Stars on Steroids: Fast Evaluation of Multi-source Star Twig Queries in RDBMS

Erwin Leonardi[1,2], Sourav S. Bhowmick[1,2], and Fengrong Li[3]

[1] Singapore-MIT Alliance, Nanyang Technological University, Singapore
[2] School of Computer Engineering, Nanyang Technological University, Singapore
[3] 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. Experiments carried out confirm that our proposed approach build on top of an off-the-shelf commercial RDBMS has excellent real-world performance.

1 Introduction

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 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.*, [8, 12] 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.*, [8]. Optimization of an XQuery query is achieved in two

QID	Query	# of Results
Q1	`declare namespace PDBx = 'http://deposit.pdb.org/pdbML/pdbx.xsd';`	64
	`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:citationCategory/PDBx:citation/PDBx:year = "1997"`	
	`    and $pdb/PDBx:cellCategory/PDBx:cell/@entry_id = $ref2PDB`	
	`    and $interpro/@id = $ref2InterPro`	
	`   and $embl/@accession= $ref2EMBL`	
	`return $entry/reference/citation/title;`	
Q2	`for $entry in fn:collection('UNIPROT')/uniprot/entry,`	4
	`    $interpro in fn:collection('INTERPRO')/interprodb/interpro,`	
	`    $embl in fn:collection('EMBL')/EMBL_Services/entry`	
	`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 $interpro/@id = $ref2InterPro and $embl/@accession= $ref2EMBL`	
	`return $entry/name;`	
Q3	`for $entry in fn:collection('UNIPROT')/uniprot/entry,`	11
	`   $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;`	

Fig. 1. Examples of star twig queries

stages. Logical query optimization (sometimes also called query rewrite) [8,12,13] 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 XML support provided by DB2.

Unfortunately, query performance still remains a bottleneck. To get a better understanding of this problem, we experimented with the datasets in Figure 2(a) and queries $Q_1 - Q_3$. Figure 2(b) shows the query evaluation times in DB2. Observe that it can take from 4 minutes to more than 20 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 build 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 above queries can be evaluated in *less than a minute*.

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 [6]. First, SQL queries for materializing the *identifiers* of

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	259.83

(b) Query Evaluation Time (in sec.)

Fig. 2. Dataset and query evaluation times in DB2

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) 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.

Our proposed approach has excellent performance. It is significantly faster than XML support of DB2 $v9.5$ (highest observed factor being 158 times), which relies on conventional XQuery optimization techniques. Somewhat unexpectedly, we shall also show that the proposed technique outperforms one of the fastest XQuery processor (MONETDB/XQuery [2]) 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. Section 4 presents 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 5. Section 6 concludes the paper and suggests future work.

2 Related Work

There is a wealth of work on evaluating XPath expressions in a tree-unaware RDBMS [6, 7, 14] and tree-aware environment [2, 6]. However, these efforts mainly focus on various XPath axes and not on XML join operation. In all these efforts, the SQL translation algorithms 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 queries.

There has been efforts related to translating XML queries to SQL in XML publishing environment [9]. In XPeranto [16], an XQuery query is transformed into an XML Query Graph Model (XQGM) and composed with the view definition. Then it is translated to a single "outer union" SQL query to be evaluated inside the relational engine. The Agora [11] project uses *local-as-view* (LAV) approach to translate the XML query into a SQL query over virtual relational schema and then rewriting this SQL query into a query over the real relational schema. MARS [4] uses both local-as-view and *global-as-view* (GAV) approaches. It first compiles the queries, views and constraints from XML into the relational framework and then determines all minimal reformulations of the relational queries under the relational integrity constraints using a cost-based approach. In contrast, our approach is build on top of XML storage framework and translates a specific type of XML 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 [5]. In [5], mapping from the relational schema to the XML view 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. 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* (representation that makes it clear how to generate queries). 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.

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 [6]. 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 three types of nodes: location step query node (QNode), logical-AND node (ANode), and return node (RNode). Each Q has a single node of type RNode which represents the output node. While the label of ANode is always "AND", QNodes' and RNodes' 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. Specifically, a join edge (q_1, q_2) asserts that q_1 and q_2 have equal value[2]. We distinguish the RNode by underlined tag; and axes and join edges as direct and dashed edges, respectively.

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

[2] We currently support equality join condition but inequality join condition can be supported easily.

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. 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   $b_1 \wedge b_2 \wedge \ldots \wedge b_k \wedge c_1 \wedge c_2 \wedge \ldots \wedge c_m$
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* involves predicates that express join conditions over two different document sources. On the other hand, a *non-join expression* expresses a filtering condition on a single source. 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.*

- *$\mathcal{F}$ 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 the* `return` *clause containing output expression r, which is a 2-tuple (var, exp) where $var = f.var$ or $var = l.var$, $f \in \mathcal{F}$, and $l \in \mathcal{L}$.* □

3.2 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

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 $(\mathcal{J}, \mathcal{C}) \leftarrow$ distinguishExp($W$) /* Phase 3 */;
5 $SQLList$.add(whereExp2SQL($\mathcal{F}, \mathcal{J}, \mathcal{C}, \mathcal{R}$));
6 $SQLList$.add(finalResultQueryGen($\mathcal{R}$)) /* Phase 4 */ ;
7 return $SQLList$
```

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 1 are examples of star twig queries. Formally, it is defined as follows.

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 [6]. 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 [6], 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. The reader may refer to [10] for 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. The algorithm consists of four phases as discussed below.

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 Q with the expressions defined in the `let` clause (if any). The output

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

```
Input: An output expression $r \in \mathcal{R}$
Output: An SQL query $\_SQL$
1  Initialize $\_SQL = \infty$;
2  if (r is an attribute node) then
3      $PathExp \leftarrow$ pathExpOfParentNode($r$);
4  else
5      $PathExp \leftarrow r.absExp$;
6  $PathIDs \leftarrow$ getAllPathID($PathExp$);
7  $Level \leftarrow$ getNodeLevel($PathExp$);
8  $Source = r.dsName$;
9  $\_SQL$.genSQL($PathIDs$, $Level$, $Source$);
10 return $\_SQL$
```

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

```
Input: $\mathcal{F}, \mathcal{J}, \mathcal{C}, r \in \mathcal{R}$
Output: A list of SQL queries $SQLList$
1  Initialize $SQLList = \emptyset$;
2  for (each $f \in \mathcal{F}$) do
3      $\mathcal{C}_f \leftarrow$ getNonJoinExp($f.var$, $\mathcal{C}$);
4      if ($f.var = r.var$) then
5          $SQL \leftarrow$ translateWhereNonJoin($r$, $f$, $\mathcal{C}_f$);
6      else
7          $SQL \leftarrow$ translateWhereJoin($r$, $f$, $\mathcal{C}_f$, $\mathcal{J}$);
8      $SQL \leftarrow$ INSERT INTO $T\_$" $+\mathcal{R}+$ "_"$+\mathcal{F}$.indexOf($f$)+ " " $+SQL$;
9      $SQLList$.add($SQL$);
10 return $SQLList$
```

of this phase are a set of `for` clause items $\mathcal{F}$, a set of *where-expressions* $\mathcal{W}$, and the output expression $r \in \mathcal{R}$. 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 $r =$ (*$entry*, "`/name`") in Q_2 (Figure 1). Then $r.absExp$ is "`/uniprot/entry/name`" as *$entry* is bound to the expression "`/uniprot/entry`".

Phase 2: OutputExp2SQL Translation. In this phase, the algorithm analyzes the 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

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

```
Input: An output expression $r$, a for clause item $f$, $\mathcal{C}_f$
Output: An SQL query $SQL$
1  Initialize $selectClause$, $fromClause$, $whereClause$, $optionClause$;
2  $dataS \leftarrow$ source($r.var$);
3  for ($i = 1$ to $|\mathcal{C}_f|$) do
4      $c = \mathcal{C}_f[i]$;
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($c.absExp$, $\mathcal{C}_f[i-1].absExp$));
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 $SQL$
```

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 [6].

Given an output expression $r \in \mathcal{R}$, the *OutputExp2SQL* algorithm depicted in Algorithm 2 works as follows. First, the algorithm determines whether r involves an attribute node (Line 02). If it does, then the algorithm retrieves the absolute path expression of its parent node (Line 03). Otherwise, the absolute path expression of r is used (Line 05). This expression is stored in the variable *PathExp*. Based on *PathExp*, a set of path ids is retrieved from the Paths table (Line 06). 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.

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. For example, there are three pairs of

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

```
Input: An output node $r$, a for clause item $f$, $\mathcal{C}_f$, $\mathcal{J}$
Output: An SQL query $SQL$
1  Initialize $selectClause$, $fromClause$, $whereClause$, $optionClause$;
2  processExpressions($\mathcal{C}_f$);
3  $i = |\mathcal{C}_f| + 1$;
4  $\mathcal{J}_f \leftarrow \mathcal{J}$.getJoinExp($f$);
5  $\mathcal{J}_r \leftarrow \mathcal{J}$.getJoinExp($r$);
6  if ($\mathcal{J}_f \cap \mathcal{J}_r = \emptyset$) then
7      processJoinExp($\mathcal{J}_f$.getS(), $\mathcal{J}_f$.getT(), $i$, $\mathcal{C}_f[|\mathcal{C}_f|].absExp$);
8      processJoinExp($\mathcal{J}_r$.getS(), $\mathcal{J}_r$.getT(), $i$, $\mathcal{C}_f[|\mathcal{C}_f|].absExp$);
9  else
10     $\mathcal{J}_x = \mathcal{J}_f \cap \mathcal{J}_r$;
11     processJoinExp($\mathcal{J}_x$.getS(), $\mathcal{J}_x$.getT(), $i$, $\mathcal{C}_f[|\mathcal{C}_f|].absExp$);
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 $SQL$
```

(r, f) in Q_2, namely (*$entry*, *$entry*), (*$entry*, *$interpro*), and (*$entry*, *$embl*). Since (*$entry*, *$entry*) refers to the same data source (UNIPROT), the algorithm generates an SQL query that retrieves those nodes in the PathU table (generated by Phase 2) that satisfy the non-join predicates on UNIPROT. The results of this query is stored in an instance of TempTable (denoted by T_1). On the other hand, data sources of (*$entry*, *$interpro*) are not identical and hence a join query is generated that selects nodes from PathU that satisfy the join predicate (Line 11 in Q_2) and the conditions on INTERPRO (Line 9). The results of this query is stored in the temporary table T_2.

The *WhereExp2SQL* algorithm is depicted in Algorithm 3. For each $f \in \mathcal{F}$, first, it retrieves $\mathcal{C}_f \subseteq \mathcal{C}$, where $\forall c \in \mathcal{C}_f$ $c.var = f.var$ (Line 03). Then, it determines whether r and f are bound to the same data source. If $r.var = f.var$, then join across data sources is not necessary. In this case, the algorithm will invoke the *translateWhereNonJoin* algorithm (Line 05). Otherwise, it invokes the *translateWhereJoin* algorithm (Line 07). The generated SQL query is stored in a variable called *SQLList*. Lastly, 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. We now elaborate on the *translateWhereNonJoin* and *translateWhereJoin* procedures.

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* $c \in \mathcal{C}_f$, 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 $\mathcal{C}_f$, 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 SQL query generated by the *translateWhereExp* algorithm returns the identifiers of nodes satisfying r that satisfy expressions in $\mathcal{C}_f$.

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 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 $\mathcal{J}_f = \{$"$S_1 = T$"$\}$ and $\mathcal{J}_r = \{$"$S_2 = T$"$\}$. 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 $\mathcal{J}_f = \mathcal{J}_r = \{$"$S_1 = T$"$\}$. 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.

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*

QID	Query	# of Results
Q4	`for $entry in fn:collection('UNIPROT')/uniprot/entry,`	31
	`    $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;`	
Q5	`for $entry in fn:collection('UNIPROT')/uniprot/entry,`	13
	`    $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;`	
Q6	`declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd';`	2
	`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;`	
Q7	`declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd';`	2
	`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;`	
Q8	`declare namespace PDBx='http://deposit.pdb.org/pdbML/pdbx.xsd';`	1
	`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;`	

Fig. 3. Query set

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
SX	35.62	24.47	16.48	18.23	17.94	3.34	18.46	19.26
XDB2	1,421.16	238.73	259.83	130.80	131.35	128.08	112.77	104.73

Fig. 4. Query evaluation times (in sec.)

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. Then, the total number of* SQL *queries generated from Q is $(n + k)$ where (a) if the output expression $r \in \mathcal{R}$ does not contain attribute node then $k = 3$; (b) Otherwise, $k = 4$.* $\square$

Due to space constraints, the proof is given in [10].

5 Experimental Results

Prototype for star query evaluation system was implemented on top of a PM-based XML database system called SUCXENT++ [14] (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. The RDBMS used was MS SQL Server 2005 Developer Edition.

We compare our approach to the native XML support of IBM DB2 $v9.5$ (denoted by XDB2). XML support of IBM DB2 is also used as performance benchmark in [7]. For SX and XDB2, appropriate indexes were created [10]. 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 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. For XDB2, we use the *db2batch* Benchmark Tool provided by the system. All rows were fetched from the answer set; however, they were not sent to output.

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

	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
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

Fig. 5. Synthetic query sets and the K parameter

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

Fig. 6. Synthetic query sets

We would also like to observe how “far off” our approach is from one of the fastest XQuery processor (MONETDB/XQuery [2]). Hence, we used the Windows version of MONETDB/XQuery 0.24.0 (denoted as MX) downloaded from `monetdb.cwi.nl/XQuery/Download/index.html` (Win32 builds) for our study.

5.1 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 2(a). We chose eight multi-source star twig queries as shown in Figures 1 and 3 that join up to four data sources, and have between three to nine expressions in the `where` clause. We transform these queries to our model (Section 3) if necessary. Observe that the queries are highly selective (small result size).

	Z1				Z2				Z3			
	K=5	K=50	K=250	K=500	K=5	K=50	K=250	K=500	K=5	K=50	K=250	K=500
SX	0.52	0.53	0.54	0.58	0.52	0.54	0.58	0.57	0.78	0.79	0.81	0.83
XDB2	1.95	1.75	2.71	3.03	1.51	1.63	2.33	2.61	2.54	2.75	3.44	3.72
MX	7.47	7.52	7.89	8.33	7.48	7.56	8.08	8.53	7.56	7.89	9.38	10.61
MX-R	0.06	0.09	0.11	1.39	0.06	0.08	0.09	1.30	0.13	0.11	0.14	0.16

(a) UNIPROT (14 MB)

	Z1				Z2				Z3			
	K=50	K=500	K=2,500	K=5,000	K=50	K=500	K=2,500	K=5,000	K=50	K=500	K=2,500	K=5,000
SX	2.38	2.45	2.46	2.49	2.69	2.66	2.72	2.76	4.75	4.82	4.99	4.99
XDB2	13.92	18.59	201.98	393.95	12.92	16.79	40.38	67.18	40.10	45.51	56.00	72.37
MX	GDKmallocmax Error											
MX-R	0.25	11.25	59.39	114.50	0.27	11.53	59.45	114.66	0.27	11.69	61.00	117.17

(b) UNIPROT (140 MB)

	Z1				Z2				Z3			
	K=500	K=5,000	K=25,000	K=50,000	K=500	K=5,000	K=25,000	K=50,000	K=500	K=5,000	K=25,000	K=50,000
SX	27.96	28.20	46.76	47.90	19.17	18.97	19.99	40.04	45.06	45.99	63.45	65.94
XDB2	137.31	175.24	542.73	DNF	138.27	608.10	521.33	DNF	724.47	740.36	1,106.22	DNF

(c) UNIPROT (1400 MB)

	Y1			Y2		
	K=10	K=50	K=75	K=10	K=50	K=75
SX	1.87	1.86	1.87	1.88	1.90	1.91
XDB2	13.15	15.68	18.22	12.94	15.87	15.46
MX	0.80	0.78	0.78	0.80	0.80	0.80
MX-R	1.83	7.48	10.81	2.55	10.39	10.69

(d) INTERPRO (500KB)

	Y1			Y2		
	K=100	K=500	K=750	K=100	K=500	K=750
SX	2.60	2.74	2.80	2.70	2.76	2.78
XDB2	19.82	48.77	62.89	19.47	48.50	49.84
MX	0.89	0.90	0.89	0.89	0.92	0.91
MX-R	18.25	GDKMallocmax Error		22.20	GDKMallocmax Error	

(e) INTERPRO (5MB)

	Y1			Y2		
	K=1,000	K=5,000	K=7,500	K=1,000	K=5,000	K=7,500
SX	5.72	6.40	6.83	6.01	6.74	6.85
XDB2	87.77	381.34	564.97	87.20	383.47	462.89
MX	1.34	1.41	1.42	1.40	1.44	1.48
MX-R	GDKmallocmax Error					

(f) INTERPRO (50MB)

Fig. 7. Query evaluation times (in sec.)

Figure 4 depicts the query evaluation times of SX and XDB2. 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 XDB2 for all queries.

5.2 Query Evaluation Times on Synthetic Datasets

The main objective here is to study the effects of the size of intermediate results on the query evaluation times. In the sequel, the symbol DNF means that the query evaluation did not finish in 30 mins. We compare SX, XDB2, and MX. Note that due to the *GDKmallocmax* error in MX for some queries, we rewrote all queries in MX into *sequential* ones[3]. 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 MONETDB system with the rewritten queries as MX-R. We use UNIPROT and INTERPRO datasets and modified them (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 matches a non-join twig query in the XML document(s). The variation of K for different dataset sizes is depicted Figure 5(a). Figure 5(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’`.

Figures 7(a)–(c) show the query evaluation times. Note that we do not compare MX and MX-R in Figure 7(c) as it is vulnerable to the virtual memory fragmentation. We can make the following observations. Firstly, the cost of query evaluation increases

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

	W1				W2				W3			
	K=500	K=5,000	K=25,000	K=50,000	K=500	K=5,000	K=25,000	K=50,000	K=500	K=5,000	K=25,000	K=50,000
SX	26.62	26.92	27.19	27.67	27.42	27.86	28.83	30.67	57.10	57.41	59.05	60.20
XDB2	112.48	109.98	186.45	258.44	109.63	114.19	118.30	244.75	739.66	1,072.99	DNF	719.55

(a) 3 Data Sources (UNIPROT, INTERPRO, and PDB)

	V1				V2				V3			
	K=500	K=5,000	K=25,000	K=50,000	K=500	K=5,000	K=25,000	K=50,000	K=500	K=5,000	K=25,000	K=50,000
SX	34.85	35.23	35.73	35.99	35.63	35.84	36.57	37.94	65.70	65.41	67.85	68.65
XDB2	174.02	203.20	243.35	314.38	206.53	514.75	DNF	309.52	833.56	1,131.97	DNF	1,004.22

(b) 4 Data Sources (UNIPROT, INTERPRO, PDB, and EMBL)

Fig. 8. Query evaluation times (1.4GB UNIPROT (in sec.))

with the size of intermediate results for all approaches. Secondly, SX performs better than XDB2 for all queries. For instance, SX is 158 times faster than XDB2 for Z_1 when $K = 5,000$ (Figure 7(b)). Thirdly, for certain queries SX is faster than MONETDB! 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 MONETDB 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 5(c). Figure 5(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. Figures 7(d)–(f) depict the query evaluation times. Similar to above results, SX is faster than XDB2 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 MONETDB 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 5(a). We used query sets shown in Figures 6(a) and (b) joining three and four data sources, respectively. Figure 8 shows the evaluation times of queries in Figures 6(a) and (b). Note that we do not compare MX and MX-R in Figure 8 due to virtual memory fragmentation problem. Notice that SX is faster than XDB2 for all queries. Furthermore, the number of data sources involved in the join influences the query evaluation time in all approaches.

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

	K	SQ1	SQ2	SQ3	SQ4	SQ5
14 MB	5	56.52	226.58	330.18	55.60	42.80
	50	72.52	222.68	321.56	55.42	45.84
	250	71.34	256.18	319.34	83.48	55.64
	500	64.12	237.80	325.14	67.68	46.14
140 MB	50	156.28	520.82	1,742.60	88.56	60.30
	500	164.00	584.10	1,757.82	80.56	57.34
	2,500	173.24	605.90	1,758.32	89.94	58.66
	5,000	165.24	689.78	1,808.28	105.12	87.34
1.4 GB	500	875.16	4,056.80	13,853.94	268.02	95.22
	5,000	869.68	4,315.80	13,910.64	312.50	123.16
	25,000	919.72	5,025.02	13,967.08	416.80	261.66
	50,000	881.08	7,167.38	13,836.18	434.72	16,405.90

(a) Query Z_2

	K	SQ1	SQ2	SQ3	SQ4	SQ5
500 KB	10	183.26	433.04	1,123.62	102.64	49.94
	50	180.40	415.78	1,143.44	128.68	52.94
	75	194.64	395.96	1,131.92	107.14	50.14
5 MB	100	282.58	423.86	2,055.20	107.60	71.80
	500	213.32	496.96	2,388.94	162.18	140.48
	750	216.88	431.32	2,417.62	119.38	84.56
50 MB	1,000	199.18	527.22	5,628.48	171.40	164.66
	5,000	198.84	524.28	6,234.62	175.98	147.58
	7,500	203.76	535.20	6,387.98	183.88	150.80

(b) Query Y_2

Fig. 9. Sub-queries evaluation times of Z_2 and Y_2 (in msec)

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 sub-query using SX as shown in Figure 9. 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 XDB2). 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.

6 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. Rather than generating one huge complex SQL query, we translate a star query 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 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 [2] is one of the fastest XQuery processor, 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.

References

1. W3C. XQuery 1.0 Grammar Test Page (2005), `http://www.w3.org/2005/qt-applets/xqueryApplet.html`
2. Boncz, P., Grust, T., et al.: MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In: SIGMOD (2006)
3. Brantner, M., Kanne, C.-C., Moerkotte, G.: Let a Single FLWOR Bloom (to improve XQuery plan generation). In: XSym Workshop (2007)
4. Deutsch, A., Tannen, V.: MARS: A System for Publishing XML from Mixed and Redundant Storage. In: VLDB (2003)
5. Fernandez, M., Morishima, A., Suciu, D.: Efficient Evaluation of XML Middle-ware Queries. In: SIGMOD (2001)
6. Gou, G., Chirkova, R.: Efficiently Querying Large XML Data Repositories: A Survey. IEEE TKDE 19(10) (2007)
7. Grust, T., Rittinger, J., Teubner, J.: Why Off-the-Shelf RDBMSs are Better at XPath Than You Might Expect. In: SIGMOD (2007)
8. Grust, T., Sakr, S., Teubner, J.: XQuery on SQL Hosts. In: VLDB (2004)
9. Krishnamurthy, R., Kaushik, R., Naughton, J.F.: Efficient XML-to-SQL Query Translation Literature: State of the Art and Open Problems. In: XSym (2003)
10. Leonardi, E., Bhowmick, S.S., Li, F.: Fast Evaluation of Multi-source Star Twig Queries in a Path Materialization-based xml Database. Technical Report (2010), `http://www.cais.ntu.edu.sg/~assourav/TechReports/StarJoin-TR.pdf`
11. Manolescu, I., Florescu, D., Kossmann, D.: Answering XML Queries over Heterogeneous Data Sources. In: VLDB (2001)
12. O'Neal, P., O'Neal, E., Pal, S., et al.: ORDPATHs: Insert-Friendly XML Node Labels. In: SIGMOD (2004)
13. Pal, S., Cseri, I., Seeliger, O., et al.: XQuery Implementation in a Relational Database System. In: VLDB (2005)
14. Seah, B.-S., Widjanarko, K.G., Bhowmick, S.S., Choi, B., Leonardi, E.: Efficient Support for Ordered XPath Processing in Tree-Unaware Commercial Relational Databases. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 793–806. Springer, Heidelberg (2007)
15. Shanmugasundaram, J., Tufte, K., et al.: Relational Databases for Querying XML Documents: Limitations and Opportunities. In: VLDB (1999)
16. Shanmugasundaram, J., Kiernan, J., et al.: Querying XML Views of Relational Data. In: VLDB (2001)