PAPER

Decomposition Optimization for Minimizing Label Overflow in Prime Number Graph Labeling

Jaehoon KIM[†a)] ***and*** **Seog PARK**[††b)]**,** ***Members***

SUMMARY Recently, a graph labeling technique based on prime numbers has been suggested for reducing the costly transitive closure computations in RDF query languages. The suggested prime number graph labeling provides the benefit of fast query processing by a simple divisibility test of labels. However, it has an inherent problem that originates with the nature of prime numbers. Since each prime number must be used exclusively, labels can become significantly large. Therefore, in this paper, we introduce a novel optimization technique to effectively reduce the problem of label overflow. The suggested idea is based on graph decomposition. When label overflow occurs, the full graph is divided into several sub-graphs, and nodes in each sub-graph are separately labeled. Through experiments, we also analyze the effectiveness of the graph decomposition optimization, which is evaluated by the number of divisions.

key words: *graph labeling, prime number, graph decomposition, access control, query processing*

1. Introduction

Through studies on eXtensible Markup Language (XML) query processing [1]–[3] and XML access control [4], [5], we have learned that the *tree* labeling techniques significantly enhance performance. Similarly, some *graph* labeling techniques [6], [7] have been suggested for the optimization of RDF query languages in recent times. In particular, they optimize the queries on subsumption relationship over an ontology hierarchy by avoiding the costly transitive closure computations [8].

G. Wu et al. [7] have recently shown that their graph labeling based on prime numbers performs better than the interval-based scheme and the prefix-based scheme suggested by Christophides et al. [6]. When the size of a graph to be labeled is small, the prime number graph labeling can support fast query processing by only a simple divisibility test of labels. It can identify node relationships over a graph by using only elementary arithmetic operators, such as multiplication, division, and modulo. However, it has an inherent problem that originates with the nature of prime numbers. A prime number label can become significantly large since each prime number must be used exclusively. Therefore, G. Wu et al. suggested some optimization techniques (Least Common Multiple, Topological Sort, etc.) to minimize the label overflow problem, where a fixed-length variable cannot store the significantly large prime number label. However, those optimization techniques still have the label overflow problem. We can consider simply extending the variable length, but this naive extension is not a desirable solution. Let us consider an n-bit label variable as in Fig. 1. If (n) bits are wholly filled with '1' by multiplying the consecutive prime numbers ..., 11, 13, 17, ..., pn_k, simply extending the variable into $(n + m)$ bits is not effective. This is because the extended space must be large enough to store the next prime numbers which are larger than pn_k. For example, let m be 3. Although the extension allows the four additional prime numbers 2, 3, 5 and 7 ($< 2^3 = 8$), it is useless since $2^3 \ll pn_k$. In short, although we additionally assign a space that is quite large, we can only use a few additional labels. It is inefficient.

Therefore, in this study, we investigate *graph decomposition* as another optimization technique. It uses the extended space more efficiently. When there is a label overflow, a full Directed Acyclic Graph (DAG) can be divided into several sub-DAGs, and nodes in each sub-DAG can be separately labeled. The extended 3 bits are used for identifying sub-DAGs, and (n) bits are reused by re-labeling from the first prime number '2'.

However, this optimization technique is useless when the division is intensive. This is because it requires an additional calculation to identify node relationship between sub-DAGs. The more the number of divided sub-DAGs, the higher the cost of query evaluation additionally considering the node relationship between sub-DAGs. Therefore, in this paper, we also analyze the effectiveness of the graph decomposition optimization according to the number of divisions through some experiments. The experiments in Sect. 6 show that the prime number labeling scheme is superior or similar to the prefix-based labeling scheme [6] when the number of divisions is less than or equal to two.

The remainder of the paper is organized as follows. In Sect. 2, we briefly explain the prime number labeling scheme and the optimization techniques suggested by G. Wu et al. [7]. In Sects. 3, 4, and 5, we introduce the graph de-

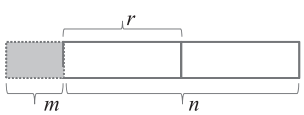


Fig. 1 A fixed-length label variable.

Manuscript received October 30, 2009.
Manuscript revised February 25, 2010.

[†]The author is with the Dept. of Information Communication, Seoil University, 49–3 Myeonmok-Dong Jungnang-Gu, Seoul, 131–702, Korea.
[††]The author is with the Dept. of Computer Science, Sogang University, 1–1 Shinsu-Dong Mapo-Gu, Seoul, 121–742, Korea.
a) E-mail: jhkimygk@seoil.ac.kr
b) E-mail: spark@dblab.sogang.ac.kr

DOI: 10.1587/transinf.E93.D.1889

composition optimization technique, and in Sect. 6 we analyze the effectiveness of the technique according to the number of divisions. Section 7 finally concludes this paper.

2. Background: Prime Number Graph Labeling and Optimization Techniques

The prime number *graph* labeling (= PNGL) suggested by G. Wu et al. [7] extends the prime number *tree* labeling suggested by X. Wu et al. [3] to DAG. The following is the definition of PNGL; since the depth-first traversal adapted in G. Wu et al.'s scheme can have many big prime numbers at upper vertices which are propagated to lower vertices, we here replace it with the breadth-first traversal.

Definition 1 (PNGL = Prime Number Graph Labeling): Let $G = (V, E)$ be the DAG. By traversing G with a breadth-first traversal, the following PNGL(v) = ($L1[v]$, $L2[v]$, $L3[v]$) is performed for each vertex $v \in V$.

$$L2(v) = \begin{cases} L1(v) \times \prod_{w \in parents(v)} L2(w), & in\text{-}degree(v) > 0 \\ 1 & in\text{-}degree(v) = 0 \end{cases}$$

$$L3(v) = \begin{cases} \prod_{w \in parents(v)} L1(w), & in\text{-}degree(v) > 0 \\ 1 & in\text{-}degree(v) = 0 \end{cases}$$

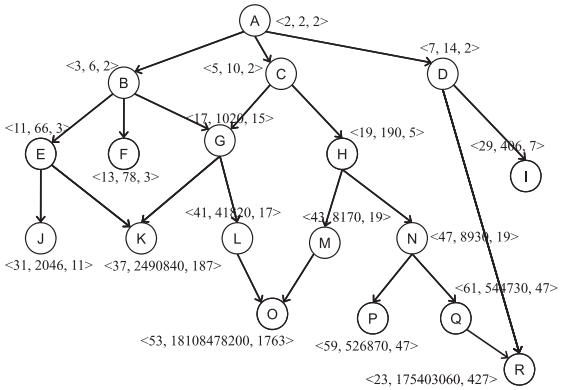
where $L1(v)$ indicates an exclusive prime number assigned to each vertex, $parents(v)$ indicates the parent vertices of a vertex, v, and *in-degree* indicates the number of parent vertices of a vertex v. $L2(v)$ is obtained by multiplying the labels of the parent vertices of v and the prime number of v, and $L3(v)$ is obtained by multiplying the prime numbers of the parent vertices of v. As an illustration, refer to the labeling in Fig. 2 (a).

Only by doing a simple arithmetic calculation on the labels, we can conveniently grasp the ancestor/descendant relationship, the parent/child relationship, and the sibling relationship between nodes. First, since a prime number is divided by 1 and itself, the ancestor/descendant relationship between two nodes can be simply determined by the divisibility test of their $L2$ labels. For example, in Fig. 2 (a), 'B' is the ancestor node of 'K' since $L2$(K) mod $L2$(B) = 0. Similarly, if and only if $L3(v_1)$ mod $L1(v_2) = 0$, v_2 is the parent node of v_1. In addition, v_1 and v_2 are siblings if and only if the greatest common divisor $gcd(L3(v_1), L3(v_2)) \neq 1$. For example, 'G' is the sibling of 'F' since $gcd(3, 15) = 3 \neq 1$. Regarding re-labeling ability for updates, PNGL performs the elementary arithmetic calculation just for the descendant nodes of an updated node. When the new 'S' node is inserted between 'B' and 'G', the $L2$ values of 'G' and its descendants are multiplied by the $L1$ of 'S'. When 'G' is deleted, the $L2$ values of the descendants of 'G' are divided by the $L2$ value of 'G'.

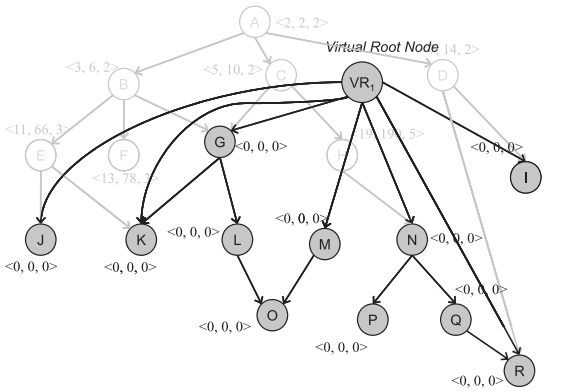
Although PNGL has the strong benefit of identifying node relationships by this simple arithmetic calculation, it has an inherent problem that originates with the nature of prime numbers. Since each prime number is exclusive in PNGL and each vertex can inherit the prime number labels from multiple parent vertices, the label size can become too large to be covered by a fixed-length variable. We name this problem *label overflow*.

In order to cope with this problem, G. Wu et al. suggested some optimization techniques.

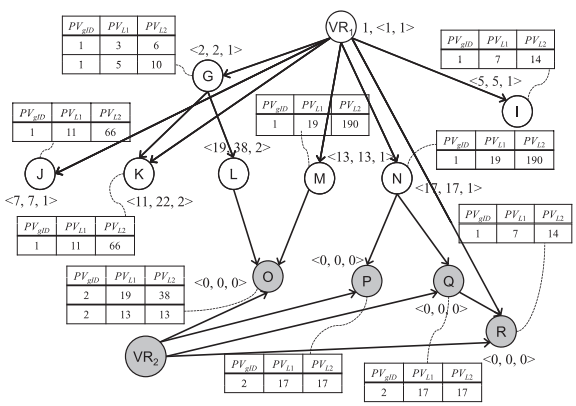
• Least Common Multiple: This optimization minimizes label overflows through removing redundancy between inher-



(a) The basic PNGL <$L1$, $L2$, $L3$>



(b) First decomposition



(c) Second decomposition & Bridging

Fig. 2 Decomposition optimization.

ited labels from ancestor nodes. For example, in Fig. 2 (a), $L2$(G) inherits $L1$(A) two times from $L2$(B) and $L2$(C). Hence, we can remove the redundancy, and the smaller $L2$(G) (= $L1$(A) × $L1$(B) × $L1$(C) × $L1$(G)) is still enough to identify the ancestor/descendant relationship. In order to remove the redundancy, G. Wu et al. used the optimization technique which multiplies a self-label $L1$ by the least common multiple of all the parents' $L2$ labels.

• Topological Sort: The purpose of this optimization is to consider a topological sort of a DAG which results in a much smaller label size. As an example, in Definition 1, we considered the breadth-first traversal instead of the depth-first traversal.

3. Decomposition Optimization

Our optimization technique is based on graph decomposition. That is, when a label overflow occurs, a full DAG is divided into some sub-DAGs, and the PNGL with the breadth-first traversal is executed for each sub-DAG. Also, connection information between sub-DAGs is maintained to calculate relationship between nodes individually belonging to other sub-DAGs.

When a DAG is given, we think that an optimal division is related with the number of sub-DAGs, the size of the connection information, and the label size in each sub-DAGs. And these factors are all to minimize query processing cost including the calculation of node relationships for the DAG. If the number of sub-DAGs and the connection information are large, the cost for calculating node relationships among sub-DAGs must be high. Also, if the label size in each sub-DAG is big, label overflow arises before long and consequently this requires another decomposition.

We think that our decomposition problem is NP-complete since the schema selection problem [9], [10] can be easily reduced to our problem. Zheng et al. [9] and Bohannon et al. [10] studied on optimally partitioning an XML graph in order to obtain the optimal relational mapping schema for a given query set. Zheng et al. [9] mentioned that the graph partition problem in Garey et al. [11], which is NP-Complete, can be reduced to the schema selection problem. Our problem is also to optimally partition an ontology graph into subgraphs in order to minimize query processing cost. In our study, the subgraphs are mapped into relational tables since large ontology data are considered. Through studying other partitioning approaches including the greedy methods in the Refs. [9], [10], we have known that a more complex partitioning approach makes the calculation of node relationships more complex and inefficient. Therefore, in this paper, we suggest a heuristic method which is based on linear decomposition by the breadth-first traversal (BFT). Although as in the schema selection problem our problem also considers given queries having the calculation of node relationships, due to such complexity and inefficiency, our linear approach does not use query processing cost as a selection criterion. That is, regardless of a given query set, our simple approach produces a general solution. Our optimal graph partition problem is defined as follows.

Definition 2 (optimal graph partition in PNGL): Let G a DAG, and let Q given queries having the calculation of node relationships for G. Also, it is assumed that PNGL is performed with BFT as in Definition 1. The optimal graph partition in *PNGL* is to optimally partition G such that the query processing cost for Q is minimal.

Instead of the BFT, we can also implement the linear decomposition by the depth-first traversal (DFT). However, since the DFT is most likely to have much bigger L2 labels than the BFT, we adopted the BFT. As might have been expected, the experimental results of Sect. 6.2 have shown that the DFT has more sub-DAGs.

The outline of our linear approach is as follows. If a graph parsed for an Resource Description Framework (RDF) document is given, PNGL is performed for the graph by the BFT. Whenever label overflow arises at a graph node, the node is connected to a virtual root node. When all nodes have been visited, PNGL is again performed for the sub-DAG of the virtual root node. Similarly, whenever there is label overflow in the sub-DAG, the node is connected to another virtual root node. Again, PNGL is performed from the virtual root node. This processing is repeated until there is no label overflow. While performing PNGL, the label information <$L1$, $L2$, $L3$> is kept in the parsed RDF graph, and the connection information between sub-DAGs is also kept in the graph. And by another BFT, the label and the connection information are stored into a stable storage (this is a database in our implementation).

The codes in the Appendix describe the representation of a single node in a DAG and the decomposition algorithm. The boolean variable $visit\_flag$ in the node structure is used for marking whether or not a node is visited already. The boolean variable $store\_flag$ is used for marking whether or not the label information of a node is stored into a stable storage. The suggested algorithm is performed with two scannings of a DAG. One scanning (lines 4 to 45) is for assigning a label to each node and the other (lines 46 to 67) is for storing the assigned labels into a stable storage.

The execution of the function *Decomp_PNGL* is as follows. The input parameter R refers to the root node of an input DAG and the input parameter gID is used for identifying sub-DAGs. First, at line 3, a virtual root node having the label <1, 1, 1> is created as in Fig. 2 (b) (hereafter denoted as *VR*). Next, from line 5 to line 45, the PNGL of Definition 1 is executed with the BFT. Lines 4, 5, 6, 11, 17, 23, 26, 30, 37, and 41 which use the queue Q, implement the BFT. For example, in Fig. 2 (a), the parent node 'B' is deleted from Q by line 5, the child nodes 'E', 'F', and 'G' are labeled by lines 7 to 43, and the child nodes are added into Q by lines 11, 17, 23, and 26. Again, the first child node 'E' is deleted from Q by line 5.

If a node is not visited, <$L1$, $L2$, $L3$> of the node is calculated at lines 13, 19, and 25. Otherwise, $L2$ and $L3$ are calculated at lines 33 and 40. If a label overflow occurs

LABEL table

gID	*L1*	*L2*	*L3*	*Name*
1	2	2	2	A
1	3	6	2	B
1	5	10	2	C
1	7	14	2	D
1	11	66	3	E
1	13	78	3	F
1	19	190	5	H
2	2	2	1	G
2	5	5	1	I
2	7	7	1	J
2	11	22	2	K
2	**13**	**13**	**1**	**M**
2	17	17	1	N
2	19	38	2	L
3	**2**	**2**	**1**	**O**
3	3	3	1	P
3	5	5	1	Q
3	**7**	**7**	**1**	**R**

BRIDGE table

PV_gID	PV_L1	PV_L2	CV_gID	CV_L2
1	3	6	2	2
1	5	10	2	2
1	7	14	2	5
1	11	66	2	7
1	11	66	2	22
1	19	190	2	13
1	19	190	2	17
2	19	38	3	2
2	13	13	3	2
2	17	17	3	3
2	17	17	3	5
1	**7**	**14**	**3**	**7**

Fig. 3 LABEL and BRIDGE tables.

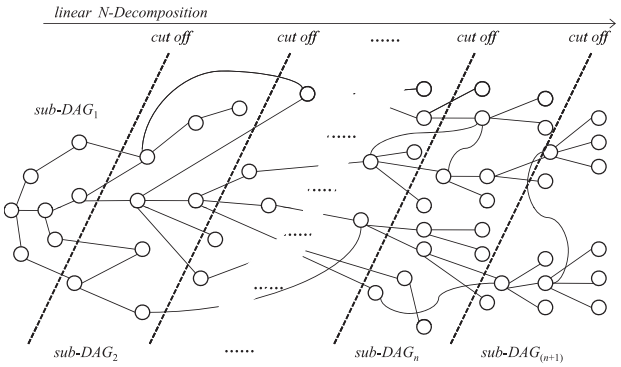


Fig. 4 Linear decomposition.

at a node, the node is connected to *VR*, and itself and its descendant nodes are marked by zero at lines 10, 16, 22, 29, and 36. For example, in Fig. 2 (b), 'G', 'I', 'J', 'K', 'M', 'N' and 'R' are connected to the VR_1 and their descendant nodes are set to zero. In the figure, we assume the maximum label value is 256. When a node is connected to *VR*, the decomposition information is stored into the table *BRIDGE*(*PV_gID*, *PV_L*1, *PV_L2*, *CV_gID*, *CV_L2*) in Fig. 3, where the prefixes *CV* and *PV* indicate a child vertex with label overflow, and its parent vertices respectively. For example, in Fig. 2 (c), when 'G' has a label overflow, ($gID(B) = 1$, $L1(B) = 3$, $L2(B) = 6$) are saved into the variables *PV_gID*, *PV_L*1 and *PV_L2* of *G*, and ($gID(G) = 2$, $L2(G) = 2$) are saved into the variables *CV_gID* and *CV_L2* of *G*. This is performed by lines 54, 58, and 64 in the algorithm. Next, the saved information is stored into the table *BRIDGE* by line 54. With the second BFT, lines 47 to 67 store the label information into the *LABEL* and the *BRIDGE* tables in Fig. 3.

Recursively, at line 70, the function *Decomp_PNGL*(VR_1, 2) is executed. In Fig. 2 (c), since *O*, *P*, *Q*, and *R* nodes have a label overflow, they are connected to the VR_2 and the function *Decomp_PNGL*(VR_2, 3) is executed again. At line 69, this successive decomposition process is executed until *VR* has no child node, that is, until there is no label overflow.

The suggested decomposition is linear. As in Fig. 4, $sub\text{-}DAG_1$, $sub\text{-}DAG_2$, ..., $sub\text{-}DAG_n$, and $sub\text{-}DAG_{n+1}$ are cut off linearly. There are $(N + 1)$ sub-DAGs by *N*-Decomposition. We call dividing a graph into $(N + 1)$ sub-DAGs as "*N*-Decomposition". For example, in Fig. 2, there are three sub-DAGs by 2-Decomposition. In order to accomplish the linear decomposition, we used the virtual root node.

We believe that a chaotic decomposition makes the PNGL complex and inefficient. That is, it degrades the key benefit of simplicity of the PNGL since answering for node relationships cannot be simply performed by only the simple divisibility test of labels. Therefore, we adopted this simple linear decomposition.

4. Answering for Node Relationships

Some typical operations such as *descendant*(), *ancestor*(), and *sibling*() are still executable through the simple divisibility test over the *BRIDGE* and the *LABEL* tables. In this study, since we are interested in implementing the operations using standard SQL engines, we will show how they can be represented in SQL. We are interested in querying very large RDF data. Let us first consider *descendant*(). For a series of sub-DAGs G_i, $1 \le i \le (N+1)$, if a node is in G_k, its descendant nodes need to be calculated for G_j, $j \ge k$ due to the linearity. For example, in Fig. 2, *descendant*(B) is calculated for $sub\text{-}DAG_1$, $sub\text{-}DAG_2$, and $sub\text{-}DAG_3$, but *descendant*(L) is calculated for $sub\text{-}DAG_2$ and $sub\text{-}DAG_3$. The following SQL expressions are for *descendant*(B). Here, the symbol '→' means that some nodes in two sub-DAGs are connected, and the symbol '%' is the modulo operator in SQL.

$sub\text{-}DAG_1$:

```
select name from LABEL
where gID = 1 and L2 % 6 = 0  /* L2(B) = 6 */
```

$sub\text{-}DAG_1 \rightarrow sub\text{-}DAG_2$, $sub\text{-}DAG_1 \rightarrow sub\text{-}DAG_3$:

```
select name from LABEL,
  (select CV_gID, CV_L2
   from BRIDGE
   where PV_gID = 1 and PV_L2 % 6 = 0) u
where LABEL.gID = u.CV_gID and
      LABEL.L2 % u.CV_L2 = 0
```

$sub\text{-}DAG_1 \rightarrow sub\text{-}DAG_2 \rightarrow sub\text{-}DAG_3$:

```
select name from LABEL,
  (select BRIDGE.CV_gID, BRIDGE.CV_L2
   from BRIDGE,
     (select CV_gID, CV_L2
      from BRIDGE
      where PV_gID = 1 and PV_L2 % 6 = 0) u
    where PV_gID = u.CV_gID and PV_L2 % u.CV_L2 = 0) v
where LABEL.gID = v.CV_gID and LABEL.L2 % v.CV_L2 = 0
```

In the above expressions, we can see that the node relationship between sub-DAGs is simply identified using the self-join and the modulo operation over the *BRIDGE* table. The SQL expression for *ancestor*() can be represented similar to *descendant*(). The following example is for *ancestor*(K).

$sub\text{-}DAG_2$:

```
select name from LABEL
```

```
  where gID = 2 and 22 % L2 = 0  /* L2(K) = 22 */
sub-DAG2 → sub-DAG1:
  select name from LABEL,
    (select PV_gID, PV_L2
     from BRIDGE
     where CV_gID = 2 and 22 % CV_L2 = 0) u
  where LABEL.gID = u.PV_gID and
        u.PV_L2 % LABEL.L2 = 0
```

Next, let us consider *sibling*(). For a series of sub-DAGs G_i, $1 \le i \le (N+1)$, if a node is in G_k, its sibling nodes exist in G_k and in G_j, which satisfies $G_j \rightarrow G_k$ or $G_k \rightarrow G_j$, $j \in i$. For example, the sibling nodes of 'G' in sub_DAG_1 exist in sub_DAG_1 and sub_DAG_2. The sibling nodes of 'R' in sub_DAG_3 exist in sub_DAG_2, the sibling nodes of 'F' in sub_DAG_1 exist in sub_DAG_1 and sub_DAG_2. Therefore, to implement the *sibling*() operation, the upward case ($G_j \rightarrow G_k$) and the downward case ($G_k \rightarrow G_j$) are both tested. The SQL expressions for *sibling*(G) are as follows. *gcd*() is a stored procedure to calculate the greatest common divisor.

```
sub-DAG2:
  select name from LABEL
  where gID = 2 and gcd(1, L3) != 1  /* L3(G) = 1 */
  union
  select name from LABEL
  where gID = 2 and L2 in (
    select CV_L2 from BRIDGE
    where CV_gID = 2 and PV_L1 in (
      select PV_L1 from BRIDGE
      where CV_gID = 2 and CV_L2 = 2)  /* L2(G) = 2 */
sub-DAG1 → sub-DAG2:
  select name from LABEL,
    (select PV_gID, PV_L1
     from BRIDGE
     where CV_gID = 2 and CV_L2 = 2) u  /* L2(G) = 2 */
  where LABEL.gID = u.PV_gID and
        LABEL.L3 % u.PV_L1 = 0    /* 'LABEL.gID = u.PV_gID' is for
testing all Gjs which satisfies Gj → Gk, 1 ≤ j < k. */
sub-DAG2 → sub-DAG3:      /* However, since G has no sibling node in
sub-DAG3, this query returns null. */
  select name from LABEL,
    (select CV_gID, CV_L2
     from BRIDGE
     where PV_gID = 2 and 1 % PV_L1 = 0) u  /* L3(G) = 1 */
  where LABEL.gID = u.CV_gID and
        LABEL.L2 = u.CV_L2    /* 'LABEL.gID = u.CV_gID' is for testing
all Gjs which satisfy Gk → Gj, k < j ≤ (N+1). */
```

There is a special case where a large number of sibling nodes can belong to several sub-DAGs. See the Fig. 5. In Fig. 5 (b), by the label overflow in Fig. 5 (a), $sub\text{-}DAG_2$ with the virtual root node VR_1 is configured. Again, in Fig. 5 (c), by the label overflow in Fig. 5 (b), $sub\text{-}DAG_3$ with the virtual root node VR_2 is configured. In this case, as in Fig. 5 (c), the nodes in $sub\text{-}DAG_3$ have <1, 1, 1> which represents PV_gID, PV_L1, and PV_L2 for VR_1 in $sub\text{-}DAG_2$. Remember that a virtual root node has the label <1, 1, 1> in Sect. 3. Since the BRIDGE information for the original parent node v in $sub\text{-}DAG_1$ is lost, we cannot calculate all sibling nodes. To cope with this problem, we use inheriting the BRIDGE information <PV_gID, PV_L1, PV_L2> of the node v into sub-DAGs, and this inheritance arises only

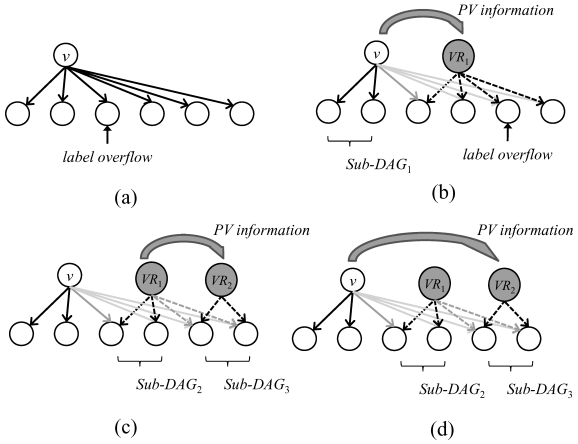


Fig. 5 Inheritance of BRIDGE information.

when child nodes are detached from a virtual root node. In Fig. 5 (d), the child nodes in $sub\text{-}DAG_3$ have the BRIDGE information <PV_gID, PV_L1, PV_L2> of v. Since all sibling nodes belonging to several sub-DAGs have the same PV (parent vertex) information, we can correctly calculate the operations *sibling*(), *descendant*(), and *ancestor*(). In order to implement the inheritance, we added the condition '$tmpn1.L1$!= 1' into line 57 and 63 in the suggested algorithm of Appendix. Note that the $L1$ for a virtual root node is 1. By the condition, the BRIDGE information assigned once in Fig. 5 (a) is kept in the consecutively decomposed sub-DAGs.

5. Re-Labeling for Updates

In this section, for update operations, we consider how re-labeling can be expressed by SQL queries on the *LABEL* and the *BRIDGE* tables.

5.1 Deletion

Deletion operation can be performed for edges or nodes. For example, as for edge deletion, we can delete the edge ('B', 'G') in Fig. 2 (a); this means that a $subClassOf$ or a $subPropertyOf$ relationship [11] is broken. Also, as for node deletion, we can delete the node 'G'; this means that a class, a property, or an instance is removed. Since both re-labeling mechanisms for edge deletion and node deletion are the same, we here focus on node deletion.

The re-labeling cost in the deletion operation is understandably low. This is because re-labeling needs to be performed just for the sub-DAG which a deleted node belongs to. For example, when the node 'B' is deleted, we need to adjust the $L2$ values of the descendant nodes of 'B' just in the sub-DAG_1 and the $L3$ values of the child nodes of 'B'. Let us see the following SQL queries. We first update the *LABEL* table.

```
delete from LABEL
where gID = 1 and L1 = 3    /* L1(B) = 3 */
update LABEL set L3 = L3 / 3
```

where gID = 1 and $L3$ % 3 = 0 /* $L1(B)$ = 3 */

update $LABEL$ set $L2$ = $L2$ / 6
where gID = 1 and $L2$ % 6 = 0 /* $L2(B)$ = 6 */

Next, we update the $BRIDGE$ table.

delete from $BRIDGE$
where PV_gID = 1 and PV_L1 = 3 /* $L1(B)$ = 3 */

update $BRIDGE$ set PV_L2 = PV_L2 / 6 /* $L2(B)$ = 6 */
where gID = 1 and PV_L2 % 6 = 0

Let us now consider a special deletion operation which removes all the descendant nodes of a deleted node. An example is to remove all the descendant nodes of 'B'. In this case, the re-labeling procedure must be performed for all sub-DAGs rather than a specific sub-DAG. Hence, the re-labeling cost can be high according to the number of divisions. The re-labeling procedure is as follows. First, using the $BRIDGE$ table, the descendant nodes of a deleted node are deleted from $LABEL$. As in the *descendant* operation, for a series of sub-DAGs G_i, $1 \leq i \leq (N+1)$, if a deleted node is in G_k, re-labeling need to be performed for G_j, $j \geq k$. The following SQL queries are for deleting 'B' and its descendant nodes.

sub-DAG$_1$:
delete from $LABEL$
where gID = 1 and $L2$ % 6 = 0 /* $L2(B)$ = 6 */

sub-DAG$_1$ → *sub-DAG*$_2$, *sub-DAG*$_1$ → *sub-DAG*$_3$:
delete from $LABEL$ where exists
(select CV_L2
from $BRIDGE$
where PV_gID = 1 and PV_L2 % 6 = 0 and
$LABEL.gID = BRIDGE.CV\_gID$ and $LABEL.L2$ % $BRIDGE.CV\_L2$ = 0)

sub-DAG$_1$ → *sub-DAG*$_2$ → *sub-DAG*$_3$:
delete from $LABEL$ where exists
(select $BRIDGE.CV\_L2$
from $BRIDGE$,
(select CV_gID, CV_L2
from $BRIDGE$
where PV_gID = 1 and PV_L2 % 6 = 0) u
where $PV\_gID = u.CV\_gID$ and $BRIDGE.PV\_L2$ % $u.CV\_L2$ = 0 and $LABEL.gID = BRIDGE.CV\_gID$ and $LABEL.L2$ % $BRIDGE.CV\_L2$ = 0)

Next, using the $LABEL$ table, delete the orphaned nodes from the $BRIDGE$ table which do not exist in the $LABEL$ table.

delete from $BRIDGE$ where not exists
(select $L2$ from $LABLE$ where $gID = BRIDGE.PV\_gID$ and $L2 = BRIDGE.PV\_L2$)

delete from $BRIDGE$ where not exists
(select $L2$ from $LABLE$ where $gID = BRIDGE.CV\_gID$ and $L2 = BRIDGE.CV\_L2$)

5.2 Insertion

A node/edge insertion can bring about a chain of label overflows between sub-DAGs. When a new node is inserted into a sub-DAG, it can have some overflown descendant nodes in the sub-DAG. When the overflown descendant nodes are inserted into the next sub-DAG, they can also have some overflown descendant nodes in the sub-DAG. In the worst case, these consecutive insertions can be repeated until reaching the last sub-DAG. The re-labeling cost can be significant. This problem is a limitation in the suggested decomposition optimization. To minimize this insertion problem, we adopt an approach to reserve some space in a fixed-length label variable for later additional insertions. As in Fig. 1, only (n-r) bits are used for the initial labeling, and the remaining space r is reserved for later additional insertions. An insertion is allowed just within the reserved space. It is important to understand that this approach still allows many more insertions than the basic PNGL. The proof is straightforward. For example, in Fig. 1, let us consider the n-bits variable is almost fully occupied by the basic PNGL. If so, a label overflow can easily arise by only a few additional insertions. However, suppose that the fully occupied labels are transformed into $(n - r)$ bits labels in several sub-DAGs by N-Decomposition. There are apparently as many additional insertions for each sub-DAG as can be accommodated by the reserved space r. In our implementation, we set up r to a half of n.

Based on the above approach, we consider the following two re-labeling cases for a node/edge insertion.

• Non-leaf node/edge insertion in a sub-DAG: Let us consider inserting a new node 'S' between 'A' and 'B' as an illustration. In the sub-DAG$_1$, the smallest next prime number for 'S' is 17. See the $L1$ values of which gID is 1 in Fig. 3. Hence the label of 'S' is <17, 34, 2>, and the labels of 'B', 'E', and 'F' are recalculated. This operation can be performed by the following SQL queries.

insert into $LABEL$ values(1, 17, 34, 2, 'S')

update $LABEL$ set $L2$ = $L2$ * 17
where gID = 1 and $L2$ % 6 = 0 /* $L2(B)$ = 6 */

update $LABEL$
set $L3$ = $L3$ / 2 * 17 /* $L1(A)$ = 2, $L1(S)$ = 17 */
where gID = 1 and $L1$ = 3 /* $L1(B)$ = 3 */

Next, we also must update $L2$ in the BRIDGE table.

update $BRIDGE$ set PV_L2 = PV_L2 * 17
where PV_gID = 1 and PV_L2 % 6 = 0 /* $L2(B)$ = 6 */

• Leaf node insertion in a sub-DAG: Let us consider inserting 'S' node between 'B' and 'G' as an illustration. In this case, we only need to replace 'B' with 'S' in the BRIDGE table.

insert into $LABEL$ values(1, 17, 108, 3, 'S');

update $BRIDGE$ set PV_L2 = 108 /* $L2(S)$ = 108 */
where PV_gID = 1 and PV_L1 = 3 /* $L1(B)$ = 3 */

update $BRIDGE$ set PV_L1 = 17 /* $L1(S)$ = 17 */
where PV_gID = 1 and PV_L1 = 3

6. Performance Evaluation

6.1 Experimental Setup

In the graph decomposition optimization technique, the more the number of divided sub-DAGs, the higher the eval-

Table 1 Experimental parameters.

Parameter	Range	Description
$\#n$	100 to 1,000	Number of nodes in a DAG
$\#fo$	5 to 20	Average number of fan-out (= outdegree) of a tree node
$\#fi$	1 to 5	Average number of fan-in (= indegree) of a graph node
$\#vl$	long (8 bytes)	Length of a variable to store each prime number label

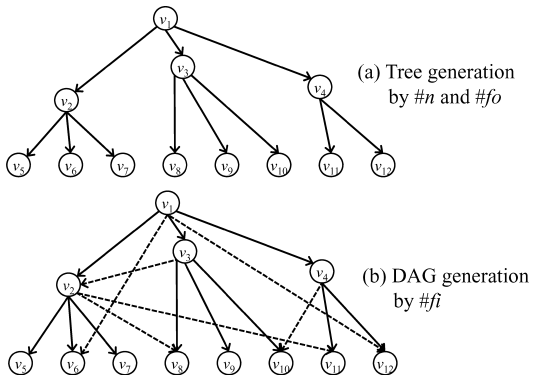


Fig. 6 Test DAG generation.

uation cost of a query. This is because query evaluation requires as many join operations over the *BRIDGE* table as the number of divisions. In this section, we analyze how much the graph decomposition optimization can extend the basic PNGL and also analyze how much decomposition is acceptable. In previous studies [6], [7], we can see that the prefix-based scheme (Dewey Decimal Coding) has, on average, the best performance, and G. Wu et al. [7] showed that the PNGL is better than the prefix-based scheme. Therefore, as an indicator of the acceptable graph decomposition, we consider the performances of the basic PNGL and the prefix-based scheme.

Table 1 summarizes some important experimental parameters. First, a tree as in Fig. 6 (a) is randomly generated according to $\#n$ and $\#fo$, and then a test DAG as in Fig. 6 (b) is generated according to $\#fi$. For example, in Fig. 6 (a), $\#n$ is twelve and $\#fo$ is three since each node has three child nodes. In Fig. 6 (b), $\#fi$ is about two since each node has two parent nodes on the average. Since the PNGL is suitable for few data, we tested it within $\#n = 1000$ and within $\#fo = 20$. We used a *long* variable (eight bytes) in Java for each of the prime number labels $L1$, $L2$, and $L3$. For the decomposition optimization, we used one additional byte to identify sub-DAGs, which is indexed by a B-tree. As explained in Sect. 5.2, we used just four bytes among the allocated eight bytes to cope with later additional insertions. For the prefix-based scheme, we used the unicode string and adopted the relational representations of UPrefix in the Ref. [6]. All experiments were performed on a Windows XP computer with 1 GB of memory and 3.0 GHz Pentium(R) IV CPU. All codes were written in Java, and MS SQL server 2000 (Personal Edition) was used as a database server.

6.2 Label Extension

We first analyze how much labels can be additionally assigned by the decomposition optimization. Table 2 shows that the basic PNGL has many cases of label overflow. Even in the case of $\#n = 100$ and $\#fo = 5$ which has small nodes, label overflow occurred. However, our suggested decomposition optimization prevents the label overflow with some division. The measured numerical values in the table represent N-Decomposition. In addition, it shows that the decomposition optimization by BFT has much less division than DFT. This is because DFT is most likely to have much bigger $L2$ labels, as mentioned in Sect. 3. We can also see that almost every column of the basic PNGL has the label overflow when the fan-out is small. This is because, for an equal number of graph nodes, the smaller the fan-out of a DAG, the deeper the depth of the DAG. When the depth of a DAG is significant, it is most likely to have big prime number labels.

6.3 Label Size and Construction Time

In this subsection, we analyze the label construction time of the decomposition optimization and its storage cost for saving labels. First, when $\#fo = 20$, we measured the label construction time of each method. The graph of Fig. 7 (a) compares the construction times of the basic PNGL, the decomposition optimization, and the prefix scheme. We can see that the decomposition optimization has a similar construction time to the prefix scheme. Through some additional experimental results as in Fig. 7 (b) (c), we verified that the construction time of the decomposition optimization is similar or superior to the prefix scheme. In the graphs of Fig. 7 (b) (c), we cannot measure the construction time of the basic PNGL since it has a label overflow.

Regarding the comparison of label size, the graphs of Fig. 8 show that the decomposition optimization consumes less storage space than the prefix scheme. Similarly, in the graphs of Fig. 8 (b) (c), we cannot measure the basic PNGL since it has a label overflow.

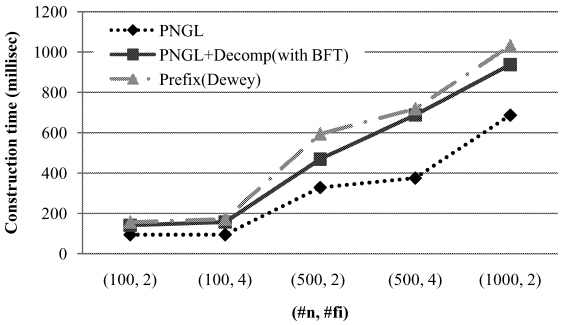
6.4 Response Time of Typical Operations

In this subsection, we evaluate the response time of typical operations according to the number of divisions. We tested the primary operations *descendant*(v), *ancestor*(v), and *sibling*(v) where v represents an arbitrary vertex. The measured times in Fig. 9 represent the average execution time of each operation for randomly selected vertices. For example, when v_1, v_2, v_4, v_7, and v_{10} are selected in the DAG of Fig. 6, a measured time in Fig. 9 (a) is the average of the execution times of *descendant*(v_1), *descendant*(v_2), *descendant*(v_4), *descendant*(v_7), and *descendant*(v_{10}). We ensured vertices were selected evenly for the whole graph.

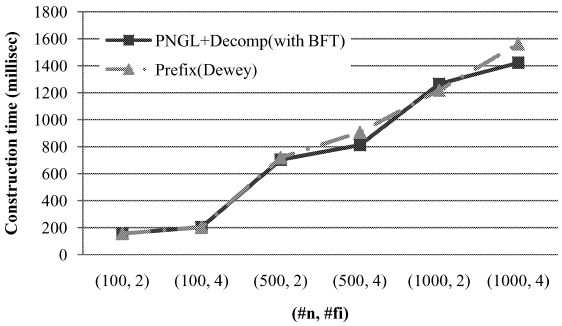
The graph of Fig. 9 (a) shows the comparison for the

Table 2 Label extension by the decomposition optimization
('×': label overflow, '-': non label overflow, figures: N-Decomposition).

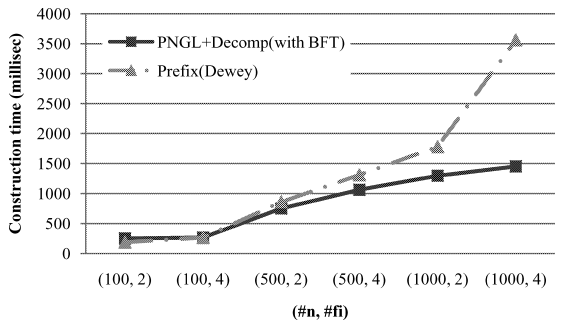
$\#fo$	5						10						20					
$\#n$	100		500		1000		100		500		1000		100		500		1000	
$\#fi$	2	4	2	4	2	4	2	4	2	4	2	4	2	4	2	4	2	4
PNGL	×	×	×	×	×	×	-	-	×	×	×	×	-	-	-	-	-	×
PNGL+Decomp (with BFT)	1	2	2	3	3	3	1	1	1	2	2	2	0	0	1	1	1	1
PNGL+Decomp (with DFT)	2	2	2	4	3	4	1	1	2	2	2	3	0	0	1	1	2	2



(a) $\#fo = 20$

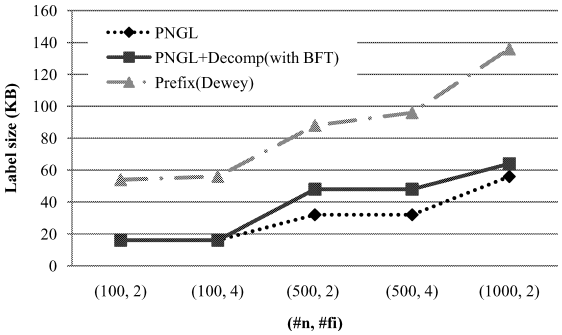


(b) $\#fo = 10$

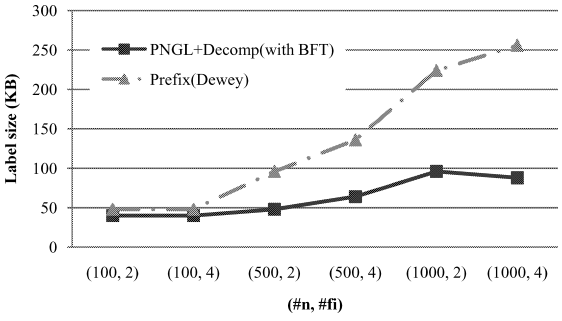


(c) $\#fo = 5$

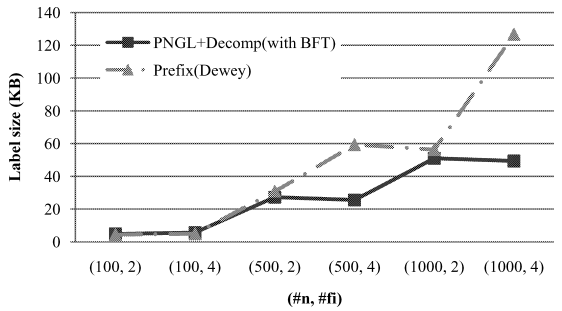
Fig. 7 Label construction time.



(a) $\#fo = 20$



(b) $\#fo = 10$



(c) $\#fo = 5$

Fig. 8 Total label size.

descendant operation according to the number of divisions. Note that the X-axis represents N-Decomposition only for 'PNGL+Decomp (with BFT)'. We can see that 1-Decomposition has a performance similar to the other two schemes, and 2-Decomposition has slightly worse performance than the prefix scheme. Additionally, the expense of 3-Decomposition is high. In the cases of 2-Decomposition and 3-Decomposition, we cannot measure the basic PNGL due to label overflow.

The graph of Fig. 9 (b) shows the comparison for the *ancestor* operation. We can see that 1-Decomposition and 2-Decomposition has a better or similar performance compared with the other two schemes and 3-Decomposition has slightly worse performance. In the case of 1-Decomposition, since the decomposition optimization performs the *ancestor* operation just for the corresponding sub-DAGs, the execution time can be lower than the basic PNGL. For example, in Fig. 2 (b), *ancestor*('E') is performed just for the sub-DAG_1 rather than the whole graph. The graph of Fig. 9 (c) is for the *sibling* operation. We can see that all decompositions have a similar performance to the prefix scheme. In the case of 1-Decomposition, the ba-

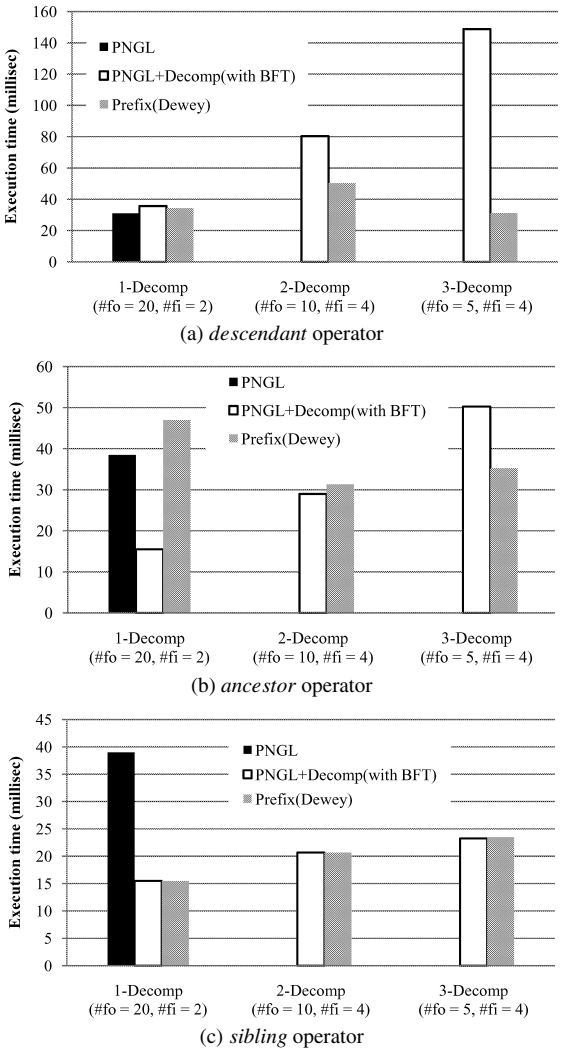


Fig. 9 Response time of typical operations according to *N*-Decomposition (#*n* = 1000).

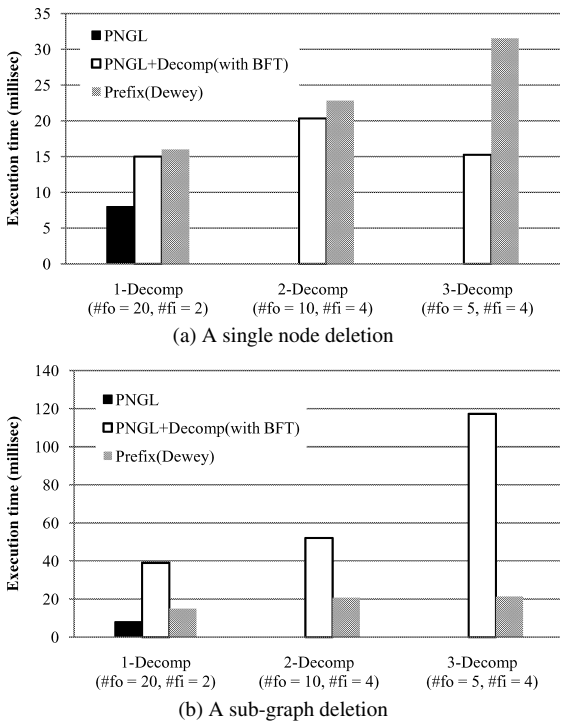


Fig. 10 Re-labeling cost according to *N*-Decomposition (#*n* = 1000).

sic PNGL has slightly worse performance since the *sibling* operation is performed for the whole DAG.

Through the above experiments, we believe that 1-Decomposition has a similar performance to the other two schemes and 2-Decomposition is acceptable although it has slightly worse performance in the *descendant* operation. From 3-Decomposition on, the decomposition optimization is not recommended. In Table 2, the decomposition optimization with BFT is not practical for the cases of (#fo: 5, #*n*: 500, #fi: 4) and (#fo: 5, #*n*: 1000, #fi: 2 or 4) due to 3-Decomposition.

6.5 Re-Labeling Performance

In this subsection, we evaluate the re-labeling performance for updates. First, the graph of Fig. 10 (a) shows the comparison of deleting a specific node. As in explained in Sect. 5.1, the decomposition optimization performs better than the prefix scheme since the re-labeling is performed just for the corresponding sub-DAG. From this result, we can also estimate the comparison result for the insertion operation. This is because, as explained in Sect. 5.2, an insertion also is allowed just within the corresponding sub-DAG. Next, the graph of Fig. 10 (b) shows the comparison for a special deletion operation which removes a specific node and its descendant nodes together. In this case, the re-labeling performances of the decomposition optimization are slightly worse than the prefix scheme in 1- or 2-Decomposition. 3-Decomposition is not desirable.

7. Conclusions

When the data to be labeled are few, the prime number graph labeling provides a novel label mechanism by using only elementary arithmetic calculation such as multiplication, division, and modulo. However, there is an inherent problem of label overflow. In this paper, we have introduced another optimization technique based on graph decomposition for minimizing the problem. The suggested optimization technique can effectively extend the range of prime number labeling. Experimental results have shown that just 1- or 2-Decomposition has a better performance than the prefix-based labeling scheme. However, even within 1- or 2-Decomposition, we can perform quite a lot of additional labeling.

Acknowledgments

We would like to thank the anonymous referees for their valuable comments on earlier draft of this paper.

This study is supported in part by the Second Stage of

BK21.

References

[1] Q. Li and B. Moon, "Indexing and querying XML data for regular path expressions," Proc. 27th International Conference on Very Large Databases, pp.361–370, Sept. 2001.
[2] H. Kaplan, T. Milo, and R. Shabo, "A comparison of labeling schemes for ancestor queries," Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms, pp.954–963, Jan. 2002.
[3] X. Wu, M. Lee, and W. Hsu, "A prime number labeling scheme for dynamic ordered XML trees," Proc. 20th International Conference on Data Engineering, pp.66–78, April 2004.
[4] J.G. Lee, K.Y. Whang, W.S. Han, and I.Y. Song, "The dynamic predicate: Integrating access control with query processing in XML databases," The VLDB J., vol.16, no.3, pp.371–387, July 2007.
[5] S. Yokoyama, M. Ohta, K. Katayama, and H. Ishikawa, "An access control method based on the prefix labeling scheme for XML repositories," Proc. 16th Australasian Database Conference (ADC2005), pp.105–113, 2005.
[6] V. Christophides, G. Karvounarakis, D. Plexousakis, M. Scholl, and S. Tourtounis, "Optimizing taxonomic semantic web queries using labeling schemes," J. Web Semantics, vol.11, no.1, pp.207–228, Nov. 2003.
[7] G. Wu, K. Zhang, C. Liu, and J. Li, "Adapting prime number labeling scheme for directed acyclic graphs," Proc. Database Systems for Advanced Applications (DASFAA), pp.787–796, April 2006.
[8] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl, "RQL: A declarative query languages for RDF," Proc. 11th International World Wide Web Conference, pp.592–603, May 2002.
[9] S. Zheng, J. Wen, and H. Lu, "Cost-driven storage schema selection for XML," Proc. Database Systems for Advanced Applications (DASFAA), pp.337–344, March 2003.
[10] P. Bohannon, J. Freire, P. Roy, and J. Simeon, "From XML schema to relations: A cost-based approach to XML storage," Proc. 18th International Conference on Data Engineering, pp.64–75, Feb. 2002.
[11] M.R. Garey, D.S. Johnson, and L.J. Stockmeyer, "Some simplified NP-complete graph problems," Theor. Comput. Sci., vol.1, no.3, pp.237–267, 1976.
[12] RDF Primer, W3C Recommendation, http://www.w3.org/TR/rdf-primer/

Appendix: The Suggested Decomposition Optimization Algorithm

```
class Node
{
  long L1;
  long L2;
  long L3;
  long[ ] PV_gID = null;
  long[ ] PV_L1 = null;
  long[ ] PV_L2 = null;
  boolean visit_flag = false;
  boolean store_flag = false;
};


function Decomp_PNGL (R, gID)
{
1 Node tmpn1, tmpn2;
2 Queue Q;
3 create a virtual root node VR with the label <1, 1, 1>;
4 Q.addq(R)
5 while((tmpn1 = Q.deleteq()) != null){   /* This first scanning of a DAG assigns a label to each node. */
6   while(tmpn2 = tmpn1.nextChild()){ /* visit the child nodes of tmpn iteratively */
7     if(tmpn2.visit_flag == false){
8       tmpn2.visit_flag = true;
9       if(tmpn1.L1 == 0){
10        tmpn2.L1 = tmpn2.L2 = tmpn2.L3 = 0;
11        Q.addq(tmpn2); continue;
12      }
13      assign the L1 value to tmpn2;
14      if(tmpn2.L1 is overflow){
15        make tmpn2 be the child of VR;
16        tmpn2.L1 = tmpn2.L2 = tmpn2.L3 = 0;
17        Q.addq(tmpn2); continue;
18      }
19      assign the L2 value to tmpn2 using tmpn2.L1 and tmpn1.L2;
20      if(tmpn2.L2 is overflow){
21        make tmpn2 be the child of VR;
22        tmpn2.L1 = tmpn2.L2 = tmpn2.L3 = 0;
23        Q.addq(tmpn2); continue;
24      }
25      assign the L3 value to tmpn2 using tmpn1.L1;
26      Q.addq(tmpn2);
27    }else{   /* if tmpn2 is visited */
28      if(tmpn1.L1 == 0){
29        tmpn2.L1 = tmpn2.L2 = tmpn2.L3 = 0;
30        Q.addq(tmpn2); continue;
31      }
32      if(tmpn2.L1 != 0){
33        assign the L2 value to tmpn2 using tmpn2.L2 and tmpn1.L2;
34        if(tmpn2.L2 is overflow){
35          make tmpn2 be the child of VR;
36          tmpn2.L1 = tmpn2.L2 = tmpn2.L3 = 0;
37          Q.addq(tmpn2); continue;
38        }
39        if(tmpn2.L3 % tmpn1.L1 != 0)
40          assign the L3 value to tmpn2 using tmpn2.L3 and tmpn1.L1;
41        Q.addq(tmpn2);
42      }
43    }
44  }
45 }
46 Q.addq(R)
47 while((tmp1 = Q.deleteq()) != null){   /* This second scanning of a DAG stores the label of each node into a stable storage. */
48   while(tmpn2 = tmpn1.nextChild()){ /* visit the child nodes of tmpn iteratively */
49     if(tmpn2.store_flag == false){
50       tmpn2.store_flag = true;
51       if(tmpn2.L1 != 0){
52         store the label of tmpn2 into the LABEL table;
53         if(The values PV_gID, PV_L1, and PV_L2 of tmpn2 exist)
54             store the connection information (PV_gID, PV_L1, PV_L2, gID(tmpn2), L2(tmpn2)) into the BRIDGE table;
55         Q.addq(tmpn2);
56       }else{
57         if(tmpn1.L1 != 0 && tmpn1.L1 != 1){
58           assign the values PV_gID, PV_L1, and PV_L2 to tmpn2;
59           Q.addq(tmpn2);
60       }
```

```
61        }
62      }else{   /* if tmpn2 is stored */
63        if(tmpn1.L1 != 0 && tmpn1.L1 != 1 && tmpn2.L1 == 0)
64          assign the values PV_gID, PV_L1, and PV_L2 to tmpn2;
65      }
66    }
67 }
68 gID++;
69 if(VR has child nodes)
70   Decomp_PNGL(VR, gID);
71 }
```

Jaehoon Kim received the B.S. and M.S. degrees in computer science from Konkuk University in 1997 and 1999, respectively and the Ph.D. degree in computer science from Sogang University in 2005. He worked as a senior researcher at Telecommunication R&D center in Samsung Electronics Company from 2005 to 2006. From 2006 to 2009, he was a research professor of computer science at Sogang University. From 2009, he is a professor of Dept. of information communication at Seoil University. His major research areas are database, database security, contextual information management in ubiquitous computing, and Semantic Web.

Seog Park received the B.S. degree in computer science from Seoul National University, Korea, in 1978, the M.S. and the Ph.D. degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 1980 and 1983, respectively. He is a professor of computer science at Sogang University, Seoul, Korea. Since 1983, he has been working in the Department of Computer Science of the College of Engineering, Sogang University. His major research areas are database security, real-time systems, data warehouse, digital library and web database. Dr. Park is a member of the IEEE Computer Society, ACM, and the Korea Information Science Society. Also, he has been a member of Database Systems for Advanced Applications (DASFAA) steering committee since 1999.