Efficient Evaluation of High-Selective XML Twig Patterns with Parent Child Edges in Tree-Unaware RDBMS

Sourav S. Bhowmick[1,2] Erwin Leonardi[1,2] Hongmei Sun[2]

[1]Singapore-MIT Alliance, Nanyang Technological University, Singapore
[2]School of Computer Engineering, Nanyang Technological University, Singapore
{assourav, lerwin,sunh0001}@ntu.edu.sg

ABSTRACT

Recent study showed that native twig join algorithms and *tree-aware* relational framework significantly outperform *tree-unaware* approaches in evaluating structural relationships in XML twig queries. In this paper, we present an efficient strategy to evaluate *high-selective* twig queries containing *only parent-child relationships* in a tree-unaware relational environment. Our scheme is built on top of our SUCXENT++ system. We show that by exploiting the encoding scheme of SUCXENT++, we can devise efficient strategy for evaluating such twig queries. Extensive performance studies on various data sets and queries show that our approach performs better than a representative tree-unaware approach (GLOBAL-ORDER) and a state-of-the-art native twig join algorithm (TJFAST) on all benchmark queries with the highest observed gain factors being 243 and 95, respectively. Additionally, our approach reduces significantly the performance gap between tree-aware and tree-unaware approaches and even outperforms a tree-aware approach (MONETDB/XQUERY) for certain high-selective twig queries. We also report our insights to the plan choices a relational optimizer made during twig query evaluation by *visually* characterizing its behavior over the relational selectivity space.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems – *Relational databases*.

General Terms: Algorithms, Design, Experimentation.

Keywords: XML, tree-unaware RDBMS, twig query evaluation, parent child edges.

1. INTRODUCTION

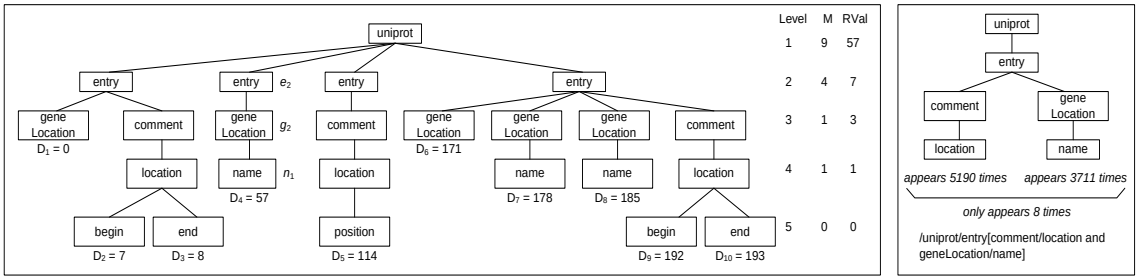
Finding all occurrences of a twig pattern in a database is a core operation in XML query processing. The basic strategy is to (i) first develop a labeling scheme to capture the structural information of XML documents, and then (ii) perform twig pattern matching based on the labels alone without traversing the original XML documents [11]. For the first sub-problem of designing appropriate labeling scheme, various methods have been proposed that are primarily based on tree-traversal order [1, 7, 8], region encoding [3] or path expressions [11, 15]. By applying these labeling schemes, one can determine the structural relationship between two elements in XML documents from their labels alone. The goal of second sub-problem of matching twig patterns is to devise efficient techniques for structural relationship matching.

In literature, evaluation strategies of twig pattern matching can be broadly classified into the following three types: (a) *binary-structure matching*, (b) *holistic twig pattern matching*, and (c) *string matching*. In the *binary-structure matching* approach, the twig pattern is first decomposed into a set of binary (parent-child and ancestor-descendant) relationships between pairs of nodes. Then, the twig pattern can be matched by matching each of the binary structural relationships against the XML database, and "stitching" together these basic matches [1, 6, 8, 10, 14]. In the *holistic twig pattern matching* approach, the twig query is decomposed into its corresponding path components and each decomposed path component is matched against the XML database. Next, the results of each of the query's path expressions are joined to form the result to the original twig query [3, 11]. Lastly, approaches like ViST [16] are based on *string matching* method and transform both XML data and queries into sequences and answer XML queries through subsequence matching.

It has been observed that typical twig queries on XML databases retrieve only a small portion of the data, generating, however, a large number of intermediate results [9]. Hence, one of the key challenge in twig query evaluation is to develop techniques that can reduce generation of large intermediate results. For instance, the binary-structure matching approaches may introduce very large intermediate results. Consider the sample document fragment from UNIPROTKB/ SWISS-PROT and the twig query in Figures 1(a) and 1(b), respectively. The path match ($e2$, $g2$, $n1$) for path `entry/geneLocation/name` does not lead to any final result since there is no `comment/location` path under $e2$. Note that this problem is exacerbated for queries that are *high-selective*[1] but each path in the query is *low-selective*. For example, the query in Figure 1(b) is high-selective as it returns only 8 results. However, all the paths are low-selective. Note that the number associated with each rooted path in the query represents the number of occurrences of the path in the XML database. To solve this problem, the *holistic twig pattern matching* has been developed in order to minimize the intermediate results. Although several of these approaches (such as *TwigStack* [3]) achieve optimality for twig queries with ancestor-descendant (AD) relationships, these solutions may still generate large numbers of useless matches when the queries contain parent-child (PC) relationships [10]. In this paper, we propose a novel approach for evaluating *high-selective* twig queries containing *parent-child relationship*.

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

CIKM '07, November 6–8, 2007, Lisboa, Portugal



(a) Example of XML Data (b) Example of Twig Query

Figure 1: Example of XML data and twig query.

1.1 Motivation

Recently, several native implementation of holistic twig join algorithms have focused on efficient evaluation of twig queries containing PC relationships [4, 10, 11]. Simultaneously, finding ways to evaluate such twig queries in relational environment has gained significant momentum in recent years. Specifically, there has been a host of work [2, 3, 5, 7, 8] on enabling relational databases to be *tree-aware* by invading the database kernel to implement XML support. On the other side of the spectrum, some completely jettison the approach of internal modification of the RDBMS for twig query processing and resort to alternative *tree-unaware* approach [6, 13, 14, 15] where the database kernel is not modified in order to process XML queries.

While the state-of-the-art tree-aware and native approaches are certainly innovative and powerful, we have found that these strategies are not directly applicable to relational databases. The RDBMS systems need to enhance their array of query processing strategies by incorporating special purpose external index systems, algorithms and storage schemes to perform efficient XML query processing. However, such invasion of the system's kernel seems hardly an option for any DBMS vendor. On the other hand, there are considerable benefits in tree-unaware approaches with respect to portability as they do not invade the database kernel. Consequently, they can easily be incorporated in an off-the-shelf RDBMS. However, one of the key stumbling block for the acceptance of tree-unaware approaches has been query performance. To get a better understanding of this problem, we experimented with 100MB and 1GB XBench DCSD datasets [18]. We use four *high-selective* twig queries containing only PC relationships as shown in Figure 2 where K denotes the number of subtrees returned by a query. We compare the query evaluation times (denoted as T_i where i denotes an approach) of GLOBAL-ORDER [15] (denoted as GO), a tree-unaware approach, with a twig join algorithm (TJFAST [11] (denoted as TJ)) and a tree-aware approach (MONETDB/XQUERY [2] (denoted as MX)). One can observe that native and tree-aware approaches significantly outperform GO for all queries. *Is it possible to design a tree-unaware storage scheme that can significantly reduce this performance gap between these approaches or outperform them?* In this paper, we show that it is indeed possible to devise such tree-unaware strategy for high-selective twig queries containing *only parent child edges*.

Query	T_{GO}/T_{TJ} (1GB)		T_{GO}/T_{MX} (100MB)	
	K=20	K=100	K=20	K=100
/catalog/item/publisher/contact_information [FAX_number and web_site]	6.23	7.49	23.26	26.16
/catalog/item/publisher/contact_information [FAX_number and web_site and phone_number]	3.22	2.37	27.79	32.34
/catalog/item[related_items and pricing/ quantity_in_stock]	8.29	12.51	34.74	28.01
/catalog/item[related_items and attributes/ size_of_book]	8.72	11.25	33.53	27.37

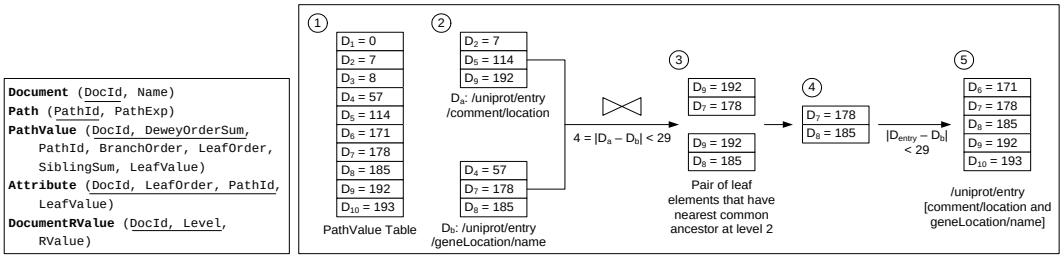
Figure 2: Performance of GO against TJFAST & MONETDB.

1.2 Overview

Our approach for twig query evaluation is based on the SUCXENT++ system [13], a tree-unaware approach designed primarily for query-mostly workloads. It stores only leaf elements, their corresponding data values, auxiliary encodings and root-to-leaf paths. The key features of SUCXENT++ are as follows. Firstly, it uses a novel labeling scheme that *does not* require labeling of internal elements in the XML tree. These labels are designed to process *ordered* XPATH queries efficiently [13]. In this paper, we shall show that these labels can also be used to efficiently evaluate high-selective twig queries containing PC edges only. Specifically, we use the DeweyOrderSum and RValue attributes to evaluate twig queries efficiently. Secondly, by storing only root-to-leaf paths it has lower storage size and, consequently, lower I/O-cost for query processing. Thirdly, it uses a novel *query hints-based strategy* to efficiently evaluate XPATH queries by enforcing "left-to-right" join order.

Our study revealed that our approach significantly outperforms TJFAST [11] and GLOBAL-ORDER [15] for 1GB dataset. Specifically, for 76% of the benchmark queries, SUCXENT++ is 14 - 243 times faster than GLOBAL-ORDER. It is also 3-95 times faster than TJFAST for 74% of benchmark queries. Furthermore, it significantly reduces the performance gap with MONETDB/XQUERY. While both TJFAST and GLOBAL-ORDER are slower than MONETDB for all benchmark queries, interestingly, for several high-selective twig queries SUCXENT++ has comparable performance or even faster than MONETDB. In summary, the main contributions of this paper are as follows.

- Based on a novel labeling scheme, in Section 3, we present an efficient algorithm for twig query evaluation by efficiently determining nearest common ancestor (NCA) of two elements in an XML document. Importantly, our proposed algorithm is capable of working with any off-the-shelf RDBMS without any internal modification.
- Through an extensive experimental study in Section 4, we show that our approach significantly outperforms state-of-the-art tree-unaware schemes and native twig join algorithms for evaluating high-selective twig queries containing only PC relationships. Additionally, our approach reduces significantly the performance gap with a tree-aware (MONETDB [2]) approach and even outperform it for certain queries.
- In Section 5, we provide insights to the plan choices a relational optimizer makes during twig query evaluation by *visually* characterizing its behavior over the relational selectivity space. To the best of our knowledge, this is the first effort



(a) SUCXENT++ Schema (b) Example of Twig Query Evaluation

Figure 3: SUCXENT++ schema and an example of twig query evaluation.

that attempts to systematically study the behavior of the optimizer in the *context of* XPATH *processing*.

Next, we briefly review the storage scheme of SUCXENT++.

2. SCHEMA OF SUCXENT++

The schema of Sucxent++ [13] is shown in Figure 3(a). The Document table 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 the Path table. For each leaf element n in T, we shall create a tuple in the PathValue table.

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

To discuss DeweyOrderSum, SiblingSum and RValue, we introduce some auxiliary definitions. Consider a sequence of leaf elements C: $\langle n_1, n_2, n_3, \ldots, n_r \rangle$ in T. Then, C is a *k-consecutive leaf elements* 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 element 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 elements* of T, denoted as M_k, if there does not exist a k-consecutive leaf elements C' and $|C|<|C'|$.

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

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

To evaluate non-leaf elements, we define the *representative leaf element* of a non-leaf element n to be its first descendant leaf element. Note that the BranchOrder attribute records the level of the NCA of two consecutive leaf elements. Let C be the sequence of descendant leaf elements of n and n_1 be the first element in C. We know that the NCA of any two consecutive elements in C is also a descendant of element n. This implies (a) except n_1, BranchOrder of an element in C is at least the level of element n and (b) the NCA of n_1 and its immediately preceding leaf element is not a descendant of element n. Therefore, BranchOrder of n_1 is always smaller than the level of n. The reader may refer to [13] for details on how these attributes are used to efficiently evaluate *ordered* XPATH axes (following, preceding, following-sibling, and preceding-sibling) and position predicates. Observe that the above implementation of SUCXENT++ is purely relational in the sense that we do not require to invade the relational database kernel to implement XPATH support.

3. EVALUATION OF TWIG QUERIES

In this section, we present the evaluation strategy of twig queries containing parent-child edges in SUCXENT++.

3.1 Data Model

We model XML documents as ordered trees. In our model we ignore comments, processing instructions and namespaces. Queries in XML query languages make use of twig patterns to match relevant portions of data in an XML database. The twig pattern node may be an element tag, a text value or a wildcard "*". We distinguish between query and data nodes by using the term "node" to refer to a query node and the term "element" to refer to a data element in a document. In this paper, we focus only on twig pattern edges that represent parent-child relationships (denoted by "/").

A twig query can be considered as a collection of *rooted path* patterns, where a *rooted path* pattern (RP) is a root-to-leaf path in the query. Each rooted path represents a sequence of nodes having parent-child edges. 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*. For example, Figure 1(b) is an example of a twig query containing parent-child edges and has two rooted paths:

uniprot/entry/comment/location and uniprot/entry/geneLocation/name. Further, the node entry is a NCA node with NCA-level two.

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

3.2 Twig Pattern Matching

Given a twig query Q and document D, our goal is to use the encoding scheme of SUCXENT++ to efficiently determine those root-to-leaf paths that satisfy Q. Note that these paths must satisfy the NCA node constraints of the twig query. For example, consider the twig query in Figure 1(b) over the document in Figure 1(a). The root-to-leaf paths in the document must satisfy the following conditions. First, they must be instances of the rooted paths uniprot/entry/geneLocation/name and uniprot/entry/comment/location. Second, the NCA element of an instance of a pair of these rooted paths must be an instance of the entry node having NCA-level equal to 2. Consequently, the root-to-leaf paths represented by the DeweyOrderSums D_4 and D_5 in Figure 1 do not satisfy the query but the paths represented by D_8, D_9, and D_{10} do. Hence, given a set of root-to-leaf paths satisfying the rooted paths of a twig query, the key challenge here is to determine efficiently whether these paths satisfy the NCA node constraints in the query. In this section, we shall discuss how this issue is addressed in SUCXENT++. We begin by formally introducing the following lemma that we will be using subsequently.

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 element in an* XML *document at level* ℓ*.* □

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

In SUCXENT++, the first sibling has local order equal to 1. Given Ord(n,t) of n at each level $t \in [k,\ell]$, any ancestors of n at level $t-1$ has at least [Ord(n,t)-1] that are not n nor n's ancestor. Each of these elements either is a leaf element, or has at least one descendant leaf element. Hence, an ancestor of n at level $t-1$ has, excluding n, at least [Ord(n,t)-1] descendant leaf elements, 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 an element at level $k-1$ with at least $(\sum_{t=k}^{\ell}$[Ord(n,t)-1]$)+1$ descendant leaf elements (including n). This implies that $\sum_{t=k}^{\ell}$[Ord(n,t)-1] $\leq |M_{k-1}|$. Therefore,

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

□

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Hence, elements 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. □

Similarly, we can prove the following lemma. Due to space constraints, the proof is given in [17].

```
evaluatePC-TwigQuery ( queryTwig )

01 i = 1
02 for every rootedPath in the queryTwig {
03   from_sql.add("PathValue as Vi")
04   where_sql.add("Vi.pathid in rootedPathi.getPathId()")
05   where_sql.add("Vi.branchOrder < rootedPathi.level()")
06   if (i > 1) {
07     where_sql.add("Vi.DeweyOrderSum BETWEEN
         Vi-1.DeweyOrderSum –
           RValue(rootedPathi.NCAlevel() - 1) + 1 AND
         Vi-1.DeweyOrderSum +
           RValue(rootedPathi.NCAlevel() - 1) - 1")
08   }
09   i++
10 }
11 select_sql.add("DISTINCT Vi-1.docId, ... , Vi-1.DeweyOrderSum")
12 order_sql.add("ORDER BY Vi-1.docId, Vi-1.DeweyOrderSum")
13 if ((i-1)>1)
14    option_sql.add("OPTION (FORCE ORDER)");
15    return select_sql + from_sql + where_sql + order_sql + option_sql
16 else
17    return select_sql + from_sql + where_sql + order_sql
```

(a) *evaluatePC-TwigQuery* algorithm

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


01 SELECT DISTINCT V2.DocId, V2.BranchOrder, V2.DeweyOrderSum,
        V2.PathId, V2.LeafValue, V2.LeafOrder
02 FROM PathValue V1, PathValue V2
03 WHERE V1.pathid in (2,3,4)
04 AND V1.branchOrder < 4
05 AND V2.docId = V1.docId
06 AND V2.pathid in (5)
07 AND V2.branchOrder < 4
08 AND V2.DeweyOrderSum BETWEEN
     V1.DeweyOrderSum - CAST(29 as BIGINT) + 1 AND
     V1.DeweyOrderSum + CAST(29 as BIGINT) – 1
09 ORDER BY V2.DocId, V2.DeweyOrderSum
10 OPTION (FORCE ORDER)
```

(b) An example of Translated SQL query

Figure 4: *evaluatePC-TwigQuery* algorithm.

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

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

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

Let us illustrate with an example the above lemmas and theorem. Consider the last leaf element in Figure 1. The DeweyOrderSum of this element is 193. Let D_1 be the DeweyOrderSum of leaf elements that have NCA at level 2. Using the above theorem, D_1 falls within the following range: $(R_2 - 1)/2 + 1 \leq |D_1 - 193| < (R_1 - 1)/2 + 1 \Rightarrow 4 \leq |D_1 - 193| < 29$ which returns the sixth, seventh, and eighth leaf elements (DeweyOrderSums are 171, 178, and 185, respectively). Let D_2 be the DeweyOrderSum of leaf elements that have NCA at level 4. Then D_2 falls within the following range: $(R_4 - 1)/2 + 1 \leq |D_2 - 193| < (R_3 - 1)/2 + 1 \Rightarrow 1 \leq |D_2 - 193| < 2$ which returns the ninth leaf element (DeweyOrderSum is 192). Now let say we want to get the leaf elements that have NCA at level 2 or deeper and let D_3 be the DeweyOrderSum of these elements. D_3 falls within the following range: $|D_3 - 193| < (R_1 - 1)/2 + 1 \Rightarrow |D_3 - 193| < 29$ (Lemma 2) which returns the last five leaf elements.

We now illustrate Theorem 1 further in the context of a twig query. Consider the query in Figure 1(b) and the fragment of the PathValue table in Figure 3(b) (Step 1). Note that for clarity, we only show the DeweyOrderSums of the root-to-leaf paths in the PathValue table. Let D_a be DeweyOrderSum of the representative leaf elements satisfying `/uniprot/entry/comment/location` (second, fifth, and ninth leaf elements) and D_b be DeweyOrderSum of the representative leaf elements satisfying `/uniprot/entry/geneLocation/name` (fourth, seventh, and eighth leaf elements). This is illustrated in step 2 of Figure 3(b). From the query we know that D_a and D_b have NCA at level 2 (`/uniprot/entry` level). Hence, based on Theorem 1 we can find pairs of (`location`,`name`) elements which have NCA at level 2. D_a and D_b fall on the following range: $(R_2 - 1)/2 + 1 \leq |D_a - D_b| < (R_1 - 1)/2 + 1 \Rightarrow 4 \leq |D_a - D_b| < 29$ which return the (seventh, ninth) and (eighth, ninth) leaf elements pairs (Step 3 of Figure 3(b)). We can easily return the `entry` subtree by applying Lemma 2 on any one of these pairs (Steps 4 and 5 of Figure 3(b)). Observe that the leaf element pairs with DeweyOrderSums 7 and 57 do not satisfy the above condition and hence do not satisfy the query. Note that since from the XPATH we know that D_a and D_b cannot have NCA at level greater than 2, we only need to use Lemma 2 for matching twig patterns. Observe that the above approach can reduce unnecessary comparison as we can determine the NCA directly by using the DeweyOrderSum and RValue attributes.

3.3 Query Translation Algorithm

Given a query twig (XPATH), the `evaluatePC-TwigQuery` procedure (Figure 4(a)[2]) outputs SQL statement. A SQL statement consists of four clauses: $select\_sql$, $from\_sql$, $where\_sql$, and $order\_sql$. In addition, we also have an option clause ($option\_sql$) to enforce a "left-to-right" join order on the translated SQL query using query hints. The performance benefits in SUCXENT++ because of such enforcement is discussed in [13]. 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, for a single XML document, we also preprocess RValue.

The procedure firstly breaks the query twig into its subsequent rooted path (Line 02). Then for every rooted path, it gets the representative leaf elements of the rooted path by using PathId and BranchOrder (Lines 04-05). After that, for the second rooted path onwards, it uses Lemma 2 to get the pair of leaf elements that have NCA at the NCA-level (Line 07). After processing the set of rooted paths, we return all attributes of the rightmost rooted path except SiblingSum (Line 11). As the results must be in document order, we sort it according to the DocId and DeweyOrderSum attributes (Line 12). Finally, if there are more than one table joined in the SQL query (Line 13), then the procedure adds the option clause to the SQL query (Lines 14–15) and returns the final SQL statement. Otherwise, the procedure directly returns the SQL statement (Line 17). For example, consider the query in Figure 1(b). The output SQL statement is shown in Figure 4(b). Lines 03-04 and 06-07 are used to get the representative leaf elements of the respective rooted path. Line 08 is used to get the pair of leaf elements that have NCA at the NCA-level. Line 09 sorts the results by the DocId and DeweyOrderSum attributes. Line 10 enforces the join order option.

4. PERFORMANCE STUDY

We now present the performance results of our proposed approach and compare it with a state-of-the-art tree-unaware approach

[2]We assume the attribute and element nodes are stored in PathValue table instead of separate tables as shown in Figure 3(a).

ID	Average Number of Nodes			Depth	File Size (MB)	DB Size (MB)
	Internal	Leaf	Total			
DC10	77,803	147,431	225,234	8	10	16
DC100	774,487	1,467,713	2,242,200	8	100	148
DC1000	7,751,764	14,690,848	22,442,612	8	1000	1487

(a) Data sets

Query	XPath	No. of PathValue tables
Q1	/catalog/item/publisher/contact_information[FAX_number and web_site]	3
Q2	/catalog/item/authors/author[name/middle_name and contact_information/mailing_address/name_of_state]/ contact_information/email_address	3
Q3	/catalog/item/publisher/contact_information[FAX_number and web_site and phone_number]	4
Q4	/catalog/item[related_items and pricing/quantity_in_stock]	3
Q5	/catalog/item[related_items and attributes/size_of_book]	3
Q6	/catalog/item/publisher/contact_information[FAX_number and web_site and phone_number and mailing_address/ name_of_state]	5
Q7	/catalog/item[related_items and attributes/size_of_book]/publisher/contact_information[FAX_number and web_site]	6

(b) Benchmark Twig Query Set

Query	K	10MB		100MB		1000MB	
		Min	Max	Min	Max	Min	Max
Q1	0 - 250	589	2,148	6,210	18,819	62,384	188,157
Q2	0 - 250	3,076	3,285	30,997	31,483	425,303	425,553
Q3	0 - 250	394	1,756	4,179	16,644	41,636	415,219
Q4	0 - 250	1,201	1,377	12,335	12,737	124,768	125,466
Q5	0 - 250	1,233	1,379	12,414	12,654	124,630	125,470
Q6	0 - 250	316	1,933	3,085	18,858	31,321	437,635
Q7	0 - 250	600	2,044	6,181	18,866	62,067	188,123

(c) Statistics of instances of rooted paths

	10MB	100MB	1000MB
GO	18,774	142,713	1,466,554
SX	11,679	112,129	1,089,090
TJFast	75,268	794,112	8,750,003

(d) Shredding and Label Generation Times (msec)

Figure 5: Query and data sets.

and a native implementation of twig join algorithm. Since there are several tree-unaware schemes proposed by the community, our selection choice was primarily influenced by the following two criteria. First, the representative storage scheme should not be dependent on the availability of DTD/XML schema. Second, the selected approach must have good query performance for a variety of XPATH axes (ordered as well as unordered) for *query-mostly* workloads. Hence, we chose the GLOBAL-ORDER storage scheme as described in [15]. We chose TJFAST [11] as a representative of native implementation of twig join algorithm because it has better performance in terms of I/O cost and CPU time compared to *TwigStack* [3] and *TwigStackList* [10]. Note that we did not select TWIG^{2}STACK [4] as we were unable to get the implementation from the authors due to legal reasons. However, we shall still compare our results intuitively based on the results presented in [4].

Prototypes for SUCXENT++ (denoted as SX), GLOBAL-ORDER (denoted as GO), and TJFAST[3](denoted as TJ) were implemented with Java. The experiments were conducted on an Intel Pentium 4 3GHz machine running on Windows XP with 1GB of RAM. The RDBMS used was Microsoft SQL Server 2005 Developer Edition.

Data and Query Sets: In our experiments, we used XBench DCSD [18] as synthetic dataset. We vary the size of XML documents from 10MB to 1GB (denoted as DC10, DC100, and DC1000, respectively). For simplicity, we generate only element nodes (no attribute nodes). Figure 5(a) shows the characteristics of the datasets used. Recall that we wish to explore twig queries that are high-selective although the paths are low-selective. Hence, we modified XBench dataset so that we can control the number of subtrees (denoted as K) that matches a twig query and the number of instances of the rooted paths in the XML document. We set $K \in \{0, 10, 20, 50, 100, 250\}$. Note that $K = 0$ is significant in an environment where users would like to issue exploratory ad hoc queries. In this case, the user would like to know quickly if the query returns any results. If the result set is empty then he/she can further refine his/her query accordingly. Figure 5(b) depicts the benchmark XPATH queries containing only PC edges as well as the number of PathValue tables involved in the translated SQL. The corresponding SQL queries are given in [17]. Note that these queries have different twig structures in terms of depth of the twig, number of rooted paths, and number of NCA nodes. The number of occurrences of subtrees that satisfies a twig query Q and the minimum and maximum numbers of instances of rooted paths of Q in the datasets are shown in Figure 5(c).

Test Methodology: Appropriate indexes were constructed for all approaches (except for TJFAST) through a careful analysis on the benchmark queries. Particularly, for SUCXENT++ we create the following indexes on PathValue table: (a) unique clustered index on PathId and DeweyOrderSum, and (b) non-unique, non-clustered Index on PathId and BranchOrder. Furthermore, since our dataset consists of a single XML document, we removed the DocId column from the tables in SX and GO. Prior to our experiments, we ensure that statistics had been collected. The bufferpool of the RDBMS was cleared before each run. The queries were executed in the *reconstruct* mode [15] where not only the internal nodes are selected, but also all descendants of those nodes. Each query was executed 6 times and the results from the first run were always discarded.

[3]The implementation of this algorithm was kindly provided by the first author Jiaheng Lu from the National University of Singapore.

4.1 Shredding and Label Generation Times

Prior to processing twig queries in SX (or GO), it is necessary to shred the XML document into the RDBMS. In the case of TJ, it is necessary to generate the *extended dewey* labels [11] before it can be used for twig query evaluation. Figure 5(d) show the insertion times for the SX and GO approaches as well *extended dewey* generation time of TJ. It can be observed that SUCXENT++ performs the best for all experiments as it stores the least amount of data compared to GO. Also, observe that the shredding time of SX is up to 8 times faster than extended dewey generation time for TJ.

4.2 Query Evaluation Times

Figure 6 depicts the twig query evaluation times of SX, GO, and TJ for different values of K.

Comparison with GLOBAL-ORDER (GO): First, we observe that SX significantly outperforms GO. As the data size increases, the difference between SX and GO increases. For DC10 dataset, SX is $2-36$ times faster than GO for 74% queries (31 out of 42 queries). On the other hand, GO is faster than SX for queries $Q4$ and $Q5$ when $K \geq 20$ and when $K = 10$ and $K \geq 50$, respectively. However, for DC100, SX is faster than GO for 95% (40 out of 42) queries. Particularly, for 81% queries it is 2-62 times faster than GO. For DC1000 dataset, SX is faster than GO for *all* queries. In fact, we noticed that it is $10 - 243$ times faster for 76% of the benchmark queries (all queries except for $Q6$ and $Q7$ when $K \geq 10$).

The reasons for such significant performance differences between SX and GO can be summarized as follows. Firstly, SX uses an efficient strategy based on Theorem 1 to reduce useless comparisons. Furthermore, the number of join operations in GO is more than SX. For example, for *Q2*, GO and SX join eight tables and three tables, respectively. Secondly, GO stores every element of an XML document whereas SX stores only the root-to-leaf paths. Consequently, the number of tuples in the Edge table is much more than that in the PathValue table.

Another interesting observation is that the performance difference between SX and GO grows with the increase in selectivity of the queries (lower K value) as well as data size. For $K = 250$,

	DC10			DC100			DC1000		
	SX	TJ	GO	SX	TJ	GO	SX	TJ	GO
Q1	28.00	157.00	733.80	143.40	1,047.00	2,994.00	1,297.20	9,734.00	80,493.80
Q2	37.80	937.00	1,309.40	318.20	7,750.00	6,146.60	4,336.80	78,954.00	111,085.40
Q3	22.00	328.00	793.60	108.80	3,188.00	4,282.20	867.60	31,203.00	100,358.40
Q4	30.60	140.00	480.20	139.80	969.00	2,960.80	1,321.40	9,328.00	104,983.60
Q5	28.80	156.00	618.20	146.60	922.00	2,829.60	1,322.60	8,437.00	79,768.80
Q6	16.00	563.00	554.60	90.40	5,453.00	5,570.40	679.80	64,219.00	165,011.60
Q7	104.60	531.00	509.20	667.00	3,297.00	3,190.80	6,421.80	55,625.00	121,077.40

(a) K = 0

	DC10			DC100			DC1000		
	SX	TJ	GO	SX	TJ	GO	SX	TJ	GO
Q1	102.40	234.00	683.00	185.40	1,047.00	2,912.60	1,297.20	9,922.00	62,137.40
Q2	49.20	843.00	1,000.60	940.20	7,828.00	4,765.40	9,223.80	79,906.00	155,225.20
Q3	67.20	390.00	685.00	233.60	3,188.00	4,599.40	1,899.80	31,172.00	97,541.00
Q4	511.60	187.00	572.60	702.20	968.00	2,838.40	2,072.00	8,484.00	100,562.00
Q5	566.60	172.00	547.60	583.00	938.00	3,125.60	1,322.60	8,406.00	89,932.00
Q6	85.60	625.00	1,027.80	2,651.20	5,516.00	5,332.40	25,132.40	59,437.00	156,453.40
Q7	140.00	609.00	619.00	3,789.20	3,297.00	3,643.00	33,910.20	58,093.00	139,502.80

(b) K = 10

	DC10			DC100			DC1000		
	SX	TJ	GO	SX	TJ	GO	SX	TJ	GO
Q1	106.40	172.00	768.40	281.40	1,063.00	3,052.40	1,394.60	9,781.00	60,914.40
Q2	44.80	938.00	1,341.80	970.40	7,891.00	5,597.60	9,294.80	79,672.00	148,968.60
Q3	108.80	328.00	810.80	257.60	3,172.00	4,429.20	1,887.20	31,359.00	100,982.60
Q4	715.80	140.00	569.00	1,378.20	1,016.00	3,365.60	2,427.00	8,484.00	70,298.00
Q5	507.00	156.00	586.60	828.80	953.00	3,248.00	2,171.60	8,453.00	73,674.40
Q6	83.60	562.00	837.00	2,648.00	5,531.00	5,604.40	25,279.00	58,625.00	134,971.60
Q7	167.40	547.00	898.20	3,653.60	4,421.00	4,966.00	33,984.00	61,516.00	217,533.60

(c) K = 20

	DC10			DC100			DC1000		
	SX	TJ	GO	SX	TJ	GO	SX	TJ	GO
Q1	151.00	218.00	813.00	282.00	1,390.00	2,905.00	1,414.00	10,282.00	94,490.60
Q2	63.40	937.00	918.60	1,047.40	7,906.00	4,445.40	9,289.80	79,500.00	165,193.80
Q3	141.60	609.00	809.00	494.60	3,375.00	4,480.40	1,880.40	31,719.00	108,656.20
Q4	1,131.20	219.00	758.40	973.40	1,188.00	2,894.80	1,997.40	9,828.00	92,458.20
Q5	800.40	187.00	699.60	946.60	1,078.00	3,327.40	2,000.80	9,437.00	105,996.00
Q6	129.40	1,094.00	1,006.80	2,672.80	5,625.00	5,468.80	25,100.20	58,328.00	125,850.00
Q7	232.20	593.00	1,054.80	3,695.00	5,063.00	4,011.40	33,830.60	69,765.00	160,203.00

(d) K = 50

	DC10			DC100			DC1000		
	SX	TJ	GO	SX	TJ	GO	SX	TJ	GO
Q1	233.80	360.00	814.40	381.00	1,547.00	3,024.60	1,528.40	10,172.00	76,146.60
Q2	61.00	938.00	1,334.80	949.20	7,907.00	4,704.40	9,204.80	79,719.00	128,644.60
Q3	217.00	578.00	1,033.40	437.60	3,438.00	4,345.20	1,963.40	31,750.00	75,394.00
Q4	1,281.20	172.00	783.60	1,523.80	1,032.00	3,064.00	2,331.80	9,172.00	114,753.40
Q5	1,373.20	234.00	828.60	1,476.20	1,062.00	3,079.40	2,570.40	9,109.00	102,477.00
Q6	195.00	953.00	879.60	2,753.20	5,718.00	5,035.60	25,091.60	57,953.00	160,047.00
Q7	312.40	766.00	662.20	3,915.75	6,735.00	4,168.20	33,922.00	82,203.00	143,880.80

(e) K = 100

	DC10			DC100			DC1000		
	SX	TJ	GO	SX	TJ	GO	SX	TJ	GO
Q1	693.40	312.00	802.00	669.40	1,406.00	3,040.40	1,643.00	10,609.00	69,973.00
Q2	92.60	953.00	809.20	1,559.40	7,890.00	4,300.60	9,393.40	79,531.00	151,147.40
Q3	386.40	531.00	943.60	756.80	3,438.00	4,532.20	2,112.40	31,734.00	111,894.40
Q4	3,043.40	188.00	1,182.60	3,106.40	1,093.00	3,733.20	3,852.00	9,297.00	106,146.40
Q5	3,338.80	187.00	1,175.00	2,994.00	1,093.00	3,452.40	3,997.00	9,109.00	107,271.00
Q6	397.00	937.00	1,149.80	3,016.20	5,688.00	4,892.00	25,407.20	57,813.00	189,539.40
Q7	473.00	1,140.00	1,304.20	4,110.20	9,938.00	4,054.40	34,434.20	117,657.00	129,659.80

(f) K = 250

Figure 6: Query evaluation times (in msec).

SX is $3.8 - 53$ times faster than GO for DC1000 dataset. This reduces to up to 8.7 times for DC10 dataset. However, for $K = 0$, the observed gain factor increases from $5 - 36$ times for DC10 to $19 - 243$ times for DC1000.

Comparison with TJFAST (TJ): SX outperforms TJ with the increase in data size. For DC10 and DC100, it is $2 - 35$ and $2 - 60$ times faster than TJ, respectively, for 64% queries (27 out of 42 queries). On the other hand, for DC10 dataset, TJ outperforms SX for queries $Q4$ and $Q5$ when $K \geq 10$, and for $Q1$ when $K = 250$. For DC100, only 14% (6 queries) in TJ are faster than SX. Similar to GO, as data size increases to 1GB, SX outperforms TJ for all queries. Specifically, 74% queries in SX are $3 - 95$ times faster than TJ.

Similar to our observation on selectivity earlier, the performance difference between SX and TJ grows with the increase in selectivity of the queries and data size. For $K = 250$, SX is $2 - 15$ times faster than TJ for DC1000 dataset. This factor increases to $6.4 - 95$ times for $K = 0$.

Comparison with TWIG^{2}STACK: As mentioned earlier, we did not compare our approach with TWIG^{2}STACK as we were not able to obtain the source code from the authors. However, in [4], Chen et al. compared the performance of TWIG^{2}STACK with TJ. They showed that in general TWIG^{2}STACK is 2-3 times faster than TJ. Based on this observation, we expect SX to outperform TWIG^{2}STACK for majority of the benchmark queries as SX is at least 2 and 3 times faster than TJ for 76% (96 out of 126 queries) and 59% queries (74 queries), respectively, in all the datasets.

4.3 Comparison with MonetDB/XQuery

Recently, in [2], it has been shown that MONETDB/XQUERY, a *tree-aware* XQuery implementation built on the foundation of the main memory DBMS MONETDB, is among the fastest and most scalable XQuery processor and outperforms the current generation of XQuery systems by quite a big margin. Hence, we would like to observe how "far off" SX is from MONETDB in comparison with GO and TJ. We used the Windows version of MONETDB/XQuery 0.16.0 [2] (denoted as MX) downloaded from http://monetdb.cwi.nl/XQuery/Download/index.html (Win32 builds).

Figure 7 shows how SX, GO, and TJ perform compared to MX. We code each approach with "YN", where 'Y' is one of 'SX' (SUCXENT++), 'GO' (GLOBAL-ORDER), or 'TJ'(TJFAST), and 'N' is the dataset ('10' for DC10 and '100' for DC100). We define a *MonetDB Factor* (denoted as MXF) as T_{YN}/T_{MX} where T_{MX} and T_{YN} are the query evaluation times for MONETDB and SX/GO/TJ, respectively. Note that we did not show any results of MONETDB for 1GB dataset as it is currently vulnerable to the virtual memory fragmentation in Windows environment[4]. Consequently, it failed to shred 1GB XBench dataset.

We can make the following two key observations from Figure 7. First, *the performance gap between tree-unaware approaches and* MONETDB *is significantly reduced when it is compared against* SUCXENT++. Except for $Q4$ and $Q5$, SX has the best performance compared to GO and TJ. For $Q1$, $Q2$, and $Q3$, MX is at most 7 times faster than SX for majority of the queries whereas it is up to 32 times faster than GO and TJ. Specifically, MX is at least 10 times faster than GO and TJ for 95% and 55% of benchmark queries, respectively. However, only 31% (26 queries out of 84) of benchmark queries are at least 10 times slower in SX compared to MX. Second, *the performance difference between* SX *and* MX *is significantly reduced for majority of high-selective twig queries (low K value)*. Interestingly, SX has comparable performance or faster than MX for most of the benchmark queries when $K = 0$. Note that both TJ and GO are slower than MX for *all* benchmark queries.

5. CHARACTERIZING BEHAVIOR OF RELATIONAL OPTIMIZER

In this section, we explore and characterize how the relational optimizer behaves for evaluating twig queries over the relational selectivity space. We begin by introducing a powerful tool called PICASSO [12], which we shall be using subsequently.

5.1 PICASSO

Given a SQL *query template* and a relational engine, PICASSO [12] is a tool that automatically generates a variety of diagrams that characterize the behavior of the engine's optimizer over the relational selectivity space. It is operational on a suite of industrial-strength database query optimizers including DB2, Oracle, Sybase, and SQL Server. A PICASSO *query template* is an SQL query that additionally features predicates of the form "*relation.attribute :varies*" - these attributes are termed as *Picasso Selectivity Predicates* (PSP). The reader may refer to [17] for a sample template.

[4]We have confirmed this vulnerability with Peter Boncz, one of the architect of MONETDB/XQuery.

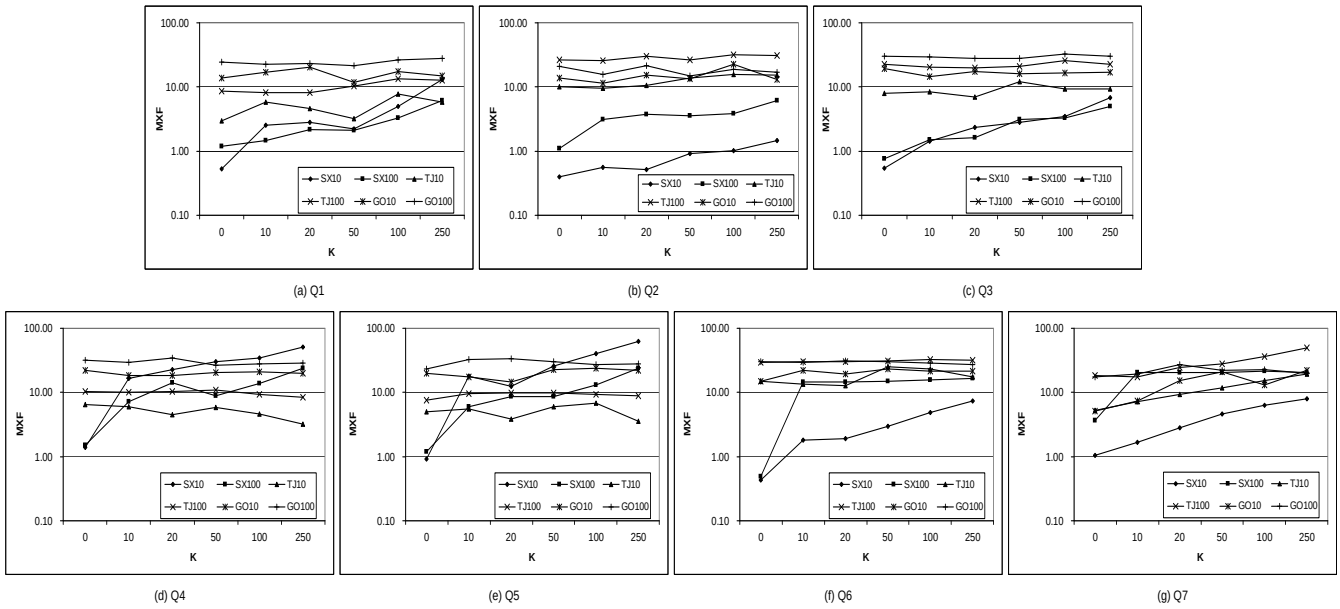


Figure 7: Comparison with MONETDB.

Query	DC10			DC100			DC1000		
	Plan Card	80% Coverage	Gini Index	Plan Card	80% Coverage	Gini Index	Plan Card	80% Coverage	Gini Index
Q1	21	42.86%	0.50	14	28.57%	0.67	11	45.45%	0.57
Q2	18	22.22%	0.73	11	36.36%	0.63	10	40.00%	0.56
Q3	27	29.63%	0.65	14	35.71%	0.61	10	30.00%	0.65
Q4	27	25.93%	0.65	14	35.71%	0.63	11	54.55%	0.41
Q5	21	28.57%	0.65	13	30.77%	0.67	8	37.50%	0.65
Q6	26	34.62%	0.63	16	31.25%	0.59	10	30.00%	0.67
Q7	33	39.39%	0.53	20	30.00%	0.60	12	41.67%	0.58
Average	**24.71**	**31.89%**	**0.56**	**14.57**	**32.62%**	**0.63**	**10.29**	**39.88%**	**0.58**

Figure 8: Skew in plan space coverage.

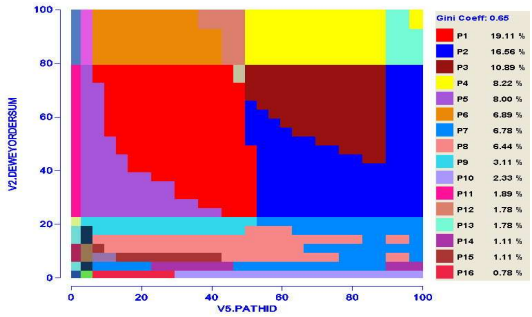


Figure 9: Plan Diagram of Q3 (DC10).

Each template defines an n-dimensional relational selectivity space, where n is the number of PSP relations. That is, the selectivity of each of the PSP relations is varied over the range [0-100%].

There are few conditions to satisfy for the choice of PSP. First, each relation can participate in at most one PSP. Second, the PSP relations should feature only in join predicates in the query, but not in any other equality or range predicates. Third, the PSP attributes must have pre-generated statistical summaries and should be on dense-domain attributes in high-cardinality relations.

With this information, the tool automatically generates SQL queries that are evenly spaced across the relational selectivity space (the statistics present in the database catalogs are used to compute the selectivities). For example, with a grid spacing of 100×100, a *plan diagram* is produced by firing 10000 queries, each query covering 0.01 percent of the plan diagram area. The resulting plans are stored persistently in the database, and in the postprocessing phase, a unique color is assigned to each distinct plan, and the area covered by the plan is also estimated. For each plan diagram, the corresponding *cost diagram* is obtained by feeding the query points and their associated costs to a 3-D visualizer.

5.2 Experimental Setup

We generate query templates for the query sets, $Q1$ through $Q7$, in Figure 5(b). We use all three XBench datasets when $K = 50$. To ensure coverage of the full range of selectivities, the relational axes in the plan diagrams are chosen from the large-cardinality tables occurring in the query (PathValue table). We chose the PathId and DeweyOrderSum attributes of the rightmost and leftmost tables in the FROM clause, respectively, as PSP. Note that for ease of presentation and visualization, the query workloads are restricted to 2-dimensional selectivity spaces. Further, we use a query grid spacing of 30×30, unless explicitly mentioned otherwise. Finally, as our focus here is characterizing the choices made by the optimizer, for every query the plan to execute the query was generated, but not executed.

5.3 Plan Diagram

A *plan diagram* in PICASSO is a color-coded pictorial enumeration of the execution plan choices of a database query optimizer over the relational selectivity space. In this section, we analyze the plan choices made by the optimizer for the queries in Figure 5(b).

Skewness of plan space coverage: We start off our analysis of plan diagrams by investigating the *skew* in the space coverage of the optimal set of plans. Figure 8 shows the skewness in plan space coverage for various benchmark queries. The "*Plan Card*" column represents the cardinality of the optimal plan set; the "*80% Coverage*" column represents the minimum percentage of plans required to cover 80% of the space, and the last column measures the Gini Index.

The statistics shown in Figure 8 leads to the following observations. Firstly, that the cardinality of the optimal plan set can reach high values for a large number of queries. Further, the average cardinality decreases with the increase in dataset size. Note that these numbers are conservative in that they are obtained with a 30×30 grid - with finer granularity grids, the plan cardinality may increase

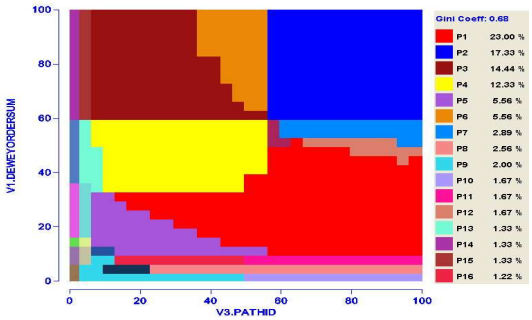


Figure 10: Plan Diagram of Q4 (DC10).

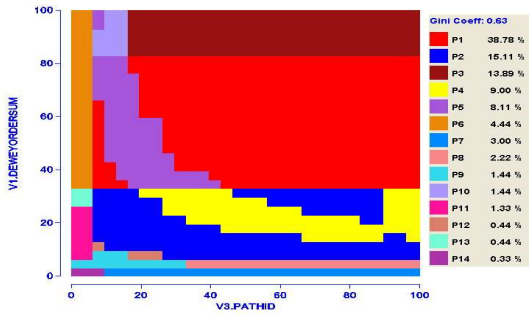


Figure 11: Plan Diagram of Q4 (DC100).

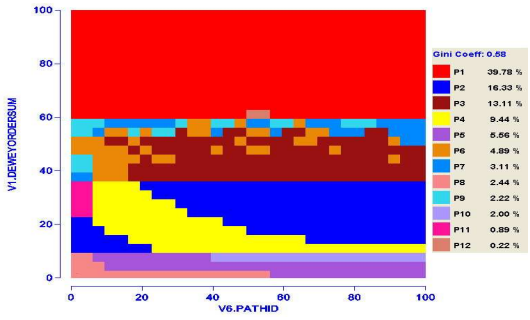


Figure 12: Plan Diagram of Q7 (DC1000).

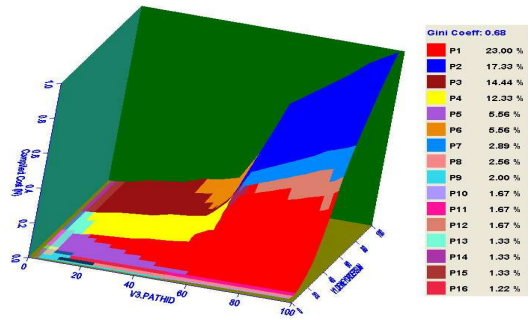


Figure 13: Cost Diagram of Q4 (DC10).

even further [12]. Secondly, in most queries 22% to 45% plans cover 80% of the space, highlighting the inequity in the plan space distribution. This is captured by the Gini index values, which are mostly in excess of 0.56, indicating high skew in the plan space distribution. In summary, the relational optimizer makes fine-grained choices in the context of XPATH evaluation.

Representative plan diagram patterns: We now present some representative patterns (Figures 9 to 12[5]) that emerged in the plan diagrams across some of the benchmark queries ($Q3$, $Q4$, and $Q7$) and datasets and summarize our observations. The reader may refer to [17] to view all diagrams. First, in all plan diagrams, we noticed that the variety of plans and their space distribution differs significantly for a specific query with the increase in data size. This is highlighted by the plan diagrams of $Q4$ in Figures 10 and 11 for datasets DC10 and DC100. Second, in several plan diagrams across different datasets, the optimizer frequently changes its plans when the selectivities of PathId and DeweyOrderSum are high (less than 15%). Majority of these plans, however, occupies a small area in the selectivity space. Third, we noticed that a given optimal plan may have *duplicates* in that it may appear in several disjoint locations. For example, $P4$ (yellow) occurs in two disjoint locations at the upper right quadrant in Figure 9. Similarly, $P5$ (violet) is present in upper left quadrant (around 33% – 80% selectivity of DeweyOrderSum axis) as well as top part (selectivity of DeweyOrderSum is more than 92%) of the plan diagram in Figure 11. Apart from duplicates, we also see existence of instances of *plan islands*, where a plan is completely enclosed by another. For example, in Figure 12 some of $P6$ (yellowish brown) is contained as a set of islands in $P3$ (brown) in the middle part of the diagram. In general, the reason for the occurrence of such duplicates and islands is because the optimizer makes fine-grained plan choices even though the costs of the competing plans are close to one another in that area [12]. For example, when plans $P6$ and $P3$ in Figure 12 are compared, we find that the former uses *hash match* whereas the latter does not employ any [17]. Finally, in majority of the plan diagrams, we find *plan switch-points* [12]. These are lines parallel to the axes that run through the entire selectivity space, with a plan shift occurring for all plans bordering the line. Specifically, we observed that there are one or more plan switch-points for majority of the plan diagrams when the selectivities of DeweyOrderSum or PathId are very high (typically, less than 20% and 10%, respectively).

[5]For clarity, we recommend viewing all diagrams presented in this section directly from the color PDF, or from a color print copy. Also, for clarity, sometimes we only show a subset of the color codes on the right hand side of the plan and cost diagrams.

5.4 Cost Diagram

Cost diagram is complimentary to the plan diagram and represents 3D-visualization of the estimated plan execution costs over the same selectivity space. We now present some representative patterns (Figures 13 to 15) that emerged in the cost diagrams across some of the benchmark queries ($Q4$ and $Q7$) and datasets and summarize our observation. The reader may refer to [17] to view all cost diagrams. The diagrams compliment the results in Figure 6(d). In majority of the diagrams the cost is low when the selectivities of DeweyOrderSum and PathId are high (Figures 14 and 15), especially for larger datasets, *confirming efficient evaluation times of* SX *for high-selective* XPATH *queries*. Consider the cost diagrams of $Q4$ in Figures 13 and 14 for another example. Note that the evaluation time of $Q4$ for DC10 is relatively higher than that for DC100 (Figure 6(d)). This is reflected in Figure 13 where the cost in DC10 increases at comparatively higher selectivity (around 40% selectivity of DeweyOrderSum axis) compared to that in DC100 (around 80% selectivity of DeweyOrderSum) depicted in Figure 14.

6. RELATED WORK

Our approach differs from existing tree-unaware techniques [6, 13, 14, 15] in the following ways. First, we use a novel and powerful numbering scheme that only encodes the leaf elements and the levels of the XML tree. In contrast, most of the tree-unaware

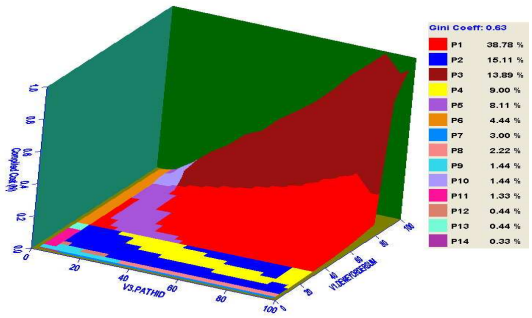


Figure 14: Cost Diagram of Q4 (DC100).

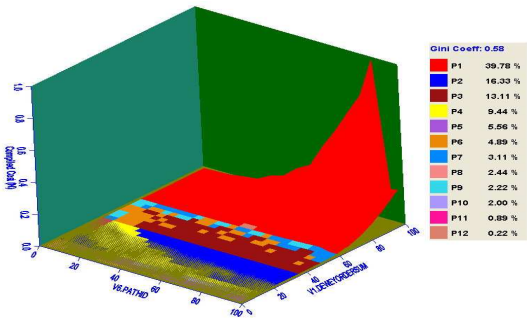


Figure 15: Cost Diagram of Q7 (DC1000).

approaches encode both internal and leaf elements. Second, the translated SQL of SUCXENT++ does not suffer from large number of joins. Specifically, if there are n rooted paths in the twig query then we use $n-1$ joins. Third, all previous tree-unaware approaches, reported query performance on XML documents with small/medium sizes – smaller than 500MB. We investigate query performance on large datasets (up to 1GB). This gives more insights on the scalability of the state-of-the-art tree-unaware approaches for twig query processing. Finally, for the first time, we report visually the plan choices a tree-unaware relational optimizer makes during twig query evaluation over the selectivity space.

In our previous work [13], we focused on efficiently evaluating ordered path expressions rather than tree-structured queries. In this paper, we investigate how the encoding scheme in [13] can be used for efficiently processing high-selective tree-structured twig queries containing only parent-child relationship.

Recently, there have been several efforts to efficiently evaluate twig queries containing PC relationships in native XML storage [4, 10, 11]. Our work differs in the following ways. First, we take relational-based approach and showed that this approach is more efficient especially for high-selective twig queries. Second, the native approaches typically report query performance on documents smaller than 600MB and containing at most 8 million nodes. In contrast, we explore the scalability of our approach for larger XML documents having more than 22 million nodes. In fact, our results show that SUCXENT++ is, in general, more scalable than these native approaches. Lastly, majority of the twig queries considered in the experiments of native strategies contain a combination of AD and PC relationships whereas we evaluate the "worst case" scenario where all relationships in the twig query are parent-child in nature.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we present an efficient strategy to evaluate high-selective twig queries having parent-child relationship in a tree-unaware relational environment. Our scheme is build on top of SUCXENT++ [13]. We showed that by exploiting the encoding scheme of SUCXENT++ we can reduce useless structural comparisons in order to evaluate twig queries. Our results showed that our proposed approach significantly outperforms GLOBAL-ORDER and TJFAST, two representative tree-unaware and native schemes, respectively. Although tree-aware approaches are often the best in terms of query performance [2], our scheme reduces significantly the performance gap between tree-aware and tree-unaware approaches and even outperform it for certain high-selective twig queries. Importantly, unlike tree-aware approaches, our scheme does not require invasion of the database kernel to improve query performance and can easily be built on top of any commercial RDBMS.

Additionally, using PICASSO, we attempted to analyze the behavior of a commercial relational optimizer on SQL queries translated from XML twig patterns. Our study revealed that all the queries have a large number of plans covering the relational space and are heavily skewed highlighting the fine-grained nature of the optimizer. The plan diagrams showed that the optimizer often makes frequent plan switches when the selectivities of PathId and DeweyOrderSum are high. The cost diagrams revealed that in general the cost is low when the selectivities of DeweyOrderSum and PathId are high especially for larger datasets. Thus, supporting efficient evaluation times of our approach for high-selective queries. In the future, we would like to conduct a deeper investigation of the behavior of relational optimizer for wider variety of XPATH queries.

Acknowledgements. We would like to thank Jayant Haritsa of Indian Institute of Science for providing the PICASSO software. We appreciate the effort of the PICASSO team for efficiently resolving several implementation issues related to PICASSO.

8. REFERENCES

[1] S. AL-KHALIFA, H.V. JAGADISH, J. M. PATEL ET AL. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *In ICDE*, 2002.
[2] 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.
[3] N. BRUNO, N. KOUDAS, D. SRIVASTAVA. Holistic Twig Joins: Optimal XML Pattern Matching. *In SIGMOD*, 2002.
[4] S. CHIEN, H-G. LI ET AL. Twig^{2}Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents. *In VLDB*, 2006.
[5] D. DEHAAN, D. TOMAN ET AL. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Coding. *In SIGMOD*, 2003.
[6] D. FLORESCU, D. KOSSMAN. Storing and Querying XML Data using an RDBMS. *IEEE Data Engg. Bulletin*. 22(3), 1999.
[7] T. GRUST, J. TEUBNER, M. V. KEULEN. Accelerating XPath Evaluation in Any RDBMS. *In ACM TODS*, 29(1), 2004.
[8] Q. LI, B. MOON. Indexing and Querying XML Data for Regular Path Expressions. *In VLDB*, 2001.
[9] W. LIAN, N. MAMOULIS, D. W. CHEUNG, S. M. LIU. Indexing Useful Structural Patterns for XML Query Processing. *In TKDE*, 17(7), July 2005.
[10] J. LU, T. CHEN, T. W. LING. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. *In CIKM*, 2004.
[11] J. LU, T. W. LING ET AL. From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. *In VLDB*, 2005.
[12] N. REDDY, J. HARITSA. Analyzing Plan Diagrams of Database Query Optimizers. *In VLDB*, 2005.
[13] B.-S SEAH, K. G. WIDJANARKO, S. S. BHOWMICK, B. CHOI, E. LEONARDI. Efficient Support for Ordered XPath Processing in Tree-Unaware Commercial Relational Databases. *In DASFAA*, 2007.
[14] J. SHANMUGASUNDARAM, K. TUFTE ET AL. Relational Databases for Querying XML Documents: Limitations and Opportunities. *In VLDB*, 1999.
[15] I. TATARINOV, S. VIGLAS, K. BEYER, ET AL. Storing and Querying Ordered XML Using a Relational Database System. *In SIGMOD*, 2002.
[16] H. WANG, S. PARK, W. FAN, P. S. YU. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. *In SIGMOD*, 2003.
[17] S. S. BHOWMICK, E. LEONARDI, H. SUN. Efficient Evaluation of High-Selective XML Twig Patterns with Parent Child Edges in Tree-Unaware RDBMS. *Technical Report*, 2007. Available at `http://www.cais.ntu.edu.sg/~assourav/TechReports/PCTwig-TR.pdf`.
[18] B. YAO, M. TAMER ÖZSU, N. KHANDELWAL. XBench: Benchmark and Performance Testing of XML DBMSs. *In ICDE*, 2004.