Dynamic Intervals: Handling Intermediate XML Query Results (even using SQL)

Mariano P. Consens, David DeHaan, and David Toman

Department of Computer Science
University of Waterloo, Canada
{mconsens,dedehaan,david}@uwaterloo.ca

Abstract

The W3C XQuery language recommendation, based on a hierarchical and ordered document model, supports a wide variety of constructs and use cases. Concomitantly, there is a diversity of approaches and strategies for evaluating XQuery expressions, in many cases only dealing with limited subsets of the language.

In this paper we describe an implementation approach that handles XQuery with nested FLWR expressions, element constructors and built-in functions (including structural comparisons). Our proposal maps XQuery expressions to a single equivalent SQL query over a novel *dynamic interval* encoding of a collection of XML documents as relations, augmented with information tied to the query evaluation environment.

While commercial SQL engines can then be used to evaluate the resulting queries, there are areas where performance can be significantly improved. We introduce additional relational query plan operators that fully exploit properties of ordered relational data representing generic XML documents. The techniques presented in this paper enable query execution plans based on sorting and linear merge-join operations even in the presence of nested FLWR constructs.

1 Introduction

There is growing recognition that the XQuery XML query language proposal by the W3C [BCF+01] can rapidly gain a level of adoption within the industry comparable to the one SQL has gained over the past 20 years.

The XQuery language supports a wide variety of usage scenarios. The language design has been further shaped by a desire to include the XPath language [CD99] (with all of its sequence and tree traversal functionality) as a subset. In addition to being a feature rich language, XQuery has the desirable property of having language constructs that are fully compositional.

The evaluation of XQuery expressions represents no less of a challenge than the one faced by SQL engines. A distinctive construct in the language is the so called FLWR expression (for-let-where-return), which plays a similar role to the SQL SELECT block, and can also be arbitrarily nested. The following sample XQuery illustrates the use of nested FLWR expressions.

```
<references>
  <publication>
    <title>Answering XML Queries on Heterogeneous Data Sources</>
    <author><fname>Ioana</>  <lname>Manolescu</></>
    <author><fname>Daniela</><lname>Florescu</></>
    <author><fname>Donald</> <lname>Kossmann</></>
  </publication>
  <publication>
    <title>An XML Programming Language for Web Service
           Specification and Composition</>
    <author><fname>Daniela</><lname>Florescu</></>
    <author><fname>Donald</> <lname>Kossmann</></>
  </publication>
</references>
```

Figure 1: References.

```
<contacts>
  <author>
    <name><fname>Ioana</>  <lname>Manolescu</></>
    <address>INRIA-Rocquencourt</address>
    <email>Ioana.Manolescu@inria.fr</email>
  </author>
  <author>
    <name><fname>Daniela</><lname>Florescu</></>
    <address>Propel</address>
    <email>Daniela.Florescu@propel.com</email>  </author>
  <author>
    <name><fname>Donald</> <lname>Kossmann</></>
    <address>TU Munchen</address>
    <email>kossmann@in.tun.de</email>
  </author>
</contacts>
```

Figure 2: Contacts.

Example 1.1 Consider a first XML document `references` containing information about publications and their authors (identified by their first and last names), and a second XML document `contacts` that lists contact information for authors (mailing address and e-mail addresses). A sample instance of `references` and `contacts` appears in Figures 1 and 2, respectively. Over such documents, the XQuery expression

```
for $p in references/publication
let $pca=(for $pa in $p/author,
              $ca in contacts/author
         where $pa = $ca/name
         return $ca )
return
  <pubcontacts>
    <title>$p/title</>
    <firstauthor>$pca[1]</firstauthor>
    <otherauthors>
      for $oa in $pca[2 to last()]
      return
        <otherauthor>
          <name>$oa/name</name>
          <email>$oa/email</email>
        </otherauthor>
     </otherauthors>
   </pubcontacts>
```

produces a new document tagged `<pubcontacts>` that lists for each publication in the document `references` (in the *same order* as they appear in `references`) the contact information from the document `contacts` of the first author (`<firstauthor>`) followed by the remaining (`<otherauthors>`) authors of the publication (again, with the first and the other authors taken in the *same order* as they appear in the `references` document).

```
<pubcontacts>
  <title>Answering XML Queries on Heterogeneous Data Sources</>
  <firstauthor>
     <name><fname>Ioana</>  <lname>Manolescu</></name>
     <address>INRIA-Rocquencourt</address>
     <email>Ioana.Manolescu@inria.fr</email>
  <otherauthors>
    <otherauthor>
      <name><fname>Daniela</><lname>Florescu</></>
      <email>Daniela.Florescu@propel.com</email>
    </otherauthor>
    <otherauthor>
      <name><fname>Donald</> <lname>Kossmann</></>
      <email>kossmann@in.tun.de</email>
    </otherauthor>
  </otherauthors>
</pubcontacts>
<pubcontacts>
  <title>An XML Programming Language for Web Service
         Specification and Composition</>
  <firstauthor>
    <name><fname>Daniela</><lname>Florescu</></>
    <address>Propel</address>
    <email>Daniela.Florescu@propel.com</email>
  </firstauthor>
  <otherauthors>
    <otherauthor>
      <name><fname>Donald</> <lname>Kossmann</></>
      <email>kossmann@in.tun.de</email>
    </otherauthor>
  </otherauthors>
</pubcontacts>
```

Figure 3: EXAMPLE XQUERY RESULT

The result of the nested FLWR query on the sample instances referenced above appears in Figure 3. Note that the compositional nature of XQuery expressions allows for the example query provided above to be nested within other FLWR expressions, effectively turning the result discussed into an intermediate document created during query evaluation.

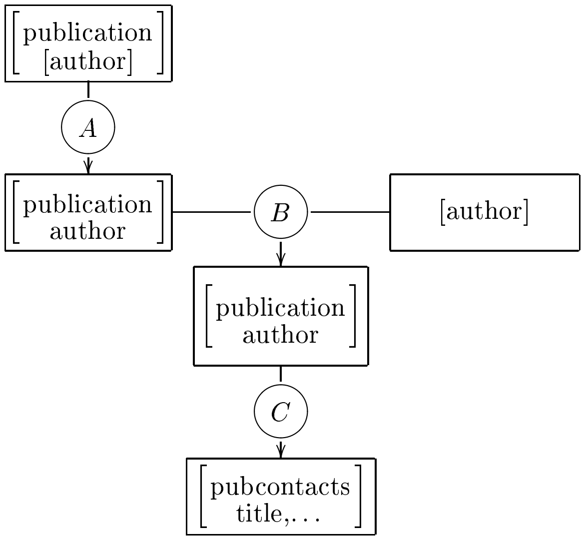

Figure 4: HIGH-LEVEL EXECUTION PLAN.

A diversity of strategies for evaluating queries over XML data has been proposed. One approach favours the implementation of XML specific query processors [MAG+97, ACV+00]. Another common approach attempts to leverage relational implementations by resorting to encoding XML data as relations and translating XML queries into relational ones [STZ+99, TDCZ02, BBM+01]. Translating XML queries into relational ones is particularly relevant when the XML data is already stored in a relational system. This may happen because relational data is used to produce XML documents, or because XML documents are stored in the RDBMS, or a combination of both. Even when the source XML is external to the relational system, there are still significant motivations to use a SQL engine as the target query processor for XQuery expressions: maturity of the implementations, extensive tuning, proven scalability, sophisticated optimizers, etc.

Despite the reasons just given for leveraging relational technology there are significant challenges in evaluating XQuery expressions with a SQL processor. The example evaluation strategy below introduces some of the issues.

Example 1.2 The (very high level) execution plan in Figure 4 can be generated to evaluate the query in Example 1.1 using a relational engine.

The boxes in the plan represent relations that encode the input XML documents, intermediate results needed during the execution, and the (encoding of) the resulting XML document. They are labeled with the main XML elements in the documents whose data is used as input to the operations A, B and C. A suitable encoding of these XML documents as relations is the first issue to be dealt with. Note that the topmost relation (representing the `references` document) has nested authors for each publication. The order and hierarchical structure of the document elements needs to be preserved in the relational encoding in the general case. There are circumstances where there are known mappings of XML elements directly into relational structures, in which case the encoding could take advantage of this information.

The plan presented above roughly proceeds as follows: operation A unnests authors within publications (while preserving order information) to enable a join operation B with the authors document. This join produces a new relation as output with pairs of publications and the contact information of their authors (which again must preserve the document order information). Notice that the join operation B corresponds to the translation of the XQuery condition on structural (or deep) equality of the author XML elements. Operation C produces the final output document by selecting the first author and other authors (taking into account the order information) for each publication.

There are challenges in choosing efficient implementations for the operations in the plan, a task that is framed by the choice of the encoding and translation used. Consider the join operation B. Using a simple merge-join algorithm would be desirable. However, for a merge-join to be applicable to B we need sufficiently simple conditions on the attributes of the input relations representing the documents. If the merge join is applicable, we also require the ability to re-order inputs (from document order) and outputs (back to document order).

The approach discussed in this paper makes possible the use of a SQL engine for efficiently evaluating XQuery expressions such as the one in the previous example. Section 2 provides a concise definition of XQuery. Section 3 introduces *dynamic intervals*, an interval based relational encoding for XML documents augmented with information tied to the query evaluation environment.

The translation of an XQuery expression into a single SQL expression is given in Section 4, fully accounting for arbitrarily nested FLWR expressions, element constructors and the language built-in functions for dealing with document structure and order.

While commercial SQL engines can be used to evaluate the resulting SQL queries produced by the translation in Section 4, there are areas where performance can be further improved. Consequently, Section 5 describes additional relational query plan operators that fully exploit properties of ordered data sets during query execution. The techniques presented in the paper enable query execution plans based on sorting and linear merge-join operations even in the presence of nested FLWR constructs (which would otherwise fall into runaway nested loop join strategies).

Section 6 provides experimental data that illustrates the behaviour of a variety of possible relational implementations of an XQuery fragment. We conclude in Section 7.

1.1 Related Work and Contributions

The use of an interval encoding for dealing with hierarchical queries in the relational world is well known [Cel94]. Manipulating SGML data in the SQL system described in [BCD+98] relied on interval processing. More recently, the behaviour of interval processing within relational systems has been studied in the context of XML containment queries [ZND+01], and [FM00, YASU01] consider the problem of mapping queries based on dialects of XPath (subsets of XQuery not including nested FLWR expressions) to SQL queries over interval encodings.

The novel aspect of the *dynamic interval* encoding proposed in this paper lies in an interval representation of an XML document during query processing that is dependent on the query.

Once a relational encoding of XML documents is selected, a translation of XQuery expressions to a relational query language (e.g., SQL) needs to be specified.

The translation presented in [MFK01] captures a substantial subset of XQuery and deals with generic documents. However, it does rely on including a transitive closure relation to encode document hierarchy (at a potential $O(n^2)$ storage cost) and it does make use of multiple SQL expressions (potentially limiting the relational optimizer opportunities).

The translation introduced in this work allow the SQL-based evaluation of arbitrarily nested FLWR expressions, preserving document ordering and supporting structural comparisons. There is no comparable analysis to the one presented in the paper that shows that predictably good query plans could be generated for the SQL produced by a generic XQuery translation working with arbitrary XML documents. Finally, while the strategies are described in the context of a relational implementation, it is clear that a native XML system can make use of them by manipulating dynamic intervals as an internal representation.

2 XQuery in a Nutshell

The goal of this section is to provide succinct definitions for basic XML documents and a query language that capture the complexities of dealing with the full XQuery language.

We first introduce a simple model for XML documents as an ordered forest of rooted, node-labeled, ordered trees.

Definition 2.1 (Simple XML Forests) Let String be a set of strings. We define the set XF of XML forests inductively by

$$\mathsf{XF} \quad = \quad [\,] \;\mid\; [\texttt{<s>}f\texttt{</s>}] \;\mid\; f@g$$

where s is in String, f and g are in XF, $[\,]$ denotes an empty forest, $[\texttt{<s>}f\texttt{</s>}]$ denotes a forest containing

Constructors:	$[\,] : \mathsf{XF}$	the empty forest constructor
	$\mathsf{XNode} : \mathsf{String} \times \mathsf{XF} \to \mathsf{XF}$	the element constructor[1]
	$@ : \mathsf{XF} \times \mathsf{XF} \to \mathsf{XF}$	the concatenation operator
Horizontal Operations:	$\mathsf{head} : \mathsf{XF} \to \mathsf{XF}$	first element of a forest
	$\mathsf{tail} : \mathsf{XF} \to \mathsf{XF}$	all but the first element of the forest
	$\mathsf{reverse} : \mathsf{XF} \to \mathsf{XF}$	the forest in reverse order (top-level only)
	$\mathsf{select}_s : \mathsf{XF} \to \mathsf{XF}$	subforest with trees labeled s in their root
	$\mathsf{distinct}_= : \mathsf{XF} \to \mathsf{XF}$	subforest of =-distinct trees (1st preserved)
	$\mathsf{sort}_< : \mathsf{XF} \to \mathsf{XF}$	a forest ordered by $<$
Vertical Operations:	$\mathsf{roots} : \mathsf{XF} \to \mathsf{XF}$	a forest of root nodes
	$\mathsf{children} : \mathsf{XF} \to \mathsf{XF}$	a forest of all children in original order
	$\mathsf{subtreesdfs} : \mathsf{XF} \to \mathsf{XF}$	a forest of all subtrees in DFS order
Boolean Operations:	$\mathsf{equal} : \mathsf{XF} \times \mathsf{XF} \to \mathsf{Bool}$	test for structural equality
	$\mathsf{less} : \mathsf{XF} \times \mathsf{XF} \to \mathsf{Bool}$	structural ordering
	$\mathsf{empty} : \mathsf{XF} \to \mathsf{Bool}$	test for emptiness

Figure 5: Basic Operations on XML Forests.

$$
\begin{array}{rcl}
[\![\, x \,]\!]\, E & = & E(x) \\
[\![\, \mathsf{XFn}(e_1, \ldots, e_k) \,]\!]\, E & = & [\![\, \mathsf{XFn} \,]\!]([\![\, e_1 \,]\!]\, E, \ldots, [\![\, e_k \,]\!]\, E) \\
[\![\, \mathsf{let}\, x = e \,\mathsf{in}\, e' \,]\!]\, E & = & [\![\, e' \,]\!](E[x := ([\![\, e \,]\!]\, E)]) \\
[\![\, \mathsf{where}\, \varphi\, \mathsf{return}\, e \,]\!]\, E & = & \text{if } [\![\, \varphi \,]\!]\, E \text{ then } [\![\, e \,]\!]\, E \text{ else } [\,] \\
[\![\, \mathsf{for}\, x \in e\, \mathsf{do}\, e' \,]\!]\, E & = & \mathsf{let}\ [v_1, \ldots, v_k] = [\![\, e \,]\!]\, E \\
 & & \quad \mathsf{in}\ [\![\, e' \,]\!](E[x := v_1]) @ \ldots @ [\![\, e' \,]\!](E[x := v_k])
\end{array}
$$

Figure 6: Semantics of FLWR-like Expressions.

a single tree with a root labeled s and an ordered forest f as children, and $f@g$ denotes a concatenation of two ordered forests.

The preceding definition has no explicit accounting of node identity, node typing, "don't care" child node order, and so on. Such features, however, can be easily added by additional encoding conventions that relate either to node labeling or to subtree patterns. A text leaf node with CDATA `text` can be encoded using the label "`text`", while an element `tag` is encoded using the label "`<tag>`". A similar approach can be taken to represent attributes, and so on.

Definition 2.2 (Minimal XQuery) The syntax for a minimal XQuery-like language is given by the following BNF rule

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \mathsf{XFn}(e_1, \ldots, e_k) \\
 & | & \mathsf{let}\, x = e \,\mathsf{in}\, e' \\
 & | & \mathsf{where}\, \varphi\, \mathsf{return}\, e \\
 & | & \mathsf{for}\, x \in e\, \mathsf{do}\, e'
\end{array}
$$

where $e, e_1, \ldots, e_k$ are expressions, x is a variable, XFn is one of the functions on XML forests given in Figure 5 and φ is a boolean condition.

In Figure 6, the semantics of these expressions is defined inductively with respect to an *environment* E that provides XML forest values for free variables, using a denotational semantics definition of the $[\![\, . \,]\!]$ map.

[1] Adds a string labelled root to a forest.

The language defined above can be seen as a reduced version of the XQuery core language employed to formalize the semantics of XQuery [FFM+01]. Despite its simplicity, the language still captures the intricate nuances present in the XQuery's FLWR expressions. The syntax allows arbitrary composition of basic function invocations, local variable definitions, filtering by Boolean conditions and iteration over the trees in XML forests.

3 Intervals and Environments

To manipulate XML documents in a relational system, we need to select an *encoding*—that is, a relational representation that captures enough information about XML to map documents to and from relations. We do this in two stages. First we encode an XML forest as triples: the node label plus the right and left endpoints of the intervals associated with the node.

Definition 3.1 (Interval Encoding) Let $f \in \mathsf{XF}$ be an XML forest. We define an instance of a ternary relation $X \subseteq \mathsf{String} \times \mathsf{Nat} \times \mathsf{Nat}$ to contain a tuple (s, l, r) for every node labelled s (node s, in short) in the forest f. The values l and r are constrained as follows:

- node s_1 is a ancestor of node s_2 in f if and only if the corresponding tuples $(s_1, l_1, r_1) \in X$, and $(s_2, l_2, r_2) \in X$ representing these two nodes satisfy $l_1 < l_2$ and $r_2 < r_1$ and
- if the node s_1 is a left sibling of the node s_2 in f then the corresponding tuples $(s_1, l_1, r_1) \in X$ and $(s_2, l_2, r_2) \in X$ satisfy $r_1 < l_2$.

We say that the (instance of the) relation X *represents*(is an encoding for) the XML forest f. We associate a value $w \in \mathsf{Nat}$ that satisfies the condition $w > \max\{r : (s, l, r) \in X\}$ with a fixed representation X of an XML forest f, and we call w the *width of* (the representation of) f.

The preceding definition of interval encoding is a very general one that captures several more specific cases. Two that are worth highlighting are the interval encoding where the endpoints denote the offsets in a string representation of the XML document and the interval encoding were the endpoints denote the ordinals associated with the encoded XML node in a pre-order and post-order traversal of the ordered XML forest.

The second stage in our definition of an encoding is to extend the representation to capture a sequence of environments (introduced in Figure 6). The motivation is to be able to simulate (with a fixed relational query) the evaluation of the FLWR queries. In the case of "$\mathsf{for}\, x \in e\, \mathsf{do}\, e'$" expressions, the semantic equation defines an iterator over a sequence of such environments; the result is then the concatenation of the results from the individual iterations. Within this sequence of environments the intervals are dynamically allocated with increasing widths to maintain the separation of XML forests belonging to each of the environments.

Definition 3.2 (Dynamic Intervals) Let $[E_1, \ldots, E_n]$ be a sequence of environments over the variables $x_1, \ldots, x_m$ of the form $E_i = \langle x_1 = t^i_1, \ldots, x_m = t^i_m \rangle$. We say that an instance of a relational schema $I, T_{x_1}, \ldots, T_{x_m}$, where $I \subseteq \mathsf{Nat}$ and $T_{x_i} \subseteq \mathsf{String} \times \mathsf{Nat} \times \mathsf{Nat}$, *represents* $[E_1, \ldots, E_n]$ if

- $I = \{i_1, \ldots, i_n\}$ such that $i_k < i_l$ whenever $k < l$ and
- $T_{x_j} = \bigcup_{k=1}^{n} \{(s, l + i_k w_{x_j}, r + i_k w_{x_j}) : (s, l, r) \in X^k_j\}$ where X^k_j represents t^k_j

for $w_{x_j} = max\{w : w \text{ width of } X^k_j\}$. We call w_{x_j} the *width* of (the representation of) x_j.

In essence, we have created an encoding for a sequence of tuples of XML forests representing the values for variables (i.e., environments), together with an *index set* that tells what range the values bound to variables occupy in the overall interval encoding. The dynamic interval representation serves the dual purpose of maintaining the separation of an arbitrary sequence of environments while at the same time being able to be interpreted as the representation for the single forest that is the result of concatenating all the environments in the sequence.

Example 3.3 We refer once more to Example 1.1 and show below the encoding for document `references` together with the environment *outside* the outermost `for` loop of the query.

i
0

s	l	r
publication	1	36
title	2	5
Answering XML ...	3	4
author	6	15
fname	7	10
Ioana	8	9
lname	11	14
Manolescu	12	13
⋮	⋮	⋮
publication	37	62
title	38	41
An XML Programming ...	39	40
author	42	51
fname	43	46
Daniela	44	45
lname	47	50
Florescu	48	49

We show below the dynamic interval encoding for the same document `references`; this time the encoding includes the environment *inside* the outermost `for` loop of the query in Example 1.1, using a width of 64 for the original document.

i
1
37

s	l	r
publication	65	100
title	66	69
Answering XML ...	67	68
author	70	79
fname	71	74
Ioana	72	73
lname	75	78
Manolescu	76	77
⋮	⋮	⋮
publication	2405	2430
title	2406	2409
An XML Programming ...	2407	2408
author	2410	2419
fname	2411	2414
Daniela	2412	2413
lname	2415	2418
Florescu	2416	2417

We conclude this section by making the observation that the ternary relation representation that has a column with the string value of the node label is not intended as the only option for a representation that associates intervals (and environments) with XML elements, attributes, CDATA, and so on. We can clearly introduce node identities and typing information to represent the actual XML data model (as opposed to the simple model presented here).

4 SQL Translation for XQuery

The translation of XQuery expressions to relational queries is presented in several steps. The first one consists of translations for the basic operations on XML forests outlined in Figure 5. The remaining steps describe a fully compositional translation for FLWR expressions that mirror the semantic rules in Figure 6.

The translation is presented as a sequence of SQL *templates* for fragments of queries that are composed to produce the final query. Each template can be understood as a relational view with table (query) parameters that are to be substituted by other templates/views. We use the notation

```
CREATE VIEW V(T1,...,Tk) AS Q(T1,...,Tk)
```

to stand for a template with parameters T1,...,Tk.

4.1 Operators

Each operator presented in Figure 5 has an associated template in the translation. For instance, the template for the roots operator is

```
CREATE VIEW ROOTS(T_e) AS
  SELECT u.s, u.l AS lr, u.r AS rr
  FROM   T_e u
  WHERE NOT EXISTS (
    SELECT *
    FROM   T_e v
    WHERE  v.l < u.l AND u.r < v.r )
```

Similarly, for each of the XFn operations we can provide a function w_{XFn} that provides the width of the relational representation generated by the template as a function of the widths of the template's inputs. Hence, we have that $w_{[]} = 0$, $w_{\mathsf{XNode}} = w_x + 2$, $w_{@} = w_{x_1} + w_{x_2}$, $w_{\mathsf{roots}} = w_x$, $w_{\mathsf{head}} = w_x$, $w_{\mathsf{tail}} = w_x$, $w_{\mathsf{reverse}} = w_x$, $w_{\mathsf{distinct}} = w_x$, $w_{\mathsf{roots}} = w_x$, $w_{\mathsf{children}} = w_x$, $w_{\mathsf{subtrees}} = w_x^2$, and so on. We show below the template for another operator using a concrete example.

Example 4.1 The template defining the view for the singleton constructor of a node labeled <pubcontacts> is

```
CREATE VIEW TXNODE_publication AS
  ( SELECT 'pubcontacts', 0 AS l, N AS r
    FROM   UNIT )
  UNION ALL
  ( SELECT s, l+1 as l, r+1 as r
    FROM   TE )
```

where UNIT is a table containing a single tuple (which is used to construct constant tuples in SQL), and N is a constant exceeding the width of TE. Note that this operation is used in the construction of the return statement producing the result for the query of Example 1.1

In what follows, we assume that $I, T_{x_1}, \ldots, T_{x_m}$ represents the sequence of environments $[E_1, \ldots, E_n]$ as in Definition 3.2. The construction is extended to FLWR expressions e; the pair I, T_e then represents the sequence of forests obtained by evaluating e in $[E_1, \ldots, E_n]$. The remaining subsections provide the definitions for the views T_e (inductively on the structure of e).

4.2 Functions

Given XFn a k-ary function on XF represented by a relational template $Q_{\mathsf{XFn}}(T_1, \ldots, T_k)$ (these are the templates introduced in the previous subsection), we define a template

```
CREATE VIEW T_XFn(e_1,...,e_m)(T_e_1, ..., T_e_m) AS
 SELECT s, l + i * w_XFn AS l, r + i * w_XFn AS r
 FROM   I, Q_XFn (
   ( SELECT s, l - i * w_e_1, r - i * w_e_1
     FROM   T_e_1
     WHERE  i * w_e_1 <= l  AND  r < (i + 1) * w_e_1 ),
     ...,
   ( SELECT s, l - i * w_e_k, r - i * w_e_k
     FROM   T_e_k
     WHERE  i * w_e_k <= l  AND  r < (i + 1) * w_e_k )
 )
```

where $Q_{\mathsf{XFn}}(T_1, \ldots, T_m)$ is the template that defines the basic function XFn (cf. Section 4.1) and w_{e_i} are the widths of the template's inputs and w_{XFn} is the width of the output. The pair of tables $I, T_{\mathsf{XFn}(e_1,\ldots,e_m)}$ then represents the sequence $[\mathsf{XFn}(t_1^1, \ldots, t_m^1), \ldots, \mathsf{XFn}(t_1^n, \ldots, t_m^n)]$.

Example 4.2 Continuing with our example, the <pubcontacts>e</pubcontacts> constructor compiles to

```
CREATE VIEW TXNODE-a-E AS
  SELECT s, l+i*52 AS l, r+i*52 AS r
  FROM  I,
    ( SELECT 'publication', 0 AS l, 51 AS r
      FROM  UNIT
    UNION ALL
      SELECT s,i l+1 AS l, r+1 AS r
      FROM
      ( SELECT s, l-i*50 AS l, r-i*50 AS r
        FROM  TE
        WHERE  i*50<=l AND r<(i+1)*50
      )
    )
```

for $w_e = 50$. Lines 4 to 7 in the SQL code above correspond to the expanded template for the constructor translation given in Example 4.1.

4.3 Assignment

Manipulating the environment is at the core of the translation of the $\mathsf{let}\, x = e \,\mathsf{in}\, e'$ assignment expression: this expression increases the size of the environment by adding the binding for $x = e$ before executing e'. Therefore, we first *extend* the relational representation of the sequence of environments. The new set of environment indices I' is defined by:

```
CREATE VIEW I' AS  SELECT * FROM I
```

and the tables holding the encodings of values for expressions e_i by:

```
CREATE VIEW T'_e_i AS  SELECT * FROM T_e_i
```

and the table T'_x holding the values for x in the new environment as a synonym for T_e in the old one:

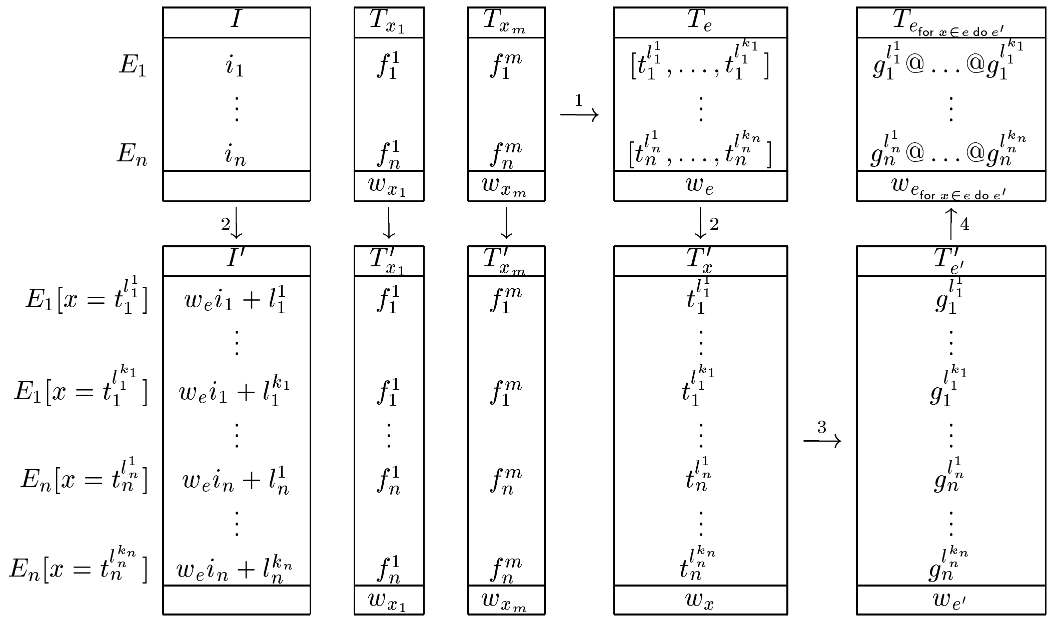

Figure 7: TRANSLATION OF "$\mathsf{for}\, x \in e \,\mathsf{do}\, e'$" IN THE ENVIRONMENT FOR $x_1, \ldots, x_m$.

CREATE VIEW T'_x AS SELECT * FROM T_e

We then apply the translation of e' with respect to this set of (primed) views I',T'_{e_i}, and T'_x obtaining $T'_{e'}$. Finally, we define the result as

CREATE VIEW $T_{\mathsf{let}\, x=e \,\mathsf{in}\, e'}$ AS SELECT * FROM $T'_{e'}$

and $w_{\mathsf{let}\, x=e \,\mathsf{in}\, e'} = w_{e'}$.

4.4 Conditional

Similarly to the assignment translation, we first define a representation $I', T'_{e_1}, \ldots, T'_{e_m}$ of a subsequence of $[E_1, \ldots, E_n]$ in which each environment satisfies φ. The new set of environment indices is defined by

```
CREATE VIEW I'(T_{e_1}, ..., T_{e_k}) AS
 SELECT i
 FROM   I
 WHERE  EXISTS Q_φ (
   ( SELECT s, l - i * w_{e_1}, r - i * w_{e_1}
     FROM   T_{e_1}
     WHERE  i * w_{e_1} ≤ l  AND r < (i + 1) * w_{e_1} ),
     ...,
   ( SELECT s, l - i * w_{e_k}, r - i * w_{e_k}
     FROM   T_{e_k}
     WHERE  i * w_{e_k} ≤ l  AND r < (i + 1) * w_{e_k} )
 )
```

and the relations representing the expressions e_i by

```
CREATE VIEW T'_{e_i} AS
 SELECT s, l, r
 FROM T_{e_i}, I'
 WHERE i * w_{e_i} ≤ l AND r < (i + 1) * w_{e_i}
```

We use this this representation to define T'_e as

CREATE VIEW $T_{\mathsf{where}\,\varphi\,\mathsf{return}\, e}$ AS SELECT * FROM T'_e

where $w_{\mathsf{where}\,\varphi\,\mathsf{return}\, e} = w_e$.

4.5 Iteration

We define $I', T'_{e_1}, \ldots, T'_{e_m}, T'_x$ to represent the sequence of environments

$$[E_1[x = t_1^{l_1^1}], \ldots, E_1[x = t_1^{l_1^{k_1}}], \ldots, \\ E_n[x = t_n^{l_n^1}], \ldots, E_1[x = t_n^{l_n^{k_n}}]$$

for $[t_i^{l_i^1}, \ldots, t_i^{l_i^{k_i}}] = [\![e]\!] E_i$, $0 < i \leq n$. We then apply the translation of the expression e' on this sequence as required by the semantics of the $\mathsf{for}\, x \in e \,\mathsf{do}\, e'$ iterator expression. The representation is defined as follows:

```
CREATE VIEW I'(T_e) AS
  SELECT i * w + lr AS i
  FROM   I, ROOTS(T_e)
  WHERE  i * w_{e_i} ≤ lr AND rr < (i + 1)w_{e_i}
```

```
CREATE VIEW T'_x(T_e) AS a
  SELECT s, l - i * w_e + (i * w_e + lr) * w_e AS l,
            r - i * w_e + (i * w_e + lr) * w_e AS r
  FROM   T_e, ROOTS(T_e), I
  WHERE  i * w_{e_i} ≤ lr AND rr < (i + 1)w_{e_i}
    AND  lr ≤ l AND r ≤ rr
```

```
CREATE VIEW T'_{e_i}(T_{e_i}, T_e) AS a
  SELECT s, l - i * w_{e_i} + (i * w_e + lr) * w_{e_i} AS l,
            r - i * w_{e_i} + (i * w_e + lr) * w_{e_i} AS r
  FROM   T_{e_i}, ROOTS(T_e), I
  WHERE  i * w_{e_i} ≤ lr AND rr < (i + 1)w_{e_i}
    AND  i * w_{e_i} ≤ l AND r < (i + 1)w_{e_i}
```

The effect of this transformation for our running example is shown in the second part of Example 3.3. On this representation we define $T'_{e'}$ inductively. The result of the iterator is then

```
CREATE VIEW T_{for x∈e do e'} AS  SELECT * FROM T'_{e'}
```

where $w_{\mathsf{for}\,x\in e\,\mathsf{do}\,e'} = w_e w_{e'}$. Note that $T_{\mathsf{for}\,x\in e\,\mathsf{do}\,e'}$ differs from $T'_{e'}$ only in its *width* (which is adjusted as a result of the concatenation of the results of the individual iterations of the $\mathsf{for}\,x \in e\,\mathsf{do}\,e'$ iterator).

Figure 7 illustrates the four steps involved in the translation of a $\mathsf{for}\,x \in e\,\mathsf{do}\,e'$ expression: the translation (1) of the subexpression e, the extension (2) of the original environment, the translation (3) of e' with respect to this new environment, and the return (4) to the original environment.

4.6 Summary of the Translation

Expressive power results characterizing the FLWR expressions as well as the translation described above are presented in [TW02], where the following result is demonstrated. Given $I, T_{x_1}, \ldots, T_{x_m}$ (a representation of the environments $[E_1, \ldots, E_n]$) and a FLWR expression e over $x_1, \ldots, x_m$ the I, T_e defined by the translation given in this section represents $[[\![e]\!] E_1, \ldots, [\![e]\!] E_n]$.

Another property of the translation is that the values of the interval encoding produced during the execution of the I, T_e queries are bounded by a polynomial in the size of the values in the encoding of the inputs. The degree of the polynomial depends only on the size (depth of nesting) of the expression e.

While the bound on the width of the intervals is a satisfactory result from the theoretical point of view, in any practical implementation we need to map the interval endpoints to a (usually fixed) machine representation of integers. However, we can always allocate sufficient number of integer-valued attributes *at query compilation time* to hold the values of the left and right endpoints of the intervals at every stage of evaluation of the given expression. Moreover, this number can be statically determined from the expression e (as the number of bits needed to represent the interval endpoints grows at most linearly with the size of the query). In addition, we have implemented a *compaction* operation that reduces the gaps in the interval encoding in time linear in the size of its input (cf. Section 5). The compaction operation can also be used for the purpose of producing an interval representation with the property of reflecting positions of the XML elements in the textual representation of the document.

Example 4.3 Consider the following query fragment, extracted from the inner loop of the query in Example 1.1.

```
for $oa in $pca[2 to last()]
return
  <otherauthor>
    <name>$oa/name</name>
    <email>$oa/email</email>
  </otherauthor>
```

Assume that we have available the relations I and T_{pca2} which encode the contents of the path expression `$pca[2 to last()]` in the current environment, and the relation $T_{ROOTSpca2}$ which is obtained by applying the roots function to the relation T_{pca2}. The variable w is a constant representing the width of T_{pca2} calculated from information in the system catalog. The query fragment is translated to the following set of SQL statements. (Note that the CREATE VIEW statements which are used for compactness of representation could be rewritten as subqueries. Also note that the given statements are the naive translation, which could be simplified in several places by exploiting characteristics specific to a particular function.)

```
CREATE VIEW I' AS
  SELECT i*2+lr AS i
  FROM   I, T_ROOTSpca2

CREATE VIEW T'_oa AS
  SELECT s, l-i*w+(i*w+lr)*w AS l,
         r-i*w+(i*w+lr)*w AS r
  FROM   T_pca2, T_ROOTSpca2, I
  WHERE  lr<=l AND r<=rr

CREATE VIEW T'_oa.name AS
  SELECT s, l+i*w AS l, r+i*w AS r
  FROM   I', (
    SELECT s, l-i*w AS l, r-i*w AS r
    FROM   T'_oa
    WHERE  i*w<=l AND r<(i+1)*w ) u
  WHERE  u.s="name"
  AND NOT EXISTS (
    SELECT *
    FROM   (
      SELECT s, l-i*w AS l, r-i*w AS r
      FROM   T'_oa
      WHERE  i*w<=l AND r<(i+1)*w ) v
    WHERE  v.l<u.l AND u.r<v.r
  )
```

The view $\mathtt{T}'_{oa.email}$ analogous to $\mathtt{T}'_{oa.name}$

```
CREATE VIEW T'_return AS
  SELECT s, l+i*(2*w+6) AS l, r+i*(2*w+6) AS r
  FROM I', (
     SELECT "otherauthor" AS s, 0 AS l, 2*w+5 AS r
     FROM UNIT
    UNION ALL
     SELECT "name" AS s, 1 AS l, w+2 AS r
     FROM UNIT
    UNION ALL
     SELECT s, 2+(l-i*w) AS l, 2+(r-i*w) AS r
     FROM T'_oa.name
     WHERE i*w<=l AND r<(i+1)*w
    UNION ALL
     SELECT "email" AS s, w+3 AS l, 2*w+4 AS r
     FROM UNIT
    UNION ALL
     SELECT s, w+4+(l-i*w) AS l, w+4+(r-i*w) AS r
     FROM T'_oa.email
     WHERE i*w<=l AND r<(i+1)*w
   )
```

The final result is in $\mathtt{T}'_{return}$, with $w_{return} = 2 * w + 6$.

5 Implementation Strategy

To make the XQuery translation practical, we need to examine if the query plans produced are amenable to efficient relational execution strategies. Indeed, we almost immediately find several patterns that negatively affect performance in of the off-shelf systems. In particular, processing joins that relate tuples via possibly several *integer order conditions* (which are frequent in the translation of basic XML operators) often lead to unacceptable performance penalty when executed using a relational system. In this section we present additional relational operators that enable the creation of efficient query plans.

A distinguishing characteristic of the operators introduced below is that they expect their inputs *sorted* by the *left endpoint* of the interval values in the encoding. The operators also maintain the same ordering in the outputs, usually running in linear time. This property allows for the creation of query execution plan fragments that involve a sequence of linear time operations. While this is possible for most of the queries that implement the *basic* operations on XML forests (cf. Figure 5), when the situation calls for a differently ordered representation the cost of reorderings is still $O(n \log n)$ time (versus falling into quadratic time operations, such as a nested loop joins).

5.1 Basic XML Operations

Consider the query that defines the ROOTS operation, given in Section 4.1). A generic relational system takes $O(n^2)$ time in worst case as the NOT EXISTS clause cannot be resolved due to the presence of *two* order selection conditions. However, the query can be implemented by an efficient plan by the addition of the operation described below.

Algorithm 5.1 Given the precondition that the tuples in the input Iterator T are ordered by l values.

```
Iterator Roots(Iterator T) {

int max=0; // distance covered by current root

Tuple TT;  // tuples for environment and tuples

Tuple fetch() {
   while (true) {
     TT = T.fetch();
     // no more tuples
     if (TT==null)  return END-OF-INPUT;
      // next root?
      if (TT.l>max) {
        max = TT.r;
        return TT;
     } // otherwise its a child; loop
   }
 }
}
```

The algorithm runs in time $O(n)$ and space $O(1)$ in the number of tuples on the input iterator.

5.2 Deep Comparisons

An important primitive operation is a deep comparison that compares sets of tuples that encode two forests to decide their relative (structural) ordering. This operation can then be used to sort representations by the structure of arbitrary XML fragments.

Using standard relational technology, the running time of a condition comparing two subsets of tuples in a representation is likely to fall into a quadratic time plan. The special purpose operation introduced below can decide deep equality in linear time (once more on the assumption that the inputs to the operation are sorted by the left side of the interval representation).

Algorithm 5.2 (Deep Comparison) Given two iterators that produce tuples sorted by their l values, the following procedure decides which of the iterators contains the smaller tree (in structural order).

```
Comparison DeepCompare(Iterator L,R) {
// returns LESS, EQUAL, or GREATER

Stack ST;    // Stack of pairs (lr,rr) of integers

Tuple TL,TR; // Holders of (s,l,r) triples

while (true) {
  // fetch next pair of tuples
  TL = L.fetch();    TR = R.fetch();
  // no more tuples
  if (TR==null)&&(TR==NULL) return EQUAL;
  // ``smaller'' forest done
  if (TL==null)             return LESS;
  if (TR==null)             return GREATER;
  // check if the nodes represented by these
  // two tuples fall in the same place in the
  // two forests
  while (ST.nonEmpty())&&
        ((TL.r<ST.top().lr)||(TR.r<ST.top().rr)) {
    // check for a ``missing sibling''
    if (TL.r>ST.top().lr)   return LESS;
    if (TR.r>ST.top().rr)   return GREATER;
    // otherwise the nodes match so far
    s.pop();
  }  // positions match--order by label:
  if (TL.s<TR.s)            return LESS;
  if (TL.s>TR.s)            return GREATER;
  // everything matches so far, so repeat
  ST.push(TL.r,TR.r)
}
}
```

The algorithm runs in time linear in the size of the XML forests and the required stack size is bounded by the *depth* of the encoded forests.

5.3 Overall Execution Strategy

The main thrust behind the approach described here is to be able to create plans that employ linear (merge) operations that take advantage of the order of the relational representation (which coincides with the document order).

There are clearly those situations, such as a join on values in the document, where requiring the processing to occur in document order causes a quadratic nested loop join to become the only option (which, as shown by the experimental results in the next section, produces totally unacceptable performance). The above circumstances validates

the importance of the property of the representation chosen: to always preserve the document order information despite reorderings.

We describe now a TreeMergeJoin operation for processing FLWR expressions of the form

$$\mathsf{for}\, x \in z/a\, \mathsf{do}\, \mathsf{for}\, y \in z/b\, \mathsf{do}\, \mathsf{where}\, x = y\, \mathsf{return}\, e$$

We first create two environments: I^x, T^x_z, T^x_x and I^y, T^y_z, T^y_y, resulting from translating $\mathsf{for}\, x \in z/a\, \mathsf{do}\, \_$ and $\mathsf{for}\, y \in z/b\, \mathsf{do}\, \_$, respectively. The two environments I^x, T^x_x and I^y, T^y_y form the inputs to the *TreeMergeJoin* operator. The operator makes use of the deep comparison operation (described in the previous subsection) to sort the forests in the two inputs, followed by the deep equality operation to merge the two streams. The output of the operator is a set of pairs of integers representing the *matching* trees in $I^x \times I^y$. The result is then joined to the T^x_z and T^y_z tables to produce the new environment needed to evaluate the expression e.

Example 5.3 Let us revisit the high level execution plan in Example 1.2 and, in light of the discussion in this section, provide more details about the operations in the plan. Operations A and C can be performed by a sequence of linear time operators working on relations in document order. Operation B corresponds to a join that can be implemented by the TreeMergeJoin described above.

6 Experimental Results

We describe below a series of experiments using a subexpression of the (by now) familiar XQuery in Example 1.1:

```
for $pa in references/publication/author,
    $ca in contacts/author
where $pa = $ca/name
return $ca
```

In the previous section we discussed a TreeMergeJoin operation that is applicable in the evaluation of the preceding query. We will compare this evaluation strategy again a number of alternatives (from nested loops to merge joins), including purely relational implementations.

The following is a list of possible plans that perform the join of n publications against m authors:

XMLNLJ A nested-loop based plan operating over encoded XML that preserves document order at all steps. This is the plan that would have to be executed if the encoding did not allow for reconstructing document order. Plans executed by vanilla relational systems are likely to perform even worse than this due to lack of support for efficient containment and deep equality operators, which this plan does use. The execution time of this plan is bounded by $O(n^2m+nm^2)$.

XMLNLJ2 (XML Nested Loops) An alternate nested-loop plan. This plan preserves the document order of the joined documents, but allows minor reordering when the inputs to the join are constructed from the results of the path expressions (in this way simulating special-purpose operations to efficiently construct join inputs). The execution time of this plan is bounded by $O(nm \log(nm))$.

XMLNLJS (XML Nested Loops with schema) A nested-loop solution that takes advantage of knowing additional schematic information to avoid deep equality comparisons. The execution time of this plan is bounded by $O(nm)$.

XMLMSJ (XML Merge-Sort) A Tree Merge-Sort based solution. Inputs to the join are sorted using deep less comparisons, and then merge joined using deep equality comparisons. The required document order is reestablished at the end of the plan by a final sort. The execution time of this plan is bounded by $O(dn \log(n) + dm \log(m))$, where d is the maximum number of nodes occurring in the join field.

XMLMSJS (XML Merge Sort with schema) A Tree Merge-Sort based solution that takes advantage of knowing additional schematic information to avoid deep Less and deep equality comparisons. The execution time of this plan is bounded by $O(n \log n + m \log m)$.

RELNLJ (Relational Nested Loops) A standard nested-loop join over a traditional relational representation of the data. This plan provides a baseline of comparison for the other nested-loop plans. Its execution time is bounded by $O(nm)$.

RELMSJ (Relational Merge Sort) A standard merge-sort join over a traditional relational representation of the data. This plan provides a baseline of comparison for the other merge-sort join plans. Its execution time is bounded by $O(n \log n + m \log m)$.

In addition to the cost of performing the join itself, there are costs associated with extracting the inputs to the join, and with constructing the results:

EXTRACT A plan that extracts the results of the path expressions into intermediate relations for input to the join. In this plan we extract directly from the encoded source document. It is important to note that if the path expressions operate on leaf documents, this stage could be vastly improved by the use of indices; however, our results demonstrate the worst-case scenario when no such indices exist (which occurs when path expression operate on the results of an outer query).

CONSTRUCT A plan that constructs the resulting XML output by calculating the contents of the `return` statement for each row in the result of the join. Note that this stage could also potentially benefit from the use of indices.

The total time to execute the XQuery on a source document is given by:

Document	DB1	DB2	DB3	DB4	DB5
XMLNLJ	1.71	13.19	1499.78	DNF	DNF
XMLNLJ2	2.45	3.49	36.32	2340.48	DNF
XMLMSJ	0.68	0.73	0.90	3.14	23.75
XMLNLJS	0.43	0.86	12.53	387.16	DNF
XMLMSJS	0.41	0.43	0.68	2.05	13.50
EXTRACT	0.97	1.14	2.30	11.83	95.30
CONSTRUCT	0.56	0.75	2.42	14.04	148.86
RELNLJ	0.42	0.72	9.99	308.70	DNF
RELMSJ	0.41	0.41	0.64	1.62	10.03

Figure 8: Running Times

EXTRACT + (join plan) + CONSTRUCT

As well, note that the times for RELNLJ and RELMSJ only include the time taken to perform the join of the relations, not the time needed to create the relational representation from the XML, nor the time needed to construct the final XML document. The experiments were conducted using a prototype relational engine, written in Java, which had been extended with the operators described in this paper. Timmings were taken using a dual-processor Pentium III 1.0GHz with 896MB RAM running RedHat Linux 7.1.

Two sets of the references and contacts documents each were used, with five different document sizes per set. The first set of documents modified the original documents described in Example 1.1 by replacing the `author` element content by a single string value. The purpose of this substitution was to allow pure relation join strategies to applicable to this set of documents. The cardinality of the elements in the set for each of the five sizes are:

DB1 10 authors and 16 publications (165 elements).
DB2 20 authors and 64 publications (551 elements).
DB3 80 authors and 512 publications (3987 elements).
DB4 320 authors and 4096 publications (30275 elements).
DB5 1280 authors and 32768 publications (235779 elements).

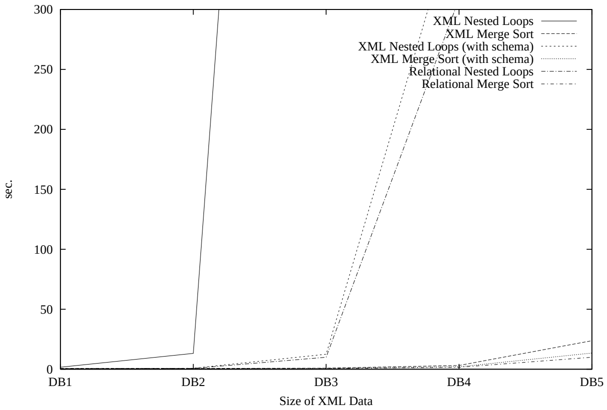

Figure 9: Comparing Approaches

The table in Figure 8 and the corresponding graphs in Figure 9 present the running times for the experiment against the documents **DB1** to **DB5**; DNF stands for executions that were terminated before completion. The second set of documents, **DB1D** to **DB5D**, had

Document	DB1D	DB2D	DB3D	DB4D	DB5D
XMLNLJ	3.88	47.29	5980.32	DNF	DNF
XMLNLJ2	2.55	3.70	41.88	4779.36	DNF
XMLMSJ	0.70	0.82	1.33	7.16	57.80
EXTRACT	0.98	1.21	2.92	16.50	146.17
CONSTRUCT	0.61	0.87	3.21	21.86	171.97

Figure 10: Running Times for Complex Values

the same cardinalities as the first, but used `author` elements with further structure, as presented in Example 1.1. On this second set of documents only plans capable of dealing with arbitrary subtrees in the join fields could be used. The table in Figure 10 presents the run times for the documents **DDB1** to **DDB5**.

The experimental data clearly shows (even on small amounts of data) the impact of the nested-loop versus merge-join decision dominating the behaviour well above any other consideration. We would also like to highlight the (relatively) low overhead of the strategy proposed in this paper (capable of dealing with generic XML) compared to completely relational strategies.

The last experiment (Figures 11 and 12) complements the above experiments by varying the size of the parts of the XML document *on which the join condition* `$pa=$ca/name` *is evaluated* from 1 to 2047 nodes while fixing the actual number of authors and publications (to 80 and 512, respectively). The experiment was performed using two different shapes for the fragment, a complete binary tree (deep) and a two-level tree (shallow), respectively. The experiment shows that, modulo the time spent in the `DeepCompare` algorithm from Section 5.2, the nested loop and merge sort join algorithms behave is essentially the same as their relational counterparts. In addition the `DeepCompare` algorithm's runtime is linear in the *size* of the compared documents and virtually independent of their structure.

7 Conclusion

We have introduced an encoding based on an interval representation of XML documents that captures additional information during query processing. We then presented a strategy for generating predictably good query plans from the SQL produced by a generic XQuery translation working with arbitrary XML documents. The resulting plans involve primitive operations that work on (almost) linear time on inputs provided in document order, and can continue maintaining the order. A deep (structural) comparison operator was introduced and its use in a merge-join strategy applicable to arbitrary XML values was discussed.

We have shown a limited overhead competing against a pure relational execution plan. However, we do not suggest employing the representation described

Nodes in join key		1	3	7	15	31	63	127	255	511	1023	2047
XMLNLJ2	deep	36.3	39.5	59.4	173.7	766.8	1576.4	DNF	DNF	DNF	DNF	DNF
	shallow	36.5	39.5	59.2	163.4	778.4	1582.4	DNF	DNF	DNF	DNF	DNF
XMLMSJS	deep	0.9	1.2	1.8	3.2	5.9	11.4	23.1	45.2	89.2	187.3	370.3
	shallow	0.9	1.1	1.8	3.2	5.9	11.4	23.2	45.5	89.5	185.0	375.9

Figure 11: Join Time relative to Join Document Size

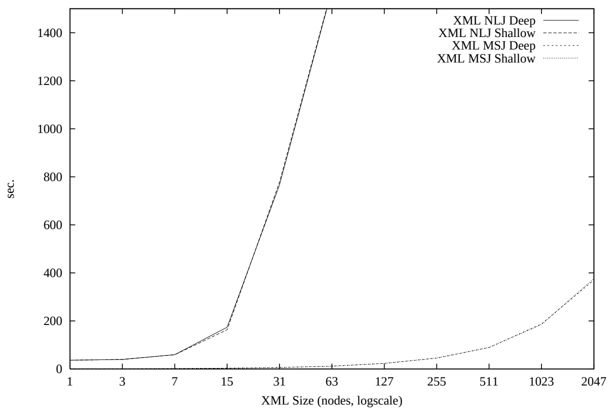

Figure 12: Time relative to Join Document Size

here when additional information allows processing to occur directly on relational data and the nature of the query is amenable to this option.

While we have concentrated on techniques that support a relational system implementation, several aspects of the strategies that we have pursued can contribute to a native XML system as well.

References

[ACV+00] V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying XML documents in Xyleme. In *Proceedings of the ACM SIGIR 2000 Workshop on XML and Information Retrieval*, 2000.

[BBM+01] D. Barbosa, A. Barta, A. O. Mendelzon, G. A. Mihaila, F. Rizzolo, and P. Rodriguez-Guianolli. ToX - the Toronto XML Engine. In *Workshop on Information Integration on the Web*, pages 66–73, 2001.

[BCD+98] L. J. Brown, M. P. Consens, I. J. Davis, C. R. Palmer, and F. W. Tompa. Structured Text ADT for object-relational databases. *Theory and Practice of Object Systems*, 4(4):227–244, 1998.

[BCF+01] S. Boag, D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Technical report, W3C, 2001.

[CD99] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Technical report, W3C, 1999.

[Cel94] J. Celko. Trees, Databases and SQL. *DBMS*, 7(10):48–57, 1994.

[FFM+01] P. Fankhauser, M. Fernndez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. Technical report, W3C, 2001.

[FM00] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In *Proceedings WebDB 2000*, volume 1997 of *Lecture Notes in Computer Science*, pages 125–136, 2000.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: a database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[MFK01] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries over Heterogenous Data Sources. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 241–250, 2001.

[STZ+99] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *The VLDB Journal*, pages 302–314, 1999.

[TDCZ02] F. Tian, D. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. Technical report, University of Wisconsin, 2002.

[TW02] D. Toman and G. E. Weddell. Querying XML: On the Utility of Interval Encoding . Technical report, University of Waterloo, 2002.

[YASU01] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, 1:110–141, 2001.

[ZND+01] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 425–436, 2001.