Adapting Prime Number Labeling Scheme for Directed Acyclic Graphs

Gang Wu, Kuo Zhang, Can Liu, and Juanzi Li

Knowledge Engineering Lab, Department of Computer Science,
Tsinghua University, Beijing 100084, P.R. China

Abstract. Directed Acyclic Graph(DAG) could be used for modeling subsumption hierarchies. Several labeling schemes have been proposed or tailored for indexing DAG in order to efficiently explore relationships in such hierarchy. However few of them can satisfy all the requirements in response time, space, and effect of updates simultaneously. In this paper, the prime number labeling scheme is extended for DAG. The scheme invests intrinsic mapping between integer divisibility and subsumption hierarchy, which simplifies the transitive closure computations and diminishes storage redundancy, as well as inherits the dynamic labeling ability from original scheme. Performance is further improved by introducing some optimization techniques. Our extensive experimental results show that prime number labeling scheme for DAG outperforms interval-based and prefix-based labeling schemes in most cases.

1 Introduction

Directed Acyclic Graph(DAG) is an effective data structure for representing subsumption hierarchies in applications, e.g. OO programming, software engineering, and knowledge representation. The growing number and volume of DAGs in such systems inspire the demands for appropriate index structures for DAG.

Labeling schemes[8] are widely used in indexing tree or graph structured data considering their avoiding expensive join operations for transitive closure computations. Determinacy, compaction, dynamicity, and flexibility are factors for labeling scheme design besides speedup [10]. However, the state of art labeling schemes for DAG could not satisfy most above requirements simultaneously.

One major category of labeling schemes for DAG is spanning tree based. Ordinarily, they first find a spanning tree and assign labels for vertices following the tree's edges, and then propagate additional labels to record relationships of the non-tree edges. Christophides compared two such schemes [4], i.e. interval-based [7] and prefix-based [3]. Evaluations to the non-tree edges relationships 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 [9] and 2-hops [5]. Though bit vector can process operations on DAG more efficiently, it is static and requires global rebuilding of labels when

M.L. Lee, K.L. Tan, and V. Wuwongse (Eds.): DASFAA 2006, LNCS 3882, pp. 787–796, 2006.

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 [10]. Prime number labeling scheme associates each vertex with a unique prime number, and labels each vertex by multiplying parents' labels and the prime number owned by the vertex. Compared with prefix-based scheme, the effect of updating is almost the same, and the query response time and the size requirements are even smaller. However, no further work has been performed on extending the idea to the case of DAG.

We extend the scheme by labeling each vertex with an integer which equals to the arithmetic product of the prime numbers associating with the vertex and all its ancestors. Independent of spanning tree, the scheme can efficiently explore the subsumption hierarchies in a DAG by checking the divisibility among the labels. It also inherits dynamic update ability and compact size feature from its predecessor. Experimental results indicate that prime number labeling scheme is an efficient and scalable scheme for indexing DAG with appropriate extensions and optimizations. The major contributions are summarized as follows.

- Extend original prime number scheme[10] for labeling DAG and supporting the processing of typical operations on DAG.
- Optimize the scheme on space and time requirements in terms of the characteristics of DAG and prime numbers.
- A generator is implemented to generate arbitrary complex synthetic DAG for the extensive experiments. Space requirement, construction time, scalability, and the impact of selectivity and update are all studied in the experiments.

2 Prime Number Labeling Scheme for DAG

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 queries on those known typical operations related to *reachability*[1]. (See [4] for formal expressions). Vertex update is another kind of operation worthy of note because it may bring reorganizations to the index structure.

2.1 Prime Number Labeling Scheme for DAG - Lite

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

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

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.

[1] DAG(*directed acyclic graph*), and *reachability* are general definitions in graph theory. Given two vertices v and w, $v \rightsquigarrow w$ is used to indicate that w is reachable from v.

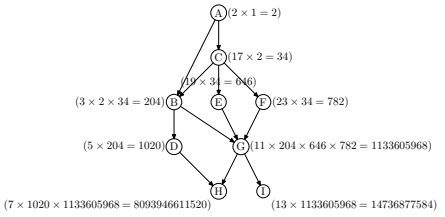

Fig. 1. PLSD-Lite

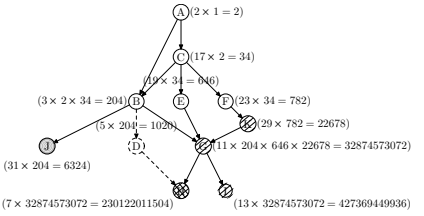

Fig. 2. Updates in PLSD-Lite

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

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

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

Lemma 1 [2] implies that for any vertex 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 DAG. For any two vertices $v, w \in V$ where $L_{lite}(v) = (c[v])$ and $L_{lite}(w) = (c[w])$, $v \rightsquigarrow w \iff c[v] | c[w]$.*

Consequently, 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 $2|1020$. 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. 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 simple solution to nca evaluating. First put all the common ancestors of both vertices into a set. Then filter out vertices whose descendants are also within the set.

As stated in [10], re-labeling happens with the insertion or deletion of a vertex, and only affects the descendants. After deleting vertex D, inserting leaf vertex J and non-leaf vertex K, we have Figure 2. As a new leaf, J does not affect other vertices in the DAG. Insertion of K only affects descendants G, H, I and K itself. Vertex H is affected by the deletion of ancestor D too.

However, PLSD-Lite lacks enough information to identify parents/child relation. In order to support this operation, we need to separately record the prime number that identifies the vertex and the additional information about parents.

2.2 Prime Number Labeling Scheme for DAG - Full

Definition 2. *Let $G = (V, E)$ be a DAG. A **Prime Number Labeling Scheme for DAG - Full** (PLSD-Full) associates each vertex $v \in V$ 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*

[2] All the proofs in this paper are omitted for the length limited.

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

and

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

We term $p[v]$ as "self-label", $c_a[v]$ ($c[v]$ in Definition 1) as "ancestors-label", 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.

Theorem 2. *Let $G = (V, E)$ be a DAG, and vertex $v \in V$ 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 $w \in V$ that takes p_i as the self-label for $1 \leq i \leq r$, 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 $u \in V$ that takes p'_i as the self-label for $1 \leq i \leq s$, and u is one of the parents of v.*

Therefore, 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 $1311 = 3 \times 19 \times 23$, 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. Unique factorization also can be used to obtain ancestors. Three ancestors A, B and C of vertex D could be identified by factoring "1020". Though trial division itself could be used to do integer factorization, we can choose faster integer factorization algorithm alternately especially for small integers. Corollary 1 further expresses PLSD-Full's sibling evaluation ability.

Corollary 1. *Let $G = (V, E)$ be a DAG. For any two vertices $v, w \in V$ 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 if and only if the greatest common divisor* $\gcd(c_p[v], c_p[w]) \neq 1$.

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 has two siblings E and F because $\gcd(34, 17) = 17 \neq 1$.

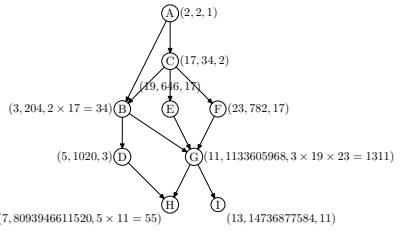

Fig. 3. PLSD-Full

3 Optimization Techniques

Elementary arithmetic operations employed by PLSD become time-consuming when their inputs are large numbers. Some optimizations are introduced here.

Least Common Multiple. There is apparent redundancy in previous 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.

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

Theorems 1 and 2 still hold. Define $lcm(a_1, a_2, ..., a_n)$ to be the *least common multiple* of n integers $a_1, a_2, ..., a_n$. In particular, we define $lcm(a) = a$ here.

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

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, Equation 5 holds. 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. Figure 4 has a smaller max-length of ancestors-label.

Topological Sort. Previous selection of prime number for the self-label is arbitrary as long as any two vertices have different self-label. A naive approach is assigning each vertex met in depth-first search of DAG a prime number ascendingly. 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[6] of a DAG provides the character we need. 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.

Leaves Marking. As an optimization for reducing label size, even numbers e.g. $2^1, 2^2, ..., 2^n$ are used as self-labels for leaf vertices in [10], which gives us another

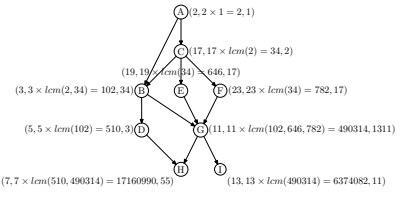

Fig. 4. Least Common Multiple

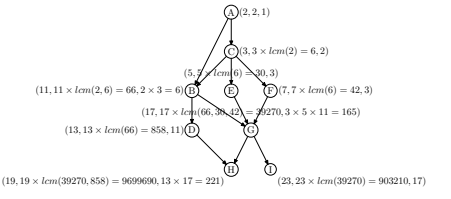

Fig. 5. Topological Sort

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 set leaf's ancestors-label to be negative. 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.

Descendants-Label. In the same idea of ancestors-label, we can extend LDUP-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 a reverse version of Equation 3. Now, $descendants(v)$, can be evaluated by factoring descendants-label. In section 4 we will give empirical results on querying descendants and leaves using this technique.

4 Performance Study

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

4.1 Experiment Settings

Taking the queries on RDF class hierarchies as an application background for DAG, we setup test bed on RSSDB v2.0 [2]. In this case, each vertex stands for a class in the RDF metadata, and each edge stands for the hierarchy relationship between a pair of classes in the RDF metadata. RDF metadata is parsed and stored in PostgreSQL (win32 platform v8.0.2 with Unicode configuration).

Though least common multiple, topological sort, and leaves marking are optional optimizations, they are integrated in our default PLSD-Full implementation. PLSD-Lite and PLSD-Full without these optimizations are ignored for their apparent defects. Furthermore, 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. 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 (UPrefix). All the implementations are developed with JDK1.5.0. Database is connected through PostgreSQL 7.3.3 JDBC2 driver. All the experiments are conducted on a PC with single 2.66GHz Intel Pentium 4 CPU, 1GB DDR-SDRAM, 80GB IDE hard disk, and Microsoft Windows 2003 Server.

The relational representations of UPrefix and PInterval, including tables, indexes, and buffer settings, are the same to [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 at least 32589158477190044730 which exceeds the bound of *bigint* (8 bytes, between ±9223372036854775808). 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 in main memory. Similarly, we use PLSDF-D(self-label: text, label: text, parent-label: text, descendants-label: text, uri: text) to represent PLSDF-D. For PLSDF and PLSDF-D, we only build B-tree indexes on self-labels. Buffer settings are the same to those of UPrefix and PInterval.

4.2 Data Sets

To simulate diverse cases of DAG, we implement a RDF metadata generator to generate RDF file with arbitrary complexity and scale of RDF class hierarchies. Generator's input includes the count of vertices, the max depth of spanning tree, the max fan-out of vertices, and the portion of fan-in (*ancestors/precedings*). The output is a valid RDF file. We concatenate the values of above four parameters and the count of edges with hyphens to identify a DAG. Listed in Table 1, two groups of DAGs are generated for evaluating the performance.

Table 1. Data Sets

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
90000-16-2-0.000053-44946	16.3	90000	44946	16	2	0.000053	3

4.3 Space Requirement and Construction Time

As shown in Figure 6(a), PLSDF and PLSDF-D have much smaller average space requirement and mild trend of increase. The underlying cause is twofold. First, both are simply composed of only one table whose row size equals to the count of vertices, and one B-tree index. In contrast, Interval and UPrefix consist of three tables (and more indexes) to record additional information besides spanning tree. Another cause is that all the data type in the table of PLSDF or PLSDF-D are *text* which will be "compressed by the system automatically, so the physical requirement on disk may be less"[1]. 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. 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.

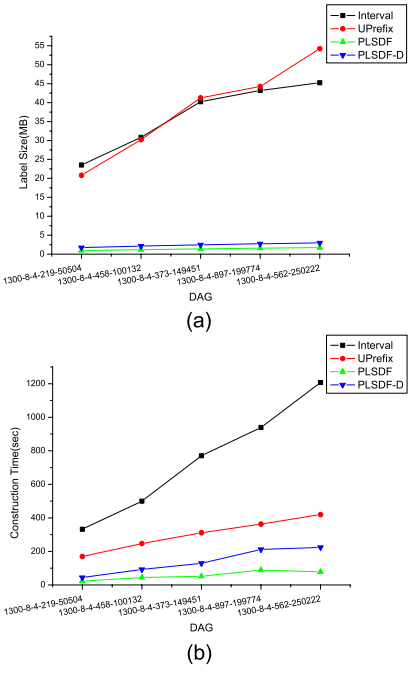

Fig. 6. Label Size(a) & Construction Time(b)

	Operation Type	Selectivity
Q1	ancestors	2.53%
Q2	descendants	20.08%
Q3	siblings	2.98%
Q4	leaves	38.67%
Q5	nca	0.011%

Fig. 7. Test Typical Operations for Overall Performance

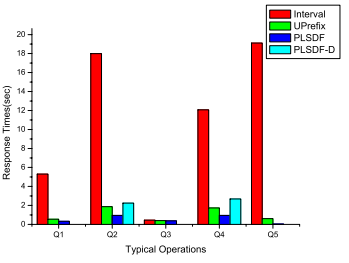

Fig. 8. Overall Performance

4.4 Response Time of Typical Operations

Overall Performance. DAG "9000-8-4-0.004-45182" is used here. The operations are listed in Figure 7. The total elapsed time is shown in Figure 8. PLSDF-D is applied only to Q2 and Q4 to examine the effectiveness of descendants-label, because it is the same to PLSDF for the other three queries. 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 concise table structure of PLSDF/PLSDF-D and computative elementary arithmetic operations 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. Then do factorization using the labels. The only database access happens in the first step.

Impact of Varying Selectivity. Selectivity experiment result is shown in Figure 9. Diagrams in the 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 average length from the spanning-tree root. The metric of Y-axis is the response time. PLSDF displays almost constant time performance for all operations. Though the change of response time is indistinguishable in some extensions, PLSDF stays at a disadvantage at a very low selectivity especially for Q2 and Q4. Fortunately, PLSDF-D

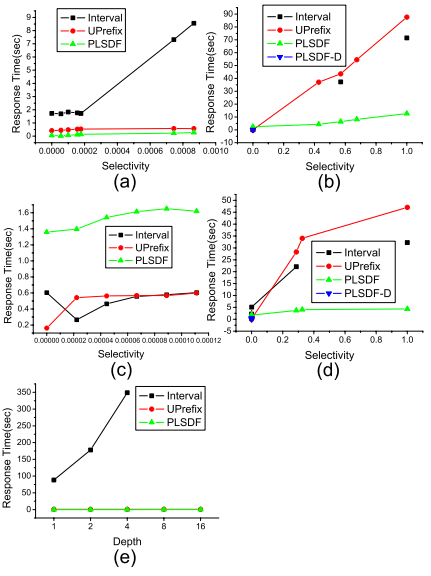

Fig. 9. Impact of varying selectivity

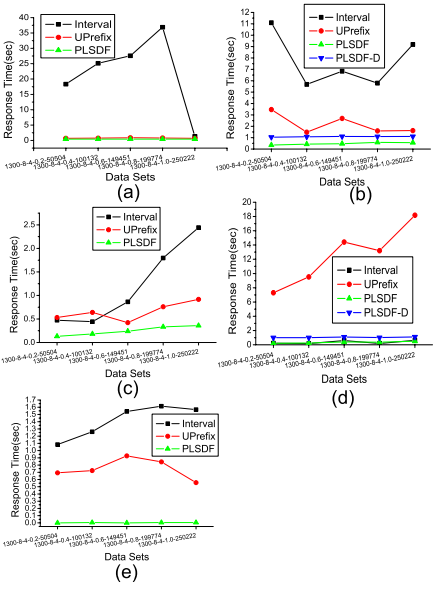

Fig. 10. Scale-up Performance

counterbalances this with descendants-label. 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. PLSDF has to traverse among the vertices and compute greatest common divisor one at a time.

Scale-Up Performance. We carry out scalability tests 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 10 corresponds to operations from Q1 to Q5 respectively. Interval and UPrefix are affected by both the size of the DAG. Unlike the other two labeling schemes, PLSDF and PLSDF-D perform good scalability in all cases.

4.5 Effect of Updates

The "Un-ordered Updates" experiments exhibited in [10] are repeated. Here we only give the results of updates on non-leaf vertices (updates on leaf have the same results to that of XML tree, see Section 2.1). 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 11 shows our experimental results which coincides 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 to be re-labeled.

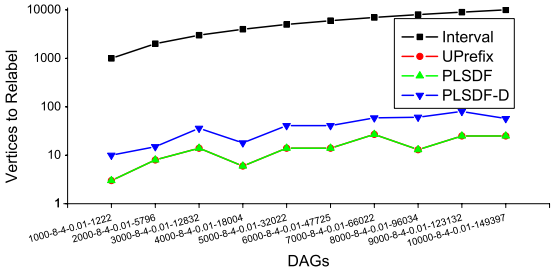

Fig. 11. Effect of Updates

5 Conclusion

Prime number labeling scheme for DAG takes advantage of the mapping between integers divisibility and vertices reachability. No extra information is required to be stored for non-spanning tree edges, and the utilizations of elementary arithmetic operations avoid time-consuming database operations. Performance is further improved with the optimization techniques. Analysis also indicates that re-labeling only happens when a non-leaf vertex is inserted or removed.

References

[1] Postgresql 8.0.3 documentation. available at http://www.postgresql.org/docs/8.0/interactive/index.html.
[2] D. Beckett. Scalability and storage: Survey of free software / open source rdf storage systems. Technical Report 1016, ILRT, June 2003. http://www.w3.org/2001/sw/Europe/reports/rdf_scalable_storage_report/.
[3] O. C. L. Center. Dewey decimal classification. 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.
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
[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.