a_j^v) can continue in $S \times A_l$ only when (s_i, v) is accessible in the AND/OR graph. To process the AND condition and the continuance condition, the pruning phase creates an AND node (s_i, a_j, v) for each (s_i, a_j^v), and connects them as follows: (1) all transitions of (s_i, a_j^v) are transferred to (s_i, a_j, v); (2) add an edge $(s_i, a_j^v) \rightarrow (s_i, a_j, v)$; (3) add an edge pair $(s_i, a_j, v) \rightarrow (s_i, v)$ and $(s_i, v) \rightarrow (s_i, a_j, v)$ for (s_i, a_j, v) and the AND node (s_i, v). This edge pair can be connected directly since (s_i, a_j^v) is unique within $S \times A_l$ for s_i in S and a_j^v in A_l, and so is (s_i, a_j, v).

The pruning phase computes the maximal accessibility property as before, and produces pruned product automata. When (s_i, a_j, v) is not on a path from the accessible initial state to some accessible terminal state in $\mathrm{S} \times \mathrm{A}_l$ (s_i, a_j, v) becomes inaccessible in $\mathrm{S} \sqcap \mathrm{A}_l$. When (s_i, a_j, v) is inaccessible, (s_i, v) is also inaccessible, and vice versa. In contrast, the previous query pruning ignores both the AND condition and the continuance condition for the states (s_i, a_j^v). Instead, it uses a certain individual product automaton $\mathrm{S} \times \mathrm{A}_k$ for some original condition, whose destination variable is v, of the given query and the OR node (s_i, v, A_k) for v and $\mathrm{S} \times \mathrm{A}_k$.

5.2.3. The extensions in composing the resulting optimized query

For every pruned product automaton $\mathrm{S} \sqcap \mathrm{A}_l$, convert each path from the accessible initial state to an accessible terminal state into a set of conditions each of which is equivalent to a sub-path delimited by accessible AND nodes (s, a, v) on the path. After that, eliminate all duplicate conditions, and combine all conditions from a common source variable to a common destination variable using the set of labels delimited by "|".

5.2.4. The time complexity

The variable annotations are used for a limited number of states in all the product automata of the AND/OR graph. Thus, considering additional AND nodes (s, a, v) introduced by the variable annotations, the time complexity of the previous query pruning is not compromised by the extensions. Consequently, the pruning step runs in polynomial time with respect to the number of states in the regular path expressions and the number of schema nodes in the schema.

5.2.5. The example

Recall the four least maximized conditions for the example query at the end of the previous subsection. We depict the pruned product automata of Condition (3) and Condition (4) with some related portions of the AND/OR graph in Fig. 5. All nodes of dotted lines are inaccessible, whereas the others are accessible. Unlike the AND/OR graph in Fig. 2(c), since (s_3, a_2, B) and $(s_{14}, a_5, \mathrm{A})$ of Condition (3) are inaccessible, so are (s_3, B) and (s_{14}, A). In addition, (s_8, C) is inaccessible because (s_3, B) and hence (s_3, a_2, B) of Condition (4) are inaccessible. Thus, the resulting optimized query is as follows:

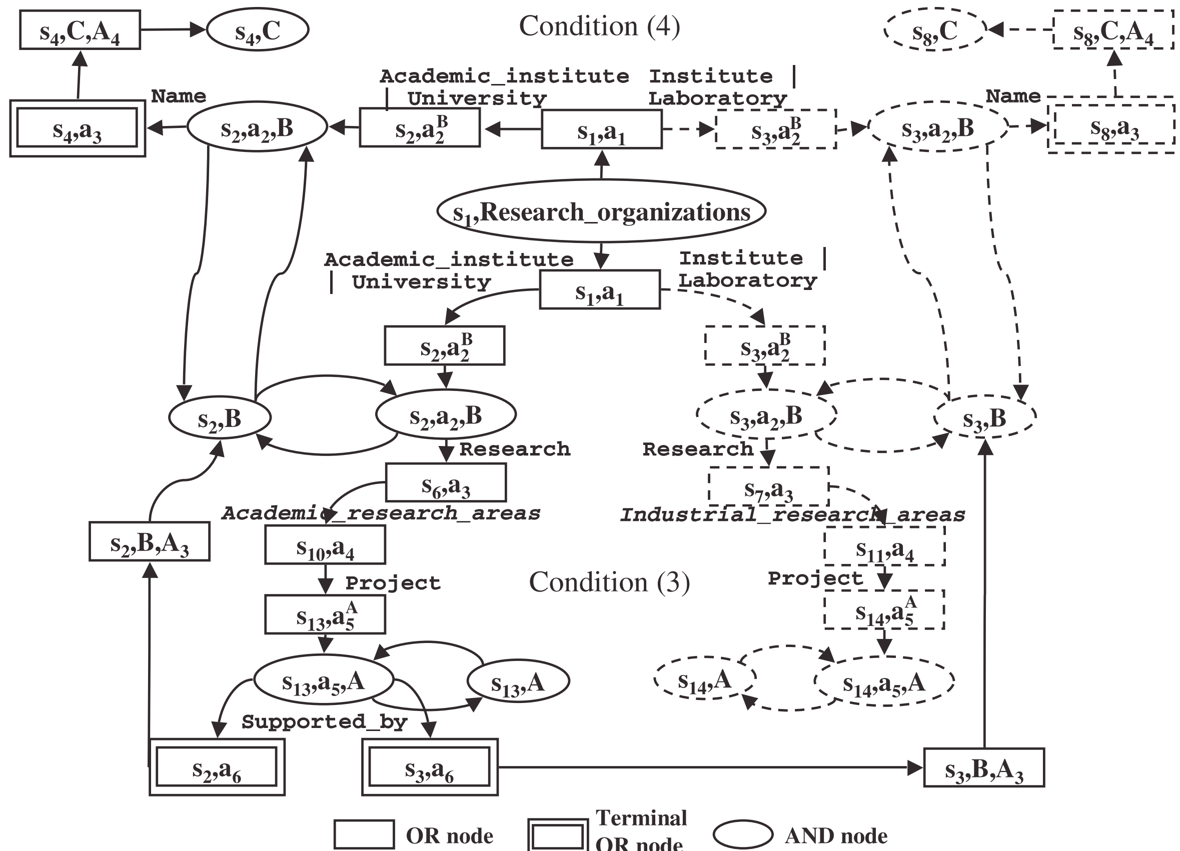

Fig. 5. A part of the resulting AND/OR graph for the example query.

```
select C
 where (Academic_institute | University).Department.Faculty.Professor.Project.A
       in Research_organizations,                                              (1)
       (Academic_institute | University).B in Research_organizations,         (2)
       Name.C in B,                                                            (3)
       Research.Academic_research_areas.Project.A in B,                       (4)
       Supported_by.B in A                                                     (5)
```

Note that this query is fully optimized. If the edges labeled as `Institute` or `Laboratory` are encountered during evaluation, their sub-graphs are never traversed. In addition, `Industrial_research_areas` edge denoting the research areas only for `Institute` or `Laboratory` does not appear in Condition (4).

5.2.6. Correctness and effectiveness

Let $\theta(v)$ be a data node that substitutes for the variable v in the substitution θ. The pruning phase discovers which AND nodes (s_i, v) are accessible in the AND/OR graph G for any query Q, giving us which data nodes $\in \text{ext}(s_i)$ possibly substitute for v. Thus, the correctness of the pruning phase can be stated as follows. For any data D conforming to the schema S and any variable v in Q, the AND node (s_i, v) in G is accessible if there exists a substitution θ of Q's variables such that $\theta(v) \in \text{ext}(s_i)$ for some schema node s_i in S. On the other hand, the converse means the effectiveness that the pruning phase discovers valid substitutions only.

Theorem 2. *For any query Q and schema S, let Q^S be the optimized query produced by the two-phase query pruning with respect to S. Q^S is equivalent to Q over S, and contains no schema-level excessive paths.*

Proof. We show that, for any data D conforming to S and any variable v in Q, the AND node (s_i, v) in the AND/OR graph G for Q is accessible iff there exists a substitution θ of Q's variables such that $\theta(v) \in \text{ext}(s_i)$ for some schema node s_i in S.

Firstly, we prove the if part by constructing an accessibility property A, a set of accessible nodes, of G based on θ. A consists of (1) all AND nodes (s_i, v) for which $\theta(v) \in \text{ext}(s_i)$, (2) all OR nodes (s_i, v, A_l) for which $\theta(v) \in \text{ext}(s_i)$, (3) all states (s_i, a), and AND nodes (s_i, a, v) if v is annotated at (s_i, a), in the pruned product automaton $\text{S} \sqcap \text{A}_l$ of the condition $\text{R}_l.v_{l_2}$ in v_{l_1}, for which there exists the path accepted by A_l from $\theta(v_{l_1})$ to $\theta(v_{l_2})$ in D such that an intermediate data node $x \in \text{ext}(s_i)$ on the path corresponds to the state a of A_l. Since θ is a substitution which satisfies all conditions in Q, A is indeed an accessibility property. Thus, if $\theta(v) \in \text{ext}(s_i)$, then (s_i, v) is accessible.

Secondly, we prove the only if part by contradiction. Assume $\theta(v) \neq \text{ext}(s_i)$ for any θ. Consider the original condition $R_l.v$ in v_s of Q and the automaton A_l for the condition. Because of the assumption, A_l does not accept any path from $\theta(v_s)$ to some data node $x \in \text{ext}(s_i)$. After preprocessing, the condition is incorporated into at least a least maximized condition. Hence some part of the automaton A_L for the least maximized condition corresponds to A_l. We denote the part as $\text{A}_L : \text{A}_l$, and denote its terminal state as a_l. Consider the product automaton $\text{S} \times \text{A}_L$. If the terminal state of the A_L is that of $\text{A}_L : \text{A}_l$, the OR node (s_i, v, A_L) and the AND node (s_i, v) are inaccessible since $\text{A}_L : \text{A}_l$ does not accept any path to (s_i, a_l). Otherwise, all paths that pass through (s_i, a_l) become inaccessible by $\text{A}_L : \text{A}_l$. If v is annotated at a_l of $\text{A}_L : \text{A}_l$, the AND node (s_i, a_l, v) and (s_i, v) are inaccessible since $\text{A}_L : \text{A}_l$ does not accept any path to (s_i, a_l^v). These contradict the sufficient condition that (s_i, v) is accessible. Thus, if (s_i, v) is accessible, $\theta(v) \in \text{ext}(s_i)$. □

6. Experiments

We have implemented, in Java, a prototype system capable of performing both the previous query pruning and the two-phase query pruning. Using the prototype, we have conducted several meaningful experiments. For the experiments, we have used three kinds of typical schemas, which are linear schemas, branching schemas, and cyclic/joining schemas. All the experiments are performed on a Pentium III PC (700 MHz) with 192 MB of memory.

6.1. Schemas and queries

The shape of the linear schemas is depicted in Fig. 6(a). Experiments are conducted with various values for `n` and `b`. The parameters, `n` and `b`, mean the schema depth and the number of outgoing branches for the node s[0] (i.e., the root pointed to by the globally

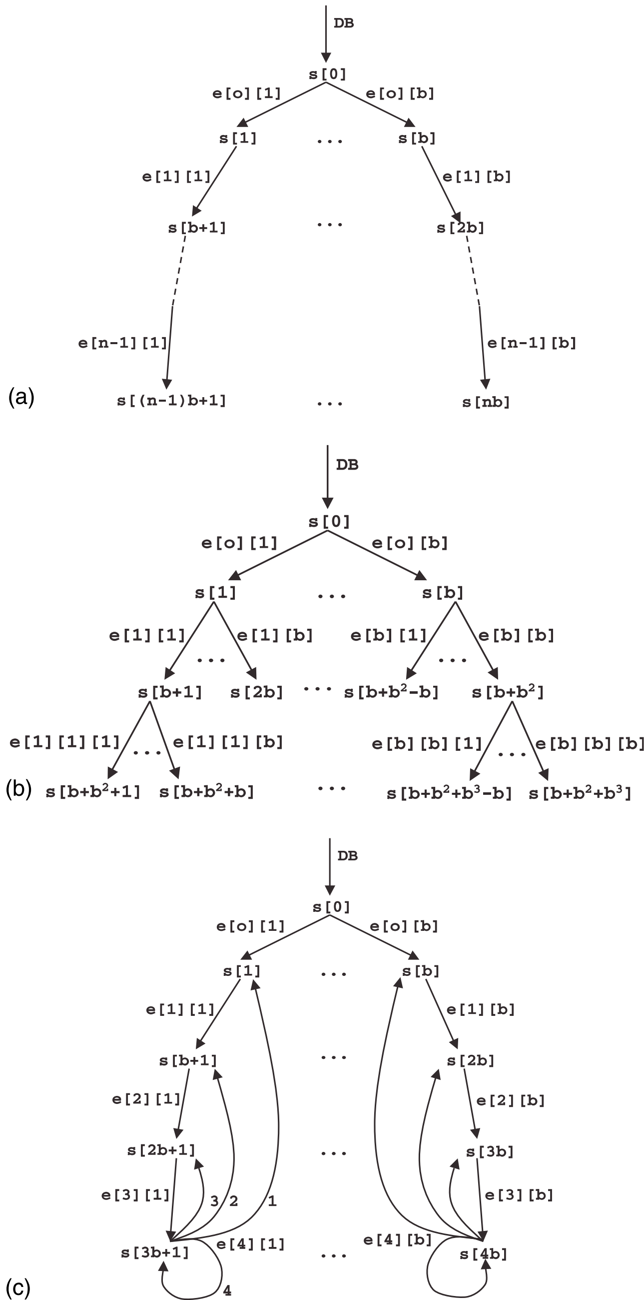

Fig. 6. Three kinds of typical schemas: (a) shape of the linear schemas, (b) shape of the branching schema, (c) shape of the cyclic/joining schemas.

named variable DB), respectively. The schema depth is the number of levels in the schema counted from the root whose depth is 0. The bigger the values for b and n are, the bigger the size of a linear schema is. In the experiments, we have used values 5, 10, and 15 for b. For n, we have used values 2, 3, 4, and 5. The path expressions of queries used in the experiments are listed in Table 2, and the number of path expressions of linear queries are identical to the schema depth. These experiments will show the effect of both the number of path expressions and the size of schema.

The branching schema in the shape depicted in Fig. 6(b) has fixed values 15 and 3 for b and n, respectively. In this schema, b means the number of outgoing branches for each node. Therefore, it has many more nodes than the linear schema with the same values for b and n has. We experiments with various branching queries of different branching depths to show the effect of the different branching depths. As listed in Table 2, differentiating the branching depth naturally results in different numbers of path expressions. The numbers of path expressions in the queries are 6, 5, and 4, for each branching depth of 0, 1, and 2, respectively.

Fig. 6(c) shows the shape of the cyclic/joining schemas. The schemas have different joining depths with fixed values 15 for b and 5 for n. In Fig. 6(c), we depict the different joining depths using four alternative edges pointing to different nodes according to each joining depths in lieu of single deepest edges for the nodes at depth 4. Furthermore, we number the four alternative edges with their respective joining depths for the edge e[4][1]. We experiment with various cyclic/joining queries (listed in Table 2) on the schemas to show the effect of the various joining depths. These experiments will show the effect of both cyclic join and its joining depth.

Finally, we conduct an experiment on the running example as well.

6.2. Experimental results

The experimental results on linear schemas and queries are summarized in Fig. 7. Fig. 7(a) shows that total execution time for the previous query pruning (denoted by org) grows exponentially with respect to the number of path expressions. In contrast, total execution time for the two-phase query pruning (denoted by 2P) does not exceed 100 ms for all cases. This tendency is not changed by various b values, but becomes conspicuous by bigger values.

Fig. 7(b) shows the total number of node accesses during computing maximal accessibility property. This indicates that the total number of node accesses grows according to the growth of both number of path expressions and schema size, but never grows exponentially. Furthermore, it is noticeable that the pruning phase of the two-phase query pruning (denoted by 2P) runs more efficiently than the previous one. The reason is that the two-phase query pruning computes maximal accessibility property earlier. Maximal accessibility property is computed by finding new inaccessible nodes from accessible nodes iteratively until no more change. Initially, all nodes in the AND/OR graph are accessible. In the computation, the two-phase query pruning requires fewer iterations than the previous query pruning.

Table 2
The queries used in the experiments

Type	Name	Schema depth	Path expressions
Linear	Ql2	2	`_.X1 in DB, e[1][1].X2 in X1`
	Ql3	3	`_.X1 in DB, _.X2 in X1, e[2][1].X3 in X2`
	Ql4	4	`_.X1 in DB, _.X2 in X1, _.X3 in X2, e[3][1].X4 in X3`
	Ql5	5	`_.X1 in DB, _.X2 in X1, _.X3 in X2, _.X4 in X3, e[4][1].X5 in X4`
		Branching depth	
Branching	Qb0	0	`_.X1 in DB, _.X12 in X1, e[1][1][1].X13 in X12`
			`_.X2 in DB, _.X22 in X2, e[15][15][15].X23 in X22`
	Qb1	1	`_.X1 in DB, _.X12 in X1, e[1][1][1].X13 in X12`
			`_.X22 in X1, e[1][15][15].X23 in X22`
	Qb2	2	`_.X1 in DB, _.X12 in X1, e[1][1][1].X13 in X12`
			`e[1][1][15].X23 in X12`
		Joining depth	
Cyclic/joining	Qj1	1	`_.X1 in DB, _.X2 in X1, _.X3 in X2, _.X4 in X3, e[4][1].X1 in X4`
	Qj2	2	`_.X1 in DB, _.X2 in X1, _.X3 in X2, _.X4 in X3, e[4][1].X2 in X4`
	Qj3	3	`_.X1 in DB, _.X2 in X1, _.X3 in X2, _.X4 in X3, e[4][1].X3 in X4`
	Qj4	4	`_.X1 in DB, _.X2 in X1, _.X3 in X2, _.X4 in X3, e[4][1].X4 in X4`

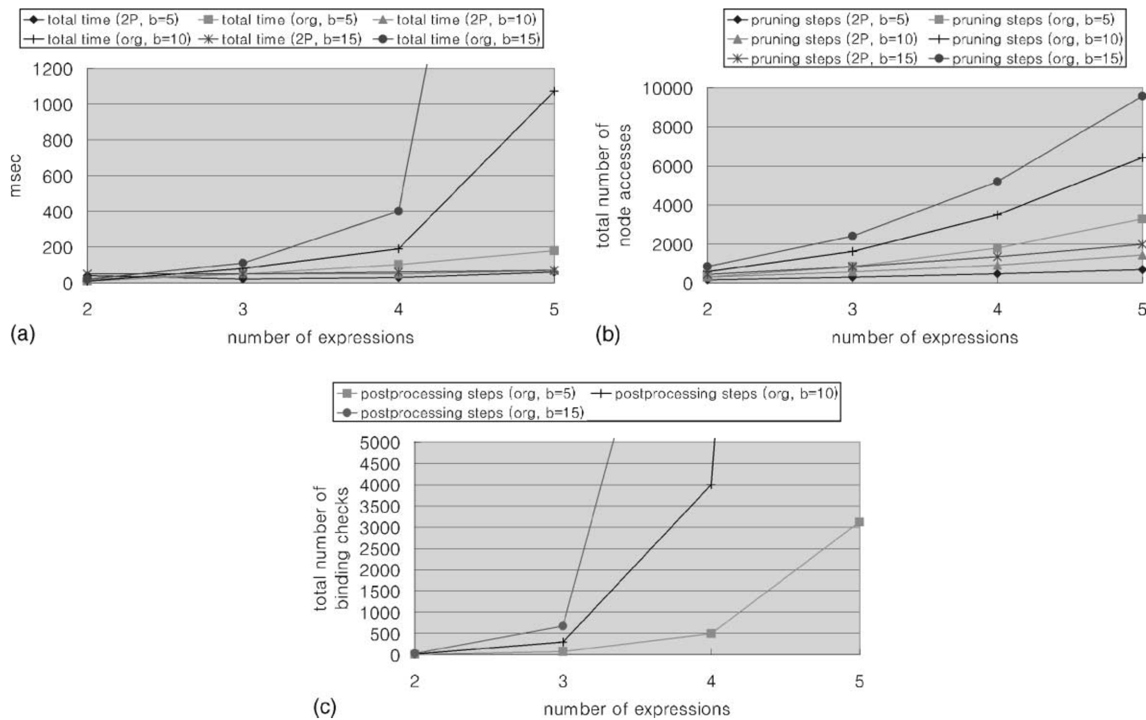

Fig. 7. The experimental results on linear schemas and queries: (a) total execution time, (b) total number of node accesses during pruning, (c) total number of binding checks during the post-processing.

According to our experimental results, while the previous query pruning requires six iterations, the two-phase query pruning requires only two iterations when `n` value is 5, regardless of `b` values.

Consequently, the origin of the exponential growth is the exponential combinations of sub-results. The previous query pruning has to check all the combinations using product automata through the post-processing, resulting in exponential growth as depicted in Fig. 7(c). In contrast, according to the experiments, the preprocessing phase accesses each path expression twice. That is, if `n` is 4, only eight accesses are needed to compute the least maximized condition set. This tendency is not changed even if we apply the linear queries to other schemas such as branching schemas and cyclic/joining schemas.

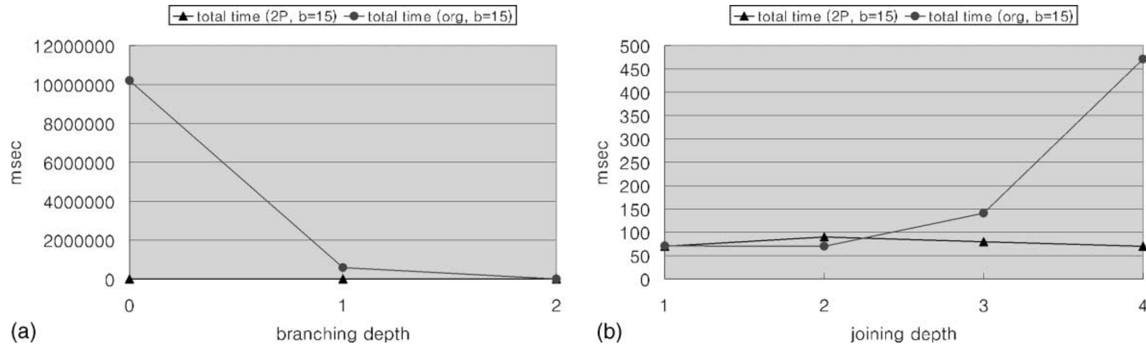

Fig. 8. The experimental results on branching depths and joining depths: (a) total execution time for branching queries, (b) total execution time for cyclic/joining queries.

The experimental results on the effect of different branching depths are given in Fig. 8(a). As we have already mentioned, deeper branching depths can be expressed with fewer path expressions. Thus, the total execution time of the previous query pruning grows exponentially as branching depth approaches to depth 0. This is also due to the exponential combinations of sub-results. In contrast, the total execution time of the two-phase query pruning is less than 2854 ms for all branching depths.

In the case of cyclic/joining queries, the total execution time of the previous query pruning approaches to that of the two-phase query pruning as the joining depth goes up to depth 1. This is due to the cyclic join. For instance, in the case of query Qj1, the number of sub-results (accessible AND nodes in the AND/OR graph) for the join variable `X1` decreases from 15 to 1 since only the single AND node (`s[1], X1`) remains accessible after pruning the last path expression (`e[4][1].X1 in X4`). In addition, this also reduces the number of sub-results further during the next iteration. According to the experimental results, the previous query pruning completes its execution within 71 ms when joining depth is less than 2. This result is better than that of the two-phase query pruning (within 90 ms). However, if the joining depth goes deep exceeding depth 3, the effect of cyclic join diminishes rapidly. Hence, total execution time of the previous query pruning grows exponentially. Again, the origin of the exponential growth is the exponential combinations of sub-results as well.

The experimental result on the running example also illustrates the effect of cyclic join. The total execution time of the previous query pruning is 40 ms, and that of the two-phase query pruning is 70 ms. It is the combined effect of the small schema size (14 nodes) and the shallow cyclic join (depth 1) by the join variable `B` that renders the previous query pruning efficient. The reason why the two-phase query pruning runs a little longer is that its AND/OR graph contains more nodes, whereas number of iterations is 3 for the two techniques. The experimental results are given in Table 3. Notice that the post-processing time (10 ms) is greater than the pre-processing time (0 ms).

Table 3
The experimental result on the running example

The two-phase query pruning	
Total number of expression accesses	13
Preprocessing time (ms)	0
Number of nodes in the AND/OR graph	588
Number of iterations	3
Total number of node accesses	1764
Pruning time (ms)	70
Total execution time (ms)	70
The query pruning with the post-processing	
Number of nodes in the AND/OR graph	350
Number of iterations	3
Total number of node accesses	1050
Pruning time (ms)	30
Total number of binding checks	40
Post-processing time (ms)	10
Total execution time (ms)	40

Number of path expressions = 5.

Finally, Table 4 presents the resulting queries of both the previous query pruning without the post-processing and the two-phase query pruning when `b` value is 15. This shows the effectiveness of the two-phase query pruning. We exclude the similar resulting queries for the other values of `b` to avoid meaningless repetition. In addition, we exclude the resulting query for the running example since we have presented it already in the previous sections. We are able to have equivalent queries to those of the two-phase query pruning if we post-process the resulting queries of the previous query pruning. However, the two-phase query pruning may produce fewer path expressions since it may omit useless intermediate variables from the given queries during preprocessing.

Table 4
The resulting queries from the experiments

<table>
<tr><th>Type</th><th>Name</th><th>The query pruning without the post-processing</th><th>The two-phase query pruning</th></tr>
<tr><td>Linear</td><td>Ql2</td><td>(e[0][1]|e[0][2]|···|e[0][15]).X1 in DB,
e[1][1].X2 in X1</td><td>e[0][1].e[1][1].X2 in DB</td></tr>
<tr><td></td><td>Ql3</td><td>(e[0][1]|e[0][2]|···|e[0][15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[1][15]).X2 in X1,
e[2][1].X3 in X2</td><td>e[0][1].e[1][1].e[2][1].X3 in DB</td></tr>
<tr><td></td><td>Ql4</td><td>(e[0][1]|e[0][2]|···|e[0][15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[1][15]).X2 in X1,
(e[2][1]|e[2][2]|···|e[2][15]).X3 in X2,
e[3][1].X4 in X3</td><td>e[0][1].e[1][1].e[2][1].e[3][1].X4 in DB</td></tr>
<tr><td></td><td>Ql5</td><td>(e[0][1]|e[0][2]|···|e[0][15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[1][15]).X2 in X1,
(e[2][1]|e[2][2]|···|e[2][15]).X3 in X2,
(e[3][1]|e[3][2]|···|e[3][15]).X4 in X3,
e[4][1].X5 in X4</td><td>e[0][1].e[1][1].e[2][1].e[3][1].-
e[4][1].X5 in DB</td></tr>
<tr><td>Branching</td><td>Qb0</td><td>(e[1]|e[2]|···|e[15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[15][15]).X12 in X1,
e[1][1][1].X13 in X12,
(e[1]|e[2]|···|e[15]).X2 in DB,
(e[1][1]|e[1][2]|···|e[15][15]).X22 in X2,
e[15][15][15].X23 in X22</td><td>e[1].e[1][1].e[1][1][1].X13 in DB,
e[15].e[15][15].e[15][15][15].X23 in DB</td></tr>
<tr><td></td><td>Qb1</td><td>(e[1]|e[2]|···|e[15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[15][15]).X12 in X1,
e[1][1][1].X13 in X12,
(e[1][1]|e[1][2]|···|e[15][15]).X22 in X1,
e[1][15][15].X23 in X22</td><td>e[1].X1 in DB,
e[1][1].e[1][1][1].X13 in X1,
e[1][15].e[1][15][15].X23 in X1</td></tr>
<tr><td></td><td>Qb2</td><td>(e[1]|e[2]|···|e[15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[15][15]).X12 in X1,
e[1][1][1].X13 in X12, e[1][1][15].X23 in X12</td><td>e[1].e[1][1].X12 in DB,
e[1][1][1].X13 in X12,
e[1][1][15].X23 in X12</td></tr>
<tr><td>Cyclic/joining</td><td>Qj1</td><td>(e[0][1]|e[0][2]|···|e[0][15]).X1 in DB,
e[1][1].X2 in X1, e[2][1].X3 in X2,
e[3][1].X4 in X3, e[4][1].X1 in X4</td><td>e[0][1].X1 in DB,
e[1][1].e[2][1].e[3][1].e[4][1].X1 in X1</td></tr>
<tr><td></td><td>Qj2</td><td>(e[0][1]|e[0][2]|···|e[0][15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[1][15]).X2 in X1,
e[2][1].X3 in X2, e[3][1].X4 in X3,
e[4][1].X2 in X4</td><td>e[0][1].e[1][1].X2 in DB,
e[2][1].e[3][1].e[4][1].X2 in X2</td></tr>
<tr><td></td><td>Qj3</td><td>(e[0][1]|e[0][2]|···|e[0][15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[1][15]).X2 in X1,
(e[2][1]|e[2][2]|···|e[2][15]).X3 in X2,
e[3][1].X4 in X3, e[4][1].X3 in X4</td><td>e[0][1].e[1][1].e[2][1].X3 in DB,
e[3][1].e[4][1].X3 in X3</td></tr>
<tr><td></td><td>Qj4</td><td>(e[0][1]|e[0][2]|···|e[0][15]).X1 in DB,
(e[1][1]|e[1][2]|···|e[1][15]).X2 in X1,
(e[2][1]|e[2][2]|···|e[2][15]).X3 in X2,
(e[3][1]|e[3][2]|···|e[3][15]).X4 in X3,
e[4][1].X4 in X4</td><td>e[0][1].e[1][1].e[2][1].e[3][1].X4 in DB,
e[4][1].X4 in X4</td></tr>
</table>

7. Concluding remarks

We have presented an effective and scalable two-phase query pruning for multiple regular path expressions, which eliminates schema-level excessive paths without compromising the time complexity. On the basis of the new concepts to cope with the ineffectiveness of the previous query pruning, the preprocessing phase finds every least maximized condition from an input query in polynomial time with respect to the number of regular path expressions. The pruning phase, which is an extended version of the previous query pruning, produces fully optimized path expressions in polynomial time with respect to the number of states in the regular

path expressions and the number of schema nodes in the semistructured schema.

In addition, we have proved that the preprocessing phase is sound and complete, and the two-phase query pruning is correct and effective. Finally, we have conducted several experiments, and the experimental results show that our two-phase query pruning is both effective and scalable unlike the previous approach.

Acknowledgements

This research was supported by the Brain Korea program of the Ministry of Education & Human Resources Development.

References

Abiteboul, S., 1997. Querying semi-structured data. In: Proceedings of the Sixth International Conference on Database Theory, pp. 1–18.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L., 1997. The Lorel query language for semistructured data. International Journal on Digital Libraries 1 (1), 68–88.

Abiteboul, S., Buneman, P., Suciu, D., 2000. Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Buneman, P., 1997. Semistructured data. In: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 51–61.

Buneman, P., Davidson, S.B., Fernandez, M.F., Suciu, D., 1997. Adding structure to unstructured data. In: Proceedings of the Sixth International Conference on Database Theory, pp. 336–350.

Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y., 1999. Rewriting of regular expressions and regular path queries. In: Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 194–204.

Christophides, V., Abiteboul, S., Cluet, S., Scholl, M., 1994. From structured documents to novel query facilities. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 313–324.

Christophides, V., Cluet, S., Moerkotte, G., 1996. Evaluating queries with generalized path expressions. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 413–422.

Fernandez, M.F., Suciu, D., 1998. Optimizing regular path expressions using graph schemas. Proceedings of the 14th International Conference on Data Engineering, pp. 14–23 (The full version is available at http://www.cs.washington.edu/homes/suciu/files/paper-techrep.ps).

Goldman, R., Widom, J., 1997. DataGuides: enabling query formulation and optimization in semistructured databases. In: Proceedings of the 23rd International Conference on Very Large Data Bases, pp. 436–445.

Halevy, A.Y., 2000. Theory of answering queries using views. SIGMOD Record 29 (4), 40–47.

Kifer, M., Kim, W., Sagiv, Y., 1992. Querying object-oriented databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 393–402.

McHugh, J., Widom, J., 1999a. Compile-time path expansion in lore. Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats.

McHugh, J., Widom, J., 1999b. Optimizing branching path expressions. Technical Report, Stanford University.

McHugh, J., Widom, J., 1999c. Query optimization for XML. In: Proceedings of the 25th International Conference on Very Large Data Bases, pp. 315–326.

Nestorov, S., Ullman, J.D., Wiener, J.L., Chawathe, S.S., 1997. Representative objects: concise representations of semistructured, hierarchical data. In: Proceedings of the 13th International Conference on Data Engineering, pp. 79–90.

Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J.D., Widom, J., 1995. Querying semistructured heterogeneous information. In: Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases, pp. 319–344.

Chang-Won Park is a Ph.D. candidate of the Division of Computer Science at Korea Advanced Institute of Science and Technology. His current research interests include XML and semistructured databases, query processing for XML and Semistructured data.

Chin-Wan Chung is professor and chair of the Division of Computer Science at the Korea Advanced Institute of Science and Technology (KAIST). From 1983 to 1993, he was a senior research scientist and a staff research scientist in the Computer Science Department at the General Motors Research Laboratories (GMR). He received a Ph.D. from the University of Michigan in 1983. While at GMR, he developed DATAPLEX, a heterogeneous distributed database management system integrating relational databases and hierarchical databases. At KAIST, he developed a full scale object-oriented spatial database management system called OMEGA, which supports ODMG standards. His current research interests include multimedia databases, XML, OLAP and spatio-temporal databases.