The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations using Efficient Arithmetic Operations

Felix Weigel   Klaus U. Schulz
Centre for Information and Language Processing
University of Munich (LMU), Germany
Oettingenstraße 67, D-80538 Munich
{weigel,schulz}@cis.uni-muenchen.de

Holger Meuss
European Southern Observatory (ESO)
Headquarter Garching, Germany
Karl-Schwarzschild-Straße 2, D-85748 Garching
hmeuss@eso.org

ABSTRACT

We introduce a family of numbering schemes for the nodes of tree databases that are based on a structural summary for the database, such as the DataGuide. Using such a scheme, given the node IDs of two database nodes and the corresponding nodes in the structural summary we may *decide* the extended XPath relations *Child*, $Child^+$, $Child^*$, *Following*, *NextSibling*, $NextSibling^+$, $NextSibling^*$ for the nodes without access to the database. Similarly we can *reconstruct* the parent node and neighboured siblings of a given node. All decision and reconstruction steps are based on simple arithmetic operations. The BIRD scheme offers high expressivity and needs modest storage capacities. Compared to other identification schemes with similar expressivity, BIRD performs best in terms of both storage consumption and execution time for decision and reconstruction. A very attractive feature of the BIRD scheme is that all extended XPath relations can be decided and reconstructed in constant time, i.e. independent of tree position and distance of the nodes involved.

1. INTRODUCTION

Tree databases are important for many reasons. Since trees provide a formal model for XML, HTML, and LDAP directories, query formalisms for tree databases help to process data on the web, to extract and integrate data from distinct repositories and sites [13], to organize the exchange of commercial and scientific data, and to access user-specified corporate resources. Query formalisms for XML, representing a combination of information retrieval and database techniques, are paramount in the future development of search engines for the web and for digital libraries. Further applications arise in the field of computational linguistics, where tree databases are used for representing parsed fragments of natural language [30].

In the meantime, an impressive number of query formalisms for tree databases and XML have been proposed [6, 19, 5, 1, 11, 3, 4, 31, 20] and many systems have been developed that offer distinct functionalities for querying trees and XML [23, 16, 29, 26]. In most of these cases, the underlying evaluation algorithms use a characteristic set of fundamental operational steps that may be described as "decision" or "reconstruction" of tree relations:

- *Decision.* Given two database nodes and a binary tree relation, decide if the relation holds between the nodes.
- *Reconstruction.* Given a database node and a functional[1] tree relation, R, compute the R-image of the node.

Unlike decision, where potential ancestors, siblings etc. to be checked are already known, reconstruction starts from a given node and reproduces those nodes in its tree neighbourhood having a specific relation to that node. Standard relations for describing unranked ordered trees are the "generalized XPath axes" [5, 14]: *Child*, *NextSibling*, their inverses *Parent*, *PreviousSibling*, the (reflexive-) transitive closures of these relations, as well as *Following* and its inverse. The relations *Parent*, *NextSibling* and *PreviousSibling* are functional. Examples and details that explain the use of decision and reconstruction operations for generalized XPath axes in query evaluation are given in Section 2.

Since most of the above query formalisms and systems have to deal with large data sets, efficiency of the underlying evaluation algorithms is a central concern. We focus on the question how special conventions for assigning unique identifiers to the nodes of a tree database (also called *node identification* or *numbering schemes*) can help to solve decision and reconstruction problems efficiently for the above generalized XPath axes without access to the tree database, thus avoiding I/O-operations. Node identification schemes are largely complementary to other optimization techniques for tree queries such as special-purpose index structures and join algorithms. Hence the latter can benefit from intelligent identification schemes. The most basic numbering scheme for tree data, which assigns IDs in ascending preorder, clearly does not meet this end. For judging the quality of a naming scheme, three properties are essential:

- *Expressivity.* Which decision and reconstruction problems are supported by the scheme in the sense that explicit access to the database can be avoided, given node identifiers?
- *Runtime performance.* How long does it take to solve the decision and reconstruction problems that are sup-

[1] A binary tree relation R is *functional* iff for every node n there exists at most one node m such that $R(n, m)$ holds.

Dagstuhl Seminar Proceedings 05061
Foundations of Semistructured Data
http://drops.dagstuhl.de/opus/volltexte/2005/229

ported? Are there any dependencies on properties of the nodes involved, e.g., their depth or distance?

- *Storage consumption.* Which storage capabilities are needed for realizing the identification scheme, given a large tree database?
- *Robustness.* Is is possible to add new nodes to the database without spoiling the IDs already assigned to the existing parts of the documents?

Main Contribution. In this paper we suggest a new node identification scheme for tree databases. Node identifiers are integers, called *BIRD numbers* (Balanced Index-based numbering scheme for Reconstruction and Decision). BIRD numbering is compatible with preorder enumeration in the sense that nodes that come later in the preorder traversal have larger BIRD numbers, but not all possible numbers are used in the BIRD scheme. In addition, each node has a *weight*. Deciding (reconstructing) tree relations boils down to trivial arithmetic tests (calculations) based on BIRD numbers and weights.

As an illustration, consider the tree shown in Figure 1. Each node n is annotated with its BIRD number $Id(n)$ (in bold) and with its weight $w(n)$ (in brackets). For any descendant n' of a node n we have $Id(n) < Id(n') < Id(n) + w(n)$. *Decision* problems for any XPath axis can be solved based on the following observations: node n' is a descendant of a given node n iff $Id(n') - (Id(n') \bmod w(n)) = Id(n)$.[2] To check if node n' is a following sibling of n we test if $Id(n) < Id(n')$ and n, n' have the same father (fathers are reconstructed, s.b.). Node n' follows n (in the sense of XPath's `following` relation) iff $Id(n') \geq Id(n)+w(n')$. Furthermore, once we know the weight b of the unknown father of a given node n, then we can *reconstruct* the BIRD number of the father, which is $Id(n) - (Id(n) \bmod b)$. This briefly indicates how BIRD identification may be used to decide generalized XPath axes for two given nodes and to partially reconstruct tree neighbourhood (here: parent). Reconstruction along other functional axes is discussed below.

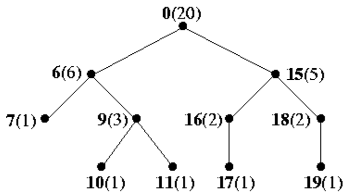

Figure 1: BIRD numbering: BIRD numbers and weights (in brackets).

We shall see that the BIRD scheme supports decision (reconstruction) of *all* (functional) generalized XPath axes. The triviality of the above arithmetic operations shows that high efficiency can be guaranteed if we have fast access to weights. To this end, BIRD weights are stored in a tree-formed structural summary or index (e.g., the DataGuide [12] of the database) that is held in main memory. Matters of storage requirements are considered by introducing various variants of BIRD numbering schemes that offer distinct compromises between expressivity of the scheme and the size of the resulting BIRD numbers. This size is also influenced by the choice of the structural summary. In this sense, BIRD identification defines a family of possible schemes. Evaluation results (see Section 9) show that BIRD outperforms other node identification schemes for tree databases: using BIRD, basic decision and reconstruction steps are solved faster than with other schemes.

[2]For integers k, l ($l \neq 0$), let $k \bmod l$ denote the unique number $m \equiv k$ modulo l s.th. $0 \leq m < l$.

Further Contributions

- We provide an *abstract view on query formalisms for XML and tree databases* that helps to explain the role of decision and reconstruction operations for tree relations. We show how to place existing systems and techniques in this picture.
- We review various node identification schemes for tree databases known from the literature and *classify* these schemes in terms of the decision and reconstruction steps that are supported, looking at a list of important relations on trees.
- We present the results of an *extensive evaluation experiment*, where various node identification schemes have been applied to four XML databases ranging from small (1.3 MByte) to big (8.4 GByte) size. The computation time with BIRD is almost always faster than that of other schemes, up to two orders of magnitude. A thorough profiling of all test runs quantifies the impact of individual evaluation stages on the overall performance of all schemes.

The structure of the paper is as follows. Section 2 briefly explains the use of decision and reconstruction steps in query formalisms for tree databases and XML. Section 3 provides some formal background. Section 4 introduces the family of BIRD numbering schemes. In Section 5 we show that BIRD numbering supports reconstruction of all functional generalized XPath axes. In Section 6 we show that BIRD numbering supports decision of all generalized XPath axes mentioned above. Section 7 reviews and analyzes other identification schemes for tree databases suggested in the literature and compares their expressivity. Section 9 describes the experimental evaluation of selected schemes, both in terms of their efficiency in decision and reconstruction and their storage consumption.

In this paper, a considerable number of binary relations on trees are considered. For definitions and notational conventions we refer to Section 3.

2. MOTIVATING DECISION AND RECONSTRUCTION

Looking at the core functionalities and abstracting from specific details, queries against tree databases and XML typically are built using unary predicates (e.g., labeling conditions, name tests in XPath) and binary tree relations. Query plans for evaluating such queries cover a spectrum of strategies with the following two extreme positions:

1. We may use the unary conditions to fetch a set of candidate image nodes for every single query node. In a second step, pairs of candidates from distinct sets are combined using joins, which amounts to solving a decision problem for the respective generalized XPath axis.
2. Since candidate sets for unselective unary predicates may be very large, we may alternatively fetch only the candidate sets for highly restricted query nodes (e.g., query leaves with selective keywords). From these nodes, candidates for other query nodes are obtained via reconstruction.

Reconstruction steps are particularly interesting along binary relations R that are functional (*Parent*, *PreviousSibling*, *NextSibling*, *i-th-Child* for $i \geq 1$, or any composition of these relations) or selective in the sense that database nodes typically have a small set of possible R-successors (transitive or reflexive-transitive closures of *Parent*, *PreviousSibling*, *NextSibling*). Given a query containing a condition $R(x, y)$ for such a relation R, if we already have a small candidate set for x, then reconstruction along R efficiently computes all relevant candidates for y. By contrast, if the unary conditions for y is weak, obtaining a consistent candidate set for x and y via decision might be costly.

Different query plans which are more or less close to either of the above positions are explained in [24]. The evaluation strategies described in [22, 34, 15, 2, 8] follow the first paradigm, whereas [7, 26, 29] adhere to the second paradigm. In either case, the use of an appropriate node identification scheme is a reliable means to improve on query performance. As shown in Sections 7 and 9, the identification schemes proposed in the literature differ in how many and which relations can be decided and reconstructed based on node IDs only, and also in how fast this can be done under specific circumstances. The following sections introduce the BIRD scheme as a new numbering scheme with efficient support for deciding and reconstructing all extended XPath axes.

3. FORMAL BACKGROUND

Let Σ denote a finite alphabet, called the *alphabet of labels*.

DEFINITION 3.1. *A* database *is a finite ordered rooted tree* $DB = \langle N, n_r, Child, NextSibling, L\rangle$ *where* N *is the finite and non-empty set of nodes, Child is a binary relation on* N *such that* $\langle N, Child, n_r\rangle$ *is an unordered rooted tree with root* n_r*, NextSibling* $\subseteq N \times N$ *relates a child with its immediate right sibling in the obvious way, and* $L : N \to \Sigma$ *assigns a label* $L(n) \in \Sigma$ *to each node* $n \in N$*.*

Besides *Child* and *NextSibling*, a considerable number of further tree relations will be touched: $Parent = Child^{-1}$, $PreviousSibling = NextSibling^{-1}$, *NextInDocOrder* (relating a node to the next node in a pre-left traversal), the (reflexive and) transitive closures of the above relations, *Following*, for $i \geq 1$ the proximity relations $Parent^i$ (= `ancestor::*[i]` in XPath notation), $Child^i$, $NextSibling^i$, $PreviousSibling^i$, as well as *i-th-Following* and *i-th-Child* (=`following::*[i]` and `child::*[i]`, respectively). All relations are defined as usual. The functional relations *Parent*, *PreviousSibling*, and *NextSibling* define functions *parent*, *prevSibling*, and *nextSibling* in a canonic way.

DEFINITION 3.2. *Let DB be a database with set of nodes* N*. A* structural summary *for DB is a finite (not necessarily ordered) rooted tree Ind with set of nodes* M*, together with a surjective mapping* $\Phi : N \to M$ *that preserves roots and Child-relationship in the obvious sense.* Φ *is called the* index mapping*. For* $m \in M$*, the set* $\Phi^{-1}(m)$ *is called the* set of database nodes with index node m.

A structural summary can be considered as a special kind of index structure. In what follows, by an index, we always mean a structural summary. The *DataGuide* [12] (or *1-Index* [27], being equivalent to the DataGuide for tree databases) will serve as our standard example of a structural summary. Note, however, that BIRD may well be used with other index structures (see also the final remark in Section 9.1). To introduce the DataGuide, the following notions are needed.

DEFINITION 3.3. *Let* $DB = \langle N, Child, NextSibling, L, n_r\rangle$ *be a database. A string* $\pi \in \Sigma^+$ *is called a* label path *of DB iff there exists a sequence of nodes* $n_0, n_1, \ldots, n_k \in N$ *(*$k \geq 0$*) such that* $n_0 = n_r$*,* $\langle n_i, n_{i+1}\rangle \in Child$ *for* $0 \leq i < k$ *and* $\pi = L(n_0)L(n_1)\cdots L(n_k)$*. In this situation,* π *is called the* label path of n_k*, we write* $\pi = lp(n_k)$*. The* length *of* π *is* k*. A label path* π *of DB is* maximal *iff* π *is not a proper prefix of any label path* ϱ *of DB.*

Note that each label path π is non-empty and starts with $L(n_r)$. $LP(DB)$ denotes the set of all label paths of the database *DB*.

DEFINITION 3.4. *The* height *of a database DB is the maximal length of a label path of DB.*

DEFINITION 3.5. *Let* $DB = \langle N, Child, NextSibling, L, n_r\rangle$ *be a database with root* n_r*. The* DataGuide *of DB is the finite rooted unordered node-labeled tree DG(DB) with set of nodes* $LP(DB)$ $L(n_r)$ *is the root of DG(DB),* $\varrho \in LP(DB)$ *is a child of* $\pi \in LP(DB)$ *iff there exists a label* $l \in \Sigma$ *such that* $\varrho = \pi l$*, and the label of* $\pi \in LP(DB)$ *is the last symbol of* π*.*

It is easy to see that for each database *DB* there exists exactly one DataGuide *DG(DB)* of the above form. Obviously, *DG(DB)* represents a structural summary for *DB* with index mapping *lp*.

EXAMPLE 3.6. *Figure 2 shows a database DB and its DataGuide DG(DB). Nodes of DB and DG(DB) are labeled with numeric information for the child-balanced numering scheme that is introduced below.*

DEFINITION 3.7. *Let* $DB = \langle N, Child, NextSibling, L, n_r\rangle$ *be a database. A function* $f : N \to I\!N$ *is* compatible *with the preorder* $<_{pre}$ *on DB iff* $m <_{pre} n$ *implies that* $f(m) < f(n)$*, for all* $m, n \in N$*.*

4. THE FAMILY OF BIRD NUMBERING SCHEMES

BIRD numbering schemes for the nodes of a database *DB* as introduced below are always compatible with the preorder relation on the database in the sense of Definition 3.7. When enumerating the nodes, for each node $n \in N$ of the database we will need a certain interval size, or *weight*, to number all nodes in the subtree with root n. Our numbering schemes are based on a structural summary *Ind* of *DB* with index mapping Φ. We unify all interval sizes needed for database nodes with the same index node m, selecting the maximal interval size among all members of the equivalence class $\Phi^{-1}(m)$. This unified interval size is attached to the associated node m of the structural summary. When enumerating the nodes of the database, we reserve this interval size for all subtrees rooted at any of the nodes in $\Phi^{-1}(m)$. Since in general not all these subtrees are of the same size, some numbers remain unused in the enumeration.[3]

Because of obvious space limitations, we only consider "balanced" variants of the BIRD scheme. Here the weights

[3] Unused numbers may also be reserved deliberately for future node insertions into the database.

for index nodes are unified among all children (or grandchildren, etc.) of a given index node. There also exists an unbalanced variant, which yields the smallest weights and node numbers, but is less expressive. In our experiments we found the the storage requirements for balanced variants are modest, hence larger numbers obtained from balanced schemes are tolerable.

4.1 Balanced weights of index nodes

Let n denote a node of a tree with root n_r, let $s \geq 1$. By the *s-step ancestor* of n, we mean the ancestor of n that is reached in exactly s parent steps. As a matter of fact, the s-step ancestor of n is defined if and only if n is a node in depth $s' \geq s$, using the standard notion of the depth of a node in a tree. Since balanced weights are based on maximal interval sizes of siblings, cousins, grand-cousins, etc. in a tree, we need the following definition of s-equivalent nodes. Basically, two nodes are 1-equivalent, iff they are siblings, 2-equivalent, iff they are siblings or cousins (i.e. share the same grandparent) etc.

DEFINITION 4.1. *The equivalence relations $\sim_s$ ($s \geq 1$) on the set of nodes N of a given tree are inductively defined as follows:*

1. *for all $n, n' \in N$: $n \sim_1 n'$ iff the 1-step ancestors (i.e. parents) of n and n' are defined and coincide.*
2. *Let $s \geq 1$. For all $n, n' \in N$: $n \sim_{s+1} n'$ iff $n \sim_s n'$, or the $s+1$-step ancestors of n and n' are defined and coincide.*

If $n \sim_s n'$, we say that n and n' are s-equivalent. *By $[n]_s$ we mean the equivalence class of node n w.r.t. $\sim_s$.*

DEFINITION 4.2. *Let $DB = \langle N, Child, NextSibling, L, n_r\rangle$ be a database, let $n \in N$. Let $n_1, \ldots, n_k$ ($k \geq 0$) denote the sequence of all children of n in the canonical left-to-right ordering as specified by the NextSibling relation. Let Ind be a structural summary for DB with index mapping Φ. The* index node sequence of the children of n *is the sequence $insc(n) := \Phi(n_1)\cdots\Phi(n_k)$. Let m denote a node of Ind. The* set of index node sequences *associated with m is $INS(m) := \{insc(n) \mid n \in \Phi^{-1}(m)\}$.*

DEFINITION 4.3. *Let DB denote a database, let Ind denote a structural summary for DB. Let $s \geq 1$. The* s-balanced pre-weight $w'_s(m)$ *and the* s-balanced weight $w_s(m)$ *of an index node m are recursively defined in a bottom-up manner as follows:*

$$w'_s(m) \;:=\; \begin{cases} w_s(m_1)\cdot max\{|\chi|+1 \mid \chi \in INS(m)\} \\ \quad \textit{iff } m \textit{ has any child } m_1, \\ 1 \textit{ otherwise,} \end{cases}$$

$$w_s(m) \;:=\; max\{w'_s(m')) \mid m \sim_s m'\}.$$

Here $|\chi|$ denotes the length (number of elements) of the sequence χ.

The fact, that weights of s-equivalent nodes are equal due to the maximum operation over the pre-weights yields the name of *balanced* indexing schemes.
This guarantees also the well-definedness of pre-weights $w'_s(m)$, since two children m_1 and m_2 of m have the same s-balanced weights $w_s(m_1) = w_s(m_2)$.
1-balanced weights are also called *child-balanced* weights. If $s = h$ denotes the height of the database DB, then $w_s(m)$ is called the *totally balanced weight* of the index node m.

EXAMPLE 4.4. *Consider the database DB with the summary DG(DB) shown in Figure 2 (a) and (b), respectively. Each node m of the DataGuide (Figure 2 (b)) is annotated with its child-balanced weight $w_1(m)$ and, for convenience, with its child-balanced pre-weight $w'_1(m)$ (numbers in brackets). We also depict all index node sequences for m (rectangles) and the number $max\{|\chi|+1 \mid \chi \in INS(m)\}$ (right side of rectangles). To simplify notation we only write the last symbol of each label path in an index node sequence.*

We will now show for the left-most path (racbc) in the DataGuide (Figure 2 (b)), how the depicted pre-weights and weights are computed. The procedure runs bottom-up and begins with leafs racbc and racbb, which have pre-weights $w'_1(racbc) = w'_1(racbb) = 1$ since they have no children. The maximum pre-weights w'_1 among the two siblings is 1, therefore $w_1(racbc) = w_1(racbb) = 1$.
Now we go up one step and compute the pre-weight of index node racb, which is associated via Φ^{-1} with database nodes 111, 114, and 282 (big numbers[4] in Figure 2 (a)). 111 and 282 have no children, but for 114 we can compute $insc(114)$ = racbc racbb (abbreviated as c for racbc and b for racbb in the bottom left rectangle). Therefore $INS(racb)$ = {racbc racbb}. The maximum length of sequences in $INS(racb)$ is 2, increased by 1 yields 3 (next to the bottom left rectangle). The children of racb have weight 1, therefore the pre-weight $w'_1(racb) = 3$. Now the weight $w_1(racb)$ is computed: The bottom-up algorithm has already computed the pre-weights for the siblings racc and racd, which is 1 for leaves. The weight of each of the three siblings racb, racc, and racd is the maximum of their pre-weights, i.e. 3. Note that, due to this maximum operation, the weight of racc and racd has been increased to 3 compared to their pre-weight of 1.
In the next level, we first compute the pre-weight for rac, which is associated with database nodes 90, 105, 135, 152, 270, 330, 345, 380. They have two distinct index node sequences of children, namely racc racb racb for 105 and for 285 racd racc racd racb. The maximal length of these two sequences is 4, increased by 1 results in the 5 next to the middle left rectangle. This value is now multiplied with the weight $w_1(racb) = 3$ of the children of rac, resulting in a pre-weight value $w'_1(rac) = 15$ for rac. Node rac has two siblings, both with pre-weight 1, therefore $w_1(rac) = 15$.
In the following levels, pre-weights and weights are computed in exactly the same way, until we reach the root with weight $w(r) = 450$.

In the remainder of the paper, Ind_{w_s} denotes the variant of the structural summary *Ind* for the database *DB* where each index node m is labeled with its s-balanced weight $w_s(m)$, as illustrated in Figure 2 *(b)* for the DataGuide.

4.2 Balanced enumeration of database nodes

We now describe the *s-balanced numbering scheme*, which assigns an integer $Id_s(n)$ to each node n of *DB*, given the annotated index Ind_{w_s}. In the special case where $s = h$ represents the height of the database, the scheme is called the *totally balanced numbering scheme.*

DEFINITION 4.5. *Let $s \geq 1$. The number $Id_s(n_r)$ for the root n_r is any multiple of $w_s(\Phi(n_r))$. Let n denote an ar-*

[4] For convenience, we use these numbers in this explanation, although the procedure how they are computed is explained later, in the next section.

bitrary node of DB. Let $n_1, \ldots, n_k$ ($k \geq 1$) denote the sequence of all children of n in the canonical left-to-right ordering. Given the number $\mathrm{Id}_s(n)$ for the parent node n and the balanced weight $w = w_s(\Phi(n_1)) = \ldots = w_s(\Phi(n_k))$, the number $\mathrm{Id}_s(n_1)$ for the first child n_1 is the smallest multiple of w larger than $\mathrm{Id}_s(n)$. The number for the i-th child n_i for $2 \leq i \leq k$ is $\mathrm{Id}_s(n_i) := \mathrm{Id}_s(n_1) + (i-1) \cdot w$.

EXAMPLE 4.6. *In Figure 2 (a), each database node n is annotated with $\mathrm{Id}_1(n)$ (large number). The enumeration started with 0 for the root node, and went top-down through the tree in manner described above. Note that weights for index nodes and identifiers of database node are defined in a way that all node identifiers in the subtree of a node n are guaranteed to fall into the interval $[Id_s(n), Id_s(n){+}w_s(lp(n))[$ where $\mathrm{lp}(n)$ is exactly $\Phi(n)$. This important relation between weights of index nodes and database node identifiers is established in Lemma 4.9. The right border of the interval of each node is denoted with the small numbers in brackets in Figure 2 (a).*

EXAMPLE 4.7. *In Figure 3 (a), each database node n is annotated with its totally balanced enumeration number $\mathrm{Id}_4(n)$ (large number) and with $\mathrm{Id}_4(n) + w_4(\mathrm{lp}(n))$ (small number in brackets).*

The following two lemmas show that index node weights define intervals for the identifiers of nodes and their subtrees. This lemma is important, since it guarantees that identifiers for nodes are indeed unique. In addition, it shows that the function *Id* is compatible with the preorder $<_{pre}$ in the sense of Definition 3.7.

LEMMA 4.8. *Let $s \geq 1$. Let n be a node of DB, let $n_1, \ldots, n_k$ denote the sequence of all children of n in the canonical left-to-right ordering. Let $w := w_s(\Phi(n_1)) = \ldots = w_s(\Phi(n_k))$. Then we have*

$$\begin{aligned} \mathrm{Id}_s(n) &< \mathrm{Id}_s(n_1) < \ldots < \mathrm{Id}_s(n_k) \\ &< \mathrm{Id}_s(n_k) + w \leq \mathrm{Id}_s(n) + w_s(\Phi(n)). \end{aligned}$$

LEMMA 4.9. *Let $s \geq 1$. Let DB be a database with set of nodes N and root n_r. Let Ind be a structural summary for DB with index mapping Φ. Regardless of the initial assignment of $\mathrm{Id}_s(n_r)$,*

1. *for all $n \in N$: $\mathrm{Id}_s(n) \equiv 0 \mod w_s(\Phi(n))$,*
2. *the mapping Id_s is injective and compatible with the preorder.*

Proof. 1) follows immediately from Definition 4.5, and 2) from Lemma 4.8. □

The following lemma shows how the growth of node IDs is limited by the height and branching degree of the tree:

LEMMA 4.10. *Let $s \geq 1$. Let DB be a database with height h, maximal branching degree b, set of nodes N and root n_r. Assume that we assign to n_r the value $\mathrm{Id}_s(n_r) := 0$. Then $\mathrm{Id}_s(n) \leq (b+1)^h$ for all $n \in N$.*

Proof. Let m be an index node. Let $d(m)$ denote the depth of m in the index tree, let $h(m) := h - d(m)$. Starting from leaves of the index tree, a simple induction shows that $w_s(m) \leq (b+1)^{h(m)}$. We have $w_s(\Phi(n_r)) \leq (b+1)^h$. The result follows from Lemmata 4.8 and 4.9. □

5. RECONSTRUCTING THE TREE STRUCTURE

We discuss how parts of the tree structure of the database can be reconstructed without accessing the database, given the number of a node and the corresponding index node with its weight. In what follows, *DB* denotes a database, *Ind* denotes a structural summary for *DB* with index mapping Φ.

LEMMA 5.1. *[Parent and ancestor reconstruction] Let $s, i \geq 1$. Assume that for some database node n we are given its number $\mathrm{Id}_s(n)$ and the index node $m := \Phi(n)$. Then, using Ind_{w_s} we may solve the following tasks without access to DB: Decide if there exists an ancestor n' of n that is reached from n with exactly (at least) i parent steps. In the affirmative case, compute the number $\mathrm{Id}_s(n')$ and the index node $m' := \Phi(n')$ corresponding to n'.*

Proof. Obviously, n has an ancestor n' that can be reached with exactly i parent steps iff $m := \Phi(n)$ has such an ancestor, m'. Using Ind_{w_s} we may decide this question, finding m' in the affirmative case. By Lemma 4.9, $Id_s(n')$ is a multiple of $w_s(m')$. It follows from Lemma 4.8 that $Id_s(n')$ is the greatest multiple of $w_s(m')$ smaller than $Id_s(n)$. □

LEMMA 5.2. *[Reconstruction of i-th child] Let $s, i \geq 1$. Assume that for some database node n we are given its number $\mathrm{Id}_s(n)$ and the index node $m := \Phi(n)$. Then, using Ind_{w_s} we may compute the number $\mathrm{Id}_s(n_i)$ of the i-th child n_i of n, assuming that this child exists, without access to DB.*

Proof. Using Ind_{w_s} we fetch the weight $w = w_s(m')$ of the children m' of m. By definition, $Id_s(n_1)$ is the smallest multiple of w larger than $Id_s(n)$, and for $i > 1$ we have $Id_s(n_i) = Id_s(n_1) + w(i-1)$. □

In general, we cannot directly compute the index node $\Phi(n_i)$ corresponding to the i-th child n_i, unless we have further information (when using the DataGuide we need the label). Note, however, that we know the weight of $\Phi(n_i)$ since the scheme is child-balanced.

LEMMA 5.3. *[Reconstruction of i-th left sibling] Let $s, i \geq 1$. Assume that for some database node n we are given its number $\mathrm{Id}_s(n)$ and $m = \Phi(n)$. Then, using Ind_{w_s} we may solve the following task without access to DB: Decide if n has at exactly (at least) i siblings that precede n in the left-to-right ordering. If n has at least i preceding siblings, compute the number $\mathrm{Id}_s(n_i)$ of the i-th preceding sibling n_i of n.*

Proof. We may assume that n has a parent node n'. Let $Id_s(n')$ denote its number, calculated as described in Lemma 5.1. Let $w = w_s(m)$. By Lemma 4.8, n has at least i preceding siblings iff $Id_s(n') < Id_s(n) - i \cdot w$. From Definition 4.5 it follows that n has exactly i preceding siblings iff $Id_s(n) - (i+1) \cdot w \leq Id_s(n') < Id_s(n) - i \cdot w$. If the i-th preceding sibling exists, it has the number $Id_s(n) - i \cdot w$. □

Similarly as for the i-th child, we cannot directly compute the index node corresponding to the i-th left sibling n_i, unless we have further information. Nodes n_i and n have the same weight.

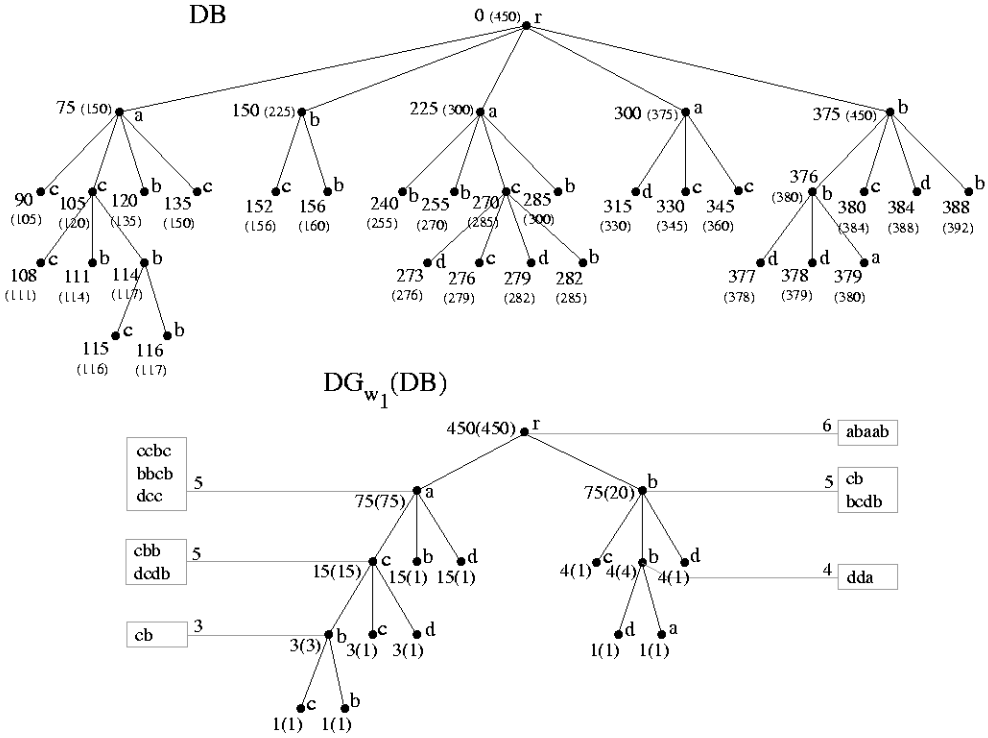

Figure 2: Child-balanced numbering scheme. *(a)* Database. *(b)* DataGuide for *(a)*. For each node π of *DG(DB)*, the number $max\{|\chi| + 1 \mid \chi \in \mathit{INS}(\bar{\pi})\}$ is indicated, π is annotated with its child-balanced weights, cf. Examples 4.4 and 4.6.

LEMMA 5.4. *[Reconstruction of i-th right sibling] Let $s, i \geq 1$. Assume that for some database node n we are given its number $Id_s(n)$ and the index node $m := \Phi(n)$. Then, using Ind_{w_s} we may compute the number $Id_s(n_i)$ of the i-th right sibling n_i of n, assuming that this sibling exists, without access to DB.*

An attractive feature of the totally balanced scheme is the following.

LEMMA 5.5. *Let DB be a database of height h. Let m' be a child of the index node m. Then, $w_h(m)$ is a multiple of $w_h(m')$. Given the number $Id_h(n)$ for the parent database node n with children $n_1, \ldots, n_k$ (in left-to-right ordering) and the balanced weight $w = w_h(\Phi(n_1)) = \ldots = w_h(\Phi(n_k))$, we have $Id_h(n_i) = Id_h(n) + i \cdot w$.*

Proof. The first statement is a simple consequence of the fact that all index nodes with the same depth in the index tree are assigned the same weight by w_h. By Definition 4.3, each pre-balanced weight w'_h on the parent level is a multiple of this weight. Hence the same holds for the maximum, which yields the weight for the parent level. The second statement follows easily. □

REMARK 5.6. *The same is not always true if $k < h$. Figure 2 illustrates this for $k = 1$ and $h = 4$.*

REMARK 5.7. *[Reconstruction of descendants] A simple consequence of Lemma 5.5 is the following. Given a node n with number $Id_h(n)$, the node number $Id_h(n')$ of any descendant n' of n, specified in the form "i_m-th child of the … of the i_1-th child of n", can be computed without access to the database, using totally balanced weights stored in DG_{w_h}(DB). Note, however, that in general we cannot guarantee the existence of this node without accessing DB.*

REMARK 5.8. *[Reconstruction of arbitrary weights] When using the totally balanced numbering scheme, from the number $Id_h(n)$ of a database node n we can reconstruct the weight $w_h(\Phi(n))$, given the list of the uniform weights of all levels of the index tree. In fact $w_h(\Phi(n))$ is the largest weight w stored in our list such that $Id_h(n) \equiv 0 \mod w$. (As a by-product, the depth of n is obtained this way.) Hence, Lemmata 5.1, 5.2, 5.3 and 6.2 can be refined in the sense that we do not need to know the index node m corresponding to n.*

REMARK 5.9. *The higher the balancing degree s, the fewer DataGuide nodes are needed for storing weights. For $s = h$, an h-tuple of weights suffices for tree reconstruction. In special cases, however, it might be convenient to store the weights redundantly in all nodes of the index. This is true, e.g., when using the DataGuide as weight index and as a path index during query evaluation.*

REMARK 5.10. *The results obtained for the totally balanced enumeration scheme are summarized in Figure 4 (a). Given the number $Id_h(n)$ of a database node n, we immediately know how many ancestors n' of n there are, and we can compute the numbers $Id_h(n')$ of all these ancestors without accessing the database. Furthermore we can deduce the number of preceding left siblings n'' for each of these nodes as well as their numbers $Id_h(n'')$. In the remaining regions of the tree (indicated by small dots) we know the number of each possible node; yet we cannot decide which numbers*

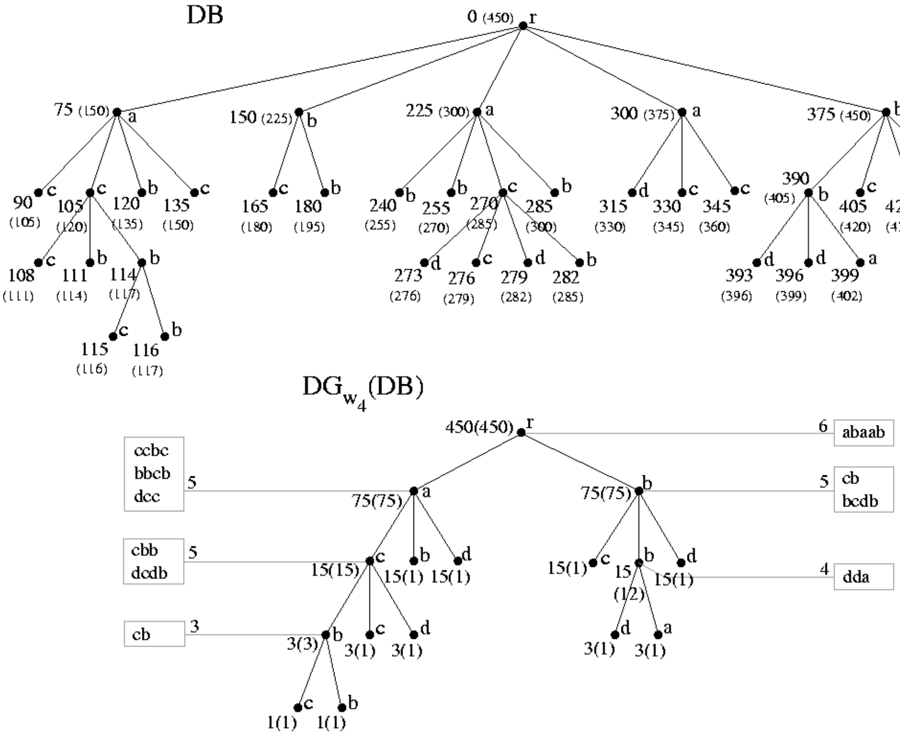

Figure 3: Totally balanced (here: 4-balanced) numbering scheme. *(a)* Database. *(b)* DataGuide for *(a)*. Nodes of *DG(DB)* are annotated with totally balanced weights, cf. Examples 4.7.

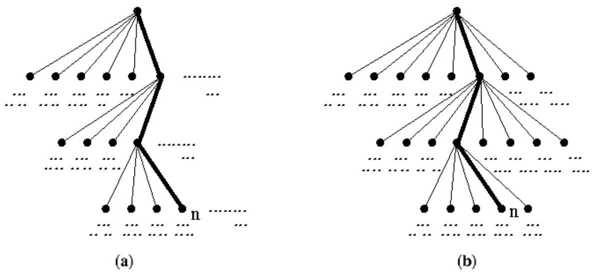

Figure 4: *(a)* The part of a database *DB* that can be reconstructed from a node n with number $Id_h(n)$ (totally balanced scheme) and Ind_{w_h} with totally balanced weights. *(b)* The corresponding part of *DB* that can be reconstructed using a double enumeration, cf. Remark 5.10.

correspond to existing nodes and which numbers are unassigned. We can generalize this picture, using a symmetric second enumeration based on the inverse postorder. *The inverse postorder behaves like a "right-to-left" preorder. Assigning to each node a pair of numbers, according to a preorder (→) and a "right-to-left preorder" (←) traversal of the tree, we can compute for each node $n \in N$ with a pair of numbers $\langle Id_h^{\rightarrow}(n), Id_h^{\leftarrow}(n)\rangle$ the number of left-hand and right-hand siblings as well as their respective number pairs.*

6. DECIDING GENERALIZED XPath RELATIONS

In this section we consider the generalized[5] XPath axes *Child*, *Child**, *Child*$^+$, *NextSibling*, *NextSibling**, *NextSibling*$^+$, and *Following*. If R is any of these relations and if *DB* is a database with nodes n and n', we write $DB \models R(n, n')$ iff the relation R holds in *DB* between n and n' (e.g., $DB \models Child(n, n')$ iff n' is a child of n). As always we fix a structural summary *Ind* with index mapping Φ.

The following lemma shows that using e.g. the child-balanced scheme, a superset of all XPath axes[6] is decidable without any I/O operation (see Table 1).

LEMMA 6.1. *[Deciding generalized XPath axes] Let $s \geq 1$. Assume we are given*

- *the number $Id_s(n)$ of the database node n,*
- *the index node $m = \Phi(n)$ corresponding to n,*
- *the number $Id_s(n')$ of a second database node n'.*

Let R be any of the following relations: Child, Child$^+$, Child*, NextSibling, NextSibling$^+$, NextSibling*, Following. Then, using Ind_{w_s} we may decide if $DB \models R(n, n')$ (or if $DB \models R(n', n)$) without access to the database DB.

Proof. See Table 1. □

For the sake of completeness, we mention some other problems that may be decided with similar methods. Proofs are simple.

[5] *NextSibling* and *NextSibling** are not supported in XPath but nevertheless considered here.
[6] Although we do not consider the `attribute` and `namespace` axes here, they can be treated similarly to the `child` axis.

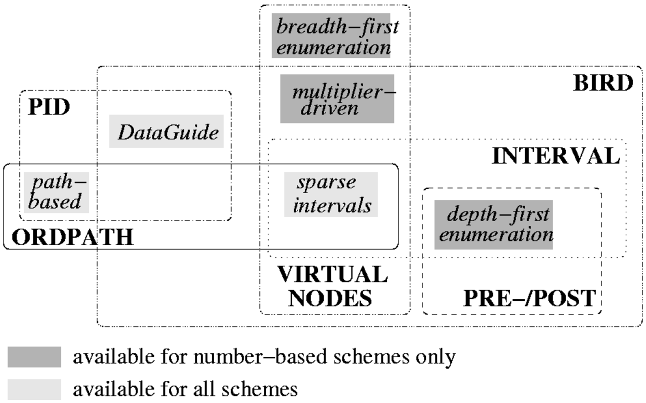

Figure 5: Characteristic properties of ID schemes.

LEMMA 6.2. *[Deciding proximity relations] Let $s, i \geq 1$. Assume we are given the number $Id_s(n)$ of the database node n, the index node $m = \Phi(n)$, and the number $Id_s(n')$ of a second node $n' \in N$. Then, using Ind_{w_s} we may decide the following questions without access to the database DB:*

1. *DB $\models$ Parent$^i(n, n')$?*
2. *DB $\models$ PreviousSibling$^i(n, n')$?*
3. *DB $\models$ NextSibling$^i(n, n')$?*

7. RELATED WORK: OTHER NODE IDENTIFICATION SCHEMES

In this section we first describe the characteristic properties that distinguish identification schemes known from the literature. We then analyze the schemes in more detail on the basis of these properties before systematically comparing their expressivity.

Characteristic properties. Node identification schemes used for decision and reconstruction and their characteristic properties are illustrated in Figure 5. *Number-based* schemes use atomic numbers to identify nodes, whereas *path-based* schemes employ non-atomic number sequences as node IDs. The latter essentially use a sequence of relative positions of nodes among their siblings to identify a node, and consequently have IDs of varying length. Many schemes use a *sparse ID set*, i.e., the ID space is not contiguous and contains unused IDs. Sparse ID sets have the advantage that they can cope with limited updates and node insertions without having to reassign node IDs. Number-based schemes use either *depth-first* or *breadth-first* tree traversal, the assignment of node IDs is *multiplier-driven* or not. Multiplier-driven schemes are sparse ID schemes that assign only multiples of a certain basic weight as node identifiers. Exploiting certain arithmetic relations, they can compute path decisions and reconstructions numerically. Finally, some ID schemes are *DataGuide-based*: they take advantage of extracting path-related information and storing it in a DataGuide or a similar structural summary, using it when necessary during path reconstruction or decision.

Other node identification schemes. Path-based node identification schemes such as *Dewey Order* [32] use the entire root path $\langle c_0, \ldots, c_k \rangle$ of a node at level k as node ID. Each *offset* c_i denotes the position of n's ancestor at level i among its siblings. This path encoding implies that node IDs have no fixed size and may vary according to the depth of a node and its position among its siblings. In [32], the individual offsets c_i are encoded in UTF-8 to reduce the overall ID size. Since the offsets are independent of each other, Dewey Order supports (limited) updates without altering all IDs assigned to other nodes. As shown in [32], renumbering is restricted to the descendants and following siblings of the node being inserted.

The Dewey Order-based *ORDPATH* scheme [28] uses skew binary encodings privileging smaller offsets, which occur much more often than greater ones in Dewey Order. Still ORDPATH consumed up to twice as much space as BIRD in our experiments. For path reconstruction, the encoded ORDPATH IDs are parsed and split into their offset components. Ascending in the document tree by one level is equivalent to removing the last offset component from the ORDPATH ID. All path relations are decided by bit-wise comparison of the encoded IDs. However, the IDs must be decoded into their offset components first in order to find the component boundaries in the bit string (except for deciding *NextInDocOrder*$^+$.) [28] describes an update mechanism for ORDPATH which reserves unused IDs for future insertions at any position in the document tree. By virtue of this sparse encoding ORDPATH is the only known scheme to allow for arbitrary updates without changing any existing ID. However, to achieve this robustness ORDPATH loses some of its expressivity, supporting neither decision of the *NextSibling* relation nor reconstruction of sibling or child nodes (see below).

Another path-based scheme, similar to ORDPATH (without updates), was proposed in [7]: binary *Path Identifiers (PIDs)* encode complete root paths as sequences of offsets among children with the same label (not all children as with the ORDPATH scheme). To save space, offsets for children which do not have any sibling with the same label are not encoded. Information as to which path steps are skipped this way is stored in a DataGuide [12]. In contrast to ORDPATHs, PIDs do not mark the boundaries of individual offset components in the bit string, but store the number of bits used to encode a given offset in the corresponding DataGuide node. Updates in PID are not supported, but only a local renaming of node IDs is necessary if new nodes are inserted. The PID scheme is the least expressive scheme among those supporting both path reconstruction and decision, but as our experiments showed, its node IDs are the smallest in size. Note that without reference to the corresponding DataGuide node, PIDs are not guaranteed to be unique. If the DataGuide node is given, they still follow neither document order (unlike BIRD and most other schemes) nor breadth-first order (as Virtual Nodes, see the next paragraph).

The *Virtual Nodes* scheme [21] is the only number-based scheme identifying nodes by their breadth-first rank. It uses a sparse ID set: the document tree is regarded as having a uniform arity k, i.e. all inner nodes are treated as having exactly k children. This means that many IDs are reserved for so-called *virtual nodes* which do not exist physically in the document tree, since many nodes have less than k children. The resulting sparse encoding leads to a significantly higher space consumption compared to other schemes (see Section 9). The advantage of assuming a uniform arity is that for path reconstruction and decision multiplier-driven

$DB \models \mathit{Child}(n, n')$ `child`	We check if m has any child, say, m', using Ind_{w_s}. In the negative case, n' is not a child of n. In the positive case let $w = w_s(m')$. Then $DB \models \mathit{Child}(n, n')$ iff $\mathit{Id}_s(n')$ is a multiple of w and $\mathit{Id}_s(n) < \mathit{Id}_s(n') < \mathit{Id}_s(n) + w_s(m)$. The numbers $w_s(m')$ and $w_s(m)$ are obtained from Ind_{w_s}.
$DB \models \mathit{Child}^+(n, n')$ `descendant`	We retrieve $w_s(m)$ using Ind_{w_s}. Then $DB \models \mathit{Child}^+(n, n')$ iff $\mathit{Id}_s(n) < \mathit{Id}_s(n') < \mathit{Id}_s(n) + w_s(m)$.
$DB \models \mathit{Child}^*(n, n')$ `descendant-or-self`	Obviously this is a variant of the previous decision problem.
$DB \models \mathit{Child}(n', n)$ `parent`	We proceed as in Lemma 5.1.
$DB \models \mathit{Child}^+(n', n)$ `ancestor`	We iterate the procedure described in Lemma 5.1 for $i = 1$ until reaching either n' (positive result) or a node n'' where $\mathit{Id}_s(n'') < \mathit{Id}_s(n')$ (negative result).
$DB \models \mathit{Child}^*(n', n)$ `ancestor-or-self`	Obviously this is a variant of the previous decision problem.
$DB \models \mathit{NextSibling}(n, n')$	We obtain $w_s(m)$ and m's parent m'' from Ind_{w_s} and compute the number $\mathit{Id}_s(n'')$ of the parent n'' of n in DB (cf. Lemma 5.1). $DB \models \mathit{NextSibling}(n, n')$ holds iff $\mathit{Id}_s(n') = \mathit{Id}_s(n) + w_s(m)$ and $\mathit{Id}_s(n') < \mathit{Id}_s(n'') + w_s(m'')$.
$DB \models \mathit{NextSibling}^+(n, n')$ `following-sibling`	We obtain $w_s(m)$, m'' and $\mathit{Id}_s(n'')$ as above (cf. $DB \models \mathit{NextSibling}(n, n')$). $DB \models \mathit{NextSibling}^+(n, n')$ holds iff $\mathit{Id}_s(n') - \mathit{Id}_s(n)$ is positive and a multiple of $w_s(m)$ and if $\mathit{Id}_s(n') < \mathit{Id}_s(n'') + w_s(m'')$.
$DB \models \mathit{NextSibling}^*(n, n')$	Obviously this is a variant of the previous decision problem.
$DB \models \mathit{NextSibling}(n', n)$	We proceed as in Lemma 5.3 ($l = 1$).
$DB \models \mathit{NextSibling}^+(n', n)$ `preceding-sibling`	We obtain $w_s(m)$ and m's parent m'' from Ind_{w_s} and compute the number $\mathit{Id}_s(n'')$ of the parent n'' of n in DB (cf. Lemma 5.1). $DB \models \mathit{NextSibling}^+(n', n)$ holds iff $\mathit{Id}_s(n) - \mathit{Id}_s(n')$ is positive and a multiple of $w_s(m)$ and if $\mathit{Id}_s(n'') < \mathit{Id}_s(n')$.
$DB \models \mathit{NextSibling}^*(n', n)$	Obviously this is a variant of the previous decision problem.
$DB \models \mathit{Following}(n, n')$ `following`	The relation holds iff $\mathit{Id}_s(n) + w_s(m) \leq \mathit{Id}_s(n')$, by Lemmata 4.9 and 4.8. The weight $w_s(m)$ is obtained from Ind_{w_s}.
$DB \models \mathit{Following}(n', n)$ `preceding`	The relation holds iff $\mathit{Id}_s(n') < \mathit{Id}_s(n)$ and n' is not an ancestor of n. The latter problem is decided as described above (cf. $DB \models \mathit{Child}^+(n', n)$/`ancestor`).

Table 1: Proof for Lemma 6.1. Relations decidable using any s-balanced BIRD scheme with $s > 0$. Given node numbers $\mathit{Id}_s(n)$ and $\mathit{Id}_s(n')$ as well as the index node $m = \Phi(n)$ holding the corresponding weight, all relations are decidable without access to the database. Corresponding XPath axes are given with n as context node. For example, $\mathit{Child}(n', n)$ means n is a child of n', corresponding to the `parent` axis. For notation, see Lemma 6.1.

formulae can be used: Simple arithmetic computations decide tree relations or determine parent (and, applied iteratively, any ancestor) or sibling of a node.

Besides the aforementioned approaches, several number-based node identification schemes have been proposed which only support path decision. Node IDs in the *pre-/postorder encoding* are pairs $\langle pre, post \rangle$ consisting of the node's preorder and postorder ranks. As mentioned in [10], simple comparison operations on the interval $[pre, post]$ decide the *Child*$^+$ (and *Child**) relations. [15] shows how to decide *Following* and *After*. It also shows how the pre-/postorder encoding is used in XPath Accelerator: being equipped with appropriate index structures and embedded into a relational system, it can decide the remaining XPath axes. Although this requires access to database tables, it performs very efficiently due to its tight integration with the database system.

A related scheme is the *interval encoding* [22, 34], whose IDs are pairs $\langle pre, size \rangle$ where pre is the node's preorder rank and $size$ is an integer equal to or larger than the number of descendants of that node. (Actually, [34] essentially replaces the $size$ component with $pre + size$.) In both cases, path decision operates on the interval $[pre, pre + size]$. In contrast to pre-/postorder encoding, interval encoding features a mechanism for limited updates, since it uses a sparse ID set.

scheme	path decision								path reconstruction				
	$Child^i(m,n)$	$Child^+(m,n)$	$NextSibling^i(m,n)$	$NextSibling^+(m,n)$	$Following(m,n,i)$	$Following(m,n)$	$NextInDocOrder^i(m,n)$	$NextInDocOrder^+(m,n)$	$parent^i(n)$	j-th-$child(n)$	$prevSibling^i(n)$	$nextSibling^i(n)$	i-th-$commonAnc(m,n)$
BIRD	•	•	•	•		•		•	•	3	•	3	•
ORDPATH	•	•	2	•		•		•	•	2, 3	2	2, 3	•
Virtual Nodes	•	•	•	•					•	3	•	3	•
PID	•	•							•				•
pre-/postorder	1	•				•	•	•					
interval encod.	1	•				•		•					
preorder							•	•					

• supported
1 requires level
2 non-dynamic version only
3 supported, but may not exist physically

Table 2: Expressivity of different ID schemes.

Expressivity. As mentioned in the introduction, the quality of a node identification scheme can be measured looking at three criteria: Expressivity, efficiency, and storage consumption. The discussion of storage consumption and efficiency is postponed to the next section. Expressivity of the various schemes is desribed in Table 2: A bullet in a cell indicates that a node identification scheme supports the evaluation of the tree relation defining the column *without* access to any database table. Table 2 is divided into node identification schemes supporting path reconstruction (the first four rows) and ones supporting decision only. Numbers in the table cells describe the following restrictions: (1) Path reconstruction in forward direction, i.e. construction of children or right siblings, is hypothetical in the sense that node identifiers can be constructed that do not correspond to actual nodes of the database tree. (2) ORDPATH can reconstruct siblings only in its non-dynamic form, that abandons its update capability. (3) For pre-/post, the child decision problem can only be solved with additional information in the form of level information.

Table 2 contains two functions not yet defined, but useful in actual XPath implementations: *j-th-child*(n) denotes the j-th child of n, and *i-th-commonAnc*(m, n) i-th common ancestor of m and n (bottom-up). The values for i are positive or negative integers for all relations in the decision part, whereas the values for i and j are restricted to positive integers for all functions in the construction part.
For most of the schemes it is either described in the original literature how they treat a given decision or reconstruction problem, or it is straightforward: The functions *i-th-commonAnc* and *lowest-commonAnc* for example can be constructed by path-based identification schemes by simply following the two sequences up to a given level, or until they diverge. For number-based schemes, iterative steps have to be applied with subsequent comparisons.

In the decision part, BIRD, ORDPATH, and Virtual Nodes support most problems. ORDPATH cannot decide horizontal proximity (e.g. `following-sibling::*[1]` in XPath) in its original (dynamic) version. Since Virtual Nodes is based on a preorder enumeration of the tree, it can decide order only on a sibling basis, but not the more general *NextInDocOrder* and *Following* relations. BIRD, ORDPATH, and Virtual Nodes have roughly the same expressivity in the reconstruction part, with ORDPATH being slightly inferior, since it can not reconstruct siblings in its original form. ORDPATH can reconstruct siblings only in its non-dynamic form, which abandons its update capability.

8. UPDATES WITH THE BIRD SCHEME

Updates of indexed document collections may affect the IDs assigned to individual document nodes, depending on (1) where the update occurs (e.g., inside an existing document or in a new document), (2) which kind of update occurs (insertion vs. removal of a node) and, in case of an insertion, (3) how many nodes are added at a given position. A node removal can be handled by simply leaving that node's ID unassigned. In the following, we therefore focus on node insertion.

In some scenarios, updates occur either rarely (like in static databases containing, e.g., medical, juridical, geographical or historical information), or new data are first collected and then added to the database in a bulk update once in a while (e.g., in digital archives, linguistic corpora, encyclopedias and dictionaries, product catalogues, or digital libraries). Under such circumstances, robustness is a minor concern, whereas storage demands and runtime performance are much more important. A straightforward solution is to reindex the entire document collection from time to time. On the other hand, in dynamic databases whose contents change frequently, like news repositories, auction servers, or flight booking services, such a strategy is clearly infeasible. Here node insertions must be done *incrementally*, i.e., without affecting too many of the nodes indexed before.

Although a thorough investigation of updates with BIRD is outside the scope of this work and remains to be done, we sketch two different strategies here to illustrate that the BIRD scheme is appropriate not only for static databases, but capable to adapt to different kinds of dynamic data. The second strategy is also interesting from a theoretical point of view since it generalizes the update technique of Dewey encoding.

Sparse ID encoding. As illustrated in Figure 2, a balancing degree $s \geq 1$ causes a certain amount of IDs to be

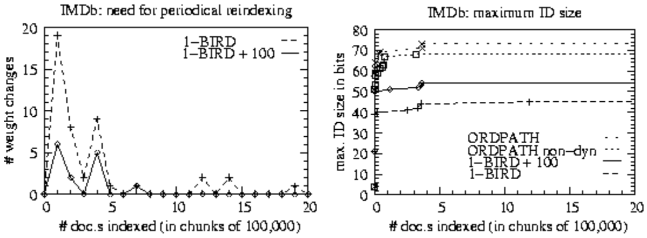

Figure 6: Robustness and ID size on *IMDb*.

left unassigned. For instance, with the child-balanced BIRD scheme ($s = 1$), 75 IDs are reserved for the subtree rooted at the node with the ID 150 in Figure 2 *(a)* although the subtree contains only two nodes. This is because the node 150 inherits the weight 75 via child balancing from its left sibling, whose subtree is much greater. When inserting nodes in the subtree below node 150, the odds are that the corresponding IDs are still unassigned such that no reindexing is necessary. Of course, inserting a node in a subtree whose ID space is exhausted causes an overflow. As a result, the weight not only of the overflowing node, but also of its siblings in the DataGuide changes (again due to child balancing). This update may propagate up through the DataGuide and thus spoil the weights of all document nodes. Because overflows cause a periodical reindexing of the entire document collection, BIRD's inherent update capabilities due to the sparse encoding just described should be relied upon only when the data is known to remain reasonably homogeneous over time, with only little difference in the size of subtrees below the same label path. To reduce the overflow risk further, one may also deliberately leave some extra IDs unassigned, as suggested by [22], at the expense of an increased ID size (see below).

In many applications node insertions do not occur at arbitrary positions in the document tree, but only at the end of the collection (i.e., after the last node visited in a preorder traversal). This further reduces the risk of overflow. As a special case, consider collections of bibliographic data like *DBLP* [9] or the large *Internet Movie Database* (*IMDb*) [17] (see also Table 3), where the bulk of insertions happen when adding new documents (i.e., in the case of *IMDb*, new files describing movies, actors, directors, or producers). This does not alter the nodes in existing documents (unless, for a balancing degree $s \geq 1$, the new document changes the weights of one or more label paths, in which case the node IDs of at least all nodes with that path throughout the database are affected). Hence for such collections of more or less homogeneous documents with updates at the document level only, incremental updates are not mandatory.

As an example of a large real-world document collection of the kind just described, we had the *IMDb* collection converted to XML and indexed the resulting 8.4 GB of XML data (nearly 2,000,000 documents) in chunks of 1,000 documents (about 4-6 MB per chunk). Figure 6 shows BIRD's overflow behaviour and space consumption as more and more documents are added. In a first experiment, no future insertions were anticipated, i.e., the weight of a given label path is always just as large as it must be to accommodate the largest known subtree below that path. We then indexed *IMDb* once again, this time reserving extra IDs for 100 potential child node insertions below any overflowing node during the weight computation ("BIRD + 100" in the figure).

The plot on the left in Figure 6 illustrates how many times at least one weight in the DataGuide was changed while adding 100,000 documents, thus causing a reindexing of the entire collection. The two large peaks at the beginning indicate that the BIRD weights were reasonably stable after indexing the first 400,000 documents, or 20% of the data. Up to that point, a large number of overflows occurred in the first experiment (dashed line), which was reduced significantly by applying the extra-sparse encoding (solid line). Note that in these early stages reindexing is much cheaper than later on, after many documents have been added to the collection. In the sequel, the need for reindexing dwindles quickly, especially for BIRD + 100 which triggers only one more weight update before adding 1,300,000 documents without overflow.

In the right plot in Figure 6 we observe an early saturation of the ID sizes (the maximum was mostly reached after indexing less than 20% of the documents) and a very low overall space consumption for BIRD (at most 45 bits per ID, given more than 83,000,000 nodes). Obviously reserving extra IDs to increase the robustness of the scheme is not expensive in terms of storage: the greatest BIRD ID in the extra-sparse encoding ("BIRD + 100", at most 54 bits per ID) still occupies far less than 64 bits, a critical boundary in our runtime experiments (see Section 9.3). Although with a maximum depth of five the *IMDb* collection is fairly shallow, ORDPATH IDs grow rapidly beyond the 64-bit line (max. ID size 73 bit), even when the sparse encoding for future updates is disabled, which keeps the IDs as small as possible ("ORDPATH non-dyn" in Figure 6; max. ID size 68 bit). The latter is similar to Dewey Order, but with the more compact ORDPATH binary encoding applied.

Layered BIRD scheme. As mentioned in Section 7, Dewey Order gracefully handles node insertions because altering a given component, or *layer*, of a Dewey ID does not affect the remaining parts of the ID. One can regard Dewey Order and its derivates, such as ORDPATH, as a special case of layer-based ID schemes where each layer corresponds to one level in the document tree. Yet in general, multiple levels may be subsumed by the same layer and therefore represented by the same component of a multi-layer ID.

Figure 7 *(a)* depicts the same document collection as Figures 2 and 3 before, but with two layers covering the five levels in the documents and, consequently, with BIRD IDs consisting of two components. The upper layer covers the three topmost levels. Nodes on these levels in the documents have as first ID component ordinary BIRD IDs and as an implicit second component 0 (omitted in the figure). Lower-level nodes inherit the first ID component from their lowest ancestor on the upper level, while the second component is also an ordinary BIRD ID. For instance, all nodes below node 7 on the lower layer in Figure 7 *(a)* have 7 as their first ID component. Similar to Dewey Order encoding, the second component of their IDs is independent of the upper-layer component, which facilitates incremental insertions on any layer. For instance, any number of children may be added below node 7 (with IDs 7/12, 7/15, . . . , according to the BIRD scheme on the lower layer), without affecting the IDs of any node on the upper layer or any of their descendants on the lower layer. In fact, overflows may only occur inside a document subtree on a given layer (e.g., if a right sibling of node 7/11 had to be added). But since there may

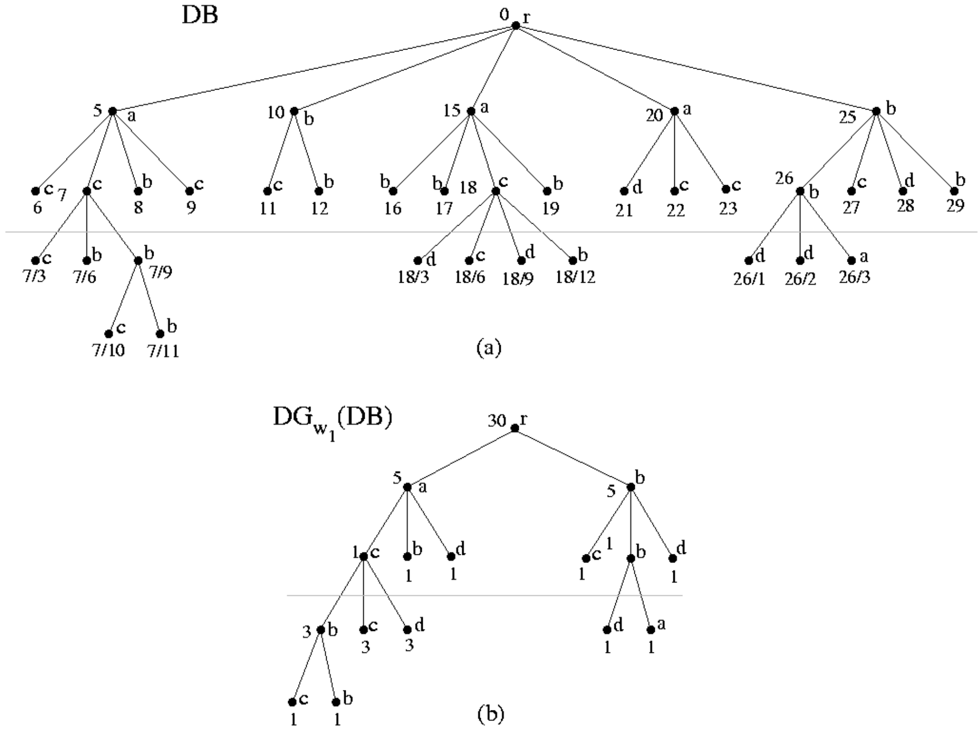

Figure 7: Layered child-balanced numbering scheme with two layers. *(a)* Database. *(b)* DataGuide for *(a)*.

be any number of subtrees on any layer, the layered BIRD scheme still supports arbitrary many insertions (though not at all positions in the document tree).

The BIRD weights on each layer are easy to determine using the bottom-up procedure described in Section 4.1. A new layer is introduced in the DataGuide as soon as a suitable insertion point is reached (e.g., right above the "movie" level in the *IMDb* collection). Thus any number of layers may be created, up to the extreme case where each document level is on a different layer, and layered BIRD coincides with Dewey Order. Layering also helps to prevent individual weights from growing too large: when the desired upper bound is reached, the current layer is closed, and weighting restarts with a leaf value of 1. In fact, any layer may even span only part of a level in the document tree, and different label paths may cross a different number of layers. Thus the IDs of two nodes on the same document level need not consist of the same number of components, e.g., if the first node is part of a much richer subtree requiring more layers than the second one. The exact number and position of the layer boundaries in the DataGuide determines the size of the resulting layered BIRD IDs as well as the positions in the document tree where unlimited insertions are supported. As other Dewey derivates, layered BIRD benefits from a suitable ID encoding (e.g., UTF-8 as proposed by [32], or ORDPATH's binary encodings [28]) for storing the variable-sized ID components in a compact manner.

Finally, all decision and reconstruction operations on BIRD IDs are easily adapted to the layered variant. As a matter of fact, only one ID component is manipulated like in the unlayered case, whereas all other components are either ignored or removed from the ID. For instance, in order to reconstruct the $\mathit{parent}^i(n)$ relation, one first goes up i levels in the DataGuide to determine the weight of the ancestor to be reconstructed. If one or more layer boundaries are crossed, the ID components corresponding to the layers below the boundaries are discarded. The ancestor ID's component on the target layer is computed from the corresponding descendant component as usually, for the number of levels covered by that layer; any higher-layer components remain unchanged. Consider, e.g., the node 7/10 in Figure 7 *(a)*. If $i = 1$ then no layer boundary is traversed, and the ancestor ID of 7/10 is computed as $7/(10 - (10 \bmod 3)) = 7/9$. For $i = 2$, the second ID component is removed, and BIRD reconstruction computes $7 = 7-(7 \bmod 1)$ as the first component of the ancestor ID. Similarly, all higher ancestors of 7/10 are reconstructed: $\mathit{parent}^3(7/10) = 7-(7 \bmod 5) = 5$, and $\mathit{parent}^4(7/10) = 7 - (7 \bmod 30) = 0$. For deciding $\mathit{Child}^+(m, n)$, we check whether the relation holds for m's and n's ID components on m's layer and whether all preceding components are equal in both IDs. Comparing document nodes according to the $\mathit{NextInDocOrder}^+(m, n)$ relation is done component-wise in top-down direction, as with Dewey Order.

9. EXPERIMENTS AND EVALUATION

This section reports on our experimental evaluation of different identification schemes, namely BIRD (child-balanced, i.e. $s = 1$), ORDPATH [28] (encoded with max. 9 bits for length and max. 20 bits for offset components), Virtual Nodes [21], and PID [7]. We applied each scheme to the first three document collections listed in Table 3, which differ considerably in size and structural complexity (in terms of the number and length of the label paths occurring in the documents). We implemented the four schemes to be compared as described in the original literature. In line with the quality criteria mentioned in Section 1, we examine the

name	XML size	# nodes	# label paths	depth
Cities	1.3 MB	36,375	253	7
DBLP	157 MB	5,390,160	129	7
XMark	1,145 MB	20,532,979	549	13
IMDb	8,633 MB	83,404,825	276	5

Table 3: Document collections.

scheme	ID size (bits)			total storage (kB)			
				variable ID size		fixed ID size	
	min.	max.	avg.	absolute	% pre	absolute	% pre
BIRD	1	24	22	104	161	113	150
ORDPATH	2	49	33	151	232	223	305
(non-dyn)	2	41	27	123	189	186	255
Virtual N.	1	58	37	168	261	264	363
PID	1	14	11	50	78	64	88
preorder	1	16	14	65	100	73	100

Table 4: Storage consumption for *Cities*.

scheme	ID size (bits)			total storage (MB)			
				variable ID size		fixed ID size	
	min.	max.	avg.	absolute	% pre	absolute	% pre
BIRD	1	37	36	25	170	25	161
ORDPATH	2	53	37	26	186	36	240
(non-dyn)	2	52	36	25	179	35	233
Virtual N.	1	95	37	25	174	64	413
PID	1	28	21	14	99	19	122
preorder	1	23	21	14	100	15	100

Table 5: Storage consumption for *DBLP*.

scheme	ID size (bits)			total storage (MB)			
				variable ID size		fixed ID size	
	min.	max.	avg.	absolute	% pre	absolute	% pre
BIRD	1	44	43	113	188	113	177
ORDPATH	2	86	48	124	207	221	345
(non-dyn)	2	77	43	111	185	198	309
Virtual N.	1	198	81	210	350	508	794
PID	1	29	20	54	90	74	116
preorder	1	25	23	60	100	64	100

Table 6: Storage consumption for *XMark*.

storage consumption (see Section 9.1) and the runtime performance of all identification schemes, both for individual reconstruction and decision operations (see Section 9.2) and for entire tree queries (see Section 9.3). Experimentals results on the robustness of BIRD are given in Section 8.

As testbed we used the native XML retrieval system X^2 [25]. X^2 is implemented in Java (J2SDK 1.4.2) and accesses a relational database backend via JDBC. Query evaluation and join algorithms manipulate trees in main memory after sets of document nodes have been fetched from the RDBS. Since the algorithms may be further optimized, we focus on a comparison of the retrieval results for different ID schemes, rather than on absolute performance numbers, which are machine-dependent anyway. All tests were carried out sequentially on an i686 computer with an AMD Athlon XP 2600+ CPU running at 2138 MHz with 256 kB cache. The machine has 1 GB RAM and runs Slackware Linux 1.9 with kernel 2.4.26. The relational backend is PostgreSQL 7.3.2 running on the same machine as X^2, with database cache disabled. Apart from these two processes, the computer was idle during the experiments.

9.1 Storage consumption

The storage consumption of various identification schemes on the four document collections are given in Tables 4 to 6. The first three columns after the scheme name contain the minimum, maximum, and average number of bits used for a single ID, respectively. The remaining columns list the storage needed for all IDs together, both as an absolute value in MB (kB for *Cities*) in columns five and seven, and relative to the corresponding result obtained for the preorder scheme (columns six and eight), which is the baseline in our experiments. The relative values are computed on bit counts, whereas the absolute values are rounded to the nearest MB (kB for *Cities*).

We apply two different methods to compute the total storage consumed by a given identification scheme. On the one hand, we sum up the exact bit counts needed for the IDs, assuming that IDs can be stored with variable size. This produces the absolute (relative) values in the fifth (sixth) column, which follow the average ID sizes in column four. On the other hand, it is more realistic to assume that when stored in the database, all IDs assigned to nodes in the same document collection take up the same space. The total storage taken up by such fixed-size IDs is the product of the maximum ID size, as given in column three, and the total number of nodes in the collection (see Table 3). The resulting values appear in columns seven (absolute) and eight (again relative to the values obtained for the preorder scheme).

We found that the BIRD scheme almost always takes up considerably less space than ORDPATH and especially Virtual Nodes, the two schemes which are closest to BIRD in terms of expressivity (see Section 7). When assigning fixed-size IDs BIRD reduces the space consumption by nearly a factor 2 for ORDPATH and between 2.2 and 4.5 for Virtual Nodes. The reason is that for BIRD the maximum ID size is much closer to the average size than for ORDPATH and Virtual Nodes, which therefore incur a significant storage overhead for fixed-size IDs. For variable-size IDs this factor decreases, but BIRD IDs still are clearly smaller than those of other schemes.

As the only approach (except preorder) with smaller IDs than BIRD, the PID scheme optimizes storage at the expense of expressivity, as shown in Table 2. Remarkably, PID occupies less space than the preorder scheme in our experiments, at least when assuming variable-size IDs. In the underlying trade-off between expressivity and space consumption, the PID scheme chooses an intermediate position between schemes with high expressivity and storage consumption, such as Virtual Nodes, on the one hand and schemes with low expressivity and storage consumption, such as pre-/postorder encoding or interval encoding, on the other hand.

In further experiments with more deeply nested, text-oriented document collections, such as the *INEX* benchmark corpus consisting of extremely heterogeneous and layout-polluted research articles [18], we observed that on average BIRD IDs grow larger than ORDPATH IDs (97 vs. 60 bits; Virtual Nodes 78 bits), whereas their maximum size is still smaller than that of ORDPATH (98 vs. 135 bits; Virtual Nodes 217 bits). Child-balancing here blows up the weights of label paths leading to subtrees which greatly vary in size. Obviously, this could be avoided if equal weights were assigned to nodes with a similar number of descendants, rather than with equal label paths. Designing the corresponding weight index to be used as structural summary clearly departs from the DataGuide, but as mentioned in Section 3, BIRD may be combined with any index structure providing efficient access to the weights. Preliminary experiments

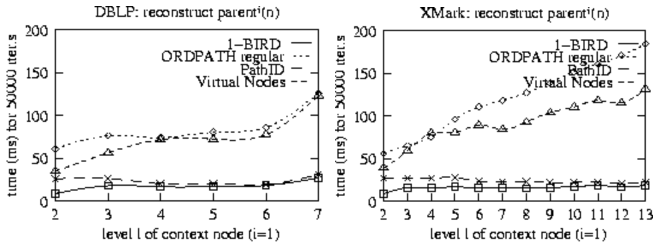

Figure 8: Reconstructing ancestors from varying levels.

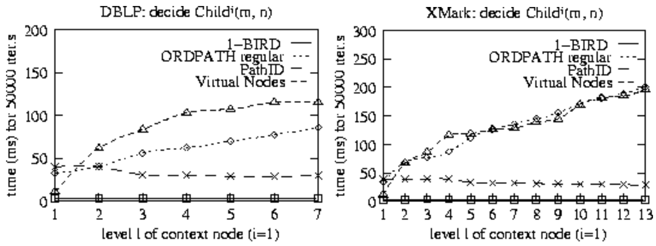

Figure 10: Deciding fixed ancestor from varying levels.

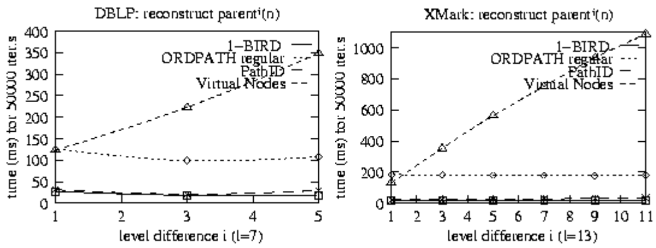

Figure 9: Reconstructing ancestors in varying proximity.

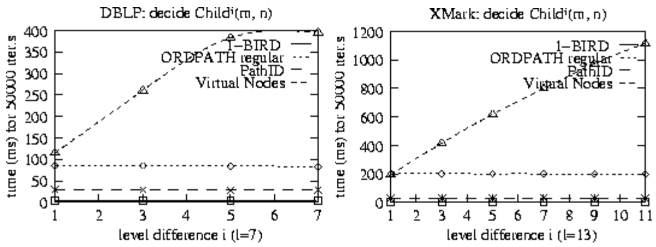

Figure 11: Deciding varying ancestors from fixed level.

show that for the *INEX* collection, the maximum ID size may be reduced to 64 bits, i.e., below the performance-critical boundary discussed in the next section, although the resulting weight index is huge. The exact size of the IDs as well as of the weight index depends on which document nodes share the same index node and weight, being regarded as equivalent in terms of their subtree sizes. The finer the underlying equivalence relation, the better the weights reflect the actual subtree sizes, but the more index nodes are needed. Future work may be concerned with methods to optimize this trade-off between ID size and index size.

9.2 Decision and reconstruction speed

The first set of runtime experiments measure the efficiency of decision and reconstruction with different ID schemes. Figures 8 to 11 plot the computation time needed for various decision and reconstruction problems on the *DBLP* and the *XMark* collection. Results for *Cities* are not shown, but reveal similar tendencies. All four schemes (excluding preorder, for obvious reasons) were tested with the same set of synthetically generated problems. Since the speed of individual operations cannot be measured with sufficient confidence, the figures represent the accumulated time (in milliseconds) needed for 50,000 repetitions of each decision or reconstruction. Note that this subsumes all necessary operations including, e.g., DataGuide accesses for BIRD or PID and ID comparison during decision.

Reconstruction. Figure 8 shows the time needed to reconstruct the parents of nodes at different levels. For *DBLP* (left-hand side) and *XMark* (right-hand side), PID is almost as fast as BIRD, whereas ORDPATH and Virtual Nodes are slower by at least a factor 4. On *XMark*, the difference between BIRD and ORDPATH is up to one order of magnitude. Obviously the performance of both BIRD and PID is independent of the level of the source node. For ORDPATH, the computation time grows with the depth of the source node. The reason is that ORDPATH bit strings must be parsed top-down (i.e., from left to right) down to the level of the source node. The deeper the source node is located in the tree, the longer the parsing takes. We observe the same effect for Virtual Nodes on *DBLP* and *XMark* although in theory its ancestor reconstruction works in constant time (see below). Presumably the data structure representing numbers of arbitrary size, used for Virtual Nodes here because of the sheer length of the IDs, creates an overhead for arithmetic operations on ID values. Since breadth-first IDs grow larger on deeper levels, this explains why the performance of Virtual Nodes degradates in Figure 8. The effect is not observed for *Cities* where Virtual Nodes IDs fit in 64 bits (not shown in the figure).

Figure 9 illustrates the orthogonal situation: here the $parent^i(n)$ relation is reconstructed from source nodes at a fixed depth in the tree (level 7 for *DBLP*, 13 for *XMark*), with varying distance i. As in Figure 8, BIRD and PID are much faster than ORDPATH and Virtual Nodes (nearly one order of magnitude; mind the different scales) and reveal no dependency on the number of levels to be traversed. Using the DataGuide as a path summary, both schemes climb up a path in the index (which takes practically constant time), rather than reconstructing all ancestors iteratively like Virtual Nodes which therefore suffers from a linear degradation for bigger distances i. ORDPATH's bit shift operations are indifferent to proximity.

Decision. Figures 10 and 11 are based on a similar setting as Figures 8 and 9, but this time for the decision of the $Child^i(m,n)$ relation. We observe the same dependencies on the level of the source node and the distance to the target node as before. BIRD is as fast as for reconstruction (3 ms for 50,000 iterations), whereas PID is one order of magnitude slower. On *DBLP*, BIRD outperforms ORDPATH and Virtual Nodes by a factor 30 or 40, respectively (up to 100 for Virtual Nodes with a level difference of 7); on *XMark*, the difference is nearly two orders of magnitude (up to 400 for Virtual Nodes with a level difference of 13).

Asymptotic behaviour. Table 7 summarizes the dependencies of all ID schemes on different properties of the nodes involved in a decision or reconstruction problem. The re-

scheme	path decision								path reconstruction				
	Child$^i(m,n)$	*Child*$^+(m,n)$	*NextSibling*$^i(m,n)$	*NextSibling*$^+(m,n)$	*Following*(m,n,i)	*Following*(m,n)	*NextInDocOrder*$^i(m,n)$	*NextInDocOrder*$^+(m,n)$	*parent*$^i(n)$	*j-th-child*(n)	*prevSibling*$^i(n)$	*nextSibling*$^i(n)$	*i-th-commonAnc*(m,n)
BIRD	•	•	•	•		•		•	•	•	•	•	d
ORDPATH	l	•	l	l		•		•	l	•	l	l	l–d
Virtual Nodes	p	d	•	•					p	•	•	•	l–d
PID	•	•							•				d
pre-/postorder	•	•				•	•	•					
interval encoding	•	•				•		•					
preorder							•	•					

• constant l linear in source node level p linear in proximity
d linear in distance to target node

Table 7: Asymptotic behaviour of different ID schemes.

sults are based on a theoretical analysis and were mostly confirmed in our experiments with BIRD, ORDPATH, PID, Virtual Nodes and preorder. For any given ID scheme (row) and decision or reconstruction problem (column), • means computation in constant time (i.e., no dependency), l means linear degradation with growing depth of the source node, and p linear degradation with growing proximity distance (e.g., for *parent*$^i(n)$, the number i of levels to be traversed upward). For *Child*$^+(m,n)$ and *i-th-commonAnc*(m,n), d indicates a linear dependency on the actual distance to the target node (i.e., m or the i-th common ancestor of m and n), which in the latter case depends partly on the proximity i. Unsupported problems are left unmarked (as in Table 2).

As shown in the first row of Table 7, BIRD solves all decision and nearly all reconstruction problems in constant time. For *Child*$^i(m,n)$, *Child*$^+(m,n)$ and *parent*$^i(n)$, we assume that ancestor nodes in the DataGuide are practically accessible in constant time, which our experiments confirm (see above). BIRD computes *i-th-commonAnc*(m,n) by reconstructing part of the root path of either m and n bottom-up, intertwined with constant-time decisions, until the i-th common ancestor is found. Therefore this operation depends only on the distance d to the target node.

Due to its left-to-right bit encoding, ORDPATH's performance depends linearly on the level of the source node in most cases. Deciding *Child*$^i(m,n)$, *NextSibling*$^i(m,n)$, *NextSibling*$^+(m,n)$ or reconstructing their counterparts requires access to a certain level relative to the source node's level, which means that the entire bit string down to that level must be parsed—unlike PID where the number of bits to be shifted is available bottom-up in the DataGuide. The same is true for *i-th-commonAnc*(m,n). Unlike BIRD, ORDPATH reconstructs the common path prefix of m and n top-down. Since the length of this prefix is $l-d$, higher common ancestors are reconstructed faster than lower ones.

QID	SCHEME	PATH JOIN STRATEGY		
		ALWAYS	FIRST	NEVER
Q0	BIRD	4353	4107	7913
	ORDPATH	4759	4170	8176
	PathID	4817	4415	8557
	Virtual Nodes	9244	19829	33120
	Preorder	122235	4015	7892
Q1	BIRD	125	249	138
	ORDPATH	158	270	162
	PathID	139	268	156
	Virtual Nodes	260	4324	6472
	Preorder	4559	4587	6288
Q2	BIRD	4337	4241	11693
	ORDPATH	4625	4431	12249
	PathID	4773	4625	12902
	Virtual Nodes	9074	10232	320639
	Preorder	114915	5871	16156
Q3	BIRD	170	150	174
	ORDPATH	270	171	191
	PathID	266	147	191
	Virtual Nodes	483	331	10154
	Preorder	4398	4239	8244

Table 8: Runtime performance for tree queries against *DBLP* (avg ms).

The Virtual Nodes scheme needs time linear in the proximity parameter i for deciding *Child*$^i(m,n)$ or reconstructing *parent*$^i(n)$, as shown in Figures 9 and 11. Analogously, *Child*$^+(m,n)$ depends linearly on the distance d to the target node. All three problems are solved by iterative parent reconstruction, which explains this behaviour. As discussed above, the strong dependency on the source node level observed in our experiments for *Child*$^i(m,n)$ and *parent*$^i(n)$ on *DBLP* and *XMark* (see Figures 8 and 10) is not justified theoretically and therefore omitted in Table 7 (but nevertheless relevant in practice). As for BIRD, deciding or reconstructing the sibling relations and *j-th-child*(n) with Virtual Nodes is trivial and works in constant time. For *i-th-commonAnc*(m,n), combining reconstruction and decision like with BIRD would lead to a complexity of $O(d^2)$ because deciding *Child*$^i(m,n)$ is already linear in d. As a remedy, one can reconstruct the complete root paths of m and n and then locate the i-th common ancestor at level $l-d$ top-down, as with ORDPATH.

PID supports *Child*$^i(m,n)$, *Child*$^+(m,n)$ and *parent*$^i(n)$ in constant time, again assuming instant access to ancestors in the DataGuide. Consequently, the scheme solves *i-th-commonAnc*(m,n) in time linear in the target node distance, regardless of the source node level, like BIRD (but unlike ORDPATH and Virtual Nodes).

All decision and reconstruction problems supported by pre-/postorder encoding, interval encoding and preorder require only constant-time arithmetic operations.

9.3 Runtime performance for tree queries

To quantify how much the differences in decision and reconstruction speed observed in Section 9.2 affect the overall performance for entire tree queries, we evaluated four sample queries using the same schemes as in the previous section, both against the *DBLP* and the *XMark* collection (see Table 12). To avoid artefacts due to file system cache effects, the best and the worst result of six consecutive iterations of each query were discarded. The remaining four iterations of the same query (occasionally fewer for some long-running queries) were then averaged. Tables 8 and 9 and contain the total evaluation times (without profiling). A second set of runs was used to measure the contribution of individual query stages, as given in Tables 10 and 11. Due to the nature of X^2's native tree query language, the tests involve the decision of *Child*$^i(m,n)$ and the reconstruction of *parent*$^i(n)$ only (see Appendix B). Our query language corresponds to a subset of XPath supporting the axes `child`,

QID	SCHEME	PATH JOIN STRATEGY		
		ALWAYS	FIRST	NEVER
Q0	BIRD	617	597	4817
	ORDPATH	1534	1535	12343
	PathID	662	577	5320
	Virtual Nodes	1723	5760	295084
	Preorder	23925	7569	20613
Q1	BIRD	2634	2591	6745
	ORDPATH	6248	6293	20068
	PathID	2908	2855	8231
	Virtual Nodes	6913	636649	4749424
	Preorder	92430	97455	188456
Q2	BIRD	14385	14072	19529
	ORDPATH	37149	36355	49827
	PathID	14919	14978	20668
	Virtual Nodes	36854	65589	82331
	Preorder	567524	13799	18282
Q3	BIRD	30	37	9957
	ORDPATH	98	86	25733
	PathID	36	42	11102
	Virtual Nodes	86	89	228493
	Preorder	1047	1057	14521

Table 9: Runtime performance for tree queries against *XMark* (avg ms).

`attribute` and `descendant`. Nodes can be selected based on their label and/or textual content. Note that an answer to a query comprises the matches to all nodes in the query tree, not just one focussed node as in XPath. The same evaluation algorithm is used for all ID schemes, just the reconstruction, decision, and comparison operations vary. The only exception is that schemes which do not preserve preorder (i.e., PID and Virtual Nodes) cannot benefit from certain optimizations (see below and Appendix B). As a baseline, we use preorder IDs with brute-force reconstruction and decision: reconstructing the i-th ancestor of a node requires i look-ups in a parent/child index mapping the preorder ID of any node to the ID of its parent node. The parent/child index is stored as a table in the relation backend.

In order to estimate the benefits of reconstruction operations (which are not supported by all ID schemes, see Section 7), we implemented and tested the three *path join strategies* *ALWAYS*, *FIRST*, and *NEVER* which differ in their use of reconstruction of the $parent^i(n)$ relation. Appendix B explains the strategies in detail. In short, *ALWAYS* means that the matches of any branching node in the query tree are joined with those of its child nodes by reconstructing the ancestors of the child matches and testing whether they are contained in the branching node's set of matches. Since X^2 evaluates queries bottom-up, the first child of any branching query node does not undergo the path join (which would fail for the empty set of parent matches), but simply propagates its matches up to the parent node by reconstruction. The same is true for the second strategy, *FIRST*, which treats only subsequent children differently. Here the path join decides for each pair of matches to the branching node and its child node whether the $Child^i(m, n)$ relation holds. No test for set containment is needed, and schemes respecting document order may benefit from optimizations saving the decision for some pairs of nodes. The third strategy, *NEVER*, avoids reconstruction altogether, even for the first child of a given branching node. Instead of propagating matches upward in the query tree, all nodes in the documents with a path matching the path of the branching node are retrieved and then joined with the matches of its first child query node by deciding the $Child^i(m, n)$ relation. Subsequent children are handled as described for the *FIRST* strategy.

Summary. The following key results sum up the outcome of our experiments (see below for a detailed analysis):

Result 1. *The BIRD scheme performs best for virtually all queries and path join strategies, both on DBLP and XMark.*

The overall performance in all tests against the *DBLP* and *XMark* collections is given in Tables 8 and 9. Each of the three rightmost columns corresponds to one of the three path join strategies explained above. BIRD almost always outperforms the other schemes, beaten only once by PID (*DBLP*: Q3 *FIRST*; *XMark*: Q0 *FIRST*) and twice by preorder (*DBLP*: Q0 *FIRST* and *NEVER*; *XMark*: Q2 *FIRST* and *NEVER*). The most efficient schemes compared to BIRD are PID (*DBLP*: factor ≤ 1.6; *XMark*: factor ≤ 1.2) and ORDPATH (*DBLP*: factor ≤ 1.6; *XMark*: factor ≤ 3.3). In terms of absolute numbers, the greatest difference between BIRD and PID is 1.2 seconds on *DBLP* and 1.5 seconds on *XMark*. ORDPATH is on *DBLP* up to 0.6 seconds slower and on *XMark* up to 30 seconds. The distance to Virtual Nodes is considerable (*DBLP*: factor ≤ 58; *XMark*: factor ≤ 704 compared to BIRD). In extreme cases, Virtual Nodes is one order of magnitude slower than the baseline, preorder, and even more compared to the other schemes, especially when reconstruction is disabled (e.g., Q1 *NEVER* in Table 9). The exact performance differences vary dramatically with the time spent on ID comparisons (see also the following results). In terms of absolute numbers, the greatest difference between BIRD and Virtual Nodes is more than one hour. As could be expected, brute-force reconstruction and decision with preorder IDs is usually very slow, especially when other schemes benefit from extensive use of inmemory reconstruction. Evaluation with preorder IDs takes up to 40 times or 10 minutes longer than with BIRD IDs.

Result 2. *The efficiency of ID comparisons has a greater impact on the overall performance than reconstruction and decision, and can be affected by the ID size.*

A detailed profiling of different evaluation ingredients (see Tables 10 and 11) proves that most of the query evaluation time is spent on comparing node IDs, both during decision and, most prominently, when manipulating the sets of potential matches fetched or reconstructed before. While decision and reconstruction contribute up to one second to the total evaluation time, ID comparison easily takes two orders of magnitude longer. Accordingly, the time spent on reconstruction and decision differs by one second or less among the schemes (ignoring cases where Virtual Nodes must perform far more decision operations than the other schemes, see Result 4), whereas the efficiency of ID comparison can make a difference of 20 seconds and more. As the difference between Virtual Nodes and the other schemes on *DBLP* shows, the size of the IDs can have a huge impact on the performance of all ID operations (most notably, the frequent comparisons): as the only scheme whose IDs do not fit the native 64-bit data types provided by most high-level programming languages, Virtual Nodes suffers from a considerable overhead even for the strategy *ALWAYS* (a second handicap of Virtual Nodes for the other two strategies is subsumed under Result 4). ORDPATH is subject to the same effect on *XMark* where its IDs grow larger than 64 bits, too. While the impact of the ID size depends on the underlying computer architecture as well as the data structures used, schemes exceeding a certain ID size will always incur some runtime overhead, not to speak of the disk space they occupy.

Result 3. *Reconstruction is of paramount importance to efficient query evaluation because it saves ID fetching and comparison.*

A comparison of the three path join strategies *ALWAYS*, *FIRST* and *NEVER* (see Section 9.3) clearly shows that reconstruction is key to efficient query evaluation. Performance decreases dramatically for all schemes and almost all queries when reconstruction is disabled (strategy *NEVER*, as opposed to *FIRST* and *ALWAYS*). The fact that the huge overhead incurred by *NEVER* is mainly due to ID comparisons rather than node fetching illustrates that our results do not only apply to native retrieval systems like X^2 but also, perhaps to a lesser extent, to other engines where fetching is cheaper (such as purely relational systems). BIRD, ORDPATH and PID prefer *FIRST* with its mixture of reconstruction and decision, owing to their efficient decision

QID	SCHEME	PATH JOIN STRATEGY (USE OF RECONSTRUCTION)														
		ALWAYS					FIRST					NEVER				
		REC.	DEC.	JOIN	FETCH	COMP.	REC.	DEC.	JOIN	FETCH	COMP.	REC.	DEC.	JOIN	FETCH	COMP.
Q0	BIRD	95	0	555	1344	3163	0	0	479	1372	3145	0	0	454	3288	5777
	ORDPATH	309	0	754	1458	3305	0	1	463	1442	3337	0	1	482	3435	6071
	PathID	105	0	633	1321	3210	0	0	489	1302	3235	0	1	515	3293	5862
	Virtual Nodes	279	0	871	1789	8196	0	11155	13600	1593	8229	0	19207	22846	3189	14217
	Preorder	105985	52	115670	104931	3298	112	96	560	1423	3126	177	191	657	2728	5762
Q1	BIRD	3	1	46	44	87	3	2	171	53	83	0	1	142	51	94
	ORDPATH	6	2	82	43	90	5	5	169	46	96	0	9	146	44	102
	PathID	3	1	60	36	86	3	2	99	42	88	0	7	178	45	93
	Virtual Nodes	7	4	83	37	224	8	3951	4621	51	337	0	6415	7826	39	232
	Preorder	3991	1671	2614	3953	108	3950	2493	2617	3896	109	5558	5963	6333	5484	136
Q2	BIRD	121	0	276	1465	2842	1	0	137	1359	2847	0	2	265	4695	7910
	ORDPATH	275	0	436	1498	3045	4	0	136	1402	2983	0	8	304	4844	8461
	PathID	116	0	290	1377	2934	1	1	148	1316	3025	0	5	301	3956	8117
	Virtual Nodes	279	0	480	1653	7937	5	1683	1915	1943	7485	0	313732	372942	5635	20749
	Preorder	101840	2	109615	101066	3040	1577	237	385	2799	2818	4259	4560	4886	7950	7960
Q3	BIRD	4	0	0	33	113	3	0	0	27	119	0	3	238	63	149
	ORDPATH	25	0	0	35	114	10	0	0	40	124	0	14	270	66	168
	PathID	2	0	0	32	113	5	0	0	41	121	0	10	253	50	156
	Virtual Nodes	22	0	0	40	460	9	0	0	38	373	0	9821	12011	55	402
	Preorder	3677	0	0	3654	153	3645	0	0	3592	136	7189	7767	8284	7088	192

Table 10: Profile of the runtime performance for tree queries against the *DBLP* collection (avg. ms).

QID	SCHEME	PATH JOIN STRATEGY (USE OF RECONSTRUCTION)														
		ALWAYS					FIRST					NEVER				
		REC.	DEC.	JOIN	FETCH	COMP.	REC.	DEC.	JOIN	FETCH	COMP.	REC.	DEC.	JOIN	FETCH	COMP.
Q0	BIRD	23	0	34	214	322	4	0	18	119	321	0	6	913	1710	2427
	ORDPATH	43	0	54	92	1416	22	1	39	87	1390	1	60	765	1856	11229
	PathID	10	0	31	234	303	5	1	26	101	322	0	14	431	2190	3206
	Virtual Nodes	89	1	91	161	1562	20	4138	4385	123	1516	1	288881	325576	3663	11933
	Preorder	22489	57	17651	22199	355	6402	1218	1244	6354	354	14764	15668	16427	16071	3191
Q1	BIRD	22	1	224	449	1856	24	4	226	431	1879	0	28	2442	2181	5560
	ORDPATH	135	13	396	422	6276	127	33	451	408	6236	0	299	4691	2543	18612
	PathID	45	4	239	406	1882	28	9	268	391	1895	0	71	2856	2142	5475
	Virtual Nodes	246	43	485	520	6840	201	630458	672159	736	6561	0	4789521	5255989	3306	19672
	Preorder	85921	14391	31301	84439	2256	88464	34430	34903	86842	2346	161888	174078	179089	161040	6225
Q2	BIRD	277	0	1119	4823	10647	0	0	739	4401	10614	0	0	991	6559	14670
	ORDPATH	1208	0	2483	5772	34048	0	0	1296	5058	34620	0	1	1117	8291	34493
	PathID	297	0	1346	4664	10621	0	0	856	3962	10690	0	0	782	7214	14297
	Virtual Nodes	1381	0	2681	6153	33399	0	28756	32258	5915	34196	0	35127	39637	10018	43217
	Preorder	523485	272	553241	516648	11439	357	304	1209	4617	10590	558	555	1345	6670	13854
Q3	BIRD	1	0	0	9	32	1	0	0	8	29	0	1	64	3676	8707
	ORDPATH	3	0	0	9	68	2	0	0	8	80	0	46	159	4980	23936
	PathID	2	0	0	8	26	0	0	0	7	30	0	2	71	3200	8543
	Virtual Nodes	2	0	0	9	72	3	0	0	9	81	0	200549	242698	5103	22680
	Preorder	874	0	0	867	34	879	0	0	867	37	4941	5004	5117	8305	8243

Table 11: Profile of the runtime performance for tree queries against the *XMark* collection (avg. ms).

techniques. Virtual Nodes, by contrast, suffers from a massive join overhead for this strategy, caused by the breadth-first order of its IDs (see Result 4). With its different join algorithm, *ALWAYS* brings Virtual Nodes a little closer to the other three schemes.

RESULT 4. *ID schemes preserving document order benefit greatly from path join optimizations.*

The path join strategies involving decision, i.e., *FIRST* and *NEVER*, locate ancestor/descendant pairs in sets of matches to two given query nodes. Processing these ID sets in document order has the advantage that not all possible ID pairs (i.e., the full Cartesian product) need to be checked, which may save many decision (and, consequently, comparison) operations, as explained in Appendix B. Obviously schemes like BIRD, ORDPATH and preorder benefit from this optimization whereas Virtual Nodes, whose IDs are assigned in a breadth-first traversal of the document tree, typically must decide ancestorship for many more ID pairs. The resulting overhead explains why for *FIRST* and *NEVER*, Virtual Nodes is far less competitive than for *ALWAYS*. The PID scheme, although violating the document order between arbitrary nodes, is also amenable to the optimization provided that only sets of nodes with the same label path are joined (because among these nodes, the document order is preserved). Since our test system X^2 always retrieves and joins nodes belonging to the same DataGuide node, this condition is satisfied and PID can be handled as if it were fully compatible with document order.

Detailed analysis. Table 12 lists the eight queries we run against the *XMark* and *DBLP* collections, four against each. Queries with equal number resemble each other to a certain extent: both *XMark*'s and *DBLP*'s Q0 queries are small trees with a single branching node, a textual constraint and a moderate number of results (where matches for all query nodes are counted as mentioned above). The Q1 queries are structurally similar but lack the textual constraint, which makes them less selective than their Q0 counterparts. The Q2 queries stress the path join capabilities of the system, whereas each of the Q3 queries consists of only one path.

The detailed performance results for all queries against the *DBLP* and *XMark* collections are given in Tables 10 and 11, respectively. For each of the three path join strate-

QID	HITS	QUERY
Q0	136	//article[./author[contains(., "codd")] and ./title]
Q1	4805	//incollection[./author and ./title]
Q2	1269	//article[.//author[contains(., "li")] and .//title//i and .//year]
Q3	5419	//book//cite//*/attribute::label

QID	HITS	QUERY
Q0	128	//europe//item[.//parlist[contains(.,"bedford")] and .//emph/keyword]
Q1	14699	//europe//item[.//parlist and .//emph/keyword]
Q2	225	//site//n america//item[.//description//keyword[contains(., "abandon") and .//bold] and .//name and .//*/attribute::category]
Q3	1777	//people//person//address//city[contains(., "munich")]

Table 12: Tree queries run against the collections *DBLP* (left) and *XMark* (right).

gies, there are five columns listing the average time in milliseconds spent by a given ID scheme in different evaluation stages for a given query. Each of the five stages accumulates all instances of one of the following problems that occur during evaluation of a single query:

REC. reconstruction of the $parent^i(n)$ relation

DEC. decision of the $Child^i(m, n)$ relation[7]

JOIN path join (subsumes part of REC., DEC. and COMP.)

FETCH retrieval of document nodes from the RDBS[8]

COMP. node ID comparison

Running Q0 against the both collections produces largely similar results. When applying the *ALWAYS* strategy, BIRD outperforms ORDPATH and PID and is 2-3 times faster than Virtual Nodes thanks to faster reconstruction, whereas preorder is prohibitively slow. This changes when the *FIRST* strategy introduces decision. On *DBLP*, preorder evaluation of Q0 is even slightly faster than BIRD (2.2%) and outperforms Virtual Nodes by far. The latter Virtual Nodes is especially handicapped during the join. On *XMark*, preorder is clearly inferior to any other scheme for *FIRST*. PID and BIRD are more than twice as fast as ORDPATH and beat Virtual Nodes by one order of magnitude. Applying *NEVER* slows down evaluation roughly by a factor 2 on *DBLP* and much more on *XMark*. Due to faster decision, BIRD remains on the top.

Evaluating Q1 on *XMark* takes somewhat longer than evaluating Q0 (typically one order of magnitude) because due to the missing textual query constraints, far bigger node sets must be joined. The size of the query results differs by two orders of magnitude. BIRD and PID retrieve more than 14,000 nodes in less than 3 seconds, followed by ORDPATH (6 seconds). As before, performance breaks down when reconstruction is disabled. Thus the performance ranking is similar to Q0 except that for *FIRST* and *NEVER*, Virtual Nodes is far slower even than the baseline since its join handicap weighs particularly heavy for this query. On *DBLP*, Q1 reveals as pattern similar to Q0 but is evaluated much faster. The reason is that the number of matches to all three query nodes in Q0, ignoring the textual constraint, exceeds that for Q1 by two orders of magnitude (e.g., 157,382 titles in Q0 vs. 1,195 titles in Q1). Therefore joining is much easier for Q1 even though the final result is bigger than that of Q0. As a consequence, nearly 5000 nodes are retrieved in only a few hundred milliseconds by most schemes and strategies.

[7]This subsumes part of COMP.Note that the Virtual Nodes scheme decides $Child^i(m, n)$ by reconstructing $parent^i(n)$ and then testing whether the reconstructed ancestor ID equals m. This extra reconstruction is subsumed by DEC. and not included in REC. values.

[8]Note that since preorder IDs support neither decision nor reconstruction, REC., DEC. and JOIN may subsume considerable portions of fetching time in the baseline tests.

The evaluation of Q2 on *XMark* is lengthy despite the small number of final matches. After all, joining sets of some 100,000 `name`, 100,000 `bold`, and 380,000 `category` nodes with the 102 `keyword` nodes containing the query keyword puts the system to a hard test. Without decision, BIRD and PID do the job in 14 seconds, saving 20 seconds compared to ORDPATH and Virtual Nodes. As for Q0 and Q1, the baseline is not competitive. With the *FIRST* strategy, where decision comes into play, the former three schemes are not affected whereas the response time of Virtual Nodes grows by a factor 1.8 due to the join overhead. Interestingly, preorder benefits largely from decision for joining, increasing its performance by a factor 40 compared to *ALWAYS*, and evaluates Q2 slightly faster than BIRD. The top-down join algorithm applied by *FIRST* (see Appendix B) saves preorder much time for reconstruction (and hence fetching). Disabling reconstruction decreases the performance by roughly a factor 3, but the scheme ranking remains the same.

On *DBLP*, the task is somewhat easier (as long as reconstruction is allowed) because the `//title//i` branch has only 664 matches, which quickly narrows down the 3,747 candidates of the leftmost branch in the Q2 tree. Consequently, performance figures for *ALWAYS* and *FIRST* hardly change compared to Q0 (BIRD before ORDPATH, PID, as well as Virtual Nodes and preorder). With reconstruction disabled, however, fetching 157,382 `article` matches slows down the evaluation and increases the differences between individual ID schemes. As observed for *XMark*'s Q2 query, BIRD outperforms ORDPATH and PID by 1 second, preorder by 4.5 seconds, and Virtual Nodes by 5 minutes. The latter again suffers from the join overhead.

Finally, the queries Q3 are degenerated trees each consisting of a single path, such that there are no decision and join costs for *ALWAYS* and *FIRST*. As could be expected, differences between these two strategies in the performance of any given ID scheme are negligible on either collection. BIRD retrieves 1,777 matches from *XMark* in 30 milliseconds on average, more than three times as fast as ORDPATH. PID comes close behind. Disabling reconstruction, the *NEVER* strategy entails fetching for all inner nodes on the query path. While on *DBLP* this causes 3,748 nodes to be fetched, which affects only the performance of Virtual Nodes and preorder whose decision is less efficient, on *XMark* 382,316 nodes undergo fetching and joining. Again BIRD and PID cope best with the decision problem (10 and 11 seconds, respectively), followed by preorder (14 seconds), ORDPATH (25 seconds, due to ID comparison), and Virtual Nodes (3.8 minutes, due to the join overhead).

10. CONCLUSION

In this paper we introduced the BIRD family of tree numbering schemes based on structural summaries that allows to efficiently decide and reconstruct tree relations with simple arithmetic operations. We showed that decision and reconstruction of tree relations is a central building block of most query strategies. We analyzed and compared properties and expressivity of other node identification schemes and identify a trade-off between evaluation time, storage consumption and expressivity, where BIRD appears to be a favourable choice. Finally, we presented the results of extensive tests, proving that BIRD is almost always faster than identification schemes of comparable expressivity (up to two orders of magnitude in extreme cases), while being reasonably small in size.

Future work includes a generalization of the notion of structural summaries in order to further reduce the storage consumption of BIRD IDs. Besides, the update mechanism sketched in this paper need to be elaborated in detail. We also plan to analyze how BIRD can be integrated in constraint-based evaluation of tree queries.

Acknowledgements

The authors would like to thank Gerhard Weikum and Ralf Schenkel at the Max-Planck Institute of Computer Science in Saarbrücken (Germany) for providing them with their XML convertor for the *IMDb* collection, as well as Sebastian Hick at the Centre for Information and Language Processing, Munich (Germany), for helpful comments on the experiments.

11. REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*, pages 141–152, 2002.

[3] R. Baeza and G. Navarro. XQL and proximal nodes. In *Proceedings of the SIGIR Workshop on XML and Information Retrieval*, 2000.

[4] S. Berger, F. Bry, O. Bolzer, T. Furche, D. Olteanu, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: Twin Query Languages for the Semantic Web. In *Proceedings of 3rd International Semantic Web Conference (ISWC2004)*, 2004.

[5] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. W3C Working Draft, October 2004.

[6] S. Boag, D. Chamberlin, , M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3C Working Draft, October 2004.

[7] J.-M. Bremer and M. Gertz. An Efficient XML Node Identification and Indexing Scheme. Technical Report CSE-2003-04, Department of Computer Science, University of California at Davis, 2003.

[8] S.-Y. Chien, Z. Vagena, D. Zhang, and V. J. Tsotras. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 263–274, 2002.

[9] Digital Bibliography And Library Project (DBLP). Available at `dblp.uni-trier.de`. A service offered by the University of Trier, Germany.

[10] P. Dietz and D. Sleator. Two Algorithms for Maintaining Order in a List. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[11] N. Fuhr and K. Großjohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In *Research and Development in Information Retrieval*, pages 172–180, 2001.

[12] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 436–445, 1997.

[13] G. Gottlob and C. Koch. Monadic Datalog and the Expressive Power of Web Information Extraction Languages. *Journal of the ACM*, 51(1):74–113, 2004.

[14] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive Queries over Trees. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS*, pages 189–200, 2004.

[15] T. Grust. Accelerating XPath location steps. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 109–120, 2002.

[16] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, August 2004.

[17] Internet Movie Database (IMDb). Available at `www.imdb.com`.

[18] Initiative for the Evaluation of XML Retrieval (INEX). Available at `inex.is.informatik.uni-duisburg.de:2004`. Organised by the DELOS Network of Excellence for Digital Libraries.

[19] M. Kay. XSL Transformations (XSLT) Version 2.0. W3C Working Draft, November 2004.

[20] W. E. Kimber. HyTime and SGML: Understanding the HyTime HYQ Query Language. Technical Report Version 1.1, IBM Corporation, August 1993.

[21] Y. K. Lee, S.-J. Yoo, K. Yoon, and P. B. Berra. Index structures for structured documents. In *Proceedings of the 1st ACM International Conference on Digital Libraries*, pages 91–99, 1996.

[22] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th VLDB Conference*, pages 361–370, 2001.

[23] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997.

[24] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran. Indexing Semistructured Data. Technical report, Stanford University, Computer Science Department, Database Group, January 1998.

[25] H. Meuss, K. Schulz, and F. Bry. Visual Querying and Exploration of Large Answers in XML Databases with X^2: A Demonstration. In *Proceedings of the 19th International Conference on Database Engineering*

(ICDE), pages 777–779, 2003.

[26] H. Meuss, K. U. Schulz, F. Weigel, S. Leonardi, and F. Bry. Visual Exploration and Retrieval of XML Document Collections with the Generic System X^2. *Journal of Digital Libraries, Special Issue on Information Visualization Interfaces*, October 2004.

[27] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, pages 277–295, 1999.

[28] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 903–908, 2004.

[29] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V. V. Zolotov. Indexing XML Data Stored in a Relational Database. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 1134–1145, 2004.

[30] The Penn Treebank Project. Available at www.cis.upenn.edu/∼treebank/home.html. A service offered by the University of Pennsylvania.

[31] T. Schlieder. ApproXQL: Design and implementation of an approximate pattern matching language for XML. Technical Report B 01-02, Freie Universitat Berlin (Germany), 2001.

[32] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–215, 2002.

[33] F. Weigel, H. Meuss, F. Bry, and K. U. Schulz. Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. In *Proceedings of the 26th European Conference on Information Retrieval (ECIR)*, pages 378–393, 2004.

[34] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 425–436, 2001.

APPENDIX

A. NODE IDENTIFICATION WITH BIRD

Indexing with the BIRD ID scheme goes through three phases: first, all nodes in the document tree are visited once in order to have their label path added to the DataGuide index, if necessary, and to find the maximal number of children of any node with a given label path. The latter information is necessary for weight creation, which takes place in the second phase in a bottom-up traversal of the DataGuide tree created in phase 1. Finally, all document nodes are assigned BIRD IDs based on the weights computed in the previous phase.

The top-level indexing procedure in Listing 1 follows this outline by calling for each phase one of a set of dedicated subroutines to be explained in the following sections. Beforehand the root nodes of both the document tree and the index tree are stored in global variables. Note that the index root is one level higher than the root of the document tree and does not represent any document node. The mapping I is used to obtain access to all index nodes at a specific level in the DataGuide tree during weight creation. The mapping C stores for each index node the maximal child count of all document nodes represented by that index node. Needed for weight creation only, both data structures are discarded after the node identification is finished. The global variable b holds the last assigned BIRD ID.

```
// createBIRD: node identification with BIRD
proc createBIRD ()

  // initialize global variables
  d_r := the root ID of the document tree
  i_r := the root ID of the empty DataGuide tree
  I := an empty map: level ↦ index node
  C := an empty map: index node ↦ child count
  b := −1

  // phase 1: create the DataGuide index (cf. Listing 2)
  call createDataGuide (d_r, i_r)

  // phase 2: create the index node weights (cf. Listing 3)
  call createWeights ()

  // phase 3: create the document node IDs (cf. Listing 4)
  call createIDs ()

end proc
```

Listing 1: Node identification with BIRD.

A.1 DataGuide creation

The procedure *createDataGuide* in Listing 2 recursively traverses the document tree in preorder and updates the DataGuide where necessary. Newly created DataGuide nodes are added to the global mapping I (line 39). Furthermore, each index node is associated with the maximal number of children of any document node represented by that index node. To this end, the global mapping C is updated for each document node being indexed (line 45).

```
21  // createDataGuide: creates the DataGuide, and collects the
22  // maximal child tuple size of each index node
23  // (1st top-down pass through the document tree)
24  // → d: the root of the document subtree to be processed
25  // → i_p: the index node representing the parent of d
26  proc createDataGuide (d: document node, i_p: index node)
27
28     // get the DataGuide node i representing d
29     if i_p has a child with d's label then
30        i := the child of i_p with d's label
31        c := the child count associated with i in C
32
33     // if there is no such node, update the DataGuide
34     else
35        i := a new index node with d's label
36        add i to i_p's children in the DataGuide
37        l := d's level in the document tree
38        I_l := the index nodes associated with l in I
39        I_l := I_l ∪ {i}
40        c := 0
41     end if
42
43     // update the maximal child count for i
44     c := max (c, number of children of d)
45     map i to c in C
46
47     // recursively process the subtree rooted at d
48     for all children d_c of d from left to right do
49        call createDataGuide (d_c, i)
50     end for
51
52  end proc
```

Listing 2: DataGuide creation.

A.2 Weight creation

Both the pre-weights and the final weights from Definition 4.3 are computed by the procedure *createWeights*. The code shown in Listing 3 assumes a balancing degree $s \geq 1$. Index nodes are visited in a levelwise bottom-up iteration to make sure that final weights are already available for all children of any node being weighted. First, the pre-weight of an index node i is computed (lines 62 to 70), as specified in Definition 4.3. While leaves have a fixed pre-weight of 1, for inner index nodes it is computed from the uniform final weight of any of their children and the maximal child count stored in the global mapping C.

The final weight of i (lines 73 to 82) is the maximum of the pre-weights of all nodes which are s-equivalent to i (see Definition 4.1). The set $[i]_s$ is easily computed by navigating the DataGuide. Since all s-equivalent nodes are on the same level by definition, the pre-weights of all nodes in $[i]_s$ are guaranteed to be available when weighting i. All nodes in $[i]_s$ (including i) are assigned the same final weight and then discarded to avoid duplicate computation and weighting of nodes in the same equivalence set (line 80).

```
53  // createWeights: computes the BIRD weight of each
54  // index node (bottom-up pass through the DataGuide)
55  proc createWeights ()
56
57     // visit all index nodes bottom-up
58     for all levels l in I in descending order do
59        I_l := the index nodes associated with l in I
60
61        // create pre-weights
62        for all index nodes i ∈ I_l do
63           if i has children then
64              i_c := any child of i
65              c := the child count associated with i in C
66              i.weight := i_c.weight · (c + 1)
67           else
68              i.weight := 1
69           end if
70        end for
71
72        // create weights for s-equivalent index nodes
73        for all index nodes i ∈ I_l do
74           w := 1
75           for all i_s ∈ [i]_s do
76              w := max (w, i_s.weight)
77           end for
78           for all i_s ∈ [i]_s do
79              i_s.weight := w
80              I_l := I_l \ {i_s}
81           end for
82        end for
83
84     end for
85
86  end proc
```

Listing 3: BIRD weight creation.

A.3 ID creation

```
87  // createIDs: assigns a BIRD ID to each document node
88  // (2nd top-down pass through the document tree)
89  // → d: the root of the document subtree to be processed
90  // → i_p: the index node representing the parent of d
91  proc createIDs (d: document node, i_p: index node)
92
93     // assign an unused BIRD ID to d
94     i := the child of i_p with d's label
95     d.id := smallest multiple n of i.weight with n > b
96     b := d.id
97
98     // recursively process the subtree rooted at d
99     for all children d_c of d from left to right do
100       call createIDs (d_c, i)
101    end for
102
103 end proc
```

Listing 4: BIRD ID creation.

Assigning BIRD IDs based on the weights computed in Listing 3 is straightforward. The procedure *createIDs* in Listing 4 again traverses the document tree in preorder, assigning each document node the smallest free ID which is a multiple of that node's final weight (line 95). The fact that the IDs are created in preorder makes BIRD compatible with the document order.

B. PATH JOIN STRATEGIES

This section explains the three different path join strategies we applied in our experiments for entire tree queries. Recall from Section 9.3 that only the relations $\mathit{Child}^i(m,n)$ and $\mathit{Child}^+(m,n)$ can be expressed in the queries. In the sequel, we assume that nodes in the query tree are matched bottom-up and that there is a means to retrieve document nodes with a given label path and/or textual content from the documents (i.e., in the case of X^2, the relational backend). We use a DataGuide variant called *CADG* [33] for this purpose. The DataGuide also provides the level of any document node being fetched, such that $\mathit{Child}^+(m,n)$ relations in the query actually entail decision of the $\mathit{Child}^i(m,n)$ relation or reconstruction of $\mathit{parent}^i(n)$.

The three strategies differ in how often nodes are fetched, and how two sets of nodes matching a parent and a child query node are joined to find out which node pairs satisfy the $\mathit{Child}^i(m,n)$ relation. For the sake of simplicity, the algorithms given below assume that any match to the child query node has at most one ancestor matching the parent query node. This may not be true for recursive collections or XPath queries involving the `*` node test and the `ancestor` axis. To cope with these cases, all three join procedures listed below are actually called multiple times for the same pair of query nodes, each time joining two sets of nodes containing only nodes with the same label path. This makes sure that no two ancestors of a given node are in the same set being joined.

B.1 ALWAYS

The path join strategy *ALWAYS* applies reconstruction of $\mathit{parent}^i(n)$ to all query nodes. When the procedure *recAlways* is called for a query node q_p and some child node q_c for the first time, the set of matches for q_p is still empty. The ancestors of all matches to the child node q_c are reconstructed (lines 12 to 15 in Listing 5) and used as q_p's set of matches in subsequent calls to *recAlways*.

If q_p has been processed before for some sibling of q_c, matches to q_p are already available in the set E_p (otherwise the query would have been rejected as unsatisfiable). The **loop** in the **else** branch in Listing 5 reconstructs the ancestor at the level of q_p of any match e_c to q_c (line 21) and tests whether it is a member of E_p. If so, subsequent matches to q_c are checked by decision (line 25) until a match is found which is not a descendant of the ancestor just reconstructed for e_c. This technique saves reconstruction time for nodes whose ancestor may already be available and works for both schemes in preorder (e.g., BIRD and ORDPATH) and breadth-first schemes (like Virtual Nodes). If the ancestor reconstructed for e_c is not a member of E_p, then e_c is discarded.

```
1  // recAlways: using reconstruction for all child query nodes
2  // → q_c: the child query node to be joined
3  // → q_p: the parent query node to be joined
4  proc recAlways (q_c: query node, q_p: query node)
5
6    // get matches for q_c and q_p
7    E_c := matches retrieved for q_c, in ascending order
8    E_p := matches retrieved for q_p, in ascending order
9
10   // first child query node
11   if E_p = ∅ then
12     for all e_c ∈ E_c do
13       e_p := e_c's ancestor matching q_p
14       E_p := E_p ∪ {e_p}
15     end for
16
17   // subsequent child query nodes
18   else
19     e_c := the first member of E_c
20     loop
21       e_p := e_c's ancestor matching q_p
22       if e_p ∈ E_p then
23         repeat
24           e_c := the next member of E_c
25         until e_c is not a descendant of e_p
26       else
27         E_c := E_c \ {e_c}
28         e_c := the next member of E_c
29       end if
30     end loop
31   end if
32
33 end proc
```

Listing 5: Path join strategy *ALWAYS*.

B.2 FIRST

The next strategy, *FIRST*, restricts reconstruction to the first child of a query node q_p, which is processed as for *ALWAYS* (see Section B.1). Matches to any subsequent child node q_c are joined with matches to q_p in a nested loop, as shown in the **else** branch in Listing 6. The outer loop iterates in ascending ID order through the set E_p of matches to q_p, which is advantageous when joining restricted sets of ancestor matches with large sets of descendant matches. This explains why most ID schemes perform better with *FIRST* than with *ALWAYS* (see Section 9.3).

For any $e_p \in E_p$, the matches to q_c which are smaller than e_p are discarded since they are definitely not part of e_p's subtree (neither for preorder nor for breadth-first schemes). Matches greater than e_p undergo decision of the $\mathit{Child}^i(m,n)$ relation with e_p (not reconstruction as in the procedure *recAlways*). Note that for ID schemes following document order (such as BIRD, ORDPATH and preorder), all descendants of e_p, if any, appear in a contiguous sequence directly after the greatest match to q_c which is smaller than e_p. These descendants are identified by repeated decision of $\mathit{Child}^i(m,n)$ (line 58). Under certain conditions this also applies to the PID scheme (see Result 4 in Section 9.3). By contrast, breadth-first schemes like Virtual Nodes must check all descendant candidates greater than e_p (line 66), including breadth-first successors of nodes outside e_p's subtree, because a breadth-first traversal of a tree may enter and leave the subtree of any node more than once. As a minor optimization compatible with breadth-first IDs, we mark the first match to q_c right of e_p's subtree as the starting point for the next iteration of the outer loop, since nodes between e_p and that node (in breadth-first order) are guaranteed to be descendants of e_p (and hence not of e_p's breadth-first successor in E_p, which is itself not a descendant of e_p as explained above). But still Virtual Nodes experiences a considerable overhead when joining large sets of descendant matches using *recFirst*, as discussed in Section 9.3.

```
34 // recFirst: using reconstruction for the first child query node
35 // → q_c: the child query node to be joined
36 // → q_p: the parent query node to be joined
37 proc recFirst (q_c: query node, q_p: query node)
38
39   // get matches for q_c and q_p
40   E_c := matches retrieved for q_c, in ascending order
41   E_p := matches retrieved for q_p, in ascending order
42
43   // first child query node
44   if E_p = ∅ then
45     ... (cf. recAlways, lines 12 to 15 in Listing 5)
46
47   // subsequent child query nodes
48   else
49     e_c := the first member of E_c
50     for all e_p ∈ E_p do
51       while e_c < e_p do
52         E_c := E_c \ {e_c}
53         e_c := the next member of E_c
54       end while
55
56       // BIRD, ORDPATH, preorder
57       if IDs are assigned in document order then
58         while e_c is a descendant of q_p do
59           e_c := the next member of E_c
60         end while
61
62       // Virtual Nodes
63       else
64         pos := |E_c|
65         loop
66           if e_c is not a descendant of q_p then
67             E_c := E_c \ {e_c}
68             if pos = |E_c| then
69               pos := the position of e_c in E_c
70             end if
71           end if
72           e_c := the next member of E_c
73         end loop
74         e_c := the member of E_c at position pos
75       end if
76
77     end for
78   end if
79
80 end proc
```

Listing 6: Path join strategy *FIRST*.

B.3 NEVER

The third path join strategy, *NEVER*, eliminates the remaining reconstruction step for the first query child node from *recFirst*. The resulting procedure *recNever*, given in Listing 7, processes matches to all children of a query node q_p alike – by deciding $\mathit{Child}^i(m, n)$ as in *recFirst* – after the matches for q_p have been fetched in the first place (line 12). The experiments in Section 9.3 discourage the use of this strategy, not only because of the additional fetching cost, but mainly due to the extra join for the first query child node.

```
1  // recNever: using decision for all child query nodes
2  // → q_c: the child query node to be joined
3  // → q_p: the parent query node to be joined
4  proc recNever (q_c: query node, q_p: query node)
5
6    // get matches for q_c and q_p
7    E_c := matches retrieved for q_c, in ascending order
8    E_p := matches retrieved for q_p, in ascending order
9
10   // first child query node
11   if E_p = ∅ then
12     E_p := all elements matching q_p's path
13   end if
14
15   // all child query nodes
16   ... (cf. recFirst, lines 49 to 77 in Listing 6)
17
18 end proc
```

Listing 7: Path join strategy *NEVER*.