Efficient Support for Ordered XPath Processing in Tree-Unaware Commercial Relational Databases

Boon-Siew Seah[1,2] Klarinda G. Widjanarko[1,2] Sourav S. Bhowmick[1,2]
Byron Choi [1] Erwin Leonardi[1,2]

[1]School of Computer Engineering, Nanyang Technological University, Singapore
[2]Singapore-MIT Alliance, Nanyang Technological University, Singapore
{821123145823,klarinda,assourav,kkchoi,lerwin}@ntu.edu.sg

May 23, 2007

Abstract

In this paper, we present a novel ordered XPATH evaluation in *tree-unaware* RDBMS. The novelties of our approach lies in the followings. (a) We propose a novel XML storage scheme which comprises *only* leaf nodes, their corresponding data values, order encodings and their root-to-leaf paths. (b) We propose an algorithm for mapping ordered XPATH queries into SQL queries over the storage scheme. (c) We propose an optimization technique that enforces all mapped SQL queries to be evaluated in a "left-to-right" join order. By employing these techniques, we show, through a comprehensive experiment, that our approach not only scales well but also performs better than some representative tree-unaware approaches on more than 65% of our benchmark queries with the highest observed gain factor being 1939. In addition, our approach reduces significantly the performance gap between tree-aware and tree-unaware approaches and even outperforms a state-of-the-art tree-aware approach for certain benchmark queries.

1 Introduction

With the rapid emergence of XML as the *de facto* standard for exchanging data on the Web, the interest in efficiently querying growing XML data sources has increased. One of the salient features of XML data is that it is order-sensitive. Supporting an ordered data model of XML as well as ordered XML queries, *ordered* XPATH *axes and position predicates* in particular, have been the key to successful XML applications, *e.g.,* [12]. In this paper, we present a novel approach to efficiently evaluate ordered XPATH queries in a relational database.

Current approaches for evaluating XPATH expressions in relational databases can be arguably categorized into two representative types. They either resort to encoding XML data as tables and translating XML queries into relational queries [3, 4, 5, 6, 10, 11, 15] or store XML data as a rich data type and process XML queries by enhancing the relational infrastructure [9]. The former approach can further be classified into two representative types. Firstly, a host of work on processing XPATH queries on *tree-unaware* relational databases has been reported [5, 10, 11] – these approaches do not modify the database kernels. Secondly, there have been several efforts on enabling relational databases to be *tree-aware* by invading the database kernel to implement XML support [3, 4, 6, 15]. It has been shown that the latter approaches appear scalable and, in particular, perform orders of magnitude faster than some tree-unaware approaches [3, 6].

In this paper, we focus on supporting *ordered* XPATH evaluation in a *tree-unaware* relational environment. There is a considerable benefit in such an approach with respect to portability and ease of implementation on top of an off-the-shelf RDBMS. Although a diverse set of strategies for evaluating XML queries in tree-unaware relational environment have been recently proposed, few have undertaken a comprehensive study on evaluating ordered XPATH queries. Tatarinov *et al.* [12] is the first to show that it is indeed possible to support ordered

XPATH queries in relational databases. However, this approach does not scale well with large XML documents. In fact, as we shall show in Section 7, the GLOBAL-ORDER approach in [12] failed to return results for 20% of our benchmark queries on 1GB dataset in 60 minutes. Furthermore, this approach resorts to manual tuning of the relational optimizer when it failed to produce good query plans. Although such a manual tuning approach works, it is a cumbersome solution.

In this paper, we address the above limitations by proposing a novel scheme for ordered XPATH query processing. Our storage strategy is built on top of SUCXENT++ [10], by extending it to support efficient processing of ordered axes and predicates. SUCXENT++ is designed primarily for query-mostly workloads. We exploit SUCXENT++'s strategy to store leaf nodes, their corresponding data values, auxiliary encodings and root-to-leaf paths. In contrast, some approaches, *e.g.,* [6, 15], explicitly store information for all nodes of an XML document. Specifically, the followings remark the novelties of our storage scheme. (1) For each level of an XML document, we store an attribute called RValue which is an enhancement of the original RValue, proposed in [10], for processing recursive XPATH queries. (2) For each leaf node we store three additional attributes namely BranchOrder, DeweyOrderSum and SiblingSum. These attributes are the foundation for our ordered XPATH processing. The key features of these attributes are that they enable us (a) to compare the order between non-leaf nodes by comparing the order between their *first descendant leaf* nodes only; and (b) to determine the nearest common ancestor of two leaf nodes efficiently. As a result, it is not necessary to store the order information of non-leaf nodes. Furthermore, given any pair of nodes, these attributes enable us to evaluate position-based predicates efficiently.

As highlighted in [12], relational optimizers may sometimes produce poor query plans for processing XPATH queries. In this paper, we undertake a novel strategy to address this issue. As opposed to manual tuning efforts, we propose an *automatic* approach to enforce the optimizer to replace previously generated poor plans with probably better query plans, as verified by our experiments. Unlike tree-aware schemes, our technique is *non-invasive* in nature. That is, it can easily be incorporated *without modifying the internals* of relational optimizers. Specifically, we enforce a relational optimizer to follow a *"left-to-right"* join order and enforce the relational engine to evaluate the mapped SQL queries according to the XPATH steps specified in the query. The good news is that this technique can select better plans for the majority of our benchmark queries across all benchmark datasets. As we shall see in Section 7, the performance of previously-inefficient queries in SUCXENT++ is significantly improved. The highest observed gain factor is 59. Furthermore, queries that failed to finish in 60 minutes were able to do so now, in the presence of such a join-order enforcement. This is indeed stimulating as it shows that some sophisticated internals of relational optimizers not only are irrelevant to XPATH processing but also often confuse XPATH query optimization in relational databases. Overall a "join-order-conscious" SUCXENT++ significantly outperforms both GLOBAL-ORDER and SHARED-INLINING[11] in at least 65% of the benchmark queries with the highest observed gain factors being 1939 and 880, respectively. To the best of our knowledge, this is the first effort on exploiting a non-invasive automatic technique to improve query performance *in the context of* XPATH *evaluation in relational environment*.

Recently, [3] showed that MONETDB is among the most efficient and scalable tree-aware relational-based XQuery processor and outperforms the current generation of XQuery systems significantly. Consequently, we investigated how our proposed technique compared to MONETDB. Our study revealed some interesting results. First, although MONETDB is 11-164 and 3-74 times faster than GLOBAL-ORDER and SHARED-INLINING, respectively, for the majority of the benchmark queries, this performance gap is significantly reduced when MONETDB is compared to SUCXENT++. Our results show that not only MONETDB is now 1.3-16 times faster than SUCXENT++ with join-order enforcement but surprisingly our approach is faster than MONETDB for 33% of benchmark queries! Additionally, MONETDB (Win32 builds) failed to shred 1GB dataset as it is vulnerable to the virtual memory fragmentation in Windows environment. Note that, this is in contrary to the results in [3] where MONETDB was built on top of Linux 2.6.11 operating system (8GB RAM), using a 64-bit address space, and was able to efficiently shred 11GB dataset.

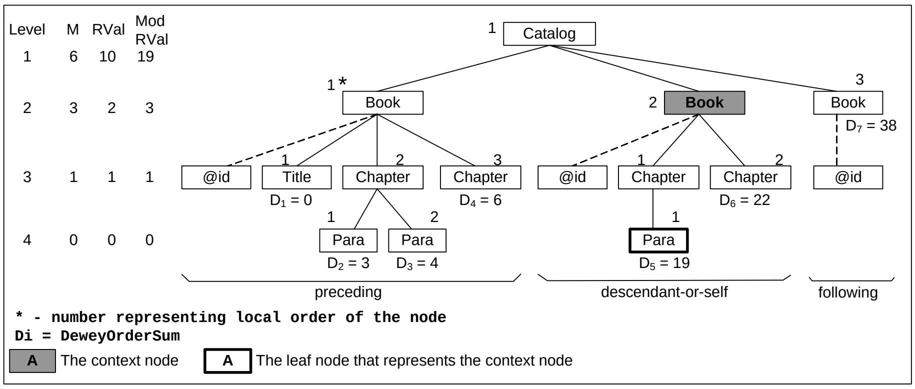
In summary, the main contributions of this paper are as follows. In Section 4, we describe our novel schema-oblivious relational storage scheme for XML. In Section 5, we present how ordered XPATH queries are supported in our proposed storage scheme. In Section 6, we proposed a novel "left-to-right" join order-based technique to

Document (<u>DocId</u>, Name)
Path (<u>PathId</u>, PathExp)
PathValue(<u>DocId, PathId, LeafOrder</u>, BranchOrder, BranchOrderSum, LeafValue)
DocumentRValue (<u>DocId, Level</u>, RValue)

(a) Original Schema of SUCXENT++

Document (<u>DocId</u>, Name)
Path (<u>PathId</u>, PathExp)
PathValue(<u>DocId, DeweyOrderSum</u>, PathId, BranchOrder, LeafOrder, SiblingSum, LeafValue)
Attribute (<u>DocId, LeafOrder, PathId</u>, LeafValue)
DocumentRValue (<u>DocId, Level</u>, RValue)

(b) Modified Schema of SUCXENT++



(c) XML Data

Figure 1: SUCXENT++ schema and example of XML data

improve query plan selection of relational query optimizers. To the best of our knowledge, this is the first effort on exploiting such non-invasive automatic technique to improve query performance *in the context of* XPATH *processing in relational environment*. Through an extensive experimental study in Section 7, we show that our approach significantly outperforms existing tree-unaware approaches for ordered XPATH queries. Additionally, our approach reduces significantly the performance gap between tree-aware and tree-unaware approaches and even outperform a state-of-the-art tree-aware approach for certain benchmark queries.

2 Related Work

Most of the previous tree-unaware approaches, except [12], focused on proposing efficient evaluation for `children` and `descendant-or-self` axes and positional predicates in XPATH queries. In this paper, the main focus is on the evaluation for `following`, `preceding`, `following-sibling`, and `preceding-sibling` axes as well as *position-based* and *range* predicates. All previous approaches, reported query performance on small/medium XML documents – smaller than 500 MB. We investigate query performance on large synthetic and real datasets. This gives insights on the scalability of the state-of-the-art tree-unaware approaches for ordered XML processing.

Compared to the tree-aware schemes [3, 4, 6, 15], our technique is *tree-unaware* in the sense that it can be built on top of any commercial RDBMS without modifying the database kernel. The approaches in [4, 15] do not provide a systematic and comprehensive effort for processing ordered XPATH queries. Although the scheme presented in [3, 4, 6] can support ordered axes, no comprehensive performance study has demonstrated with a variety of ordered XPATH queries. Furthermore, these approaches did not exploit the "left-to-right" join order technique to improve query plan selection.

In [12], Tatarinov *et al.* proposed the first solution for supporting ordered XML query processing in a relational database. A modified EDGE table [5] was the underlying storage scheme. They described three order encoding methods: *global*, *local*, and *dewey* encodings. The best query performance was achieved with the *global* encoding for query-mostly workloads and with *dewey* encoding for a mix of queries and updates. Our focus differs from the above approach in the following ways. First, we focus on query-mostly workloads. Second, we consider a novel order-conscious storage scheme that is more space- and query-efficient and scalable when compared to the *global* encoding.

3 Background on SUCXENT++

Our approach for ordered XPATH processing relies on the SUCXENT++ approach [10]. We begin our discussion by briefly reviewing the storage scheme of SUCXENT++. Foremost, in the rest of the paper, we always assume

DocRValue

DocId	Level	RValue
1	1	10
1	2	2
1	3	1

PathValue

DocId	Leaf Order	Branch Order	PathId	Dewey OrderSum	Sibling Sum	Leaf Value
1	1	0	2	0	0	D1
1	2	2	3	3	0	D2
1	3	3	3	4	1	D3
1	4	2	4	6	3	D4
1	5	1	3	19	19	D5
1	6	2	4	22	22	D6
1	7	1	5	38	38	D7

Path

PathId	PathExp
1	.catalog#.book#.@id#
2	.catalog#.book#.title#
3	.catalog#.book#.chapter#.para#
4	.catalog#.book#.chapter#
5	.catalog#.book#

Attribute

DocId	LeafOrder	PathId	LeafValue
1	1	1	book 01
1	5	1	book 02
1	7	1	book 03

Figure 2: XML data in RDBMS

document order in our discussions. The SUCXENT++ schema is shown in Figure 1(a). 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 leaf node n in T, we shall create a tuple in the PathValue table. We now elaborate the meaning of the attributes of this relation.

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 nearest common ancestor of n_1 and n_2. That is, n_1 and n_2 *intersect* at the BranchOrder level. The data value of n is stored in n.LeafValue.

To discuss BranchOrderSum 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. Then, RValue of level ℓ, denoted as R_ℓ, is 1 if ℓ = L_{max}. Otherwise, $R_\ell = R_{\ell+1} \times |M_{\ell+1}| + 1$. Now we are ready to define the BranchOrderSum attribute. Let N to be the set of leaf nodes preceding a leaf node n. n.BranchOrderSum is 0 if n.LeafOrder = 1 and $\sum_{m \in N} R_{m.BranchOrder}$ otherwise.

Based on the definitions above, Prakash *et al.* [10] defined Property 1 (below) which is essential to determine ancestor-descendant relationships efficiently.

Property 1 Given two leaf nodes n_1 and n_2, $|n_1$.BranchOrderSum - n_2.BranchOrderSum$| < R_\ell$ implies the nearest common ancestor of n_1 and n_2 is at a level greater than ℓ. □

4 Extensions of SUCXENT++

To support ordered XML queries, the order information of nodes must be captured in the XML storage scheme. Unfortunately the LeafOrder and BranchOrderSum attributes only encode the global order of all leaf nodes. Since (order) information of non-leaf nodes is not explicitly stored, it must be derived from the attributes of leaf nodes.

We now present how the original SUCXENT++ schema is extended to process ordered XPath queries efficiently. First, we move information related to attribute nodes from PathValue table to a new Attribute table. Second, we introduce new attributes to encode the relative order between (both non-leaf and leaf) nodes. Specifically, DeweyOrderSum and SiblingSum are introduced to replace BranchOrderSum. Second, the definition of RValue is modified such that RValue and DeweyOrderSum preserve the properties presented [10]. The modified schema is shown in Figure 1(b) and Figure 2 shows the shredded version of the example XML document.

```
Algorithm to Compute DeweyOrderSum

for each non-attribute leaf node n_i arranged in document order {

    if (n_i.branchOrder > 0) {
        n_i.DeweyOrderSum = dSum[ n_i.branchOrder + 1 ] + ModRValue( n_i.branchOrder );
        for ( level = n_i.branchOrder + 1; level <= maxDepthOfXMLDoc; level++ )
            dSum[level] = n_i.DeweyOrderSum;
    }
    else {    //initialization for the first leaf node
        n_i.DeweyOrderSum = 0;
        for ( level = 1; level <= maxDepthOfXMLDoc; level++ )
            dSum[level] = 0;
    }

}
```

Figure 3: Algorithm to compute DeweyOrderSum

4.1 Attribute Table

The PathValue table originally stored information related to both element and attribute nodes. However, to avoid mixing the order of element and attribute nodes, we separate the attribute nodes into Attribute table. The Attribute table consists of the following columns: DocId, LeafOrder, PathId, LeafValue. As we shall see later, a non-leaf node can be represented by the first descendant leaf nodes. Therefore, an attribute node is identified by DocId and LeafOrder of its parent node and its PathId.

4.2 Modified RValue Attribute

Conceptually, RValue is used to encode the level of the nearest common ancestor of any pairs of leaf nodes. To ensure a property like Property 1 holds after modifications, intuitively, we "magnify" the gap between RValues, as shown in Definition 1. Relative order information is then captured in these gaps.

Definition 1 [ModifiedRValue] *Let L_{max} be the largest level of an* XML *tree T. **ModifiedRValue** 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 1(c). $L_{max} = 4$. The values of $|M_1|$, $|M_2|$, and $|M_3|$ are 6, 3, and 1, respectively. Then, $R'_3 = 1$, $R'_2 = 2 \times 1 \times |M_3| + 1 = 3$, and $R'_1 = 2 \times 3 \times |M_2| + 1 = 19$. To ensure the evaluation of queries other than ordered XPATH queries is not affected by the above modifications, the RValue attribute in DocumentRValue stores $\frac{R'_\ell - 1}{2} + 1$ instead of R'_ℓ.

4.3 DeweyOrderSum and SiblingSum Attributes

Next, we define the first attribute related to ordered XPATH processing. Consider the path query `/catalog/book[1]/chapter[1]` and Figure 1(c). Since only leaf nodes are stored in the PathValue table, the new attribute DeweyOrderSum of leaf nodes captures order information of the non-leaf nodes. At first glance, a simple representation of the order information could be a Dewey path. For instance, the Dewey path of the first `chapter` node of the first `book` node is "1.1.2". However, using such Dewey paths has two major drawbacks. Firstly, string matching of Dewey paths can be computationally expensive. Secondly, simple lexicographical comparisons of two Dewey paths may not always be accurate [12]. Comparing "1.2" and "10.2" in lexicographical order will indicate that "10.2" appears before "1.2" [12]. Hence, we define DeweyOrderSum for this purpose:

Definition 2 [DeweyOrderSum] *Consider an* XML *document T and a leaf node n at level ℓ in T.* $\mathsf{Ord}(n,k) = i$ 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 $\Phi(j)$=[$\mathsf{Ord}(n,j)$-1]$\times R'_{j-1}$.* □

```
Algorithm to Compute SiblingSum

01 for each non-attribute leaf node ni arranged in document order {

02     if( ni.branchOrder > 0 ) {

03         sOrd = siblingOrder.Order( ni.branchOrder + 1, elementName[ni.branchOrder + 1] );
           //siblingOrder.Order(level, elementname) keep tracks of the same-sibling order
           //for a node, given the level and the element name for that level

04         ni.SiblingSum = sSum[ ni.branchOrder ] + (sOrd-1) * ModRValue( ni.branchOrder );

05         for ( level = ni.branchOrder + 1; level <= maxDepthOfXMLDoc; level++)
06             sSum[level] = ni.SiblingSum;
07         for ( level = ni.branchOrder + 2; level <= ni.depth; level++)
08             siblingOrder.Order(level, elementName[level]);

09     }

10     else { //initialization for the first leaf node
11         ni.SiblingSum = 0;
12         for ( level = 1; level <= maxDepthOfXMLDoc; level++ )
13             sSum[level] = 0;
14         for ( level = 1; level <= ni.depth; level++ )
15             siblingOrder.Order(level, elementName[level]);
16     }

17 }
```

Figure 4: Algorithm to compute SiblingSum

For example, consider the rightmost `chapter` node in Figure 1(c) which has a Dewey path "1.2.2". Using the ModifiedRValue values derived previously, the DeweyOrderSum of this node can then be calculated as follows: $n.\text{DeweyOrderSum} = (Ord(n, 2) - 1) \times R'_1 + (Ord(n, 3) - 1) \times R'_2 = 1 \times 19 + 1 \times 3 = 22$.

Figure 3 shows the algorithm to derive DeweyOrderSum during document shredding. dSum is an array to store the DeweyOrderSum for each level with respect to the current node n_i. Since n_i.BranchOrder is the level of the nearest common ancestor between the current node n_i and the previous node, it implies that the local order of current node n_i at level BranchOrder + 1 is increased by 1 and the local order of n_i at level $\leq$ BranchOrder remains unchanged. Therefore, n_i.DeweyOrderSum equals to dSum[BranchOrder $+$ 1] + ModifiedRValue(BranchOrder) (line 03). dSum is also updated accordingly (lines 04-05).

Note that DeweyOrderSum is not sufficient to compute position-based predicates with `QName` name tests, *e.g.*, `chapter`[2]. Hence, the SiblingSum attribute is introduced to the PathValue table.

Definition 3 [SiblingSum] *Consider an* XML *document T and a leaf node n at level ℓ in T.* 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 $\Psi(j)$ = [*Sibling(n, j)*-1]$\times R_{j-1}$.* □

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 first `book` element in Figure 1(c). The local orders of `title` and the first and second `chapter` nodes are 1, 2 and 3, respectively. On the other hand, the same-tag-sibling order of these nodes are 1, 1 and 2, respectively.

The algorithm to compute SiblingSum is shown in Figure 3. SiblingOrder.Order(level, elementName) is used to calculate Sibling(n_i, k) for the current node n_i where k = level and τ = elementName. n_i.SiblingSum equals to sSum[BranchOrder] + [Sibling(n_i , n_i.BranchOrder+1) - 1] $\times$ ModifiedRValue(BranchOrder) (line 04).

4.4 Preservation of SUCXENT++'s Features

The above modifications do not adversely affect the document reconstruction process and efficient evaluation of non-ordered XPATH queries, as discussed in [10]. Recall that given a pair of leaf nodes, Property 1 was used in [10] to efficiently determine the nearest common ancestor of the nodes. Since we have modified the definition of RValue and replaced the BranchOrderSum attribute with the DeweyOrderSum attribute, this property is not applicable to the extended SUCXENT++ scheme. It is necessary to ensure that a corresponding property holds in the extended system.

LEMMA 1 $\sum_{j=k}^{\ell} \Phi(j) \leq \frac{R'_{k-2}-1}{2}$ *where $\Phi(j)$ =[*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.* □

Based on the above lemma, it is straightforward to show that $\sum_{j=k}^{\ell} \Phi(j) < R'_{k-2}$.

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

For example, consider the second leaf node in Figure 1(c). DeweyOrderSum of this node is 3. Let D_1 be the DeweyOrderSum of leaf nodes that have nearest common ancestor at level 2. Using the above theorem, D_1 falls within the following range: $(R'_2 - 1)/2 + 1 \leq |D_1 - 3| < (R'_1 - 1)/2 + 1 \Rightarrow 2 \leq |D_1 - 3| < 10$ which returns the first and fourth leaf nodes (DeweyOrderSum = 0 and 6, respectively). Let D_2 be the DeweyOrderSum of leaf nodes that have nearest common ancestor at level 3. D_2 falls within the following range: $(R'_3 - 1)/2 + 1 \leq |D_2 - 3| < (R'_2 - 1)/2 + 1 \Rightarrow 1 \leq |D_2 - 3| < 2$ which returns the third leaf node (DeweyOrderSum = 4). Now let say we want to get the leaf nodes that have nearest common ancestor at level 2 or deeper and let D_3 be the DeweyOrderSum of these nodes. D_3 falls within the following range: $|D_3 - 3| < (R'_1 - 1)/2 + 1 \Rightarrow |D_3 - 3| < 10$ which returns the first four leaf nodes.

We illustrate Theorem 1 further with XQUERY example. Consider the following XQUERY on the XML tree in Figure 1(c).

```
FOR     $b IN document("catalog")/catalog/book[1]
RETURN  $b/chapter/para
```

Let D_a be DeweyOrderSum of the first leaf node satisfying `/catalog/book[1]`, which is the first `title` node. The `RETURN` clause implies the path `/catalog/book/chapter/para`. In this particular case, the nearest common ancestor between `para` nodes and `book` node is at level 2. This implies that the nearest common ancestor between `para` nodes and the first leaf node satisfying `book` node is at level 2 or deeper. Let D_b be DeweyOrderSum of the resulting nodes that satisfy both paths. Since the level of intersection is 2 or deeper, $|D_b - D_a| < (R'_1 - 1)/2 + 1$. From Figure 1(c), $D_a = 0$ and $R'_1 = 19$. Hence, nodes in the query result set must satisfy the inequality: $|D_b - 0| < (19 - 1)/2 + 1$. Note that DeweyOrderSum of the second and third leaf nodes (`para` nodes) are 3 and 4. Since $|3 - 0| < 10$ and $|4 - 0| < 10$, these nodes satisfies the above query. Whereas, the fifth leaf node whose DeweyOrderSum is 19 does not satisfy the above query.

The proofs of the lemma, theorems and propositions are given in Appendix A.

5 Ordered XPath Processing

This section describes how ordered XPATH queries are supported by the modified schema. First, we propose a method of node order comparison in the absence of non-leaf nodes. Next, we show how ordered XPATH queries are supported in detail. Finally, we present a translation algorithm of ordered XPATH queries and SQL.

5.1 Non-leaf Node Order Comparison

Our strategy for comparing the order of non-leaf nodes is based on the following observation. If node n_0 precedes (resp. follows) another node n_1, then descendants of n_0 must also precede (resp. follow) the descendants of n_1. Therefore, instead of comparing the order between non-leaf nodes, we compare the order between *their descendant leaf nodes.* For this reason, we define a *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 nearest common ancestor 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 nearest common ancestor of any two consecutive nodes in C is also a descendant of node n. This implies (1) except n_1, BranchOrder of a node in C is at least the level of node n and (2) the nearest common ancestor 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. We summarize this property in Property 2.

Property 2 Let n be a non-leaf node at level ℓ and $C = \langle n_1, n_2, n_3, \ldots, n_r\rangle$ be the sequence of descendant leaf nodes of n in document order. Then, n_1.BranchOrder $< \ell$ and n_i.BranchOrder $\geq \ell$, where $i \in (1,r]$. □

Definition 4 [DeweyOrderSum of non-leaf nodes] *Let $S = \langle i_1, i_2, i_3, \ldots, i_{r_1}\rangle$ be a sequence of non-leaf sibling nodes of a non-leaf node i_0 in document order. Let $C = \langle n_1, n_2, \ldots, n_{r_2}\rangle$ be the sequence of leaf nodes of S and n_{j_2} is denoted as the first descendant leaf node of i_{j_1}. Then, i_{j_1}.DeweyOrderSum $= n_{j_2}$.DeweyOrderSum.* □

In the above definition, DeweyOrderSum of a leaf node is *conceptually* propagated to its ancestor nodes. Consequently, the following proposition holds.

Proposition 1 *Let $C = \langle n_1, n_2, n_3, \ldots, n_r\rangle$ be a sequence of sibling nodes. Consider n_i where $1 < i \leq r$ and the level of n_i is ℓ, where $\ell > 1$. Let m be n_i or descendant of n_i. Then, n_1.DeweyOrderSum+ [Ord(n_i) - Ord(n_1)] $\times R'_{\ell-1} \leq m$.DeweyOrderSum $< n_1$.DeweyOrderSum+ [(Ord(n_i) - Ord(n_1))+1] $\times R'_{\ell-1}$ where Ord(n_i) and Ord(n_1) are the local order of n_i and n_1, respectively.* □

By using the above proposition, we can compare the order of two non-leaf nodes without evaluating every sibling nodes in the sequence. Also, since n_1 is the first sibling, Ord(n_1) $= 1$. Therefore, based on the above proposition, the following holds: n_1.DeweyOrderSum+ [Ord(n_i)-1]$\times R'_{\ell-1} \leq m$.DeweyOrderSum $<$ n_1.DeweyOrderSum+ Ord(n_i) $\times R'_{\ell-1}$.

Similar propositions for SiblingSum can be established in a straightforward manner.

5.2 Support for Ordered XPath Queries

We now present how various types of ordered XPATH queries are supported by the modified SUCXENT++. Due to space constraints, we only focus on how DeweyOrderSum and ModifiedRValue are used for query processing. Similar technique can be applied to evaluations with SiblingSum.

Position predicates. Position-based predicates, *i.e.,* predicates of the form position()=i, select the node at the i-th position of the sequence of *inner focus context nodes*. We propose to compute the i-th node without evaluating every node in the sequence by applying Proposition 1. For example, suppose n_1 be the first `book` node of the sequence of `book` nodes (the context nodes) in Figure 1(c). Observe that n_1.DeweyOrderSum $= 0$ as its representative leaf node is the first leaf node of the XML tree. We now employ the inequality in Proposition 1 to select a sibling node, *e.g.,* the second `book` node n_2. Here, Ord(n_2) $= 2$, $\ell = 2$, $R'_1 = 19$, and n_1.DeweyOrderSum $= 0$. Then, $0+1\times19 \leq n_2$.DeweyOrderSum $< 0+2\times19 \Rightarrow 19 \leq n_i$.DeweyOrderSum $<$ 38. The nodes in this range are the descendant leaf nodes of n_2. Such simple arithmetic calculations can be efficiently implemented in a relational database.

fn : last() can be computed by first determining all sibling nodes that satisfy the specific path and then finding the node with the largest DeweyOrderSum.

Position predicate on child axes. This class of queries can be translated into a child axis followed by a position predicate, in which one must select the i-th child of the context node. Our strategy is to determine the first child of a context node and then the child's i-th sibling node as described above. First, by using Definition 4, we know that if n_2 is the first child of n_1, then n_1.DeweyOrderSum = n_2.DeweyOrderSum. Second, Proposition 1 provides us a method for selecting the i-th sibling node of a node.

Reconsider Figure 1(c) and the XPATH query `/catalog/*[2]`. The query result is the second child of `catalog` node. Suppose n_0 is the context node `/catalog`. Let n_1 be the first sibling node in the sequence returned by the expression `/catalog/*`. Then, n_0.DeweyOrderSum = n_1.DeweyOrderSum. Since the first sibling in that sequence is n_1 and all siblings of n_1 is in that sequence, we can now utilize Proposition 1 to select the leaf nodes of the second node in the context.

The range operator, *e.g.*, `[position()=2 TO 10]`, can be easily handled in similar fashion.

Following and preceding axes. `following` axis selects all nodes which follow the context node excluding the descendants of the context node. `preceding` axis, on the other hand, selects all nodes which precede

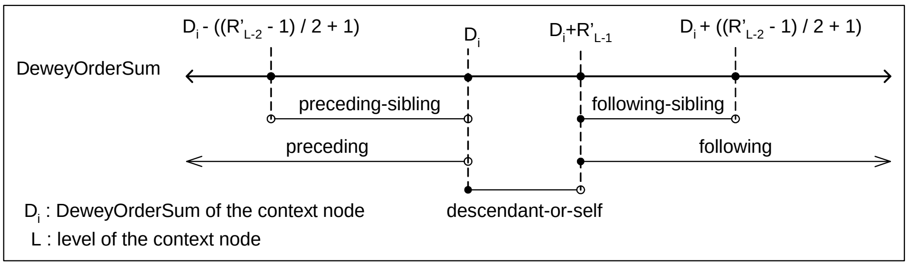


Figure 5: Relationship between DeweyOrderSum and RValue.

---

the context node excluding the ancestors of the context node. Similar to position predicates, we summarize a property of DeweyOrderSum to facilitate efficient processing of these axes.

Proposition 2 *Let n_a and n_b be two nodes in the* XML *tree T and n_b is a context node at level ℓ_b where $\ell_b > 1$. Then, the following statements hold:*

1. n_a.DeweyOrderSum $\geq n_b$.DeweyOrderSum$+R'_{\ell_b-1}$ *if and only if n_a follows n_b and is not a descendant of n_b;*
2. *Similarly,* n_a.DeweyOrderSum $< n_b$.DeweyOrderSum *if and only if n_a precedes n_b and n_a is neither a descendant nor an ancestor of n_b.*

□

We illustrate this proposition in Figure 5. Now let us consider the following examples. Suppose that we evaluate the `following` axis on the first `book` node n_b in Figure 1(c). Here, n_b.DeweyOrderSum $= 0$, $\ell = 2$ and $R'_1 = 19$. Let N be the nodes in the result of the evaluation of `following` axis. Then, by using Proposition 2, $n \in N$ must satisfy this inequality: n.DeweyOrderSum $\geq 0 + 19$. Similarly, suppose we evaluate the `preceding` axis on the last `book` node n'_b in Figure 1(c). n'_b.DeweyOrderSum $= 38$. Denote the sequence of nodes satisfying the `preceding` axis to be N'. Then $n \in N'$ must satisfy the following inequality: n.DeweyOrderSum < 38. Another example can be found in Figure 1(c).

Note that since SUCXENT++ only stores the leaf nodes, returning the internal nodes and the whole subtree as required by `following::*` and `preceding::*` axis require extra processing as the resulting XML document may have a very different structure or schema than the original XML document. This is achieved in SUCXENT++ during the *result construction* phase. Observe that, in our query, if we use `QName` as the name test (for example `following::title`), and the path from the root to `QName` element is unique then no extra processing is required.

If the level of the `QName` is greater than the level of the context node (e.g. `/catalog/book[2]/preceding::title` in Figure 1(c)), then we can use Proposition 2 and PathId to return the resulting nodes. The same also applies if the level of the `QName` equals to the level of the context node (e.g. `/catalog/*[1]/following::book` in Figure 1(c)). However, if the level of the `QName` is less than the level of the context node (e.g. `/catalog/*[1]/chapter/para/following::chapter` in Figure 1(c)), we need to use Theorem 1 to exclude the nodes that have common ancestor at `QName` level or deeper. These three cases are illustrated in Figure 6.

Following-sibling and preceding-sibling axes. `following-sibling` axis selects the children of the context node's parent that occur after the context node in document order whereas `preceding-sibling` axis selects the children of the context node's parent that occur before the context node in document order. Support for `following-sibling` (resp. `preceding-sibling`) axis can be achieved with an additional constraint on the `following` (resp. `preceding`) axis – the selected nodes must be siblings of the context node.

Proposition 3 *Let n_a and n_b be two nodes in the* XML *tree T and n_b is the context node at level ℓ_b where $\ell_b > 2$. Then, the following statements hold:*

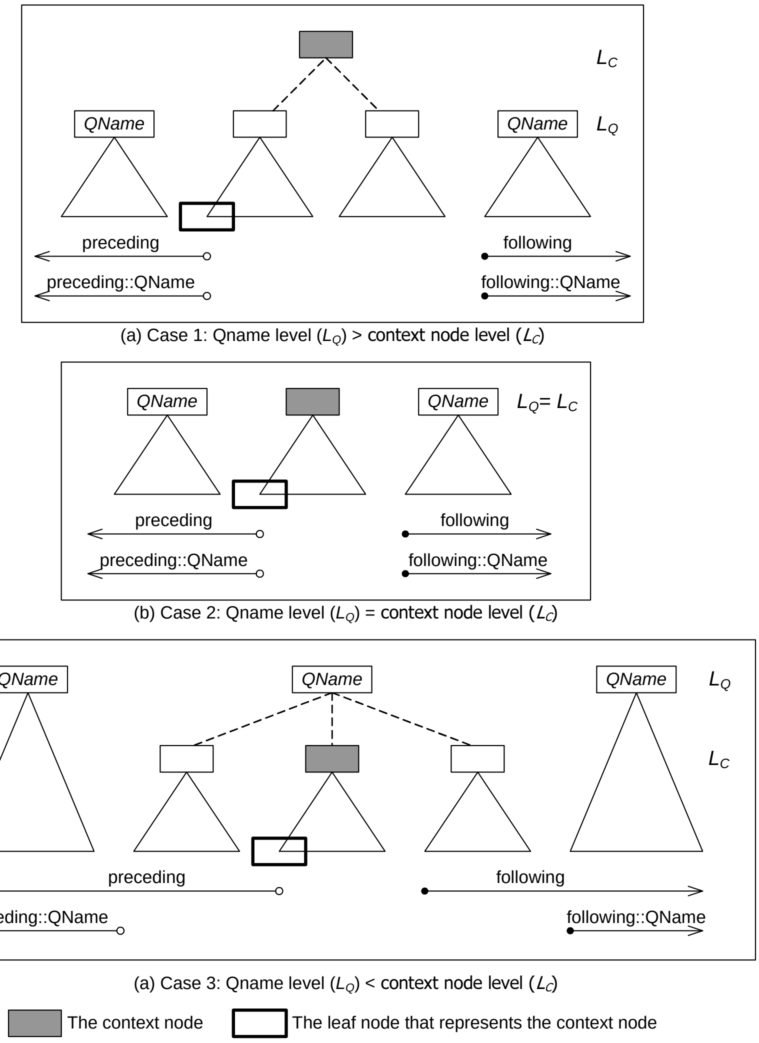


Figure 6: following::QName

---

1. n_b.DeweyOrderSum $+R'_{\ell_b-1} \leq n_a$.DeweyOrderSum $< n_b$.DeweyOrderSum $+(R'_{\ell_b-2}-1)/2+1$ *if and only if n_a is a sibling of n_b and n_a follows n_b.*
2. n_b.DeweyOrderSum$-(R'_{\ell_b-2}-1)/2-1 < n_a$.DeweyOrderSum $< n_b$.DeweyOrderSum *if and only if n_a is a sibling of n_b and n_a precedes n_b.*

□

The above proposition is illustrated in Figure 5. Now let us consider the following examples. Suppose we evaluate the `following-sibling` axis on the first `title` node n_t in Figure 1(c). Here n_t.DeweyOrderSum $= 0$, $\ell = 3$, $R'_1 = 19$, and $R'_2 = 3$. Denote N to be the nodes reachable via the `following-sibling` axis from n_t. Using Proposition 3, $0 + 3 \leq n_k$.DeweyOrderSum $< 0 + (19-1)/2 + 1$ where $n_k \in N$. That is, $3 \leq n_k$.DeweyOrderSum < 10. Hence, the second (DeweyOrderSum = 3) and the third (DeweyOrderSum = 6) `chapter` nodes are in this range. Now, suppose we evaluate the `preceding-sibling` axis at the last `chapter` node n_c in Figure 1(c). Here n_c.DeweyOrderSum $= 22$. Let N be the nodes which satisfy the `preceding-sibling` axis. Therefore, $22 - (19-1)/2 - 1 < n_r$.DeweyOrderSum < 22 where $n_r \in N$. That is, $12 < n_r$.DeweyOrderSum < 22. Hence, the `chapter` node with DeweyOrderSum = 19 satisfies this bound. Another example can be found in Figure 1(c).

```
processPathExpr (XPath)

01 for every step in the XPath {
02   if (step.getAxis() == CHILD and
               step.hasPredicate() == FALSE)
03     currentPath.add(nametest, step.getAxis())
04   else {
05     from_sql.add("PathValue as V_i")
06     if(currentPath.level() > 1) {
07       where_sql.add("V_i.pathid in currentPath.getPathId()")
08       where_sql.add("V_i.branchOrder < currentPath.level()")
09     }
10     processAxis(step, currentPath)
11     processPredicate(step, currentPath)
12   }
13   if (step.isLast() and currentPath.needUpdate()) {
14     from_sql.add("PathValue as V_i")
15     where_sql.add("V_i.pathid in currentPath.getPathId()")
16   }
17 }
18 select_sql.add("V_i.leafValue, V_i.leafOrder, ... ")
19 return select_sql + from_sql + where_sql +
         where_sql.unionWithAttribute()
```

(a)The processPathExpr Algorithm

```
processAxis (step, currentPath)

01 switch (step.getAxis()){
02   child:
03     where_sql.add("V_i.DeweyOrderSum BETWEEN
            V_i-1.DeweyOrderSum - RValue(currentPath.level() - 1) + 1 AND
            V_i-1.DeweyOrderSum + RValue(currentPath.level() - 1) - 1 ")
04   following:
05     where_sql.add("V_i.DeweyOrderSum >=
            V_i-1.DeweyOrderSum + 2 * RValue(currentPath.level()) - 1 ")
06     if (currentPath.QNameLevel(nametest) < currentPath.level()
07       where_sql.add("+ RValue(currentPath.QNameLevel(nametest) - 1) - 1")
08   preceding:
09     where_sql.add("V_i.DeweyOrderSum < V_i-1.DeweyOrderSum ")
10     if (currentPath.QNameLevel(nametest) < currentPath.level()
11       where_sql.add("- RValue(currentPath.QNameLevel(nametest) - 1) + 1")
12   following-sibling:
13     where_sql.add("V_i.DeweyOrderSum BETWEEN
            V_i-1.DeweyOrderSum + 2 * RValue(currentPath.level()) - 1 AND
            V_i-1.DeweyOrderSum + RValue(currentPath.level() - 1) - 1 ")
14   preceding-sibling:
15     where_sql.add("V_i.DeweyOrderSum BETWEEN
            V_i-1.DeweyOrderSum - RValue(currentPath.level() - 1) + 1 AND
            V_i-1.DeweyOrderSum - 1 ")
16 }
17 currentPath.add(nametest, step.getAxis())
```

(b)The processAxis Algorithm

Figure 7: Procedure `processPathExpr` and Procedure `processAxis`.

```
processPredicate (step, currentPath)

01 switch (step.getAxis()) {
02   CHILD:
03      n_from = step.getPredicateFrom() - 1
04      n_to   = step.getPredicateTo()
05   FOLLOWING-SIBLING:
06      n_from = step.getPredicateFrom()
07      n_to   = step.getPredicateTo() + 1
08   PRECEDING-SIBLING:
09      n_from = - step.getPredicateFrom()
10      n_to   = - step.getPredicateTo() + 1
11 }
12 switch (step.getPredicateType()){
13   position based predicate without name test:
14      where_sql.add("V_i.DeweyOrderSum BETWEEN
             V_i-1.DeweyOrderSum + n_from * (2 * RValue(currentPath.level()) - 1) AND
             V_i-1.DeweyOrderSum + n_to * (2 * RValue(currentPath.level()) - 1) - 1 ")
15   position based predicate with name test:
16      where_sql.add("V_i.SiblingSum BETWEEN
             V_i-1.SiblingSum + n_from * (2 * RValue(currentPath.level()) - 1) AND
             V_i-1.SiblingSum + n_to * (2 * RValue(currentPath.level()) - 1) - 1 ")
17 }
```

Figure 8: Procedure `processPredicate`.

5.3 Ordered XPath Query Translation Algorithm

Based on the properties defined in the previous subsection, we present an algorithm, shown in Figures 7 and 8, for generating SQL from ordered XPATH queries. Our algorithm assumes an XPATH expression is represented as a sequence of steps where a step may be associated with predicates. 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 some simple string manipulations and simple SUCXENT++ joins for constructing valid SQL statements. In addition to preprocessing PathId as mentioned in [10], for a single XML document, we also preprocess RValue to reduce the number of joins. The translation consists of three main procedures.

`processPathExpr` **(Figure 7(a)):** It analyzes the steps of an input XPATH expression (Line 01) and outputs a SQL statement. If the step consists of a child axis only (Lines 02-03), then we simply maintain a global variable $currentPath$ which records the simple downward path from the root to the context nodes.[1] Otherwise, when the step involves ordered predicates/other axes, we add predicates which select a superset of the next context nodes (Lines 05-09) and then call `processAxis` and `processPredicate` (Lines 10-11) with $currentPath$ to obtain the next context nodes. We add predicates in Lines 08 to determine the representative nodes of the context nodes. Finally, we collect the final results (Line 19).

`processAxis` **(Figure 7(b)):** This procedure translates a step, together with $currentPath$, based on the step type (Line 01). Lines 02-03, 04-11 and 12-15 encode Theorem 1, Proposition 2 and Proposition 3, respectively.

[1]The details for maintaining $currentPath$ is simple but lengthy. For simplicity, we omitted such discussions.

```
01  WITH V (leafValue, pathID, branchOrder, DeweyOrderSum,
             DocId, LeafOrder   ) AS (
02  SELECT V2.leafValue, V2.pathID, V2.branchOrder,
            V2.DeweyOrderSum, V2.DocId,  V2.LeafOrder
03     FROM PathValue V1, PathValue V2
04     WHERE V1.docId = 1
05     AND V1.SiblingSum BETWEEN
               0 + 0 * (2 * 10 - 1) AND
               0 + 1 * (2 * 10 - 1) - 1
06     AND V1.pathid in (5,4,3,2)
07     AND V1.branchOrder < 2
08     AND V2.docId = V1.docId
09     AND V2.DeweyOrderSum BETWEEN
               V1.DeweyOrderSum - 10 + 1 AND
               V1.DeweyOrderSum + 10 - 1
10     AND V2.pathid in (4,3,2)
11     AND V2.DeweyOrderSum BETWEEN
               V1.DeweyOrderSum + 1 * (2 * 2 - 1) AND
               V1.DeweyOrderSum + 3 * (2 * 2 - 1) - 1
12  )
13  SELECT V.* , 1 AS Attr
14     FROM V
15  UNION ALL
16  SELECT A.leafValue, A.pathID, V.branchOrder, V.DeweyOrderSum,
            A.DocId, A.LeafOrder ,  0 AS Attr
17     FROM Attribute A, V
18     WHERE A.DocId = V.DocId AND A.LeafOrder = V.LeafOrder
19     AND A.PathId in (0)
20  ORDER BY DocId, DeweyOrderSum, Attr
```

Figure 9: SQL example: `/catalog/book[1]/*[position()=2 to 3]`

```
01  WITH V (leafValue, pathID, branchOrder, DeweyOrderSum,
             DocId, LeafOrder   ) AS (
02  SELECT DISTINCT V2.leafValue, V2.pathID, V2.branchOrder,
                      V2.DeweyOrderSum, V2.DocId, V2.LeafOrder
03     FROM PathValue V1, PathValue V2
04     WHERE V1.docId = 1
05     AND V1.pathid in (4,3)
06     AND V1.branchOrder < 3
07     AND V2.docId = V1.docId
08     AND V2.DeweyOrderSum >=  V1.DeweyOrderSum + 2 * 2 – 1
09             + 10 - 1
10     AND V2.pathid in (5,4,3,2)
11  )
12  SELECT V.* , 1 AS Attr
13     FROM V
14  UNION ALL
15  SELECT A.leafValue, A.pathID, V.branchOrder, V.DeweyOrderSum,
            A.DocId, A.LeafOrder ,  0 AS Attr
16     FROM Attribute A, V
17     WHERE A.DocId = V.DocId AND A.LeafOrder = V.LeafOrder
18     AND A.PathId in (1)
19  ORDER BY DocId, DeweyOrderSum, Attr
```

Figure 10: SQL example: `/catalog/book/chapter/following::book`

`processPredicate` **(Figure 8(a)):** This procedure mainly translates position predicates. Lines 01-11 determine the range of position specified by the predicate. Given these, Lines 12-17 implement Proposition 1.

We now illustrate the details of the translation algorithms with five examples related to translation of position-based predicates.

Example 1 [Position-based predicates] *Consider the path expression `/catalog/book[1]/*[position()=2 to 3]`. The translated SQL is shown in Figure 9. `/catalog/book[1]` is translated to Lines 05-07. Theorem 1 is used to get the children of `/catalog/book[1]` (lines 08-10), and `/*[2]` is translated to Lines 11. Lines 13-19 are used to union the resulting element nodes with the attribute nodes and the last line is to order the result by document order.*

Example 2 [SQL for following axis] *Consider the path expression `/catalog/book/chapter/following::book`. The translated SQL is shown in Figure 10. `/catalog/book/chapter` is translated to Lines 05-06. Proposition 2 is used to get the `following` nodes (line 08) and since the level of `book` is higher then the level of `/catalog/book/chapter`, then Theorem 1 is used to exclude the nodes that have common ancestor at level 2 or deeper (line 09).* PathId *is used to return only the `book` nodes (line 10).*

Example 3 [SQL for preceding axis] *Consider the path expression `/catalog/book/chapter/preceding::book`. The translated SQL is similar to Figure 10 except that the predicate in line 08 is*

```
01  WITH V (leafValue, pathID, branchOrder, DeweyOrderSum,
            DocId, LeafOrder  ) AS (
02  SELECT DISTINCT V2.leafValue, V2.pathID, V2.branchOrder,
                    V2.DeweyOrderSum, V2.DocId,  V2.LeafOrder
03     FROM PathValue V1, PathValue V2
04     WHERE V1.docId = 1
05     AND V1.SiblingSum BETWEEN
               0 + 1 * (2 * 10 - 1) AND
               0 + 2 * (2 * 10 - 1) - 1
06     AND V1.pathid in (5,4,3,2)
07     AND V1.branchOrder < 2
08     AND V2.docId = V1.docId
09     AND V2.DeweyOrderSum >= V1.DeweyOrderSum + 2 * 10 - 1
10     AND V2.pathid in (5,4,3,2)
11     AND V2.DeweyOrderSum BETWEEN
               V1.DeweyOrderSum + 1 * (2 * 10 - 1) AND
               V1.DeweyOrderSum + 2 * (2 * 10 - 1) - 1
12  )
13  SELECT V.* , 1 AS Attr
14     FROM V
15  UNION ALL
16  SELECT A.leafValue, A.pathID, V.branchOrder, V.DeweyOrderSum,
         A.DocId, A.LeafOrder ,  0 AS Attr
17     FROM Attribute A, V
18     WHERE A.DocId = V.DocId AND A.LeafOrder = V.LeafOrder
19     AND A.PathId in (1)
20  ORDER BY DocId, DeweyOrderSum, Attr
```

Figure 11: SQL example: `/catalog/book[2]/following-sibling::*[1]`

replaced with V2.DeweyOrderSum $<$ V1.DeweyOrderSum *due to Proposition 2 and line 09 is replaced with* `- 10 + 1` *due to Theorem 1.*

Example 4 [SQL for following-sibling axis] *Consider the path expression* `/catalog/book[2]/following-sibling::*[1]`*. The translated* SQL *is shown in Figure 11.* `/catalog/book[2]` *is translated to Lines 05-07.* `/following-sibling::*` *is translated to Lines 08-10. Since the level of* `/catalog/book` *is 2, the translated SQL for* `following-sibling` *is similar to* `following` *axis (line 9).* `*[1]` *is translated to line 11.*

Example 5 [SQL for preceding-sibling axis] *Consider the path expression* `/catalog/book[2]/preceding-sibling::*[1]`*. The translated* SQL *is similar to Figure 11 except that the predicate in line 09 is replaced with* V2.DeweyOrderSum $<$ V1.DeweyOrderSum.

6 Join Order Enforcement

Due to the tree-unaware nature of the underlying relational storage scheme as well as the lack of appropriate XML statistics, relational optimizers may generate inefficient query plans. In order to address this problem, some approaches have resorted to manual tuning of query plans [12] while others invade the database kernel to make it tree-aware [3, 4]. The former approach has not been scalable as it requires significant human intervention whereas the later approach may require non-trivial modifications of the internals of a RDBMS. In this section, we propose a simple yet effective technique to generate better query plans automatically *without invading the database kernel*.

As discussed in Section 5.3, in order to evaluate an (ordered) XPATH query in SUCXENT++, each XPATH axis is translated into a join between the PathValue table and intermediate results (*i.e.,* the context nodes). For example, in Figures 9-11, PathValue V1 returns the representative nodes of the context nodes to calculate PathValue V2. Due to the lack of tree awareness, the relational optimizer is not capable of transforming the order of joins intelligently. Consequently, it may generate poor join order that typically requires caching large intermediate results in the database bufferpool. This is particularly important to NL joins, where large and deep loops are prohibitive. For example, the first few joins of a "right-to-left" join order may easily yield a large number of context nodes. To respond to this, we propose to enforce a "left-to-right" join order on the translated SQL query. Also, this evaluation order "naturally corresponds" to the order of XPATH steps specified in the XPATH expression. By employing this technique, the relational optimizer does not explore the large number of permutations of join order. We apply join order if the translated SQL query involves more than one PathValue

ID	Total Number			Size (MB)	Max Depth
	Node	Attribute	Total		
DC10	225,234	15,000	240,234	10.3	8
DC100	2,242,200	150,000	2,392,200	103.3	8
DC1000	22,442,612	1,500,000	23,942,612	1033.3	8
DBLP	8,222,945	1,665,930	9,888,875	335	6

(a) Features of Dataset

ID	Query	Res. Card.
D1	/dblp/*[100000]/author	2
D2	/dblp/article/author[2]	190,838
D3	/dblp/*[600000]/pages/preceding-sibling::*	6
D4	/dblp/*[600000]/pages/following-sibling::*	5

(c) Benchmark queries for DBLP

ID	Query	Res. Card. (10MB)	Res. Card. (100MB)	Res. Card. (1000MB)
Q1	/catalog/item[1000]	66	119	74
Q2	/catalog/*[1000]	66	119	74
Q3	/catalog/item[position()=1000 to 10000]/*[position()=2 to 7]	104,272	626,812	627,200
Q4	/catalog/item[position()=1000 to 10000]/authors/author	65,161	392,930	393,350

ID	Query	Res. Card. (10MB)	Res. Card. (100MB)	Res. Card. (1000MB)
Q5	/catalog/*[1500]/publisher/following-sibling::*	30	34	34
Q6	/catalog/*[1500]/publisher/following-sibling::*[5]	7	7	7
Q7	/catalog/*[1500]/publisher/preceding-sibling::*	21	37	54
Q8	/catalog/*[1500]/publisher/preceding-sibling::*[2]	19	35	52
Q9	/catalog/*[X]/following::title	250	2,500	25,000
Q10	/catalog/*[Y]/preceding::title	249	2,499	24,499

X = 2250, 22500, 225000 for DC10, DC100, DC1000 respectively; Y = 250, 2500, 25000 for DC10, DC100, DC1000 respectively

(b) Benchmark queries for DC10, DC100, and DC1000

Figure 12: Dataset and Benchmark Queries.

relation. In addition, if the PathValue table appears in the SQL query only once, we let the relational optimizer to decide the plan for the join between the PathValue table and the Attribute table.

The above enforcement can easily be implemented by *query hints* in commercial databases. Regarding our implementation, we use `OPTION(FORCE ORDER)` to implement the above technique in SUCXENT++. The strength of this approach lies in its simplicity in implementing on any commercial RDBMS that supports query hints.

7 Performance Study

In this section, we present the results of our performance evaluation on our proposed approach, a tree-unaware schema-oblivious approach (GLOBAL-ORDER [12]), a tree-unaware schema-conscious approach (SHARED-INLINING [11]), and a tree-aware approach (MonetDB [3]). Prototypes for modified SUCXENT++ (denoted as SX), SUCXENT++ with join order enforcement (denoted as SX-JO), GLOBAL-ORDER (denoted as GO) and SHARED-INLINING (denoted as SI) were implemented with JDK 1.5. We used the Windows version of MONETDB/XQuery 0.12.0 (denoted as MXQ) downloaded from http://monetdb.cwi.nl/XQuery/Download/index.html. The experiments were conducted on an Intel Xeon 2GHz machine running on Windows XP with 1GB of RAM. The RDBMS used was Microsoft SQL Server 2005 Developer Edition. Note that we did not study the performance of XML support of SQL Server 2005 as it can only evaluate the first two ordered queries in Figure 12(b).

7.1 Experimental Setup

Data and query sets. In our experiments, XBENCH [13] dataset was used for synthetic data. Data-centric (DC) documents were considered with data sizes ranging from 10MB to 1GB. In addition, we used a real dataset, namely DBLP XML [16]. Figure 12 (a) shows the characteristics of the datasets used. Two sets of queries were designed to cover different types of ordered XPATH queries. In additional, the cardinality of the results was varied. Figures 12 (b) and 12 (c) show the benchmark queries on XBENCH and DBLP, respectively. XPATH queries with descendant axes were not included as they had been studied in [10].

Test methodology. The XPATH queries were executed in the *reconstruct* mode where not only the non-leaf nodes, but also all their descendants, were selected. Appropriate indexes were constructed for all approaches (except for MONETDB) through a careful analysis on the benchmark queries. Prior to our experiments, we ensured that statistics on relations were 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.

ID	DC10					DC100					DC1000			
	MXQ	SI	GO	SX	SX-JO	MXQ	SI	GO	SX	SX-JO	SI	GO	SX	SX-JO
Q1	44.17	1,042.33	843.17	58.33	58.33	80.50	5,967.00	13,177.17	47.67	47.67	39,152.67	85,223.50	61.67	61.67
Q2	36.17	1,041.17	862.33	27.67	27.67	114.67	5,967.00	7,653.67	60.33	60.33	39,152.67	86,271.17	44.50	44.50
Q3	492.33	4,935.33	7,163.00	75,236.00	5,885.00	3,023.67	31,229.50	43,517.67	DNF	47,664.17	64,976.50	134,293.83	DNF	368,666.00
Q4	226.50	3,138.83	4,517.33	2,726.00	2,726.00	1,364.33	17,574.33	30,352.50	14,266.33	14,266.33	44,738.67	286,369.00	56,665.17	56,665.17
Q5	41.83	385.33	1,359.67	13.00	28.17	85.83	1,740.67	7,176.50	5,133.67	209.00	7,563.33	1,026.17	49,795.33	1,036.67
Q6	41.50	41.17	1,233.67	63.67	72.83	88.67	437.67	7,121.67	339.00	248.50	1,951.00	889.83	54,927.67	925.67
Q7	36.33	708.67	1,594.00	63.67	78.50	81.17	4,223.33	7,161.33	5,236.20	208.20	30,292.83	908.17	50,419.83	1,000.50
Q8	39.00	688.17	1,556.33	125.67	35.67	85.83	3,522.17	7,301.83	365.83	222.83	6,702.00	868.67	54,610.83	1,144.17
Q9	36.00	91.00	3,244.50	132.67	137.83	174.67	804.83	8,809.00	650.83	668.67	6,264.50	DNF	42,872.00	7,992.17
Q10	39.00	72.50	5,007.17	153.17	137.50	177.17	511.00	8,129.83	680.17	702.33	1,720.33	DNF	42,925.17	8,456.50

(a) For DC10, DC100, and DC1000 (in msec)

ID	MXQ	SI	GO	SX	SX-JO	ID	MXQ	SI	GO	SX	SX-JO
D1	1,927.80	6,264.17	24,975.17	55.00	55.00	D3	2,143.60	82,539.00	32,829.17	46,827.50	2,008.83
D2	2,803.00	12,596.67	39,912.00	DNF	32,605.83	D4	2,859.20	81,575.00	32,795.00	46,820.50	1,886.83

(b) For DBLP (in msec)

Figure 13: Query Performance (in msec).

7.2 Query Evaluation Times

Figures 13(a) (resp. 13(b)) presents the query evaluation times for the approaches on DC (resp. DBLP) dataset. Queries that Did Not Finish within 60 minutes were denoted as DNF.

Enforcement of Join Order. The SX and SX-JO columns in Figure 13 describes the effect of enforcing join order in SUCXENT++. Note that we did not enforce the join order for queries `Q1`, `Q2`, `Q4`, and `D1` when the PathValue table appears in the translated SQL queries only once.

We made three main observations from our results as follows. First, in almost all cases the query performance improved significantly when join order is enabled. For instance, for DBLP the performance of queries `D3` and `D4` were improved by factors of 23 and 25, respectively. In fact, 18 out of 24 queries in Figure 13 benefited from join order enforcement. Second, the benefit of this technique increases as the dataset size increases. For instance, for the 1GB dataset the performances of `Q5` to `Q8` improved by 47 to 59 times. Furthermore, queries that failed to return results previously in 60 minutes (`Q3`, `D2`) were now able to return results across all benchmark datasets. Third, the penalty of join order for most of the benchmark queries, if any, was low on all benchmark datasets. In fact, the largest penalty on the query performance due to join order enforcement was 22*ms*. In Section 7.3, we shall elaborate on the effectiveness of join order enforcement by analyzing the query plans.

Comparison with GLOBAL-ORDER and SHARED-INLINING. Overall SX-JO outperformed both SI and GO in at least 65% of the benchmark queries with the highest observed gain factors being 880 and 1939, respectively. GO showed non-monotonic behavior for `Q5−Q8` and as a result the performance of SX-JO was comparable to GO for these queries on DC1000. However, SX-JO significantly outperformed SI for `Q5−Q8` (up to 30 times). Note that for DC1000, GO failed to return results for queries `Q9` and `Q10`. Finally, for the DBLP dataset, SX-JO significantly outperformed GO and SI for `D1`, `D3`, and `D4`, with the highest observed gain factor 454 and 114, respectively.

Comparison with MONETDB. Our study in the context of MONETDB revealed some interesting results. First, MXQ was 11-164 and 3-74 times faster than GO and SI, respectively, for the majority of the benchmark queries. However, this performance gap was significantly reduced when it was compared against SX-JO. Our results showed that MXQ was 1.3-16 times faster than SX-JO. Surprisingly our approach was faster than MONETDB for 33% of benchmark queries! Specifically, SX-JO was faster than MXQ for `Q2`, `Q5`, and `Q8` on DC10 and `Q1` and `Q2` on DC100. Also, for the real dataset (DBLP) SX-JO was faster than MXQ for `D1`, `D3`, and `D4` with the highest observed factor being 35. Unfortunately, we could not report the results of MXQ for DC1000 because it failed to shred the document. The reason of this problem is that MXQ (Win32 builds) is currently vulnerable to the virtual memory fragmentation in Windows environment. MXQ also does not evaluate predicates applied after reverse axis in reverse document order, but in document order. Therefore, in `Q8`, it evaluated the second `preceding-sibling` element in document order, not in reverse document order (not in accordance to W3C XPath recommendation [17]).

7.3 Sucxent++ Query Plan Analysis

In this subsection, we present an analysis of the query plans for the queries that are greatly benefited by join order enforcement. Before we proceed any further, we wish to clarify that our goal in this paper is to present a novel scheme for efficient processing of ordered XPATH queries in relational databases and highlight the interesting behavior of a commercial optimizer in this context. We stress that, not being privy of the internals of the optimizer, some of the remarks made in the subsequent discussion related to the query optimizer are speculative in nature and should therefore be treated as such. Our intention is primarily to inform the community to the phenomena that we have encountered during our investigation, with the hope that they may prove useful in building the next generation of XML-enabled database systems.

Plan analysis of DC10 Q3: The SQL syntax for Q3 is similar to example in Figure 9. In the SQL statement, two PathValue tables are joined together (lines 03-11) to form a temporary table V (line 01). V is used to return the element nodes (lines 13-14) and attribute nodes (lines 16-19).

The portions of the query plans for Q3 without/with join order are shown in Figure 14 and Figure 15. The query plan trees for both approach consist of primarily two subtrees. One subtree (Figure 14(a) and Figure 15(a)) computes the V table and then returns all the attributes of V. The other subtree (Figure 14(b) and Figure 15(b)) depicts the plan for computing the V table followed by joining it to the Attribute table.

Let us discuss the first subtree. Without join order, SQL Query Optimizer is "not smart enough" to decide how to select both of the PathValue tables leading to larger intermediate result. Take a look at the upper part of Figure 14(a). Notice the size of arrow going out from the ClusteredIndexSeek-PathValue table is large. The size of arrow is proportional to the result size. This is because the seek predicates used are not specific enough; therefore, more rows are returned. And in the lower part of Figure 14(a), rather than using clustered index seek and filter, SQL Query Optimizer uses several steps which lead to longer query processing time. The most expensive operation (as seen by the percentage and the arrow size) is the Index Spool (Eager Spool).

With join order enforcement (Figure 15(a)), SQL Query Optimizer uses better seek predicates for the upper PathValue table resulting in smaller intermediate result size and uses less steps for the lower PathValue table leading to a more efficient processing.

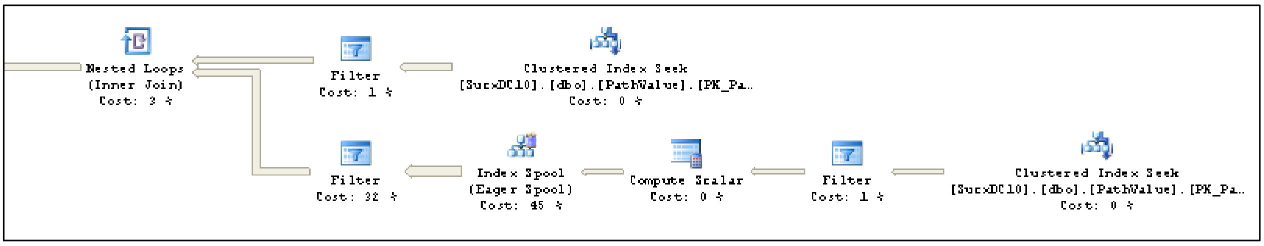
For the second subtree, with join order (Figure 15(b)), SQL Query Optimizer joins the two PathValue tables, then does a hash match with the Attribute table. Whereas without join order (Figure 14(b)), SQL Query Optimizer firstly joins the Attribute table with the PathValue table, then joins the result with two PathValue tables. The total number of joins is greater by one and there is more processing compared to the join order approach.

Plan analysis of DC100 Q5,Q7, DC1000 Q5–Q8: The SQL syntax for Q5–Q8 is similar to example in Figure 11 except that for Q5 and Q7, line 11 is not applicable. In the SQL statement, two PathValue tables are joined together (lines 03-11) to form a temporary table V (line 01). V is used to return the element nodes (lines 13-14) and attribute nodes (lines 16-19).

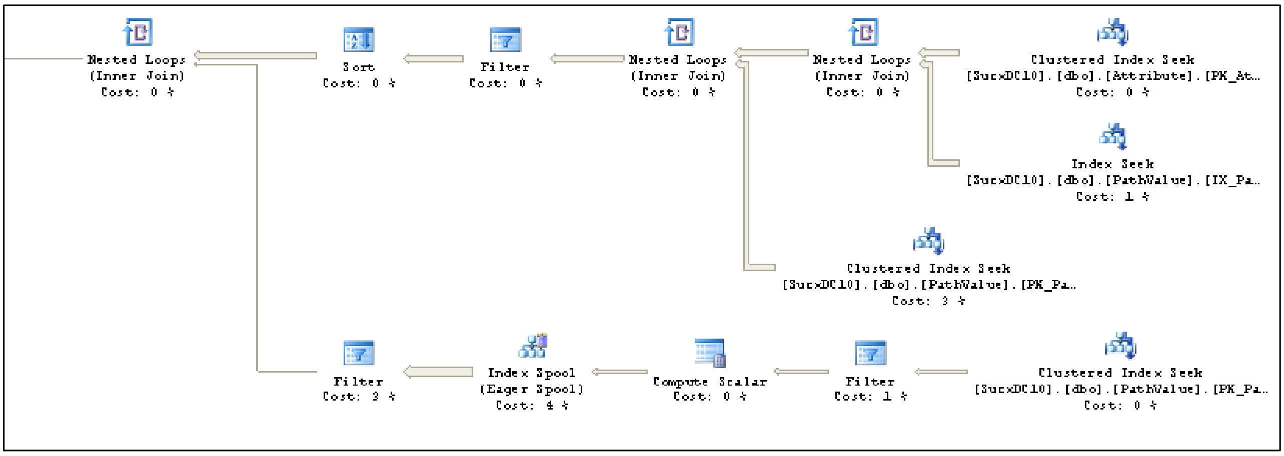
The query plan for DC100 Q7, DC1000 Q5–Q8 are similar to query plan for DC100 Q5 with some minor differences. Therefore, we only discuss the query plan for DC100 Q5. Similar to Q3, the query plan trees for both without join order (Figure 16) and with join order (Figure 17) approaches consist of two subtrees.

Let us discuss the first subtree. Similar to what happens in Q3, without join order, the intermediate result size is much greater than with join order. As can be seen in the upper part of Figure 16(a), the result size of the ClusteredIndexSeek-PathValue is large; this is due to the seek predicates used by SQL Query Optimizer is not specific enough. And in the lower part of the figure, the large result size of Table Spool (Lazy Spool) causes the cost to be large as well. Whereas in Figure 17(a), the seek predicate used is better and Table Spool is not required to process the result.

For the second subtree, interestingly, without join order (Figure 16(b)), SQL Query Optimizer chooses better seek predicate and better steps for the two PathValue tables. But even though the cost to calculate the second subtree is relatively low, since the cost of calculating the first subtree is high, the total query time is still high. With join order (Figure 17(b)), SQL Query Optimizer joins the two PathValue tables, then joins it with the Attribute table.

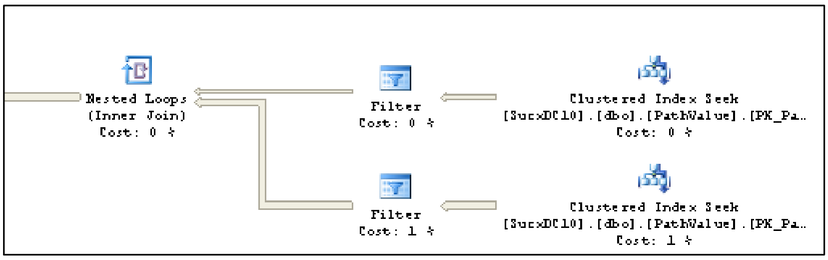


(a) Subtree to compute V table and return V table

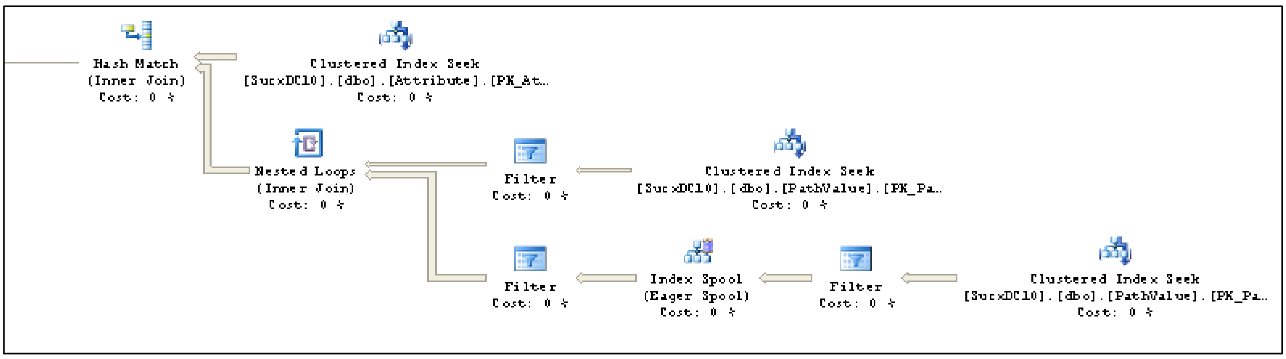


(b) Subtree to compute V table followed by join with Attribute

Figure 14: Portion of SUCXENT++ Query Plan DC10 Q3 (without join order)

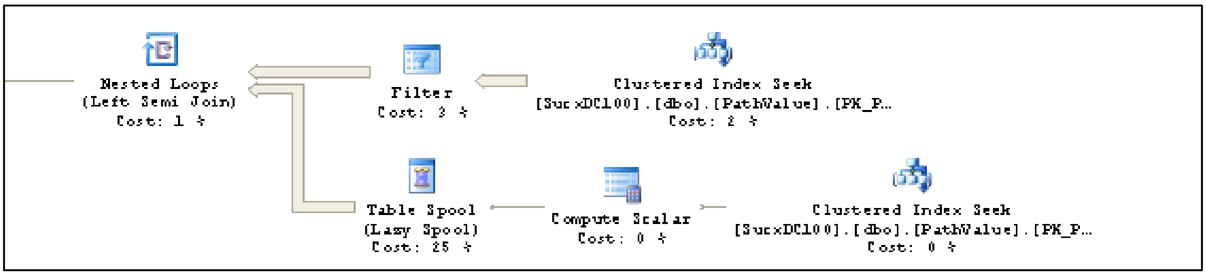


(a) Subtree to compute V table and return V table

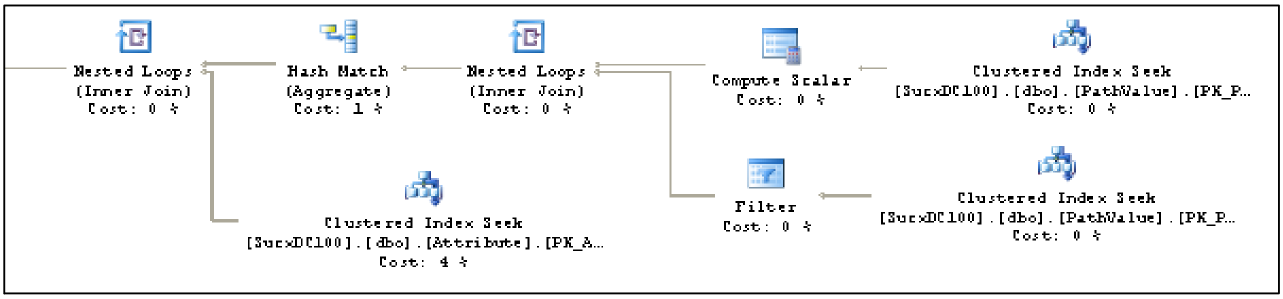


(b) Subtree to compute V table followed by join with Attribute

Figure 15: Portion of SUCXENT++ Query Plan DC10 Q3 (with join order)

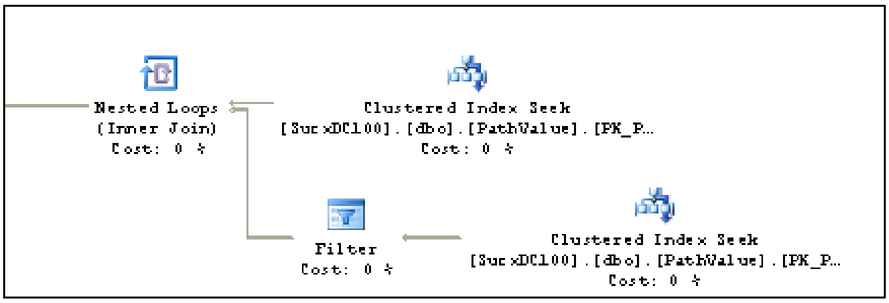


(a) Subtree to compute V table and return V table

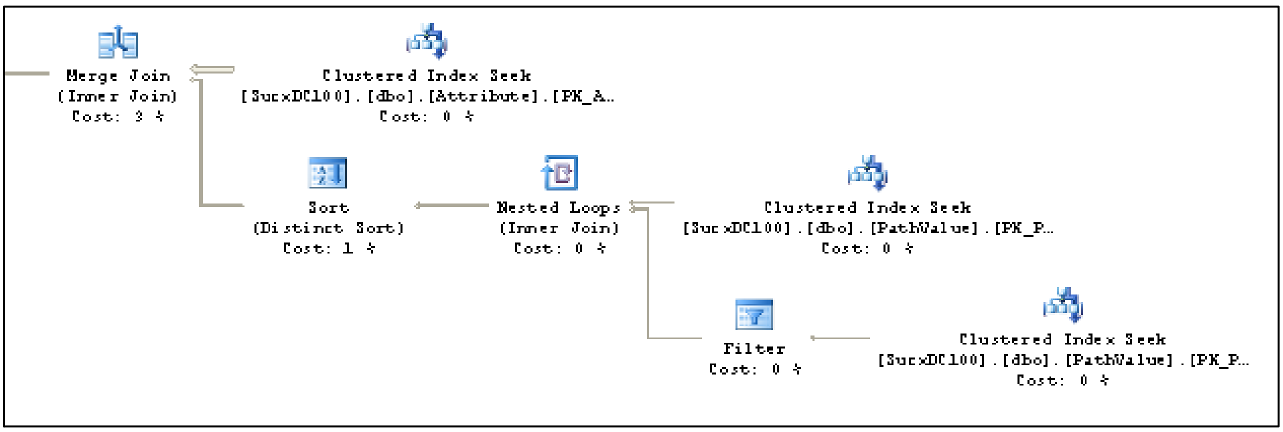


(b) Subtree to compute V table followed by join with Attribute

Figure 16: Portion of SUCXENT++ Query Plan DC100 Q5 (without join order)



(a) Subtree to compute V table and return V table



(b) Subtree to compute V table followed by join with Attribute

Figure 17: Portion of SUCXENT++ Query Plan DC100 Q5 (with join order)

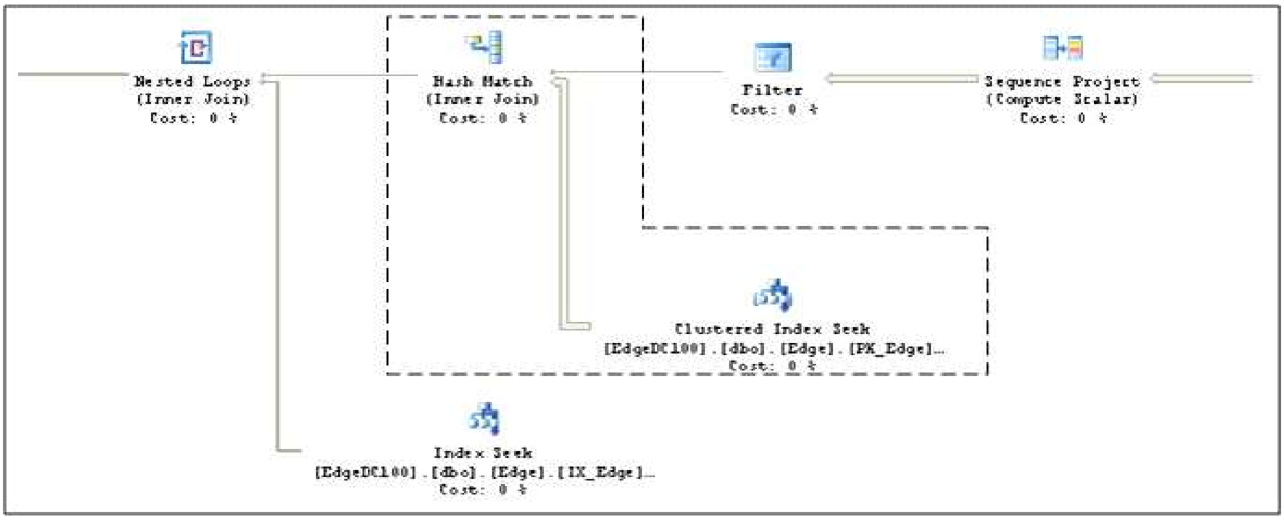


Figure 18: Portion of EDGE Query Plan DC100 Q5

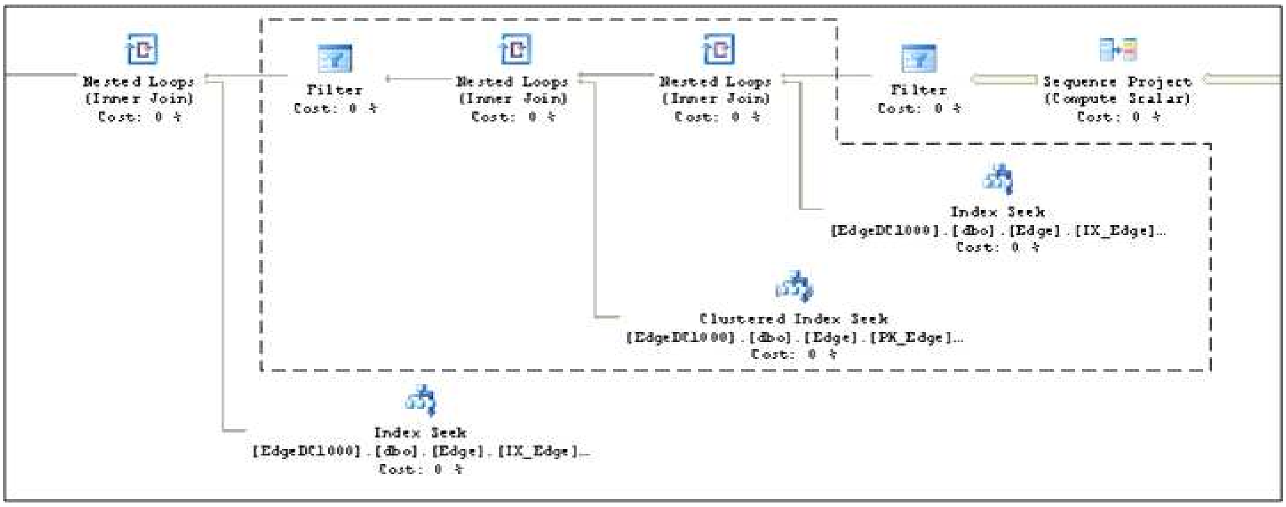


Figure 19: Portion of EDGE Query Plan DC1000 Q5

7.4 Edge Query Plan Analysis

This subsection discusses query plans of Edge which have anti-monotonic behavior.

Plan analysis of DC100 and DC1000 `Q5-Q8`: The reason why DC100 and DC1000 `Q6-Q8` have anti-monotonic behavior is similar to `Q5`. Therefore, we only discuss query plan for `Q5`.

The query plan for DC100 and DC1000 `Q5` are similar except for the portion to get the `publisher` node. To get `publisher` node, after getting the context node (`/catalog/*[1500]`), we need to get the children of the context node which satisfy path id of `publisher`. The dotted box in Figure 18 and Figure 19 highlights the portion to get the `publisher` node for DC100 and DC1000. The part on the right side of the dotted box is where the context nodes are retrieved.

In DC100 (Figure 18), the SQL Query Optimizer choose to get the result by using clustered index seek and do a hash match with the context node. But it appears that the clustered index seek return a lot of result which make the overall process expensive. In DC1000 (Figure 19), the SQL Query Optimizer gets the `publisher` node by using several steps but with smaller result size for each step, which leads to faster processing.

Plan analysis of DC10 and DC100 `Q9-Q10`: The reason why DC100 and DC1000 `Q9-Q10` have anti-monotonic behavior are similar. Therefore, we only discuss query plan for `Q10`.

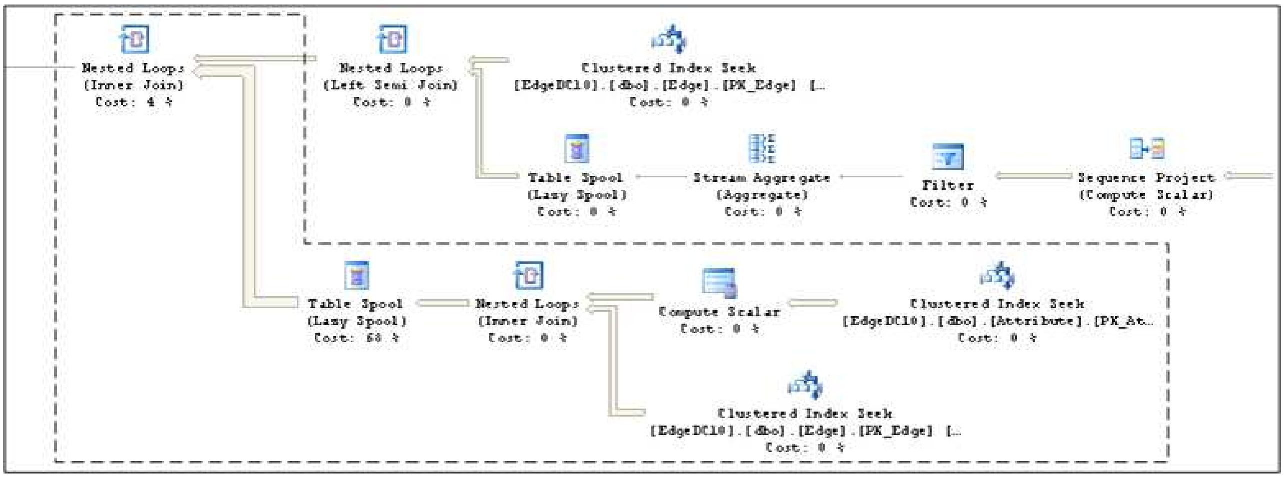


Figure 20: Portion of EDGE Query Plan DC10 Q10

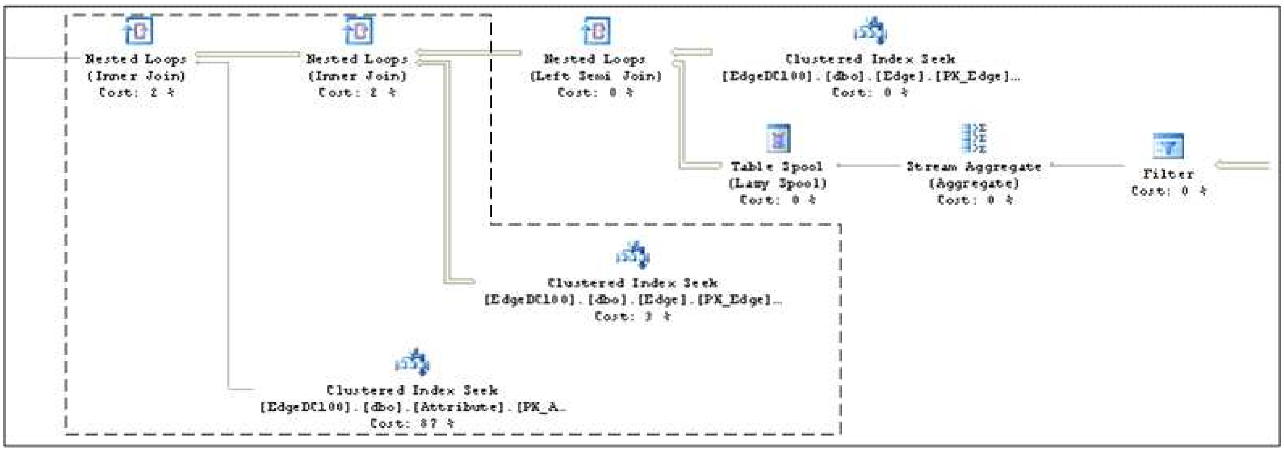


Figure 21: Portion of EDGE Query Plan DC100 Q10

Different with [12] which do not consider attribute, we introduce an additional table Attribute to store all of the attributes. To get attribute nodes, firstly we need to get all of the element nodes, both non-leaf nodes (i.e. context nodes) and all of their descendants (note that we run the experiment in reconstruct mode). After that, we join the Attribute table with the element nodes based on the id to get the attribute nodes. The final result is a `UNION ALL` between element nodes with the attribute nodes.

The main difference between DC10 and DC100 is in how the attribute nodes are retrieved. The dotted box in Figure 20 and Figure 21 highlights the portion where SQL Query Optimizer joins the Edge table with the Attribute table for DC10 and DC100. The part on the right side of the dotted box is where the non-leaf nodes are retrieved.

In DC100, SQL Query Optimizer firstly joins all except the last Edge table to get the non-leaf nodes. The last Edge table is used to retrieve all of the element nodes (non-leaf nodes and their descendant), then the result is joined with Attribute table to get the attribute nodes (dotted box in Figure 21). But in DC10, SQL Query Optimizer joins the Edge table with Attribute table (dotted box in Figure 20), then joins the result with the non-leaf element nodes which is more expensive.

8 Conclusions and Future Work

In this paper, we presented a scalable storage scheme for ordered XPATH evaluation in relational environment. Our scheme extends SUCXENT++ [10] for the support of the processing of ordered axes and predicates while maintaining its original properties. The mapped SQL queries were forced to execute a "left-to-right" join order. We showed that this technique could improve query performance notably. In addition, our results showed that our proposed approach outperforms other representative *tree-unaware* approaches for the majority of the benchmark queries. Although *tree-aware* approaches were often the best in terms of query performance [3], the "join-order conscious" SUCXENT++ reduced the performance gap between tree-aware and tree-unaware approaches significantly and could outperform a state-of-the-art tree-aware approach (MONETDB) for certain benchmark queries. Importantly, unlike tree-aware approaches, our approach did not require any invasion of the database kernels to improve query performance and could easily be built on top of any off-the-shelf commercial RDBMS.

As part of our future work, we are studying the "join order" phenomena encountered during our investigation. We are also exploring other non-invasive mechanisms for improving XPATH query performance on a relational backend.

References

[1] K. BEYER, R. J. COCHRANE, V. JOSIFOVSKI, ET AL. System RX: One Part Relational, One Part XML. *In SIGMOD*, 2005.

[2] P. BOHANNON, J. FREIRE, P. ROY, J. SIMEON. From XML Schema to Relations: A Cost-based Approach to XML Storage. *In ICDE*, 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] D. DEHAAN, D. TOMAN, M. P. CONSENS, M. T. OZSU. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Coding. *In SIGMOD*, 2003.

[5] D. FLORESCU, D. KOSSMAN. Storing and Querying XML Data using an RDBMS. *IEEE Data Engg. Bulletin*. 22(3), 1999.

[6] T. GRUST, J. TEUBNER, M. V. KEULEN. Accelerating XPath Evaluation in Any RDBMS. *In ACM TODS*, 2004.

[7] T. GRUST, J. TEUBNER, M. V. KEULEN. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. *In Proc. of VLDB*, 2003.

[8] Z. H. LIU, M. KRISHNAPRASAD, V. AURORA. Native XQuery Processing in Oracle XMLDB. *In SIGMOD*, 2005.

[9] S. PAL, I. CSERI, O. SEELIGER ET AL. XQuery Implementation in a Relational Database System. *In VLDB*, 2005.

[10] S. PRAKASH, S. S. BHOWMICK, S. K. MADRIA. Efficient Recursive XML Query Processing Using Relational Databases. *In DKE)*, 58(3), 2006.

[11] J. SHANMUGASUNDARAM, K. TUFTE ET AL. Relational Databases for Querying XML Documents: Limitations and Opportunities. *In VLDB*, 1999.

[12] I. TATARINOV, S. VIGLAS, K. BEYER, ET AL. Storing and Querying Ordered XML Using a Relational Database System. *In SIGMOD*, 2002.

[13] B. YAO, M. TAMER ÖZSU, N. KHANDELWAL. XBench: Benchmark and Performance Testing of XML DBMSs. *In ICDE*, Boston, 2004.

[14] M. YOSHIKAWA, T. AMAGASA, T. SHIMURA, AND S. UEMURA. XRel: A Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM TOIT* 1(1):110-141, 2001.

[15] C. ZHANG, J. NAUGHTON, D. DEWITT, Q. LUO AND G. LOHMANN. On Supporting Containment Queries in Relational Database Systems. *In SIGMOD*, 2001.

[16] DBLP XML Record. *http://dblp.uni-trier.de/xml/*.

[17] XML Path Language (XPath) 2.0: W3C Proposed Recommendation 21 November 2006. *http://www.w3.org/TR/xpath20/*

A Proofs

A.1 Proof of Lemma 1

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

Recall convention that the first sibling has $\mathsf{LocalOrder}$ 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,

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

A.2 Proof of Theorem 1

To prove Theorem 1, we separate it into two parts: $|n_1.\mathsf{DeweyOrderSum}\text{ - }n_2.\mathsf{DeweyOrderSum}| < \frac{R'_\ell - 1}{2} + 1$ and $\frac{R'_{\ell+1}-1}{2} + 1 < |n_1.\mathsf{DeweyOrderSum}\text{ - }n_2.\mathsf{DeweyOrderSum}|$

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

Assume the level of the nearest common ancestor of n_1 and n_2 is $\leq \ell$, then $|n_1.\mathsf{DeweyOrderSum}\text{ - }n_2.\mathsf{DeweyOrderSum}| < (R'_\ell$ - 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 $j \geq \ell + 1$. Consider the following cases.

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

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

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:

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

Case n_1.LeafOrder $<$ n_2.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:

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

Therefore,

$$|\Delta| \geq (\frac{R'_\ell - 1}{2} + 1) \quad (contradiction)$$

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**:**

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

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

$$\begin{aligned}|\Delta| &\geq \frac{R'_k - 1}{2} + 1 \\ &> (\frac{R'_\ell - 1}{2} + 1) \quad since\ k < \ell\ (contradiction)\end{aligned}$$

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.* □

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**:**

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

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 conclude that if $\frac{R'_{\ell+1}-1}{2} + 1 \leq |n_1.\mathsf{DeweyOrderSum}$ - $n_2.\mathsf{DeweyOrderSum}| < \frac{R'_{\ell}-1}{2} + 1$ then the level of the nearest common ancestor of n_1 and n_2 is $\ell + 1$.

A.3 Proof of Proposition 1

Let ℓ_{f1} be the level of the leaf node representing n_1 and ℓ_{fi} be the level of the leaf node representing m. Given that n_1 and n_i are siblings and m is either n_i or n_i's descendant, both n_1 and m must have the same ancestors at level $\ell - 1$ or lesser. Therefore, Ord(n_1,j) = Ord(m,j) for $1 \leq j < \ell$ and Ord(m,ℓ) = Ord(n_i,ℓ). Then,

$$\begin{aligned}
\Delta &= m.\mathsf{DeweyOrderSum} - n_1.\mathsf{DeweyOrderSum} \\
&= \sum_{j=2}^{\ell_{fi}} \Phi_i(j) - \sum_{j=2}^{\ell_{f1}} \Phi_1(j) \\
&= \sum_{j=\ell}^{\ell_{fi}} \Phi_i(j) - \sum_{j=\ell}^{\ell_{f1}} \Phi_1(j) \\
&= [\mathsf{Ord}(m,\ell) - \mathsf{Ord}(n_1,\ell)] \times R'_{\ell-1} + \sum_{j=\ell+1}^{\ell_{fi}} \Phi_i(j) - \sum_{j=\ell+1}^{\ell_{f1}} \Phi_1(j) \\
&= [\mathsf{Ord}(n_i,\ell) - \mathsf{Ord}(n_1,\ell)] \times R'_{\ell-1} + \sum_{j=\ell+1}^{\ell_{fi}} \Phi_i(j) - \sum_{j=\ell+1}^{\ell_{f1}} \Phi_1(j)
\end{aligned}$$

Since first descendant leaf node of n_1 is the representative leaf node of n_1, $\sum_{j=\ell+1}^{\ell_{f1}} \Phi_1(j) = 0$. We know $\sum_{j=k}^{\ell} \Phi(j) \geq 0$ since $\Phi(j) \geq 0 \; \forall \; j$. Also from Lemma 1 $\sum_{j=k}^{\ell} \Phi(j) < R'_{k-2}$ for $k > 2$. Then for $H = [\mathsf{Ord}(n_i, \ell) - \mathsf{Ord}(n_1, \ell)] \times R'_{\ell-1}$ the following holds,

$$\begin{aligned} H &\leq \Delta < H + R'_{\ell+1-2} \\ H &\leq \Delta < [\mathsf{Ord}(n_i, \ell) - \mathsf{Ord}(n_1, \ell) + 1] \times R'_{\ell-1} \end{aligned}$$

Manipulating the above inequality by replacing Δ with m.DeweyOrderSum-n_1.DeweyOrderSum and Ord(n_i, ℓ) with Ord(n_i), we get

$$\begin{aligned} & n_1.\mathsf{DeweyOrderSum} + [\mathsf{Ord}(n_i) - \mathsf{Ord}(n_1)] \times R'_{\ell-1} \\ \leq \;& m.\mathsf{DeweyOrderSum} \\ < \;& n_1.\mathsf{DeweyOrderSum} + [\mathsf{Ord}(n_i) - \mathsf{Ord}(n_1) + 1] \times R'_{\ell-1} \end{aligned}$$

A.4 Proof of Proposition 2

Case 1: Let n_d be a descendant of n_b at level ℓ_b and n_d follows n_b in document order. Additionally, let ℓ_{fd} be the level of the leaf node representing n_d and ℓ_{fb} the level of the leaf node representing n_b. Let $\Delta = n_d.\mathsf{DeweyOrderSum} - n_b.\mathsf{DeweyOrderSum}$. Then,

$$\Delta = \sum_{j=2}^{\ell_{fd}} \Phi_d(j) - \sum_{j=2}^{\ell_{fb}} \Phi_b(j)$$

Since n_d is a descendant of n_b, $\mathsf{Ord}(n_b, j) = \mathsf{Ord}(n_d, j)$ for $1 \leq j < \ell_d$. Then, $\Phi_d(j) - \Phi_b(j) = 0 \; \forall \; j \leq \ell_d$. Also, $\ell_b < \ell_d$. Thus,

$$\Delta = \sum_{j=\ell_b+1}^{\ell_{fd}} \Phi_d(j) - \sum_{j=\ell_b+1}^{\ell_{fb}} \Phi_b(j)$$

Since $\sum_{j=k}^{\ell} \Phi(j) < R'_{k-2}$ (Lemma 1),

$$\begin{aligned} \sum_{j=\ell_b+1}^{\ell_{fd}} \Phi_d(j) - R'_{\ell_b-1} &< \Delta < R'_{\ell_b-1} - \sum_{j=\ell_b+1}^{\ell_{fb}} \Phi_b(j) \\ -R'_{\ell_b-1} &< \Delta < R'_{\ell_b-1} \\ n_b.\mathsf{DeweyOrderSum} - R'_{\ell_b-1} &< n_d.\mathsf{DeweyOrderSum} \\ &< n_b.\mathsf{DeweyOrderSum} + R'_{\ell_b-1} \end{aligned}$$

All descendants of n_b must satisfy the above inequality. Therefore, if n_a.DeweyOrderSum $\geq n_b$.DeweyOrderSum + $R'_{\ell_{b-1}}$ where $\ell_b > 1$, then n_a cannot be a descendant of n_b. Furthermore, given the total ordering of DeweyOrderSum among nodes and $R'_j > 0$ for $j > 0$, n_a must follow n_b.

Case 2: We must show that if n_a.DeweyOrderSum $< n_b$.DeweyOrderSum, then n_a is not a descendant of n_b, n_a is not an ancestor of n_b, and n_a precedes n_b. Let n_d be a descendant of n_b. n_b.DeweyOrderSum is the DeweyOrderSum of the first descendant of n_b. Then n_d.DeweyOrderSum $\geq n_b$.DeweyOrderSum. Hence, if n_a.DeweyOrderSum $< n_b$.DeweyOrderSum, then n_a is not a descendant of n_b. Since SUCXENT++ does not store non-leaf nodes, it is guaranteed that selected nodes are not ancestors of n_b. Finally, DeweyOrderSum is a total order among nodes, and hence if n_a.DeweyOrderSum $< n_b$.DeweyOrderSum, then n_a must precede n_b.

A.5 Proof of Proposition 3

Case 1: First we show that n_a follows n_b. From Proposition 2, since $n_a.\mathsf{DeweyOrderSum} \geq n_b.\mathsf{DeweyOrderSum} + R'_{\ell_b - 1}$, then n_a follows n_b in document order and n_a is not a descendant of n_b. To show that n_a is sibling of n_b, we need to prove that the nearest common ancestor is at level ℓ_{b-1}. Based on Theorem 1

$$n_a.\mathsf{DeweyOrderSum} - n_b.\mathsf{DeweyOrderSum} < (R'_{\ell_b - 2} - 1)/2 + 1$$
$$n_a.\mathsf{DeweyOrderSum} < n_b.\mathsf{DeweyOrderSum} + (R'_{\ell_b - 2} - 1)/2 + 1$$

Case 2: First we show that n_a precedes n_b. Since $n_b.\mathsf{DeweyOrderSum} < n_a.\mathsf{DeweyOrderSum}$, then n_a precedes n_b in document order and n_a is not a descendant of n_b. To show that n_a is sibling of n_b, we need to prove that the nearest common ancestor is at level ℓ_{b-1}. Based on Theorem 1

$$n_b.\mathsf{DeweyOrderSum} - n_a.\mathsf{DeweyOrderSum} < (R'_{\ell_b - 2} - 1)/2 + 1$$
$$n_a.\mathsf{DeweyOrderSum} > n_b.\mathsf{DeweyOrderSum} - [(R'_{\ell_b - 2} - 1)/2 + 1]$$