# Efficient Evaluation of NOT-Twig Queries in Tree-Unaware Relational Databases

Kheng Hong Soh[2] and Sourav S. Bhowmick[1,2]

[1] Singapore-MIT Alliance, Nanyang Technological University, Singapore
[2] School of Computer Engineering, Nanyang Technological University, Singapore
assourav@ntu.edu.sg

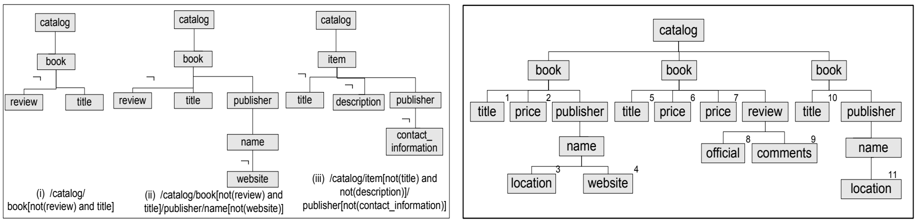
**Abstract.** Despite a large body of work on XML query processing in relational environment, systematic study of NOT-*twig queries* has received little attention in the literature. Such queries contain not-predicates and are useful for many real-world applications. In this paper, we present an efficient strategy to evaluate NOT-twig queries on top of a dewey-based *tree-unaware* system called SUCXENT++ [11]. We extend the encoding scheme of SUCXENT++ by adding two new labels, namely AncestorValue and AncestorDeweyGroup, that enable us to *directly* filter out elements satisfying a not-predicate by comparing their *ancestor group identifiers.* In this approach, a set of elements under the same common ancestor at a specific level in the XML tree is assigned *same ancestor group identifier.* Based on this encoding scheme, we propose a novel SQL translation algorithm for NOT-twig query evaluation. Real and synthetic datasets are employed to demonstrate the superiority of our approach over industrial-strength RDBMS and native XML databases.

## 1 Introduction

Querying XML data over relational framework has gained popularity due to its stability, efficiency, expressiveness, and its wide spread usage in the commercial world. On the one hand, there has been a host of work, *c.f.*, [3], on enabling relational databases to be *tree-aware* by invading the database kernel to support XML. On the other hand, some completely jettison the invasive approach and resort to a *tree-unaware* approach, *c.f.*, [4, 7, 11, 13, 14], where the database kernel is not modified to support XML queries.

Generally, the tree-unaware approach reuses existing code, has a lower cost of implementation, and is more portable since it can be implemented on top of off-the-shelf RDBMSs. This has triggered recent efforts to explore how far we can push the idea of using mature tree-unaware RDBMS technology to design and build a relational XQuery processor [4, 5, 7]. Particularly, a wealth of existing literature has extensively studied evaluation of various navigational axes in XPath expressions and optimization techniques in a tree-unaware environment [4, 5, 7, 11, 13, 14]. *However, to the best of our knowledge, no systematic study has been carried out in efficiently evaluating* NOT-*twig queries in this relational environment.* Such queries contain not-predicates and are useful for many real-world applications. For example, the query `/catalog/book[not(review) and`

© Springer-Verlag Berlin Heidelberg 2011



(a) Examples of NOT-twig query (b) An XML document

**Fig. 1.** Examples of NOT-twig queries and XML document

| Id | Size | No. of Attributes | No. of Nodes | Total | Max Depth |
|---|---|---|---|---|---|
| DC10 | 10.3MB | 15,000 | 152,673 | 167,673 | 8 |
| DC100 | 103.3MB | 150,000 | 1,520,322 | 1,670,322 | 8 |
| DC1000 | 1033.3MB | 1,500,000 | 15,215,868 | 16,715,868 | 8 |

(a) XBENCH data sets

| Id | Size | No. of Attributes | No. of Nodes | Total | Max Depth |
|---|---|---|---|---|---|
| U28 | 28MB | 771,880 | 422,972 | 1,194,852 | 6 |
| U284 | 284MB | 7,791,620 | 4230,003 | 12,021,623 | 6 |
| U2843 | 2843MB | 77,977,270 | 42,421,745 | 120,399,015 | 6 |

(b) UniProt data sets

| Data Set | Q1 | |
|---|---|---|
| | XSysA | XSysB |
| DC10 | 1,439 | 334 |
| DC100 | 2,332 | 2,823 |
| DC1000 | 11,298 | 57,309 |

(d) XBench data set

| Data Set | Q2 | | Q3 | |
|---|---|---|---|---|
| | XSysA | XSysB | XSysA | XSysB |
| U28 | 20,399 | 2,457 | 5,899 | 1,977 |
| U284 | 207,982 | 24,429 | 47,386 | 20,819 |
| U2843 | - | - | - | - |

(e) UniProt data set

| Id | Data Source | NOT-twig Query |
|---|---|---|
| Q1 | XBench | /catalog/item/authors/author[not(biography)]/name |
| Q2 | UniProt | /uniprot/entry[not(geneLocation/name) and not(comment/location)] |
| Q3 | UniProt | /uniprot/entry[not(geneLocation) and not(protein/domain)]/comment[not(note) and not(event)] |

(c) NOT-twig queries

**Fig. 2.** Data sets and query evaluation times (in msec.)

`title]` retrieves all `book`s that have a `title` but no `reviews` (Figure 1(a)(i)). Figures 1(b)(ii) and 1(c)(iii) show graphical representations of two more NOT-twig queries.

At first glance, it may seem that such lack of study may be primarily due to the fact that we can efficiently evaluate these NOT-twig queries by leveraging on the XML query processor of an existing industrial-strength RDBMS and relying on its query optimization capabilities. However, our initial investigation showed that fast evaluation of NOT-twigs still remains a bottleneck in several industrial-strength RDBMSs. To get a better understanding of this problem, we experimented with the XBench DCSD [15] and UNIPROT (downloaded from `www.expasy.ch/sprot/`) data sets shown in Figures 2(a) and 2(b) and queries $Q_1$ – $Q_3$ in Figure 2(c). We fix the result size of $Q_1$ to be 500. Figures 2(d) and 2(e) report the query evaluation times in two commercial-strength RDBMSs. Note that due to legal restrictions, these systems are anonymously identified as *XSysA* and *XSysB* in the sequel. Observe that the evaluation cost can be expensive as it can take up to 208 seconds to evaluate these queries. Also, both these commercial systems do not support processing of XML documents having size greater than 2GB (U2843 data set). *Is it possible to design a tree-unaware scheme that can address this performance bottleneck?* In this paper, we demonstrate that novel techniques built on top of an industrial-strength RDBMS can make up for a large part of the limitation. We show that the above queries can be evaluated in *a second or less* on smaller data sets and *less than* 13$s$ for $Q2$ on U284 data sets.

We built our proposed NOT-twig evaluation technique on top of dewey-based SUCXENT++ system [2, 11], a tree-unaware approach designed primarily for read-mostly workloads. As SUCXENT++ is designed primarily for fast evaluation of normal path and twig queries, it does not support efficient evaluation of NOT-twig queries. Hence, in Section 3 we extend SUCXENT++ encoding scheme by adding two new labels, namely AncestorValue and AncestorDeweyGroup, to each level and leaf elements, respectively. These labels enable us to efficiently group a set of elements under the same common ancestor at a specific level with the *same ancestor group identifier*. As we shall see later, this will allow us to efficiently filter out elements satisfying a not-predicate by comparing their *ancestor group identifiers*.

Based on the *extended* encoding scheme, we propose a novel SQL translation algorithm for NOT-twig evaluation (Section 4). In our approach, we use the AncestorDeweyGroup and AncestorValue labels to evaluate *all* paths in a NOT-twig query. In Section 5, we demonstrate with exhaustive experiments that the proposed approach is significantly faster than XML supports of *XSysA* and *XSysB* (highest observed factor being 40 times).

Our proposed approach differs from existing efforts in evaluating NOT-twigs using structural join algorithms [1, 8, 10, 16] in the following ways. Firstly, we take relational-based approach instead of native strategy used in aforementioned approaches. Secondly, our encoding scheme is different from the above approaches. In [16], region encoding scheme is employed to label the elements whereas a pair of *(path-id, node id)* [9] is used in [10]. In contrast, we use a dewey-based scheme where only the leaf elements and the levels of the XML tree are explicitly encoded. Thirdly, these existing approaches typically report query performance on documents smaller than 150MB and containing at most 2.5 million nodes. In contrast, we explore the scalability of our approach for larger XML documents (2.8GB size) having more than 120 million nodes.

## 2 Preliminaries

**XML Data Model.** We model XML documents as ordered trees. In our model we ignore comments, attributes, processing instructions, and namespaces. Queries in XML query languages make use of twig patterns to match relevant portions of data in an XML database. A twig pattern can be represented as a tree containing all the nodes in the query. A node $m_i$ in the pattern 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. Each node $m_i$ and its parent (or ancestor) $m_j$ are connected by an edge, denoted as $edge(m_i, m_j)$.

A twig query contains a collection of *rooted path* patterns. A *rooted path* pattern (RP) is a path from the root to a node in the twig. Each rooted path represents a sequence of nodes having parent-child (PC) or ancestor-descendant (AD) edges. We classify the rooted paths into two types: *root-to-leaf* and *root-to-internal* paths. A *root-to-leaf* path is a RP from the root to a leaf node in the query. In contrast, a RP ending at a non-leaf node is called a *root-to-internal* path. If the number of children of a node in the twig query is more than one, then we call this node a NCA (nearest common ancestor) *node*. Otherwise, when the node has only one child, it is a *non*-NCA *node*. The level of the NCA node is called NCA-*level*.

In this paper, we focus on twig queries with not-predicates. We refer to such queries as NOT-*twig queries*. The twig pattern edges of a NOT-twig query can be classified into one of the following two types. (a) *Positive edge:* This corresponds to an $edge(m_i, m_j)$ without not-predicate in the query expression. It is represented as "|" or "||" in a twig pattern for PC or AD edges, respectively. Node $m_j$ is called the *positive* PC *(resp.* AD*)* child of $m_i$. A rooted path that contains only positive children is called a *normal* rooted path. (b) *Negative edge:* This corresponds to an $edge(m_i, m_j)$ with not-predicate and is represented as "$|\neg$" or "$||\neg$" in the twig for PC or AD edges, respectively. In this case, node $m_j$ is called the *negative* PC *(resp.* AD*)* child of $m_i$. A rooted path pattern that contains a negative child is called a *negative* rooted path. For example, consider the NOT-twig query in Figure 1(a)(ii). *edge*(`book,title`) and *edge*(`book,publisher`) are positive edges whereas *edge*(`book,review`) and *edge*(`name,website`) are negative edges. Node `book` has three children, in which `title` and `publisher` are positive PC children and node `review` is a negative PC child. The RP `catalog/book/review` is a negative RP as it contains the negative PC child `review`. On the other hand, `catalog/book/title` is a normal RP.

**NOT-Twig Pattern Matching.** Given a NOT-twig query $Q$, a query node $n$, and an XML tree $D$, an element $e_n$ (with the tag $n$) in $D$ satisfies the subquery rooted at $n$ of $Q$ iff: (1) $n$ is a leaf node of $Q$; or (2) For each child node $n_c$ of $n$ in $Q$: (a) If $n_c$ is a positive PC (resp. AD) child of $n$, then there is an element $e_{n_c}$ in $D$ such that $e_{n_c}$ is a child (resp. descendant) element of $e_n$ and satisfies the sub-query rooted at $n_c$ in $D$. (b) If $n_c$ is a negative PC (resp. AD) child of $n$, then there does not exists any element $e_{n_c}$ in $D$ such that $e_{n_c}$ is a child (resp. descendant) element of $e_n$ and satisfies the sub-query rooted at $n_c$ in $D$.

## 3 Encoding Scheme

In this section, we first briefly describe the encoding scheme of SUCXENT++ [2, 11] and highlight its limitations in efficiently processing NOT-twig queries. Then, we present how it can be extended to efficiently support queries with not-predicates.

### 3.1 SUCXENT++ Schema and Its Limitations

In SUCXENT++, each level $\ell$ of an XML tree is associated with an attribute called RValue (denoted as $R_\ell$). Each leaf element $n$ is associated with four attributes,

Document

| DocID | Name |
|---|---|
| 1 | catalog.xml |

DocumentRValue

| DocID | Level | RValue |
|---|---|---|
| 1 | 1 | 29 |
| 1 | 2 | 4 |
| 1 | 3 | 2 |
| 1 | 4 | 1 |

Path

| PathID | PathExp |
|---|---|
| 1 | .catalog#.book#.title# |
| 2 | .catalog#.book#.price# |
| 3 | .catalog#.book#.publisher#.name#.location# |
| 4 | .catalog#.book#.publisher#.name#.website# |
| 5 | .catalog#.book#.review#.official# |
| 6 | .catalog#.book#.review#.comments# |

PathValue

| DocID | Leaf Order | Brach Order | PathID | Dewey Order Sum | Sibling Sum | Leaf Value |
|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | ..... |
| 1 | 2 | 2 | 2 | 7 | 0 | .... |
| 1 | 3 | 2 | 3 | 14 | 0 | ..... |
| 1 | 4 | 4 | 4 | 15 | 0 | ..... |
| 1 | 5 | 1 | 1 | 57 | 57 | ..... |
| 1 | 6 | 2 | 2 | 64 | 57 | ..... |
| 1 | 7 | 2 | 2 | 71 | 64 | ...... |
| 1 | 8 | 2 | 5 | 78 | 57 | ...... |
| 1 | 9 | 3 | 6 | 81 | 57 | ...... |
| 1 | 10 | 1 | 1 | 114 | 114 | ...... |
| 1 | 11 | 2 | 3 | 121 | 114 | ..... |

**Fig. 3.** Storage of a shredded XML document

namely LeafOrder, BranchOrder, DeweyOrderSum, and SiblingSum. Each non-leaf element $n'$ is *implicitly* assigned the DeweyOrderSum of the first descendant leaf element. Here we briefly define the relevant attributes necessary to understand this paper. The reader may refer to [2, 11] for details related to their roles in XML query processing.

The schema of SUCXENT++ [2, 11] is as follows: (a) Document(<u>DocID</u>, Name), (b) Path(<u>PathId</u>, PathExp), (c) PathValue(<u>DocID, DeweyOrderSum</u>, PathId, BranchOrder, LeafOrder, SiblingSum, LeafValue), and (d) DocumentRValue(<u>DocID, Level</u>, RValue). Document stores the document identifier DocID and the name Name of a given input XML document $D$. Each distinct root-to-leaf path appearing in $D$, namely PathExp, is associated with an identifier PathId and stored in Path table. Essentially each path is a concatenation of the labels of the elements in the path from the root to the leaf. An example of the Path table containing the root-to-leaf paths of Figure 1(b) is shown in Figure 3. Note that '#' is used as a delimiter of steps in the paths instead of '/' for reasons described in [14].

For each leaf element $n$ in $D$, a tuple in the PathValue table is created to store the LeafOrder, BranchOrder, DeweyOrderSum, and SiblingSum values of $n$. The data value of $n$ is stored in LeafValue. Given two leaf elements $n_1$ and $n_2$, $n_1$.LeafOrder $<$ $n_2$.LeafOrder *iff* $n_1$ precedes $n_2$ in document order. LeafOrder of the first leaf element in $D$ is 1 and $n_2$.LeafOrder $=$ $n_1$.LeafOrder$+1$ *iff* $n_1$ is a leaf element immediately preceding $n_2$. For example, the superscript of each leaf element in Figure 1(b) denotes its LeafOrder value.

Given two leaf elements $n_1$ and $n_2$ where $n_1$.LeafOrder$+1$ $=$ $n_2$.LeafOrder, $n_2$.BranchOrder is the level of the nearest common ancestor (NCA) of $n_1$ and $n_2$. For example, the BranchOrder of the `location` leaf element with LeafOrder value 3 in Figure 1(b) is 2 as the NCA of this element and the preceding `price` element is at the second level. Note that the BranchOrder of the first leaf element is 0.

Next we define RValue. We begin by introducing the notion of *maximal k-consecutive leaf-node list.* Consider a list of consecutive leaf element $\mathcal{S}$: $[n_1, n_2, n_3, \ldots, n_r]$ in $D$. Let $k \in [1, L_{max}]$ where $L_{max}$ is the largest level of $D$. Then, $\mathcal{S}$ is called a *$k$-consecutive leaf-node list* of $D$ *iff* $\forall 0 < i \leq r$ $n_i$.BranchOrder $\geq k$. $\mathcal{S}$ is called a *maximal* $k$-consecutive leaf-node list, denoted as $M_k$, if there does not exist a $k$-consecutive leaf-node list $\mathcal{S}'$ such that $|\mathcal{S}|<|\mathcal{S}'|$. For example, $M_2$ in Figure 1(b) contains four leaf elements as $|\mathcal{S}| = 4$ for $M_2$.

The RValue of level $\ell$, denoted as $R_\ell$, 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 Figure 1(b). Here $L_{max} = 5$. The values of $|M_2|$, $|M_3|$, and $|M_4|$ are 4, 1, and 1, respectively. Then, $R_4 = 1$, $R_3 = 2 \times 1 \times |M_4| + 1 = 3$, $R_2 = 2 \times 3 \times |M_3| + 1 = 7$, and $R_1 = 2 \times 7 \times |M_2| + 1 = 57$. In order to facilitate evaluation of XPath queries, the RValue attribute in DocumentRValue stores $\frac{R_\ell - 1}{2} + 1$ instead of $R_\ell$ (denoted as $R'_\ell$). For instance, in Figure 3 the RValue of level 1 is stored as 29 instead of 57.

DeweyOrderSum is used to encode an element's order information together with its ancestors' order information using a single value. Let $parent(w)$ denote the parent of an element $w$. Consider a leaf element $n$ at level $\ell$ in $D$. Then, for $1 < k \leq \ell$, $\mathsf{Ord}(n,k) = i$ *iff* (i) there exists an element $a$ at level $k$ which is either an ancestor of $n$ or $n$ itself; and (ii) $a$ is the $i$-th child of $parent(a)$. For example, consider the rightmost leaf element in Figure 1(b) (denoted as $d$). $\mathsf{Ord}(d,2) = 3$ as the rightmost book element in the second level is an ancestor of $d$ as well as the third child of the root. Similarly, $\mathsf{Ord}(d,3) = 2$.

Then DeweyOrderSum of $n$, $n$.DeweyOrderSum, is defined as $\sum_{j=2}^{\ell} \Phi(j)$ where $\Phi(j)$=[$\mathsf{Ord}(n,j)$-1]$\times R_{j-1}$. The DeweyOrderSum of the first leaf element is 0. Reconsider the rightmost leaf element again. It has a Dewey path "1.3.2.1.1". DeweyOrderSum of this element is: $n$.DeweyOrderSum $= (Ord(n,2) - 1) \times R_1 + (Ord(n,3) - 1) \times R_2 + (Ord(n,4) - 1) \times R_3 + (Ord(n,5) - 1) \times R_4 = 2 \times 57 + 1 \times 7 + 0 \times 3 + 0 \times 1 = 121$. The DeweyOrderSum of remaining elements are shown in the DeweyOrderSum attribute of the PathValue table in Figure 3.

**Limitations of SUCXENT++.** DeweyOrderSum and RValue attributes are designed primarily to evaluate normal twig queries. Consequently, they are unable to *directly* filter out elements satisfying negative RPs without having to first evaluate the rooted paths as normal RPs and then use the intermediate results to filter out irrelevant elements (see details in [12]). For instance, for the query in Figure 1(a)(i), DeweyOrderSum and RValue attributes fail to reveal those title and review elements that *do not* share the same common book ancestors without exhaustively comparing them. Furthermore, they do not always support efficient evaluation of descendant (ancestor) axis. In the subsequent sections, we shall present a novel technique that addresses these limitations.

## 3.2 AncestorValue Attribute

We now elaborate on the extension of the encoding scheme of SUCXENT++. Due to space constraints, the proofs of lemmas and theorem presented in the sequel are given in [12]. Each level $\ell$ of an XML tree is added an attribute called AncestorValue along with its existing RValue. Each leaf element $n$ is added an attribute called AncestorDeweyGroup. These attributes are materialized in the DocumentRValue and PathValue tables, respectively. As we shall see later, our proposed strategy aims to group a set of leaf elements under the same common ancestor at level $\ell$ with the *same ancestor group identifier*. AncestorDeweyGroup and AncestorValue attributes will be used to compute these identifiers.

AncestorValue, similar to RValue, is used for encoding the level of the NCA of any pairs of leaf elements.

**Definition 1.** ***[AncestorValue]*** *Let $L_{max}$ be the maximum level of an* XML *tree. Then the* ***AncestorValue*** *of level $\ell$ for $0 < \ell < L_{max}$, denoted as $A_\ell$, is defined as follows: (a) If $\ell = L_{max} - 1$, then $A_\ell = 1$; (b) If $0 < \ell < L_{max} - 1$, then $A_\ell = A_{\ell+1} \times (|M_{\ell+1}| + 1)$.*

For example, reconsider the XML tree in Figure 1(b). Here $L_{max} = 5$, $|M_4| = 1$, $|M_3| = 1$, and $|M_2| = 4$. Hence, $A_4 = 1$, $A_3 = 1 \times (1+1) = 2$, $A_2 = 2 \times (1+1) = 4$, and $A_1 = 4 \times (4+1) = 20$.

**Lemma 1.** *Let $\ell$ be a level in an* XML *tree where $0 < \ell < L_{max}$. Then, $A_\ell$ is divisible by all $A_{\ell+m}$ where $0 < m < (L_{max} - \ell)$.*

Consider the previous example. Let $\ell = 2$. Then, $0 < m < 3$. Hence based on the above lemma, $A_2/A_3 = 4/2 = 2$ and $A_2/A_4 = 4/1 = 4$. Note that existing RValue do not have such divisibility property [12].

### 3.3 AncestorDeweyGroup Attribute

The AncestorDeweyGroup attribute, similar to DeweyOrderSum, is used to encode an element's order information using a single value. The only difference between AncestorDeweyGroup and DeweyOrderSum is that the former uses each level's AncestorValue whereas the latter uses the RValue of each level.

**Definition 2.** ***[AncestorDeweyGroup]*** *Consider a leaf element $n$ at level $\ell$ in an* XML *document. Then, for $1 < k \leq \ell$, $\mathsf{Ord}(n, k) = i$ iff (i) there exists an element $a$ at level $k$ which is either an ancestor of $n$ or $n$ itself; and (ii) $a$ is the $i$-th child of $parent(a)$. Then* ***AncestorDeweyGroup*** *of $n$, $n$.AncestorDeweyGroup, is defined as $\sum_{j=2}^{\ell} \Omega(j)$ where $\Omega(j)$=[$\mathsf{Ord}(n, j)$-1]$\times A_{j-1}$.*

For example, reconsider the last leaf element in Figure 1(b) with Dewey value "1.3.2.1.1". AncestorDeweyGroup of this element is: $n$.AncestorDeweyGroup $= (Ord(n, 2) - 1) \times A_1 + (Ord(n, 3) - 1) \times A_2 + (Ord(n, 4) - 1) \times A_3 + (Ord(n, 5) - 1) \times A_4 = 2 \times 20 + 1 \times 4 + 0 \times 2 + 0 \times 1 = 44$. The AncestorDeweyGroup values of remaining leaf elements in Figure 1(b) are (in document order): 0, 4, 8, 9, 20, 24, 28, 32, 34, and 40.

## 4 Ancestor Group-Based Approach

We begin by formally introducing the notion of *ancestor group identifier*. Then, we present how such identifiers can be used for evaluating NOT-twig queries.

## 4.1 Ancestor Group Identifier

Informally, given an internal element $n$ at level $\ell > 1$ of an XML tree, a unique ancestor group identifier with respect to $\ell$ is assigned to all the descendant leaf element(s) of $n$. It is computed using AncestorDeweyGroup values of the leaf elements and the AncestorValue of level of $n$.

**Definition 3.** *[Ancestor Group Identifier] Let $n_i$ be a leaf element in the XML tree $D$. Let $n_a$ be an ancestor element of $n_i$ at level $\ell > 1$. Then* ***Ancestor Group Identifier*** *of $n_i$ w.r.t $n_a$ at level $\ell$ is defined as* $\mathcal{G}_i^\ell = \left\lfloor \frac{n_i.\mathsf{AncestorDeweyGroup}}{A_{\ell-1}} \right\rfloor$.

For example, consider the leaf elements $n_1$, $n_2$, $n_3$, and $n_4$ (we denote a leaf element as $n_i$ where $i$ is its LeafOrder value) in Figure 1(b). The AncestorDeweyGroup values of these elements are 0, 4, 8, and 9, respectively. Also, $A_1 = 20$ and $A_2 = 4$. If we consider the first `book` element at level 2 as the ancestor element of these elements, then $\mathcal{G}_1^2 = \left\lfloor \frac{0}{A_{2-1}} \right\rfloor = 0$, $\mathcal{G}_2^2 = \left\lfloor \frac{4}{A_{2-1}} \right\rfloor = 4/20 = 0$, $\mathcal{G}_3^2 = \left\lfloor \frac{8}{20} \right\rfloor = 0$, and $\mathcal{G}_4^2 = \left\lfloor \frac{9}{20} \right\rfloor = 0$. However, if we consider the `publisher` element at level 3 as ancestor element, then $\mathcal{G}_3^3 = \left\lfloor \frac{8}{4} \right\rfloor = 2$, and $\mathcal{G}_4^3 = \left\lfloor \frac{9}{4} \right\rfloor = 2$. Note that we do not define ancestor group identifier with respect to the root element ($\ell = 1$) because it is a trivial case as all leaf elements in the document shall have same identifier values.

**Ancestor group identifiers of non-leaf elements:** Observe that in the above definition only the leaf elements have *explicit* ancestor group identifiers. We assign the ancestor group identifiers to the internal elements implicitly. The basic idea is as follows. Let $n_c$ be the NCA at level $\ell$ of two leaf elements $n_i$ and $n_j$ with ancestor group identifiers equal to $\mathcal{G}^\ell$. Then, the ancestor group identifiers of all non-leaf elements in the subtree rooted at $n_c$ is $\mathcal{G}^\ell$. For example, reconsider the first `book` element at level 2 as the root of the subtree. Then, the ancestor group identifiers of the `publisher` and `name` elements are 0. Note that these identifiers are not stored explicitly as they can be computed from AncestorDeweyGroup and AncestorValue values.

**Role of ancestor group identifiers to evaluate descendant axis.** Observe that a key property of the ancestor group identifier is that all descendants of an ancestor element at a specific level must have *same* identifiers. We can exploit this feature to efficiently evaluate descendant axis. Given a query `a//b`, let $n_a$ and $n_b$ be elements of types $a$ and $b$, respectively. Then, whether $n_b$ is a descendant of $n_a$ can be determined using the above definition as all descendants of $n_a$ must have *same* ancestor group identifiers. As we shall see later, *this equality property is also important for our* NOT*-twig evaluation strategy.*

**Remark.** Due to the lack of divisibility property of RValue (Lemma 1), it cannot be used along with the DeweyOrderSum to correctly compute the *ancestor group identifiers* of elements. Consequently, they are not particularly suitable for efficient evaluation of NOT-twig queries. Due to the space limitations, these issues are elaborated in [12].

## 4.2 Computation of Common Ancestors

**Lemma 2.** *Let $n_i$ and $n_j$ be two leaf elements in $D$ at level $\ell_1$ and $\ell_2$, respectively. Let $\ell < \ell_1$ and $\ell < \ell_2$. (a) If $\mathcal{G}_i^\ell \neq \mathcal{G}_j^\ell$ then $n_i$ and $n_j$ do not have a common ancestor at level $\ell$. (b) If $\mathcal{G}_i^\ell = \mathcal{G}_j^\ell$ then $n_i$ and $n_j$ must have a common ancestor at level $\ell$.*

*Example 1.* Consider the leaf elements $n_1$, $n_2$, $n_5$, and $n_6$ in Figure 1(b). The AncestorDeweyGroup values of these elements are 0, 4, 20, and 24, respectively. Also, $A_1 = 20$. Then, with respect to level 2 $\mathcal{G}_1^2 = \left\lfloor \frac{0}{20} \right\rfloor = 0$, $\mathcal{G}_2^2 = \left\lfloor \frac{4}{20} \right\rfloor = 0$, $\mathcal{G}_5^2 = \left\lfloor \frac{20}{20} \right\rfloor = 1$, and $\mathcal{G}_6^2 = \left\lfloor \frac{24}{20} \right\rfloor = 1$. Based on Lemma 2, since $\mathcal{G}_1^2 \neq \mathcal{G}_5^2$ then $n_1$ and $n_5$ does not have a common ancestor at level 2. Similarly, ($n_1$, $n_6$), ($n_2$, $n_5$), and ($n_2$, $n_6$) do not have common ancestors at the second level.

Since $\mathcal{G}_1^2 = \mathcal{G}_2^2$, $n_1$ and $n_2$ must have a common ancestor at level 2 (the first `book` element in Figure 1(b)). ■

Observe that by using Lemma 2 we can filter out leaf elements that belong to the same common ancestor directly for negative rooted paths.

**Theorem 1.** *Let $r_k$ and $r_m$ be two* RP*s in a query $Q$ on $D$. Let $N_k$ and $N_m$ be the sets of leaf elements that match $r_k$ and $r_m$, respectively in $D$. Let $n_i \in N_k$ and $n_j \in N_m$. For $\ell > 1$, $n_i$ must have the same ancestor as $n_j$ at level $\ell$ iff $\mathcal{G}_i^\ell = \mathcal{G}_j^\ell$.*

Note that Lemma 2 and Theorem 1 can also be used for internal elements since ancestor group identifier of an internal element of a subtree rooted at the NCA is identical to that of any leaf element in the subtree (Section 4.1). Also, it immediately follows from the above theorem that $n_i$ needs to be filtered out if $r_m$ is a negative RP in $Q$. Note that we ignore the trivial case of $\ell = 1$ [12].

*Example 2.* Assume that the `price` and `location` elements in Figure 1(b) match a normal and a negative RPs, respectively in a NOT-twig query. Hence, we want to filter out all leaf elements having the same ancestor as `location` at level 2. Let $n_i \in N_{price}$ and $n_j \in N_{location}$ where $N_{price}$ and $N_{location}$ are sets of leaf elements satisfying the normal and negative RPs, respectively. Here $N_{price} = \{n_2, n_6, n_7\}$, $N_{location} = \{n_3, n_{11}\}$, and $A_{2-1} = 20$. The AncestorDeweyGroup values of $n_2$, $n_6$, and $n_7$ are 4, 24, and 28, respectively. Similarly, AncestorDeweyGroup values of $n_3$ and $n_{11}$ are 8 and 44, respectively. Then, $\mathcal{G}_2^2 = \mathcal{G}_3^2 = 0$, $\mathcal{G}_6^2 = \mathcal{G}_7^2 = 1$, and $\mathcal{G}_{11}^2 = 2$. Consequently, based on Theorem 1 $n_2$ has to be filtered out as $n_2$ share the same ancestor as $n_3$ (at level 2) which matches the negative RP. ■

## 4.3 Evaluation of NOT-Twig Queries

We now discuss in detail how ancestor group identifiers are exploited for evaluating NOT-twig queries. As our focus is on not-predicates, for simplicity we assume that *edge*$(m_i, m_j)$ in a query is PC edge. Note that the proposed technique can easily support AD edges as discussed in Section 4.1.

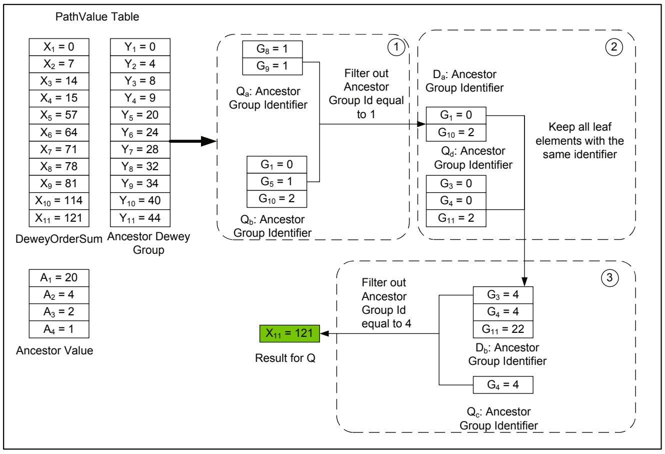


**Fig. 4.** Overview of NOT-twig evaluation

Consider the evaluation of the query $Q$ in Figure 1(a)(ii) on the XML document in Figure 1(b). Figure 4 depicts a step-by-step evaluation of $Q$. In this example, we consider the fragment of the PathValue table in Figure 3 for illustration. Note that for clarity, in Figure 4 we only show DeweyOrderSums and AncestorDeweyGroups in the PathValue table. The DeweyOrderSum and AncestorDeweyGroup of each leaf element are denoted as $X_i$ and $Y_i$, respectively, where $i$ is the LeafOrder value of the element. First, $Q$ is decomposed into the following normal rooted path patterns (without not-predicates). These paths are extracted from $Q$ in left-to-right order and consists of all root-to-leaf paths in $Q$ and the rightmost root-to-internal path representing the path after removing all qualifiers ($Q_a$: `/catalog/book/review`, $Q_b$: `/catalog/book/title`, $Q_c$: `/catalog/book/publisher/name/website`, $Q_d$: `/catalog/book/publisher/name`).

**Evaluation order of RPs.** If RPs are evaluated sequentially in left-to-right order ignoring the presence of negative RPs, then it will produce incorrect answers. Hence, we follow the following order. If the rooted path (say $r$) being evaluated is a negative RP then it is not evaluated immediately. On the other hand, if $r$ is a normal RP, then it is evaluated immediately. First, elements matching $r$ is evaluated with those that match the first preceding normal RP (if exists). Next, the elements will be evaluated with previously encountered negative RPs (if any) to filter out irrelevant elements. For example, in the aforementioned query $Q_a$ is not immediately evaluated as it is a negative RP. Next, the normal RP $Q_b$ is encountered. Since there does not exist any normal RP preceding $Q_b$, it is evaluated along with the negative RP $Q_a$. Next, the evaluation of the negative RP $Q_c$ is skipped and normal RP $Q_d$ is encountered. Since $Q_b$ is the first preceding normal RP, $Q_d$ is evaluated along with $Q_b$. Lastly, $Q_d$ is evaluated in conjunction with the previously recorded negative RP $Q_c$. Hence, the order of evaluation of the above query is: $Q_a$ and $Q_b$ (results are represented as $D_a$), $Q_d$ and $D_a$ (results are represented as $D_b$), and $D_b$ and $Q_c$.

**Evaluation of RPs.** In Step 1, the negative RP $Q_a$ and normal RP $Q_b$ are evaluated. Note that the NCA level of these RPs is 2. Since $Q_a$ is a negative RP, all elements that satisfy $Q_b$ but not $Q_a$ are required. Therefore, we can directly select these elements using Theorem 1 for level 2. All elements in the results of $Q_b$ that share same ancestor group identifiers with the results of $Q_a$ are removed. Since $\mathcal{G}_8^2 = \mathcal{G}_9^2 = \mathcal{G}_5^2 = 1$, $n_5$ will be removed. Therefore, this step returns elements $n_1$ and $n_{10}$ (denoted as $D_a$) as their ancestor group identifiers are not equal to 1. In Step 2, we compute the ancestor group identifiers of all elements satisfying $D_a$ and $Q_d$ and retrieve those elements that share the same identifiers. This results in the leaf elements $n_3$, $n_4$, and $n_{11}$ ($D_b$). Finally, we process the previous negative RP $Q_c$. We now retrieve all leaf elements in $D_b$ that are missing in $Q_c$ using Theorem 1. Here $\ell = 4$ (name element). Observe that for $D_b$, $\mathcal{G}_3^4 = \mathcal{G}_4^4 = 4$ and $\mathcal{G}_{11}^4 = 22$. For $Q_c$, $\mathcal{G}_4^4 = 4$. Since ancestor group identifier of $n_4$ satisfying $Q_c$ is identical to those of $n_3$ and $n_4$, we remove $n_3$ and $n_4$ from $D_b$ (Step 3). Since there are no more rooted paths, the final result is $n_{11}$.

## 4.4 SQL Translation Algorithm

**The Query Decomposition Phase.** First, given a NOT-twig query $Q$, the SQL translation algorithm decomposes $Q$ into a list of normal and negative rooted paths $T$. It extracts from $Q$ the root-to-leaf paths and rightmost root-to-internal path (in absence of qualifiers), and store them into a list $T$ in the following order. First, all root-to-leaf paths are inserted according to the left-to-right order of $Q$. Next, the root-to-internal path is added in $T$. The list also stores predicate information. We assume that $T$ has a *size* method which returns the total number of RPs in $T$ and a *countNotPred* method which returns the total number of negative RPs.

**The SQL Generation Phase.** This phase generates the SQL query $S_{not}$ for retrieving elements that satisfy $Q$. This query only retrieves the LeafOrder values of the matching elements. The algorithm is shown in Algorithm 1. Given a set of rooted paths $T$ of $Q$, the *generateSQLforNot* procedure outputs a SQL statement consisting of three clauses: *select_sql*, $from\_sql$, and *where_sql*. In the sequel we assume that a clause has an *add()* method which encapsulates some simple string manipulations and simple joins for constructing valid SQL statements. Also, the *NCAlevel()* function computes the level of an NCA in $Q$. We preprocess the PathId and RValue to reduce the number of joins.

For each rooted path $r_i \in T$, the procedure first checks if it is a negative RP. Recall that a negative RP is not evaluated immediately. Specifically, all *consecutive* negative RPs are recorded (using the counter *cntNotPred*) until the next normal RP is encountered (Lines 03–04). When a normal RP $r_i$ is encountered, it checks if it is a root-to-leaf path (Line 08). If it is then the algorithm generates the SQL fragment that retrieves the representative leaf elements by using instances of $r_i$'s PathId and BranchOrder values (Line 09). Next, the algorithm generates statement for NCA computation of normal RPs in the following ways.

**Algorithm 1.** Algorithm ***generateSQLforNot***.

```
   Input: A list of normal and negative RPs $T$
   Output: SQL query $S_{not}$
 1 Initialize $cntNotPred = 0$;
 2 for $(i = 1$ to $T.size())$ do
 3     if (rooted path $r_i$ is negative RP) then
 4         $cntNotPred{+}{+}$;
 5     else
 6         from_sql.add("PathValue AS $V_i$");
 7         where_sql.add("$V_i$.PathId IN $r_i$.getPathId()");
 8         if $(i < T.size())$ then
 9             where_sql.add("$V_i$.BranchOrder < $r_i$.level()");
10         if $(i > 1$ and $cntNotPred = 0)$ then
11             where_sql.add("$V_i$.AncestorDeweyGroup/AncestorValue ($r_{i-1}$.NCAlevel() - 1)
               = $V_{i-1}$.AncestorDeweyGroup/AncestorValue ($r_{i-1}$.NCAlevel() - 1");
12         else
13             set $x = cntNotPred$;
14             while $(x > 0)$ do
15                 where_sql.add("$V_i$.AncestorDeweyGroup/ AncestorValue
                   ($r_{i-x}$.NCAlevel()-1) NOT IN (");
16                 where_sql.add(select_sql.add( "$V_{i-x}$.AncestorDeweyGroup/
                   AncestorValue($r_{i-x}$.NCAlevel()-1)"));
17                 where_sql.add(from_sql.add("PathValue AS $V_{i-x}$"));
18                 where_sql.add(where_sql.add("$V_{i-x}$.PathId IN $r_{i-x}$.getPathId()))");
19                 where_sql.add(where_sql.add( "$V_{i-x}$.BranchOrder <$r_i$.level())"));
20                 $x$- - ;
21             if $(i - cntNotPred > 1)$ then
22                 where_sql.add("$V_i$.AncestorDeweyGroup/ AncestorValue
                   ($r_{i-cntNotPred-1}$.NCAlevel()-1) =
                   $V_{i-cntNotPred-1}$.AncestorDeweyGroup/
                   AncestorValue($r_{i-cntNotPred-1}$.NCAlevel()-1)");
23             set $cntNotPred = 0$;
24 select_sql.add("DISTINCT $V_i$.DocID, $V_i$.LeafOrder");
25 return $S_{not}$ = select_sql + from_sql + where_sql;
```

- *$r_i$ is the first* RP *in $T$:* Let $r_1$ $(i = 1)$ be a normal RP in $T$ (without not-predicate). In this case, $r_1$ does not have any preceding RP. Then, $r_1$ will not be evaluated immediately (conditions in Lines 10 and 21 are not satisfied) as a pair of RPs is required for NCA evaluation (Theorem 1).
- *$r_i$ is not the first* RP *in $T$ and $i > 1$:* In this case, the algorithm may have encountered a normal RP $r_j$ earlier $(j < i)$. Hence, if $countNotPred = 0$ it will execute Line 11 to generate the SQL statement to retrieve pairs of leaf elements that have NCA at the specified NCA level (based on Theorem 1). Otherwise, if $countNotPred > 0$ then the condition in Line 21 is true. Consequently, Line 22 will be used to generate the SQL fragment for NCA evaluation.

For all consecutive negative RPs, the procedure directly evaluates them using ancestor group identifiers (Lines 14-20). Specifically, Line 16 returns the ancestor group identifiers and Line 15 filters out elements based on Lemma 2. Note that the counter $cntNotPred$ will be reset to 0 whenever the procedure encounters a normal RP (Line 23).

**The Final SQL Generation Phase.** Finally, in this phase the final SQL query $S$ for retrieving entire subtrees that match $Q$ is generated. This procedure is

**Algorithm 2.** Algorithm ***finalSQLGen***

```
Input: SQL query $S_{not}$, number of RPs $x$, number of negative RPs $y$
Output: SQL query $S$
1 order_sql.add("DocID, LeafOrder") ;
2 select_sql.add("$V_{x+1}$.LeafValue, ... $V_{x+1}$.LeafOrder");
3 from_sql.add("(" + $S_{not}$ + ") AS $V_x$ INNER JOIN PathValue $V_{x+1}$ ON $V_{x+1}$.DocID = $V_x$.DocID
  AND $V_{x+1}$.LeafOrder = $V_x$.LeafOrder");
4 where_sql.add("$V_{x+1}$.PathID IN $r_x$.getPathID()");
5 if ($x - y > 1$) then
6     option_sql.add("FORCE ORDER, ORDER GROUP");
7 else
8     option_sql.add("ORDER GROUP");
9 return $S$ = select_sql + from_sql + where_sql + order_sql + option_sql;
```

```
01    SELECT V5.LeafValue, V5.PathId, V5.BranchOrder, V5.DeweyOrderSum,
              V5.AncestorDeweyGroup, V5.DocId, V5.LeafOrder
02    FROM (
03          SELECT   DISTINCT V4.DocID, V4.LeafOrder
04          FROM     PathValue AS V2, PathValue AS V4
05          WHERE    V2.PathId IN (3) AND V2.BranchOrder < 3
06          AND      V2.AncestorDeweyGroup / 20 NOT IN (
07                        SELECT    V1.AncestorDeweyGroup / 20
08                        FROM      PathValue  AS V1
09                        WHERE     V1.PathId IN (2) AND V1.BranchOrder < 3 )
10          AND      V4.PathId IN (4, 5)
11          AND      V4. AncestorDeweyGroup / 20 = V2. AncestorDeweyGroup / 20
12          AND      V4.AncestorDeweyGroup / 2 NOT IN (
13                        SELECT    V3.AncestorDeweyGroup / 2
14                        FROM      PathValue AS V3
15                        WHERE     V3.PathId IN (5) AND V3.BranchOrder < 5 )
16    ) AS V4 INNER JOIN PathValue AS V5 ON V5.DocID = V4.DocID
       AND V5.LeafOrder = V4.LeafOrder
17    WHERE    V5.PathID IN (4, 5)
18    ORDER BY DocId, LeafOrder
19    OPTION (FORCE ORDER, ORDER GROUP)
```

**Fig. 5.** Translated SQL query

outlined in Algorithm 2 and contains five clauses: *select_sql*, *from_sql*, *where_sql*, *order_sql*, and *option_sql*. It includes an addition instance of PathValue $V_{x+1}$ which uses the same path in the PathValue table $V_x$ representing the rightmost root-to-internal path in $S_{not}$ (Line 04). $V_{x+1}$ is joined on DocID and LeafOrder attributes with $V_x$ to retrieve entire subtrees of matched elements (Line 03). Since the results must be in document order, the tuples are sorted according to DocID and LeafOrder attributes using the *order_sql* clause (Line 01). Lastly, the option clause (*option_sql*) is used to enforce the `distinct` and `order by` operations to use sort operator using the `ORDER GROUP` query hint (Lines 05 - 08). Also, if there exists at least one normal root-to-leaf path in $Q$ then `FORCE ORDER` hint is used to enforce a "left-to-right" join order on the translated SQL query (Line 06). The performance benefits of join order enforcement is highlighted in [4, 7, 11]. Note that the translated SQL has at least one instance of PathValue table representing the normal root-to-internal path. Further, if all root-to-leaf paths in $Q$ are negative RPs, then join order enforcement is discarded as these paths will be evaluated by subqueries (generated by Lines 15–19 in Algorithm 1).

Reconsider the query $Q$ in Section 4.3. The list of root-to-leaf and root-to-internal paths $T$ is: $[r_1 = Q_a, r_2 = Q_b, r_3 = Q_c, r_4 = Q_d]$. The translated SQL is shown in Figure 5. The reader may refer to [12] for details related to this example.

## 5 Performance Study

In this section, we present the experiments conducted to evaluate the performance of our proposed approach and report some of the results obtained. A

| Id | Query |
|---|---|
| Q1 | /catalog/item/publisher/contact_information[not(website) and phone_number] |
| Q2 | /catalog/item/publisher[not(contact_information/website) and contact_information/phone_number]/name |
| Q3 | /catalog/item[not(subject) and not(description)]/title |
| Q4 | /catalog/item/authors/author[not(biography)]/name |
| Q5 | /catalog/item[not(description) and not(subject)]/authors/ author[not(biography)]/contact_information[not(email_address)]/ mailing_address |
| Q6 | /catalog/item[pricing/quantity_in_stock]/authors/author[not(biography) and contact_information/mailing_address/name_of_country]/ name[not(middle_name)]/first_name |
| Q7 | /catalog/item[date_of_release]/pricing[not(when_is_available)]/cost |
| Q8 | /catalog/item[not(media) and not(pricing/quantity_in_stock)]/ publisher[not(contact_information/FAX_number)] |
| Q9 | /catalog/item[title and not(subject) and not(publisher/contact_information/ website)]/authors/author[biography]/date_of_birth |
| Q10 | /catalog/item[not(media) and not(subject)]/publisher[contact_information/ phone_number]/name |
| Q11 | /catalog/item[not(description)]/authors/author[not(contact_information/ email_address)]/biography |
| Q12 | /catalog/item[not(media) and not(attributes)]/publisher/ contact_information[not(website) and not(FAX_number)] |

(a) XBench query sets

| Id | Query | No. of matching subtrees | | |
|---|---|---|---|---|
| | | U28 | U283 | U2843 |
| UQ1 | /uniprot/entry[not(geneLocation/name) and not(comment/location)] | 3212 | 35277 | 354813 |
| UQ2 | /uniprot/entry[not(organismHost) and not(evidence) and geneLocation] | 157 | 1622 | 16392 |
| UQ3 | /uniprot/entry[not(gene)]/ protein[not(component) and not(domain)] | 189 | 1626 | 16479 |
| UQ4 | /uniprot/entry[not(geneLocation) and not(protein/domain)]/comment[not(note) and not(event)] | 13835 | 140196 | 1403859 |
| UQ5 | //comment[not(event)]/following::text | 14620 | 145799 | 1461240 |
| UQ6 | //comment[not(note) and not(event)]/ preceding::text | 14620 | 145799 | 1461240 |
| UQ7 | /uniprot/entry/comment[not(note) and not(event)]/preceding::component | 160 | 1576 | 15599 |
| UQ8 | /uniprot/entry[not(geneLocation)]/protein/ following::component | 158 | 1574 | 15597 |
| UQ9 | /uniprot/entry/protein[not(component)]// domain//name | 185 | 1433 | 13914 |

(b) UniProt query sets

**Fig. 6.** Query sets

more detailed results is available in [12]. Prototype for our ancestor group-based approach (denoted by AG-SX) was implemented by extending SUCXENT++ using Java JDK 1.6. The experiments were conducted on an Intel Pentium IV 3GHz machine running on Windows XP Service Pack 2 with 2GB RAM. The RDBMS used was MS SQL Server 2008 Developer Edition.

We are not aware of any existing tree-unaware approaches that have undertaken a systematic study to evaluate NOT-twig queries. Hence, we compared our approach to the native XML supports of *XSysA* and *XSysB* (Recall from Section 1). For all these approaches appropriate indexes were created. Prior to our experiments, we ensure that statistics had been collected. The bufferpool of the RDBMS was cleared before each run. The queries in AG-SX were executed in the *reconstruct* mode [13] where not only the internal elements are selected, but also all descendants of those elements. Each query was executed 6 times and the results from the first run were always discarded. All rows were fetched from the answer set; however, they were not sent to output. Note that we did not select *TwigStackList*$\neg$ [16] and $NJoin$ [10] as we were unable to get the implementation from the authors. However, an intuitive comparison with these approaches is discussed later.

**Datasets.** We use XBench DCSD [15] shown in Figure 2(a) as synthetic data set. We also modified the data set so that we can control the number of subtrees (denoted as $K$) that matches a NOT-twig query and the number of instances of the rooted paths in the XML document. We set $K \in \{100, 500\}$. We use the UNIPROT dataset shown in Figure 2(b) as real-world data set. Since the original UNIPROT data is 2.8GB in size (denoted as U2843), we also truncated this document into smaller XML documents of sizes 28MB and 284MB (denoted as U28 and U284, respectively) to study scalability of various systems.

**Querysets.** Figure 6 depicts the benchmark queries. As our primary objective is to assess the performance of not-predicates evaluation, we choose two categories of queries. In the first category ($Q1 - Q12$ and $UQ1 - UQ4$), we fix the XPath axis in the twigs to *child* and generate queries by varying the number of

| Id | DC10 | | | DC100 | | | DC1000 | | |
|---|---|---|---|---|---|---|---|---|---|
| | AG-SX | XSysA | XSysB | AG-SX | XSysA | XSysB | AG-SX | XSysA | XSysB |
| Q1 | 111 | 346 | 205 | 377 | 588 | 1,761 | 3,022 | 3,303 | 48,288 |
| Q2 | 107 | 475 | 197 | 183 | 806 | 2,075 | 1,273 | 3,359 | 47,208 |
| Q3 | 249 | 595 | 178 | 517 | 952 | 1,712 | 1,363 | 5,394 | 42,980 |
| Q4 | 223 | 574 | 331 | 495 | 1,585 | 3,082 | 3,385 | 12,348 | 59,087 |
| Q5 | 362 | 2,501 | 200 | 925 | 3,055 | 1,564 | 6,804 | 10,879 | 44,289 |
| Q6 | 256 | 3,038 | 631 | 624 | 3,950 | 5,889 | 4,205 | 14,648 | 93,219 |
| Q7 | 166 | 1,119 | 306 | 376 | 1,678 | 2,789 | 1,742 | 6,283 | 49,235 |
| Q8 | 279 | 1,228 | 290 | 684 | 1,620 | 2,235 | 4,616 | 7,209 | 47,209 |
| Q9 | 179 | 1,436 | 324 | 631 | 2,240 | 2,822 | 5,313 | 9,331 | 63,980 |
| Q10 | 135 | 1,642 | 230 | 268 | 2,357 | 1,962 | 1,669 | 8,910 | 50,842 |
| Q11 | 188 | 1,008 | 206 | 318 | 1,559 | 1,708 | 2,709 | 9,748 | 60,109 |
| Q12 | 382 | 972 | 231 | 842 | 1,366 | 2,026 | 4,997 | 6,921 | 46,924 |

(a) K=100

| Id | DC10 | | | DC100 | | | DC1000 | | |
|---|---|---|---|---|---|---|---|---|---|
| | AG-SX | XSysA | XSysB | AG-SX | XSysA | XSysB | AG-SX | XSysA | XSysB |
| Q1 | 124 | 883 | 249 | 452 | 1,165 | 1,796 | 3,148 | 4,156 | 47,556 |
| Q2 | 104 | 964 | 248 | 209 | 1,325 | 1,843 | 1,378 | 4,244 | 45,871 |
| Q3 | 232 | 1,576 | 213 | 392 | 1,732 | 1,601 | 1,401 | 5,643 | 45,209 |
| Q4 | 184 | 1,439 | 334 | 475 | 2,332 | 2,823 | 3,790 | 11,298 | 57,309 |
| Q5 | 346 | 3,351 | 296 | 908 | 4,000 | 1,791 | 7,206 | 12,134 | 42,109 |
| Q6 | 269 | 4,022 | 686 | 641 | 4,712 | 5,767 | 4,543 | 16,180 | 94,770 |
| Q7 | 162 | 1,967 | 321 | 410 | 2,832 | 2,777 | 1,806 | 7,282 | 50,259 |
| Q8 | 297 | 2,247 | 241 | 727 | 2,691 | 2,163 | 4,705 | 8,937 | 46,295 |
| Q9 | 171 | 2,175 | 479 | 655 | 3,369 | 2,694 | 4,953 | 10,153 | 63,820 |
| Q10 | 154 | 2,386 | 280 | 275 | 3,431 | 2,024 | 1,641 | 9,975 | 51,872 |
| Q11 | 212 | 1,719 | 288 | 331 | 2,593 | 2,001 | 2,711 | 10,887 | 59,952 |
| Q12 | 404 | 1,877 | 270 | 867 | 2,490 | 2,013 | 5,009 | 7,424 | 49,825 |

(b) K= 500

| Id | U28 | | | U284 | | | U2843 |
|---|---|---|---|---|---|---|---|
| | AG-SX | XSysA | XSysB | AG-SX | XSysA | XSysB | AG-SX |
| UQ1 | 1,530 | 20,399 | 2,457 | 12,809 | 207,982 | 24,429 | 128,208 |
| UQ2 | 1,130 | 1,541 | 450 | 21,740 | 11,279 | 4,391 | 86,816 |
| UQ3 | 494 | 2,299 | 591 | 485 | 15,038 | 5,342 | 3,302 |
| UQ4 | 505 | 5,899 | 1,977 | 1,210 | 47,386 | 20,819 | 6,681 |
| UQ5 | 1,604 | NS | NS | 8,878 | NS | NS | 86,555 |
| UQ6 | 1,829 | NS | NS | 12,488 | NS | NS | 113,547 |
| UQ7 | 815 | NS | NS | 3,615 | NS | NS | 31,310 |
| UQ8 | 659 | NS | NS | 2,896 | NS | NS | 26,265 |
| UQ9 | 2,040 | 4,965 | 3,988 | 9,420 | 65,624 | 34,716 | 124,900 |

(c) Query performance on UniProt data sets

**Fig. 7.** Query evaluation times of AG-SX, *XSysA*, and *XSysB* (in msec.)

normal and negative rooted paths, number of NCA nodes, and structure of twigs. For instance, $Q3 - Q5$, $Q8$, $Q11$, $Q12$, $UQ1$, $UQ3$, and $UQ4$ are queries with purely not-predicates while the remaining queries contain a mixture of normal and negative rooted paths. The number of instances of root-to-leaf paths that matches the query set varies between 150 and $2,035,889$. In the second category ($UQ5$-$UQ9$), we include different XPath axes (e.g., descendant, following, preceding) in the NOT-twigs to study the performance of these queries in the presence of such axes.

## 5.1 Query Evaluation Times

Figure 7 depicts the NOT-twig query evaluation times. As *XSysA* and *XSysB* are unable to handle XML documents having size larger than 2GB, no query evaluation times are reported for these approaches on U2843 data set. Also, as AG-SX is orders of magnitude faster than the not-predicate evaluation approach on the original SUCXENT++ (see [12]), we only report query evaluation times of AG-SX. The symbol NS in Figure 7 denotes that the query is not currently supported in the current version of a particular system.

We observe that AG-SX significantly outperforms both *XSysA* and *XSysB* for majority of the queries (highest observed factors being 37 and 40, respectively). As the data size increases, the performance gap between AG-SX and these approaches increases. Particularly, we noticed that except for $Q5$, our proposed approach is at least 9 times faster than *XSysB* for all values of $K$. For the real-world data sets (U28 and U284), AG-SX is faster than *XSysA* and *XSysB* for 90% and 80% of the benchmark queries, respectively. In summary, AG-SX outperforms *XSysA* and *XSysB* primarily due to the effectiveness of the former approach to generate a relatively simple SQL statement, which exploits ancestor group identifiers to efficiently evaluate common ancestors and not-predicates using the equality property (Theorem 1). Also, interestingly *XSysA* is less efficient than *XSysB* for smaller data sets (DC10 and DC100). However, it is faster than *XSysB* for DC1000.

**Comparison with** $TwigStackList\neg$ **[16] and** $NJoin$ **[10]:** Based on the results reported in [10, 16] we can make the following observations. For a data set of size 100MB and less than 2.5 million nodes, the average running time of benchmark NOT-twig queries using *TwigStackList*¬ is $15 - 30s$ [16] whereas majority of our queries on similar data sets take less than a second to retrieve results. In [10], it is shown that *NJoin* is 2-3 faster than *TwigStackList*¬ for simple NOT-twig queries. Based on this observation, we expect AG-SX to outperform these approaches.

## 6 Conclusions and Future Work

In this paper, we present an efficient strategy to evaluate NOT-twig queries in a tree-unaware relational environment. We extended the encoding scheme of dewey-based SUCXENT++ [11] by adding two new labels, namely AncestorValue and AncestorDeweyGroup, that enable us to efficiently filter out elements satisfying a not-predicate by comparing their ancestor group identifiers. We proposed a novel NOT-twig query evaluation algorithm that reduce useless structural comparisons by exploiting these labels. Our results showed that the our proposed approach have superior performance compared to existing state-of-the-art tree-unaware and native approaches. In future, we plan to investigate if some of the optimization techniques proposed in [4] (e.g., choosing right join algorithms, eliminating redundant ordering (if any)) are beneficial for evaluating NOT-twig queries in our proposed framework.

## References

1. Al-Khalifa, A., Jagadish, H.V.: Multi-level Operator Combination in XML Query Processing. In: ACM CIKM (2002)
2. Bhowmick, S.S., Leonardi, E., Sun, H.: Efficient Evaluation of High-Selective XML Twig Patterns with Parent Child Edges in Tree-Unaware RDBMS. In: ACM CIKM (2007)
3. Boncz, P., Grust, T., et al.: MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In: SIGMOD (2006)
4. Georgiadis, H., Vassalos, V.: Xpath on Steroids: Exploiting Relational Engines for Xpath Performance. In: SIGMOD (2007)
5. Georgiadis, H., et al.: Cost-based Plan Selection for XPath. In: SIGMOD (2009)
6. Gou, G., Chirkova, R.: Efficiently Querying Large XML Data Repositories: A Survey. IEEE TKDE 19(10) (2007)
7. Grust, T., et al.: Why Off-the-Shelf RDBMSs are Better at XPath Than You Might Expect. In: SIGMOD (2007)
8. Jiao, E., Ling, T.-W., Chan, C.-Y.: PathStack: A Holistic Path Join Algorithm for Path Query with Not-Predicates on XML Data. In: Zhou, L.-z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, pp. 113–124. Springer, Heidelberg (2005)
9. Li, H., Li Lee, M., Hsu, W.: A Path-Based Labeling Scheme for Efficient Structural Join. In: Bressan, S., Ceri, S., Hunt, E., Ives, Z.G., Bellahsène, Z., Rys, M., Unland, R. (eds.) XSym 2005. LNCS, vol. 3671, pp. 34–48. Springer, Heidelberg (2005)

10. Li, H., Lee, M.-L., et al.: A Path-Based Approach for Efficient Structural Join with Not-Predicates. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 31–42. Springer, Heidelberg (2007)
11. Seah, B.-S., Widjanarko, K.G., et al.: Efficient Support for Ordered XPath Processing in Tree-Unaware Commercial Relational Databases. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 793–806. Springer, Heidelberg (2007)
12. Soh, K.-H., Bhowmick, S.S.: Efficient Evaluation of not-Twig Queries in A Tree-Unaware RDBMS. Technical Report (December 2009), `http://www.cais.ntu.edu.sg/~assourav/TechReports/NotTwig-TR.pdf`
13. Tatarinov, I., Viglas, S., et al.: Storing and Querying Ordered XML Using a Relational Database System. In: SIGMOD (2002)
14. Yoshikawa, M., et al.: XRel: A Path-based Approach to Storage and Retrieval of XML documents Using Relational Databases. ACM TOIT 1(1) (2001)
15. Yao, B., Tamer Özsu, M., Khandelwal, N.: XBench: Benchmark and Performance Testing of XML DBMSs. In: ICDE (2004)
16. Yu, T., Ling, T.-W., Lu, J.: TwigStackList¬: A Holistic Twig Join Algorithm for Twig Query with Not-Predicates on XML Data. In: Li Lee, M., Tan, K.-L., Wuwongse, V. (eds.) DASFAA 2006. LNCS, vol. 3882, pp. 249–263. Springer, Heidelberg (2006)