Fast Evaluation of Multi-source Star Twig Queries in a Path Materialization-based XML Database

Erwin Leonardi[†] **Sourav S Bhowmick**[†] **Fengrong Li**[§]

[†]School of Computer Engineering
Nanyang Technological University, Singapore

[§]Japan Advanced Institute of Science and Technology
Japan
lerwin|assourav@ntu.edu.sg, lifr@nagoya-u.jp

Abstract

Despite a large body of work on XML twig query processing in relational environment, systematic study of XML join evaluation has received little attention in the literature. In this paper, we propose a novel and non-traditional technique for fast evaluation of *multi-source star twig* queries in a *path materialization*-based RDBMS. A *multi-source star twig* joins different XML documents on values in their nodes and the *XQuery graph* takes a star-shaped structure. Such queries are prevalent in several domains such as life sciences. Rather than following the conventional approach of generating one huge complex SQL query from a twig query, we translate a star query into a list of SQL sub-queries that only materializes *minimal information* of underlying XML subtrees as intermediate results. We have implemented this scheme on top of a path materialization-based XML storage system called SUCXENT++. Experiments carried out confirm that our proposed approach built on top of an off-the-shelf commercial RDBMS has excellent real-world performance.

1 Introduction

XML has emerged as the leading textual language for representing and exchanging data over the Web in a wide variety of domains. This has generated tremendous interest in the mainstream database community to propose innovative solutions for storage and query processing of large volumes of XML data on top of relational as well as native framework [8]. Consequently, query languages such as XPath and XQuery have been receiving a great deal of attention from the community lately.

Efficient evaluation of XML queries that correlate (join) multiple input documents to integrate data from different sources is highly important due to its several real-world applications. For example, querying biological data across multiple sources is a key activity for many biologists. If these sources represent data in XML format (e.g., INTERPRO (`www.ebi.ac.uk/interpro/`), UNIPROT (`www.expasy.ch/sprot/`), PDB (`www.pdb.org`), EMBL (`www.ebi.ac.uk/embl/`)), then XQuery can be used to formulate meaningful queries over these data sources. Figure 1 shows examples of XML representations of two sources. Figure 2 shows three example queries. Observe that Q_1, Q_2, and Q_3 correlate four, three, and two data sources, respectively. Also, in each query the join conditions share a common data source. For instance, in Q_1 UNIPROT is joined with INTERPRO, PDB, and EMBL. Similarly, in Q_2 UNIPROT is joined with INTERPRO and EMBL. Consequently, each of these queries can be represented as a star-shaped query graph where a node represents a data source and an edge represents existence of a join expression between a pair of sources. We refer to such queries as *multi-source star twig* queries (*star queries* for brevity). *In this paper, we focus on fast evaluation of this type of queries in a relational environment.*

At first glance, it may seem that we can efficiently evaluate star queries by leveraging on an existing relational XQuery processor, *c.f.*, [10, 16] and relying on its query optimization capabilities. Specifically in an XQuery processor, an XQuery query is often rewritten to an equivalent, logically simpler XQuery and then translated to a *single*, complex SQL query, *c.f.*, [10]. Optimization of an XQuery query is achieved in two stages. Logical query optimization (sometimes also called query rewrite) [10, 15–17] results in rewrites of XQuery statements to avoid duplicate and full navigations. On the other hand, physical query optimization depends on the storage method of the data being queried. For instance, we can store and query XML representations of INTERPRO, UNIPROT, PDB, and EMBL using the XML query processor of an industrial-strength RDBMS denoted as XDB (Due to legal restrictions, this processor is anonymously identified as XDB in the sequel).

Unfortunately, query performance still remains a bottleneck. To get a better understanding of this problem, we experimented with the datasets in Figure 3(a) and queries $Q_1 - Q_3$. Figure 3(b) shows the query evaluation times in XDB. Observe that it can take from 4 minutes to more than 30 minutes to evaluate these queries. *Is it possible to design a scheme that can address this performance bottleneck?* In this paper, we demonstrate that techniques built on top of an existing off-the-shelf RDBMS can make up for a large part of the limitation. In particular, we show that the

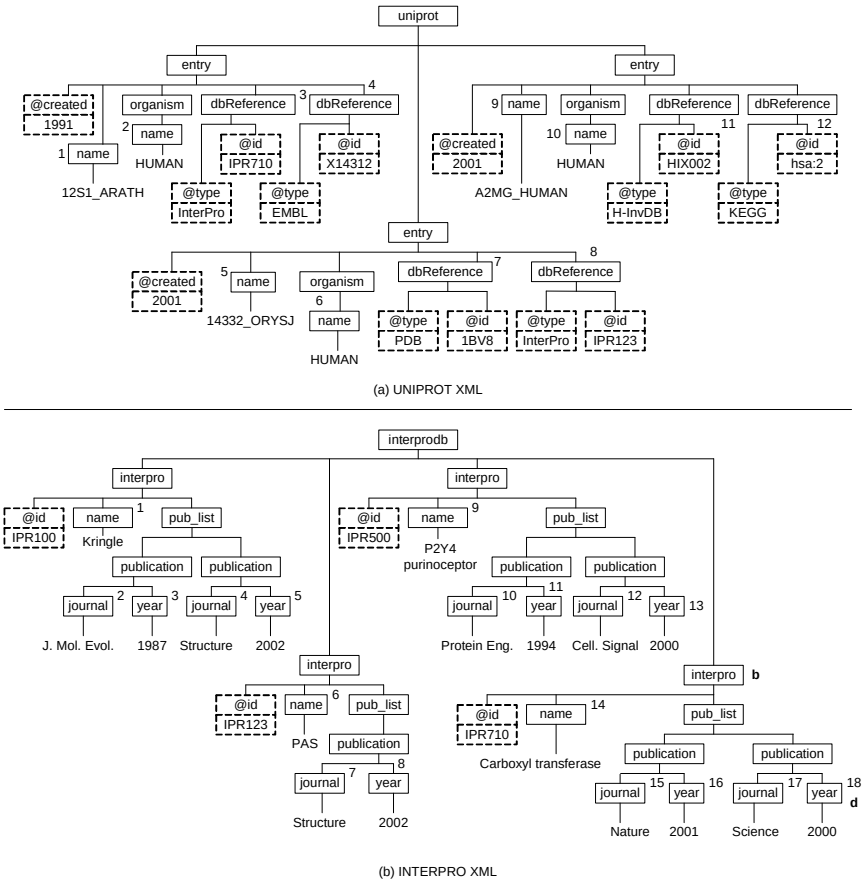


Figure 1: XML representations of UNIPROT and INTERPRO data sources.

above queries can be evaluated in *less than a minute*.

1.1 Overview

We take an alternative non-traditional strategy that bypasses logical XQuery optimization and relies solely on the relational optimizer to achieve superior performance for evaluating star queries. This approach is perhaps surprising because the design goals of our strategy seem to be diametrically opposite to traditional relational XQuery processors. Specifically, given a star query Q, our proposed algorithm translates it into a list of SQL queries without undertaking any logical query optimization over a *path materialization*-based storage scheme [8]. First, SQL queries for materializing the *identifiers* of nodes or subtrees satisfying the expressions in the `return` clause are generated. Based on these materialized identifiers, SQL queries for *non-join* expressions in the `where` clause are generated followed by queries for *join* expressions. These queries are executed in sequence and the results are materialized in temporary tables. The identifiers of nodes (subtrees)

QID	Query	# of Results
Q1	01 declare namespace PDBx = 'http://deposit.pdb.org/pdbML/pdbx.xsd'; 02 for $entry in fn:collection('UNIPROT')/uniprot/entry, 03 $interpro in fn:collection('INTERPRO')/interprodb/interpro, 04 $embl in fn:collection('EMBL')/EMBL_Services/entry, 05 $pdb in fn:collection("PDB")/PDBx:datablock 06 let $ref2PDB := $entry/dbReference[@type="PDB"]/@id 07 let $ref2EMBL := $entry/dbReference[@type="EMBL"]/@id 08 let $ref2InterPro := $entry/dbReference[@type="InterPro"]/@id 09 let $temp:=$embl/@created 10 where $entry/keyword = 'ATP-binding' 11 and $entry/organism/name = 'Human' 12 and $interpro/pub_list/publication/journal = 'Science' 13 and fn:starts-with(xs:string($temp), '1996') 14 and $pdb/PDBx:citationCategory/PDBx:citation/PDBx:country = "US" 15 and $pdb/PDBx:citationCategory/PDBx:citation/PDBx:year = "1997" 16 and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $ref2PDB 17 and $interpro/@id = $ref2InterPro 18 and $embl/@accession= $ref2EMBL 19 return $entry/reference/citation/title;	64
Q2	01 for $entry in fn:collection('UNIPROT')/uniprot/entry, 02 $interpro in fn:collection('INTERPRO')/interprodb/interpro, 03 $embl in fn:collection('EMBL')/EMBL_Services/entry 04 let $ref2EMBL := $entry/dbReference[@type="EMBL"]/@id 05 let $ref2InterPro := $entry/dbReference[@type="InterPro"]/@id 06 let $temp:=$embl/@created 07 where $entry/keyword = 'ATP-binding' 08 and $entry/organism/name = 'Human' 09 and $interpro/pub_list/publication/journal = 'Science' 10 and fn:starts-with(xs:string($temp), '1996') 11 and $interpro/@id = $ref2InterPro and $embl/@accession= $ref2EMBL 12 return $entry/name;	4
Q3	01 for $entry in fn:collection('Uniprot')/uniprot/entry, 02 $ip in fn:collection('InterPro')/interprodb/interpro 03 let $id := $entry/dbReference/@id 04 where $entry/organism/name = 'HUMAN' 05 and $entry/@created = '2001' 06 and $ip/pub_list/publication/journal = 'Structure' 07 and $ip/pub_list/publication/year = '2002' 08 and $ip/@id = $id 09 return <result><entry_name>{$entry/name}</entry_name> <protein_name>{$ip/name}</protein_name></result>;	26

Figure 2: Examples of star twig queries.

satisfying Q are then computed from these materialized results. A key feature of these materialized results is that we only store *minimal* information (identifiers of nodes) required for evaluating Q. This obviously has positive impact on the storage and query processing costs of temporary tables as we can efficiently store large intermediate result nodes for a given query. Finally, the last step of the algorithm is to issue an SQL query to retrieve *complete* information from the base table(s) containing XML documents by matching the identifiers of the result subtrees.

It may seem that the above strategy of translating an XML query into a list of SQL queries instead of a single complex SQL query has been adequately addressed before in the context of XML *publishing* environment [7, 11]. In this environment, data is originally in relational form and is viewed and queried as XML. In contrast, our proposed technique is built on top of XML *storage* environment where data is originally in XML format and is stored and queried in an RDBMS. Due to the mismatch in the environment, the techniques used for SQL translation in XML publishing cannot

Source	Size	No. of Files	No. of Attributes	No. of Nodes	Level
UNIPROT	1.4 GB	1	38,380,645	28,247,711	6
INTERPRO	50 MB	1	944,564	754,607	5
PDB	613 MB	70	1,521,615	12,535,308	4
EMBL	1.28 GB	10	13,311,359	16,707,319	6

(a) Real World Data Sets

QID	XDB2
Q1	1,421.16
Q2	238.73
Q3	DNF

(b) Query Evaluation Time (in sec.)

* DNF means that the query evaluation did not finish in 30 mins

Figure 3: Dataset and query evaluation times in XDB.

be directly mapped to the XML storage environment [12]. This is because these techniques generate optimal list of SQL queries by exploiting either the query cost estimates from the target query optimizer [7] or relational integrity constraints [11]. Unfortunately, such cost estimates are not readily available as the relational engine does not understand tree-shaped data. Furthermore, any SQL generation technique requires understanding of the structural relationships of XML elements. This is not necessary in XML publishing environment as the underlying data is relational in nature. We elaborate on the differences further in Section 2.

Our proposed algorithm is built on top of the SUCXENT++ system [2, 18], a *path materialization*-based approach [8] designed primarily for read-mostly workloads. Based on the encoding scheme of SUCXENT++, we demonstrate the use of "XQuery-to-list of SQL" translation strategy to accelerate evaluation of star queries in a widely available commercial RDBMS (denoted as SDB in the sequel). In particular, our proposed approach has excellent real-world performance. It is significantly faster than XML support of XDB (highest observed factor being 158 times), which relies on conventional XQuery optimization techniques. Somewhat unexpectedly, we shall also show that the proposed technique outperforms a state-of-the-art column store-based XQuery processor (MONETDB/XQuery [3]) for several queries (highest observed factor being 46 times)!

The rest of our paper is organized as follows. We compare our approach with related work in Section 2. Section 3 formally defines the notion of multi-source star twig queries. Sections 4 and 5 present in detail the algorithm for evaluating star queries on top of a path materialization-based relational storage. We evaluate and compare the performance of our proposed technique through an extensive set of experiments in Section 6. Section 7 concludes the paper and suggests future work.

2 Related Work

XPath and XQuery processing and optimization. There is a wealth of work on evaluating XPath expressions in a tree-unaware RDBMS [2, 8, 9, 18, 21, 22] and tree-aware environment [3, 8]. However, these efforts mainly focus on various XPath axes and not on XML join operation. In particular, the work reported in this paper

differs from efforts related to SUCXENT++ [2,18] in the following ways. Firstly, [18] focused on evaluating *ordered* axes in linear XPath expressions. In [2], Bhowmick et al. described efficient technique for evaluating twig queries having parent-child relationships. In contrast, in this paper we focus our attention to complex XQueries containing XML joins. Secondly, like many other works, the SQL translation algorithms in [2,18] generate a single complex SQL whereas here we focus on generating a sequence of SQL queries. Consequently, in this paper we materialize *minimal* subtree information to reduce the size of the intermediate tables generated by the list of SQL queries. Complete information related to subtrees that satisfy the query is only retrieved during the final step of query execution. Such "lazy" approach to retrieve subtree information is not necessary in approaches that are based on a single SQL query. Also, in contrast to previous efforts, the proposed algorithm is sensitive to the order of evaluation of different components (i.e., `return` clause, *join* expressions, *non-join* expressions) of the star XQueries.

Several works on XQuery processing and optimization adopt a traditional approach based on rewrite rules. In [5, 10, 15–17, 19], the authors discuss various rules for XQuery normalization or for transformation tasks such as XQuery-to-SQL translation, elimination of unnecessary ordering operations or introduction of a tree-pattern operator in query plans. In [14], XML document projection is used for query optimization. Given an XQuery expression Q over a document D, these works focus on identifying and projecting out the parts of D that are not useful for the evaluation of Q. These efforts focus on general XQuery or a proper subset of it. In contrast, we present techniques for optimizing performance of a special type of XQuery (star queries) in a tree-unaware environment by decomposing a star query into a sequence of SQL queries instead of a single complex SQL. Furthermore, several of the tree-unaware approaches are orthogonal to the proposed technique presented here.

Query translation in XML publishing environment. There has been efforts related to translating XML queries to SQL in XML publishing environment. A detailed description of some of the existing work on XML-to-SQL query translation is given in [12]. In XPeranto [19], a general framework for processing arbitrarily complex XQuery queries over XML view is presented. An XQuery query is transformed into an XML Query Graph Model (XQGM) and composed with the view definition. A set of rewrite optimization techniques are proposed for elimination of intermediate XML fragments construction and to push down predicates. Then it is translated to a single "outer union" SQL query to be evaluated inside the relational engine. The Agora [13] project uses *local-as-view* (LAV) approach and provides an algorithm to translate XQuery FLWR expressions into SQL in two steps. First, it translates the XML query into a SQL query over virtual relational schema and then it rewrites this SQL query into a query over the real relational schema. MARS [6] uses both local-as-view and *global-as-view* (GAV) approaches to translate an XQuery query to SQL. It first compiles the queries, views and constraints from XML into the relational framework and then takes a cost-based approach to determine all minimal reformulations of

the relational queries under the relational integrity constraints. In contrast, our approach is built on top of XML storage framework and translates a specific type of XQuery query to *a list of* SQL queries instead of a single SQL query.

More germane to this work is efforts in the XML publishing environment that translate an XML query to a list of SQL queries [7, 11]. In [7], relational schema to the XML view mapping is specified using a declarative query language RXL. In order to create the XML view, optimal set of SQL queries are generated to extract and group data from the underlying relational engine. The authors propose a plan generation technique that partitions a *view tree* into one or more subtrees; for each subtree, one SQL query is generated. In general, there are $2^{|E|}$ possible translations of an RXL query into one or more queries, where $|E|$ is the number of edges in the query's view tree. In contrast, the number of SQL queries in our approach is linear to the number data sources to be joined and the number of *output expressions* in the query. Furthermore, [7] supports XML-QL instead of XQuery. It is not clear what subset of XML-QL is handled by the current solution.

Krishnamurthy et al. [11] proposed a translation technique that exploits the relational integrity constraint information to obtain optimized SQL queries. Note that such integrity constraint information are not usable in a tree-unaware, schema-oblivious XML storage environment. Additionally, this approach only supports simple path expressions whereas we support more complex join XQueries. Furthermore, no published performance study is available of this approach. On the other hand, we undertake an exhaustive performance study to demonstrate the superiority of our approach.

3 Multi-source Star Twig Pattern

3.1 Multi-source Twig Pattern

Most XML processors, both native and relational, have overwhelmingly focused on *single-source* twig queries modeled as a twig pattern tree [8]. A *single-source* twig query is evaluated on a set of documents represented by a single XML schema or DTD. However, as discussed in Section 1, related data in many real-world applications may span across multiple data sources with different schemas. Consequently, our query model should support queries over such multiple data sources using joins. We refer to such twig queries as *multi-source twig patterns*.

A multi-source twig pattern Q is a graph with four types of nodes: *tag* node (`QNode`), *value* node (`VNode`), logical-AND node (`ANode`), and return node (`RNode`). Each `QNode` represents a node test (*i.e.,* the corresponding element or attribute label in Q). A `VNode` is a leaf node in Q and consists of two components: (a) a string value v and (b) a comparison operator or XQuery function op (*e.g.,* contains) over v. Each Q has a single node of type `RNode` which represents the output node. While labels of `ANode` is always "AND", `QNode`s' and `RNode`s' labels are tags. An edge in Q can be of two types, namely, *axes edge* and *join edge*. The former represents

parent-child or attribute relationship [1] between a pair of nodes belonging to the same source whereas the latter connects two nodes from two different sources[2]. Specifically, a join edge (q_1, q_2) asserts that q_1 and q_2 have equal value [3]. We denote the `RNode` by underlined tag; and axes and join edges as direct and dashed lines, respectively.

3.2 Representing MUST Pattern using XQuery

Observe that a multi-source twig query can be represented by an XQuery query $Q = (\mathcal{F}, \mathcal{L}, \mathcal{W}, \mathcal{R})$ where $\mathcal{F}$ is a set of `for` clause items, $\mathcal{L}$ is a set of expressions defined using the `let` clause, $\mathcal{W}$ is a set of predicates in the `where` clause, and $\mathcal{R}$ is an output expression specified in the `return` clause. Note that a path expression in these clauses can easily be constructed from a MUST pattern Q by concatenating the sequence of `QNode`s representing node tests and axes edges representing location steps (we ignore the `ANode` nodes) in a root-to-node path in Q[4]. Specifically, the syntax of Q is as follows.

`for`	$\$x_1$ *in* $p_1, \ldots, \$x_n$ *in* p_n
`let`	$\$y := q_1$
`let`	...
`where`	$\mathcal{W}$
`return`	r

Note that there must be at least two `for` clause items in Q that are bound to two different document sources. The `let` clause simply declares a variable and gives it a value. We categorize the *where-expressions* in $\mathcal{W}$ into two types, namely *join expressions* and *non-join expressions*. A *join expression* $p_1 = p_2$ involves predicates that express join conditions over two document sources represented by the path expressions p_1 and p_2. Each join edge in a MUST pattern creates a join expression connecting two path expressions. On the other hand, a *non-join expression* $p\ op\ v$ expresses a filtering condition on a single document source. Note that op and v are represented by a `VNode` in a MUST pattern. Note that a join expression can also be expressed in a `for` clause using qualifier. In this paper, we ignore join expressions in the `for` clause, which can always be reformulated away using `where` clause. Finally, an *output expression* r in the `return` clause is of type `RNode`.

Definition 1 ***[XQuery Representation of Multi-source Twig]*** *Let var be the name of variable binding, exp be a path expression, $op \in \{=, \neq, >, \geq, <, \leq\}$ be an operator, and val be a value. Given an expression exp, the function source(exp) maps exp to the document source D over which exp is valid. Then, an* ***XQuery*** *query $Q = (\mathcal{F}, \mathcal{L}, \mathcal{W}, \mathcal{R})$ is a multi-source twig query if the followings are true.*

[1] We consider XPath navigation only along the `child` (`/`) and `attribute` (`/@`) axes. Extension to other navigation axis is orthogonal to the proposed technique.

[2] Note that our model can be trivially extended to support join between same sources.

[3] We only supports equality join condition but inequality join condition can be supported trivially.

[4] For clarity, we shall add the XQuery `doc` function as prefix to a path expression in a `for` clause whenever appropriate.

- $\mathcal{F}$ *is a set of* `for` *clause items such that* $|\mathcal{F}| \geq 2$. *An item* $f \in \mathcal{F}$ *is a triple (var, dsName, exp), where source(exp) = dsName. Furthermore,* $\exists\, f_i \in \mathcal{F} \wedge f_j \in \mathcal{F}$ *such that* $f_i.dsName \neq f_j.dsName$ *for* $i \neq j$ *and* $1 < i, j \leq |\mathcal{F}|$.
- $\mathcal{L}$ *is a set of* `let` *clause items where* $l \in \mathcal{L}$ *is a 2-tuple (var, exp).*
- *Let S and T be path expressions containing* $var = f.var$ *or* $var = l.var$ *where* $f \in \mathcal{F}$, $l \in \mathcal{L}$. *Then,* $\mathcal{W}$ *is a set of conjunctive predicates in the* `where` *clause where* $\mathcal{W} = \mathcal{J} \cup \mathcal{C}$ *and* $\mathcal{J} \cap \mathcal{C} = \emptyset$. $\mathcal{J}$ *is a non-empty set of join expressions where* $b \in \mathcal{J}$ *is of the form* $S\ op\ T$. $\mathcal{C}$ *is a set of non-join expressions where* $c \in \mathcal{C}$ *is of the form* $S\ op\ val$.
- $\mathcal{R}$ *is a set of output expressions in the* `return` *clause. Each output expression r is a 2-tuple (var, exp) where* $var = f.var$ *or* $var = l.var$, $f \in \mathcal{F}$, *and* $l \in \mathcal{L}$. □

Example 1 The query Q_3 in Figure 2 consists of the followings.

- $\mathcal{F} = \{f_1, f_2\}$ where $f_1.var$ = *"$entry"*, $f_2.var$ = *"$ip"*, $f_1.dsName$ = "`Uniprot`", $f_2.dsName$ = "`InterPro`", $f_1.exp$ = "`/uniprot/entry`", and $f_2.exp$ = "`/interprodb/interpro`".
- $\mathcal{L} = \{l_1\}$ where $l_1.var$ = *"$id"* and $l_1.exp$ = "`$entry/dbReference/@id`".
- In the `where` clause, $\mathcal{J} = \{b_1\}$ and $\mathcal{C} = \{c_1, c_2, c_3, c_4\}$ where b_1 = ("`$ip/@id`", "=", "`$id`"), c_1 =("`$entry/organism/name`", " = ","`HUMAN`"), c_2 = ("`$entry/@created`", "=", "`2001`"), c_3 =("`$ip/pub_list/publication/journal`", "=", "`Structure`"), and c_4 =("`$ip/pub_list/publication/year`", "=", "`2002`").
- $\mathcal{R} = \{r_1, r_2\}$ where r_1 = ("*$entry*", "`/name`") and r_2 = ("*$ip*","`/name`") ∎

3.3 Star Twig Pattern

An XQuery representation of a multi-source twig query can be conveniently represented using an *XQuery graph*. Similar to a query graph of an SQL query, an XQuery graph is an undirected graph with nodes $D_1 \ldots D_n$. For every join expression between the document sources D_i and D_j, we add an edge between D_i and D_j. This edge is labeled by the join expression. The nodes are labeled with corresponding non-join expressions.

An XQuery graphs can have many different shapes such as chain queries, star queries, tree queries, cyclic queries, clique queries, etc. Note that these classes are not disjoint and that some classes are subsets of other classes. In this paper, *we focus on star queries joining different* XML *documents*. Intuitively, in a *multi-source star twig* query all join expressions share a common document source and hence forms a star-shaped query graph. For example, queries in Figure 2 are examples of star twig queries. Formally, it is defined as follows.

Algorithm 1: The ***StarTwig2SQL*** algorithm.

Input: Star twig query Q
Output: A list of SQL queries *SQLList*
1 Initialize $SQLList = \emptyset$;
2 $(\mathcal{F}, \mathcal{W}, \mathcal{R}) \leftarrow$ **parseXQuery**(Q) /* Phase 1*/;
3 $SQLList$.**add**(**outputExp2SQL**($\mathcal{R}$)) /* Phase 2 */;
4 $(\mathcal{J}, C) \leftarrow$ **distinguishExp**(W) /* Phase 3 */;
5 $SQLList$.**add**(**whereExp2SQL**($\mathcal{F}, \mathcal{J}, C, \mathcal{R}$));
6 $SQLList$.**add**(**finalResultQueryGen**($\mathcal{R}$)) /* Phase 4 */ ;
7 **return** *SQLList*

Definition 2 ***[Multi-source Star Twig Query]*** *Let $Q = (\mathcal{F}, \mathcal{L}, \mathcal{W}, \mathcal{R})$ be a multi-source XQuery query. Then Q is called a* ***multi-source star twig*** *query if any one of the following conditions is true: (a) $|\mathcal{J}| = 1$ and $source(b.S) \neq source(b.T)$ where $b \in \mathcal{J}$. (b) If $|\mathcal{J}| > 1$ then $\forall\ i \neq j$ $source(b_i.S) = source(b_j.S)$ and $source(b_i.S) \neq source(b_i.T)$ where $b_i \in \mathcal{J}$, $b_j \in \mathcal{J}$ and $1 \leq i, j \leq |\mathcal{J}|$.* □

4 Star Twig Query Evaluation

In this section, we shall elaborate on the algorithm for translating a star twig query to a list of SQL queries over relational framework. State-of-the-art relational approaches for XML storage can be broadly classified into four types, namely, *node* approach, *edge* approach, *path materialization (PM)* approach, and DTD approach [8]. For the sake of generality, in this paper we assume that the XML data are schemaless. Since the PM approach has advantages over the rest when XML data are schemaless [8], our proposed algorithm is built on top of this storage approach. Importantly, we present a generic algorithm that is independent of any specific PM approach. We assume that paths, contents of leaf nodes, and attributes associated with a XML tree are materialized in Paths, PathsContent, and Attributes relations, respectively. In the next section, we shall give an example of how various subroutines in the algorithm can be realized on a specific PM approach.

The algorithm for SQL translation is shown in Algorithm 1. It consists of four phases: the *XQuery parsing* phase, the *OutputExp2SQL translation* phase, the *WhereExp2SQL translation* phase, and the *Final results generator* phase. We shall elaborate on these phases in turn. In the sequel, we shall use the query Q_3 in Figure 2 as our running example to facilitate our discussion.

4.1 Phase 1: XQuery Parsing

In the first phase, a multi-source star twig query Q is parsed using XPath 2.0/XQuery 1.0 Parser Build [1] (Line 02). During the parsing process, the algorithm identifies different components of Q based on the star twig query model discussed in the preceding section. Also, the algorithm replaces the variable references in

Algorithm 2: The ***outputExp2SQL*** algorithm.

Input: A set of output expressions $\mathcal{R}$
Output: A list of SQL queries *SQLList*

```
01 Initialize SQLList = ∅;
02 for (each r ∈ R) do
03     Initialize _SQL = ∞;
04     if (r is an attribute node) then
05         PathExp ← pathExpOfParentNode(r);
06     else
07         PathExp ← r.absExp;
08     PathIDs ← getAllPathID(PathExp);
09     Level ← getNodeLevel(PathExp);
10     Source = r.dsName;
11     _SQL.genSQL(PathIDs, Level, Source);
12     Add _SQL into SQLList;
13 return SQLList
```

Q with the expressions defined in the `let` clause (if any). The output of this phase are a set of `for` clause items $\mathcal{F}$, a set of *where-expressions* $\mathcal{W}$, and a set of output expressions $\mathcal{R}$.In addition, we also determine the absolute path expressions of $r \in \mathcal{R}$, $c \in \mathcal{C}$, and $b \in \mathcal{J}$. The absolute path expression of r is denoted by $r.absExp$. For example, consider r_1 = (*$entry*, "`/name`") in Q_3. Then $r_1.absExp$ is "`/uniprot/entry/name`" as *$entry* is bound to the expression "`/uniprot/entry`".

4.2 Phase 2: OutputExp2SQL Translation

In this phase, the algorithm analyzes each output expression $r \in \mathcal{R}$ and generates an SQL query for materializing the *identifiers* of the XML subtrees that satisfy r (Line 03). An *identifier* of a node n in an XML tree D (denoted by *nId*) is one or more attributes of n that can uniquely identify n in D. The materialized identifiers of r are stored in a temporary relation PathU(DocId, nId). Note that we materialize the identifiers instead of entire subtrees because it is more space-efficient (the size of materialized identifier table is always smaller than or equal to the table containing entire materialized subtrees). Also, we do not need to materialize the level of r *explicitly* as it can be computed on-the-fly in a PM-based storage approach. It is worth mentioning that the identifier scheme is not tightly coupled to any specific numbering scheme as any scheme that can uniquely identify nodes in an XML tree can be used as an identifier. For instance, the *preorder* and *dewey order* values of nodes can be used for *region encoding* and *dewey number-based* labeling schemes, respectively [8].

Given a set of output expressions $\mathcal{R}$, the *outputExp2SQL* algorithm depicted in Algorithm 2 works as follows. For each output expression $r \in \mathcal{R}$, the algorithm

Algorithm 3: The ***whereExp2SQL*** algorithm.

Input: $\mathcal{F}, \mathcal{J}, C, \mathcal{R}$
Output: A list of SQL queries *SQLList*

```
01 Initialize SQLList = ∅;
02 for (each r ∈ R) do
03     for (each f ∈ F) do
04         C_f ← getNonJoinExp(f.var, C);
05         if (f.var = r.var) then
06             SQL ← translateWhereNonJoin(r, f, C_f);
07         else
08             SQL ← translateWhereJoin(r, f, C_f, J);
09         SQL ← INSERT INTO T_" +R.indexOf(r)+ "_"+F.indexOf(f)+ " "
           +SQL;
10         SQLList.add(SQL);
11 return SQLList
```

first determines whether r involves an attribute node (Line 04). If it does, then the algorithm retrieves the absolute path expression of its parent node (Line 05). Otherwise, the absolute path expression of r is used (Line 07). This expression is stored in the variable *PathExp*. Based on *PathExp*, a set of path ids is retrieved from the Paths table (Line 08). Also, the algorithm computes the node level of r using *PathExp*. Then the SQL query for materializing nodes satisfying r (PathU table) is generated by exploiting the Paths, Attributes, and PathsContent relations.

4.3 Phase 3: WhereExp2SQL Translation

Here, we translate the *where-expression* into a list of SQL queries. The result of each SQL query is stored in a temporary table that is an instance of the relation TempTable(DocId, nId). This phase starts by distinguishing the join and non-join expressions followed by invocation of the *whereExp2SQL* algorithm (Lines 04–05, Algorithm 1). Intuitively, for each pair of output expression r and an item f of the `for` clause expressions it generates an SQL query. If r and f refer to the *same* data source D then it generates a non-join query that evaluates the conditions specified in the *where-expression* related to D. Otherwise, if r and f refer to *different* sources, namely D_1 and D_2, respectively, then a join query is generated that satisfies the join predicate(s) as well as non-join predicates on D_2.

The *whereExp2SQL* algorithm is depicted in Algorithm 3. For each $r \in \mathcal{R}$ it performs the following steps. For each $f \in \mathcal{F}$, first, it retrieves $C_f \subseteq C$, where $\forall c \in C_f\ c.var = f.var$ (Line 04). Then, it determines whether r and f are bound to the same data source by comparing *r.var* and *f.var*. If $r.var = f.var$, then join across data sources is not necessary. In this case, the algorithm will invoke the *translateWhereNonJoin* algorithm (Line 06). Otherwise, it invokes the *translate-WhereJoin* algorithm (Line 08). We shall elaborate on these algorithms later. The

Algorithm 4: The ***translateWhereNonJoin*** algorithm.

Input: An output expression r, a for clause item f, C_f
Output: An SQL query *SQL*

```
1  Initialize selectClause, fromClause, whereClause, optionClause;
2  dataS ← source(r.var);
3  for (i = 1 to |C_f|) do
4      c = C_f[i];
5      if (c is a condition on attribute) then
6          Generate SQL statements for fromClause and whereClause;
7      else
8          Add SQL statements to whereClause;
9      Add instance of PathsContent representing dataS to the fromClause;
10     if (i > 1) then
11         whereClause.add(evalTwig(c.absExp, C_f[i − 1].absExp));
12 Add instances of PathsContent to fromClause;
13 whereClause.add(evalTwig(r.absExp, c.absExp));
14 Add nId, docId to selectClause;
15 SQL = selectClause + fromClause + whereClause;
16 return SQL
```

generated SQL query is stored in a variable called *SQL*. Next, an `INSERT` statement is appended to the generated SQL query so that the results of the query can be directly stored in the temporary table. The modified *SQL* is then added to *SQLList*.

The *translateWhereNonJoin* Algorithm: Given a pair of (r, f) representing the *same* source, the *translateWhereNonJoin* algorithm (Algorithm 4) generates a non-join SQL query. For each *where-expression* $c \in C_f$, the algorithm first checks whether c is specified on an attribute. If it is, then it will add SQL statements to the `where` and `from` clauses of the translated SQL query by exploiting the Paths and Attributes relations (Line 06). These statements retrieve path ids based on $c.absExp$ satisfying the value conditions on the attributes. If c is *not* specified on an attribute then these expressions are added to the `where` clause (Line 08). If there are more than one conditions in C_f, then it represents a twig query pattern. Consequently, SQL statement for evaluating the twig pattern is added using *evalTwig* procedure (Line 10). Next, the algorithm specifies the condition between these expressions and r using *evalTwig* procedure (Line 11) as we are interested in only those nodes that satisfy the output expression. The PathU table is used for this purpose. The generated SQL query returns the identifiers of nodes satisfying r that satisfy expressions in C_f.

The *translateWhereJoin* Algorithm. Given a pair of (r, f) representing two *different* sources, the *translateWhereJoin* algorithm (Algorithm 5) generates the join query. First, the SQL fragment for evaluation of non-join conditions on the source represented by f is generated as we are interested in those joinable nodes that

Algorithm 5: The ***translateWhereJoin*** algorithm.

Input: An output expression r, a for clause item f, C_f, $\mathcal{J}$
Output: An SQL query *SQL*

```
1  Initialize selectClause, fromClause, whereClause, optionClause;
2  processExpressions(C_f);
3  i = |C_f| + 1;
4  J_f ← J.getJoinExp(f);
5  J_r ← J.getJoinExp(r);
6  if (J_f ∩ J_r = ∅) then
7      processJoinExp(J_f.getS(), J_f.getT(), i, C_f[|C_f|].absExp);
8      processJoinExp(J_r.getS(), J_r.getT(), i, C_f[|C_f|].absExp);
9  else
10     J_x = J_f ∩ J_r;
11     processJoinExp(J_x.getS(), J_x.getT(), i, C_f[|C_f|].absExp);
12 i = i + 1;
13 Add instances of PathsContent relation to fromClause;
14 whereClause.add(evalTwig(r.absExp, T.absExp));
15 Add nId, docId to selectClause;
16 SQL = selectClause + fromClause + whereClause;
17 return SQL
```

satisfy the predicates on this source. The steps for this are encapsulated in the *processExpression* function and are same as the ones in Lines 03–11 of Algorithm 4. The next step is to general the SQL fragment for the join expressions (Lines 04–11). First, the algorithm creates two subsets of $\mathcal{J}$, namely $\mathcal{J}_f$ and $\mathcal{J}_r$, containing sets of join expressions involving the sources of f and r, respectively (Lines 04–05). If $(\mathcal{J}_f \cap \mathcal{J}_r = \emptyset)$, then the algorithm processes each of the join expressions by invoking the *processJoinExp* algorithm twice (Lines 07–10). The functions **getS** and **getT** return the S and T components of a join expression $S\ op\ T$, respectively (see Definition 1). Let us elaborate on this scenario with an example. Consider a query Q that contains $f.var \in \{f_1, f_2, f_3\}$. Let $\mathcal{J}$ in Q contains two join expressions, namely $S_1 = T$ and $S_2 = T$ where S_1, S_2, and T are path expressions representing three different data sources and contain $f_2.var$, $f_3.var$, and $f_1.var$, respectively. Let $\mathcal{R} = \{r\}$ where $r.var = f_3.var$. Now consider the pair (r, f_2) in the context of Algorithm 5. Here $\mathcal{J}_f = \{\text{“}S_1 = T\text{”}\}$ and $\mathcal{J}_r = \{\text{“}S_2 = T\text{”}\}$. Since $\mathcal{J}_f \cap \mathcal{J}_r = \emptyset$, Lines 07–08 are executed. In this case, the algorithm processes the join between S_1 and T first followed by the join between S_2 and T. Note that there is no join expression of the form “$S_1 = S_2$”. On the other hand, if $(\mathcal{J}_f \cap \mathcal{J}_r \neq \emptyset)$, then the algorithm will retrieve the common join expressions (denoted by $\mathcal{J}_x$) between $\mathcal{J}_f$ and $\mathcal{J}_r$, and process them by invoking the *processJoinExp* procedure (Lines 11). To elaborate further, consider the pair (r, f_1) in the context of the above example. Here $\mathcal{J}_f = \mathcal{J}_r = \{\text{“}S_1 = T\text{”}\}$. Hence, Line 11 is executed. The objective of *processJoinExp* procedure is to generate the SQL fragments involving the join expressions. For each join expression $S\ op\ T$, it checks the type of node (attribute or element) in

S and T and corresponding SQL fragments are added to `where` and `from` clauses. Lastly, Algorithm 5 evaluates the twig fragment consisting of the join expression and the output expression r using *evaluateTwig* procedure. Note that this procedure is similar to the one discussed in the context of *TranslateWhereNonJoin* algorithm.

Example 2 Consider the query Q_3 in Example 1. Consider the scenario when r_1 = "`$entry/name`" and f_1 = "*$entry*". In this case, the set of non-join expressions are C_f = {`$entry/organism/name = 'HUMAN'`, `$entry/@created = '2001'`} (Line 4, Algorithm 3). Since the document sources of r_1 and f_1 are identical (*$entry*), the algorithm concludes that no join is needed and invokes the *translateWhereNonJoin* algorithm. When f_2 = "*$ip*", then C_f = {`$ip/pub_list /publication/journal ='Structure'`, `$ip/pub_list /publication/ year ='2002'`}. As the data sources of r_1 and f_2 are now different, join is required. In this case, b = "`$ip/@id = $entry/dbReference/@id`". Consequently, the algorithm invokes the *translateWhereJoin* procedure.

Similarly, the above steps are repeated for r_2 = "`$ip/name`". Now the data sources are different for f_1. Hence, join operation is needed using the join expression b. As a result, the algorithm invokes *translateWhereJoin*. Lastly, for f_2 the sources of r_2 and f_2 are identical. Consequently, *translateWhereNonJoin* is invoked. ∎

4.4 Phase 4: Final Results Generator

Finally, this phase generates a set of SQL queries for retrieving the final results in two steps. The first step is to combine the results of SQL queries generated in Phase 3 (Line 02). Note that the results of these queries can be combined by performing intersection operation over them. The results of the SQL queries generated in this step are sets of *identifiers* satisfying the output expression r stored in the PathUFinal(DocId, nId) table. In the second step the algorithm retrieves *complete* information related to these nodes (remaining attributes in PathsContent) for generating the final result. Specifically, it generates an SQL query by joining the PathUFinal and PathsContent tables. The results are sorted in document order.

Theorem 1 *Let $Q = (\mathcal{F}, \mathcal{L}, \mathcal{W}, \mathcal{R})$ be a multi-source star twig query involving n different data sources. Let p be the number of output expressions in $\mathcal{R}$ that do not contain attribute nodes. Let q be the number of output expressions in $\mathcal{R}$ that contain attribute nodes. Then, the total number of* SQL *queries generated from Q is $(n+3)p + (n+4)q$.*

Proof 1 *In Phase 2, the algorithm generates $(p+q)$* SQL *queries. Note that $|\mathcal{R}| = p+q$. Furthermore, $(n \times (p+q))$* SQL *queries are generated in Phase 3. The last phase results $(2p+3q)$* SQL *queries. The total number of* SQL *queries generated from Q is $(p+q) + (np+nq) + (2p+3q) = (n+3)p + (n+4)q$.* □

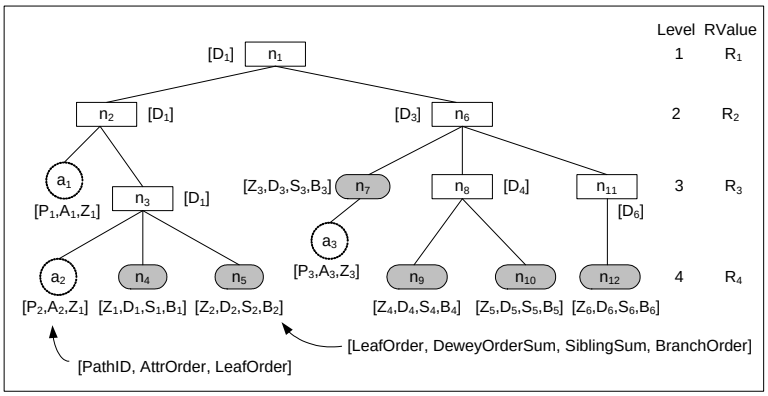


Figure 4: Encoding scheme of SUCXENT++.

5 Implementing Star Twig Query Evaluation on SUCXENT++

In this section, we shall elaborate on how our proposed star twig query evaluation algorithm introduced in the preceding section is implemented on a path materialization-based (PM) XML storage scheme. We begin by briefly describing a PM-based storage scheme called SUCXENT++ [2, 18], which we shall be using as the framework for evaluating star twig queries.

5.1 SUCXENT++

We begin by providing an overview of the encoding scheme used in SUCXENT++ for XML trees. Note that the scheme does not require a relational back-end to support SQL/XML standard or XML data type. Consider Figure 4. Each level ℓ of a XML tree is associated with an attribute called RValue (denoted as R_ℓ). Each leaf element node n is associated with four attributes, namely LeafOrder, BranchOrder, DeweyOrderSum, and SiblingSum. Each non-leaf element node n' is *implicitly* assigned the DeweyOrderSum of the first descendant leaf node for reasons discussed later. Each attribute node (denoted as a_i) is associated with AttrOrder, LeafOrder of its parent node, and its PathId. We now elaborate on these attributes in the context of the relational schema of SUCXENT++.

The schema of SUCXENT++ [2, 18] is as follows.

- Document(Docid, Name)
- Path(PathId, PathExp)
- PathValue(DocId, DeweyOrderSum, PathId, BranchOrder, LeafOrder, SiblingSum, LeafValue)
- Attribute(DocId, LeafOrder, PathId, LeafValue, AttrOrder)
- DocumentRValue(DocId, Level, RValue)

Document stores the document identifier Docid and the name Name of a given input XML document D. We associate each distinct root-to-leaf path appearing in D,

Uniprot_DocumentRValue

DocID	Level	RValue
1	1	4
1	2	1
1	3	1

Uniprot_Path

PathID	PathExp
1	.uniprot#.entry#.@created#
2	.uniprot#.entry#.name#
3	.uniprot#.entry#.organism#.name#
4	.uniprot#.entry#.dbReference#.@type#
5	.uniprot#.entry#.dbReference#.@id#
6	.uniprot#.entry#.dbReference#

Uniprot_Attribute

DocID	Leaf Order	PathID	Attr Order	LeafValue
1	1	1	1	1991
1	3	4	2	InterPro
1	3	5	3	IPR710
1	4	4	4	EMBL
1	4	5	5	X14312
1	5	1	6	2001
1	7	4	7	PDB
1	7	5	8	1BV8
1	8	4	9	InterPro
1	8	5	10	IPR123
1	9	1	11	2001
1	11	4	12	H-InvDB
1	11	5	13	HIX002
1	12	4	14	KEGG
1	12	5	15	hsa:2

Uniprot_PathValue

DocID	Leaf Order	Branch Order	PathID	Dewey Order Sum	Sibling Sum	LeafValue
1	1	0	2	0	0	12S1_ARATH
1	2	2	3	1	0	HUMAN
1	3	2	6	2	0	NULL
1	4	2	6	3	1	NULL
1	5	1	2	7	7	14332_ORYSJ
1	6	2	3	8	7	HUMAN
1	7	2	6	9	7	NULL
1	8	2	6	10	8	NULL
1	9	1	2	14	14	A2MG_HUMAN
1	10	2	3	15	14	HUMAN
1	11	2	6	16	14	NULL
1	12	2	6	17	15	NULL

Figure 5: Storage of shredded XML document in Figure 1(a).

Interpro_DocumentRValue

DocID	Level	RValue
1	1	77
1	2	10
1	3	2
1	4	1

Interpro_Attribute

DocID	LeafOrder	PathID	AttrOrder	LeafValue
1	1	1	1	IPR100
1	6	1	2	IPR123
1	9	1	3	IPR500
1	14	1	4	IPR710

Interpro_Path

PathID	PathExp
1	.interprodb#.interpro#.@id#
2	.interprodb#.interpro#.name#
3	.interprodb#.interpro#.pub_list#.publication#.journal#
4	.interprodb#.interpro#.pub_list#.publication#.year#

Interpro_PathValue

DocID	Leaf Order	Brach Order	PathID	Dewey Order Sum	Sibling Sum	LeafValue
1	1	0	2	0	0	Kringle
1	2	2	3	19	0	J. Mol. Evol.
1	3	4	4	20	0	1987
1	4	3	3	22	3	Structure
1	5	4	4	23	3	2002
1	6	1	2	153	153	PAS
1	7	2	3	172	153	Structure
1	8	4	4	173	153	2002
1	9	1	2	306	306	P2Y4 purinoceptor
1	10	2	3	325	306	Protein Eng.
1	11	4	4	326	306	1994
1	12	3	3	328	309	Cell. Signal
1	13	4	4	329	309	2000
1	14	1	2	459	459	Carboxyl transferase
1	15	2	3	478	459	Nature
1	16	4	4	479	459	2001
1	17	3	3	481	462	Science
1	18	4	4	482	462	2000

Figure 6: Storage of shredded XML document in Figure 1(b).

namely PathExp, with an identifier PathId and store this information in Path table. Essentially each path is a concatenation of the labels of the elements in the path from the root to the leaf. An example of the Path table containing the root-to-leaf paths of Figure 1(b) is shown in Figure 6. Note that we use '#' as a delimiter of steps in the paths instead of '/' for reasons described in [22].

For each element leaf node n in D, we shall create a tuple in the PathValue table which stores the LeafOrder, BranchOrder, DeweyOrderSum, and SiblingSum values of n. The data value of n is stored in LeafValue. We now elaborate on these attributes. Given two element leaf nodes n_1 and n_2, n_1.LeafOrder $< n_2$.LeafOrder *iff* n_1 precedes n_2 in document order. LeafOrder of the first leaf node in D is 1 and n_2.LeafOrder = n_1.LeafOrder+1 *iff* n_1 is a leaf node immediately preceding n_2. For example, the superscript of each leaf element node in Figure 1 denotes its LeafOrder value. Given two element leaf nodes n_1 and n_2 where n_1.LeafOrder+1 = n_2.LeafOrder, n_2.BranchOrder is the level of the nearest common ancestor (NCA) of n_1 and n_2. Note that the BranchOrder of the first leaf node is 0.

The BranchOrder has an interesting property. Let s be a non-leaf node at level

ℓ. Let n_1 be the first descendant leaf node of s. Then, except for n_1, BranchOrder values of all the descendant leaf nodes of s are at least ℓ. The BranchOrder of n_1 is less than ℓ. Observe that the NCA of n_1 and its immediately preceding leaf node is not a descendant of s. This property will be exploited later to implicitly encode the non-leaf nodes.

Each level ℓ of an XML tree is associated with an attribute called RValue (denoted as R_ℓ). We first introduce the notion of *maximal k-consecutive leaf-node list* which is used to define RValue. Consider a list of consecutive leaf node $\mathcal{S}$: $[n_1, n_2, n_3, \ldots, n_r]$ in D. Let $k \in [1, L_{max}]$ where L_{max} is the largest level of D. Then, $\mathcal{S}$ is called a *k-consecutive leaf-node list* of D *iff* $\forall 0 < i \leq r$ n_i.BranchOrder $\geq k$. $\mathcal{S}$ is called a *maximal* k-consecutive leaf-node list, denoted as M_k, if there does not exist a k-consecutive leaf-node list $\mathcal{S}'$ such that $|\mathcal{S}| < |\mathcal{S}'|$. For example, M_2 in Figure 1(b) contains four leaf nodes as $|\mathcal{S}| = 4$ for M_2.

The RValue of level ℓ, denoted as R_ℓ, is defined as follows: (i) If $\ell = L_{max} - 1$ then $R_\ell = 1$; (ii) If $0 < \ell < L_{max} - 1$ then $R_\ell = 2R_{\ell+1} \times |M_{\ell+1}| + 1$. For example, consider Figure 1(b). Here $L_{max} = 5$. The values of $|M_1|$, $|M_2|$, $|M_3|$, and $|M_4|$ are 17, 4, 3, and 1, respectively. Then, $R_4 = 1$, $R_3 = 2 \times 1 \times |M_4| + 1 = 3$, $R_2 = 2 \times 3 \times |M_3| + 1 = 19$, and $R_1 = 2 \times 19 \times |M_2| + 1 = 153$. In order to facilitate evaluation of XPath queries, the RValue attribute in DocumentRValue stores $\frac{R_\ell - 1}{2} + 1$ instead of R_ℓ (denoted as R'_ℓ). For instance, in Figure 6 the RValue of level 1 is stored as 77 instead of 153.

DeweyOrderSum is used to encode a node's order information together with its ancestors' order information using a single value. Let $parent(w)$ denote the parent of an node w. Consider a leaf node n at level ℓ in D. Then, for $1 < k \leq \ell$, Ord(n, k) = i *iff* (i) there exists an node a at level k which is either an ancestor of n or n itself; and (ii) a is the i-th child of $parent(a)$. For example, consider the rightmost `year` leaf node in Figure 1(b) (denoted as d). Ord$(d, 2) = 4$ as the node b in the second level is an ancestor of d as well as the fourth child of the root. Similarly, Ord$(d, 3)$ = 2.

Then DeweyOrderSum of n, n.DeweyOrderSum, is defined as $\sum_{j=2}^{\ell} \Phi(j)$ where $\Phi(j)$=[Ord(n, j)-1]$\times R_{j-1}$. The DeweyOrderSum of the first leaf node is 0. Consider the rightmost leaf node in Figure 1(b). It has a Dewey path "1.4.2.2.2". DeweyOrderSum of this node is: n.DeweyOrderSum = $(Ord(n, 2) - 1) \times R_1 + (Ord(n, 3) - 1) \times R_2 + (Ord(n, 4) - 1) \times R_3 + (Ord(n, 5) - 1) \times R_4 = 3 \times 153 + 1 \times 19 + 1 \times 3 + 1 \times 1 = 482$. The DeweyOrderSum of remaining nodes are shown in the DeweyOrderSum attribute of the PathValue table in Figure 6. The SiblingSum is used to compute position-based predicates with name tests. We do not elaborate further as it is beyond the scope of the paper.

Comparison of ordering of non-leaf nodes: SUCXENT++'s strategy for comparing the order of non-leaf nodes is based on the following observation. If node n_0 precedes (resp. follows) another node n_1, then descendants of n_0 must also precede (resp. follow) the descendants of n_1. Therefore, instead of comparing the order between non-leaf nodes, the order between *their descendant leaf nodes* is compared. For this reason, the first descendant leaf node of a non-leaf node n is defined as

the *representative leaf node* of n. Here the property of BranchOrder as discussed earlier is exploited to identify the first descendant leaf node. The DeweyOrderSum of the representative leaf node is *conceptually* propagated to its ancestor non-leaf nodes. Hence, each non-leaf node is *implicitly* assigned the DeweyOrderSum of the first descendant leaf node. Note that these values are not stored explicitly in SUCXENT++ as they can be retrieved from the DeweyOrderSum of representative leaf nodes.

Lastly, the Attribute table stores the attribute nodes. As a non-leaf node can be represented by the first descendant leaf nodes, an attribute node is identified by DocId and LeafOrder of parent node and its PathId. In addition to these attributes, we store another attribute called AttrOrder to encode the total ordering of the attributes in the entire XML document. Given two attribute nodes n_1 and n_2, n_1.AttrOrder $< n_2$.AttrOrder *iff* n_1 precedes n_2 in document order. AttrOrder of the first attribute node is 1 and n_2.AttrOrder = n_1.AttrOrder+1 *iff* n_1 is an attribute node immediately preceding n_2.

Figure 6 depicts an example of storage of XML representation of INTERPRO data source (Figure 1(b)) in SUCXENT++. Note that for each data source, we create an instance of the schema. We store large text content (e.g., protein sequences) in a separate relation called TextContent that has same schema as PathValue.

Computation of NCA. We now briefly discuss how to compute the NCA of two nodes efficiently in SUCXENT++ using the following lemma and theorem.

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

Theorem 2 *Let n_1 and n_2 be two distinct leaf nodes in an XML tree and $\ell > 0$. If* $\frac{R_{\ell+1}-1}{2} + 1 \leq |n_1$.DeweyOrderSum - n_2.DeweyOrderSum$| < \frac{R_\ell - 1}{2} + 1$ *then the level of the NCA of n_1 and n_2 is $\ell + 1$.* □

The reader may refer to [2] for proofs. Consider the last leaf node in Figure 1(b) (denoted by d). The DeweyOrderSum of this node is 482. Let X be the DeweyOrderSum of leaf nodes that have NCA at level 4. Using the above theorem, X falls within the following range: $(R_4 - 1)/2 + 1 \leq |X - 482| < (R_3 - 1)/2 + 1 \Rightarrow 1 \leq |X - 482| < 2$ which returns the 17^{th} leaf node (DeweyOrderSum is 481). Similarly, when the NCA is at level 3 the 15^{th} and 16^{th} nodes are returned as they satisfy $(R_3 - 1)/2 + 1 \leq |X - 482| < (R_2 - 1)/2 + 1 \Rightarrow 2 \leq |X - 482| < 10$. Note that Theorem 2 can also be used for internal nodes as SUCXENT++ represents each internal node with its first descendant leaf node.

Corollary 1 *Let n_1 and n_2 be two distinct leaf nodes in an XML tree. If* $|n_1$.DeweyOrderSum - n_2.DeweyOrderSum$| \geq \frac{R_1 - 1}{2} + 1$ *then the NCA of n_1 and n_2 is the root node.* □

For example, consider the leaf nodes with LeafOrder values 2 and 6 (in Figure 1(b)) and DeweyOrderSums 19 and 153, respectively. Since $153 - 19 > 77$, the NCA is the root node (`interprodb`).

```
01 INSERT INTO PathU_1
02 SELECT DOCID, DEWEYORDERSUM
   FROM UNIPROT_PATHVALUE
03 WHERE PATHID IN (2)
04   AND BRANCHORDER < 3;
05 INSERT INTO PathU_2
06 SELECT DOCID, DEWEYORDERSUM
   FROM INTERPRO_PATHVALUE
07 WHERE PATHID IN (2)
08   AND BRANCHORDER < 3;
```

(a) Phase 2: Generate SQL Queries

```
01 INSERT INTO T_1_1
02 SELECT DISTINCT V3.DOCID, V3.DEWEYORDERSUM
03 FROM  UNIPROT_ATTRIBUTE A1, UNIPROT_PathValue V1, UNIPROT_PathValue V2, PathU_1 V3
04 WHERE A1.PATHID IN (1) AND A1.LEAFVALUE = '2001'
05   AND A1.LEAFORDER = V1.LEAFORDER
06   AND V2.PATHID IN (3) AND V2.LEAFVALUE = 'HUMAN'
07   AND V2.DeweyOrderSum BETWEEN V1.DeweyOrderSum –
         CAST(4 as BIGINT) + 1 AND V1.DeweyOrderSum + CAST(4 as BIGINT) – 1
08   AND V3.DeweyOrderSum BETWEEN V2.DeweyOrderSum –
         CAST(4 as BIGINT) + 1 AND V2.DeweyOrderSum + CAST(4 as BIGINT) – 1
09 OPTION (FORCE ORDER);
```

(b) Phase 3: Generate SQL Query (1)

```
01 INSERT INTO T_1_2
02 SELECT DISTINCT V5.DOCID, V5.DEWEYORDERSUM
03 FROM INTERPRO_PATHVALUE V1, INTERPRO_PathValue V2,
       INTERPRO_PathValue V3, INTERPRO_ATTRIBUTE A3,
       UNIPROT_ATTRIBUTE A4, UNIPROT_PathValue V4,
       PathU_1 V5
04 WHERE V1.PATHID IN (3) AND V1.LEAFVALUE = 'Structure'
05   AND V2.PATHID IN (4) AND V2.LEAFVALUE = '2002'
06   AND V2.DeweyOrderSum BETWEEN
            V1.DeweyOrderSum - CAST(10 as BIGINT) + 1
         AND V1.DeweyOrderSum + CAST(10 as BIGINT) – 1
07   AND V3.DeweyOrderSum BETWEEN
            V2.DeweyOrderSum - CAST(77 as BIGINT) + 1
         AND V2.DeweyOrderSum + CAST(77 as BIGINT) – 1
08   AND A3.LEAFORDER = V3.LEAFORDER AND A3.PATHID IN (1)
09   AND A3.LEAFVALUE = A4.LEAFVALUE AND A4.PATHID IN (5)
10   AND A4.LEAFORDER = V4.LEAFORDER
11   AND V5.DeweyOrderSum BETWEEN
            V4.DeweyOrderSum - CAST(4 as BIGINT) + 1
         AND V4.DeweyOrderSum + CAST(4 as BIGINT) – 1
12 OPTION (FORCE ORDER);
```

(c) Phase 3: Generate SQL Query (2)

```
01 INSERT INTO T_2_1
02 SELECT DISTINCT V5.DOCID, V5.DEWEYORDERSUM
03 FROM UNIPROT_ATTRIBUTE A1, UNIPROT_PATHVALUE V1,
        UNIPROT_PathValue V2, UNIPROT_PathValue V3,
        UNIPROT_ATTRIBUTE A3, INTERPRO_ATTRIBUTE A4,
        INTERPRO_PathValue V4, PathU_2 V5
04 WHERE A1.PATHID IN (1) AND A1.LEAFVALUE = '2001'
05   AND A1.LEAFORDER = V1.LEAFORDER
06   AND V2.PATHID IN (3) AND V2.LEAFVALUE = 'HUMAN'
07   AND V2.DeweyOrderSum BETWEEN
            V1.DeweyOrderSum - CAST(4 as BIGINT) + 1 AND
            V1.DeweyOrderSum + CAST(4 as BIGINT) – 1
08   AND V3.DeweyOrderSum BETWEEN
            V2.DeweyOrderSum - CAST(4 as BIGINT) + 1 AND
            V2.DeweyOrderSum + CAST(4 as BIGINT) – 1
09   AND A3.LEAFORDER = V3.LEAFORDER AND A3.PATHID IN (5)
10   AND A3.LEAFVALUE = A4.LEAFVALUE AND A4.PATHID IN (1)
11   AND A4.LEAFORDER = V4.LEAFORDER
12   AND V5.DeweyOrderSum BETWEEN
            V4.DeweyOrderSum - CAST(77 as BIGINT) + 1 AND
            V4.DeweyOrderSum + CAST(77 as BIGINT) – 1
13 OPTION (FORCE ORDER);
```

(d) Phase 3: Generate SQL Query (3)

```
01 INSERT INTO T_2_2
02 SELECT DISTINCT V3.DOCID, V3.DEWEYORDERSUM
03 FROM INTERPRO_PATHVALUE V1, INTERPRO_PathValue V2, PathU_2 V3
04 WHERE V1.PATHID IN (3) AND V1.LEAFVALUE = 'Structure' AND V2.PATHID IN (4) AND V2.LEAFVALUE = '2002'
05   AND V2.DeweyOrderSum BETWEEN V1.DeweyOrderSum - CAST(10 as BIGINT) + 1 AND
         V1.DeweyOrderSum + CAST(10 as BIGINT) – 1
06   AND V3.DeweyOrderSum BETWEEN V2.DeweyOrderSum - CAST(77 as BIGINT) + 1 AND
         V2.DeweyOrderSum + CAST(77 as BIGINT) – 1
07 OPTION (FORCE ORDER);
```

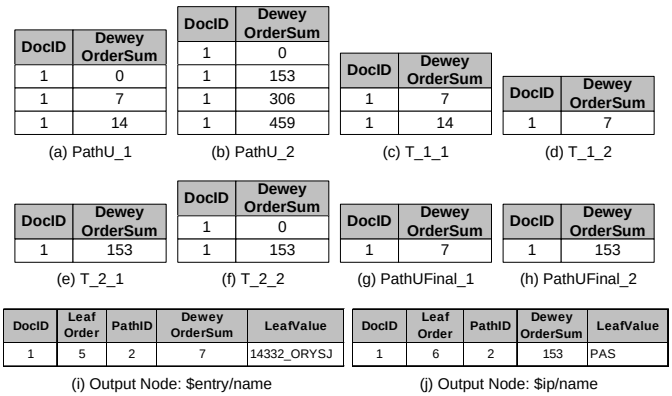
(e) Phase 3: Generate SQL Query (4)

Figure 7: Generated SQL queries.

Note that the above theorem and corollary involve non-identical elements. When the pair of elements are identical, then the NCA is computed as follows. (a) If n_1 and n_2 are non-leaf elements and their representative leaf elements are identical, then the level of the NCA of n_1 and n_2 is $MIN(level(n_1), level(n_2))$. (b) Suppose that n_1 is a non-leaf element and n_2 is a leaf element. If the representative leaf element of n_1 is identical to n_2, then the level of the NCA of these elements is the level of n_1. (c) If n_1 and n_2 are identical leaf elements then the NCA level is the level of n_1 or n_2.

5.2 Translation to A List of SQL

We now describe how the list of SQL queries are generated in SUCXENT++. Specifically, we focus on Phases 2 to 4 of our proposed algorithm. Note that the Paths, PathsContent, and Attributes relations introduced in Section 4 refer to Path, PathValue, and Attribute relations in SUCXENT++, respectively. In the sequel, we preprocess the PathId and RValue to reduce the number of joins in the translated SQL queries.



(a) PathU_1

DocID	Dewey OrderSum
1	0
1	7
1	14

(b) PathU_2

DocID	Dewey OrderSum
1	0
1	153
1	306
1	459

(c) T_1_1

DocID	Dewey OrderSum
1	7
1	14

(d) T_1_2

DocID	Dewey OrderSum
1	7

(e) T_2_1

DocID	Dewey OrderSum
1	153

(f) T_2_2

DocID	Dewey OrderSum
1	0
1	153

(g) PathUFinal_1

DocID	Dewey OrderSum
1	7

(h) PathUFinal_2

DocID	Dewey OrderSum
1	153

(i) Output Node: $entry/name

DocID	Leaf Order	PathID	Dewey OrderSum	LeafValue
1	5	2	7	14332_ORYSJ

(j) Output Node: $ip/name

DocID	Leaf Order	PathID	Dewey OrderSum	LeafValue
1	6	2	153	PAS

Figure 8: The generated temporary relations and final results.

Phase 2: OutputExp2SQL Translation. Recall that in this phase, the algorithm analyzes each output expression $r \in \mathcal{R}$ and generates an SQL query for materializing the *identifiers* of the XML subtrees that satisfy r. Since the relational backend is based on SUCXENT++, for a given document we use the DeweyOrderSum attribute as the identifier of a leaf node. If r is an internal node, then the level information of r (using BranchOrder) and DeweyOrderSum attribute of its left-most descendant leaf node is used as its identifier. Recall that the BranchOrder of the left-most descendant leaf node is less than the level of r. For an attribute node, the identifier of its parent node is used. The materialized identifiers of r are stored in a temporary relation PathU(DocId, DeweyOrderSum). We do not need to materialize the level of r *explicitly* as it can be computed on-the-fly. The SQL query for materializing nodes satisfying r is generated using the following query template QT_1:

```
INSERT INTO   PATHU
  SELECT      DocID, DeweyOrderSum
  FROM        [Source]_PATHVALUE
  WHERE       PathID IN ([PathIDs])
              AND BranchOrder < [Level]
```

Note that [*param*] in QT_1 is replaced by the value of *param* from the algorithm.

Example 3 Reconsider Q_3 and Algorithm 2. We have two output expressions r_1 = {*$entry*, "/name"} and r_2 = {*$ip*, "/name"}. Consider r_1. The algorithm sets *PathExp* to "/uniprot/entry/name". As r_1 is not an attribute node, it will fetch a set of path ids from the Path table that matches *PathExp*. Consequently, *PathIDs*={2}. Here the *Level* is equal to 3. The generated SQL query for materializing nodes satisfying r_1 is shown in Lines 01–04 of Figure 7(a). Similarly, Lines 05–08 represent the query for materializing r_2. Figures 8(a) and (b) depict two tables generated by the SQL statement in Figure 7(a). Note that this phase can be efficiently evaluated as typically the size of the Path table is relatively small. ∎

Algorithm 6: The ***translateWhereNonJoin*** algorithm on SUCXENT++.

Input: An output expression r, a for clause item f, C_f
Output: An SQL query *SQL*

```
1  Initialize selectClause, fromClause, whereClause, optionClause;
2  dataS ← source(r.var);
3  for (i = 1 to |C_f|) do
4      c = C_f[i];
5      if (c is a condition on attribute) then
6          whereClause.add("A" +i + ".PathId IN ("+ getPathId(c.absExp) + ")
           ");
7          whereClause.add("A" +i + ".LeafValue" + c.op + " " + c.val);
8          whereClause.add("A" +i + ".LeafOrder = "V" + i + ".LeafOrder");
9          fromClause.add(dataS + "_Attribute A" + i);
10     else
11         whereClause.add("V" +i + ".PathId IN ("+ getPathId(c.absExp) + ")
           ");
12         whereClause.add("V" +i + ".LeafValue" + c.op + " " + c.val);
13     fromClause.add(dataS + "_PathValue V" + i);
14     if (i > 1) then
15         whereClause.add(evalTwig(c.absExp, C_f[i − 1].absExp));
16         if (more than one documents in the collection) then
17             fromClause.add(dataS + "_DocumentRValue R" + i);
18 fromClause.add("PathU_" + r + "V" + (i + 1));
19 whereClause.add(evalTwig(r.absExp, c.absExp));
20 if (more than one documents in the collection) then
21     fromClause.add(dataS + "_DocumentRValue R" + (i + 1));
22 selectClause.add("V" + (i + 1) + ".DocID, V" + (i + 1) + ".DeweyOrderSum");
23 optionClause.add("Option (Force Order)");
24 SQL = selectClause + fromClause + whereClause + optionClause;
25 return SQL
```

Phase 3: WhereExp2SQL Translation. In this phase, we translate the *where-expression* into a list of SQL queries. Recall that it consists of the *translateWhereNon-Join* and *translateWhereJoin* procedures. We now elaborate on how these two procedures are implemented on SUCXENT++.

The translateWhereNonJoin Algorithm: Given a pair of (r, f) representing the *same* source, the Algorithm 6 generates a non-join SQL query. For each *where-expression* $c \in C_f$, the algorithm first checks whether c is specified on an attribute. If it is, then it will add SQL statements to the `where` and `from` clauses of the translated SQL query (Lines 06-09). Note that the *getPathID* function in Lines 06 and 11 is used to retrieve path ids based on *c.absExp*. Line 07 specifies the condition on the LeafValue. The expression in Line 08 binds the attribute nodes to their parent nodes. If c is *not* specified on an attribute then two SQL expressions are added to the `where` clause (Lines 11-12). If there are more than one conditions in C_f, then

Algorithm 7: The ***processJoinExp*** algorithm on SUCXENT++.

Input: S and T of a join expression, table index i
Output: Updated $fromClause$, $whereClause$, and i

```
1  i = i + 1;
2  if (S.exp involves attribute node) then
3      whereClause.add("A" +i + ".PathId IN ("+ getPathId(S.exp) + ") ");
4      whereClause.add("A" +i + ".LeafOrder = "V" + i + ".LeafOrder");
5      fromClause.add(source(S) + "_Attribute A" + i);
6      fromClause.add(source(S) + "_PathValue V" + i);
7      tempExp = "A" + i + ".LeafValue";
8  else
9      whereClause.add("V" +i + ".PathId IN ("+ getPathId(S.exp) + ") ");
10     fromClause.add(source(S) + "_PathValue V" + i);
11     tempExp = "V" + i + ".LeafValue";
12 i = i + 1;
13 if (T.exp involves attribute node) then
14     whereClause.add("A" +i + ".PathId IN ("+ getPathId(T.exp) + ") ");
15     whereClause.add("A" +i + ".LeafOrder = "V" + i + ".LeafOrder");
16     whereClause.add(tempExp + .op "A" + i + ".LeafValue");
17     fromClause.add(source(T) + "_Attribute A" + i);
18     fromClause.add(source(T) + "_PathValue V" + i);
19 else
20     whereClause.add("V" +i + ".PathId IN ("+ getPathId(T.exp) + ") ");
21     whereClause.add(tempExp + .op "V" + i + ".LeafValue");
22     fromClause.add(source(T) + "_PathValue V" + i);
23 return fromClause, whereClause, i
```

it represents a twig query pattern. Consequently, Theorem 2 can be used to efficiently evaluate the twig pattern. Hence the algorithm translates Theorem 2 into corresponding SQL statements (Lines 14–17) using *evalTwig* procedure. Note that as twig evaluation is orthogonal to star query processing, we do not elaborate on this (See [2] for details).

Next, the algorithm specifies the condition between these expressions and r using *evalTwig* procedure (Lines 18–21) as we are interested in only those nodes that satisfy the output expressions. The PathU_r table is used for this purpose. The SQL query generated by the *translateWhereNonJoin* algorithm will only return the identifiers of nodes satisfying r that satisfy expressions in C_f (Line 22). Finally, Line 23 enforces the join order option because of performance benefits discussed in [9, 18].

The translateWhereJoin Algorithm. The main step in this procedure is the *processJoinExp* algorithm which is used to generate the SQL fragments involving the join expressions. Algorithm 7 outlines the implementation of this procedure on top of SUCXENT++. For each join expression $S\ op\ T$, it checks the type of node (attribute or element) in S and T (Lines 2 and 13 in Algorithm 7) and corresponding

```
01  INSERT INTO PATHUFINAL_1
02  SELECT A.* FROM
03     (SELECT * FROM T_1_1 INTERSECT  SELECT * FROM T_1_2) AS A ;
04  INSERT INTO PATHUFINAL_2
05  SELECT A.* FROM
06     (SELECT * FROM T_2_1  INTERSECT  SELECT * FROM T_2_2) AS A ;
07  SELECT V.DOCID, V.LeafOrder, V.DeweyOrderSum, V.PathID, V.LeafValue
08  FROM PATHUFINAL_1 AS P, UNIPROT_PATHVALUE V
09  WHERE V.PATHID IN (2)
10    AND V.DeweyOrderSum BETWEEN P.DeweyOrderSum - CAST(4 as BIGINT) + 1
                              AND P.DeweyOrderSum + CAST(4 as BIGINT) - 1
11  ORDER BY V.DOCID, V.DEWEYORDERSUM
12  OPTION (FORCE ORDER);
13  SELECT V.DOCID, V.LeafOrder, V.DeweyOrderSum, V.PathID, V.LeafValue
14  FROM PATHUFINAL_2 AS P, INTERPRO_PATHVALUE V
15  WHERE V.PATHID IN (2)
16    AND V.DeweyOrderSum BETWEEN P.DeweyOrderSum - CAST(77 as BIGINT) + 1
                              AND P.DeweyOrderSum + CAST(77 as BIGINT) - 1
17  ORDER BY V.DOCID, V.DEWEYORDERSUM
18  OPTION (FORCE ORDER);
```

Figure 9: Generated SQL (Phase 4).

SQL fragments are added to `where` and `from` clauses. Observe that the join between two sources is specified in Line 16.

Example 4 For example, reconsider Example 2. Consider the scenario when r_1 = "`$entry/name`" and f_1 = "*$entry*". Since in this case no join is needed, the *translateWhereNonJoin* algorithm is invoked. This returns the SQL query in Figure 7(b). When f_2 = "*$ip*", the *translateWhereJoin* function is invoked which returns the SQL query in Figure 7(c). Similarly, for r_2 = "`$ip/name`", *translateWhereJoin* and *translateWhereNonJoin* are invoked for f_1 and f_2, respectively, generating the queries in Figures 7(d) and 7(e). ∎

Phase 4: Final Results Generator. Finally, we generate a set of SQL queries for retrieving the final results of the query. In the first step we combine the results of SQL queries generated in Phase 3 by performing intersection operation over them. The results of the SQL queries generated in this step are stored in the PathUFinal_r(DocId, DeweyOrderSum) table (Figures 8(g) and 8(h)). In the second step complete information related to these nodes (remaining attributes in the PathValue table) are retrieved for generating the final result. Note that if the output expression r contains attribute nodes, then there will be two SQL queries generated for retrieving all result nodes. Otherwise, only one SQL query is generated. The resultant relations will be used to restructure the results according to the output structure. Figure 9 depicts the final SQL queries generated by Phase 4 for Q_3.

Consider Q_3 in Figure 2. Here $n = 2$, $p = 2$, and $q = 0$. Therefore, based on Theorem 1, the number of sub-queries evaluated by our approach is $(2+3)\times 2 = 10$.

QID	Query	# of Results

Q4	for $entry in fn:collection('UNIPROT')/uniprot/entry,
	    $interpro in fn:collection('INTERPRO')/interprodb/interpro
	let $ref2Interpro := $entry/dbReference[@type="InterPro"]/@id
	where $entry/keyword = 'Cell wall' and $interpro/@id = $ref2Interpro
	  and $interpro/pub_list/publication/journal = "Bioinformatics"
	return $entry/protein;
		6

Q5	for $entry in fn:collection('UNIPROT')/uniprot/entry,
	    $interpro in fn:collection('INTERPRO')/interprodb/interpro
	let $ref2Interpro := $entry/dbReference[@type="InterPro"]/@id
	where $entry/keyword = 'Vision' and $entry/organism/name = 'Human'
	  and $interpro/pub_list/publication/journal = "Nature" and $interpro/@id = $ref2Interpro
	return $entry/gene;
		31

Q6	for $entry in fn:collection('UNIPROT')/uniprot/entry,
	    $interpro in fn:collection('INTERPRO')/interprodb/interpro
	let $ref2Interpro := $entry/dbReference[@type="InterPro"]/@id
	where $entry/keyword = "Vision" and $entry/organism/name = "Human"
	  and $interpro/pub_list/publication/journal = "Nature"
	  and $interpro/pub_list/publication/year = "1990" and $interpro/@id = $ref2Interpro
	return $entry/gene;
		13

Q7	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd';
	for $entry in fn:collection('UNIPROT')/uniprot/entry,
	    $interpro in fn:collection('INTERPRO')/interprodb/interpro,
	    $pdb in fn:collection('PDB')/PDBx:datablock
	let $ref2PDB := $entry/dbReference[@type="PDB"]/@id
	let $ref2Interpro := $entry/dbReference[@type="InterPro"]/@id
	where $entry/organism/name="Mouse" and $interpro/pub_list/publication/journal = "Nature"
	  and $interpro/@id = $ref2Interpro
	  and $pdb/PDBx:citationCategory/PDBx:citation/PDBx:country = "US"
	  and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $ref2PDB
	return $entry;
		1

Q8	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd';
	for $entry in fn:collection("UNIPROT")/uniprot/entry,
	    $interpro in fn:collection("INTERPRO")/interprodb/interpro,
	    $pdb in fn:collection("PDB")/PDBx:datablock
	let $ref2PDB := $entry/dbReference[@type="PDB"]/@id
	let $ref2Interpro := $entry/dbReference[@type="InterPro"]/@id
	where $entry/organism/name="Human" and $interpro/pub_list/publication/journal = "Nature"
	 and $interpro/pub_list/publication/year = "2000"
	 and $pdb/PDBx:citationCategory/PDBx:citation/PDBx:country = "UK"
	 and $pdb/PDBx:citationCategory/PDBx:citation/PDBx:year = "2002"
	 and $interpro/@id = $ref2Interpro and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $ref2PDB
	return $entry/gene;
		2

Q9	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd';
	for $entry in fn:collection('UNIPROT')/uniprot/entry,
	    $pdb in fn:collection('PDB')/PDBx:datablock
	let $ref2PDB := $entry/dbReference[@type="PDB"]/@id
	where $entry/keyword = '3D-structure' and $entry/organism/name = 'Human'
	  and $pdb/PDBx:citationCategory/PDBx:citation/PDBx:year = "2005"
	  and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $ref2PDB
	return $entry/sequence;
		2

Q10	for $entry in fn:collection('UNIPROT')/uniprot/entry,
	    $embl in fn:collection('EMBL')/EMBL_Services/entry
	let $ref2EMBL := $entry/dbReference[@type="EMBL"]/@id
	let $temp:=$embl/@created
	where $entry/keyword = 'ATP-binding' and $entry/organism/name = 'Human'
	  and fn:starts-with(xs:string($temp), '1996') and $embl/@accession= $ref2EMBL
	return $entry/gene;
		11

Figure 10: Query set.

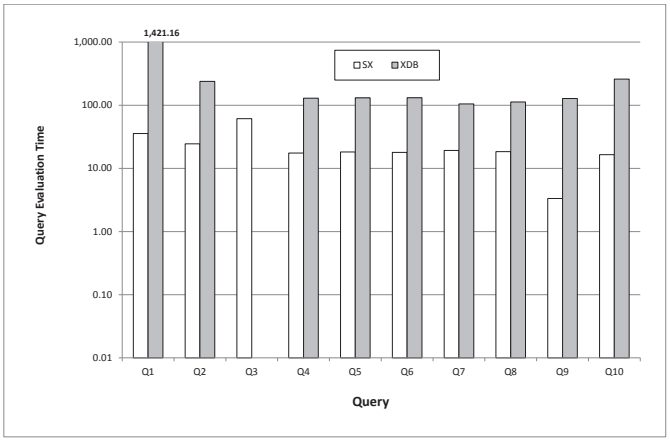


Figure 11: Query evaluation times (in sec., log-scale).

6 Experimental Results

Prototype for star query evaluation system was implemented on top of a PM-based XML database system called SUCXENT++ [2, 18] (denoted by SX) using Java JDK 1.6. The experiments were conducted on an Intel machine with Core2 Duo E6550 2.33GHz processor and 3.25GB RAM. The operating system was Windows XP Professional SP3.

We compare our approach with XDB. For SX and XDB, appropriate indexes were created. Prior to our experiments, we ensure that statistics had been collected. The bufferpool of the RDBMS was cleared before each run. The queries in SX were executed in the *reconstruct* mode [21] where not only the internal nodes are selected, but also all descendants of those nodes. Each query was executed 6 times and the results from the first run were always discarded. All rows were fetched from the answer set; however, they were not sent to output.

We would also like to observe how "far off" our approach is from one of the fastest and scalable XQuery processor MONETDB/XQuery [3], designed on top of column-oriented store [4, 20]. Hence, we used the Windows version of MONETDB/XQuery (denoted as MX) downloaded from `monetdb.cwi.nl/XQuery/Download/index.html` (Win32 builds) for our study.

6.1 Query Evaluation Times on Real Datasets

In our experiments, we used real datasets from life sciences domain as star twig queries are prevalent in this domain. Specifically, we use the XML representations of UNIPROT, PDB, INTERPRO, and EMBL downloaded from their official websites. The features of these datasets are given in Figure 3(a). We chose ten multi-source star twig queries as shown in Figures 2 and 10 that join up to four data sources, and have between three to nine expressions in the `where` clause. We transform

	14MB	140MB	1.4GB
K	5 - 500	50 – 5,000	500 – 50,000

(a) Values of K (1)

	500KB	5MB	50MB
K	10 - 75	100 - 750	1,000 – 7,500

(c) Values of K (2)

QID	Query
Z1	for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro where $entry/keyword = ' **Keyword** ' and $entry/organism/lineage/taxon = ' **Taxon** ' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id return $entry/name;
Z2	for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro where $entry/keyword = ' **Keyword** ' and $entry/organism/lineage/taxon = ' **Taxon** ' and $entry/gene/name= ' **Gene** ' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id return $entry/name;
Z3	for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro where $entry/keyword = ' **Keyword** ' and $entry/organism/lineage/taxon = ' **Taxon** ' and $entry/gene/name= ' **Gene** ' and $entry/reference/citation/authorList/person/@name = ' **Person** ' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id return $entry/name;

(b) Query Set 1

QID	Query
Y1	for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro where $interpro/@id = $entry/dbReference[@type="InterPro"]/@id and $entry/organism/lineage/taxon = 'The Taxon' and $interpro/pub_list/publication/journal = ' **Journal** ' and $interpro/pub_list/publication/year = ' **Year** ' return $entry/name ;
Y2	for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro where $interpro/@id = $entry/dbReference[@type="InterPro"]/@id and $entry/organism/lineage/taxon = 'The Taxon' and $interpro/pub_list/publication/journal = ' **Journal** ' and $interpro/pub_list/publication/year = ' **Year** ' and $interpro/taxonomy_distribution/taxon_data/@name = ' **Taxon** ' return $entry/name ;

(d) Query Set 2

Figure 12: Synthetic query sets and the *K* parameter.

these queries to our model (Section 3) if necessary. Observe that the queries are highly selective (small result size). In the next subsection, we shall investigate the effect of increasing query result size (increasing size of temporary relations) on the performance using synthetic data.

Figure 11 depicts the query evaluation times of SX and XDB. Note that we did not show any results of MX as it is vulnerable to the virtual memory fragmentation in Windows environment. Consequently, it failed to shred UNIPROT XML (1.4GB in size). Observe that SX significantly outperforms XDB for all queries. $Q3$ is not plotted for XDB due to DNF (recall that the symbol DNF means that the query evaluation did not finish in 30 mins). It is worth mentioning that unlike SX, XDB generates a single query.

6.2 Query Evaluation Times on Synthetic Datasets

We now report results related to the experiments conducted on synthetic datasets. The main objective here is to study the effects of the size of intermediate results on the query evaluation times. We compare SX, XDB, and MX. Note that due to the *GDKmallocmax* error in MX for some queries, we rewrote all queries in MX

QID	Query
W1	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd'; for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro, $pdb in fn:collection('PDB')/PDBx:datablock where $entry/keyword = 'Keyword' and $entry/organism/lineage/taxon = 'Taxon' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id =$entry/dbReference[@type="PDB"]/@id return $entry/name;
W2	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd'; for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro, $pdb in fn:collection('PDB')/PDBx:datablock where $entry/keyword = 'Keyword' and $entry/organism/lineage/taxon = 'Taxon' and $entry/gene/name= 'Gene' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $entry/dbReference[@type="PDB"]/@id return $entry/name;
W3	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd'; for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro, $pdb in fn:collection('PDB')/PDBx:datablock where $entry/keyword = 'Keyword' and $entry/organism/lineage/taxon = 'Taxon' and $entry/gene/name= 'Gene' and $entry/reference/citation/authorList/person/@name = 'Person' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $entry/dbReference[@type="PDB"]/@id return $entry/name;

(a) Query Set 3

QID	Query
V1	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd'; for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro, $pdb in fn:collection('PDB')/PDBx:datablock, $embl in fn:collection('EMBL')/EMBL_Services/entry where $entry/keyword = 'Keyword' and $entry/organism/lineage/taxon = 'Taxon' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $entry/dbReference[@type="PDB"]/@id and $embl/@accession = $entry/dbReference[@type="EMBL"]/@id return $entry/name;
V2	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd'; for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro, $pdb in fn:collection('PDB')/PDBx:datablock, $embl in fn:collection('EMBL')/EMBL_Services/entry where $entry/keyword = 'Keyword' and $entry/organism/lineage/taxon = 'Taxon' and $entry/gene/name= 'Gene' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $entry/dbReference[@type="PDB"]/@id and $embl/@accession = $entry/dbReference[@type="EMBL"]/@id return $entry/name;
V3	declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd'; for $entry in fn:collection('UNIPROT')/uniprot/entry, $interpro in fn:collection('INTERPRO')/interprodb/interpro, $pdb in fn:collection('PDB')/PDBx:datablock, $embl in fn:collection('EMBL')/EMBL_Services/entry where $entry/keyword = 'Keyword' and $entry/organism/lineage/taxon = 'Taxon' and $entry/gene/name= 'Gene' and $entry/reference/citation/authorList/person/@name = 'Person' and $interpro/pub_list/publication/journal = "The Journal" and $interpro/@id = $entry/dbReference[@type="InterPro"]/@id and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $entry/dbReference[@type="PDB"]/@id and $embl/@accession = $entry/dbReference[@type="EMBL"]/@id return $entry/name;

(b) Query Set 4

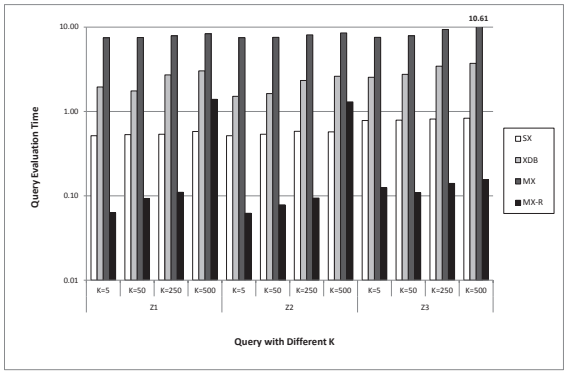
Figure 13: Synthetic query sets.

into *sequential* ones[5]. In sequential queries, non-join expressions are specified as qualifiers in path expressions of `for` clause items instead of specifying them in the `where` clause. In the sequel, we denote the MX system with the rewritten queries as MX-R.

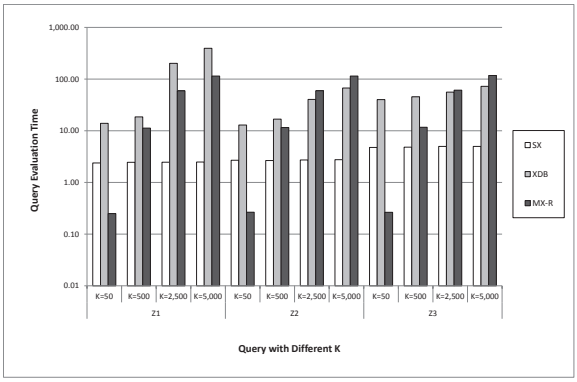
To the best of our knowledge, there does not exist any benchmark dataset designed for evaluating star twig queries. Hence, we used UNIPROT and INTERPRO datasets and modified them accordingly (discussed below) so that the size of intermediate results can be controlled. We set UNIPROT as the output data source.

Varying intermediate results of UNIPROT. We vary the size of UNIPROT documents from 14MB to 1.4GB and fix the size of INTERPRO dataset to 50MB. We control the intermediate result size by varying the number of subtrees (denoted as K) that

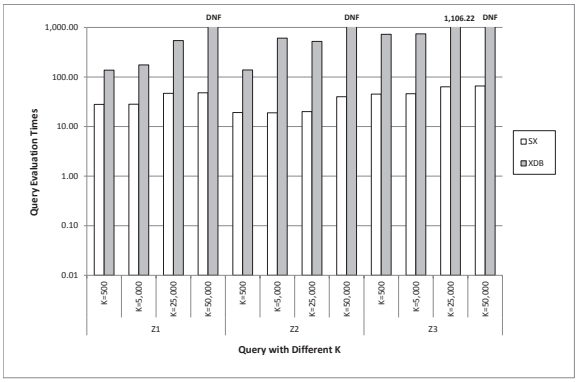
[5]This error occurred because the system cannot allocate certain amount of memory specified in the error message.



(a) 14MB

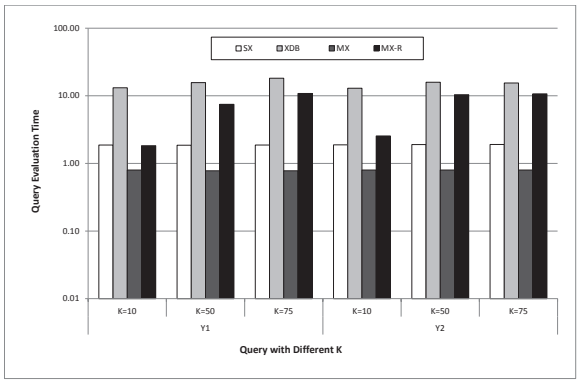


(b) 140MB

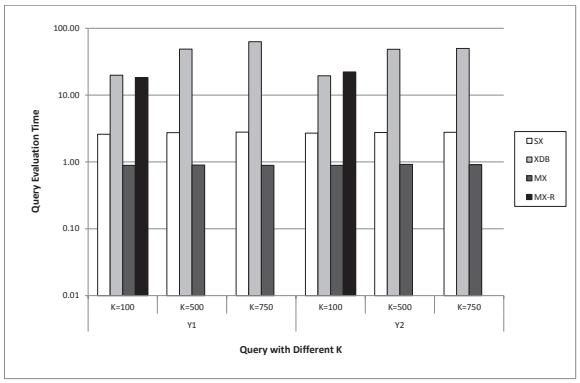


(c) 1400MB

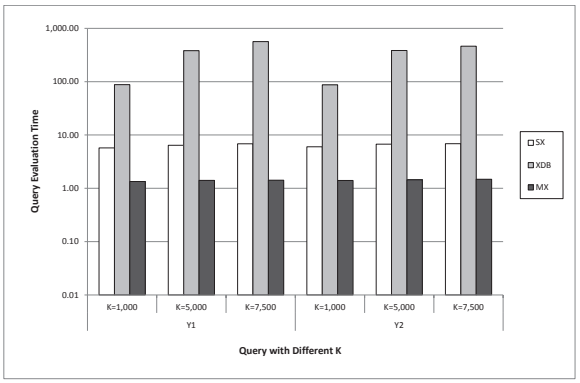
Figure 14: Query evaluation times (in sec., log-scale) on UNIPROT.



(a) 500KB



(b) 5MB



(c) 50MB

Figure 15: Query evaluation times (in sec., log-scale) on INTERPRO.

matches a non-join twig query in the XML document(s). The variation of K for different dataset sizes is depicted Figure 12(a). Figure 12(b) depicts the query set used in this set of experiments. These queries are chosen by varying the number of predicates on UNIPROT dataset from 2 to 4. We vary the result size of the highlighted predicates in the `where` clause. For instance, in Z_1 we vary the number of subtrees (K) returned by the following non-join twig condition: `$entry/keyword ='Keyword' and $entry/organism/lineage/taxon ='Taxon'`.

Figure 14 reports the query evaluation times. Note that we do not compare MX and/or MX-R in Figures 14(b) and 14(c) as it is vulnerable to the virtual memory fragmentation. We can make the following observations. Firstly, the cost of query evaluation increases with the size of intermediate results for all approaches. Secondly, SX performs better than XDB for all queries. For instance, SX is 158 times faster than XDB for Z_1 when $K = 5,000$ (Figure 14(b)). Thirdly, for certain queries SX is faster than MX. It is faster than MX for all queries for 14MB dataset (highest observed factor being 14.8 times). On the other hand, MX-R is faster than SX for 13 out of 24 queries (highest observed factor being 17.9 times). Interestingly, SX outperforms MX-R for remaining queries (up to 46 times faster). We also observe that rewriting the queries to sequential ones in MX-R performs better than MX and it can evaluate queries that previously cannot be evaluated by MX.

Varying intermediate results of INTERPRO. We now fix the UNIPROT dataset size to 140MB and vary the INTERPRO document sizes from 500KB to 50MB. The values of K for this set of experiments are depicted Figure 12(c). Figure 12(d) presents the query set. The numbers of predicates on INTERPRO dataset are set to 2 and 3 for Y_1 and Y_2, respectively. Figure 15 reports the query evaluation times. Similar to above results, SX is faster than XDB for all queries (highest observed factor being 82.7 times). However, MX performs better than SX for all queries (up to 4.8 times faster). Interestingly, we observe that MX-R cannot evaluate 10 out of 18 queries because of *GDKmallocmax* error. For the remaining queries, SX outperforms MX-R for 7 out of 8 queries (highest observed factor being 8.2 times). Hence, it is evident that rewriting XQueries to sequential ones in MX-R may not always be a beneficial strategy.

Varying number of data sources. Next, we vary the number of data sources involved in joins. Note that this also varies the number of sub-queries generated during the evaluation (Theorem 1). In addition, we also vary the intermediate result size of nodes (subtrees) of UNIPROT satisfying output expressions as depicted in Figure 12(a). We used query sets shown in Figures 13(a) and (b) joining three and four data sources, respectively. Figures 16 and 17 show the evaluation times of these queries. Note that we do not compare MX and MX-R in Figure 17 due to virtual memory fragmentation problem. Also, the results of MX and MX-R in Figures 16(b) and 17(a)–(b) are not shown because of *GDKmallocmax* error. Notice that SX is faster than XDB for all queries. Furthermore, the number of data sources involved in the join influences the query evaluation time in all approaches. On average, the performance of SX becomes 2.16 and 3.76 times slower when the numbers of data

sources are increased to three and four data sources, respectively. In the similar case, XDB is, on average, 3.19 and 14.51 times slower (without considering DNF queries).

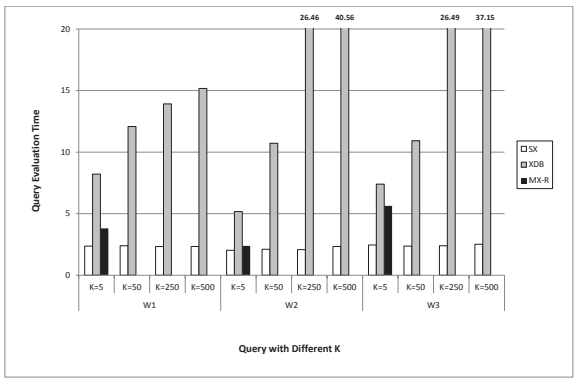
Varying number of output expressions. In this set of experiments, we shall present the evaluation times of queries with different numbers of output expressions. We vary the values of p and q. Recall that p and q are the numbers of output expressions in $\mathcal{R}$ that do not contain attribute nodes and that contain attribute nodes, respectively. We use the query templates QT_2 and QT_3 as depicted in Figures 18(a) and (b), respectively. Note that QT_2 and QT_3 join three and four data sources, respectively. The "*[ReturnClause]*" in QT_2 and QT_3 will be replaced by the return clauses shown in Figure 18(c).

Figure 19 depicts the evaluation times of the queries using query templates QT_2 and QT_3. Note that we do not show the results of MX and MX-R due to *GDKmallocmax* error. In Figure 19(a), we notice that SX outperforms XDB for 5 out of 9 queries (highest observed factor being 1.83 times, without considering DNF). For the rest of the queries, XDB is only up to 1.13 times faster than SX. We also observe that the query evaluation times of XDB show anti-monotonic behavior. In Figure 19(b), SX is faster than XDB for all queries (highest observed factor being 7.5 times).

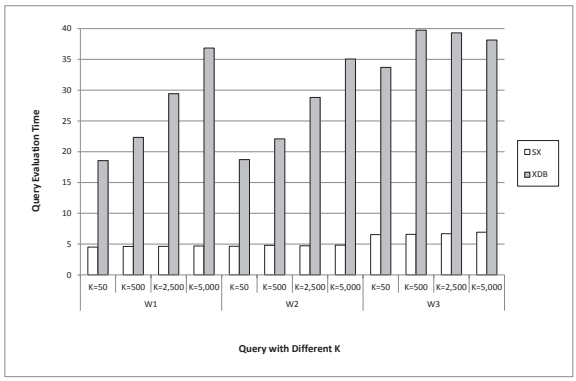
Evaluation times of sub-queries. The above results confirm the strengths of our approach. We now explore further the reasons behind such superior performance by investigating the contributions made by individual sub-queries to the execution costs of the translated SQL queries. We chose Z_2 and Y_2 as our test queries. The translated SQL query of Z_2 and Y_2 each consists of five sub-queries (denoted as SQ_1 to SQ_5). SQ_1 is used to fetch the identifiers of the output nodes (Phase 1). SQ_2 and SQ_3 materialize the results for non-join and join expressions (Phase 3). The PathUFinal relation is generated by SQ_4. SQ_5 retrieves the complete subtrees including the necessary attributes for reconstruction and all the descendant node if the output node is an internal node. We evaluate the evaluation time of each subquery using SX as shown in Figure 20. Observe that relatively the most expensive query is SQ_3 for both cases. However, the evaluation time is still below 15s (significantly lower than the evaluation times of XDB). On the other hand, SQ_1, SQ_2, SQ_4, and SQ_5 are highly efficient for almost all cases. This is primarily due to (a) efficient support of twig pattern evaluation in a PM-based XML storage approach, (b) space-efficient storage of intermediate results of the queries, and (c) small queries are less likely to stress the query optimizer.

7 Conclusions and Future Work

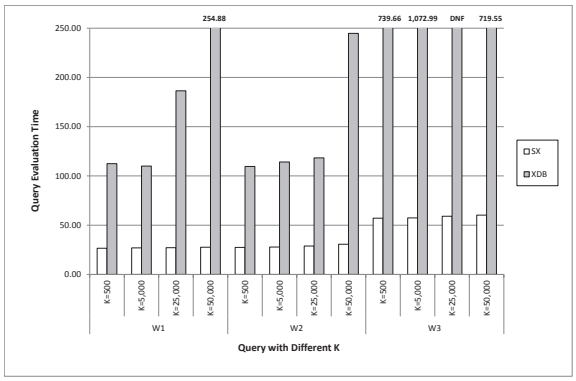
In this paper, we take a non-traditional approach in evaluating multi-source star twig queries on top of a path-based tree-unaware XML database. Our scheme is built on top of SUCXENT++ [18]. We take a non-traditional approach in evaluating such queries. Rather than generating one huge complex SQL query, we translate a



(a) 14MB

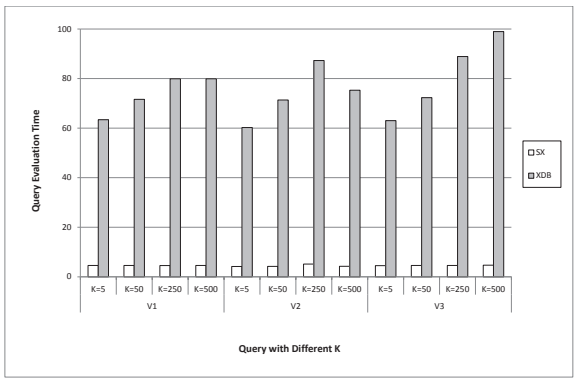


(b) 140MB

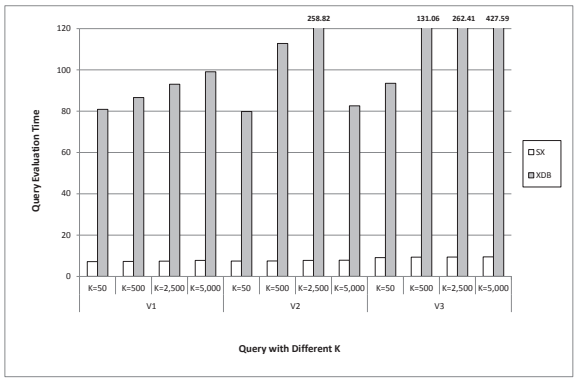


(c) 1400MB

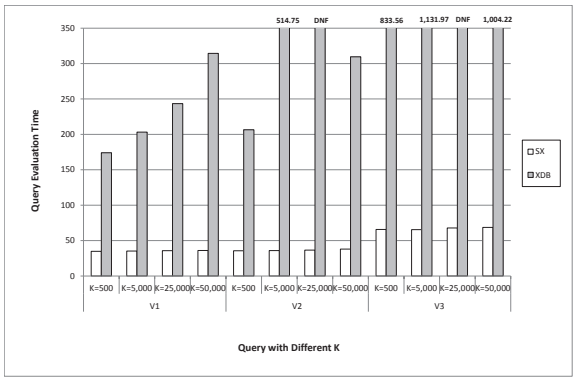
Figure 16: Query evaluation times (3 data sources, in sec.).



(a) 14MB



(b) 140MB



(c) 1400MB

Figure 17: Query evaluation times (4 data sources, in sec.).

```
xquery declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd';
for $entry in db2-fn:xmlcolumn('U1400.CONTENT')/uniprot/entry,
    $interpro in db2-fn:xmlcolumn('INTERPRO.CONTENT')/interprodb/interpro,
    $pdb in db2-fn:xmlcolumn('PDB.CONTENT')/PDBx:datablock
let $ref2PDB := $entry/dbReference[@type="PDB"]/@id
let $ref2Interpro := $entry/dbReference[@type="InterPro"]/@id
where $interpro/@id = $ref2Interpro and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $ref2PDB
  and $entry/keyword = "Phosphoprotein" and $entry/organism/name = "Human"
  and $interpro/pub_list/publication/journal = "J. Biol. Chem."
  and $pdb/PDBx:citationCategory/PDBx:citation/PDBx:country  = "US"
[ReturnClause]
```

(a) Query Template QT_2

```
declare namespace PDBx = 'http://deposit.pdb.org/pdbML/pdbx.xsd';
for $entry in fn:collection('UNIPROT')/uniprot/entry,
    $interpro in fn:collection('INTERPRO')/interprodb/interpro,
    $embl in fn:collection('EMBL')/EMBL_Services/entry,
    $pdb in fn:collection("PDB")/PDBx:datablock
let $ref2PDB := $entry/dbReference[@type="PDB"]/@id
let $ref2EMBL := $entry/dbReference[@type="EMBL"]/@id
let $ref2InterPro := $entry/dbReference[@type="InterPro"]/@id
let $temp:=$embl/@created
where $entry/keyword = 'ATP-binding' and $entry/organism/name = 'Human'
  and $interpro/pub_list/publication/journal = 'Science' and fn:starts-with(xs:string($temp), '1996')
  and $pdb/PDBx:citationCategory/PDBx:citation/PDBx:country="US"
  and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $ref2PDB
  and $interpro/@id = $ref2InterPro and $embl/@accession= $ref2EMBL
[ReturnClause]
```

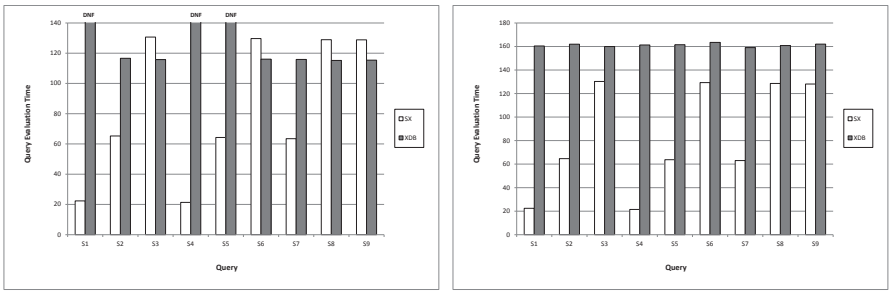
(b) Query Template QT_3

QID	Return Clause	p	q
S1	return <result><uniprot>{$entry/gene/name}</uniprot></result>;	0	1
S2	return <result><uniprot>{$entry/organism/name}</uniprot> <interpro>{$interpro/pub_list/publication}</interpro></result>;	0	2
S3	<result><uniprot>{$entry/organism/name}</uniprot> <interpro>{$interpro/pub_list/publication}</interpro> <pdb>{$pdb/PDBx:pdbx_database_statusCategory}</pdb></result>;	0	3
S4	return <result><uniprot>{$entry/name}</uniprot></result>;	1	0
S5	return <result><uniprot>{$entry/name}</uniprot> <interpro>{$interpro/pub_list/publication}</interpro></result>;	1	1
S6	return <result><uniprot>{$entry/name}</uniprot> <interpro>{$interpro/pub_list/publication}</interpro> <pdb>{$pdb/PDBx:pdbx_database_statusCategory}</pdb></result>;	1	2
S7	return <result><uniprot>{$entry/name}</uniprot><interpro>{$interpro/name}</interpro></result>;	2	0
S8	return <result><uniprot>{$entry/name}</uniprot><interpro>{$interpro/name}</interpro> <pdb>{$pdb/PDBx:pdbx_database_statusCategory}</pdb></result>;	2	1
S9	return <result><uniprot>{$entry/name}</uniprot><interpro>{$interpro/name}</interpro> <pdb>{$pdb/PDBx:citationCategory/PDBx:citation/PDBx:title}</pdb></result>;	3	0

(c) Return Clause

Figure 18: Query Templates QT_2 and QT_3, and Various Return Clause.

star XQuery into a list of SQL queries. This is surprising, because when only one SQL query is generated, it has the greatest potential for optimization by the RDBMS. We showed that by exploiting the encoding scheme of SUCXENT++ as well as materializing only minimal information of underlying XML subtrees as intermediate results we can "turbo-charge" star query processing. Though not elaborated in this paper, it is easy to see that our approach is also applicable to a host of XML databases using relational backend as well as wide varieties of complex XML queries. Our results showed that our proposed technique has excellent real-world performance, outperforming XML join support of DB2 for many queries. Although MONETDB/XQuery is often the best in terms of query performance [3], surprisingly, our results show that our scheme outperforms it for several queries. As part of future work, we would like to extend our approach to larger subset of XML queries.



(a) 3 data sources. (b) 4 data sources.

Figure 19: Query evaluation times (in sec.).

References

[1] W3C XQuery 1.0 Grammar Test Page. *http://www.w3.org/2005/qt-applets/xqueryApplet.html*, 2005.

[2] S. S. Bhowmick, E. Leonardi, H. Sun. Efficient Evaluation of High-Selective XML Twig Patterns with Parent Child Edges in Tree-Unaware RDBMS. *In ACM CIKM*, 2007.

[3] P. Boncz, T. Grust, et al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. *In SIGMOD* , 2006.

[4] P. Boncz, M. L. Kersten. MIL Primitives for Querying A Fragmented World. *In VLDB Journal* , 8(2), 1999.

[5] M. Brantner, C-C. Kanne, G. Moerkotte. Let a Single FLWOR Bloom (to improve XQuery plan generation). *In XSym Workshop*, 2007.

[6] A. Deutsch, V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. *In VLDB*, 2003.

[7] M. Fernandez, A. Morishima, D. Suciu. Efficient Evaluation of XML Middle-ware Queries. *In SIGMOD*, 2001.

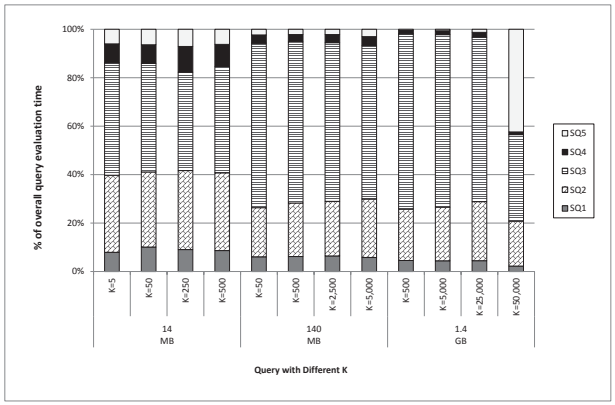
[8] G. Gou, R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *In IEEE TKDE*, 19(10), 2007.

[9] T. Grust, J. Rittinger, J. Teubner. Why Off-the-Shelf RDBMSs are Better at XPath Than You Might Expect. *In SIGMOD* , 2007.

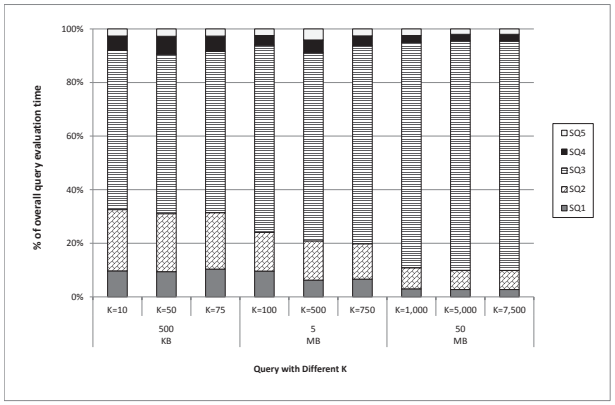
[10] T. Grust, S. Sakr, J. Teubner. XQuery on SQL Hosts. *In VLDB*, 2004.

[11] R. Krishnamurthy, R. Kaushik, et al. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? *In VLDB*, 2004.

[12] R. Krishnamurthy, R. Kaushik, J. F. Naughton. Efficient XML-to-SQL Query Translation Literature: State of the Art and Open Problems. *In XSym*, 2003.



(a) Query Z_2.



(b) Query Y_2.

Figure 20: Subqueries evaluation times of Z_2 and Y_2 (in msec).

[13] I. Manolescu, D. Florescu, D. Kossmann. Answering XML Queries over Heterogeneous Data Sources. *In VLDB*, 2001.

[14] A. Marian, J. Simon. Projecting XML Documents. *In VLDB*, 2003.

[15] P. Michiels, G. A. Mihaila, J. Simeon. Put a Tree pattern in Your Algebra. *In ICDE*, 2007.

[16] P. O'Neal, E. O'Neal, S. Pal et al. ORDPATHs: Insert-Friendly XML Node Labels. *In SIGMOD*, 2004.

[17] S. Pal, I. Cseri, O. Seeliger, et al. XQuery Implementation in a Relational Database System. *In VLDB*, 2005.

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

[19] J. Shanmugasundaram, J. Kiernan, et al. Querying XML Views of Relational Data. *In VLDB*, 2001.

[20] M. Stonebraker, D. Abadi, et al. C-store: A Column-Oriented DBMS. *In VLDB*, 2005.

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

[22] M. Yoshikawa, et al. XRel: A Path-based Approach to Storage and Retrieval of XML documents Using Relational Databases. *ACM TOIT* 1(1), 2001.