6

Prime Number Labeling Scheme for Transitive Closure Computation

Gang Wu [1] and Juanzi Li [2]
[1] *Institute of Web Science, Southeast University, Nanjing 210096, P.R.China*
[2] *Knowledge Engineering Group, Tsinghua University, Beijing 100084, P.R.China*

1. Introduction

In the context of Semantic Web, subsumption, i.e., inference of implicit subclass relationship, owl:TransitiveProperty, and owl:inverseOf, is a kind of simple yet basic description logic reasoning [7]. It is usually solved as a transitive closure computation problem [4]. Directed Graph is an effective data structure for representing such subsumption hierarchies While, the growing number and volume of directed graphs involved greatly inspire the demands for appropriate index structures.

Labeling scheme[9] is a family of technologies widely used in indexing tree or graph structured data. It assigns each vertex a well-designed label with which relationship between any two vertices can be detected or filtered efficiently. This chapter concerns only about labeling scheme among diverse index technologies considering its avoiding expensive join operation for transitive closure computation. Determinacy, compaction, dynamicity, and flexibility are factors for labeling scheme design besides speedup [11]. However, the state of art labeling schemes for directed graph could not satisfy most above requirements at the same time. Even approaches for the directed acyclic graph (DAG) is few.

One major category of labeling schemes for DAG is spanning tree based. Most of them are developed from their tree versions. The first step of labeling is to find a spanning tree and assigning labels for vertices according to tree's edges. Next, additional labels are propagated to record relationships represented through non-tree edges. Christophides et al. surveyed and compared two such schemes [4], i.e. interval-based [8] and prefix-based [3]. Whereas, the weak point of above schemes is obvious. Evaluations to the relationships implied by non-tree edges cannot take advantage of the deterministic tree label characters. Non-tree labels need not only additional storage but also special efforts in query processing. Also interval-based scheme studied in [4] has a poor re-labeling ability for updates.

There are also labeling schemes having no concern with spanning tree. Such as bit vector [10] and 2-hops [5]. Though bit vector can process operations on DAG more efficiently, it is static and requires global rebuilding of labels when updates happen. Moreover, studies show that recent 2-hops approach introduces false positives in basic reachability testing.

A novel labeling scheme for XML tree depending on the properties of prime number is proposed in [11]. Prime number labeling scheme associates each vertex with a unique prime number, and labels each vertex with the product of multiplying parents' labels and the prime number owned by the vertex. The effect of updating on vertices is almost the same to

that in prefix-based scheme. Moreover, the response time for queries and the size requirements are even smaller than those of prefix-based scheme. However no further work has been performed on extending the idea of prime number labeling scheme to the case of DAG.

In this research we expect to find out a labeling scheme for DAG to augment performance in all of the above requirements as much as possible. By taking a strong connected component in the graph as one vertex, arbitrary directed graphs (with cycles) can be treated with the proposed labeling scheme. We are stimulated by the virtues of prime number labeling scheme exhibited in indexing XML tree. We extend it by labeling each vertex in a DAG with an integer which equals to the arithmetic product of the prime number associating with the vertex and all the prime numbers associating with its ancestors. The scheme does not depend on spanning tree. Thus subsumption hierarchies represented in a DAG can be efficiently explored by checking the divisibility among the labels. It also inherits dynamic update ability and compact size feature from its predecessor. The major contributions are as follows.

- Extend original prime number scheme[11] for labeling DAG and to support the processing of typical operations on DAG.
- Optimize the scheme in terms of the characteristics of DAG and prime numbers. Topological sort and Least common multiple are used to prevent the quick expansion of label size; Leaves marking and descendants-label improve the performance of querying leaves and descendants respectively.
- A generator is implemented to generate arbitrary complex synthetic DAG for the extensive experiments. Space requirement, construction time, scalability, and impact of selectivity and update are all studied in the experiments.

Results indicate that prime number labeling scheme is an efficient and scalable scheme for indexing DAG with appropriate extensions and optimizations.

2. DAG and Typical Operations

Given a finite set V and a binary relation E on V, a *directed graph* can be represented as G = (V, E). V and E consist of all the *vertices* and *edges* in G respectively. For any pair of vertices u and u' in G, vertices sequence $< v_0, v_1, v_2, \ldots, v_k >$ is called a *path*, if $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k + 1$. A directed graph is a *directed acyclic graph* (DAG) if there is no path returning to the same vertex. Figure 1(borrowed from [4]) is a DAG.

Reachability is a basic concept in DAG. Given two vertices v and w, if there exists a path p from v to w, we say that w is reachable from v via p, or $v \overset{p}{\rightsquigarrow} w$ (or as $v \rightsquigarrow w$ out of consideration of p). Known relations, such as *parent*, *child*, *ancestor*, *descendant*, *leaf*, *sibling*, *and nearest common ancestor(s) (nca)* are derived from this concept. Queries on these relations are typical operations on DAG. While, unlike trees, order-sensitive queries such as preceding/following are meaningless for DAG. In the following discussion, given vertices v and w in DAG G, we will use *parents(v)*, *children(v)*, *ancestors(v)*, *descendants(v)*, *leaves(v)*, *siblings(v)* and *nca(v, w)* indicating the above queries respectively (See [4] for formal expressions). Update, including vertices insertions and

deletions, is another kind of operation worthy of note because it usually brings reorganizations to the DAG storage and re-labelings to the index structure.

3. Prime Number Labeling Scheme for DAG

The first part of this section shows that the *divisibility* of integers still could be the evidence theory of the prime number labeling scheme for DAG. In the second part, we prove that all of the typical operations on DAG are solvable.

3.1 Prime Number Labeling Scheme for DAG - Lite

Few modifications are required to support DAG reachability testing.

Definition 1. *Let G = (V,E) be a directed acyclic graph. A* ***Prime Number Labeling Scheme for DAG - Lite****(PLSD-Lite for short) associates each vertex v ∈ V with an exclusive prime number p[v], and assigns to v a label* $L_{\text{lite}}(v)$ = (c[*v*]), *where*

$$c[v] = p[v] \cdot \begin{cases} \prod_{v' \in parents(v)} c[v'], & in-degree(v)>0 \\ 1, & in-degree(v)=0 \end{cases} \tag{1}$$

In Figure 1, PLSD-Lite assigns each vertex an exclusive prime number increasingly from "2" with a depth-first traversal of the DAG. The first multiplier factor in the brackets of each vertex is the prime number assigned.

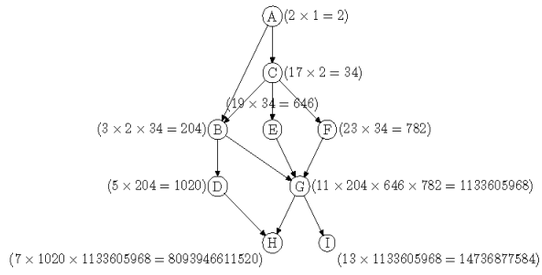

Fig. 1. Prime Number Labeling Scheme for DAG – Lite Scheme for DAG - Lite

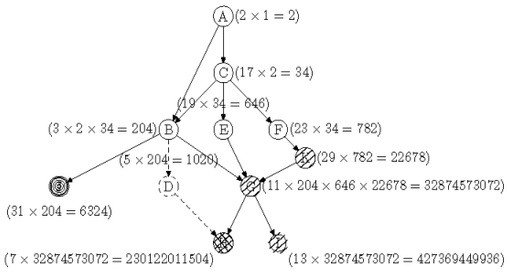

Fig. 2. Updates in Prime Number Labeling Scheme for DAG - Lite

Lemma 1. *Let G =* (V, *E*) *be a directed acyclic graph. Composite number c[v] in the* $L_{\text{lite}}(v)$ = (c[*v*]) *of a vertex v∈V can be written in exactly one way as a product ofthe form*

$$c[v] = p[v] \cdot \prod_{v' \in ancestors(v)} p[v']^{m_{v'}} \tag{2}$$

where $m_{v'} \in \mathbb{N}$.

Proof. Such a product expression can be constructed by performing transitive closure in terms of Definition 1. On the other hand, an integer has a unique factorization into primes, and hence the above product expression is unique. □

Lemma 1 implies that for any vertex in the DAG with PLSD-Lite, there is a bijection between an ancestor of the vertex and a prime factor of the label value.

Theorem 1. *Let G* = (V, E) *be a directed acyclic graph. For any two vertices v,w* ∈ *V where* $L_{\text{lite}}(v)$ = (c[*v*]) *and* L_{lite} *(w)* = (c[*w*]), $v \rightsquigarrow w \Leftrightarrow c[v] \mid c[w]$.

Proof. Let $r =< v_0, ..., v_k >$ be one of the path length *k* from vertex *v* to vertex *w*, where $v_0 = v$, $v_k = w$ and $k >= 1$. By the definition of reachability, vertex *v* must be an ancestor of vertex *w* on r. Suppose $in - degree(w) > 0$, by Definition 1, composite number c[*w*] of label $L_{lite}(w)$ = (c[*w*]), could be represented as $c[w]=\left(\prod_{i=1}^{k} p[v_i]\right)\cdot c[v_0]\cdot\prod_{v'\in parents(w)\wedge v'\neq v_{k-1}} c[v']$. Since $v_0 = v$, we conclude that $c[v] \mid c[w]$. On the other hand, $c[v] \mid c[w]$ implies c[*w*] = *k*' · *c[v]* for some integer k'. By Lemma 1, then we have that $c[w] = k' \cdot p[v] \cdot \prod_{v'\in parents(v)} p[v']^{m_{v'}}$. This factorization of *c[w]* implies that *p[v]* is a factor of *c[w].* Therefore, vertex *v* is one of the ancestors of vertex *w.* The reachability from *v* to *w* is obvious. □

A consequence of Theorem 1 is that whether two vertices have the relation of ancestor/descendant can be simply determined with PLSD-Lite. For example, in Figure 1, we have $A \rightsquigarrow D$ because $2 \mid 1020$. Whereas there is no ancestor/descendant relation between *F* and *D* because $782 \nmid 1020$. In this way, finding out all the ancestors or descendants of a given vertex is realizable by testing the divisibility of the vertex's label with the other vertices' labels in the DAG or conversely. Averagely *(N* - 1)/2 divisibility testings have to be carried out to retrieve all the ancestors or descendants for any given vertex in a *N* vertices DAG. We can determine that *D* has three ancestors *B*, *C* and *A* by examining divisibility of each vertex that has label value less than "1020". Vertex *E* and *F* are not the ancestors of *D* because their label values cannot divide 1020. Moreover, a vertex is a leaf if any other vertex's label value could not be divided by its label value. There is also a naive solution to nca evaluating according to the definition of nca and Theorem 1. First put all the common ancestors of both vertices into a set. Then filter out vertices whose descendants are also within the set. Remainings in the set are the nca of the vertices.

PLSD-Lite is also a fully dynamic labeling scheme in the presences of updates as stated in [11]. Re-labeling happens with the insertion or deletion of a vertex, and only affects the descendants of the newly inserted vertex or the deleted vertex. After deleting vertex *D*, inserting leaf vertex *J* and non-leaf vertex *K*, we have Figure 2. As a new leaf, vertex *J* does not affect other vertex in the DAG. Insertion of vertex *K* only affects descendants G, *H*, *I* and *K* itself. Vertex *H* is affected by the deletion of ancestor *D* at the same time.

However PLSD-Lite lacks enough information to identify parents/child relation, not to mention finding all the siblings of a given vertex.

3.2 Prime Number Labeling Scheme for DAG - Full

In order to support all of the operations for DAG, PLSD-Lite should be extended by separately recording the prime number that identifies the vertex and the additional information about parents.

Definition 2. *Let G* = (V, E) *be a directed acyclic graph. A* ***Prime Number Labeling Scheme for DAG - Full****(PLSD-Full for short) associates each vertex v ∈ V with an exclusive prime number p[v], and assigns to v a label* L_{full} (v) = *(p[v],* c_a*[v],* c_p*[v]), where*

$$c_a[v] = p[v] \cdot \begin{cases} \prod_{v' \in parents(v)} c_a[v'], & in-degree(v)>0 \\ 1, & in-degree(v)=0 \end{cases} \quad (3)$$

$$c_p[v] = \begin{cases} \prod_{v' \in parents(v)} p[v'], & in-degree(v)>0 \\ 1, & in-degree(v)=0 \end{cases} \quad (4)$$

We term *p[v]* as "self-label", c_a[*v*] as "ancestors-label"(also c[*v*] in Definition 1), and c_p[*v*] as "parents-label". In Figure 3, three parts in one bracket is self-label, ancestors-label, and parents-label. Theorem 1 is still applicable. Moreover, all the operations on DAG are supported by the following theorem and corollary.

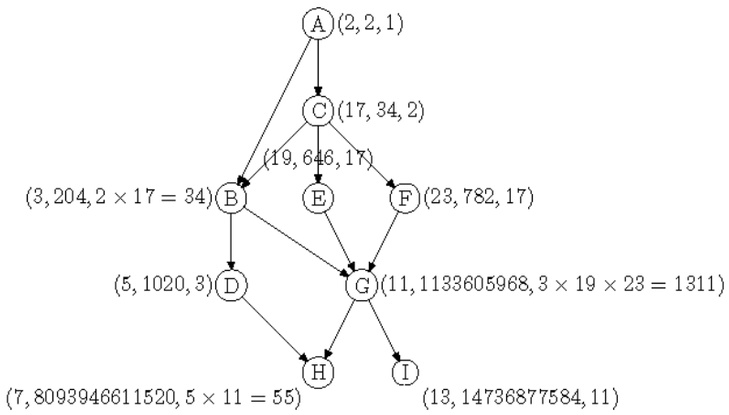

Fig. 3. Prime Number Labeling Scheme for DAG - Full

Theorem 2. *Let G* = (V, E) *be a directed acyclic graph, and vertex v ∈V has* L_{full}*(v)* = (p[*v*], c_a[*v*], c_p[*v*]). *If the unique factorization of composite integer* c_a*[v] results r different prime numbers,* $p_1 < ... < p_r$*, then there is exactly one vertex w ∈ V that takes pi as the self-label for* $1 \le i \le r$*, and w is one of the ancestors of v. If the unique factorization of composite integer* c_p*[v] results s different prime numbers,* $p'_1 < ... < p'_s$*, then there is exactly one vertex u ∈ V that takes* p'_i *as the self-label for* $1 \le i \le s$*, and u is one of the parents of v.*

The proof of Theorem 2 is obvious according to Lemma 1, the definition of parents-label and the unique factorization property of integer. It implies that we can find out all the parents of any vertex by factorizing the parents-label. For instance, since vertex *G* in Figure 3 has a parents-label 1311 = 3 x 19x23, vertices *B*, *E* and *F* are considered to be all the parents of *G*. We still have the rights to determine the parent/child relation of two vertices by checking divisibility between one's parents-label and the other's self-label in terms of Definition 2. Corollary 1 further expresses PLSD-Full's sibling evaluation ability.

Corollary 1. *Let G = (V, E) be a directed acyclic graph. For any two vertices v,w ∈V where* $L_{full}(v)$ = (p[*v*], c_a[*v*], c_p*[v]) and* $L_{full}(w)$ = (p[*w*], c_a[*w*], c_p*[w]), w and v are siblings ifand only ifthe greatest common divisor oftheir parents-label,* gcd(c_p*[v],* c_p*[w])* ≠ 1.

Proof. Suppose vertex *u*∈*V* is one of the parents of both *v* and *w* where $L_{full}(u)$ = (p[*u*], c_a[*u*], c_p[*u*]). Then c_p[*v*] = *ki* c_p*[u]* and c_p[*w*] = *ki* c_p[*u*] hold where $k_i > 1, k_2 > 1 \in \mathbb{N}$. Therefore, c_p[*u*] is the common divisor of c_p[*v*] and c_p*[w]* and hence the greatest common divisor gcd(c_p*[v],* c_p*[w])* ≠ 1 since c_p*[u]* ≠ 1. On the other hand, let gcd(c_p*[v],* c_p*[w])* ≠1 be the greatest common divisor of c_p[*v*] and c_p[*w*], which implies that there exist *r* ≥1 prime numbers $p_1^{\varepsilon_1} p_2^{\varepsilon_2} \ldots p_r^{\varepsilon_r}$ = gcd(c_p[v], c_p[w]). According to Theorem 2, the vertices *v* and *w* have a set of parents whose self-labels are prime numbers $p_1, p_2, \ldots, p_r$ respec-tively. Consequently, we conclude that vertices *v* and *w* are siblings. □

Corollary 1 enables us to discover the siblings of a vertex by testing whether the greatest common divisor of the parents-labels equals 1. In Figure 3, vertex *B* have two siblings *E* and *F* because gcd(34, 17) = 17 ≠1.
Theorem 2 provides us another measure to obtain ancestors besides doing divisibility testing one vertex after another. By applying unique factorization to the ancestors-label of vertex *D* in Figure 3, three ancestors *A*, *B* and *C* are thus identified by prime factors "3", "2" and "17" respectively. Though trial division itself could be used to do integer factorization, we can choose faster integer factorization algorithm alternately especially for small integers.

4. Optimization Techniques

As shown above, PLSD could perform all typical operations on DAG with elementary arithmetic operations such as divisibility testing, greatest common divisor evaluating, and integer factorization. Because these elementary arithmetic operations become time-consuming while their inputs are large numbers, running time is usually estimated by measuring the count of bit operations required in a number-theoretic algorithm. In other words, the more number of bits are required to represent the labels of PLSD, the more time will be spent on the operations. In this section we will introduce several optimization techniques to eliminate the count and the size of the prime factors involved in the multiplication for label generation of our scheme. Also another one is proposed at the end of this section as a complementarity of PLSD for querying descendants.

4.1 Least Common Multiple

In previous definitions, the value of a vertex's ancestors-label is constructed from multiplying its self-label by the parents' ancestors-labels. However, there is apparent redundancy in this construction of ancestors-label that power m_v' in Equation 2 magnifies the size of ancestors-label exponentially, but it is helpless for evaluating the operations of DAG. It is straightforward to remove the redundancy by simply setting m_v' to 1 in Equation 2. We have Equation 5 below.

$$c[v] = p[v] \cdot \prod_{v' \in ancestors(v)} p[v'] \tag{5}$$

The simplification is reasonable because in this case Theorems 1 and 2 still hold. Define $lcm(a_1,a_2,\ldots,a_n)$ to be the *least common multiple* of n integers $a_1, a_2,\ldots, a_n$. In particular, for an integer a we define $lcm(a) = a$ here. Thereafter we have the following equation for ancestors-label construction.

$$c[v] = p[v] \cdot \begin{cases} lcm(c[v'_1],\ldots,c[v'_n]), & in-degree(v)>0 \ and \ v'_1,\ldots,v'_n \in parents(v) \\ 1, & in-degree(v)=0 \end{cases} \tag{6}$$

The equivalence between Equation 5 and 6 can be proved with the property of least common multiple apparently. Equation 6 implies that an ancestors-label can be simply constructed by multiplying self-label by the least common multiple of all the parents' ancestors-labels. Thereafter, the value of an ancestors-label is the arithmetic product of its ancestors' as described in Equation 5. With this optimization technique, the max-length of ancestors-label in DAG is only on terms with the total count of vertices and the count of ancestors. Comparing with Figure 3, Figure 4 has a smaller max-length of ancestors-label.

4.2 Topological Sort

Previous selection of prime number for the self-label of a vertex is arbitrary only on condition that no two vertices have the same self-label. A naive approach is assigning each vertex met in depth-first search of DAG a prime number ascend-ingly. Unfortunately, Equation 5 and 2 imply that the size of a vertex's self-label has influence on all the ancestors-labels of its descendants. So vertices on the top of the hierarchy should be assigned small prime numbers as early as possible. Topological sort of a DAG can solve the problem.

"A *topological sort* of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering."[6]. Thus assigning prime numbers ascendingly to vertices with this ordering results in small self-labels precedences in the hierarchy. One of the topological sort of the DAG in Figure 1 is *"A, C, E, F, B, D, G, H,* I". Let the self-labels to be the first 9 prime numbers "2, 3, 5, 7,11,13,17,19, 23" respectively, then we get Figure 5.

4.3 Leaves Marking

As an optimization for reducing label size, even numbers such as $2^1, 2^2,\ldots, 2^n$ are used as self-labels for leaf vertices in [11], which gives us another method to identify leaves. However, the prime number theorem indicates that the growth of prime number is slower than that of power of 2, so self-labels of even number leaves increase dramatically. An alternative is to follow the rule of PLSD-FULL and simply setting leaf's ancestors-label to be negative. Then whether a vertex is a leaf could be determined by the sign of its ancestors-label. It is a meaningful technique in the case of existing large number of leaves in a DAG.

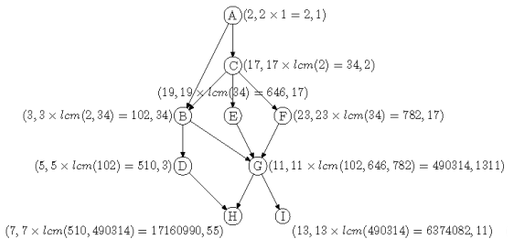

Fig. 4. PLSD-Full with Least Common Multiple Optimization

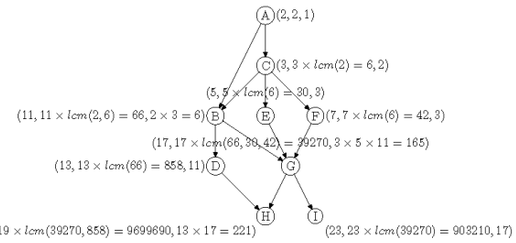

Fig. 5. PLSD-Full with Topological Sort Optimization

4.4 descendants-label

Divisibility testing and unique factorization both can be used for querying ancestors as discussed in section 3.It is feasible to spend more storage space on adding another label for helping to evaluate *descendants(v)* in consideration of existing timesaving integer factoring algorithms. In the same idea of ancestors-label, we extend PLSD-Full by adding the following so-called "descendants-label".

$$c_a[v] = p[v] \cdot \begin{cases} \prod_{v' \in children(v)} c_a[v'], & out{-}degree(v){>}0 \\ 1, & out{-}degree(v){=}0 \end{cases} \tag{7}$$

Clearly, Equation 7 is just like Equation 3 except in the reverse hierarchy. Now descendants query, *descendants(v)*, can be evaluated by factoring descendants-label in the same way provided by Theorem 2. In section 5 we will give empirical results on querying descendants and leaves using this technique.

5. Performance Study

This section presents some results of our extensive experiments conducted to study the effectiveness of prime number labeling scheme for DAG (PLSD).

5.1 Experiment Settings

Taking the queries on RDF class hierarchies as an application background for DAG, we setup test bed on top of RDF Schema Specific DataBase(RSSDB v2.0) [2], which is a persistent RDF store that generates an Object-Relational (SQL3) representation of RDF metadata. In this case, each vertex in a DAG stands for a class in the RDF metadata, and each edge in a DAG stands for the hierarchy relationship between a pair of classes in the RDF metadata. In our configuration, RDF metadata is parsed and stored in PostgreSQL (win32 platform v8.0.2 with Unicode configuration) through the loader of RSSDB.

Though least common multiple, topological sort, and leaves marking are optional optimization techniques, they are integrated in our default PLSD-Full implementation. PLSD-Full without these optimizations and PLSD-Lite are ignored for their apparent defects. Furthermore, based on this default implementation of PLSDF-Full, descendants-label is employed to examine its effects on descendants query. We also provide the Unicode Dewey prefix-based scheme and the extended postorder interval-based scheme by Agrawal

et al. The former is an implementation to the descriptions in [4], and the latter is a complementary to the scheme released with the source code of RSSDB v2.0. Hence, there are totally four competitors in our comparisons, namely, default PLSD-Full (PLSDF), PLSD-Full with descendants-label (PLSDF-D), extended postorder interval-based scheme (PInterval) and Unicode Dewey prefix-based scheme (UP-refix). All the implementations are developed in Eclipse 3.1 with JDK1.5.0. Database connection is constructed with PostgreSQL 7.3.3 JDBC2 driver build 110.

The relational representations of UPrefix and PInterval, including tables, indexes, and buffer settings, are the same to those in [4]. As for PLSDF, we create a table with four attributes: *PLSDF (self - label* : *text*, *label* : *text*, *parent - label* : *text*, *uri* : *text).* It is not surprising that we use PostgreSQL data type *text* instead of the longest integer data type *bigint* to represent the first three attributes considering that a vertex with 15 ancestors has an ancestors-label value 32589158477190044730 (2 × 3 × 5 × 7 × 11 × 13 × 17 × 19 × 23 × 29 × 31 × 37 × 41 × 43 × 47 × 53) which easily exceeds the upper bound of *bigint* (8 bytes, between ±9223372036854775808). Fortunately, the conversion from text to number is available on host language Java. Thus the number-theoretic algorithms used for PLSDF could be performed outside PostgreSQL, and become main memory operations. Similarly, we use *PLSDF - D(self - label* : *text*, *label* : *text*, *parent - label* : *text*, *descendants - label* : *text*, *uri* : *text)* to represent PLSDF-D where attribute descendants-label is added. For PLSDF and PLSDF-D, we only build B-tree indexes on self-labels because of the limitation of B-tree on large size text column, though indexes are necessary on ancestors-label and parents-label. Buffer settings are the same to those of UPrefix and PInterval.

All the experiments are conducted on a PC with single Intel(R) Pentium(R) 4 CPU 2.66GHz, 1GB DDR-SDRAM, 80GB IDE hard disk, and Microsoft Windows 2003 Server as operating system.

5.2 Data Sets and Performance Metrics

To simulate diverse cases of DAG, we implement a RDF metadata generator that can generate RDF file with arbitrary complexity and scale of RDF class hierarchies. Generator's input includes 4 parameters that describe a DAG. They are the count of vertices, the max depth of DAG's spanning tree, the max fan-out of vertices, and the portion of fan-in *(ancestors/precedings).* The count of the edges changes with the adjustment to the above values. Though PLSDF and PLSDF-D depend only on the characters of DAG, parameters related to spanning tree are still employed here because Interval and UPrefix are all based on spanning tree. The output is a valid RDF file (conforming with W3C RDF/XML Syntax Specification) that satisfies the parameters. We concatenate the values of above four parameters and the count of edges with hyphens to identify a DAG.

RDF Metadata DAG	Size (MB)	Classes/ Vertices	SubClassOf/ Edges	Depth Max	Fan-out Max	Fan-in Portion	Fan-in Max
1300-8-4-0.2-50504	2.55	1300	50504	8	4	0.2	219
1300-8-4-0.4-100132	4.62	1300	100132	8	4	0.4	458
1300-8-4-0.6-149451	6.34	1300	149451	8	4	0.6	373
1300-8-4-0.8-199774	8.05	1300	199774	8	4	0.8	897
1300-8-4-1.0-250222	9.32	1300	250222	8	4	1.0	562
450-2-32-0.7-45525	2.78	450	45525	2	32	0.7	313
90000-16-2-0.000053-44946	16.3	90000	44946	16	2	0.000053	3

Table 1. Data Sets

Listed in Table 1, two groups of DAGs are generated for evaluating the performance of PLSDF, PLSDF-D, UPrefix, and PInterval. They are used for investigating the impacts of DAG size and shape respectively on the labeling schemes. Data in the first group indicates that the file size and the count of the edges are positively related, if we fix the other parameters. While the second group shows two DAGs with different shape of spanning trees.

5.3 Space Requirement and Construction Time

The first group of DAGs in Table 1 is used here. For space requirement, we has Figure 6(a) where PLSDF and PLSDF-D have much smaller average space requirement (size of both tables and indexes), and mild trend of increase. The underlying cause is twofold. First, PLSDF or PLSDF-D is so simple that it is composed of only one table, of whom the count of the tuples is just equal to the count of vertices in the DAG, and it has only one B-tree index built. In contrast, Interval and UPrefix both consist of three tables to record additional information besides spanning tree. Meanwhile, they need more indexes built on each table. Another cause is that all of the data type in the table of PLSDF or PLSDF-D is *text* which will be "compressed by the system automatically, so the physical requirement on disk may be less"[1]. On the other hand, Figure 6(b) illustrates that PLSDF and PLSFD-D have the same gentle tendency but less construction time to UPrefix, whereas the construction time of Interval is the worst. This can be explained with the different procedures of label constructions. PLSDF and PLSDF-D create labels on the fly while processing the RDF file. However, UPrefix or Interval has to wait to create additional labels for non-spanning tree edges until the whole spanning tree is constructed. It is obvious that the count of non-spanning tree edges impacts the space requirement and label construction time for UPrefix and Interval. Another observation is that PLSDF needs few space and construction time relative to PLSDF-D. This is reasonable considering that PLSDF-D equals to PLSDF plus descendants-label.

5.4 Response Time of Typical Operations

We present our experimental results of the typical operations on DAG in this section from several aspects.

Overall Performance DAG "9000-8-4-0.004-45182" is chosen to have an experience on overall performance. The operations are listed in Figure 7.

The total elapsed time are shown in Figure 8. Interval, UPrefix, PLSDF labeling schemes are tested for all of the five operations. Moreover, PLSDF-D is applied to Q2 and Q4 to examine the effectiveness of descendants-label, while Q1, Q3, and Q5 are not necessary for PLSDF-D because it is just the same to PLSDF in these operations. For the given selectivity, PLSDF processes all the operations faster than the others. PLSDF-D exhibits accepted performance in Q2 and Q4 as well. The reason is the concise table structure of PLSDF/PLSDF-D and computative elementary arithmetic operations which avoid massive database access. For instance, the evaluation of a vertex's ancestors includes only two steps. Firstly retrieve the self-label and ancestors-label of the vertex from the table. Next do factorization using the labels according to Theorem 2. Results are self-labels identifying the ancestors of the vertex. The only database access happens in the first step. In contrast, Interval and UPrefix need more database operations, such as join operations and nested queries (See [4]). Though it seems that PLSDF-D does not have advantage over PLSDF in this case, experiments in the next part will bring us elaborative effects of PLSDF-D in different selectivity.
Another observation is that UPrefix outperforms Interval in all of the typical operations, which conflicts with the results from [4]. The cause is that Interval generates more additional information to record non-spanning tree edges than UPrefix, which is counterevidence of the excellent speed of PLSDF/PLSDF-D.

Impact of varying DAG Shape and Selectivity Here we investigate the performance under different DAG shapes, i.e., DAG with short-and-fat spanning tree, and DAG

	Operation Type	Selectivity
Q1	Ancestors	2.53%
Q2	Descendants	20.08%
Q3	Siblings	2.98%
Q4	Leaves	38.67%
Q5	nea	0.011%

Fig. 7. Test Typical Operations for Overall Performance

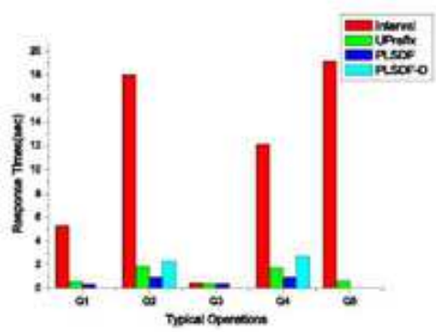

Fig. 8. Overall Performance

with tall-and-thin spanning tree. They are shown in Figure 9 to 10. Diagrams in each figure correspond to operations from Q1 to Q5 respectively. The metric of X-axis is the results selectivity of the operation except that the fifth diagram for nca uses X-axis to indicate the vertices' average length from the root of spanning-tree. The metric of Y-axis is the response time.

PLSDF displays almost constant time performance for all kinds of DAG shapes and operations. Because it does the chief computations with main memory algorithms instead of time-consuming database operations. The change of response time is indistinguishable in some extensions. The side effect is that PLSDF stays at a disadvantage at a very low selectivity especially for Q2 and Q4, e.g. in Figure 9(b) below selectivity 40%. Fortunately, PLSDF-D counterbalances this difficulty by trading off time to space with descendants-label. PLSDF-D almost has the same effect to UPrefix. Thus, it is a better plan to choose PLSDF-D at a low selectivity and switch to PLSDF when the selectivity exceeds some threshold. However, no good solution is found for PLSDF in Q3 where it costs more response time at a low selectivity. Interval and UPrefix could make use of the indexes on the parent label. Whereas PLSDF has to traverse among the vertices and compute greatest common divisor one at a time.

Scale-up Performance We carried out scalability tests of the four labeling schemes with the first group of DAGs in Table 1. Operations are made to have the equal selectivity (equal length on path for nca) for each scale of DAG size. Five diagrams in Figure 11 corresponds to operations from Q1 to Q5 respectively.

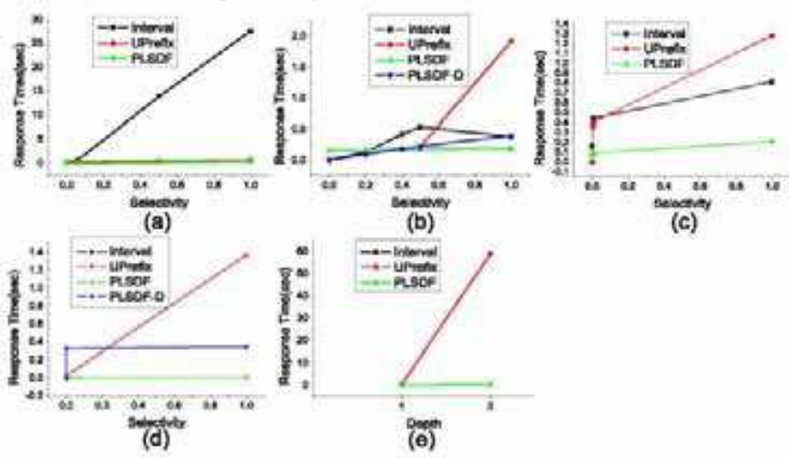

Fig. 9. DAG Shape and Selectivity of 450-2-32-0.7-45525

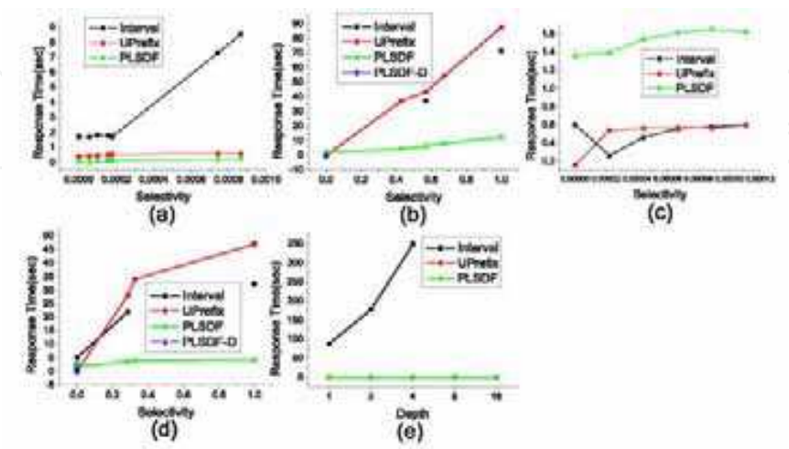

Fig. 10. DAG Shape and Selectivity of 90000-16-2-0.000053-44946

Interval and UPrefix are affected by both the size and the internal structure of the DAG (Note that the DAG is generated randomly). Unlike the other two labeling schemes, PLSDF and PLSDF-D perform good scalability in all cases.

5.5 Effect of Updates

To examine the dynamic labeling ability inherited from original prime number labeling scheme, we repeated the "Un-ordered Updates" experiments exhibited in [11], while "Order-Sensitive Updates" experiments are meaningless to our research subject. It is evident that updates on leaf vertices of DAG will have the same experimental results to that of XML tree with our analysis at the end of Section 3.1. Here we only give the results of updates on non-leaf vertices.

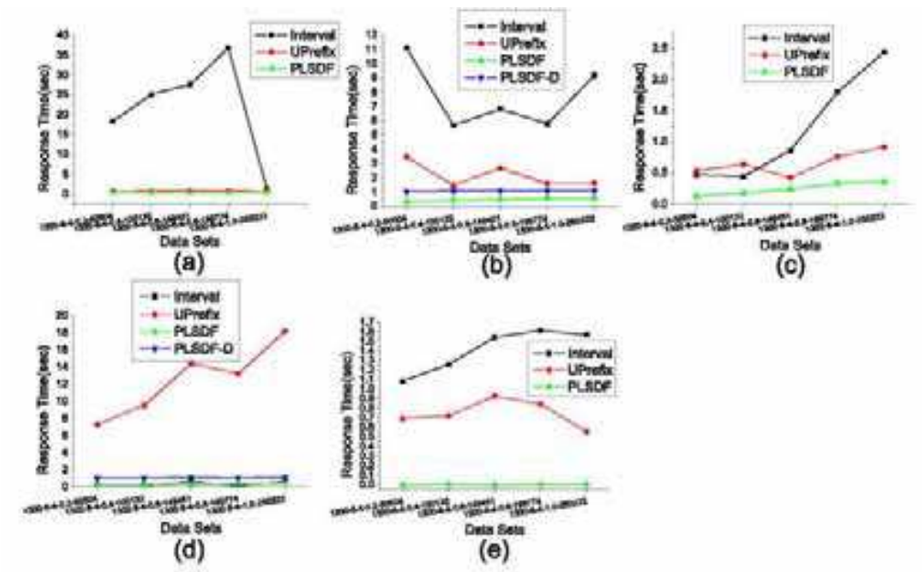

Fig. 11. Scale-up Performance

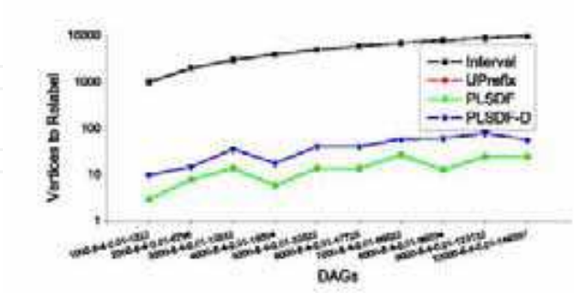

Fig. 12. Effect of Updates

Ten DAGs whose vertices increase from 1000 to 10000 are generated. We insert a new vertex into each DAG between bottom left leaf and the leaf's parent in the spanning tree. Figure 12 shows our experimental results for Interval, UPrefix, PLSDF, and PLSDF-D, which coincide with that of XML tree. PLSDF has exactly the same effect of update as Uprefix. While

additional label of PLSDF-D questionless causes more vertices, the ancestors vertices, to be re-labeled.

6. Prime Number Labeling Scheme for Arbitrary Directed Graph

DAG is a special kind of directed graph. For arbitrary directed graph, containing cycles is very common. The labeling scheme discussed above need some modifications to deal with arbitrary directed graphs. There are two steps to construct a more general labeling scheme. First, preprocess all the strong connected components in the graph. Then, construct prime number labeling scheme on the preprocessed graph.

Definition 3. *Let G = (V, E) be an arbitrary directed graph. A* ***Prime Number Labeling Scheme for Arbitrary Directed Graph****(PLSD-General for short) associates each vertex $v_i \in V$ with an exclusive prime number p_i, and assigns to v_i a label $L_{general}(v) = (p_i, ca_i, cp_i)$, where*

$$ca_i = lcm(\prod_{v_j \in comp(v_i)} p_j, lcm_{v_j \in parents(v_i)} ca_j) \tag{8}$$

$$cp_i = \begin{cases} \prod_{v_j \in parents(v_i)} p_j, & inDegree(v_i) > 0 \\ 1, & in-degree(v_i) = 0 \end{cases} \tag{9}$$

According to Theorem 2 and Corollary 1, with the help of PLSD-General, all operations mentioned in Section 2 can be implemented. The following algorithm describes a three stages generation process for PLSD-General label. First (line 1 to line 4), topologically sort the vertices in the directed graph where the orders of vertices within a strong connected component are ignored. We also assign the self-labels at this stage. Second (line 5 to line 12), compute the value of $comp(v_i)$. Finally (line 13 to line 17), compute the values of ancestors-label and parents-label.

7. Conclusion and Future Work

Prime number labeling scheme for DAG takes full advantage of the mapping between integers divisibility and vertices reachability. Operations on DAG, such as querying ancestors, descendants, siblings, leaves and nca could be easily converted to elementary arithmetic operations. The space requirement and time consuming are further reduced with the optimization techniques. Analysis also indicates that re-labeling only happens when a non-leaf vertex is inserted or removed, and affects its descendants (and ancestors if descendants-label used). By taking a strong connected component in the graph as one vertex, arbitrary directed graphs (with cycles) can be treated with the proposed labeling scheme.

Our implementation of prime number labeling scheme for DAG has least space requirement, construction time, and typical operations response time compared to interval-based and prefix-based labeling scheme in almost all of the experiments in our test bed.

```
   input  : A directed graph G = (V, E)
   output: The graph G with PLSD labels on each vertex
 1 TopoOrder [] ← TOPOLOGICAL-SORT (G);
 2 for i ← 0 to (|V| − 1) do
 3 |   Vertices [TopoOrder [i]].self-label ← NEXT-PRIME();
 4 end
 5 Components [] ← STRONGLY-CONNECTED-COMPONENTS(G);
 6 for i ← 0 to SIZE-OF(Components ) − 1 do
 7 |   ComponentsLabel [i] ← 1 ;
 8 |   for j ← 0 to SIZE-OF(Components [i]) − 1 do
 9 |   |   Components [i][j].component-id ← i ;
10 |   |   ComponentsLabel [i] ← Components [i][j].self-label ×
   |   |   ComponentsLabel [i] ;
11 |   end
12 end
13 for i ← 0 to (|V| − 1) do
14 |   v ← Vertices [TopoOrder [i]];
15 |   v.ancestors-label ← LCM(ComponentsLabel [v.component-id ],
   |   LCM_{v_j∈Parents(v)} v_j.ancestors-label);
16 |   v.parents-label ← ∏_{v_j∈Parents(v)} v_j.self-label ;
17 end
```

Algorithm 1: BuildPLSD

The main reason is that no additional information is required to be stored for non-spanning tree edges and that the utilizations of elementary arithmetic operations avoid time-consuming database operations. The extensive experiments also show good scalability and effect of update of prime number labeling scheme for DAG. The shape of DAG and the selectivity of operation results has little effect on the response time of our labeling scheme.
Doubtless, a polynomial time quantum algorithm for factoring integers is expectative. Factoring in parallel may be another more practical technology nowadays for prime number labeling scheme for DAG.

8. References

[1] Postgresql 8.0.3 documentation, 2005. Available at http://www.postgresql.0rg/docs/8.O/ interactive/ index.html.

[2] D. Beckett. Scalability and storage: Survey of free software/ open source rdf storagesystems. Technical Report 1016, ILRT, June 2003. http://www.w3.org/2001/sw/Europe/ reports/ rdLscalable_storage_report/.

[3] O.C.L. Center. Dewey decimal classification, 2009. Available at http://www.oclc.org/ dewey

[4] V. Christophides, G. Karvounarakis, D. Plexousakis, M. Scholl, and S. Tourtounis.Optimizing taxonomic semantic web queries using labeling schemes. *Journal of Web Semantics,* 11(001):207-228, November 2003.

[5] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.,* 32(5):1338-1355, 2003. T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press,September 2001.

[6] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics,* 3(2-3):158-182, 2005.

[7] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In P. M. G.Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *Proceedings of the 27th International Conference on Very Large Data Bases,* pages 361-370, Roma, Italy, 2001.

[8] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *Comput. J.,* 28(1):5-8, 1985.

[9] N. Wirth. Type extensions. *ACM Trans. Program. Lang. Syst.,* 10(2):204-214, 1988.

[10] X. Wu, M.-L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *ICDE,* pages 66-78. IEEE Computer Society, 2004.

Semantic Web
Edited by Gang Wu

ISBN 978-953-7619-54-1
Hard cover, 310 pages
Publisher InTech
Published online 01, January, 2010
Published in print edition January, 2010

Having lived with the World Wide Web for twenty years, surfing the Web becomes a way of our life that cannot be separated. From latest news, photo sharing, social activities, to research collaborations and even commercial activities and government affairs, almost all kinds of information are available and processible via the Web. While people are appreciating the great invention, the father of the Web, Sir Tim Berners-Lee, has started the plan for the next generation of the Web, the Semantic Web. Unlike the Web that was originally designed for reading, the Semantic Web aims at a more intelligent Web severing machines as well as people. The idea behind it is simple: machines can automatically process or “understand” the information, if explicit meanings are given to it. In this way, it facilitates sharing and reuse of data across applications, enterprises, and communities. According to the organisation of the book, the intended readers may come from two groups, i.e. those whose interests include Semantic Web and want to catch on the state-of-the-art research progress in this field; and those who urgently need or just intend to seek help from the Semantic Web. In this sense, readers are not limited to the computer science. Everyone is welcome to find their possible intersection of the Semantic Web.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Gang Wu and Juanzi Li (2010). Prime Number Labeling Scheme for Transitive Closure Computation, Semantic Web, Gang Wu (Ed.), ISBN: 978-953-7619-54-1, InTech, Available from: http://www.intechopen.com/books/semantic-web/prime-number-labeling-scheme-for-transitive-closure-computation

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen