Efficient Recursive XML Query Processing Using Relational Database Systems

Sandeep Prakash [a], Sourav S Bhowmick [a] and Sanjay Madria [b]

[a] *School of Computer Engineering, Division of Information Systems, Nanyang Technological University, Singapore 639798*

[b] *Department of Computer Science, University of Missouri-Rolla, Rolla 65409*

Abstract

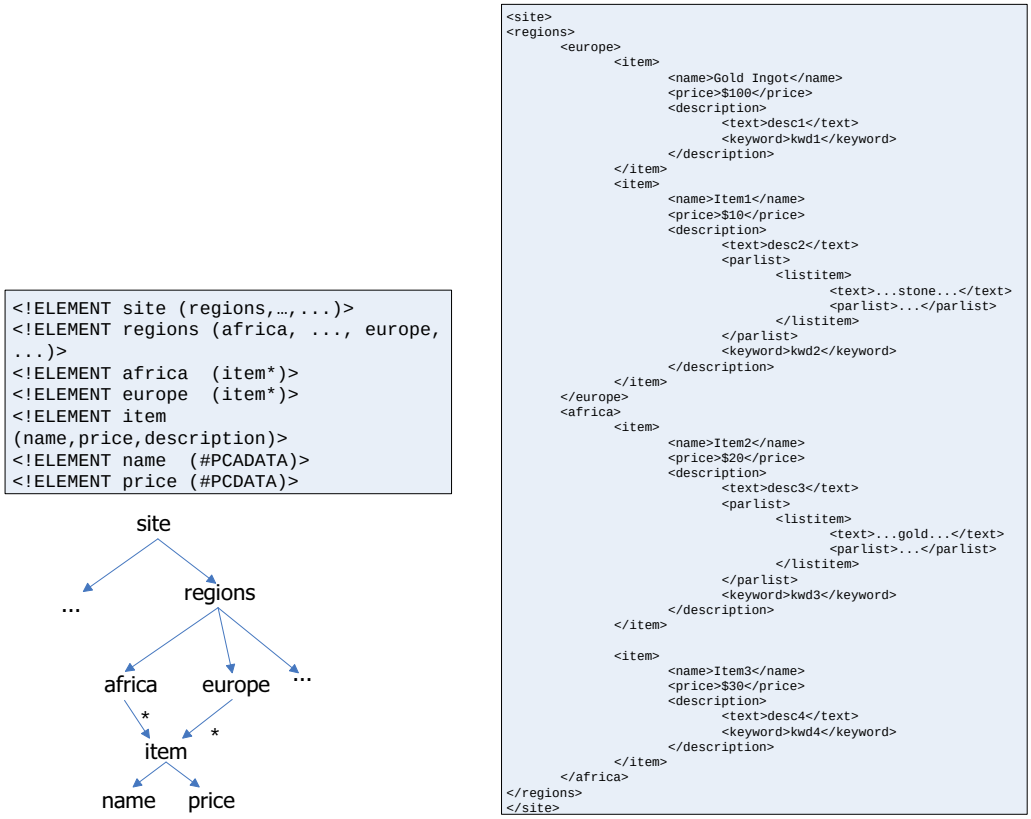
Recursive queries are quite important in the context of XML databases. In addition, several recent papers have investigated a relational approach to store XML data and there is growing evidence that schema-conscious approaches are a better option than schema-oblivious techniques as far as query performance is concerned. However, the issue of recursive XML queries for such approaches has not been dealt with satisfactorily. In this paper we argue that it is possible to design a schema-oblivious approach that outperforms schema-conscious approaches for certain types of recursive queries. To that end, we propose a novel schema-oblivious approach, called SUCXENT++ (**S**chema **U**n**c**oncious **X**ML **En**abled Sys**t**em), that outperforms existing schema-oblivious approaches such as XParent by up to 15 times and schema-conscious approaches (Shared-Inlining) by up to 8 times for recursive query execution. Our approach has up to 2 times smaller storage requirements compared to existing schema-oblivious approaches and 10% less than schema-conscious techniques. In addition SUCXENT++ performs marginally better than Shared Inlining and is 5.7 - 47 times faster than XParent as far as insertion time is concerned.

Key words: Recursive queries, XML storage, relational databases, query translation, query optimization, performance.

1 Introduction

XML is gradually becoming the standard in exchanging and representing data. Not surprisingly, effective and efficient querying of XML data has become an increasingly important issue. *Recursive XML queries* are considered to be quite significant in the context of XML query processing [9] and yet this issue has not been addressed satisfactorily in existing literature. Recursive XML

Email addresses: assourav@ntu.edu.sg, madrias@umr.edu (Sanjay Madria).



(a) Partial DTD. (b) XML document.

Fig. 1. An example.

queries are XML queries that contain the descendant axis (//). The use of the '//' is quite common in XML queries due to the semi-structured nature of XML data [9]. For example, consider the XML document in Figure 1 (for clarity, only partial structure is shown). The partial tree representation of the document is shown in Figure 2. The element `item` could occur either under `europe` or `africa`. Consider the scenario where a user needs to retrieve all `item` elements. The user will have to execute the path expression Q = /site//item. Another scenario could be that the document structure is not completely known to the user except that each `item` has a `name` and `price`. Suppose, the user needs to find out the price of the `item` with `name "Gold Ignot"`. Q = //item[name="Gold Ignot"]/price will be the corresponding path expression.

Efficient execution of XML queries, recursive or otherwise, is largely determined by the underlying storage approach. There has been a substantial research effort in storing and processing XML data. There are basically three alternatives for storing XML data: in semi-structured databases [6], in object-oriented databases [1], and in relational systems [2–5,7,14,15,18,19]. Among

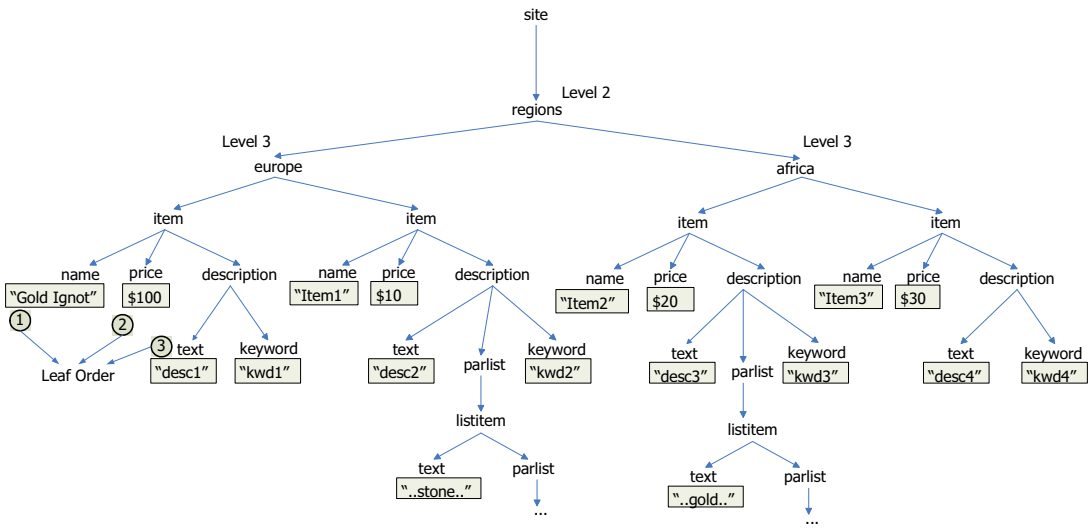


Fig. 2. Tree representation.

these approaches, the relational storage approach has attracted considerable interest with a view to leveraging their powerful and reliable data management services. In order to store an XML document in a relational database, the tree structure of the XML document must first be mapped into an equivalent, flat, relational schema. XML documents are then shredded and loaded into the mapped tables. Finally, at runtime, XML queries are translated into SQL, submitted to the RDBMS, and the results are then translated into XML.

There is a rich literature addressing the issue of managing XML documents in relational backends [2–5,7,14,15,18,19]. These approaches can be classified into two major categories as follows.

(1) *Schema-conscious approach:* This method first creates a relational schema based on the DTD/schema of the XML documents. First, the cardinality of the relationships between the nodes of the XML document is established. Based on this information a relational schema is created. The structural information of XML data is modeled by using primary-key foreign-key joins in relational databases to model the parent-child relationships in the XML tree. Examples of such approaches are Shared Inlining [14], LegoDB [2,11]. Note that this approach depends on the existence of a schema describing the XML data. Furthermore, due to the heterogeneity of XML data, in this approach a simple XML schema/DTD often produce a relational schema with many tables.
(2) *Schema-oblivious approach:* This method maintains a fixed schema which is used to store XML documents. The basic idea is to capture the tree structure of an XML document. This approach does not require existence of an XML schema/DTD. Also, number of tables is fixed in the relational schema and does not depend on the structural heterogeneity

of XML documents. Some examples of schema-oblivious approaches are Edge approach [5], XRel [18], XParent [7].

Schema-oblivious approaches have obvious advantages such as the ability to handle XML schema changes better as there is no need to change the relational schema and a uniform query translation approach. Schema-conscious approaches, on the other hand, have the advantage of more efficient query processing [16]. Also, no special relational schema needs to be designed for schema-conscious approaches as it can be generated on the fly based on the DTD of the XML document(s).

1.1 Overview

In this paper, we present an efficient approach to process recursive XML queries using a schema-oblivious approach [1]. At this point, one would question the justification of this work for two reasons. First, this issue may have already been addressed. Surprisingly, this is not the case as highlighted in [9]. Second, a growing body of work suggests that schema-conscious approaches perform better than schema-oblivious approaches. In fact, Tian et al. have demonstrated in [16] that schema-conscious approaches generally perform substantially better in terms of query processing and storage size. However, the Edge approach [5] was used as the representative schema-oblivious approach for comparison. Although the Edge approach is a pioneering relational approach, we argue that it is not a good representation of the schema-oblivious approach as far as query processing is concerned. In fact, XParent [7] and XRel [18] have been shown to outperform the Edge approach by up to 20 times, with XParent outperforming XRel [7]. However, this does not mean that XParent outperforms schema-conscious approaches. In fact, we shall show in Section 6, schema-conscious approaches still outperform XParent. Hence, it may seem that schema-conscious approaches generally outperforms schema-oblivious approaches in terms of query processing. In this paper we argue that it is indeed possible to design a schema-oblivious approach that can outperform schema-conscious approaches for certain types of *recursive queries*.

To justify our claim, we propose a novel schema-oblivious approach, called SUCXENT++ (**S**chema **Un**c**o**ncious **X**ML **En**abled Sys**t**em (pronounced "succinct++")), and investigate the performance of recursive XML queries. We only store the leaf nodes and the associated paths together with two additional attributes for efficient query processing (details follow in Section 3). SUCXENT++ outperforms existing schema-oblivious techniques, such as XParent, by up to 15 times and Shared-Inlining - a schema-conscious approach - by up to 8 times for recursive queries with characteristics described in Section 6.

[1] A shorter version of this paper has appeared in [10].

In addition, SUCXENT++ can reconstruct shredded documents up to 2 times faster than Shared-Inlining. The main reasons SUCXENT++ performs better than existing approaches are as follows.

- Significantly lower storage size and, consequently, lower I/O-cost associated with query processing;
- Fewer number of joins in the corresponding SQL queries; and
- Additional optimization techniques discussed in Section 5 improve the query plan generated by the relational query optimizer.

In summary, the main contributions of this paper are as follows.

(1) A novel schema-oblivious approach whose storage size depends only on the number of leaf nodes in the document.
(2) Through an extensive experimental study, we show that our approach significantly outperform state-of-the-art schema oblivious approaches and a schema-conscious approach for certain types of recursive queries. To the best of our knowledge, this is the first attempt to show that it is indeed possible to design a schema-oblivious approach that can outperform schema-conscious approaches as far as the execution of *certain types* of recursive XML queries is concerned.
(3) Traditional schema-oblivious approaches have been hampered by the poor query plan selection of the underlying relational query optimizer[16,19]. We developed optimization techniques to improve the query plan generated by the relational query optimizer. Our experimental results demonstrate the effectiveness and efficiency of our optimization techniques.

The rest of the paper is organized as follows. Section 2 briefly discusses existing techniques to store and query XML in an RDBMS. The database schema of SUCXENT++ is presented in Section 3. We will briefly describe how an XML document is stored in an RDBMS using SUCXENT++. In Section 4, we present how recursive XML queries are translated to SQL in SUCXENT++. This is followed by a discussion of optimization techniques to improve query performance in Section 5. Section 6 presents the performance results of SUCXENT++ and compare it with a schema-oblivious (XParent) and one schema conscious (Shared-Inlining) approaches. The last section concludes the paper.

2 Related Work

There is a substantial body of work on using relational databases to store XML documents. The various approaches differ in which meta-data they use (i.e., schema or schemaless); how the relational configuration is generated; and which information is preserved in the relational side. Table 1 gives the summary of existing approaches.

Techniques	Schema-oblivious	Cost-based	Order Preserving	Class of XML schema considered	Class of XML Query considered
STORED [4]	Yes	No	Yes	All	STORED
Edge [5]	Yes	No	Yes	All	Path expressions
Interval [3]	Yes	No	Yes	All	XQuery
XRel [18]	Yes	No	Yes	All	Path expressions
XParent [7]	Yes	No	Yes	All	Path expressions
[15]	Yes	No	Yes	All	Order-based queries
[19]	Yes	No	Yes	ll	Path expressions
SUCXENT++	Yes	No	Yes	All	XQuery
Inlining [14]	No	No	No	Recursive	-
LegoDB [2]	No	Yes	No	tree	XQuery
Oracle XML DB [21]	Yes	No	Yes	Recursive	SQL/XML restricted XPath
DB2 XML Extender [20]	Yes	No	Yes	non-recursive	SQL extensions through UDFs

Table 1
Summary of XML storage and querying techniques.

2.1 Schema-oblivious Approaches

Techniques which store XML documents in *generic* (pre-defined) relational tables are called schema-oblivious. One of the first proposals for schema-oblivious mapping of XML documents was the Edge approach [5]. In this approach, the input XML document is viewed as a graph and each edge of the graph is represented as a tuple in a single table. In a variant known as the Attribute approach, the edge table is horizontally partitioned on the tag name yielding separate table for each element/attribute. Two other alternatives, the Universal table approach and the Normalized Universal approach are proposed but shown to be inferior to the other two. In this approach, resolving ancestor-descendant relationships requires the traversal of all the edges from the ancestor to the descendant (or vice-versa). Thus it is an expensive approach as it typically requires many joins for navigating and/or reconstructing the document. Note that the Edge approach uses *recursive SQL queries* using the SQL99 *With* construct to evaluate recursive XML queries.

In STORED [4], given a semistructured database instance, STORED mapping is generated automatically using data mining techniques - STORED is a declarative query language proposed for this purpose. This mapping has two parts: a relational schema and an *overflow graph* for the data not conforming to the relational schema. STORED can be considered as schema-oblivious approach as the data inserted in the future is not required to conform to the derived schema. Thus, if an XML document with completely different structure is added to the database, the system sticks to the existing relational

Document (DocId, Name)
Path (PathId, PathExp, *CPathId*)
PathValue (DocId, PathId, LeafOrder, *CPathId*, BranchOrder, BranchOrderSum, LeafValue)
TextContent (DocId, PathId, LeafOrder, *CPathId*, BranchOrder, BranchOrderSum, Text)
DocumentRValue (DocId, Level, RValue)

(a) SUCXENT++.

LabelPath (ID, Len, Path)
DataPath (Pid, Cid)
Element (PathID, Did, Ordinal)
Data (PathID, Did, Ordinal, Value)
Ancestor (Did, Ancestor, Level)

(b) XParent.

Fig. 3. Relational schemas.

schema without any modification whatsoever. In STORED, an algorithm is outlined for translating an input STORED query into SQL. The algorithm uses inversion rules to create a single canonical data instance, intuitively corresponding to a schema. The structure component of the STORED query is then evaluated on this instance to obtain a set of results, for each of which a SQL query is generated incorporating the rest of the STORED query. However, similar to the Edge, in this approach it is necessary to perform one join per step in the path expression during query translation.

The system proposed by Zhang *et al* in [19] labels each node with its preorder and postorder traversal numbers. Then, ancestor-descendant relationships can be resolved in constant time using the property $preorder(ancestor) < preorder(descendant)$ and $postorder(ancestor) > postorder(descendant)$. However, it still results in as many joins as there are path separators.

To solve the problem of multiple joins, XRel [18] stores the path of each node in the document. For each element, the path id corresponding to the root-to-leaf path as well as an interval representing the region covered by the element are stored. Then, the resolution of path expressions only requires the paths (which can be represented as strings) to be matched using string matching operators. However, the query translation algorithm in XRel is correct for nonrecursive data sets - it turns out that it does not give the correct result when the input XML data has an ancestor and descendant element with the same tag name [8]. Moreover, this approach still makes use of the containment property mentioned above to resolve ancestor-descendant relationships. It involve joins with θ ($<$ *or* $>$) operators that have been shown to be quite expensive due to the manner in which an RDBMS processes joins [19]. In fact, special algorithms such as the Multi-predicate merge sort join algorithm [19] have been proposed to optimize these operations. However, to the best of our knowledge there is no off-the-shelf RDBMS that implements these algorithms as the issue of how we extend the relational engine to identify the use of these strategies is an open problem. In particular, the question of how the optimizer maps SQL operations into these strategies needs to be addressed.

XParent [7] solves the problem of θ-joins by using an **Ancestor** table that stores all the ancestors of a particular node in a single table. It then replaces θ-joins with *equi*-joins over this set of ancestors. However, this approach results in an explosion in the database size as compared to the original document. The number of relational joins is also quite substantial. XParent requires a join between the **LabelPath**, **DataPath**, **Element** and **Ancestor** tables for each path in the query expression. The joins are quite expensive especially when the **Ancestor** table is involved as it can be quite large in size. Note that XParent and XRel handle recursive queries like any other query.

In [15], the focus is on supporting order based queries over XML data. The schema assumed is a modified Edge relation where the path id is stored as in XRel, and an extra field for order is also stored. Three schemes for supporting order are discussed. Algorithms for translating order-based path expression queries into SQL are also provided. As this approach is based on the Edge and XRel, it suffers from the same limitations as discussed above.

In dynamic intervals approach [3], all XML data is stored in a single table containing a tuple for each element, attribute and text node. For an element, the element name and an interval representing the region covered by the element is stored. Analogous information is stored for attributes and text nodes. In order to distinguish children from descendants, a level number is recorded with each node. This approach supports a larger fragment of XQuery with arbitrarily nested FLWR expressions, element constructors and built-in functions including structural comparisons. Special purpose relational operators are proposed for better performance. We note that without these operators, the performance is likely to be inferior even for simple path expressions. As an example, using their technique, the path expression `/site/people` is translated to an SQL query involving five temporary relations created using the *With* clause in SQL99, three of which involve correlated subqueries [8]. Hence, without modifications to the relational engine, its performance may not be acceptable.

In Oracle XML DB [21] and IBM DB2 XML Extender [20], a schema oblivious way of storing XML data is provided, where the entire XML document is stored using the CLOB data type. Hence, evaluating XML queries in this case is similar to XML query processing in a native XML database. Also, many types of XML queries suffer from poor performance due to the treatment of XML documents as CLOB.

SUCXENT++ is different from existing approaches in that it only stores leaf nodes and their associated paths. For each level in an XML document, we store an attribute called *RValue*. Rather than storing the ancestor-descendant and parent-child of all nodes in the XML document, we store only the leaf nodes and their corresponding values along with the root-to-leaf paths in the

document. Additionally, for each leaf node we store two additional attributes called *BranchOrder* and *BranchOrderSum*. These attributes along with the *RValue* enable us to efficiently check whether the level of the nearest common ancestor of a pair of relevant leaf nodes satisfies the query constraints. This reduces the storage size significantly as well as the number of joins needed to be executed in the translated SQL queries. In addition, we propose optimization techniques that enable the underlying relational query optimizer to generate near-optimal query plans for our approach, resulting in a substantial performance improvement.

2.2 Schema-conscious Approaches

Departing from generic mapping as discussed above, several specialized strategies have been proposed which make use of schema information to generate efficient mappings. These approaches are called schema-conscious approaches.

In [14], three techniques for using a DTD to choose a relational schema are proposed - basic inlining, shared inlining, and hybrid inlining. The main idea is to inline all elements that occur at most once per parent element in the parent relation itself. This is extended to handle recursive DTDs.

LegoDB [2,11] takes a cost-based approach to derive a mapping that best suits a given application - characterized by a schema, query workload and document samples. LegoDB uses the information in the XML schema to derive several possible mapping alternatives, and selects the one which leads to the lowest cost for executing a given query workload over sample documents. Compared to Shared Inlining, LegoDB system exploits a richer set of mapping primitives. In addition to parent-child relationships, LegoDB also takes into account additional schema constructs such as choice and repetition, and it allows multiple mapping functions for a given construct.

Unlike the schema-oblivious approaches, schema-conscious techniques have focused on structural and constraint mapping, often ignoring the order among elements. Because these techniques ignore order, the resulting mapping are lossy [8]. For example, the mapping strategies in [14] do not allow mapped documents to be faithfully reconstructed. Furthermore, schema-conscious strategies have to treat recursion in both schema and queries as special cases. In [9], the authors propose a generic algorithm to translate recursive XML queries for schema-conscious approaches using the SQL99 *With* construct. However, no performance evaluation of the resulting SQL queries is presented and it is assumed that schema-conscious approaches will outperform schema-oblivious approaches. SUCXENT++ maintains document order and also treats recursive XML queries like any other queries.

Document (DocId, Name)
Path (PathId, PathExp, *CPathId*)
PathValue (DocId, PathId, LeafOrder, *CPathId*, BranchOrder, BranchOrderSum, LeafValue)
TextContent (DocId, PathId, LeafOrder, *CPathId*, BranchOrder, BranchOrderSum, Text)
DocumentRValue (DocId, Level, RValue)

(a) SUCXENT++.

LabelPath (ID, Len, Path)
DataPath (Pid, Cid)
Element (PathID, Did, Ordinal)
Data (PathID, Did, Ordinal, Value)
Ancestor (Did, Ancestor, Level)

(b) XParent.

Fig. 4. Relational schemas.

3 Storing XML Data

In this section we present the database schema for storing XML documents. Then, we present the algorithm for lossless extraction of XML documents from the relational database. We use the sample XML document in Figure 1 as the running example in this section.

3.1 Schema Description

We first define some symbols to facilitate exposition. Let n_k be a node in the XML tree X. Then the level of n_k is denoted as ℓ_k. We denote the maximum level of the X as L_{max}. Also, A_k denotes the set of ancestor nodes of n_k. The relational schema for SUCXENT++ is shown in Figure 4(a). There are five relations in the schema. We elaborate on the detail semantics of these relations. The semantics of **DocumentRValue** relation and *BranchOrderSum* attribute in **PathValue** table are going to be elaborated in Section 4 in the context of query processing. The *CPathId* in **Path** is discussed in Section 5 as it is used for optimization.

The Document Relation

The table **Document** is used for storing the names of the documents in the database. This name could be the file name of the XML document or its URL. Whenever a new document is inserted in SUCXENT++ its file name or URL is stored in the *Name* attribute and a unique identifier is stored in the *DocId* attribute. This identifier is used as a reference to this document in the rest of the schema. Figure 5 shows the shredded version of the example document.

Document

DocId	Name
1	Auction.xml
2	..

Path

PathId	PathExp	CPathId
1	site.regions.africa.item.description.keyword	1
2	site.regions.africa.item.description.text	3
3	site.regions.africa.item.name	5
4	site.regions.africa.item.price	7
5	site.regions.europe.item.description.keyword	2
6	site.regions.europe.item.description.text	4
7	site.regions.europe.item.name	6
8	site.regions.europe.item.price	8

PathValue

DocId	PathId	CPathId	LeafOrde	BranchOrder	BranchOrderSum	LeafValue
1	7	6	1	0	0	Gold Ignot
1	8	8	2	4	3	$100
1	6	4	3	4	6	gold
1	5	2	4	5	8	kwd1
1	...	...	...	...	...	...
1	3	5	13	3	85	Item3
1	4	7	14	4	88	$30
1	2	3	15	4	91	desc4
1	1	1	16	5	93	kwd4

DocumentRValue

DocId	Level	RValue
1	1	329
1	2	41
1	3	10
1	4	3
1	5	2
1	6	1

Fig. 5. XML data in RDBMS.

The Path Relation

The **Path** table records every unique root-to-leaf path encountered in the XML documents stored in the database. The path expression corresponding to the path is stored in the *PathExp* attribute. A unique identifier (integer type) to reference this path is stored in the *PathId* attribute. The rest of the schema uses this *PathId* to refer to this path. This table maintains path identifiers and relative path expressions recorded as instances of *PathId* and *PathExp* respectively.

The PathValue Relation

This table stores the leaf nodes of the XML documents stored in the database. Each tuple stores one leaf node. The *DocId* attribute in a tuple refers to the document the leaf node in this tuple belongs to. This refers to the *DocId* unique identifier of the **Document** table. The shredded document in Figure 5 has *DocId* equal to `1` and the leaf nodes corresponding to this document are stored in the **PathValue** table with this *DocId* value. The *PathId* attribute refers to the root-to-leaf path corresponding to this leaf node as stored in the **Path** table. The *LeafValue* attribute stores the textual content of the leaf node. Consider the first tuple in the **PathValue** table. This stores the first leaf node, *name*, of the document. As shown in Figure 2 this leaf node has the root-to-leaf path `site/regions/europe/item/name` corresponding to *PathId* equal to `7` in the **Path** table. The text value of this node is `"Gold Ingot"` and this is stored as an instance of *LeafValue* attribute.

The *LeafOrder* attribute (denoted as $leaforder(n)$) records the *order* of the leaf node n. This corresponds to the order in which leaf nodes are encountered when the document is parsed in preorder traversal. For example, the first leaf

node encountered in the document of Figure 2 is *name*. The first tuple in the **PathValue** table of Figure 5 represent this node and has a *LeafOrder* value equal to `1`. Similarly, *price* is the next leaf node encountered and, therefore, its *LeafOrder* is `2`. The text value of this node is `"$100"`. This node is shown in the second tuple in the **PathValue** table. Formally,

Definition 1 [**LeafOrder**] *Let* $P = \{n_1, n_2, \ldots, n_k\}$ *be the set of leaf nodes in* X *such that* $preorder(n_i) < preorder(n_{i+1}) \; \forall \; 0 < i < k$ *Then* $leaforder(n_1) = 1$ *and* $leaforder(n_j) = leaforder(n_{j-1}) + 1 \; \forall \; 1 < j \leq k$. ∎

Observe that the **PathValue** provides a list of leaf nodes together with their paths (as referenced by *PathId*). The tree structure shown in Figure 2 can be obtained by "stitching" together these leaf nodes. However, in order to do this, the level at which a leaf node path *intersects* the adjacent leaf node paths (nearest common ancestor of the two adjacent nodes) must be known. Consider the node with *LeafOrder*=`2` in Figure 2. The path of this leaf node will intersect with path of the leaf node with *LeafOrder* equal to `1` at the node *item* which is at level `4`. By recording this information the portion of the document corresponding to these two leaf nodes can be reconstructed. Observe that only the intersection level with the path of either of the two adjacent leaf nodes needs to be recorded. Here, the intersection level with the nodes on the left hand side is recorded. This is because standard XML parsers follow preorder traversal and hence recording the intersection level with the nodes on the right hand side would require backtracking.

The *BranchOrder* attribute (denoted as $branchorder(n)$) records the intersection level (nearest common ancestor) of the leaf node n with leaf node that immediately precedes it. For instance, the *BranchOrder* value for the node with *LeafOrder*=`2` in Figure 2 is `4`. Similarly, the node *name* with value `"Item2"` has *BranchOrder*=`2` (intersecting the node to the left at *regions*). It is useful for reconstructing the XML documents from their shredded relational format as discussed in the next section. Formally,

Definition 2 [**BranchOrder**] *Let* n_i *and* n_j *be two leaf nodes in the XML tree* X *such that* $leaforder(n_i) = leaforder(n_j) - 1$. *Let* n_a *be an internal node of* X *at level* ℓ_a *such that* $n_a \in A_i$, $n_a \in A_j$ *and* $\nexists \; n_k$ *s.t.* $\ell_k > \ell_a$, $n_k \in A_i$ *and* $n_k \in A_j$ *for* $k \neq a$. *Then (1)* $branchorder(n_1) = 0$ *if* $leaforder(n_j) = 1$ *and (2)* $branchorder(n_j) = \ell_a$ *if* $leaforder(n_j) > 1$. ∎

The TextContent Relation

Large text data (e.g., DNA sequences) can cause problems while indexing the corresponding column. So, they are not stored in the **PathValue** table. Instead, a separate table, **TextContent** that has the same structure as **PathValue** table, is maintained for large text data. In SUCXENT++, all textual data larger than 255 bytes in size is stored in this table.

```
Input: L = {n_1, ..., n_k}, a list of leaf nodes arranged in order of LeafOrder values
Output: D is the document to be returned.
 1: c is an XML node.
 2: c ← φ
 3: C ← list of XML nodes.
 4: for all n_i in L do
 5:    /* /book/authors/author would give p = [book,authors,author]*/
 6:    p is the array of nodes in a path.
 7:    p = n_i.Path.GetNodes()
 8:    /*s is a counter*/
 9:    s ← 0
10:    if c = φ then
11:       c ← new XmlDocumentNode( p[0] )
12:       /* Make c the root. This happens only once. */
13:       D.AddNode( c )
14:       C.Add( c )
15:       s ← 1
16:    else if
       then
17:       s ← n_i.BranchOrder()
18:    end if
19:    /* Keep only those nodes in C that are common between n_{i-1} and n_i. */
20:    C.ClearFromIndex( s )
21:    q is an XML node
22:    /*Need to keep it as the starting node for processing n_{i+1} */
23:    q ← c
24:    while s < p.Length() do
25:       m ← new XmlDocumentNode(p[s])
26:       q.AppendChild( m )
27:       C.Add( m)
28:       q ← m
29:       s++
30:    end while
31: end for
```

Fig. 6. Extraction algorithm.

3.2 Document Extraction Algorithm

The algorithm for reconstruction is presented in Figure 6. The input to the algorithm is a list of leaf nodes arranged in ascending *LeafOrder*. This list could be obtained as a result of a query or could simply be the whole document. The reconstruction proceeds as follows.

(1) Each leaf node path is first split into its constituent nodes (lines 5 to 7). This process can be optimized by already storing the constituent nodes of each root-to-leaf path of the **Path** table.
(2) If the document construction has not yet started (line 10) then the first node obtained by splitting the first leaf node path is made the root (lines 11 to 15).
(3) When the next leaf node and the corresponding root-to-leaf path is analyzed we only need to process the nodes starting after the *BranchOrder* level of that path as the nodes till this level have already been added to the document (lines 20 to 22). The nodes starting after this level are now added to the document (lines 27 to 31).
(4) Document extraction is completed once all the leaf nodes have been processed.

This algorithm can be used to construct the whole document if all leaf nodes in the document are presented as input. A fragment of the document can be generated if a partial list of *consecutive* leaf nodes is provided. In fact, query results are returned as a list of leaf nodes and the result XML fragment(s) is constructed using this algorithm. As an example, consider reconstructing the document fragment corresponding to the first two *LeafOrders* in Figure 2. Essentially, we start with the first two tuples (in terms of *LeafOrder*) in the **PathValue** table in Figure 5. The reconstruction proceeds as follows.

(1) The first leaf node has the path `site.regions.europe.item.name`. It is split up into the nodes *site, regions, europe, item,* and *name* (line 7). Since this is the first leaf node (line 10), all the nodes in the path are added to the document and the value of the *name* node is set to `Gold Ingot` (lines 23 to 30).
(2) The next leaf node has the path `site.regions.europe.item.price`. Now, only those nodes that start after the *BranchOrder* level of this path need to be considered (line 17). Since, its *BranchOrder* is 4, nodes are processed starting from *price* (lines 23 to 30). So, this node is added to the document and its value is set to `$100`.
(3) The other nodes are processed similarly to produce the partial or complete document.

4 Recursive Query Processing

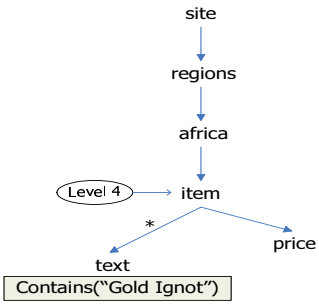
In this section, we first analyze the approach taken by existing schema-oblivious schemes and highlight their drawbacks. Then, a discussion of the SUCXENT++ approach to query processing, including translation of XQuery to SQL, will follow.

4.1 Current Schema-oblivious Approaches

Consider the recursive query *XQuery 1* in Figure 7(a). A tree representation of the query is shown in Figure 7(b). This query returns those *price* leaf nodes that intersect the constraint-satisfying *text* leaf node at *item*. Consider how XParent resolves this query. The schema for XParent is shown in Figure 4(b). XParent evaluates this query by locating leaf nodes from the **Data** table that satisfy the constraint on *text*. This involves a join between the **LabelPath** and **Data** to satisfy the path constraint `/site/regions/africa/item//text` and a predicate on the **Data** to satisfy the value constraint. Next, **LabelPath** and **Data** tables are joined again to obtain those leaf nodes that satisfy `/site/regions/africa/item/price`. These two results sets are joined using the **Ancestor** table to find nodes that have a common ancestor at level `4` (at

```
For                                                    XQuery 1
  $b in document( "auction" )/site/regions/africa/item
Where
  contains($b//text, "Gold Ignot")
Return
   <price>$b/price</price>
```

(a) XQuery example.



(b) Query tree.

Fig. 7. XQuery.

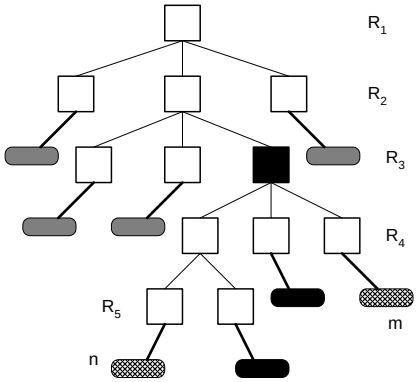
`item`). Thus, the final SQL query involves five joins - two between the **Label-Path** and **Data**, two between the **Data** and **Ancestor** and one between two **Ancestor** tables (SQL query translation details for XParent can be found in [7]). These joins can be quite expensive due to the large size of **Ancestor**. XRel follows a similar approach to resolving path expressions except that it uses the *ancestor-descendant* containment property instead of an **Ancestor** table. This produces θ-joins resulting in performance worse than XParent. A detailed evaluation of XRel vs. XParent can be found in [7].

Query performance can be improved by reducing the number of joins. The joins should be executed on smaller data to further improve performance.

4.2 The SUCXENT++ *Approach*

Reconsider the query in Figure 7. It is evident that XML fragments that satisfy the query must satisfy the following *structural* constraints: the XML document must have paths that satisfy the path expressions `/site/regions/africa/item//text` and `/site/regions/africa/item/price` and these paths must *intersect* (nearest common ancestor) at a level 4. In general, given an XQuery, often we need to determine if a pair of nodes n_1 and n_2 intersect at a specific level ℓ of the XML tree. For instance, consider the XML tree in Figure 8(a). Suppose that the root-to-leaf paths to nodes n and m satisfy the constraints of a query. Then we need to identify the intersection level or level of the nearest common ancestor of these two nodes (level 3) efficiently in order to identify related query results.

The attributes discussed till now in SUCXENT++ are insufficient for identifying such intersection level efficiently. Recall that the *BranchOrder* attribute records the intersection level of a leaf node n with leaf node that *immediately precedes* it. However, the nodes n and m may not be adjacent to one another. Hence, we need to extend the SUCXENT++ schema so that we can determine



(a) *BranchOrderSum* and *RValue*

```
1  SELECT
2     V2.*
3  FROM
4     PATHVALUE V1,
5     PATH P1,
6     PATHVALUE V2,
7     PATH P2,
8     DOCUMENTRVALUE D
9  WHERE
10    P1.PATHEXP LIKE '.site.regions.africa.item.%.text' AND
11    P1.ID = V1.PATHID AND
12    V1.LEAFVALUE = '%Gold Ignot%' AND
13    P2.PATHEXP LIKE '.site.regions.africa.item.price' AND
14    P2.ID = V2.PATHID AND
15    V1.DOCID = V2.DOCID AND
16    V1.DOCID = D.DOCID  AND
17    D.LEVEL = 3 AND
18    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

(b) Translated SQL query.

Fig. 8. Query processing in SUCXENT++.

the intersection level by inspecting the nodes n and m only without processing all the leaf nodes and root-to-leaf paths between n and m (Black leaf nodes in Figure 8(a)). We achieve this by extended the schema in the following ways. First, we store an attribute *RValue* (denoted as R_ℓ) in the **DocumentRValue** table for each level ℓ of the XML tree. Second, an attribute *BranchOrderSum*, denoted as S_n, is assigned to a leaf node with *LeafOrder* n. As we shall later, these attributes allow us to determine the intersection levels of any two leaf nodes efficiently. This results in a substantial reduction in storage size and query processing time. We now elaborate on these attributes. We begin by first defining the notion of *maximum k-consecutive leaf node set* which shall be used to define *RValue*.

Definition 3 [Consecutive Leaf Node Set] *Let* $C = \{n_1, n_2, \ldots, n_r\}$ *be a set of leaf nodes in* X *such that* $leaforder(n_j) = leaforder(n_{j+1}) - 1 \ \forall \ 1 \leq j < r$. *Then* C *is called the* ***consecutive leaf node set.*** ∎

Let us illustrate the above definition with an example. Consider Figure 2. The nodes labeled "$100", "desc1", and "kwd1" consist of a consecutive leaf node set as they have leaf order values 2, 3, and 4 respectively That is, $C_1 = \{$"$100", "$desc$", "$kwd1$"\}$. Similarly, $C_2 = \{$"$desc$", "$kwd1$", "$item1$", "10"\}$.

Definition 4 [k-Consecutive Leaf Node Set] *Let* $C = \{n_1, n_2, \ldots, n_r\}$ *be a consecutive leaf node set. Then* C *is called* ***k-consecutive leaf node set****, denoted as* C_k*, if the following conditions are true.*

- $branchorder(n_i) \geq k \ \forall \ 1 \leq i \leq r$ *and* $k \in [1, L_{max}]$.
- *If* $leaforder(n_1) > 1$ *then* $branchorder(n_\ell) < k$ *and* $leaforder(n_\ell) = leaforder(n_1) - 1$.
- *If* n_r *is not the rightmost leaf node in* X *then* $branchorder(n_{r+1}) < k$ *and* $leaforder(n_{r+1}) = leaforder(n_r) + 1$.

The number of nodes in C_k is denoted as $|C_k|$. ∎

For instance, consider C_1 in the above example. Let $k = 4$. Then, C_1 is a k-consecutive leaf node set (also denoted as C_4) as the *BranchOrder* values of the nodes "$100", "desc1", and "kwd1" are 4, 4, and 5 respectively, and the *BranchOrder* values of "Gold Ignot" (*LeafOrder* is 1) and "item1" (*LeafOrder* is 5) are 0 and 3 respectively. However, C_2 is not a k-consecutive leaf node set as it has nodes having *BranchOrder* value less than 4. Note that $C_1' = \{$"*desc*", "*kwd*1"$\}$ is not k-consecutive leaf node set as "$100" (*LeafOrder* is 2) has *BranchOrder* value equal to 4. Observe that $|C_4| = 3$.

Definition 5 [Maximum k-Consecutive Leaf Node Set] *Let C_k^1, C_k^2, C_k^3 ... C_k^m be the set of k-consecutive leaf node sets for a given k. Then C_k^j is* ***maximum k-consecutive leaf node set****, denoted as M_k, if $|C_k^j| \geq |C_k^i|$ $\forall$ $0 < i \leq m$ and $i \neq j$. The number of nodes in M_k is denoted as $|M_k|$.* ∎

Definition 6 [RValue] *Let L_{max} be the maximum level of an XML tree. Then* ***RValue*** *of level ℓ_k for $0 < k \leq L_{max}$, denoted as R_k, is defined as follows.*

(1) If $\ell_k = L_{max}$ then $R_k = 1$ and $|M_k| = 1$.
(2) If $\ell_k < L_{max}$ then $R_k = R_{k+1} \times |M_b| + 1$ where $b = k + 1$. ∎

Reconsider the XML tree in Figure 2. For simplicity, ignore the subtrees rooted at *parlist* node. Then $L_{max} = 6$. Therefore, from the above definition, $R_6 = 1$ and $|M_6| = 1$. This means that $R_5 = 1 \times 1 + 1 = 2$. The maximum number of *consecutive* leaf nodes with *BranchOrder* ≥ 5 (maximum 5-consecutive leaf node set) is 1 (Node "kwd1" with *LeafOrder* equal to 4). Therefore, $R_4 = 2 \times 1 + 1 = 3$. Similarly, $|M_4| = 3$ (e.g., *price*, *text*, *keyword* under the first *item* element). So, $R_3 = 3 \times 3 + 1 = 10$. Observe that as the level of the XML tree decreases the value of *RValue* increases. Formally,

Property 1 *Let ℓ_i and ℓ_j be two levels in the XML tree X where $0 < (i, j) \leq L_{max}$. If $\ell_i > \ell_j$ then $R_i < R_j$.* ∎

Next, we define the notion of *BranchOrderSum*.

Definition 7 [BranchOrderSum] *Let n be a leaf node of X. Let $b_n = branchorder(n)$. Then the* ***BranchOrderSum*** *of n, denoted as S_n, is defined as follows:*

- *If $leaforder(n) = 1$ then $S_1 = 0$.*
- *Otherwise, $S_n = \sum_{i=1}^{i \leq n} R_{b_i}$.* ∎

For example, *BranchOrderSum* of the first leaf node in Figure 1 is 0. Since *BranchOrder* of the second leaf node is 4 and $R_4 = 3$, the *BranchOrderSum* of the second leaf node is 3 ($S_2 = R_{b_1} + R_{b_2} = R_0 + R_4 = 3$).

We now discuss how the *RValue* and *BranchOrderSum* enables efficient query processing. We first introduce the following theorem which is key to efficient query processing in SUCXENT++.

Theorem 1 *Let n_i and n_j be two leaf nodes in the XML tree X. Let $i = leaforder(n_i)$ and $j = leaforder(n_j)$. If $|S_i - S_j| < R_\ell$ then the nearest common ancestor of n_i and n_j is at a level greater than ℓ .*

Proof 1 Let n_i and n_j be two leaf nodes in the XML tree X and $leaforder(n_i) < leaforder(n_j)$. Let the nearest common ancestor of n_i and n_j be at level ℓ' in X. We would like to prove that if $|S_i - S_j| < R_\ell$ then $\ell' > \ell$.

The *BranchOrderSums* of n_i and n_j are as follows:

$$S_i = R_{b_1} + R_{b_2} + \ldots + R_{b_i}$$
$$S_j = R_{b_1} + R_{b_2} + \ldots + R_{b_j}$$

Then, $|S_i - S_j| = |R_{b_{i+1}} + R_{b_{i+2}} + \ldots + R_{b_{j-1}} + R_{b_j}|$ where b_{i+1}, b_{i+2}, ..., b_{j-1} are the *BranchOrder* values of nodes between n_i and n_j, i.e., n_{i+1}, n_{i+2}, ..., n_{j-1}. Also, $leaforder(n_i) < leaforder(n_{i+1}) < leaforder(n_{i+2}) < \ldots < leaforder(n_{j-1}) < leaforder(n_j)$. The maximum value of $|S_i - S_j|$ depends on the *BranchOrder* values of the nodes between n_i and n_j (inclusive). Based on Property 1, the value of R_{b_k} increases as b_k decreases. Hence, we need to find the minimum possible *BranchOrder* (level) of the nodes between n_i and n_j.

As $leaforder(n_{i+1}) > leaforder(n_i)$, the nearest common ancestor of n_i and n_{i+1} (*BranchOrder*) cannot be at a level less than ℓ'. Therefore, *BranchOrder* of n_{i+1} is $\ell_+ \geq \ell'$. Similarly, it can be shown that *BranchOrder* values of n_{i+2}, n_{i+3}, ..., b_{j-1}, n_j are greater than or equal to ℓ'. Therefore, minimum value of *BranchOrder* for a node n_r is ℓ' where $i < r \leq j$. Hence,

$$\begin{aligned} |S_i - S_j|_{max} &= (|R_{b_{i+1}} + R_{b_{i+2}} + \ldots + R_{b_{j-1}} + R_{b_j}|)_{max} \\ &= |R_{\ell'} + R_{\ell'} + \ldots + R_{\ell'} + R_{\ell'}| \\ &= |kR_{\ell'}| \text{ where } k = i - j \end{aligned}$$

Therefore, we can say $|kR_{\ell'}| < R_\ell$. Now, if $\ell' = \ell$ then this statement cannot be true. Also, if $\ell' < \ell$ then it is also not true as $R_{\ell'} > R_\ell$ (Based on Property 1). Therefore, $\ell' > \ell$. ∎

The attributes *RValue* and *BranchOrderSum* allow the determination of the intersection level between any two leaf nodes efficiently. Let us elaborate on this further. Suppose a query Q is evaluated on the XML tree in Figure 8(a). We wish to determine if the level of the nearest common ancestor (node A) of the nodes n and m in Figure 8(a) satisfies the structural constraints of Q. Note that the level of the intersecting nodes (denoted as ℓ_a) can be computed

in SUCXENT++ in two ways. If there is no descendant axis preceding the intersecting node in the path expressions of Q then the exact value of ℓ_a can be computed from Q. Otherwise, the minimum value of ℓ_a can be computed from the path expressions in the **Path** table. Next, based on the above theorem, we compute the difference between the *BranchOrderSums* of n and m ($|S_n - S_m|$) and compare it with the $RValue$ of the level $\ell_a - 1$. For instance, in Figure 8(a) the nearest common ancestor of n and m is in level 3 and consequently $|S_n - S_m| < R_2$. Therefore, the subtree rooted at A satisfies the query constraints if ℓ_a is computed to be at least 3. Let us illustrate this with an example.

Consider *XQuery1* in Figure 7. The *BranchOrderSum* value for the first constraint satisfying *text* is 6. The *BranchOrderSum* value for the first *price* node is 3. Also, $R_3 = 10$. Using the theorem proven above we conclude that these two nodes have ancestors till a level greater than or equal to 3 (since $|3-6| < 10$). Since, *item* is at level 4 in both cases it is clear that they have a common *item* node and, therefore, satisfy the query. Similarly, we can conclude that the first *text* node and the *item* node with *name* `Item3` intersect at a level greater than 1 (since $R_1 = 329$ and $|85 - 3| < 329$) and therefore do not form a part of the query result.

Note that in XParent the determination of intersection level depends on the size of the **Ancestor** and **Data** tables as a join between these tables is required to determine the ancestor node at a particular level. This reduces the query processing time drastically in SUCXENT++. Since this is achieved without storing separate ancestor information, the storage requirements are also reduced significantly.

4.3 SQL Translation

In this section, we discuss how XQuery queries are translated to corresponding SQL queries in SUCXENT++. A full implementation of the XML query processing system would require a fully-functional XQuery support. However, building a system like this would take a significant number of person-years to implement. Instead, we implemented an interface that supports basic types of recursive XQuery queries which do not include aggregate functions and ordering of result elements. These queries are sufficient to justify the positive contributions made by our approach.

The main structure of an XQuery query can be formulated by an `FLWOR` expression with the help of XPath expressions. An `FLWOR` expression is constructed from `FOR`, `LET`, `WHERE`, `ORDER BY`, and `RETURN` clauses. `FOR` and `LET` clauses serve to bind values to one or more variables using path expressions. The `FOR` clause is used for iteration, resulting in a single binding for each variable. As the `LET` clause is usually used to process grouping and aggregate functions, the

processing of the `LET` clause is not discussed here. The optional `WHERE` clause specifies one or more conditions to restrict the tuples generated by `FOR` and `LET` clauses. The `RETURN` clause is used to specify an element structure and to construct the result elements in the specified structure. The optional `ORDER BY` clause determines the order of the result elements. We ignore the `ORDER BY` clause in this paper.

A basic recursive XQuery query can be formulated with a simplified `FLWOR` expression:

```
  FOR   x1 in p1, ..., xn in pn
WHERE   c
RETURN  s
```

In the `FOR` clause, iteration variables x_1, x_2, ..., x_n are defined over the path expressions p_1, p_2, ..., p_n. In the `WHERE` clause, the expression c specifies conditions for qualified binding-tuples generated by the iteration variables. Some conditions may be included in p_i to select tuples iterated by the variables x_i. In the `RETURN` clause, the return structure is specified by the expression s.

4.3.1 The Algorithm

Figure 9 shows the translation algorithm for SUCXENT++. We shall illustrate the algorithm with two recursive queries. These queries vary in the number of descendant axis in XQuery statement. These examples are taken to highlight the differences in the final translated SQL query. Note that in this paper we only focus on the recursive feature of the XML queries.

Consider the query in Figure 10(a). Figure 10(b) shows the translated SQL query as obtained by applying the translation algorithm. The translation proceeds as follows.

(1) Lines 10 to 12 in Figure 10(b) translate the part of the query that seeks an entry with `"Photography"` under its `topic` element. Note that we store only the leaf nodes, their textual content and *PathIds* in the **PathValue** table. The actual path expression corresponding to the leaf node is stored in the **Path** table. Therefore, we need to join the two to obtain leaf nodes that correspond to the path `//topic//title` and contain the value `"Photography"`. This corresponds to the lines 7 to 9 in Figure 9. Also notice the use of the function `GetPathExprCondition`. It returns an SQL fragment containing the *LIKE* operator if the path contains '//'.
(2) Line 13 (Figure 10(b)) extracts the leaf nodes that correspond to the path `//topic/description`.
(3) Lines 15 to 18 in Figure 10(b) represent the SQL fragment that ensures that the extracted leaf nodes have the correct intersection relationships.

```
Input:
   XQuery query X
Output:
   Translated SQL query Q
1:  parse( X ) /* generates the parse tree */
2:  WhereClause w = Q.getWhereClause() /* this is determined by the for and where clauses */
3:  SelectClause s = Q.getSelectClause() /* the return clause determines this */
4:  for all ComparisonExpr e_i in X do
5:    if typeof(e_i.lhs) is PathExpr and typeof(e_i.rhs) is Literal then
6:      replace e_i with
7:      GetPathExprCondition( e_i.lhs );
8:      +"and Path_i.Id=PathValuei.Id"
9:      +"and PathValue_i.value <ValueComp> e_i.Literal"
10:   else if then
11:     replace e_i with /* this is a join expression */
12:     GetPathExprCondition( e_i.lhs );
13:     +"and Path_i.Id = PathValue_i.Id"
14:     +"and Path_i+1.Path = e_i.PathExpr"
15:     +"and Path_i+1.Id = PathValue_i+1.Id"
16:     +"and PathValue_i.value <ValueComp> PathValue_i+1.value"
17:   end if
18:   w.add(e_i)
19: end for
20: for all PathExpr p_i in X do
21:   for all PathExpr p_j <> p_i in X do
22:     if p_i and p_j bind to the same variable then
23:       w.add("PathValue_i.DocId=PathValue_j.DocId")
24:       l = PreIndex(p_i , p_j )
25:       w.add("abs(PathValue_i.BranchOrderSum"
26:       +"-PathValue_j.BranchOrderSum) < l")
27:     end if
28:   end for
29:   if p_i in X.returnClause() then
30:     s.add("PathValue_i.*")
31:   end if
32: end for

33: GetPathExprCondition( pathExpr ) {
34:   if pathExpr contains '//' or pathExpr contains '*' then
35:     replace '//' with
36:   end if
37:   replace '/' with '.' /* Sucxent++ uses '.' as the path separator */
38:   return pathExpr; /* modified in the lines above to suite Sucxent++'s storage mechanism */
    }
39:
40: PreIndex( path1, path2 ) {
41:   x = highest common node between path1 and path2
      /* this represents a recursive query where the intersection level
         cannot be determined beforehand - has to be done while the query is executing */
42:   if x is preceeded by '//' in both path1 and path2 then
43:     return indexOf( x, path1 );
44:   end if
      /* even if the path involves '//' the intersection level is
         not determinable beforehand only if both paths have '//' before the
         intersection node. Else, return the following */
45:   return level of x in the path where it is not preceeded by '//'
    }
46:
```

Fig. 9. Query translation algorithm.

In this example, the nodes have to intersect at the level of the first `topic` node in either of the paths corresponding to the leaf nodes. This level is determined by the user-defined function `indexOf`. It takes as input the path expressions from **Path** table containing the `topic` element and computes the intersecting level. The query, calculates the absolute value of the difference between the *BranchOrderSum* values and ensures that it is below the *RValue* for the level returned by `indexOf`. This corresponds to lines 23 to 26 and 40 to 45 in Figure 9. Notice the use of the function `PreIndex` (Line 40). It returns '`indexOf`' only if both the paths have '//' preceding the intersection node. It must be highlighted here that the intersection level can be pre-computed in every case except when both

```
Q2: FOR $b in document( "odp.xml")//topic
WHERE $b//Title = "Photography"
RETURN $b/Description
```

(a) XQuery example

```
1  SELECT
2     V2.*
3  FROM
4     PATHVALUE V1,
5     PATH P1,
6     PATHVALUE V2,
7     PATH P2,
8     DOCUMENTRVALUE D
9  WHERE
10    P1.PATHEXP LIKE '%TOPIC.%TITLE' AND
11    P1.ID = V1.PATHID AND
12    V1.LEAFVALUE = 'PHOTOGRAPHY' AND
13    P2.PATHEXP LIKE '%TOPIC.DESCRIPTION%' AND
14    P2.ID = V2.PATHID AND
15    V1.DOCID = V2.DOCID AND
16    V1.DOCID = D.DOCID  AND
17    D.LEVEL >= INDEXOF('TOPIC', P1.PATHEXP) - 1 AND
18    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

(b) Translated SQL query.

Fig. 10. Query processing in SUCXENT++.

the paths have '//' preceding the intersection node. The next example will highlight the case where only one of the paths has a '//' preceding the intersection node.

(4) The `select` clause of Lines 1-2 return the properties of the leaf nodes corresponding to the *description* element. Note that the `RETURN` clause in the XQuery query determines the `select` clause. We return all the attributes of **PathValue** tables as these properties are needed to construct the corresponding XML fragment. The reconstruction algorithm discussed in Figure 6 is used to construct the XML document from this relational result set.

Now, consider the query in Figure 7(a). Figure 8(b) shows its SQL translation as obtained by applying the translation algorithm. The translation proceeds as follows.

(1) Lines 10 to 12 in Figure 8(b) translate the part of the query that seeks an entry with `"Gold Ignot"` under its `text` element. This part is the same as in the previous translation example.
(2) Line 13 extracts the leaf nodes that correspond to the path `/site/regions/africa/item/price`.
(3) Lines 15 to 18 in Figure 8(b) represent the SQL fragment that ensures that the extracted leaf nodes have the correct intersection relationships. In this example, the nodes have to intersect at the `item` node. The function `PreIndex` returns this value (level 4). Since the intersection has to be at level 4 or higher the *RValue* taken is that of level 3. Notice that `preIndex` returns a pre-computed value even though one of the paths has a '//'.
(4) Line 2 in Figure 8(b) (translated from Line 3 in Figure 9) returns the properties of the leaf nodes corresponding to the *text* element. These properties are needed to construct the corresponding XML fragment.

The procedure of construction of XML fragments is the same as the previous query. Observe that if the `RETURN` clause in Figure 7(a) was `<item>$b</item>`, then the line 13 in the translated SQL query (Figure 8(b)) would change to `P2.PathExp LIKE '.site.regions.africa.item%'` to extract all leaf nodes that have paths *beginning* with `$b`. This way, elements and their children can be retrieved.

Compared to XParent, SUCXENT++ uses only the **PathValue**, **Path** and **DocumentRValue** tables to evaluate a query. The size of the **PathValue** and **Path** tables is the same as that of the **Data** and **LabelPath** tables in XParent. The **DocumentRValue** has the same number of rows as the depth of the document as compared to the **Ancestor** table in XParent which stores the ancestor list of every node in the document. This results in substantially better query performance in addition to much smaller storage size.

5 Optimization Techniques

A preliminary performance evaluation using the above translation procedure yielded some interesting results. We checked the query plans generated by the query optimizer and noticed that the join between the **Path** and **PathValue** tables took a significant portion of the query processing time. This was because for most of the queries this join was being performed last. For example, in the SQL query in Figure 8(b) the join between **Path** and **PathValue** tables was performed last. The initial query plan is shown in Figure 11. We have not shown the **DocumentRValue** table in the plan, even though the query optimizer includes it, as it does not influence the optimization. The two Hash-Joins (labelled 1 and 2) in this plan are both very expensive. The first takes the **PathValue** table (with alias `v2`) as one of its inputs. The second join takes the result of this join as one of its inputs. Both these inputs are quite substantial in size resulting in very expensive join operations. In order to improve the above query plan we propose three optimization techniques that are discussed below.

5.1 Optimization for Simple Path Expressions

The join expression `v1.PathId = p1.Id and p1.PathExp =` ***`path`*** is replaced with `v1.PathId =` ***`n`*** where ***`n`*** is the `PathId` value corresponding to ***`path`*** in the table `Path`. Similarly, `v1.PathId = p1.Id and p1.PathExp LIKE` ***`path`***`%` is replaced with `v1.PathId >= n and v1.PathId <= m`. For the second case `PathId`s are assigned in lexicographic order and `(n, m)` correspond to the first and last occurrences of expressions that have the prefix *path*. This changes the query plan to the one in Figure 12. Since there is no join between the **Path-**

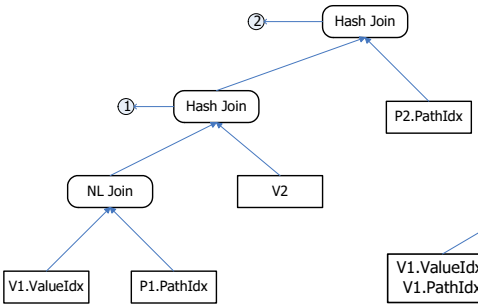


Fig. 11. Initial query plan.

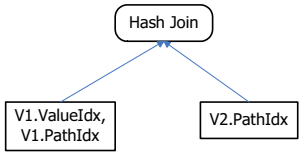


Fig. 12. Path optimization.

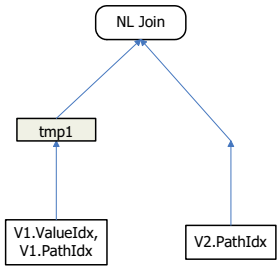


Fig. 13. Multiple-queries optimization.

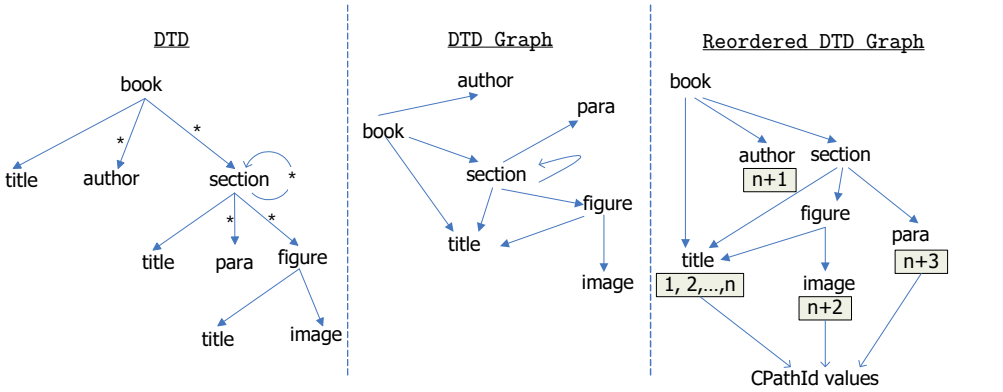


Fig. 14. Optimization for recursive path expressions.

Value and **Path** tables anymore, the joins in Lines 15-18 now get executed the last. The *PathId* and *LeafValue* predicates are evaluated earlier resulting in smaller inputs to the join operations. This optimization resulted in an improvement of up to 60% in query execution time as shown in Section 6.

5.2 Optimization for Recursive Path Expressions

A lexicographic numbering of paths is not sufficient for recursive expressions when the DTD structure is a graph. Figure 14 shows an example of such a DTD. It has a graph structure due to the recursion on the *section* element. If only lexicographic `PathId` is available, expressions such as `//title` cannot be optimized i.e., converted to a range expression instead of a join. We assign another *pathId*, called *CPathId*, to a `Path` based on the following rules.

(1) Elements in the DTD graph are ordered by the number of incoming edges. Lexicographic ordering is followed within this ordering. Figure 14 shows the "reordered" graph. The element *title* is ordered first as it has the highest number of incoming edges. $1 \ldots n$ are the *CPathId* values for paths ending in *title*.

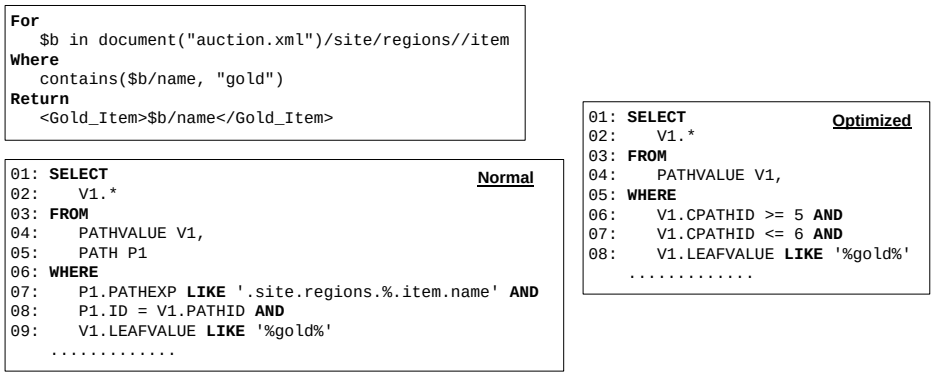


Fig. 15. Example of using *CPathId* for optimization.

(2) Cycles in the DTD graph are handled by clustering paths with the same non-recursive element after the end of the cycle. Based on this rule, `/book/section/title`, `/book/section/section/title`,..., `/book/section/.../section/title` would all occur consecutively for the DTD in Figure 14. This allows the replacement of paths such `//section//title` with range expressions in the SQL translation.

The SUCXENT++ schema has to be extended to incorporate the *CPathId* attribute together with the existing *PathId* column in **PathValue** (Figure 5). Any recursive path expression can be now be converted to a range query on the *CPathId* attribute. To expedite query processing, we also store *CPathId* in the **Path** table. For example, consider the expression `//title`. It is replaced by `(p.CPathId >= 1 and p.CPathId <= n)` as all paths ending in `title` have *CPathId* values between `1` and `n`. Similarly consider the path `//section/title`. First, the first and last *CPathId* values of `%section/title` in the **Path** table are obtained. Say, these are n_f and n_l, respectively. Then, the join expression is replaced by `(p.CPathId >=` n_f `and p.CPathId <=` n_l`)`.

Consider another query in Figure 15. The non-optimized translation involves a join between the **Path** and **PathValue** tables (line 8 in the SQL query). This join is replaced with a comparison on the *CPathId* attribute of the **PathValue** table. First, the first and last *CPathId* values of `/site/regions%item/name` are determined from the **Path** table. These two values are then used in a range comparison on the *CPathId* column of the **PathValue** table (lines 6 and 7 of the optimized SQL query). This reduce the number of joins in the final SQL query.

5.3 Optimization Using Multiple Queries

After performing the above two optimizations the new query plans still had one major limitation. The last two join expressions (lines 15-18 in Figure 7) were still being evaluated using Hash-Joins. The analysis of the two intermediate

```
SQL 1.1
1  SELECT
2     v1.* into tmp1 FROM PathValue v1, Path p1
3  WHERE p1.PathExp LIKE '.site.regions.africa.item.%.text'
4  AND v1.PathId = p1.PathId
5  AND v1.LeafValue LIKE '%Gold Ignot%'

SQL 1.2
1  SELECT
2     v1.* into tmp2 FROM PathValue v1, Path p1
3  WHERE p1.PathExp LIKE '.site.regions.africa.item.price'
4  AND v1.PathId = p1.PathId

SQL 1.3
1  SELECT
2     t2.* FROM tmp1 t1, tmp2 t2, DocumentRValue r1
3  WHERE t1.DocId = t2.DocId AND t1.DocId = r1.DocId
4  AND r1.Level = 3 AND
5  abs(t1.BranchOrderSum-t1.BranchOrderSum) < r1.RValue
```

(a) Multiple queries for Figure 7

```
SQL 1.1
1  SELECT
2     v1.* into tmp1 FROM PathValue v1, Path p1
3  WHERE p1.PathExp LIKE '.%topic.%title'
4  AND v1.PathId = p1.PathId
5  AND v1.LeafValue = 'photography'

SQL 1.2
1  SELECT
2     v1.* into tmp2 FROM PathValue v1, Path p1
3  WHERE p1.PathExp LIKE '%topic.description%'
4  AND v1.PathId = p1.PathId

SQL 1.3
1  SELECT
2     t2.* FROM tmp1 t1, tmp2 t2, DocumentRValue r1
3  WHERE t1.DocId = t2.DocId AND t1.DocId = r1.DocId
4  AND r1.Level = indexOf('topic', t1.PathExp) - 1 AND
5  abs(t1.BranchOrderSum-t1.BranchOrderSum) < r1.RValue
```

(b) Multiple queries for Figure 10.

Fig. 16. Optimization using intermediate materializations.

	SD (Single-Document)	MD (Multiple-Document)
TC (Text-centric)	Online dictionaries	Digital libraries, news corpus
DC (Data-centric)	E-commerce Catalogs	Transactional data

Fig. 17. Data set.

results used for the evaluation of the join expression found that Nested-Loop would be a better option.

Forcing a Nested-Loop-based query plan is not a good choice as there are cases where Hash-Join (or Merge-Sort join) is still a better option. Our conclusion was that we should separate the pre-join results, execute a separate join query on these temporary results and let the query optimizer decide. We materialized these pre-join results into separate temporary tables and then executed a join on these temporary tables. The query optimizer now generated a better plan for all queries. This optimization resulted in an improvement of up to 7 times as shown in Section 6. The final set of queries for the given example, in order of execution, is shown in Figure 16(a). SQL 1.1 and SQL 1.2 correspond to the intermediate results. The final result is obtained by using the intermediate results as shown in SQL 1.3. The resulting query plan is shown in Figure 13.

Figure 16(b) shows the multiple queries obtained when applied to the query in Figure 10. Notice that now the *PathExp* column of the **Path** table needs to be stored in the intermediate tables. This is because it is needed in the final query by the user-defined `indexOf` function. However, the resulting query plan is still significantly better than that for one single query.

6 Performance Evaluation

In this section, we present the results of our performance evaluation. First, we present the results for insertion and extraction times and storage space

Data set	No of Nodes		
	10MB	100MB	1GB
DC/MD	219,382	2,183,331	23,821,115
DC/SD	238,260	2,394,886	24,810,315
TC/MD	229,258	2,335,180	23,704,294
TC/SD	279,004	2,765,209	28,419,013

(a) Data set of XBench

Data set	Size (MB)	Node
ODP	142	2,884,074
XMark	150	2,668,227
Swiss-Prot	150	6,508,774

(b) Data set.

Fig. 18. Data set.

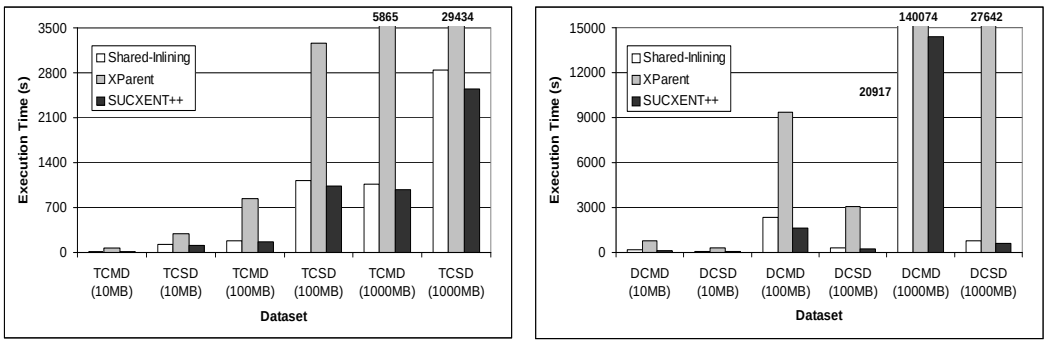
Query	Database	Query Features
Q1: FOR $b in document(“odp.xml”)//topic WHERE $b/Title = “Photography” RETURN $b/Description	**ODP[1]**	- Recursive schema - One // axis in query
Q2: FOR $b in document(“odp.xml”)//topic WHERE $b//Title = “Photography” RETURN $b/Description	ODP	- Recursive schema - Two // axis in query
Q3: FOR $b in document(“odp.xml”)//topic WHERE month($b/lastUpdate) >= 10 RETURN $b/Description	ODP	- Recursive schema - One // axis in query - typecast
Q4: FOR $b in document(“odp.xml”)//topic WHERE month($b//lastUpdate) >= 10 RETURN $b/Description	ODP	- Recursive schema - Two // axis in query - typecast
Q5: FOR $b in document(“auction.xml”)/site/regions RETURN count($b//item)	**Xmark[10]**	- One // axis - Aggregate function
Q6: FOR $b in document(“auction.xml”)/site RETURN count($b//description)+count($b//annotation)+count($b//email)	XMark	- One // axis with respect to root - Aggregate function
Q7: FOR $b in document(“auction.xml”)/site/regions/africa/item WHERE contains($b//description,”gold”) RETURN $b/name	XMark	- One // axis on Recursive portion of schema - Text search
Q8: FOR $b in document(“sprot.xml”)/sptr/entry WHERE $b/reference//authorList/person[@name=”Mueller P.”] RETURN $b/accession	**Swiss-Prot[2]**	- One // axis - Distant return and where clause
Q9: FOR $b in document(“sprot.xml”)/sptr/entry WHERE $b/reference//person[@name=”Hermann R.”] RETURN $b/reference	Swiss-Prot	- One // axis - Distant return and where clause - Shallow return clause
Q10: FOR $b in document(“sprot.xml”)/sptr/entry WHERE $b/reference//@type=”journal article” RETURN $b/accession	Swiss-Prot	- One // axis - Large result size

[1]The Open Directory Project. http://dmoz.org.
[2]The Swiss-Prot Database. http://us.expasy.org

Fig. 19. Queries and their features.

requirements of SUCXENT++ and compare it with a schema-oblivious (XParent) and a schema-conscious (Shared Inlining) approach. Next, we compare the recursive query performances of SUCXENT++ to these systems. Prototypes for SUCXENT++, XParent and Shared-Inlining were developed using Java JDK1.5 and a commercial RDBMS [2]. The experiments were conducted on a P4 1.4GHz machine with 256MB of RAM and a 40GB (7200rpm) IDE hard disk. The operating system was Windows 2000 Professional.

[2] Our licensing agreement disallows us from naming the product



(a) Text-Centric Dataset

(b) Data-Centric Dataset

Fig. 20. Insertion times.

6.1 Data Set

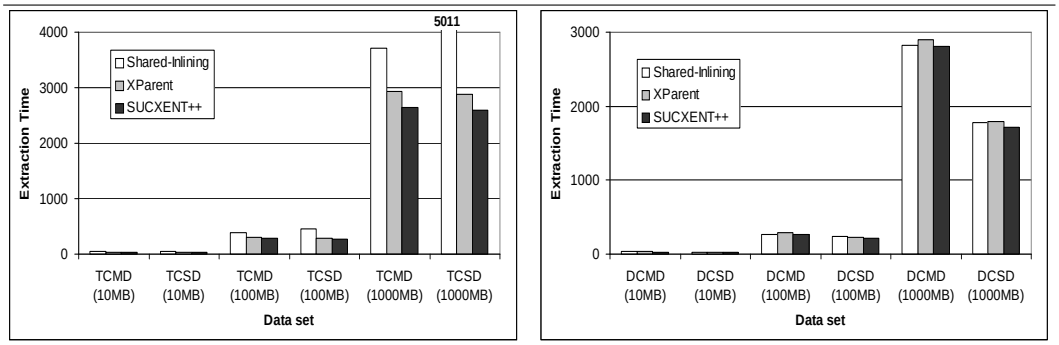
The XBench [17] data set was used for comparison of storage size, insertion and extraction times, as it provides a comprehensive range of XML document types. Both text-centric and data-centric documents are provided with data sizes ranging from 10 MB to 1 GB. Figures 17 and 18(a) summarize the characteristics of the data sets used.

For recursive query processing, we experimented with the data sets shown in Figure 18(b) and the queries shown in Figure 19. Note that the DTD graph of the ODP dataset contains cycles. Also, the SQL translation of these queries are shown in Appendix A.

6.2 Insertion Time

Figure 20 show the insertion times for the three approaches. Note that integrity constraints are not created prior to insertion. This is because we wish to study the insertion times independent of any overheads such as constraint checking. The following observations can be made based on the results for insertion times.

(1) XParent performs the worst for all four document types and across all data set sizes (5.7 - 47 times worse than SUCXENT++). This is as expected because XParent stores the greatest amount of data for a given XML document in the form of every node and its ancestor information.
(2) SUCXENT++ performs the best for all experiments. This is because it stores the least amount of data among the schema-oblivious approaches. It performs marginally better than Shared-Inlining (up to 1.5 times). This can be due to the fact that the shredding process in Shared-Inlining has to insert data into several tables whereas SUCXENT++ only needs to insert data into one table.



(a) Text-Centric Dataset

(b) Data-Centric Dataset

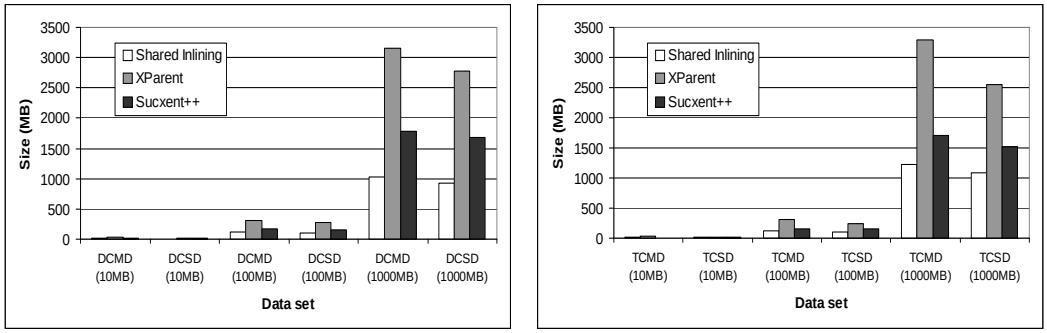
Fig. 21. Extraction times.

(3) The difference in performance increases with data size. The maximum difference in insertion performance for the 10 MB data set was 6 times for the TCSD data set between SUCXENT++ and XParent. This difference increases to 47 times for the 1 GB DCSD data set. This is because the number of non-leaf nodes (which are stored by SUCXENT and XParent in addition to the leaf nodes) increases at a faster rate than the number of leaf nodes alone.
(4) The time taken to insert the MD (multiple document) data sets is higher for both data-centric (DC) and test-centric (TC) data sets. This is because, the insertion process has to first enumerate the list of documents and then insert them one by one. This adds an additional overhead to the insertion process.

6.3 Extraction Time

Figure 21 shows the extraction times of the three approaches. The extraction time depends on the time taken to extract the relevant tuples and main-memory processing time to reconstruct the document. Based on the results in Figure 21 the following observations can be made.

(1) The extraction performance of SUCXENT++ is only slightly better than XParent. Even though the time taken to extract the relevant tuples (only leaf nodes) is smaller than the corresponding operation in XParent (that involves retrieving all the nodes of the document), we still have to perform *substring* operations to determine the nodes in a path in order to create the document tree. In Step 7 of Figure 6 the process of obtaining the node array from the path is accomplished by the *substring* operation. This means that though retrieval time from the database is better, the time taken for reconstruction is more.
(2) Even though, Shared-Inlining returns the least number of tuples while extracting a document, its performance is still marginally worse than SUCXENT++. This is because a join query needs to be executed to extract the



(a) Data-centric (b) Text-centric

Fig. 22. Storage size for data set in Figure 18(a).

document. In addition, the main memory processing time required is also higher due to the fragmented nature of the retrieved data. This difference is most significant for the TCSD data set (up to 2 times slower). The TCSD data set generates the most number of relations among the four data sets for the Shared-Inlining approach.

6.4 Storage Size

Figure 22 shows the storage sizes for the three approaches for the data set in Figure 18(a). The following observations can be made based on these results.

(1) XParent has the largest storage size (up to 2.5 times more than Shared-Inlining). In fact, XParent has a significantly larger storage size than SUCXENT++ and Shared-Inlining. This is expected as they store substantially more data in the form of ancestor information.
(2) Shared-Inlining requires the least amount of storage. In SUCXENT++ an index is created on the *PathId*, *BranchOrderSum* and *LeafValue* attributes in the **PathValue** table. Effectively, the entire data set is indexed. This is not the case in Shared-Inlining where individual columns can be indexed based on the queries being executed. The storage size values shown here are for the case where only those columns relevant to the query workload are indexed.

6.5 Recursive Query Performance

Our preliminary experiments showed that Shared-inlining outperforms SUCXENT++ for query loads similar to the ones described in [16]. However, our experiments also show that recursive queries with certain characteristics perform better in SUCXENT++ especially with the optimization techniques discussed in Section 5. Query performance is partially influenced by the flexible manner in which the XQuery `RETURN` clause can be specified. In particular,

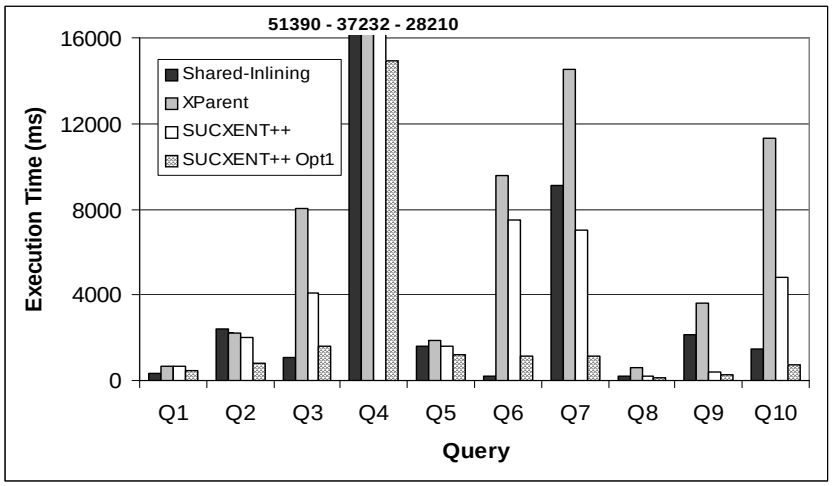
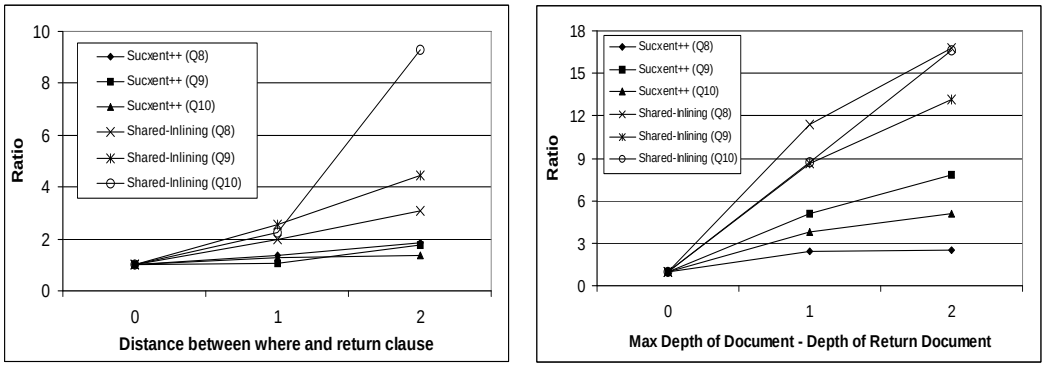


Fig. 23. Query performance.

two factors affect performance. First, the *distance* between the `WHERE` and `RETURN` clause elements, *defined as the number of edges with cardinality * (or +) between these elements in the DTD*. For example, the *distance* between *price* and *name* under *item* in the DTD in Figure 1(a) is 0 as their is no edge between them with cardinality of * or more. Similarly, the distance between *europe* and *price* is 1. The distance corresponds to the number of joins in the SQL query as generated by the Shared-Inlining approach. As another example, consider the query in Figure 7 on the document in Figure 2. The distance between the `RETURN` `WHERE` elements is greater than 1. The exact distance is not known as the schema is recursive and there could be any number of recursive *text* elements. For the Shared-Inlining approach this distance is the number of tables that need to be joined, thus effecting performance. The second factor is the depth of the element specified in the `RETURN` clause. Shallow elements would require a greater number of joins in the Shared-Inlining approach as the descendants are likely to be fragmented across several tables.

Figure 23 shows the results for query performance. Figure 24(a) shows the variation of query execution time with increasing distance between the `WHERE` and `RETURN` clauses in the XQuery query. The *ratio* in the y-axis represents the ratio of time when the distance between the `WHERE` and `RETURN` clauses is equal to n ($n > 0$) to the time when the distance is 0. Figure 24(b) shows the results as the depth of the `RETURN` clause element is reduced (or, as $(D - depth)$ is increased, where D is the maximum depth of the document). The *ratio* in the y-axis represents the ratio of time when the depth of the `RETURN` clause element is equal to n ($n > 0$) to the time when the depth is D. We now elaborate on these performance results.



(a) Variation with Distance

(b) Depth of the Return Clause

Fig. 24. Query performance.

6.5.1 Query Performance Without Optimization

Shared-Inlining performs better than SUCXENT++ for queries Q_1 and Q_3. This is because the corresponding SQL query only involves the `topic` table and there is no need for any recursive SQL query as it is known that all `topic` elements are in the `topic` table and only their immediate `title` values need to be queried. SUCXENT++, on the other hand, has to execute a join query between the **Path** and **PathValue** tables. In addition Q_3 involves a *typecast* to *date* in SUCXENT++'s case as all data is stored as `strings`. Q_2 and Q_4 are quite similar to Q_1 and Q_3 except that the `title` element can be a descendant and not just a child. A recursive SQL query is generated for the Shared-Inlining approach using the technique mentioned in [9]. In SUCXENT++ we only need to ensure that the intersection level is equal to the *length* of the path minus one as the `Description` element is a child of the `topic` element in question. XParent also benefits from this approach and therefore performs better than Shared-Inlining. For Q_5, Shared-Inlining involves a `UNION` of the joins between `item` and each of `asia`, `namerica`, `samerica`, `europe`, `africa` and `australia`. In SUCXENT++ this query merely look for paths with the expression `/site/regions/*/item`. However, the Shared-Inlining approach will perform much better if we use the knowledge that `item` is not used anywhere else in the document. Then, it would reduce to a count of the `item` table. This is highlighted by Q_6 where the Shared-Inlining approach performs much better than SUCXENT++ or XParent (by 35 times). Here, the result is merely a sum of the tuples in `description` and `annotation`. This can be done because the paths are evaluated with respect to the root and it is implied that all `description` and `annotation` elements will be counted.

SUCXENT++ performs better than Shared-Inlining for Q_7 to Q_9. This is because the result that needs to be returned is in a different subtree. This leads to a greater number of joins in Shared-Inlining whereas, the number of joins remains unaltered for SUCXENT++. The difference is greater for Q_9 (about

Query	Non Optimized		Optimized	
	Shared Inlining	XParent	Shared Inlining	XParent
Q1	0.5385	1.0292	0.7000	1.3380
Q2	1.2055	1.1093	2.9509	2.7153
Q3	0.2661	1.9536	0.6865	5.0401
Q4	1.8217	0.6026	3.4455	1.1398
Q5	1.0100	1.1907	1.3484	1.5896
Q6	0.0280	1.2726	0.1888	8.5971
Q7	1.2978	2.0717	8.1250	12.9705
Q8	1.0500	2.9100	1.3636	3.7792
Q9	5.4806	9.3049	7.3902	12.5470
Q10	0.3127	2.3518	2.0324	15.2865

Fig. 25. Summary of query performance.

5 times) than the other queries as it involves recursion, significant distance between the `RETURN` and `WHERE` clause elements and a shallow return clause in the form of the `reference` element (whose descendants are spread across 4 tables). However, SUCXENT++ performs worse for Q_{10}. This is because of the poor query plan generated by the database and can be resolved by applying the optimizations discussed in Section 5. To summarize, SUCXENT++ outperforms Shared-Inlining for 6 out of 10 queries by up to 5 times.

6.5.2 Query Performance With Optimization

Notice that there is an improvement in most queries after the optimizations. Q_1 to Q_4, Q_7 and Q_{10} show a more remarkable difference. For Q_3 and Q_4, this is partially due to the removal of the *date* typecast. When materializing the intermediate result we insert `lastUpdate` leaf nodes as *date* types. Also, all three optimizations are used for these queries. Q_5 and Q_6 only benefit from the first two optimizations. The intermediate result sizes for Q_8 and Q_9 are not large enough to benefit from the optimizations. In fact, Q_8 is adversely effected due to the overhead of the optimizations and performs worse. Q_{10}, on the other hand, shows a significant performance improvement and outperforms Shared-inlining approach. This is due to the better query plan generated as a result of using all three optimizations.

Figure 25 presents a summary of the query performance results for the compared approaches with respect to SUCXENT++. This figure shows the ratio of time taken for a given approach to the time taken in SUCXENT++.

6.5.3 Performance Variation with Distance

For this section we have used queries Q_8 to Q_{10} for comparison. The reason being that they represent real-world scenarios for queries with distant elements in the `RETURN` and `WHERE` clauses. Notice that SUCXENT++'s performance is independent of the *distance* between the `WHERE` and `RETURN` clause

elements (Figure 24(a)). This is expected as the number of join operations remains unchanged. A performance change will be seen only if the number of elements that need to be returned changes. The performance of the Shared-Inlining approach, on the other hand, is effected considerably. In fact, for queries where the elements are in the same table, Shared-Inclining outperforms SUCXENT++ significantly. As the distance increases Shared-Inlining performs worse due to the increase in the number of joins. The increase can be as much as 9 times depending on the sizes of the tables involved in the joins.

6.5.4 Performance Variation with Shallowness of `RETURN` Clause

This effects the query performance significantly as shown in Figure 24(b). The results are plotted against decreasing depth (or increasing $(D-depth)$ where D is the maximum depth of the document) of the `RETURN` clause. Notice that the performance of SUCXENT++ is also effected adversely by up to 8 times. This is because a greater number of leaf nodes need to be returned. However, there is no increase in the number of joins. Therefore, the performance degradation is not as severe as it is for Shared-Inlining which can be effected by up to 17 times.

7 Conclusions and Future Work

Recently, a growing body of work suggests that schema-conscious approaches perform better than schema-oblivious approaches. To the best of our knowledge, this paper reports the first attempt to show that it is indeed possible to design a schema-oblivious approach that can outperform schema-conscious approaches as far as the execution of *certain types* of recursive XML queries is concerned. In particular, our approach outperforms Shared-Inlining (schema-conscious approach) by up to 8 times for recursive queries that have the following features: (a) Recursion in the schema; (b) a large distance between the elements of the `WHERE` and `RETURN` clauses; and (c) shallow `RETURN` clause elements. In fact, the non-optimized version of SUCXENT++ outperformed Shared-Inlining for 6 of the 10 queries we tested. SUCXENT++ benefits substantially from the optimization techniques discussed in Section 5. The improvement is especially significant for queries that use all three optimization techniques. Once the optimization techniques were used it performed better for 7 queries. Also, SUCXENT++ performs marginally better compared to Shared Inlining as far as insertion and extraction times are concerned (up to 1.5 times and 2 times respectively).

SUCXENT++ significantly outperforms existing schema-oblivious approaches like XParent. It is 5.7 - 47 times faster than XParent as far as insertion time

is concerned. XParent takes 2.5 times more storage space compared to our approach. SUCXENT++ outperforms XParent for almost all recursive queries that we have experimented with. The non-optimized version of our approach outperforms XParent up to 9 times. The optimized version of SUCXENT++ outperforms XParent by up to 15 times. Note that the optimization techniques in SUCXENT++ can also be applied to XParent. A preliminary study of the query plans generated by the RDBMS for XParent suggests that XParent can also benefit from these optimizations. However, a considerable performance difference shall still exist between SUCXENT++ and XParent, especially for large data set, primarily due to XParent's large storage requirements and consequently, greater I/O-cost in evaluating queries.

SUCXENT++ currently uses both *PathId* and *CPathId* in **PathValue** to optimize query performance. This adds to the storage size and can be eliminated by devising a technique to combine these two identifiers. Another bottleneck in the performance of SUCXENT++ is the optimization of the `abs(v1.BranchOrderSum-v2.BranchOrderSum) < r.RValue` operation. This requires the implementation of structural-join algorithms in the database engine. However, as a preliminary study we implemented a variation of the MPMJN [19] algorithm using a stored procedure in the relational database. We saw an improvement of up to 10 times for some queries. We plan to investigate this further.

References

[1] F. BANCIHON, G. BARBEDETTE, V. BENZAKEN ET AL. The Design and Implementation of O2, an Object-oriented Database System. *proc. of the Second International Workshop on Object-oriented Database* , 1988.

[2] P. BOHANNON, J. FREIRE, P. ROY, J. SIMEON. From XML Schema to Relations: A Cost-based Approach to XML Storage. *In Proceedings of IEEE ICDE*, 2002.

[3] D. DEHAAN, D. TOMAN, M. P. CONSENS, M. T. OZSU. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Coding. *In Proceedings of ACM SIGMOD*, 2003.

[4] A. DEUTSCH, M. FERNANDEZ, D. SUCIU. Storing Semistructured Data with STORED. *In Proceedings of ACM SIGMOD*, 1999.

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

[6] R. GOLDMAN, J. MCHUGH, J. WIDOM. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. *Proceedings of WebDB '99*, pp. 25-30, 1999.

[7] H. JIANG, H. LU, W. WANG AND J. XU YU. Path Materialization Revisited: An Efficient Storage Model for XML Data. *13th Australasian Database Conference (ADC'02)* , 2002.

[8] R. Krishnamurthy, R. Kaushik, J. F. Naughton. XML to SQL Query Translation Literature: The State of the Art and Open Problem.*In Proceedings of XML Database Symposium (XSym'03)*, 2003.

[9] R. Krishnamurthy, V. T. Chakaravarthy, R. Kaushik, J. F. Naughton. Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation.*In Proceedings of IEEE ICDE*, 2004.

[10] S. Prakash, S. S. Bhowmick, S. K. Madria. Efficient Recursive XML Query Processing Using Relational Databases. *In Proc. of ER*, 2004.

[11] M. Ramanath, J. Freire, J. Haritsa, P. Roy. Searching for Efficient XML-to-relational Mappings. *In Proceedings of the International XML Database Symposium*, 2003.

[12] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu and R. Busse. XMark: A Benchmark for XML Data Management. *In Proceedings of VLDB*, 2002.

[13] A. Schmidt, M. Kersten, M. Windhouwer, F. Waas. Efficient Relational Storage and Retrieval of XML Documents. *In Proceedings of the Workshop on Web and Databases (WebDB)*, 2000.

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

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

[16] F. Tian, D. DeWitt, J. Chen and C. Zhang. The design and performance evaluation of alternative XML storage strategies. *ACM Sigmod Record, Vol. 31(1), 2002* .

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

[18] 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.

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

[20] DB2 XML Extender. `http://www-3.ibm.com/software/data/db2/extenders/xmlext/index.html`.

[21] Oracle9i XML Database Developer's Guide - Oracle XML DB Release 2 (9.2). `http://otn.oracle.com/tech/xml/xmldb/content.html`.

A SQL Queries in SUCXENT++

```
Q1: SELECT
        V2.*
    FROM
        PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
        P1.PATHEXP LIKE '%TOPIC.TITLE' AND
        P1.ID = V1.PATHID AND
        V1.LEAFVALUE='PHOTOGRAPHY' AND
        P2.PATHEXP LIKE '%TOPIC.DESCRIPTION%' AND
        P2.ID = V2.PATHID AND
        V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
        D.LEVEL = INDEXOF('TOPIC', P1.PATHEXP) - 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q2: SELECT
        V2.*
    FROM
        PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
        P1.PATHEXP LIKE '%TOPIC.TITLE' AND
        P1.ID = V1.PATHID AND
        V1.LEAFVALUE='PHOTOGRAPHY' AND
        P2.PATHEXP LIKE '%TOPIC.DESCRIPTION%' AND
        P2.ID = V2.PATHID AND
        V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
        D.LEVEL = INDEXOF('TOPIC', P1.PATHEXP) - 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q3: SELECT
        V2.*
    FROM
        PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
        P1.PATHEXP LIKE '%TOPIC.LASTUPDATE' AND
        P1.ID = V1.PATHID AND
        MONTH(CONVERT(V1.LEAFVALUE, SMALLDATETIME)) >= 10 AND
        P2.PATHEXP LIKE '%TOPIC.DESCRIPTION%' AND
        P2.ID = V2.PATHID AND
        V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
        D.LEVEL = INDEXOF('TOPIC', P1.PATHEXP) - 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q4: SELECT
        V2.*
    FROM
        PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
        P1.PATHEXP LIKE '%TOPIC.LASTUPDATE' AND
        P1.ID = V1.PATHID AND
        MONTH(CONVERT(V1.VALUE, SMALLDATETIME)) >= 10 AND
        P2.PATHEXP LIKE '%TOPIC.DESCRIPTION%' AND
        P2.ID = V2.PATHID AND
        V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
        D.LEVEL = INDEXOF('TOPIC', P1.PATHEXP) - 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q5: SELECT
        COUNT(DISTINCT(V1.VALUE)
    FROM
        PATHVALUE AS V1, PATH P1
    WHERE
        P1.PATHEXP LIKE '.SITE.REGIONS%ITEM.NAME' AND
        V1.PATHID = P1.ID

Q6: SELECT
        COUNT(DISTINCT(V1.VALUE)
    FROM
```

```
        PATHVALUE AS V1, PATH P1
    WHERE
        (
            (
                P1.PATHEXP LIKE '.SITE.%ANNOTATION%' AND
                V1.BRANCHORDER <= INDEXOF('ANNOTATION', P1.PATHEXP)) OR
                (
                    P1.PATHEXP LIKE '.SITE.%DESCRIPTION%' AND
                    V1.BRANCHORDER <= INDEXOF('DESCRIPTION', P1.PATHEXP)
                ) OR
                (
                    P1.PATHEXP LIKE '.SITE.%EMAIL%' AND
                    V1.BRANCHORDER <= INDEXOF('EMAIL', P1.PATHEXP)
                ) AND
                V1.PATHID = P1.ID

Q7: SELECT
        V2.*
    FROM
        PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
        P1.PATHEXP LIKE '.SITE.REGIONS.AFRICA.ITEM.%DESCRIPTION%' AND
        P1.ID = V1.PATHID AND
        V1.LEAFVALUE LIKE '%GOLD%' AND
        P2.PATHEXP = '.SITE.REGIONS.AFRICA.ITEM.NAME' AND
        P2.ID = V2.PATHID AND
        V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
        D.LEVEL = 3 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q8: SELECT
        V2.*
    FROM
        PATHVALUE V1, PATHVALUE V2, PATH P1, PATH P2, DOCUMENTRVALUE D
    WHERE
        P1.PATHEXP LIKE '.SPTR.ENTRY.REFERENCE.%AUTHORLIST.PERSON.@NAME' AND
        P1.ID = V1.PATHID AND
        V1.LEAFVALUE='MUELLER P.' AND
        V2.PATHEXP = '.SPTR.ENTRY.ACCESSION' AND
        P2.ID = V2.PATHID AND
        V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
        D.LEVEL = 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q9: SELECT
        V2.*
    FROM
        PATHVALUE V1, PATHVALUE V2, PATH P1, PATH P2, DOCUMENTRVALUE D
    WHERE
        P1.PATHEXP LIKE '.SPTR.ENTRY.REFERENCE.%PERSON.@NAME' P1.ID = V1.PATHID AND
        V1.LEAFVALUE='HERMANN R.' AND
        V2.PATHEXP LIKE '.SPTR.ENTRY.REFERENCE%' AND
        P2.ID = V2.PATHID AND
        V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
        D.LEVEL = 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q10:SELECT
        V2.*
    FROM
        PATHVALUE V1, PATHVALUE V2, PATH P1, PATH P2, DOCUMENTRVALUE D
    WHERE
        P1.PATHEXP LIKE '.SPTR.ENTRY.REFERENCE.%@TYPE' P1.ID = V1.PATHID AND
        V1.LEAFVALUE='JOURNAL ARTICLE' AND
        V2.PATHEXP = '.SPTR.ENTRY.ACCESSION' AND
        P2.ID = V2.PATHID AND
        V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
        D.LEVEL = 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```