Optimizing Taxonomic Semantic Web Queries using Labeling Schemes[*†]

V. Christophides G. Karvounarakis D. Plexousakis
Institute of Computer Science, FORTH,
Vassilika Vouton, P.O.Box 1385, GR 711 10, Heraklion, Greece
{christop, gregkar, dp}@ics.forth.gr
Michel Scholl
CEDRIC/CNAM
292 Rue St Martin, 75141 Paris, Cedex 03, France
scholl@cnam.fr
Sotirios Tourtounis
Department of Computer Science,
Univ. of Crete, GR 71409, Heraklion, Greece
tourtoun@csd.uch.gr

Abstract

This paper focuses on the optimization of the navigation through voluminous *subsumption* hierarchies of topics employed by Portal Catalogs like Netscape Open Directory (ODP). We advocate for the use of labeling schemes for modeling these hierarchies in order to efficiently answer queries such as subsumption check, descendants, ancestors or nearest common ancestor, which usually require costly transitive closure computations. We first give a qualitative comparison of three main families of schemes, namely bit vector, prefix and interval based schemes. We then show that two labeling schemes are good candidates for an efficient implementation of label querying using standard relational DBMS, namely the Dewey Prefix scheme and an Interval scheme by Agrawal, Borgida and Jagadish. We compare their storage and query evaluation performance for the 16 ODP hierarchies using the PostgreSQL engine.

1 Introduction

Semantic Web applications such as e-commerce, e-learning, or e-science portals and sites require advanced tools for managing metadata i.e., descriptions about the meaning, usage, accessibility or quality of information resources (e.g., data, documents, services) found on corporate intranets or the Internet. To describe resources, various structured vocabularies (i.e., thesauri) or thematic taxonomies (i.e., conceptual schemas) are widely employed by different user communities. Such descriptive schemas represent nowadays an important part of the hierarchical data available on the Web [18]. In this context, the Resource Description Framework (RDF) [4, 16] is increasingly gaining acceptance for metadata creation and exchange by providing i) a *Standard Representation Language* for descriptions based on *directed labeled graphs*; ii) a *Schema Definition Language* (RDFS) [4] for modeling user thesauri or taxonomies as class/property subsumption hierarchies (i.e., trees or DAGs); and iii) an *XML syntax* for both schemas and resource descriptions. For instance, Web Portals such as Netscape Dmoz or Chefmoz, MusicBrain, CNET, XMLTree[1] export their catalogs in RDF/S. In this paper, we are interested in labeling schemes for such hierarchical data exported by Portals, in order to optimize complex queries on their catalogs.

A Portal catalog - created according to one or more topic hierarchies (*schemas*) - is actually published on the Web as a set of statically interlinked Html pages[2]: each page contains the

[*]This work was supported in part by the European project Mesmuses (IST-2000-26074).
[†]An earlier version of this paper appears in the proceedings of WWW2003.
[1]See dmoz.org, chefmoz.org, musicbrain.org, home.cnet.com, www.xmltree.com, respectively.
[2]Note that RDF is used as an export format for bulk catalog loading.

information resources (*objects*) classified under a specific topic (*class*), as well as various kinds of relationships between topics. In particular, the *subtopic* relationship represents *subsumption* (*isA*) between classes. Then, a Portal schema forms a tree (single *isA* links) or a DAG (multiple *isA* links) of classes (at best *semi-lattices*), and assists end-user navigation: for each topic one can navigate to its subtopics (i.e., subclasses) and eventually discover the resources which are directly classified under them. In [3, 14, 20] we have studied how declarative query languages for RDF/S can support dynamic browsing interfaces and personalization of both Portal schemas and resource descriptions. This paper is a first step toward the general optimization of query languages for RDF/S. We focus on the optimization of a large class of queries, central to semantic web applications, namely the queries on class hierarchies. Basically, we optimize such queries by avoiding costly transitive closure computations over voluminous class hierarchies[3]. More precisely, we are interested in labeling schemes for RDF/S class (or property) hierarchies allowing us to efficiently evaluate descendant/ancestor, adjacent/sibling queries, as well as, finding nearest common ancestors (nca) by using only the generated labels. Compared to the transitive closure evaluation reported in our previous work [14], the performance gains for these queries are of 3-4 orders of magnitude when using adequate labeling schemes! Then, starting from a topic somewhere in the taxonomy, a user can easily and efficiently access not only its parent/children (as in existing Portals) but also the leaf topics underneath where most of the web resources are classified, discover sibling topics (where related web resources may be found) or even continue navigation from the nca of two topics in the hierarchy. It is worth noting that we focus in this paper on *intentional queries* (i.e., *schema* queries) since they represent a novel requirement for Semantic Web applications. However, our optimization techniques can be easily applied to *extensional queries* (i.e., *data* queries) involving complex data paths as in the case of RDF resource descriptions or XML documents.

Several labeling schemes for tree or graph-shaped data have been proposed for network routing [12], object programming [10, 11, 24, 2, 5, 15], knowledge representation systems [1] and recently XML search engines [9, 26, 17, 13, 8, 7, 21]. However, choosing a labeling scheme for efficiently supporting the functionality required by Web Portals is still an open issue because:

- Portal's *isA* hierarchies of classes, may range from simple trees to complex DAGs [18] while the ordering of subclasses is not important (compared to XML search engines); therefore we need a *labeling scheme for trees that can be easily extended for DAGs with a reasonable extra storage and querying cost.*
- Querying/Browsing Portal schemas heavily relies on bulk class retrieval using complex filtering conditions on subsumption relationships (unlike network routing, object programming or knowledge representation systems treating two nodes/classes at a time); thus we need a *labeling scheme generating class labels which can be efficiently processed by a database back-end* using standard index structures (i.e., B-trees).

We are interested in the tradeoff between storage and query requirements of different labeling schemes for both trees and DAGs of RDF/S classes (or properties). Our contribution, guided by the efficient implementation of label querying using standard DBMS technology, is three-fold :

- Section 2 briefly recalls the RDF/S modeling primitives used to represent the ODP Catalog and presents statistics about the size and the morphology of the ODP class semi-lattices that are used for our performance evaluation of existing labeling schemes.
- Section 3 provides a qualitative analysis of *bit-vector*, *prefix*, and *interval* based labeling schemes for tree or graph-based data exported by Portals like ODP. We pay particular attention to the expression of the core query functionality (i.e., descendant/ancestor/leaf, adjacent/sibling, nca) with each labeling scheme.
- Section 4 compares the performance of two representative labeling schemes, namely the Unicode Dewey *prefix* scheme [6] and the extended postorder *interval* scheme by Agrawal, Borgida and Jagadish [1], in terms of storage requirements and query execution time on top of an ORDBMS (PostgreSQL). We focus on the efficient translation of the different types of queries over class trees (single *isA*) into SQL, as well as, the extra cost required for DAGs (multiple *isA*).

[3]For example, the catalog of the Open Directory Portal/Dmoz comprises around 200M of Topics exported in RDF files.

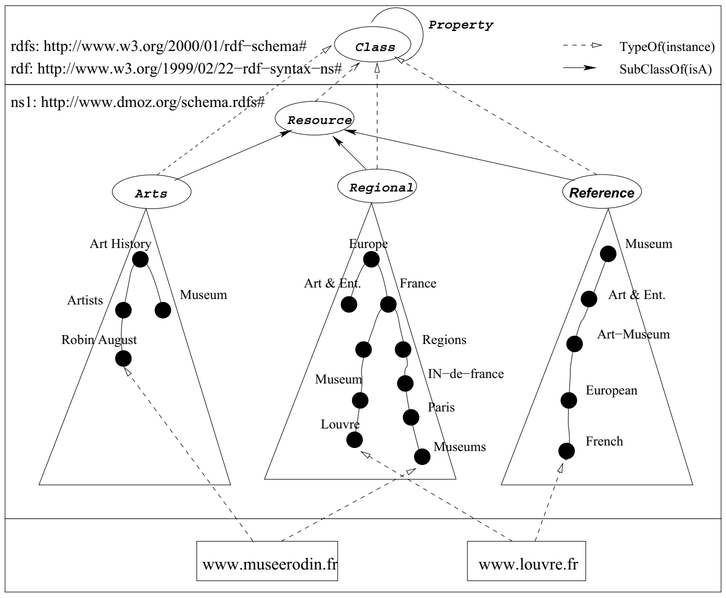


Figure 1: RDF Catalog of Open Directory Portal

2 Motivating Example: ODP

Portals aggregate and classify various information resources for diverse target audiences (e.g., enterprise, professional, trading). A portal catalog includes descriptive information about resources found on corporate intranets or the Internet. The complexity of the semantic descriptions, using thesauri, taxonomies or more sophisticated ontologies depends on the scope of the community domain knowledge as well as the nature of the available resources (sites, documents, etc.).

In most Web Portals, resources are classified under large hierarchies of topics that can be represented and exchanged using RDF/S. Figure 1 depicts a part of the RDFS schema employed by Netscape Open Directory (or Dmoz) Portal (ODP) identified by the namespace `ns1`[4]: nodes denote class names/topics (e.g., *Museum*) and solid edges denote subsumption relationships between them (e.g., *ArtMuseum* $\prec$ *Museum*). Note that the roots of all topic hierarchies (e.g., *Arts*, *Regional*, *Reference*) specialize the core RDF/S class *Resource*. These hierarchies are *class semi-lattices* and in the simplest case take the form of trees[5]. From an application viewpoint, they play the role of facets, which can be combined in order to describe and retrieve Web resources.

Using faceted classification, a resource is described (classified) using one or more topics from each facet. For example, in Figure 1 the Web site of Rodin museum in Paris is classified under both '*Reference/Museum/Art&Entertainment/Art-Museum/European/French*' and '*Regional/Europe/-France/Regions/Ile-de-France/Paris/Museums*' where the dashed edges stand for RDF/S instantiation relationships. We can observe that topic names are composed of different descriptive terms (e.g., *Museum*, *France*). The ODP schema designers partially replicate these terms in the various topic hierarchies in order to denote all the valid combinations of terms (from different facets). In our example, cultural and geographical terms (e.g., *Museum* and *France*) appear in both *Reference* and *Regional* hierarchies, while the complete path from the root of these hierarchies is used as a prefix to distinguish topic names. For simplicity, we hereforth omit the schema namespaces as well as the prefix paths.

Table 1 lists the complete statistics of 16 ODP hierarchies (version of 01/16/2001) comprising 253214 topics under which 1688037 Web resources are classified (fan-in stands for the fan-in degree of the tree, i.e. the number of direct subclasses of a given class). Note that the total number of distinct terms used by all topics is 80795 while 14355 of them (17,77%) are replicated in more than one topic name. Under these topics, a total number of 1715225 resources are classified with 118925 (6,93%) of them multiply classified under more than one topic. Moreover, due to the partial

[4]http://www.dmoz.org/schema.rdfs. For simplicity, we omit administrative metadata such as titles, mime-types, modification-dates, of Web resources represented in ODP by an OCLC Dublin-Core like schema [23].

[5]It is worth noticing that the effect of multiple *isA* is partially captured by terms replication in several hierarchies as well as other link types defined between topics such as *symbolic* and *related*.

Hierarchy	Max. Depth	Avg Depth	Max Fan-in at Depth	Avg Fan-in	#Topics	#Terms	#Resources
netscape.rdf	7	5.75	24/1	0.9948	389	203	27188
news.rdf	7	5.05	51/4	1.0027	721	411	47735
kat.rdf	7	4.84	46/4	1.0026	761	646	7730
home.rdf	8	5.43	53/4	1.0011	1722	1353	26688
health.rdf	9	6.32	52/8,5	1.0006	3202	1728	45519
shopping.rdf	9	5.67	61/2	1.0005	3349	2357	88821
games.rdf	10	6.74	125/3	1.0004	4857	3710	36181
computers.rdf	10	6.4	147/3	1.0003	6010	3259	91597
reference.rdf	13	8.73	154/3	1.0003	6483	3759	75105
business.rdf	11	6.44	52/4,5	1.0002	6833	3630	161877
recreation.rdf	11	6.8	85/3	1.0002	7269	3243	93929
science.rdf	10	8.35	314/6	1.0002	8667	6812	65939
sports.rdf	9	7.14	178/6	1.0001	10625	5927	66280
society.rdf	12	7.9	157/7	1.0001	16250	8678	161433
arts.rdf	11	7.04	267/4	1.0000	25314	16840	214795
regional.rdf	13	8.27	254/7	0.9999	150762	32594	587152
Total	13	7.83	314	0.9999	253215	80795	1715225

Table 1: Statistics of the ODP Topic Hierarchies

replication of terms, ODP topic hierarchies are relatively deep (the average depth is 7.83 and the maximum is 13) with a varying fan-in at each level (the maximum fan-in degree is 314 while the average is only 0.9999). Table 1 also illustrates the depth of classes with the maximum fan-in degree for each hierarchy. ODP subclass trees are far from complete and the largest percentage of the classes appears in the upper half of the respective trees. In addition, the maximum fan-in degree is in the middle and slightly in the upper half of the corresponding of ODP trees.

With current Portal interfaces users can either navigate through the topic hierarchies in order to locate resources of interest, or issue a full-text query on topic names and the URIs of the described resources or the text values of attributes like title, description. In the first case, users have to navigate from the root of each hierarchy down to the leaves in order to reach the resources of interest, because most of the resources are classified under the leaf topics. In the second case, users are forced to manually filter the topics and URIs returned by the full-text query. Advanced browsing/querying interfaces aim at simplifying such tasks, by permitting smooth navigation/filtering on both Portal schemas and resource descriptions. In order to support such Portal interfaces we need an efficient evaluation of a number of basic queries on class (or property) semi-lattices: (a) find direct subclasses, transitive ancestor/descendant subclasses or leaf classes; (b) find sibling (brother) or following/preceding (adjacent[6]) classes; and (c) find the nearest common ancestor(s) (nca) of two classes. Examples of these queries in a simplified schema are illustrated in Figure 2.

3 Families of Labeling Schemes

The labeling schemes proposed in the literature can be characterized by:

- The structure of the encoded data (trees, graphs, etc.);
- The supported queries (ancestor/descendant/leaf, adjacent/sibling, nca);
- The complexity of the labeling algorithms;
- The maximum or average label size;
- The query evaluation time on the resulting labels;
- The relabeling implications of incremental updates.

In this section, we present a qualitative comparison of three families of labeling schemes, namely bit-vector, prefix and interval.

[6]Note that adjacent queries do not explore semantic relationships of classes but they have been included in our study for completeness reasons w.r.t. XML XPath expressions.

3.1 Bit-Vector Schemes

The label of a node is represented by a vector of n bits where n is the number of nodes, a "1" bit at some position uniquely identifies the node in a lattice L and each node inherits the bits identifying its ancestors (or descendants) in a top-down (or bottom-up) encoding. More formally, in the algorithm proposed by Wirth [24] (see Figure 2-a), the label of a node u in L is $l(u) = \{b_1, \ldots, b_n\}$, $b_i = 1$ if the ith node is either u or an ancestor (alternatively descendant) v of u. Otherwise $b_i = 0$. Then, using binary OR (|) and AND (&) on labels, one can check whether a node v is an ancestor (descendant) of u in L: $u \prec v$ iff $l(u)\ \&\ l(v) = l(v)$ (or $l(u) \mid l(v) = l(u)$). This scheme supports subsumption checking and Least Upper Bound (LUB) or Greatest Lower Bound (GLB) operations (i.e., nca/ncd) in constant time (the time for comparing two bit vectors) while labels can be constructed in time linear in the size of L. It should be stressed that all labels have fixed size n bits and the storage required for the labels of a lattice L is exactly n^2.

More compact variations of bit-vector schemes [2, 5, 15] use new bit positions only when it is necessary to distinguish between nodes with common descendants. For instance, the total size of the bit-vectors produced by the Near Hierarchical Encoding (NHE) [15] is $2 * n * logn$ for balanced binary trees and close to $logn$ when multiple isA is low. However, the most interesting compact variations do not support all the queries we need: Caseau's scheme [5] supports only ancestor/descendant checking, while NHE [15] supports only lattice operations (LUB/GLB). In addition, NHE is able to encode arbitrary partially-ordered sets rather than lattices as in Caseau's algorithm. Ait-Kaci's scheme [2] supports all the previous operations but generates labels of size $O(logn)$ and $O(n)$ in the best and worst case respectively.

The main drawback of bit-vector labeling schemes is that ancestor/descendent/sibling queries are $O(n)$. No $O(logn)$ data structure can be used to accelerate the evaluation of these queries. Additionally, the (fixed) size of the produced labels heavily depends on the size (and the morphology for compressed variations) of the input class hierarchies making these schemes inappropriate for a database implementation especially in the presence of incremental updates.

3.2 Prefix Schemes

Prefix-based schemes directly encode the parent of a node in a tree, as a prefix of its label using for instance a depth-first tree traversal. Therefore the labels for a tree T can be computed in time linear in the number of nodes in T. The simplest algorithm is the Dewey Decimal Coding (DDC) widely used by librarians [6] (see Figure 2-b): the label of a node u in T is $l(v)l(u)$ where $l(v)$ is the label of its parent v, $l(u) \in \{0, .., 9\}$[7]. Then, one can check whether a node v is an ancestor of u in T in practically constant time by checking whether a string is a prefix of another one: $u \prec v$ iff $l(v) \in prefixes(l(u))$. The same is true for finding the nca of two tree nodes. An interesting property of prefix-based labels is their lexicographic order: the labels of nodes u in a subtree with root v are greater (smaller) than those of its left (right) sibling subtrees: $prev(l(v)) < l(v) < l(u) < next(l(v))$ where $next$('19') ='2' and $prev$('12') ='11'. Then, index structures based on the key's domain order such as the B-tree, can be used to speed-up the evaluation of our testbed queries (i.e., ancestor/descendant/leaf, preceding/following/ sibling and nca). Table 2 gives for each query expressed in a declarative way (column 1), its corresponding formulation in terms of the required conditions on the labels for different schemes. The set of conditions for the prefix-based scheme is given in column 2. Parent/children/sibling queries rely purely on string matching functions: the parent of a node in T is directly given by the greatest prefix (function $maxprefix$ returning all but the last character of the input string) of its label. Nca queries require to find common prefixes (function $prefixes$) of maximum length (function $maxlength$). Although label conditions involving user-defined functions can be translated in the recent versions of the SQL standard (SQL-99), in existing SQL engines such queries do not take benefit from indices defined on labels (i.e., they can be evaluated using only sequential scans).

In DDC, the size of the proper node label (e.g., '1', '2') at each level is exactly one byte and thus the maximum label size (in bytes) depends only on the maximum depth of T. As a matter of fact, DDC consumes per node more bits than actually required but this extra cost makes easier a string representation of labels by avoiding the introduction of separator characters like '.' at

[7]Note that the same idea is employed by ODP in order to identify topics/classes from the root of each hierarchy with user readable labels, using a vocabulary of distinct terms/words (see Table 1).

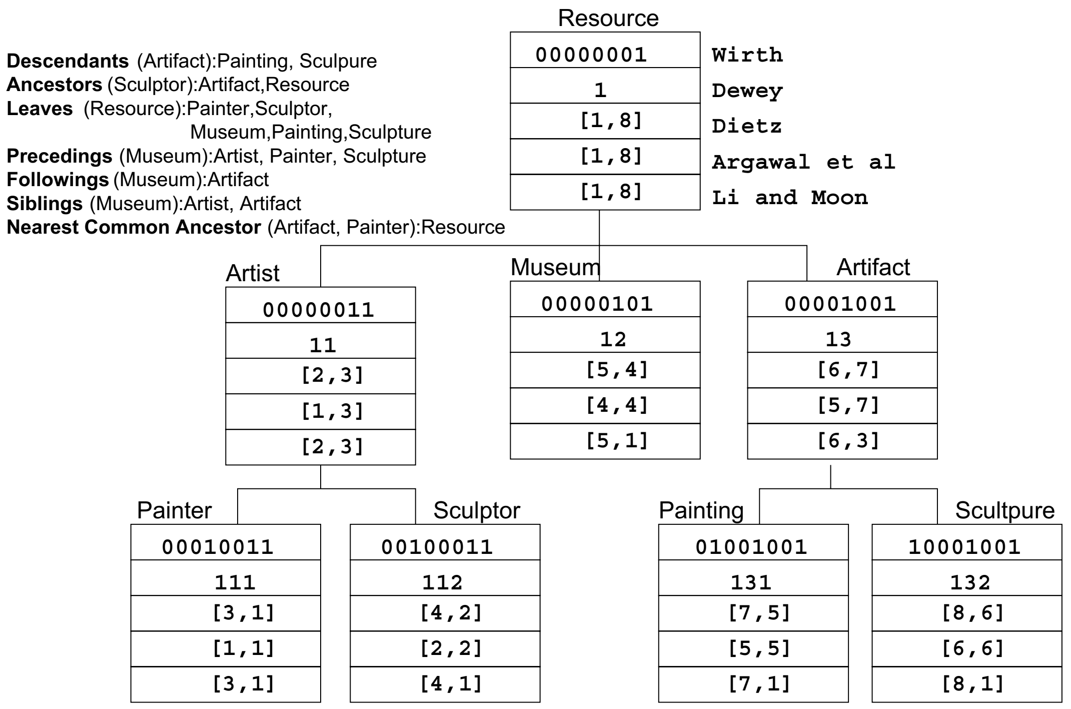


Figure 2: Labeling Schemes: a) Wirth b) Dewey c) Dietz d) Agrawal et al e) Li and Moon

each level (e.g., '`1.2`'). For fan-in degrees greater than 10, larger alphabets should be used to label each node as, for instance, the Unicode Character Set. In UTF-8 [25] a variable number of bytes are used to encode integer codes of different character sets: ASCII characters are encoded by one byte (from 0x00 to 0x7f) while characters in other sets (> 0x7f) are encoded as a multibyte sequence (consisting only of bytes in the range 0x80 to 0xfd) with the first byte indicating its length (up to 3 bytes long). Since in Portal schemas (see Table 1) the average fan-in degree is small (0.9999 compared to the maximum 314), most of the node labels require one byte per depth (i.e., can be encoded by ASCII characters). When binary alphabets are used, the maximum size of prefix-based labels (in bits) depends both on the maximum depth (d) and fan-in degree (Δ) of the encoded tree T ($dlog\Delta$). Applications of this scheme to XML tree data have been proposed in [9, 21]. Several variations provide more compact labels that minimize either the maximum size of a label (fixed size representation) or the average size of a label (variable size representation). See [13] for a comparative analysis and [12] for a recent survey.

The main advantage of prefix-based labeling schemes is their dynamicity in the presence of incremental updates. As long as ordering among descendants is not important (as in class semi-lattices), one can always add new children nodes to the right of existing nodes without having to relabel them. As a matter of fact, most of the benefits (for updates, compression) of prefix-based schemes are due to the production of labels with variable size. Unfortunately, the evaluation of queries on variable size labels relies on (bit) string manipulation functions (especially for compressed prefix variations), reducing the optimization opportunities of existing SQL query engines because the evaluation cost of user-defined functions is unknown by the optimizers. Finally, prefix-based schemes produce inflationary labels when extended for DAGs (to cater for multiple isA, see section 4.2).

3.3 Interval Schemes

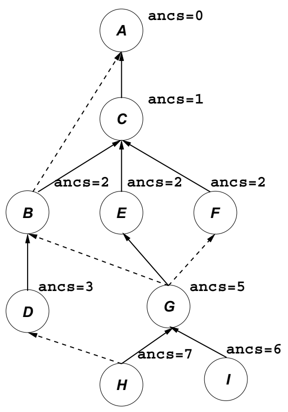
The label of a node in a tree T is given in this scheme by an interval $(start, end)$ such that it is contained in its parent's interval label. In the original scheme of Dietz [10, 11] (see Figure 2-c) each node is labeled with a pair of its preorder and postorder numbers in T: the label of a node u is $[pre(u), post(u)]$. Since an ancestor node v appears before (after) a descendant node u in the pre-(post)order traversal of T, $u \prec v$ iff $pre(v) < pre(u)$ and $post(v) > post(u)$. In addition, the intervals of two sibling nodes w and u are disjoint. The complete set of conditions for our testbed queries is given in column 5 of Table 2. Interval labels can be computed in time linear

Query	**Dewey**	**Agrawal et al**	**Li and Moon**	**Dietz/Zhang et al**
descendants(v) $\{u \mid u \prec v\}$	$l(u) < next(l(v))$ $\wedge\ l(u) > l(v)$	$index(v) <= post(u)$ $\wedge\ post(u) < post(v)$	$pre(v) < pre(u)$ $\wedge\ pre(u) + size(u)$ $<= pre(v) + size(v)$	$post(u) < post(v)$ $\wedge\ pre(v) < pre(u)$
ancestors(v) $\{u \mid u \succ v\}$	$prefixes(l(v))$	$index(u) <= post(v)$ $\wedge post(u) > post(v)$	$pre(v) > pre(u)$ $\wedge\ pre(u) + size(u)$ $>= pre(v) + size(v)$	$post(u) > post(v)$ $\wedge\ pre(v) > pre(u)$
leaves(v) $\{u \mid u \prec v$ $\wedge\ \neg\exists u' :$ $(u' \prec v$ $\wedge\ u' \prec u)\}$	$l(u) < next(l(v))$ $\wedge l(u) > l(v)$ $\wedge\neg\exists u' : (l(u') > l(v)$ $\wedge\ l(u') < next(l(v))$ $\wedge\ l(u') < next(l(u))$ $\wedge\ l(u') > l(u))$	$index(v) <= post(u)$ $\wedge post(u) < post(v)$ $\wedge\ post(u) = index(u)$	$pre(v) < pre(u)$ $\wedge\ size(u) = 1$ $\wedge\ pre(u) <$ $pre(v) + size(v)$	$post(u) < post(v)$ $\wedge\ pre(v) < pre(u)$ $\wedge\ pre(u) =$ $depth(u) + post(u)$
precedings(v) $\{u \mid u \ll v\}$	$l(u) <= prev(l(v))$ $\wedge\ l(u) >$ $maxprefix(l(v))$	$post(u) < index(v)$	$pre(u) + size(u)$ $<= pre(v)$	$post(u) < post(v)$ $\wedge\ pre(u) < pre(v)$
followings(v) $\{u \mid v \gg u\}$	$next(l(u)) >= l(v)$ $\wedge\ l(u) <$ $next(maxprefix(l(v)))$	$index(u) > post(v)$	$pre(u) >=$ $pre(v) + size(v)$	$post(u) > post(v)$ $\wedge\ pre(u) > pre(v)$
siblings(v) $\{u \mid u \leftrightarrow v\}$	$maxprefix(l(u)) =$ $maxprefix(l(v))$	$post(parent(u)) =$ $post(parent(v))$	$pre(parent(u)) =$ $pre(parent(v))$	$pre(parent(u)) =$ $pre(parent(v))$
nca(v,w) $\{u \mid u \succ v$ $\wedge\ u \succ w$ $\wedge\ \neg\exists u' :$ $(u' \succ v$ $\wedge\ u' \succ w$ $\wedge\ u' \prec u)\}$	$l(u) \in prefixes(l(v))$ $\cap\ prefixes(l(w))$ $\wedge\ maxlength(l(u))$	$index(u) <= \texttt{i}$ $\wedge post(u) > \texttt{p}$ $\wedge\ \neg\exists u' :$ $(index(u') <= \texttt{i}$ $\wedge\ post(u') > \texttt{p}$ $\wedge\ index(u')$ $<= post(u)$ $\wedge\ post(u') < post(u))$ `where` $\texttt{i} = minindex(v, w),$ $\texttt{p} = maxpost(v, w)$	$pre(u) < \texttt{pr}$ $\wedge\ pre(u) + size(u)$ $<= \texttt{ps}$ $\wedge\ \neg\exists u' :$ $(pre(u') < \texttt{pr}$ $\wedge\ pre(u') + size(u')$ $<= \texttt{ps}$ $\wedge\ pre(u') > pre(u)$ $\wedge\ pre(u') + size(u')$ $<= pre(u) + size(u))$ `where` $\texttt{pr} = minpre(v, w),$ $\texttt{ps} = max($ $pre(v) + size(v),$ $pre(w) + size(w))$	$post(u) > \texttt{p}$ $\wedge\ pre(u) < \texttt{pr}$ $\wedge\ \neg\exists u' :$ $(post(u') > \texttt{p}$ $\wedge\ pre(u') < \texttt{pr}$ $\wedge\ post(u') < post(u)$ $\wedge\ pre(u') > pre(u))$ `where` $\texttt{pr} = minpre(v, w),$ $\texttt{p} = maxpost(v, w)$

Table 2: Core Query Expressions for Trees: a) Dewey b) Agrawal et al c) Li and Moon d) Dietz/Zhang et al

in the size of T. Subsumption checking can be evaluated in constant time (i.e., comparing four integers) while the storage required for the labels of a tree T is $O(n)$ and the label size in bits is exactly $2logn$ [22]. The labeling scheme proposed in [26] for XML tree data is a straightforward extension of Dietz's scheme with *depth* information about tree nodes in order to also compute direct parent/children and leaf queries. However, for sibling queries as well as for an efficient evaluation of parent/children queries (avoiding the computation of all ancestors/descendents) we need to additionally encode the *parent* of each tree node and therefore *depth* becomes redundant.

One variation for graphs has been proposed by Agrawal, Borgida and Jagadish [1] (see Figure 2-d for trees and Figure 3-a for graphs) and relies on the introduction of a spanning tree to distinguish between tree and non-tree edges connecting class nodes. They propose a hybrid scheme in which the spanning tree edges fully take advantage of the interval-based labeling, while the non spanning tree edges require a replication of the label of their source node upwards to their target and its ancestors. Then, subsumption checking for spanning tree edges relies purely on interval inclusion test, while for the remaining edges one has to also check whether there is a path in the graph. More precisely, a node u in the spanning tree T of the graph is labeled with $[index(u), post(u)]$ where *post* is the postorder number of u and *index* is the lowest postorder number of u's descendants ($index(u) <= post(u)$ and for leaf nodes $index(u) = post(u)$). Furthermore, a node u can receive additional labels as follows: if node v is the source of a non spanning tree edge with target u, then u as well as *all* its ancestors in the graph replicate the label of v. Such a scheme favors efficient subsumption checking (i.e., comparing sets of labels for each class) in the graph while the price to be paid is the additional storage cost of propagated labels. In the worst case of bipartite graphs, the extra storage is $O(n^2)$, but fortunately this is not the case of class semi-lattices represented in RDF/S. Table 2, column 3 illustrates the expression of our testbed queries in this scheme when the



Node	Agrawal et all			Li and Moon			Dewey		
	Spanning Tree	Graph Propagation	Compres-sion	Spanning Tree	Graph Propagation	Compres-sion	Spanning Tree	Graph Propagation	Compression
A	[1,9]	[1,2] [3,3] [3,5] [3,5]	[1,9]	[1,9]	[2,2] [7,1] [6,3] [6,3]	[1,9]	'1'		
B	[1,2]	[3,3] [3,5]	[1,5]	[2,2]	[7,1] [6,3]	[2,2][6,3]	'111'	'1'	'111'
C	[1,8]	[3,3] [3,5] [3,5]	[1,8]	[4,8]	[7,1] [6,3] [6,3]	[4,8][6,3]	'11'		
D	[1,1]	[3,3]		[3,1]	[7,1]		'1111'	'1'	'1111'
E	[3,6]			[5,4]			'112'		
F	[7,7]	[3,5]		[9,1]	[6,3]		'113'		
G	[3,5]			[6,3]			'1121'	'111' '1' '113'	'1121' '111' '113'
H	[3,3]			[7,1]			'11211'	'111' '1' '1' '113' '1111'	'11211' '1111''113'
I	[4,4]			[8,1]			'11212'	'111' '1' '113'	'11212'113' '111'

Figure 3: Label Compression for Graphs: a) Agrawal et al b) Li and Moon c) Dewey

encoded class hierarchies are trees (the case of DAGs will be addressed in Subsection 4.2). Finally, to support incremental updates without node relabeling one can leave gaps between the intervals generated during the bottom-up tree traversal using some constant factor c in the postorder numbering, i.e., the label of a node u is $[index(u), c*post(u)]$. Other interval computation policies (out of the scope of this paper) use, for instance, a top-down traversal in order to encode at each level random or adaptive size gaps for node intervals w.r.t. to the prediction of future updates

It should be stressed that for trees, Agrawal, Borgida, Jagadish scheme is equivalent to the scheme proposed by Li and Moon [17] (see Figure 2-e) for encoding XML data where the label of a node u is $[pre(u), size(u)]$ ($size(u)$ denotes the size of the subtree rooted at u). It is also identical to the scheme by Schubert et al [19] (with inverse query conditions) recently studied for XML trees in [21] where the label of a node u is $[pre(u), index(u)]$ ($index(u)$ is the highest preorder number of u's descendants). Compared to these variations the *extended postorder* scheme of Agrawal et al has the following advantages: (a) it requires smaller index volumes (and update costs) since we need only a B-tree on the *post* value of labels (as opposed to Dietz's labels [10, 11] requiring indices on both $pre(u)$ and $post(u)$ values and Zhang's variation [26] requiring an extra index on *depth*); (b) it allows for more efficient query evaluation by standard SQL engines since the core conditions for the structural relationships among nodes are simpler (unlike labels in the scheme by Li and Moon involving arithmetic operations in all queries); (c) it finally exhibits interesting interval compression opportunities for graphs either by absorbing subsumed intervals or by merging adjacent intervals coming from non spanning tree edges.

Consider for example the DAG D illustrated in the left part of Figure 3. The nodes of D represent classes and the edges *isA* links defined between them. The link from B to A is redundant but such a redundancy is frequent in RDF/S schemas found on the Web [18]. Note also that precedings/followings queries (see Table 2) are meaningless in a graph setting. In order to label D, the scheme by Agrawal, Borgida, Jagadish [1] chooses an optimal spanning tree T w.r.t the number of generated labels, based on the number of ancestors per node: an edge of D from n to n' belongs to T (represented by solid lines) only if n' has the maximum number of ancestors w.r.t. the other edge target nodes with source node n. For instance, the edge from B to C belongs to the spanning tree while the edge from B to A does not (dashed line). Only non redundant edges belong to the optimal spanning tree. Then (see the right part of Figure 3) for each non spanning tree edge (e.g., from H to D the interval of the source node (e.g., `[3,3]`) is propagated to the target node (e.g., D) and recursively up to its ancestors (e.g., B, C, A). However, when propagated *upwards*, the intervals of descendent nodes may be subsumed by those of ancestors (e.g., `[3,3]` is subsumed by both C and A intervals). Therefore they can be *absorbed* by the label of a node (either from the spanning tree or propagated) representing their nca. In addition, adjacent intervals like `[1,2]` and `[3,3]` can be *merged* into a new one `[1,3]` without breaking down the interval inclusion rule which captures the node ancestor relationship (e.g., after merging B is an ancestor of D and H). Such interval merging, clearly depends on the order of edges belonging to the spanning tree [1] while it affects the identification of nodes based on their postorder number (we come back on this issue in Subsection 4.2). At the end of the compression process, the scheme requires only two additional intervals (for D and F) for the four non spanning tree edges of our example.

The same label propagation can be also applied to other interval based schemes such as the one by Li and Moon [17]. However, the compression rate is significantly reduced: interval merging is not possible while interval subsumption (w.r.t the subsumption checking conditions of Table 2) is limited (e.g., `[7,1]` is subsumed by `[6,3]`). The Dewey prefix-based scheme [6] can similarly extended with additional labels in the case of DAGs. We rely, as previously, on the same spanning tree choice but the propagation of labels is now performed *downwards* i.e., from the target of non spanning tree edges (e.g., A) to the source node (e.g., B and its descendants (e.g., D, G, H and I). The only possible compression in this scheme is the absorption of a label when it already appears as a prefix of another; for instance, `'1'` is absorbed by `'111'`, `'11111'` etc. As illustrated in Figure 3, in our simple example Dewey's scheme requires six additional labels (for G, H and I).

In summary, bit-vector based schemes do not efficiently support all our testbed queries when implemented by SQL engines. Prefix-based schemes provide simple expressions for ancestor/descendant queries based on string matching operators and allow for simple incremental updates. However, in this scheme the optimization opportunities of existing SQL engines are reduced for some of our testbed queries. Among the interval-based schemes, the extended postorder interval scheme proposed by Agrawal, Borgida, Jagadish (referred to as $PInterval$) presents several advantages among which compactness for DAG hierarchies and efficient query evaluation by standard SQL engines are noteworthy. The experimental study presented in the next section compares its performance with that of the Unicode Dewey prefix scheme (referred to as $UPrefix$) in terms of storage volumes and query evaluation time.

4 Evaluation of Labeling Schemes

In this section, we compare the storage and query performance of two labeling schemes when implemented with an SQL engine, namely the Unicode Dewey prefix-based scheme ($UPrefix$) and the extended postorder interval-based scheme by Agrawal, Borgida and Jagadish ($PInterval$). We use as a testbed for our evaluation the RDF dump of the ODP Catalog (version of 16-01-2001). We successively study the case of subclass trees (i.e., the ODP hierarchies with single *isA*) and DAGs (i.e., the ODP hierarchies are augmented with synthetically generated multiple *isA* links). Experiments were carried out on a Sun-Blade-1000, with an UltraSPARC-III 750MHz processor and 512 MB of main memory, using PostgreSQL (Version 7.2.1) with Unicode configuration. 1000 buffers (8KB) were used for data loading, index creation and querying. 16 ODP class hierarchies (see Table 1) with a total number of 253215 topics were loaded. Indices on the generated labels were constructed after file sorting on the index key in order to use packed B-trees.

We first choose a relational representation of $UPrefix$ and $PInterval$ labels in order to compare the resulting database size. The performance of the testbed queries (see Table 2) is then compared when implemented with the PostgreSQL engine.

4.1 The Case of Trees

4.1.1 Database Representation and Size

The RDF/S class (or property) hierarchy of a Portal Catalog like ODP, can be represented by one table with two attributes: the name of the class (primary key) and the name of its parent class. Because in ODP the class names are large variable size strings (path from root including namespace and path prefix) we choose the following normalized relational database schema:

$$Class(\underline{id} : int4, name : varchar(256))$$
$$SubClass(\underline{id} : int4, parent : int4)$$

where *id* is a class identifier, *name* is its name, and *parent* is the parent class identifier.

Since the labels produced by $UPrefix$ or $PInterval$ are unique, they can be used (or a part of them) as identifiers of classes in the tree. In the following, we evaluate the database and index size of the following tables replacing $SubClass$ respectively by:

$$UPrefix(\underline{label} : varchar(15), parent : varchar(15))$$
$$PInterval(index : int4, \underline{post} : int4, parent : int4)$$

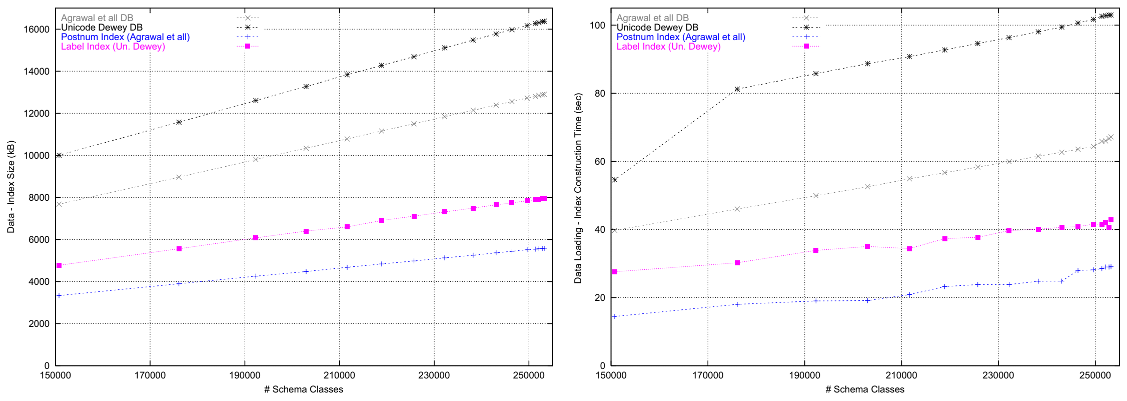


Figure 4: a) Database/Index Size and b) Construction Time for ODP Subclass Trees

where *parent* respectively stores the parent's string label or post-number value.

Two remarks are noteworthy. First the string type of attribute *label* in $UPrefix$ is determined by the maximum depth of the ODP class hierarchy (see Table 1) plus one (for the root class *Resource*) while the type of the *post* (and *index*) attribute in $PInterval$ by the total number of the ODP classes. Second, in both cases we utilize the *parent* attribute in order to reconstruct the class hierarchy in RDF/S from the database as well as to efficiently support direct parent/children/sibling queries. This choice is justified by the significant evaluation cost of these queries in SQL engines with user-defined functions like $prefix$ in $UPrefix$ or additional information on node labels like *depth* in $PInterval$ (otherwise finding the direct children of *Resource* requires a complete scan of the ODP hierarchy!).

Figure 4-a displays the size of the database (tables $UPrefix$ and $PInterval$) and the size of the index (respectively on attributes *label* and *post*) while Figure 4-b displays the construction time when the 16 ODP hierarchies (see Table 1) are loaded in decreasing order of their number of classes. More precisely, the size of table $UPrefix$ is 16376 Kb and the size of $PInterval$ is 12902 Kb both containing 253215 tuples (i.e., classes) on 2073 and 1613 disk pages respectively. Equivalently, to store the label of a class as well as the label of its superclass (i.e., a tuple) we need 52,17 bytes with $PInterval$ and 66.22 bytes with $UPrefix$. Compared to the $PInterval$ 12 bytes expected from the schema, the extra storage cost per tuple is due to an id (40 bytes) generated by PostgreSQL to identify the physical location of a tuple within its table (block number, tuple index within block). In addition, the PostgreSQL storage requirement for string types is 4 bytes plus the actual string size. For these reason we need on the average[8] 13.11 bytes for storing the class *label* in $UPrefix$. It should also be emphasized that only 0.133% of the encoded classes (2 classes have a fan-in degree > 256 with 336 subclasses) in $UPrefix$ require labels with Unicode characters exceeding the two bytes.

Table $UPrefix$ is 21.2% bigger than $PInterval$, while the size of the index on attribute *label* is 29.8% larger (1001 disk pages) than that of *post* (697 disk pages). On the other hand, data loading (index construction) time of $UPrefix$ is 34,75% (32,21%) larger than of $PInterval$. Slightly smaller size and time have been obtained for the indices on attribute *parent* in both tables (due to the indexing of smaller ranges of values). Clearly, the extra storage cost of $PInterval$ is due to a significant overhead for storing and indexing strings in the PostgreSQL DBMS.

4.1.2 Core Query Evaluation

In this subsection, we are interested in the efficient implementation of the Portal query functionality for both prefix and interval labeling schemes using standard SQL engines. Most query expressions (see Table 2) can be directly translated into SQL, using the relational schema of the previous section. The only queries for $UPrefix$ needing to be implemented by SQL stored procedures are **ancestors** (function $prefixes$) and **nca** (functions $prefixes$ and $maxlength$). Stored procedures are also employed to implement the **subsumption** checking on two class labels for both schemes. It should be stressed that for optimization reasons queries such as **leaves** for $UPrefix$ and **followings** for $PInterval$ need to be rewritten.

[8]Note that the average depth of ODP class hierarchies including the root *Resource* is 8.83 (see Table 1).

More precisely, the main performance limitation of SQL queries for $UPrefix$ is due to the presence of user-defined functions (*next*, *prev* and $maxprefix$) in the selection conditions involving the attribute $label$. Such queries are evaluated by the SQL engine without taking into account the existence of an index defined on $label$. To solve this limitation, when possible user-defined functions are evaluated prior to the execution of the SQL query. For instance, the query **descendants** of the root class *Resource* uses the condition $label >$ `'1'` $\wedge$ $label <$ $next$('1'). Since function *next* is applied to the input node of the query (e.g., the label '1' of *Resource*) the condition can be replaced by `'1'` $\wedge$ $label <$ `'2'` ($next$('1') = `'2'`) where *next* has been pre-evaluated. However, this rewriting is not always possible, as in query **leaves** where the function *next* is used in the nested subquery over the labels returned by the outer block:

```
select  label
from    UPrefix
where   label > '1' and label <'1n' and
        not exists (select  *
                    from    UPrefix u'
                    where   u'.label > label and
                            u'.label < next(label))
```

For this reason, the previous query was rewritten so as to involve only string operations and not functions on $label$:

```
select  label
from    UPrefix
where   label > '1' and label < '1' || 'xFF' and
        not exists (select  *
                    from    UPrefix u'
                    where   u'.label > label and
                            u'.label' < label || 'xFF')
```

The string operator `||` concatenates the Unicode character 'xFF' ("all-ones" byte) to the value of attribute *label*. The resulting string is the maximal string inferior to *next*(*label*)[9]. Then the index can be used during the evaluation of the nested query. Other rewritings were experimented with (e.g., using structural information represented by attribute *parent*) but the previous solution exhibited the best performance.

The only problem for the interval based scheme is related to the **followings** query. It relies on the values of the attribute *index* for which no index was constructed. In order to use only the available index on *post*, we rewrite the query as follows:

```
select  post
from    PInterval
where   post > p and index > i
```

The selection condition is equivalent to the original one $index > p$ (in $PInterval$ following nodes have always greater postorder and index numbers[10]) and query evaluation can be optimized with the use of the B-tree defined on *post*.

Except for the two previous rewritings, the evaluation of the core queries with the two labeling schemes strictly uses the conditions stated in Table 2. Each query was run several times: one initially to warm up the database buffers and then nine times to get the average execution time of a query. Recall that 1000 buffers of size 8KB and thus the indices of attributes *label* (1001 disk pages) and *post* (697 disk pages) can fit entirely in main memory. Table 3 gives the resulting execution time in seconds (using PostgreSQL `Explain Analyze` facility) for both schemes and for up to three different cases per query: each case corresponds to a different choice of input node and therefore of query selectivity.

The main observation is that the query performance of the two labeling schemes is comparable. The **leaves** query is penalized in $UPrefix$ by the use of nested queries. Compared to $PInterval$, **ancestors** and **nca** run with the former scheme in practically constant time. In all other queries, $PInterval$ exhibits slightly smaller execution times than $UPrefix$ since for the same number of

[9]Note that $label\ xFF$ is an imaginary rightmost child ('xFF' cannot actually be used in a valid UTF-8 encoding) for the node with label *label* whose immediate right following node has the label *next*(*label*).

[10]The second condition is used to eliminate ancestors.

Query	**PInterval**			**UPrefix**		
	Case 1 %Select	Case 2 %Select	Case 3 %Select	Case 1 %Select	Case 2 %Select	Case 3 %Select
subsumption check	0,00018	0,00018	0,00017	0,00017	0,00017	0,00017
Q1	2,869392	0,429730	0,00027	3,296126	0,57702	0,00027
descendants	100%	10%	0,0004%	100%	10%	0,0004%
Q2	1,18904	1,54799	0,00027	0,00163	0,00178	0,00167
ancestors	0,002%	0,00158%	0,00355%	0,002%	0,00158%	0,00355%
Q3	2,957039	0,493230	0,00027	25,54267	16,51963	0,00055
leaves	75,13%	8,46%	0,0004%	75,13%	8,46%	0,0004%
Q4	2,629188	2,34529	0,00024	2,874457	2,3453	0,0005
precedings	100%	50%	0%	100%	50%	0%
Q5	2,9721	2,48579	0,00024	3,37825	2,566292	0,0005
followings	100%	50%	0%	100%	50%	0%
Q6	0,00863	0,00054	0,00047	0,00961	0,00056	0,00049
siblings	0,1236%	0,002%	0,0004%	0,1236%	0,002%	0,0004%
Q7	3,22742	0,00041	0,000438	0,0003945	0,0003945	0,0003945
nca	0,0004%	0,0004%	0,0004%	0,0004%	0,0004%	0,0004%

Table 3: Execution Time of Core Queries for the ODP Subclass Tree

returned tuples a smaller number of disk pages need to be accessed. Finally, PostgreSQL (cost-based) query optimizer seems to favor index scans on tables *UPrefix* and *PInterval* although sequential scans should be more efficient (e.g., in queries with 50% selectivity!). This is due to inaccurate selectivity estimations (higher) of query predicates especially for string comparisons in *UPrefix*. The same plans and comparable execution times for all queries have been observed when augmenting the number of buffers from 1000 to 10000.

In **Q1** each case corresponds to the choice of a different node for which the descendants are computed: (a) in Case 1 the root (i.e., *Resource*) (b) in Case 2 a node with a medium number of descendants (i.e., *Arts*) and (c) in Case 3 a node with a minimum number of descendants. In Cases 2 and 3, the node label appears in the middle of the *post* or *label* intervals of values. PostgreSQL optimizer chooses for both labeling schemes a sequential scan for the first case and index scans for the other two. Since the interval query is based exclusively on *post* (e.g., $i <= post < p$) or *label* (e.g., $l < label < l'$) index scan is beneficial: the optimizer uses the index to access the tuple satisfying the lower bound condition and since the examined index keys are sorted, it stops sequential scan of tuples when the upper bound is reached.

The three cases of input nodes for **Q2** correspond to (a) the leftmost (b) a middle and (c) the rightmost leaf of the ODP subclass tree. The response time is significantly better for the Prefix scheme in the first two cases. PostgreSQL optimizer chooses for *PInterval* (for *UPrefix* stored procedures are used) a sequential scan for Case 1 and index scans for Cases 2 and 3. The interval query is based now on different attributes namely *post* and *index* ($index <= p \land p < post$ since for leaves $index = post$) and all values returned by the index scan (on *post*) have to be scanned to check the first condition (on *index*). The wrong selectivity estimation for the conjunction leads the optimizer to favor in Case 2 an index scan (on the half interval) which turns out to be much more costly than a sequential one (on the entire interval)!

Q3 is evaluated with the same input nodes as **Q1**. Thus, for *PInterval*, the PostgreSQL optimizer chooses the same plans in the three cases. The slightly higher execution times compared to **Q1** are due to the evaluation of the extra condition for leaves ($index = post$) given that the number of accessed disk pages are the same. On the other hand, *UPrefix* is significantly penalized by the use of the nested query: index scans are used for the nested query in all cases while a sequential scan should be used at least for Case 1.

Queries **Q4** and **Q5** employ the same input nodes as **Q2** and the three cases for **precedings** and **followings** have inverse selectivities. The execution times for queries with zero selectivities (Case 3) give us an indication about the lookup cost of indices defined on attributes *post* and *label*.

Q6 is evaluated with input nodes having the maximum, a medium and the minimum fan-in degrees of ODP subclass trees. It involves a nested loop join over two index scans: one to find the parent of a node and the other to find its direct siblings using equality on *post* (*label*) and *parent*.

Finally, **Q7** takes as an input a pair of nodes (using the same leaves as in **Q2**): in Case 1 the leftmost-rightmost leaves, in case 2 the leftmost-middle leaf and in Case 3 the middle-rightmost leaves. For $UPrefix$ a stored procedure is executed, while for $PInterval$ a nested query is evaluated using index scans for both the inner and outer blocks in the three cases. In Case 1 the resulting time for the interval based scheme is significant. However as aforementioned, a sequential scan should be chosen. For Cases 2 and 3 the response times are comparable.

4.2 The Case of DAGs

In this section we first present the relational representation of $UPrefix$ and $PInterval$ labels in the case of a subclass DAG and evaluate the extra storage cost for both labeling schemes. We then show, as for the case of trees, how **subsumption check**, **descendant**, **ancestor**, **leaves**, **siblings** and **nca** queries (**preceding** and **following** queries are not defined on DAGs) can be expressed on the label representation of the hierarchy and translated into SQL queries. We end up our study by a performance comparison of the two schemes in terms of query response time.

4.2.1 Database Representation and Size

In each labeling scheme, two tables are now necessary for representing the class hierarchy, apart from table $Class$ with attributes *id* and *name*. The first table in both schemes is the same as in the case of trees ($UPrefix$, $PInterval$). The only modification is that for DAGs, tuples in these tables represent both kinds of edges (spanning-tree or non-spanning-tree edges). The rationale behind this choice is that **siblings** (and parent/children) queries can be easily evaluated on tables $UPrefix$ and $PInterval$ using the *parent* attribute (as in the case of trees). This choice implies the extension of both tables key in order to include the *parent* attribute, as follows:

$$UPrefix(\underline{label} : varchar(15), \underline{parent} : varchar(15))$$
$$PInterval(index : int4, \underline{post} : int4, \underline{parent} : int4)$$

It should be stressed that when label compression in $PIn-\ terval$ also considers the merging of adjacent intervals, DAG nodes are not anymore identified using their postorder number. For instance, in Figure 3 both nodes C and G have as a *post* value 5. As shown in the following, the total label compression gains from merging is less than 0.6% and therefore we do not consider this compression in the following.

The second table is respectively called $DUPrefix$ and $DPInterval$ in the two schemes where D stands for DAG. In the former table, tuple (*label*, *ancestor*) indicates that the node with label *ancestor* propagates *downwards* its label to the node identified by *label*. In the latter, tuple (*index*, *post*, *ancestor*) indicates that the node with label [*index*, *post*] propagates its label *upwards* to the node identified by the post value *ancestor*. Keys are not mandatory for these tables because they are not accessed independently from the primary table (indices have been defined on attributes *ancestor* and *label* or *post*).

$$DUPrefix(label : varchar(15), ancestor : varchar(15))$$
$$DPInterval(index : int4, post : int4, ancestor : int4)$$

Looking at Figure 3, left, the label `[6,3]` of G is propagated up only to B since it is absorbed by A. Then $DPInterval$ includes one tuple $(3, 5, 2)$ where $3, 5$ account for the *index* and *post* values of G and 2 for the *post* value of B (i.e., its id). Similarily the $DUPrefix$ table includes the two tuples $('1121', '111')$, and $('11212', '111')$ that account for the propagation of B's label down to its descendants G, and I (for H, B's label is absorbed by the propagated label '1111' of D). Note the redundancy of the attribute *index*, since any node is identified by its post value. This redundancy allows for a faster SQL execution of the **descendants** query. It should be stressed that when label compression is not considered in both schemes, table $DUPrefix$ ($DPInterval$) essentially materializes the result of **descendents** (**ancestors**) query involving DAG edges.

Let us now evaluate the extra storage cost for labeling DAGs with the two schemes. Since in both cases the tables $UPrefix$ and $PInterval$ hold all the edges of the DAG (to enable reconstruction in RDF/S), the extra storage space is exactly the size of tables $DUPrefix$ and $DPInterval$: for each scheme we only need to measure the number of propagated labels. This (downwards or upwards) propagation depends on the position of the source and target nodes of the non spanning tree edges in the DAG or more precisely the number of descendants (ancestors) of source (target) nodes. The DAG testbed uses the ODP hierarchies (see Table 1) augmented with synthetically generated multiple *isA* links. The original ODP classes are decomposed into three sets according

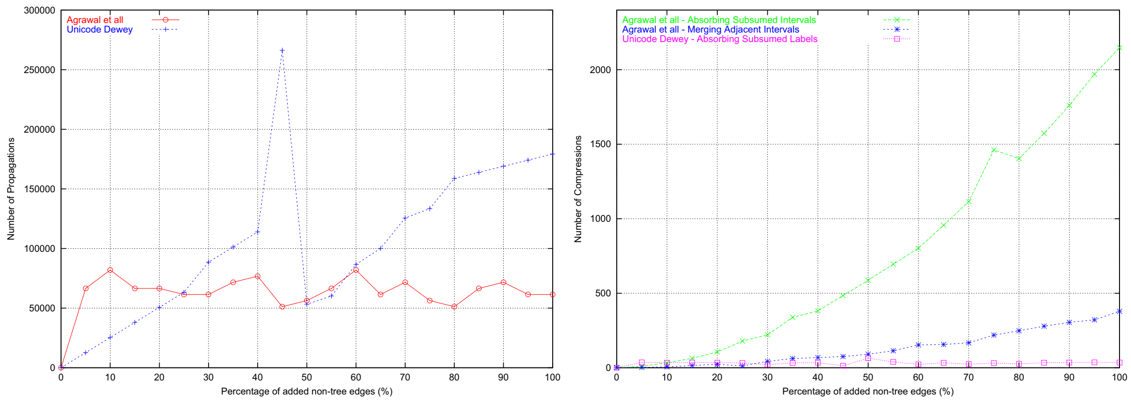


Figure 5: Label Propagation and Compression for ODP Subclass DAGs

to their depth in the tree: a) near to the root (denoted R); b) near to the leaves (denoted L); and c) in between (denoted B). Then, picking at random the source and target edge classes, additional edges are equally distributed in nine groups: RR, RL, RB, etc. In addition, the maximum fan-out degree of classes is fixed to 2 (a typical upper bound of multiple isA links as observed in [18]).

The total number of label propagations is displayed in Figure 5 versus the percentage of additional edges. The experiment was conducted incrementally until the number of original ODP tree edges is doubled (100% percentage of additional edges): for every 5% generated edges, we execute the two labeling algorithms. Note that the spanning tree computed (for both algorithms) is different at each increment step. The main observation from Figure 5 is that the number of label compressions in $PInterval$ is proportional to the number of additional edges, regardless of their positioning in the DAG, which is not the case for $UPrefix$. For this reason, the number of label propagations for $PInterval$ is stabilized between 50000-80000, while for $UPrefix$ it seems to depend on the actual number of descendants of the source class of each additional edge. Clearly, when a significant number of edges has been added (e.g., 65%) label propagation in the two schemes diverges significantly. In addition, the number of adjacent label mergings in $PInterval$ is always smaller than the number of subsumed label absorptions, while ignoring labels' merging (in order to maintain postorder numbers as class identifiers) implies only 2492 additional tuples in $DPInterval$ (i.e., 4%). Practically speaking, for 253214 additional edges (i.e., 100%) $DUPrefix$ will contain 179270 tuples and $DPInterval$ 63937 (i.e., 61445 plus 2492) when compression is based only on the absorption of subsumed labels. This DAG testbed will be used in the sequel for evaluating the query performance of both labeling schemes. When labels' compression is completely ignored, the size of table $DPInterval$ is three times bigger, while $DUPrefix$ has almost the same size (due to the very small numbers of compressions).

4.2.2 Core Query Evaluation

In Table 4, we provide, for both labeling schemes, a declarative formulation of the five testbed queries expressed in terms of the queries defined for the tree case. We denote by $Propdown(u)$ in $DUPrefix$ the set of descendant nodes of u to which u's label is propagated and by $Propanc(v)$ the set of ancestors u of v such that $v \in Propdown(u)$. Similarly, $Propup(u)$ in $DPInterval$ is the set of ancestor nodes of u to which the label of u is propagated as an additional label and $Propdesc(v)$ is the set of descendants u of v such that $v \in Propup(u)$. Subsumption checking for two DAG nodes u and v evaluates to true in $DUPrefix$ ($DPInterval$) iff the $subsumption(u, v)$ condition given in the case of trees (see Table 2 columns 2,3) is true or $u \in Propanc(v)$ ($v \in Propup(u)$). In the sequel, we provide the SQL translation of the declarative expressions for $Ddescendants$, $Dancestors$ and $Dleaves$. Clearly, label compression result to more complicated query expressions because the paths connecting two DAG nodes through non spanning tree edges are not completely materialized in tables $DUPrefix$ and $DPInterval$. On the other hand, it ensures that no descendant/ancestor is computed more than once when querying both the tables $UPrefix$ (or $PInterval$) and $DUPrefix$ (or $DPInterval$). In other words, we don't need to eliminate duplicates in the union of the two subqueries (i.e., for computing tree and DAG descendants/ancestors).

Query $Ddescendants(v)$ uses the $descendants(v)$ expression given for the case of a tree (see Table 2, columns 2,3). In both schemes, it also finds the descendants related to propagated labels

Query	**DUPrefix**	**DPInterval**
$Ddescendants(v)$	$descendants(v) \cup Propdown(v)$	$descendants(v)$ $\bigcup_{w \in Propdesc(v)} descendants(w)$
$Dancestors(v)$	$ancestors(v)$ $\bigcup_{w \in Propanc(v)} ancestors(w)$	$ancestors(v) \cup Propup(v)$
$Dleaves(v)$	$\{u \mid u \in Ddescendants(v) \wedge$ $Propdown(u) = \emptyset \wedge$ $\nexists u' \mid u' \in Ddescendants(u) \wedge$ $Propdown(u') = \emptyset\}$	$\{u \mid u \in Ddescendants(v) \wedge$ $Propdesc(u) = \emptyset \wedge$ $\nexists u' \mid u' \in Ddescendants(u) \wedge$ $Propdown(u') = \emptyset\}$
$Dsiblings(v)$	$siblings(v)$	$siblings(v)$
$Dnca(v, w)$	$\{u \mid u \in Dancestors(v) \wedge$ $u \in Dancestors(w) \wedge$ $\nexists u' \mid u' \in Dancestors(v) \wedge$ $u' \in Dancestors(w) \wedge$ $u' \in Ddescendants(u)\}$	$\{u \mid u \in Dancestors(v) \wedge$ $u \in Dancestors(w) \wedge$ $\nexists u' \mid u' \in Dancestors(v) \wedge$ $u' \in Dancestors(w) \wedge$ $u' \in Ddescendants(u)\}$

Table 4: Core Query Expressions for DAGs: a) DUPrefix b) DPInterval

of v, respectively given by *Propdown*(v) and *Propdesc*(v). In the absence of compression, the expression *Propdown*(v) would be expressed by the following simple SQL query, where 'l' denotes the label of v ($UPrefix$):

`select` *label* `from` $DUPrefix$ `where` *ancestor*= 'l'

Because of the label compression the corresponding SQL query employs also a nested query on $UPrefix$ for finding the descendants in paths involving DAG edges:

```
select  w.label
from    DUPrefix w, (select  u.label as label
                     from    UPrefix u
                     where   u.label >= 'l' and
                       u.label < 'l' || 'xFF') u'
where   w.ancestor = u'.label
```

Denoting the label of v by `[i,p]` (*PInterval*) the union subquery on *Propdesc*(v) is translated into SQL as follows:

```
select  u.post
from    PInterval u, DPInterval w
where   w.ancestor = p and u.post >= w.index
        and u.post <= w.post
```

The query $Dancestors(v)$ relies in turn on the $ancestors(v)$ expression given for the case of a tree. In both schemes, it also considers the ancestors related to propagated labels of v which are given respectively by the expressions *Propanc*(v) and *Propup*(v). The SQL translation of *Propup*(v) for *DPInterval* with label compression is given below:

```
select  v.ancestor
from    DPInterval v,
        (select  u.post as post
         from    PInterval u
         where   u.index <= p and u.post >= p) u'
where   v.post = u'.post
```

In effect, the nested query computes all tree ancestors of v (including itself), while the outer query returns the ancestors of all propagated labels of any of those nodes.

The query $Dancestors(v)$ for $DUPrefix$ is translated into a stored procedure which employs an intermediate SQL query to compute the expression *Propanc*(v):

```
function Gancestors( l )
let anc = { };
let anc += Prefixes(l)
let labels = { };
let labels += (select ancestor from DUPrefix where label = 'l');
let anc += labels
while (labels.next)
       anc += Prefixes(labels.next);
return anc;
```

where $Prefixes()$ is a function giving the set of prefixes of a string 'l'. Note that the labels of $Propanc(v)$ as well as their prefixes are included in the result of $Gancestors(v)$ (duplicate elimination is required in this case).

For query $Dleaves(v)$ we obtain the following SQL translation in $DPInterval$:

```
select  u.post
from    PInterval u
where   u.post < p and u.post >= i and u.post = u.index
        and not exists (select  *
                        from    DPinterval u'
Union All               where   u'.ancestor = u.post)
select  u.post
from    PInterval u, DPInterval w
where   w.ancestor = p and u.post >= w.index
        and u.post <= w.post
        and not exists (select  *
                        from    DPinterval u'
                        where   u'.ancestor = u.post)
```

For $DUPrefix$, the SQL query is similar and uses the SQL translation of $Ddescendants$:

```
select  label
from    UPrefix
where   label > 'l' and label < 'l' || 'xFF'
        and not exists (select  *
                        from    UPrefix u'
                        where   u'.label > label and
                                u'.label' < label || 'xFF')
        and not exists (select  *
                        from    DUPrefix u"
Union All               where   u".ancestor = label)
select  w.label
from    DUPrefix w,
        (select  u.label as label
         from    UPrefix u
         where   u.label >= 'l' and
        u.label     < 'l' || 'xFF') u'
where   w.ancestor = u'.label
        and not exists (select  *
                        from    UPrefix u"
                        where   u".label > w.label and
                                u".label' < w.label || 'xFF')
        and not exists (select  *
                        from    DUPrefix u"'
                        where   u"'.ancestor = w.label)
```

$Dsiblings(v)$ has exactly the same expression as for the tree case. The SQL translation of $Dnca(v, w)$ for $DPInterval$ is given in Appendix A and uses nested subqueries as presented previously for $Dancestors$. In $DUPrefix$ however the expression is much simpler since it relies on string functions as illustrated in Appendix B.

Table 5 shows the execution times of the testbed queries for the synthetically generated ODP DAGs (100% of Figure 5) using the same input nodes as in the case of trees (see Table 3). Due to the additional DAG edges (on the same ODP nodes) the size of tables $UPrefix$ and $PInterval$ is practically doubled and the query selectivities are accordingly increased, despite the fact that additional nodes are returned by some of our queries. The main observation is that $DPInterval$ outperforms $DUPrefix$ by up to 5 orders of magnitude for **descendants** and **leaves** queries especially for cases with high selectivity (i.e, 3). This is due to the evaluation of the nested subqueries in the from clause of these queries using merge-joins over string attributes. String sorting exhibited unacceptable execution time in PostgreSQL, compared to integer sorting involved in the evaluation of the **ancestors** query in $DPInterval$ using the same execution plan.

Query	DPInterval			DUPrefix		
	Case 1 %Select	Case 2 %Select	Case 3 %Select	Case 1 %Select	Case 2 %Select	Case 3 %Select
Q1	11,92988	0,617279	0,000263	67,12124	21,78665	20,4409
descendants	50%	5,051%	0,0004%	50%	5,051%	0,0004%
Q2	3,790538	2,6323	3,095479	0,00619	0,00514	0,00472
ancestors	0,033%	0,00454%	0,00276%	0,033%	0,00454%	0,00276%
Q3	14,50642	1,57837	0,000087	95,58222	48,192656	47,297535
leaves	21,55%	2,211%	0,0002%	21,55%	2,211%	0,0002%
Q4	0,002591	0,002442	0,00033	0,003011	0,002946	0,00032
siblings	0,0626%	0,0008%	0,0004%	0,0626%	0,0008%	0,0004%
Q5	4,870160	4,691551	3,70287	0,00058	0,00057	0,00055
nca	0,0002%	0,0002%	0,0002%	0,0002%	0,0002%	0,0002%

Table 5: Execution Time of Core Queries for the ODP Subclass DAG

On the other hand, **ancestors** and **nca** in *DUPrefix* run in practically constant time. Although not detailed in this paper, when we ignore label compression, no significant performance gains are obtained for both schemas due to the extra cost of label's sorting and duplicate elimination (i.e., `Union` vs. `Union All`) in queries, especially for string labels.

5 Summary

A number of interesting conclusions can be drawn from the conducted experiments. Firstly, for voluminous class (or property) *subsumption* hierarchies, labeling schemes bring significant performance gains (3-4 orders of magnitude) in query evaluation as compared to transitive closure computations [14]. Secondly, this gain comes with no significant increase in storage requirements for the case of tree-shaped hierarchies especially for the interval schema while the query performance for both schemes is comparable. For DAG-shaped hierarchies, we need for the interval (prefix) schema up to 2.4 (2.7) times more storage space when the propagated labels are compressed. In particular, for practical cases (i.e., small percentage of added non tree edges) the interval schema is less sensitive than the prefix one, to the propagation of labels w.r.t. the actual position of the source and target nodes of the added DAG edges. Significant divergent behavior in labels' propagation is observed when the percentage of the added DAG edges increases substantially ($> 65\%$). Thirdly, for **descendants** and **leaves** queries on DAGs, interval schemes are up to five times more costly than in the case of trees, compared to prefix ones which are up to 5 orders of magnitude more costly. However, **ancestors** and **nca** in *DUPrefix* run in practically constant time for both trees and DAGs. When labels' compression is ignored, the two schemes exhibit almost the same storage requirements while their query performance is slightly improved. This is due to the PostgreSQL questionable choice of optimization strategies for complex queries over string attributes and their surprisingly bad execution time. We are planning to study this issue w.r.t. other DBMS. Finally, our algorithm for *subsumption* DAGs can be adapted to the *data paths* of resource descriptions formed by RDF properties. As a future work we intend to compare our approach with other path indexing strategies as proposed in the literature.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. of the SIGMOD Inter. Conf. On Manag. Of Data*, pages 253–262, 1989.

[2] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Trans. on Progr. Languages and Systems*, 11(1):115–146, 1989.

[3] S. Alexaki, G. Karvounarakis, V. Christophides, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *2nd Inter. Workshop on the Semantic Web*, pages 1–13, Hong Kong, 2001.

[4] D. Brickley and R.V. Guha. Resource Description Framework (RDF) Schema Specification 1.0, W3C Candidate Recommendation, 2000.

[5] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA '93*, Washington, 1993.

[6] Online Computer Library Center. Dewey decimal classification. Available at www.oclc.org/dewey/.

[7] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *Proc. of the Inter. Conf. On Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002.

[8] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic xml trees. In *Proc. of the Twenty-first Symposium on Principles of Database Systems (PODS'02)*, Wisconsin, USA, 2002. ACM.

[9] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. of the Inter. Conf. On Very Large Data Bases (VLDB'01)*, 2001.

[10] P. F. Dietz. Maintaining order in a linked list. In *Proc. of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 122–127, San Francisco, California, USA, 1982.

[11] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the Sixteen Annual ACM Symposium on Theory of Computing (STOC'87)*, pages 365–372, New York, USA, 1987.

[12] C. Gavoille and D. Peleg. Compact and localized distributed data structures. *Journal of Distributed Computing, Special Issue for the Twenty Years of Distributed Computing Research*, 2003.

[13] R. Shabo H. Kaplan, T. Milo. A comparison of labeling schemes for ancestor queries. In *Proc of the thirteen Annual Symposium on Discrete Algorithms (SODA'02)*, San Francisco, California, USA, 2002.

[14] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proc. of the Eleventh Inter. World Wide Web Conf. (WWW'02)*, pages 592–603, Honolulu, Hawaii, USA, 2002.

[15] A. Krall, J. Vitek, and N. Horspool. Near optimal hierarchical encoding of types. In *11th European Conf. on Object Oriented Programming (ECOOP'97)*, pages 128–145, Finland, 1997.

[16] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation, 1999.

[17] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of 27th Inter. Conf. on Very Large Data Bases(VLDB'02)*, pages 361–370, Roma, Italy, 2001.

[18] A. Maganaraki, S. Alexaki, V. Christophides, and D. Plexousakis. Benchmarking rdf schemas for the semantic web. In *Proc. of the First Inter. Semantic Web Conf. (ISWC'02)*, pages 132–147, Italy, 2002.

[19] L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Accelerating deductive inference: Special methods for taxonomies, colours and times. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 187–220. Springer-Verlag, New York, 1987.

[20] N. Spyratos, Y. Tzitzikas, and V. Christophides. On personalizing the catalogs of web portals. In *Proc. of the Special Track on the Semantic Web at the 15th Inter. FLAIRS'02 Conf.*, Florida, USA, 2002.

[21] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *In Proc. of the SIGMOD Inter. Conf. On Manag. Of Data*, Wisconsin, USA, 2002.

[22] A.K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21:101–112, 1984.

[23] S. Weibel, J. Miller, and R. Daniel. Dublin Core. In *OCLC/NCSA metadata workshop report*, 1995.

[24] N. Wirth. Type extensions. *ACM Trans. on Progr. Languages and Systems*, 10(2):204–214, 1988.

[25] F. Yergeau. Utf-8, a transformation format of iso 10646, 1998. Available at utf8.com.

[26] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the SIGMOD Inter. Conf. On Manag. Of Data*, 2001.

Appendix: NCA SQL Query for DAGs

A DPInterval

```
select  v.post
from    ( (select  s.post
           from    PInterval s
           where   (s.index <= p) and (s.post > p))
        union all
          (select  c.ancestor
           from    (select  t.post
                    from    PInterval t
                    where   (t.index <= p) and (t.post >= p)) u,
                   DPInterval c
           where u.post = c.post)) v,
        ( (select  r.post
           from    PInterval r
           where   (r.index <= p') and (r.post > p'))
        union all
          (select  d.ancestor
           from    (select  z.post
                    from    PInterval z
                    where   (z.index <= p') and (z.post >= p')) k,
                   DPInterval d
           where   k.post = d.post)) w
where   v.post = w.post and
        not exists (select  v'.post
                    from    ( (select  s'.post
                               from    PInterval s'
                               where   (s'.index <= p) and (s'.post > p))
                            union all
                              (select  c'.ancestor
                               from    (select  t'.post
                                        from    PInterval t'
                                        where   (t'.index <= p) and (t'.post >= p)) u',
                                       DPInterval c'
                               where   u'.post = c'.post)) v',
                            ( (select  r'.post
                               from    PInterval r'
                               where   (r'.index <= p') and (r'.post > p'))
                            union all
                              (select  d'.ancestor
                               from    (select  z'.post
                                        from    PInterval z'
                                        where   (z'.index <= p') and (z'.post >= p')) k',
                                       DPInterval d'
                               where   k'.post = d'.post)) w'
                    where   v'.post = w'.post and (
                                ((v'.post < v.post) and (select  (g.index <= v'.post)
                                                         from    PInterval g
                                                         where   g.post = v.post))
                              or
                                (select  TRUE
                                 from    DPInterval h
                                 where   h.ancestor = v.post and
                                         (select  (h.post >= v'.post and h.index <= f.index)
                                          from    PInterval f
                                          where   f.post = v'.post))))
```

B DUPrefix

```
Algorithm NCA($l_1$, $l_2$)
(1) List $Labels_1 = \{\}$, $Labels_2 = \{\}$, $Labels = \{\}$, $Ncas = \{\}$
(2) $treeNca$ = ncaTree($l_1, l_2$)
(3) $Labels_1$ += (select $label$
                  from DUPrefix
                  where $parent\_label = l_1$)
(4) $Labels_2$ += (select $label$
                  from DUPrefix
                  where $parent\_label = l_2$)
(5) $Labels$ += $Labels_1 \cap Labels_2$
(6) if ($\exists l \in Labels$ : prefix($treeNca$, $l$))
      $Ncas$ += $Labels$
    else if ($\exists l \in Labels$ : prefix($l$, $treeNca$))
      $Ncas$ += $Labels$
      $Ncas$ -= $\{l\}$
      $Ncas$ += $\{treeNca\}$
    else
      $Ncas$ += $Labels$
      $Ncas$ += $\{treeNca\}$
(7) return $Ncas$
```