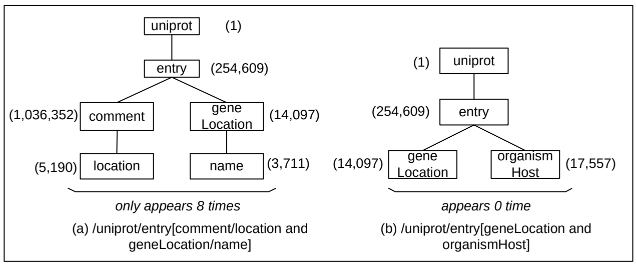
Efficient Evaluation of Nearest Common Ancestor in XML Twig Queries Using Tree-Unaware RDBMS

Klarinda G. Widjanarko **Erwin Leonardi** **Sourav S Bhowmick**

School of Computer Engineering
Nanyang Technological University, Singapore
lerwin|assourav@ntu.edu.sg

Abstract

Finding all occurrences of a twig pattern in a database is a core operation in XML query processing. Recent study showed that *tree-aware* relational framework significantly outperform *tree-unaware* approaches in evaluating structural relationships in XML twig queries. In this paper, we present an efficient strategy to evaluate a specific class of structural relationship called NCA-*twiglet* in a tree-unaware relational environment. Informally, NCA-twiglet is a subtree in a twig pattern where all nodes have the same nearest common ancestor (the root of NCA-twiglet). We focus on NCA-twiglets having parent-child relationships. Our scheme is build on top of our SUCXENT++ system. We show that by exploiting the encoding scheme of SUCXENT++ we can reduce useless structural comparisons in order to evaluate NCA-twiglets. Through a comprehensive experiment, we show that our approach is not only more scalable but also performs better than a representative tree-unaware approach on all benchmark queries with the highest observed gain factors being 352. Additionally, our approach reduces significantly the performance gap between tree-aware and tree-unaware approaches.



```
Document (DocId, Name)
Path (PathId, PathExp)
PathValue(DocId, DeweyOrderSum,
   PathId, BranchOrder, LeafOrder,
   SiblingSum, LeafValue)
Attribute (DocId, LeafOrder, PathId,
   LeafRValue)
DocumentRValue (DocId, Level, RValue)
```

(c) Schema of SUCXENT++

Figure 1: Example of twig queries and SUCXENT++ schema.

1 Introduction

Finding all occurrences of a twig pattern in a database is a core operation in XML query processing, both in relational implementations of XML databases [3, 7–9, 13, 16, 17, 21, 22], and in native XML databases [1, 4, 5, 11, 12, 14]. Consequently, in the past few years, many algorithms have been proposed to match twig patterns. These approaches (i) first develop a labeling scheme to capture the structural information of XML documents, and then (ii) perform twig pattern matching based on the labels alone without traversing the original XML documents.

For the first sub-problem of designing appropriate labeling scheme, various methods have been proposed that are based on tree-traversal order [1, 9, 10], region encoding [4, 22], path expressions [12, 17] or prime numbers [19]. By applying these labeling schemes, one can determine the structural relationship between two elements in XML documents from their labels alone. The goal of second sub-problem of matching twig patterns is to devise efficient techniques for structural relationship matching. In general, structural relationship in a twig query may be categorized in two different classes: (a) NCA-*twiglet*, and (b) *path expression*. Given a query twig pattern $Q = (V, E)$, the *nearest common ancestor* (denoted as NCA) of two nodes $x \in V$, $y \in V$ is the common ancestor of x and y whose distance to x (and to y) is smaller than the distance to x of any other common ancestor of x and y. The twig substructure rooted at such NCA node is called NCA-*twiglet*. For example, consider the twig query in Figure 1(a). The twig structure rooted at `entry` node is an example of NCA-twiglet as it is the NCA of `location` and `name` nodes. On the other hand, *path expression* is a linear structural constraint. For example, `/uniprot/entry` is a path expression in Figure 1(a). In this paper, we focus on *efficient evaluation of* NCA-*twiglets* in a *relational implementation* of XML databases.

In literature, evaluation strategies of twig pattern matching can be broadly classified into the following three types: (a) *binary-structure matching*, (b) *holistic twig pattern matching*, and (c) *string matching*. In the *binary-structure matching* approach, the twig pattern is first decomposed into a set of binary (parent-child and ancestor-descendant) relationships between pairs of nodes. Then, the twig pattern can be matched by matching each of the binary structural relationships against the

XML database, and "stitching" together these basic matches [1, 8, 10, 11, 16, 22]. In the *holistic twig pattern matching* approach, the twig query is decomposed into its corresponding path components and each decomposed path component is matched against the XML database. Next, the results of each of the query's path expressions are joined to form the result to the original twig query [4, 6, 12]. Lastly, approaches like ViST [18] and PRIX [14] are based on *string matching* method and transform both XML data and queries into sequences and answer XML queries through subsequence matching.

One of the key challenge in NCA-twiglets evaluation (as well as twig pattern matching in general) is to develop techniques that can reduce generation of large intermediate results. For instance, the binary-structure matching approaches may introduce very large intermediate results. Consider the sample document fragment from UniProtKB/ Swiss-Prot and the NCA-twiglet in Figures 2 and 1(a), respectively. The path match ($e2$, $g2$, $n1$) for path `entry/geneLocation/name` does not lead to any final result since there is no `comment/location` path under $e2$. Note that this problem is exacerbated for queries that are *high-selective*[1] but each path in the query is *low-selective*. For example, the query in Figure 1(a) is very selective as it returns only 8 results. However, all the paths are low-selective. Note that the number associated with each node in the queries in Figure 1 represents the number of occurrences of the path from the root node to the specific node in the XML database. Similarly, the query in Figure 1(b) is a high-selective query as it does not return any results although all the paths are low-selective. To solve this problem, the *holistic twig pattern matching* has been developed in order to minimize the intermediate results. In this approach, only those root-to-leaf path matches that will be in the final twig results are enumerated. However, when the twig query contains parent-child relationships, these solutions may still generate large numbers of useless matches [5]. Hence, in this paper we focus our attention on NCA-twiglets containing *parent-child relationship* and are components of *high-selective queries having low-selective paths*.

The rest of the paper is organized as follows. Section 2 gives the framework of our work and highlights the key contributions. Next, in Section 3, we formally introduce the notion of NCA-twiglet and present the algorithmic details about how NCA-twiglets are efficiently matched in our relational framework. Results of the performance evaluation are presented in Section 4. Section 5 briefly discusses related research in twig query processing in a relational environment. Finally, the last section concludes the paper.

2 Framework and Contributions

The problem of efficiently finding NCAs in a general tree has been studied extensively over the last three decades in an online and offline setting, and in various

[1]Throughout the paper, we use "high-selective" or "very selective" to characterize a twig query with few results and "low-selective" to characterize a query with many results.

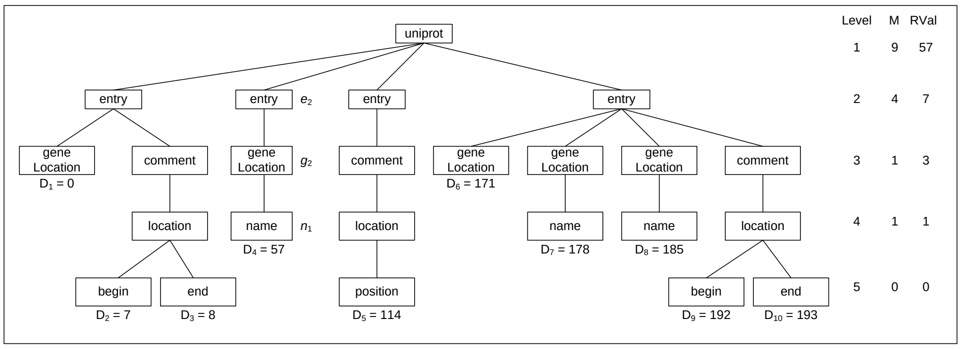


Figure 2: Example of XML data.

models of computation [2]. Most of these approaches work using some mapping of the tree to a completely balanced tree, thereby exploiting the fact that for completely binary trees the problem is easier. Different algorithms differ by the way they do the mapping. However, these techniques cannot be directly used in the XML context for the following reasons. (i) Although the labels of the nodes used in some of the NCA algorithms can compute the label of NCA in constant time [2], they are not generic enough to efficiently support evaluation of various XPATH axes. Indeed, as mentioned earlier, the XML community has resorted to devising novel labeling schemes to support efficient twig matching. (ii) Due to the nature of XML data, the mapping of an XML tree to a completely binary tree may not be an efficient technique for processing different types of XPATH axes. Consequently, the research community has proposed various techniques on native and relational frameworks to evaluate NCA-twiglets and twig queries in general.

2.1 Relational Approaches for Twig Query Processing and our Contributions

While a variety of approaches have been proposed in the literature to process twig queries in native XML storage [4,5,11,12,14], finding ways to evaluate such queries in relational environment has gained significant momentum in recent years. Specifically, there has been a host of work [3,4,7,9,10,22] on enabling relational databases to be *tree-aware* by invading the database kernel to implement XML support. On the other side of the spectrum, some completely jettison the approach of internal modification of the RDBMS for twig query processing and resort to alternative *tree-unaware* approach [8,13,15,16] where the database kernel is not modified in order to process XML queries.

While the state-of-the-art tree-aware approaches are certainly innovative and powerful, we have found that these strategies are not directly applicable to relational databases. The RDBMS systems need to augment their suite of query processing strategies by incorporating special purpose external index systems, algorithms and storage schemes to perform efficient XML query processing. Therefore, the integration of external modules into commercial relational databases could be complex and inefficient. On the other hand, there are considerable benefits in tree-unaware approaches with respect to portability as they do not invade the database kernel. Consequently, they can easily be incorporated in an off-the-shelf RDBMS. However, one of the key stumbling block for the acceptance of tree-unaware approaches has been query performance. Worse, this shortcoming is exacerbated when the number of relations participating in the query as well as their sizes are scaled upwards. In fact, recent results reveal that the tree-aware approaches appear scalable and, in particular, perform orders of magnitude faster than several tree-unaware approaches [3, 9].

We observe that the superiority of tree-aware approaches is primarily due to the following key factors.

- Most of the tree-aware approaches are based on the tree-traversal order and region encoding labeling schemes to label nodes within an XML document. However, in relational environment, efficient evaluation of containment queries using these labeling schemes *requires* modification of the SQL query optimizer [9, 22]. Consequently, existing tree-unaware approaches based on these labeling schemes suffer from performance degradation.

- In tree-unaware approaches, the relational kernel is unaware of the fact that it is operating on tree-shaped data rather than any arbitrary data points. This results in unnecessary computation as well as generation of poor query plans when processing tree-structured queries.

- MonetDB [3], an innovative and powerful tree-aware approach, is a main memory database system, which did not consider all the overhead that would have been incurred in a disk-bound database system.

In this paper, we explore the challenging problem of *efficient evaluation of* NCA-*twiglets* in a *tree-unaware* relational framework.

In summary, the main contributions of this paper are as follows. (a) Based on a novel labeling scheme, in Section 3, we present an efficient algorithm for determining nearest common ancestor (NCA) of two elements in an XML document. Our strategy accesses much fewer nodes compared to existing state-of-the-art tree-unaware approaches in order to evaluate NCA-twiglets. Importantly, our proposed algorithm is capable of working with any off-the-shelf RDBMS without any internal modification. (b) Through an extensive experimental study in Section 4, we show that our approach significantly outperforms a state-of-the-art tree-unaware scheme (Global-Order [17]) for evaluating benchmark NCA-twiglets. Additionally, our approach reduces significantly the performance gap between tree-aware (MONETDB [3]) and tree-unaware approaches.

2.2 Overview of SUCXENT++ Approach

Our approach for NCA-twiglet evaluation is based on the SUCXENT++ system [15]. It is a tree-unaware approach and is designed primarily for query-mostly workloads. Here, we briefly review the storage scheme of SUCXENT++ which we shall be using in our subsequent discussion. The SUCXENT++ schema is shown in Figure 1(c). Document stores the document identifier DocId and the name Name of a given input XML document T. We associate each distinct (root-to-leaf) path appearing in T, namely PathExp, with an identifier PathId and store this information in Path table. For each element leaf node n in T, we shall create a tuple in the PathValue table.

SUCXENT++ uses a novel labeling scheme that *does not* require labeling of internal nodes in the XML tree. For each leaf node it stores four additional attributes namely LeafOrder, BranchOrder, DeweyOrderSum and SiblingSum. Also, it encodes each level of the XML tree with an attribute called RValue. We now elaborate on the semantics of these attributes. Given two leaf nodes n_1 and n_2, n_1.LeafOrder $<$ n_2.LeafOrder *iff* n_1 precedes n_2. LeafOrder of the first leaf node in T is 1 and n_2.LeafOrder $=$ n_1.LeafOrder+1 *iff* n_1 is a leaf node immediately preceding n_2. Given two leaf nodes n_1 and n_2 where n_1.LeafOrder+1 $=$ n_2.LeafOrder, n_2.BranchOrder is the level of the NCA of n_1 and n_2. The data value of n is stored in n.LeafValue.

To discuss DeweyOrderSum, SiblingSum and RValue, we introduce some auxiliary definitions. Consider a sequence of leaf nodes C: $\langle n_1, n_2, n_3, \ldots, n_r \rangle$ in T. Then, C is a *k-consecutive leaf nodes* of T *iff* (a) n_i.BranchOrder $\geq k$ for all $i \in [1,r]$; (b) If n_1.LeafOrder > 1, then n_0.BranchOrder $< k$ where n_0.LeafOrder+1 $=$ n_1.LeafOrder; and (c) If n_r is not the last leaf node in T, then n_{r+1}.BranchOrder $<$ k where n_r.LeafOrder+1 $=$ n_{r+1}.LeafOrder. A sequence C is called a *maximal k-consecutive leaf nodes* of T, denoted as M_k, if there does not exist a k-consecutive leaf nodes C' and $|C|<|C'|$.

Let L_{max} be the largest level of T. The RValue of level ℓ, denoted as R_ℓ, is defined as follows: (i) If $\ell = L_{max} - 1$ then $R_\ell = 1$; (ii) If $0 < \ell < L_{max} - 1$ then $R_\ell = 2R_{\ell+1} \times |M_{\ell+1}| + 1$. For example, consider the XML tree shown in Figure 2. $L_{max} = 5$. The values of $|M_1|$, $|M_2|$, $|M_3|$, and $|M_4|$ are 9, 4, 1, and 1, respectively. Then, $R_4 = 1$, $R_3 = 3$, $R_2 = 2 \times 3 \times |M_3| + 1 = 7$, and $R_1 = 2 \times 7 \times |M_2| + 1 = 57$. Note that due to facilitate evaluation of XPATH queries, the RValue attribute in DocumentRValue stores $\frac{R_\ell - 1}{2} + 1$ instead of R_ℓ.

DeweyOrderSum is used to encode a node's order information together with its ancestors' order information using a single value. Consider a leaf node n at level ℓ in T. Ord$(n,k) = i$ *iff* a is either an ancestor of n or n itself; k is the level of a; and a is the i-th child of its parent. DeweyOrderSum of n, n.DeweyOrderSum, is defined as $\sum_{j=2}^{\ell} \Phi(j)$ where $\Phi(j)$=[Ord(n,j)-1]$\times R_{j-1}$. For example, consider the rightmost `name` node in Figure 2 which has a Dewey path "1.4.3.1". DeweyOrderSum of this node is: n.DeweyOrderSum $= (Ord(n,2) - 1) \times R_1 + (Ord(n,3) - 1) \times R_2 + (Ord(n,4) - 1) \times R_3 = 3 \times 57 + 2 \times 7 + 0 \times 3 = 185$. Note that DeweyOrderSum is not sufficient to compute position-based predicates with `QName` name tests, *e.g.*, entry[2]. Hence, the SiblingSum attribute is introduced to the PathValue table.

SiblingSum encodes the local order of nodes which are with the same tag name of n, namely same-tag-sibling order. For example, consider the children of the fourth entry element in Figure 2. The local orders of the three geneLocation and the comment nodes are 1, 2, 3 and 4, respectively. On the other hand, the same-tag-sibling order of these nodes are 1, 2, 3, and 1, respectively. Formally, let $\text{Sibling}(n, k) = i$ *iff* a is either an ancestor of n or n itself; k is the level of a; and the i-th τ-child of its parent (τ is the tag name of a). SiblingSum of n, n.SiblingSum, is $\sum_{j=2}^{\ell} \Psi(j)$ where $\Psi(j) = [\text{Sibling}(n, j)\text{-}1] \times R_{j-1}$.

To evaluate non-leaf nodes, we define the *representative leaf node* of a non-leaf node n to be its first descendant leaf node. Note that the BranchOrder attribute records the level of the NCA of two consecutive leaf nodes. Let C be the sequence of descendant leaf nodes of n and n_1 be the first node in C. We know that the NCA of any two consecutive nodes in C is also a descendant of node n. This implies (a) except n_1, BranchOrder of a node in C is at least the level of node n and (b) the NCA of n_1 and its immediately preceding leaf node is not a descendant of node n. Therefore, BranchOrder of n_1 is always smaller than the level of n. The reader may refer to [15] for details on how these attributes are used to efficiently evaluate *ordered* XPATH axes.

3 Evaluation of NCA-Twiglets

In this section, we present the evaluation strategy of NCA-twiglets in SUCXENT++. We begin by formally introducing the notion of NCA-twiglet.

3.1 Data Model and NCA-Twiglet

We model XML documents as ordered trees. In our model we ignore comments, processing instructions and namespaces. Queries in XML query languages make use of twig patterns to match relevant portions of data in an XML database. The twig pattern node may be an element tag, a text value or a wildcard "*". We distinguish between query and data nodes by using the term "node" to refer to a query node and the term "element" to refer to a data element in a document. In this paper, we focus only on parent-child relationships between the nodes in the twig pattern. Recall that existing holistic twig pattern matching approaches achieve optimality for ancestor-descendant relationships but may generate large numbers of useless matches when the twig query contains parent-child relations [5]. We now formally define NCA-*twiglet*.

Definition 1 (NCA-Twiglet) *Given a query twig pattern $Q = \langle V, E\rangle$, a* NCA-***Twiglet*** *$N = \langle V_n, E_n, \Re\rangle$ in Q, denoted as $N \prec Q$, is a subtree in Q rooted at node $\Re \in V$ such that (a) $V_n \subset V$ is a set of nodes whose nearest common ancestor is $\Re$, and (b) $E_n \subseteq E$.*

A NCA-twiglet consists of a collection of *rooted path* patterns, where a *rooted path* pattern (RP) is a root-to-leaf path in the NCA-twiglet. The level of the root $\mathfrak{R}$ is called NCA-*level*. For example, the NCA-twiglet in Figure 1(a) consists of the rooted paths `entry/comment/location` and `entry/geneLocation/name`. Note that each of the above RPs has a parent-child relationship between the nodes. The path from $Root(Q)$ to $\mathfrak{R}$ is called the *reachability path* of N. For instance, in the above example `/uniprot/entry` is the reachability path.

Given a NCA-twiglet $N \prec Q$ and an XML document D, a match of N in D is identified by a mapping from the nodes in N to the elements in D, such that: (a) the query node predicates are satisfied by the corresponding database elements, wherein wildcard "*" can match any single tag; (b) the parent-child relationship between query nodes are satisfied by the corresponding database elements; and (c) the reachability path of N is satisfied by the database elements. Next, we present our approach to match N in D.

3.2 NCA-Twiglet Matching

Recall that in SUCXENT++ we each root-to-leaf path of an XML document is encoded with the attributes LeafOrder, BranchOrder, DeweyOrderSum, and SiblingSum. Additionally each level of the XML tree is associated with a RValue. Hence, given the NCA-twiglet $N \prec Q$ and document D, our goal is to use these attributes to efficiently determine those root-to-leaf paths that satisfies N. We begin by formally introducing some lemmas to facilitate efficient evaluation of N.

Lemma 1 $\sum_{j=k}^{\ell} \Phi(j) \leq \frac{R_{k-2}-1}{2}$ *where* $\Phi(j)$ *=[*$\mathsf{Ord}(n, j)$*-1]*$\times R_{j-1}$, $k \in$ *(2,*ℓ*] and n is a leaf node in an* XML *document at level* ℓ. □

Proof 1 Let M_j be the maximum consecutive j-consecutive leaf node set. Then, the maximum number of consecutive leaf nodes with BranchOrder $\geq j$ is $|M_j|$. Given any node at level j, all but one of the descendants of this node has BranchOrder $\geq j$. Hence, any node at level j has at most $|M_j| + 1$ descendant leaf nodes.

In SUCXENT++, the first sibling has LocalOrder equal to 1. Given $\mathsf{Ord}(n,t)$ of n at each level $t \in [k, \ell]$, any ancestor of n at level $t-1$ has at least [$\mathsf{Ord}(n,t)$-1] that are not n nor n's ancestor. Each of these nodes either is a leaf node, or has at least one descendant leaf node. Hence, an ancestor of n at level $t-1$ has, excluding n, at least [$\mathsf{Ord}(n,t)$-1] descendant leaf nodes, all of which are descendants of the n's ancestor at level $k-1$ and are not descendants of any n's ancestor at level greater than $t-1$. Therefore, there is a node at level $k-1$ with at least $(\sum_{t=k}^{\ell}[\mathsf{Ord}(n,t)\text{-}1])+1$ descendant leaf nodes (including n). This implies that $\sum_{t=k}^{\ell}[\mathsf{Ord}(n,t)\text{-}1] \leq |M_{k-1}|$. Therefore,

$$\begin{aligned}
\sum_{j=k}^{\ell} \Phi(j) &= \sum_{j=k}^{\ell}[\mathsf{Ord}(n,j)-1]\times R_{j-1} \\
&\le \sum_{j=k}^{\ell}[\mathsf{Ord}(n,j)-1]\times R_{k-1} \\
&\le |M_{k-1}|\times R_{k-1} \\
&\le \frac{R_{k-2}-1}{2}
\end{aligned}$$

∎

Lemma 2 *Let n_1 and n_2 be two leaf nodes in an* XML *document. If* $|n_1.\mathsf{DeweyOrderSum}$ - $n_2.\mathsf{DeweyOrderSum}| < \frac{R_\ell - 1}{2} + 1$ *then the level of the nearest common ancestor is greater than ℓ.* □

Proof 2 Assume the level of the nearest common ancestor of n_1 and n_2 is $\le \ell$, then $|n_1.\mathsf{DeweyOrderSum}$ - $n_2.\mathsf{DeweyOrderSum}| < (R_\ell$ - 1)/2 + 1. Let ℓ_1 be the level of n_1 in X and ℓ_2 be the level of n_2 in X.

When level of nearest common ancestor is ℓ: In this case, $\Phi_1(j) - \Phi_2(j) = 0$ for all $j < \ell + 1$ and $\Phi_1(j) - \Phi_2(j) \neq 0$ for $j \ge \ell + 1$. Consider the following cases.

Case $n_1.\mathsf{LeafOrder} > n_2.\mathsf{LeafOrder}$**:**

$$\begin{aligned}
\Delta &= n_1.\mathsf{DeweyOrderSum} - n_2.\mathsf{DeweyOrderSum} \\
&= \sum_{j=\ell+1}^{\ell_1}\Phi_1(j) - \sum_{j=\ell+1}^{\ell_2}\Phi_2(j) \\
&= [\mathsf{Ord}(n_1,\ell+1)-1]\times R_\ell - [\mathsf{Ord}(n_2,\ell+1)-1]\times R_\ell + \\
&\quad \sum_{j=\ell+2}^{\ell_1}\Phi_1(j) - \sum_{j=\ell+2}^{\ell_2}\Phi_2(j) \qquad (1)
\end{aligned}$$

Since, $\mathsf{Ord}(n_1,\ell+1) \neq \mathsf{Ord}(n_2,\ell+1)$ and $\mathsf{Ord}(n_1,\ell+1) > \mathsf{Ord}(n_2,\ell+1)$, the above equation satisfies the following:

$$\begin{aligned}
\Delta &\ge R_\ell + \sum_{j=\ell+2}^{\ell_1}\Phi_1(j) - \sum_{j=\ell+2}^{\ell_2}\Phi_2(j) \\
&\ge R_\ell - \frac{R'_\ell - 1}{2} \quad (\textit{From Lemma } 1) \\
&\ge \frac{R_\ell - 1}{2} + 1
\end{aligned}$$

Case $n_1.\mathsf{LeafOrder} < n_2.\mathsf{LeafOrder}$**:**

Since in this case, $\mathsf{Ord}(n_1,\ell{+}1) \neq \mathsf{Ord}(n_2,\ell{+}1)$ and $\mathsf{Ord}(n_1,\ell{+}1) < \mathsf{Ord}(n_2,\ell{+}2)$, Equation 1 satisfies the following:

$$\begin{aligned}
\Delta &\leq -R_\ell + \sum_{j=\ell+2}^{\ell_1} \Phi_1(j) - \sum_{j=\ell+2}^{\ell_2} \Phi_2(j) \\
&\leq -R_\ell + \frac{R_\ell - 1}{2} \quad (\mathit{From\ Lemma}\ 1) \\
&\leq -(\frac{R_\ell - 1}{2} + 1)
\end{aligned}$$

Therefore,

$$|\Delta| \geq (\frac{R_\ell - 1}{2} + 1) \quad (\mathit{contradiction})$$

When level of nearest common ancestor is less than ℓ: Let level of nearest common ancestor be k. Then,
Case n_1.LeafOrder $>$ n_2.LeafOrder**:**

$$\begin{aligned}
\Delta &\geq \frac{R_k - 1}{2} + 1 \quad (\mathit{Shown\ to\ be\ true\ above}) \\
&> (\frac{R_\ell - 1}{2} + 1) \quad \mathit{since}\ k < \ell\ (\mathit{contradiction})
\end{aligned}$$

Case n_1.LeafOrder $<$ n_2.LeafOrder**:**

$$\begin{aligned}
|\Delta| &\geq \frac{R_k - 1}{2} + 1 \\
&> (\frac{R_\ell - 1}{2} + 1) \quad \mathit{since}\ k < \ell\ (\mathit{contradiction})
\end{aligned}$$

Hence, nodes n_1 and n_2 cannot have a nearest common ancestor at level lesser than or equal to ℓ. The level of nearest common ancestor must be greater than ℓ. ∎

Lemma 3 *Let n_1 and n_2 be two leaf nodes in an* XML *document. If* $|n_1$.DeweyOrderSum - n_2.DeweyOrderSum$| \geq \frac{R_\ell - 1}{2} + 1$ *then the level of the nearest common ancestor is equal to or smaller than ℓ.* □

Proof 3 Assume the level of the nearest common ancestor of n_1 and n_2 is $> \ell$, then $|n_1$.DeweyOrderSum - n_2.DeweyOrderSum$| \geq (R_\ell$ - $1)/2 + 1$. Let ℓ_1 be the level of n_1 in X and ℓ_2 be the level of n_2 in X. Let $k > l$ be the level of the nearest common ancestor. Therefore $\Phi_1(j) - \Phi_2(j) = 0$ for all $j < k + 1$ and $\Phi_1(j) - \Phi_2(j) \neq 0$ for $j \geq k + 1$.

Case n_1.LeafOrder $>$ n_2.LeafOrder**:**

$$\begin{aligned}|\Delta| &= |n_1.\text{DeweyOrderSum} - n_2.\text{DeweyOrderSum}| \\ &= \sum_{j=k+1}^{\ell_1} \Phi_1(j) - \sum_{j=k+1}^{\ell_2} \Phi_2(j) \\ &\leq \sum_{j=k+1}^{\ell_1} \Phi_1(j)\end{aligned}$$

Case n_1.LeafOrder $<$ n_2.LeafOrder**:**

$$\begin{aligned}|\Delta| &= -\sum_{j=k+1}^{\ell_1} \Phi_1(j) + \sum_{j=k+1}^{\ell_2} \Phi_2(j) \\ &\leq \sum_{j=k+1}^{\ell_2} \Phi_2(j)\end{aligned}$$

Based on Lemma 1:

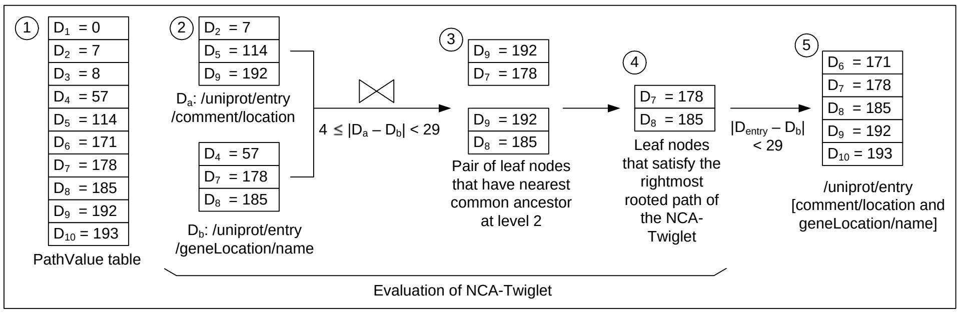
$$\begin{aligned}|\Delta| &\leq \frac{R_{k-1} - 1}{2} \\ &\leq \frac{R_\ell - 1}{2} \\ &< \frac{R_\ell - 1}{2} + 1 \ (contradiction)\end{aligned}$$

∎

Combining Lemma 2 and Lemma 3 above, we can find the exact level of the NCA.

Theorem 1 *Let* n_1 *and* n_2 *be two leaf nodes in an* XML *document. If* $\frac{R_{\ell+1}-1}{2} + 1 \leq |n_1.\text{DeweyOrderSum} - n_2.\text{DeweyOrderSum}| < \frac{R_\ell - 1}{2} + 1$ *then the level of the nearest common ancestor of* n_1 *and* n_2 *is* $\ell + 1$. □

Let us illustrate with an example the above lemmas and theorem. Consider the last leaf node in Figure 2. The DeweyOrderSum of this node is 193. Let D_1 be the DeweyOrderSum of leaf nodes that have NCA at level 2. Using the above theorem, D_1 falls within the following range: $(R_2 - 1)/2 + 1 \leq |D_1 - 193| < (R_1 - 1)/2 + 1 \Rightarrow 4 \leq |D_1 - 193| < 29$ which returns the sixth, seventh, and eighth leaf nodes (DeweyOrderSums are 171, 178, and 185, respectively). Let D_2 be the DeweyOrderSum of leaf nodes that have NCA at level 4. Then D_2 falls within the following range: $(R_4 - 1)/2 + 1 \leq |D_2 - 193| < (R_3 - 1)/2 + 1 \Rightarrow 1 \leq |D_2 - 193| < 2$ which returns the ninth leaf node (DeweyOrderSum is 192). Now let say we want to get the leaf nodes that have NCA at level 2 or deeper and let



1
D1 = 0
D2 = 7
D3 = 8
D4 = 57
D5 = 114
D6 = 171
D7 = 178
D8 = 185
D9 = 192
D10 = 193
PathValue table
2
D2 = 7
D5 = 114
D9 = 192
Da: /uniprot/entry /comment/location
D4 = 57
D7 = 178
D8 = 185
Db: /uniprot/entry /geneLocation/name
4 ≤ |Da – Db| < 29
3
D9 = 192
D7 = 178
D9 = 192
D8 = 185
Pair of leaf nodes that have nearest common ancestor at level 2
4
D7 = 178
D8 = 185
Leaf nodes that satisfy the rightmost rooted path of the NCA-Twiglet
|Dentry – Db| < 29
5
D6 = 171
D7 = 178
D8 = 185
D9 = 192
D10 = 193
/uniprot/entry [comment/location and geneLocation/name]
Evaluation of NCA-Twiglet

Figure 3: An example of NCA-twiglet evaluation.

D_3 be the DeweyOrderSum of these nodes. D_3 falls within the following range: $|D_3 - 193| < (R_1 - 1)/2 + 1 \Rightarrow |D_3 - 193| < 29$ which returns the last five leaf nodes.

We now illustrate Theorem 1 further in the context of a twig query. Consider the query in Figure 1(a) and the fragment of the PathValue table in Figure 3 (Step 1). Note that for clarity, we only show the DeweyOrderSums of the root-to-leaf paths in the PathValue table. Let D_a be DeweyOrderSum of the representative leaf nodes satisfying `/uniprot/entry/comment/location` (second, fifth, and ninth leaf nodes) and D_b be DeweyOrderSum of the representative leaf nodes satisfying `/uniprot/entry/geneLocation/name` (fourth, seventh, and eighth leaf nodes). This is illustrated in step 2 of Figure 3. From the query we know that D_a and D_b have NCA at level 2 (`/uniprot/entry` level). Hence, based on Theorem 1 we can find pairs of (`location`,`name`) nodes which have NCA at level 2. D_a and D_b fall on the following range: $(R_2 - 1)/2 + 1 \leq |D_a - D_b| < (R_1 - 1)/2 + 1 \Rightarrow 4 \leq |D_a - D_b| < 29$ which return the (seventh, ninth) and (eighth, ninth) leaf nodes pairs (Step 3 of Figure 3). We can easily return the `entry` subtree by applying Lemma 2 on any one of these pairs (Steps 4 and 5 of Figure 3). As we are only interested in finding subtrees that matches a specific NCA-twiglet, in this paper we focus on the Steps 1 to 4 of Figure 3. That is, the retrieval of all descendants for complete subtree construction (Step 5) is beyond the scope of this paper. Note that since from the XPATH we know that D_a and D_b can not have NCA at level greater than 2, we only need to use Lemma 2 for matching NCA-twiglets. Observe that the above approach can reduce unnecessary comparison as we do not need to find the grandparent of `location` and `name` nodes. We can determine the NCA directly by using the DeweyOrderSum and RValue attributes.

3.3 Query Translation Algorithm

Given a query twig (XPATH), the `evaluateNCATwiglet` procedure (Figure 4(a)) outputs SQL statement. A SQL statement consists of three clauses: *select_sql*, *from_sql* and *where_sql*. We assume that a clause has an `add()` method which encapsulates

```
evaluateNCATwiglet ( queryTwig )

01 i = 1
02 for every rootedPath in the queryTwig {
03   from_sql.add("PathValue as V_i")
04   where_sql.add("V_i.pathid in rootedPath_i.getPathId()")
05   where_sql.add("V_i.branchOrder < rootedPath_i.level()")
06   if (i > 1) {
07     where_sql.add("V_i.DeweyOrderSum BETWEEN
         V_i-1.DeweyOrderSum –
             RValue(rootedPath_i.NCAlevel() - 1) + 1 AND
         V_i-1.DeweyOrderSum +
             RValue(rootedPath_i.NCAlevel() - 1) - 1")
08   }
09   i++
10 }
11 select_sql.add("DISTINCT V_i-1.docId, V_i-1.DeweyOrderSum")
12 return select_sql + from_sql + where_sql
```

(a) *evaluateNCATwiglet* algorithm

```
XPath: /uniprot/entry[comment/location and
       geneLocation/name]


01 SELECT DISTINCT V2.DocId, V2.DeweyOrderSum
02 FROM PathValue V1, PathValue V2
03 WHERE V1.pathid in (2,3,4)
04 AND V1.branchOrder < 4
05 AND V2.docId = V1.docId
06 AND V2.pathid in (5)
07 AND V2.branchOrder < 4
08 AND V2.DeweyOrderSum BETWEEN
     V1.DeweyOrderSum - CAST(29 as BIGINT) + 1 AND
     V1.DeweyOrderSum + CAST(29 as BIGINT) - 1
```

(b) An example of Translated SQL query

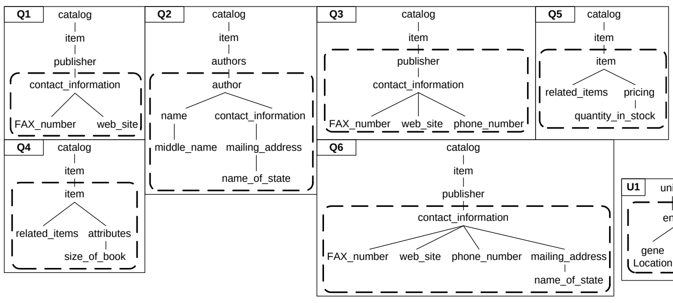
Figure 4: *evaluateNCATwiglet* algorithm.

some simple string manipulations and simple SUCXENT++ joins for constructing valid SQL statements. In addition to preprocessing PathId, for a single XML document, we also preprocess RValue to reduce the number of joins.

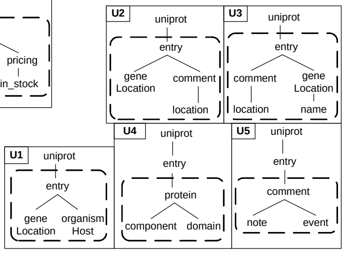
The procedure firstly breaks the query twig into its subsequent rooted path (Line 02). Then for every rooted path, it gets the representative leaf nodes of the rooted path by using PathId and BranchOrder (Lines 04-05). After that, for the second rooted path onwards, it uses Lemma 2 to get the pair of leaf nodes that have NCA at the NCA-level (Line 07). After processing the set of rooted paths, we return the DocId and DeweyOrderSum of the rightmost rooted path (Line 11) since only either one of the pairs is needed to construct the whole subtree. Finally, we collect the final SQL statement (Line 12). For example, consider the query in Figure 1(a). The output SQL statement can be seen in Figure 4(b). Lines 03-04 and Lines 06-07 are used to get the representative leaf nodes of the respective rooted path. Line 08 is used to get the pair of leaf nodes that have NCA at the NCA-level.

4 Performance Study

In this section, we present the performance results of our proposed approach and compare it with a state-of-the-art tree-unaware approach. Since there are several tree-unaware schemes proposed by the community, our selection choice was primarily influenced by the following two criteria. First, the storage scheme of representative approach should not be dependent on the availability of DTD/XML schema. Second, the selected approach must have good query performance for a variety of XPATH axes (ordered as well as unordered) for *query-mostly* workloads. Hence, we chose the GLOBAL-ORDER storage scheme as described in [17]. Prototypes for SUCXENT++ (denoted as SX), and GLOBAL-ORDER (denoted as GO) were implemented with JDK 1.5. The experiments were conducted on an Intel Pentium 4 3GHz machine running on Windows XP with 1GB of RAM. The RDBMS used was Microsoft SQL Server 2005 Developer Edition.



(a) Queries for XBench Data Set



(c) Queries for UniProt Data Set

Query	# of NCA-Twiglet	10MB		100MB		1000MB	
		# of RP		# of RP		# of RP	
		Min	Max	Min	Max	Min	Max
Q1	0 - 30	589	1,926	6,261	18,759	62,023	187,997
Q2	0 - 30	3,066	3,249	31,057	31,393	311,802	313,531
Q3	0 - 30	394	1,698	4,179	16,587	41,323	165,279
Q4	0 - 30	1,201	1,284	12,335	12,675	124,718	125,302
Q5	0 - 30	1,233	1,292	12,414	12,596	124,754	125,246
Q6	0 - 30	316	1,906	3,085	18,899	31,035	187,621

(b) Statistics of NCA and RPs for DCSD Data Sets

Query	20MB			200MB			2000MB		
	# of NCA-Twiglet	# of RP		# of NCA-Twiglet	# of RP		# of NCA-Twiglet	# of RP	
		Min	Max		Min	Max		Min	Max
U1	0	67	76	0	952	1,044	0	14,097	17,557
U2	2	57	67	8	302	1,044	347	5,190	14,097
U3	2	57	67	3	302	338	8	3,711	5,190
U4	0	44	104	0	325	520	162	5,459	13,223
U5	12	74	97	166	502	1,279	1063	4,853	10,875

(d) Statistics of NCA and RPs for UniProt Data Sets

Figure 5: Query and data sets.

Data and Query Sets: In our experiments, we used XBench DCSD [20] as synthetic dataset and UniProt[2] as real dataset. We vary the size of xml documents from 10MB to 1GB for XBench and from 20MB to 2GB for UniProt. Note that the size of the the original UniProt data is 2GB. Hence, we truncated this document into smaller xml documents of 20MB and 200MB sizes. Recall that we wish to explore twig queries that are high-selective although the paths are low-selective. Hence, we modified XBench dataset so that we can control the number of subtrees (denoted as K) that matches the nca-twiglet and the number of occurrences of the rooted paths. We set $K \in \{0, 10, 20, 30\}$ for XBench dataset. Note that we did not modify the UniProt dataset. Figures 5(a) and 5(c) depict the benchmark queries on XBench and UniProt, respectively. We vary the number of rooted paths in the queries from 2 to 4. The number of occurrences of subtrees that satisfies a nca-twiglet and the minimum and maximum numbers of occurrences of rooted paths in the datasets are shown in Figures 5(b) and 5(d) for XBench and UniProt queries, respectively.

Test Methodology: Appropriate indexes were constructed for all approaches (except for monetdb) through a careful analysis on the benchmark queries. Particularly, for Sucxent++ we create the following indexes on PathValue table: (a) unique clustered index on PathId and DeweyOrderSum, and (b) non-unique, non-clustered Index on PathId and BranchOrder. Furthermore, since our dataset consists of a single xml document, we removed the DocId column from the tables in sx and go. Prior to our experiments, we ensure that statistics had been collected. The bufferpool of the rdbms was cleared before each run. Each query was executed 6 times and the results from the first run were always discarded.

[2]Downloaded from www.ebi.ac.uk/uniprot/database/download.html.

ID	10MB				100MB				1000MB			
	K = 0	K = 10	K = 20	K = 30	K = 0	K = 10	K = 20	K = 30	K = 0	K = 10	K = 20	K = 30
Q1	25.60	27.00	28.60	27.40	142.80	157.40	151.80	158.20	1,586.80	1,564.80	1,574.20	1,596.60
Q2	61.00	62.40	62.60	69.80	430.40	418.20	475.20	427.00	3,846.80	3,631.20	3,994.80	3,619.60
Q3	68.80	85.80	85.80	84.20	100.00	182.60	185.60	189.00	990.80	1,664.80	1,620.60	1,640.40
Q4	26.00	28.00	27.60	27.40	205.20	168.20	194.40	180.00	1,980.40	1,995.40	1,950.60	1,985.20
Q5	63.80	75.40	62.20	75.60	180.20	186.80	192.80	189.20	1,994.40	1,947.60	1,963.20	1,952.60
Q6	92.00	100.40	112.40	114.00	83.60	257.00	239.20	237.60	617.20	2,161.20	2,159.00	2,159.00

(a) SUCXENT++ (XBench, in msec)

ID	10MB				100MB				1000MB			
	K = 0	K = 10	K = 20	K = 30	K = 0	K = 10	K = 20	K = 30	K = 0	K = 10	K = 20	K = 30
Q1	609.60	603.00	936.40	717.60	8,478.80	8,316.60	8,862.20	6,066.80	114,717.60	83,035.80	80,979.60	84,053.80
Q2	599.00	467.00	443.20	448.80	6,080.00	5,996.40	5,451.40	6,640.00	349,974.00	236,906.20	226,509.20	225,286.20
Q3	698.20	736.00	957.20	675.00	5,510.60	5,565.20	5,458.40	5,464.60	80,954.80	78,541.20	76,998.20	80,571.40
Q4	494.80	474.00	473.80	763.80	3,522.80	4,250.40	4,727.20	4,598.80	107,910.20	108,039.40	71,082.40	111,399.20
Q5	553.60	492.60	570.20	655.60	4,915.40	4,959.40	4,010.60	4,161.60	70,275.20	111,901.60	71,232.80	75,587.40
Q6	762.40	1,205.00	792.60	970.60	7,516.40	6,925.40	7,907.20	7,948.60	204,835.40	249,687.40	259,323.20	238,127.40

(b) Global Order (XBench, in msec)

ID	10MB				100MB			
	K = 0	K = 10	K = 20	K = 30	K = 0	K = 10	K = 20	K = 30
Q1	65.63	37.50	37.50	34.38	121.88	128.13	128.13	128.13
Q2	56.25	50.00	53.13	53.13	240.63	256.25	253.13	293.75
Q3	37.50	40.63	43.75	50.00	143.75	156.25	293.75	159.38
Q4	28.13	31.25	32.15	32.15	93.75	96.88	100.00	96.88
Q5	31.25	31.25	32.15	32.15	125.00	100.00	100.00	128.13
Q6	40.63	40.63	40.63	40.63	178.13	178.13	184.38	184.38

(c) MonetDB (XBench, in msec)

ID	SUCXENT++			Global Order			MonetDB	
	20MB	200MB	2000MB	20MB	200MB	2000MB	20MB	200MB
U1	9.00	23.40	163.60	201.00	1,424.60	17,281.60	31.25	71.88
U2	9.00	21.00	156.80	363.80	2,512.60	23,839.20	40.63	253.13
U3	5.00	12.20	86.00	266.00	2,675.20	23,705.20	40.63	250.00
U4	5.40	14.40	123.60	14.00	618.80	6,870.80	15.63	93.75
U5	5.00	22.20	123.60	438.80	2,800.00	43,502.20	40.63	312.50

(d) UniProt (in msec)

Figure 6: Performance results.

Since GO and SX have different storage approaches, the structure of the returned results are also different. Recall from Section 3.2, the goal of our study is to identify subtrees that matches the NCA-twiglet, we do not reconstruct the entire matched subtree. Particularly, for the GO approach, we return the identifier of the root of the subtree (without its descendants) that matches the NCA-twiglet. Whereas for SX, we return the DeweyOrderSum of the root-to-leaf path of the matching subtree. This path must satisfy the rightmost rooted path of the NCA-twiglet. For example, for the query in Figure 1(a), we return the identifiers of the `entry` nodes in GO and the DeweyOrderSums of the root-to-leaf paths containing the rightmost rooted path `entry/geneLocation/name` nodes in SX. Lastly, for SX we enforce a "left-to-right" join order on the translated SQL query using query hints. The performance benefits of such enforcement is discussed in [15].

NCA-twiglet evaluation times: Our experimental goal is to measure the evaluation time for determining those subtrees that match a NCA-twiglet with a specific reachability path in the twig queries in Figure 5. Figures 6(a) and 6(b) depict the NCA-twiglet evaluation times of SUCXENT++ and GLOBAL-ORDER, respectively. Figure 6(d) depicts the evaluation time for UNIPROT data set. We observe that SX significantly outperforms GO for all queries with the highest observed factor being 352 (Query *U5* on 2GB dataset). Particularly, SX is orders of magnitude faster for high-selective queries. Observe that for XBench dataset, when $K = 0$, SX is up to 332 times faster (Query *Q6* on 1GB dataset) and on average 56 times faster than GO. This is significant in an environment where users would like to issue exploratory ad hoc queries. In this case, the user would like to know quickly if the query returns any results. If the result set is empty then he/she can further refine his/her query accordingly.

sx is significantly faster than go because of the following reasons. Firstly, sx uses an efficient strategy based on Theorem 1 to reduce useless comparisons. Furthermore, the number of join operations in go is more than sx. For example, for *Q6*, go and sx join six tables and three tables, respectively. Secondly, go stores every node of an xml document whereas sx stores only the root-to-leaf paths. Consequently, the number of tuples in the Edge table is much more than that in the PathValue table.

Comparison with monetdb: Recently, in [3], it has been shown that monetdb is among the fastest and most scalable XQuery processor and outperforms the current generation of XQuery systems by quite a big margin. Although monetdb, being a main-memory database systems, does not incur additional overhead like disk-bound database systems, we would still like to observe how "far off" our proposed technique is from monetdb. We used the Windows version of monetdb/XQuery 0.16.0 [3] downloaded from http://monetdb.cwi.nl/XQuery/Download/index.html (Win32 builds). Note that the evaluation time of monetdb includes the time for subtree construction whereas the evaluation time of sx and go includes only computing the root (rooted path for sx) of the matched subtrees. Consequently, the evaluations times do not precisely reflect comparative performances of go and sx against monetdb. *However, they do tell us how "far off" tree-unaware approaches are compared to a state-of-the-art XQuery processor.* The key observation here is that go is significantly slower than monetdb even without the construction of the entire subtree. *However, this performance gap is significantly reduced when it is compared against* sx. Interestingly, for all queries on real dataset (*U1-U5* in Figure 6(d)), sx is 3-21 times faster than monetdb! Note that we did not show any results of monetdb for 1GB dataset as it is currently vulnerable to the virtual memory fragmentation in Windows environment. Consequently, it failed to shred 1GB XBench (2GB for UniProt) dataset.

5 Related Work

We now compare our proposed approach with existing tree-unaware techniques. Note that we do not compare our work with tree-aware schemes [1, 3, 7, 9–11, 22] as these techniques modify the database internals. The tree-unaware approaches in [8, 16] typically decompose the query twig pattern into a set of parent-child or ancestor-descendant binary components. Our approach is different from this class of techniques since it does not decompose the path expressions into binary relations and hence does not suffer from the explosion of the size of the intermediate result set as well as number of joins. XRel [21] stores the path of each node in the document. Then, the resolution of path expressions only requires the paths (which can be represented as strings) to be matched using string matching operators. However, the XRel approach uses several theta joins to resolve containment queries that have been shown to be quite expensive due to the manner in which an RDBMS processes joins [22]. In [17], Tatarinov *et al.* proposed the first solution for supporting ordered XML query processing by using a tree-unaware relational database. A modified EDGE table [8] was the underlying storage scheme. In comparison, our approach uses a novel encoding scheme that can evaluate NCA-twiglets more efficiently (as discussed in Section 4). Lastly, all previous tree-unaware approaches (except for [13, 15]), reported query performance on XML documents with small/medium sizes – smaller than 500 MB. We investigate query performance on large synthetic and real datasets (up to 2GB). This gives more insights on the scalability of the state-of-the-art tree-unaware approaches for twig query processing.

Our work in this paper differs from our previous approaches [13, 15] in the following ways. In [13], along with BranchOrder, we store an attribute called BranchOrderSum for each leaf node. These two attributes are used along with RValue to process recursive queries and also to find the *minimum* level of NCA. However, in our approach, the DeweyOrderSum enables us to *exactly* determine the level of a NCA. Additionally, BranchOrderSum does not support efficient evaluation of ordered XPATH axes. Consequently, DeweyOrderSum and SiblingSum are proposed in [15] to replace BranchOrderSum in order to address this problem. In this paper we do not focus on evaluating ordered path expressions. Rather, we investigate how the encoding scheme in [15] can be used for efficiently processing NCA-twiglet, a specific class of structural relationship in a twig pattern query.

6 Conclusions

The key challenge in XML twig pattern evaluation is to efficiently match the structural relationships of the query nodes against the XML database. In general, structural relationship in a twig query may be categorized in two different classes: path expression and NCA-twiglet. A path expression enforces linear structural constraint whereas NCA-twiglet specifies tree-structured relationship. In this paper, we present an efficient strategy to evaluate NCA-twiglets having parent-child relationship in a tree-unaware relational environment. Our scheme is build on top of SUCXENT++ [15]. We show that by exploiting the encoding scheme of SUCXENT++ we can reduce useless structural comparisons in order to evaluate NCA-twiglets. Our results showed that our proposed approach outperforms GLOBAL-ORDER [17], a representative *tree-unaware* approach for all benchmark queries. Although *tree-aware* approaches are often the best in terms of query performance [3], our scheme reduces significantly the performance gap between tree-aware and tree-unaware approaches. Importantly, unlike tree-aware approaches, our scheme does not require invasion of the database kernel to improve query performance and can easily be built on top of any off-the-shelf commercial RDBMS.

References

[1] S. Al-Khalifa, H.V. Jagadish, J. M. Patel et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *In ICDE*, 2002.

[2] S. Alstrup, C. Gavoille, H. Kaplan, T. Rauhe. Nearest Common Ancestors: A Survey and a new Distributed Algorithm. *In SPAA*, 2002.

[3] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. *In SIGMOD* , 2006.

[4] N. Bruno, N. Koudas, D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. *In SIGMOD*, 2002.

[5] S. Chien, H-G. Li, J. Tatemura et al. Twig^{2}Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents. *In VLDB*, 2006.

[6] S. Chien, Z. Vagena, D. Zhang et al. Efficient Structural Joins on Indexed XML Documents. *In VLDB*, 2002.

[7] D. DeHaan, D. Toman, M. P. Consens, M. T. Ozsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Coding. *In SIGMOD*, 2003.

[8] D. Florescu, D. Kossman. Storing and Querying XML Data using an RDBMS. *IEEE Data Engg. Bulletin*. 22(3), 1999.

[9] T. Grust, J. Teubner, M. V. Keulen. Accelerating XPath Evaluation in Any RDBMS. *In ACM TODS*, 29(1), 2004.

[10] Q. Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *In VLDB*, 2001.

[11] J. Lu, T. Chen, T. W. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. *In CIKM*, 2004.

[12] J. Lu, T. W. Ling, T. Y. Chen, T. Chen. From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. *In VLDB*, 2005.

[13] S. Prakash, S. S. Bhowmick, S. K. Madria. Efficient Recursive XML Query Processing Using Relational Databases. *In DKE*, 58(3), 2006.

[14] P. Rao, B. Moon. PRIX: Indexing and Querying XML Using Prufer Sequences. *In ICDE*, 2004.

[15] B.-S Seah, K. G. Widjanarko, S. S. Bhowmick, B. Choi, E. Leonardi. Efficient Support for Ordered XPath Processing in Tree-Unaware Commercial Relational Databases. *In DASFAA*, 2007.

[16] J. Shanmugasundaram, K. Tufte et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. *In VLDB*, 1999.

[17] I. Tatarinov, S. Viglas, K. Beyer, et al. Storing and Querying Ordered XML Using a Relational Database System. *In SIGMOD*, 2002.

[18] H. Wang, S. Park, W. Fan, P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. *In SIGMOD*, 2003.

[19] X. Wu, M. Lee, W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. *In ICDE*, 2004.

[20] B. Yao, M. Tamer Özsu, N. Khandelwal. XBench: Benchmark and Performance Testing of XML DBMSs. *In ICDE*, 2004.

[21] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of xml documents using relational databases. *In ACM TOIT* 1(1):110-141, 2001.

[22] C. Zhang, J. Naughton, D. Dewitt, Q. Luo and G. Lohmann. On Supporting Containment Queries in Relational Database Systems. *In SIGMOD*, 2001.