Cardinality-Aware *Purely* Relational XQuery Processor

Sherif Sakr

NICTA, University of New South Wales, Sydney, Australia
sherif.sakr@nicta.com.au

**Abstract.** Recently, the use of XML continues to grow in popularity, large repositories of XML documents are going to emerge, and users are likely to pose increasingly more complex queries on these data sets. In 2001 XQuery is decided by the World Wide Web Consortium (W3C) as the standard XML query language. In this article, we describe the design and implementation of an efficient and scalable purely relational XQuery processor which translates expressions of the XQuery language into their equivalent SQL evaluation scripts. The experiments of this article demonstrated the efficiency and scalability of our purely relational approach in comparison to the native XML/XQuery functionality supported by conventional RDBMSs and has shown that our purely relational approach for implementing XQuery processor deserves to be pursued further.

**Keywords**

XQuery Processor - XML - XPath - Relational Database.

1 Introduction

The eXtensible Markup Language (XML) (Bray, Paoli, Sperberg-McQueen, Maler, & Yergeau, 2006) has been introduced by the end of the 1990’s in order to create a standard data-format for the World Wide Web which can be easily handled by computers as well as by humans. In recent years, XML has found practical application in numerous domains including data interchange, streaming data and data storage. The semi structured nature of XML allows data to be represented in a considerably more flexible nature than in the traditional relational paradigm. However, the tree-based data model underlying XML poses many challenges especially with regard to the problem of performing efficient query evaluations.

As XML continues to grow in popularity, large repositories of XML documents are going to emerge, and users are likely to pose increasingly more complex queries on these data sets. Consequently, there is a great demand for efficient XML data management systems for managing complex queries over large volumes of the XML data. In 2001 XQuery is decided by the World Wide Web Consortium (W3C) as the standard XML query language (Boag et al., 2006).
XQuery is based on a hierarchical and ordered document model which supports a wide variety of constructs and use cases. The language addresses a wide range of requirements, thus incorporating a rich set of features.

The work of this article was developed within the Pathfinder project (Pathfinder., 2003). The aim of the Pathfinder project is to implement XQuery as a query language that can be used to query XML data stored on relational database systems. The architecture of Pathfinder is designed in a front-end/back-end fashion. Pathfinder receives an XQuery expression, which is parsed, normalized, and translated into XQuery Core. The Core expression is then simplified, type checked optimized, and translated into an intermediate algebraic plan. Initially, Pathfinder used the MonetDB main memory RDBMS as its target back-end. In this development branch (Boncz et al., 2006), the Pathfinder intermediate algebraic plan is translated into MIL (Monet Interpreter Language) code (Boncz & Kersten, 1999) which is then executed by the kernel of MonetDB. The MIL code generated by the Pathfinder compiler relies on some extensions added to the MonetDB back-end such as the staircase join algorithm (Grust, Keulen, & Teubner, 2003) which is designed as an efficient algorithm to evaluate XPath expressions. Although the approach of Pathfinder/MonetDB XQuery processor has been shown to be highly efficient and scalable, it is very tightly bound to the Monet DBMS and thus can not be used with any other relational back end. Another disadvantage of using MonetDB is that it requires huge main memory sizes to store large XML documents. The limitation of this approach was the main motivation behind our purely relational approach for implementing XQuery processor described in this article.

In this article, we describe the design and implementation of an efficient and scalable purely relational XQuery processor which translates expressions of the XQuery language into their equivalent SQL evaluation scripts. The proposed XQuery processor is enhanced with an accurate algebraic-based cost model (Sakr, 2007) which facilitates the processor’s ability to generate enhanced cardinality aware SQL translation scripts. Figure 1 illustrates the different alternative back-ends for the Pathfinder XQuery compiler. In particular, the main contributions of the work of this article is that it describes the design and implementation of an efficient and scalable purely relational XQuery processor. The proposed relational XQuery processor stores source XML documents in a relational repository using a tree aware relational encoding scheme and translates the XQuery expressions into SQL evaluation scripts. The main features of the proposed XQuery processor are:

- It supports an almost complete dialect of the XQuery language.
- It can reside on any relational database system and exploits its well known matured query optimization techniques as well as its efficient and scalable query processing techniques.
- It can target any RDBMS which supports the standard SQL:1999 language interface with no need for the relational database back-end to support the SQL/XML standard or to provide an XML column type of any kind.
Fig. 1. Alternative back-ends for the Pathfinder XQuery compiler.

- The relational database kernel remains untainted as there is no need for additional query processing operators or special structural join algorithms to be injected.
- It exhibits good performance characteristics when run against high-volume XML data as well as complex XQuery expressions.

The rest of this article is organized as follows. In Section 2, we give an overview of the XPath accelerator relational encoding scheme for the XML documents as the basis of our proposed XQuery processor. In Section 3, we give an overview of the loop-lifting compilation technique which translates XQuery expressions into their equivalent intermediate relational algebraic plans. Section 4 describes the design and the implementation of the cardinality-aware and purely relational XQuery engine which translates the intermediate Pathfinder algebraic plans into their equivalent SQL evaluation scripts that can be efficiently executed over any conventional RDBMS. In Section 5, we present a performance study of the Pathfinder as a purely relational XQuery engine. Section 6 reviews the related work before we conclude in Section 7.
2 XML Relational Encoding

Having an appropriate XML storage scheme is a crucial part for any relational implementation of an XQuery processor. Several research efforts have proposed different relational storage schemes for storing XML documents (O’Neil et al., 2004), (Li & Moon, 2001), (Amagasa, Yoshikawa, & Uemura, 2003), (Florescu & Kossmann, 1999). In (Yoshikawa, Amagasa, Shimura, & Uemura, 2001), Yoshikawa has classified the different XML relational storage schemes into two main classes:

1. **Structure-mapping storage**: a class of storage schemes which defines a relational schema that reflects the semantics of the XML document and makes use of its DTD or XML Schema information.

2. **Model-mapping storage**: a class of storage schemes which define a fixed relational schema that works for storing XML documents independent of the presence or absence of the XML document schema information.

One of the main focuses of this article is to present a scalable and efficient implementation of a purely relational XQuery processor. Therefore, having a proper relational storage for the XML documents is a crucial first step. The work detailed in this article is a part of the *Pathfinder* project which makes use of the *XPath Accelerator* designed by Torsten Grust (Grust, 2002) as a basis of its own encoding scheme.

2.1 XPath Accelerator

*XPath Accelerator* is an efficient, scalable and *Model-mapping* storage scheme which maps the information of the XML node hierarchy to a relational table and preserves the structural relationship between the XML nodes.
Given an XML tree $T$, its representation on persistent relational storage using \((pre/size/level)\) encoding is obtained by a single sequential document read using a normal SAX parser. During the parsing process, the pre-order rank $pre(v)$ is assigned for each node $v$. In a preorder traversal of a tree, each node is visited and assigned its pre-order rank before its children. Hence, the preorder traversal of a document’s tree representation is equivalent to its textual representation order, the document parser can assign the pre-order rank for each node when its \(startElement\) event is triggered. In addition to the pre-order rank $pre(v)$, each node descriptor also needs to include the following components:

- **Size\((v)\)**: is the number of nodes in the sub-tree below the node $v$.
- **Level\((v)\)**: represents the number of intermediate levels between the root node and a node $v$. This component is mainly used to distinguish between the children and descendant nodes for a node $v$.
- **Parent\((v)\)**: stores the pre-order rank for each node’s parent.
- **Kind\((v)\)**: stores the kind of the encoded document node. An encoded document node kind can be document, element, attribute, text, name space, or processing instruction node.
- **Name\((v)\)**: stores the tag name for the element nodes.
- **Value\((v)\)**: stores the atomic values for nodes with the kind of text or attribute and stores null for the nodes of the other types.
- **Fragment\((v)\)**: a unique document number is assigned for all nodes related to the same XML document or fragment. The main usage of this component is to distinguish between the nodes from multiple documents or fragments.

The \((pre/size/level)\) encoding naturally maps the encoded space of the XML node descriptors into a tabular relational representation. Figure 2 (c) represents an example of the relational representation of an XML document using the \((pre/size/level)\) encoding. The \((pre/size/level)\) encoding has a main important advantage as the element sub-tree copying necessary for representing the element construction expression is very easy and straightforward as we will show in Section 3.1.

### 2.2 XPath-to-SQL Translation

Based on the XPath Accelerator storage scheme, the evaluation conditions of the 12 XPath axes could be defined as depicted in Table 1.

Sample interpretations of the XPath axes evaluation conditions represented in Table 1 are given as follows:

- Given two XML nodes $x$ and $y$ in an XML tree $T$, $y$ is a child of $x$ if and only if $\text{parent}(y) = pre(x) \land \text{kind}(y) \neq att$.
- Given two XML nodes $x$ and $y$ in an XML tree $T$, $y$ is a descendant of $x$ if and only if $\text{pre}(y) > \text{pre}(x) \land \text{pre}(y) \leq \text{pre}(x) + \text{size}(x) \land \text{kind}(y) \neq att$.
- Given two XML nodes $x$ and $y$ in an XML tree $T$, $y$ is a following-sibling of $x$ if and only if $\text{pre}(y) > \text{pre}(x) \land \text{parent}(y) = \text{parent}(x) \land \text{kind}(y) \neq att$. 
<table>
<thead>
<tr>
<th>XPath Axis</th>
<th>Axis Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>( \text{pre}(x) = \text{pre}(y) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>attribute</td>
<td>( \text{parent}(y) = \text{pre}(x) \land \text{kind}(y) = \text{att} )</td>
</tr>
<tr>
<td>parent</td>
<td>( \text{parent}(x) = \text{pre}(y) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>child</td>
<td>( \text{parent}(y) = \text{pre}(x) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>descendant</td>
<td>( \text{pre}(y) &gt; \text{pre}(x) \land \text{pre}(y) \leq \text{pre}(x) + \text{size}(x) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>( \text{pre}(y) \geq \text{pre}(x) \land \text{pre}(y) \leq \text{pre}(x) + \text{size}(x) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>ancestor</td>
<td>( \text{pre}(y) &lt; \text{pre}(x) \land \text{pre}(y) \leq \text{pre}(y) + \text{size}(y) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>( \text{pre}(y) \leq \text{pre}(x) \land \text{pre}(x) \leq \text{pre}(y) + \text{size}(y) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>following</td>
<td>( \text{pre}(y) &gt; \text{pre}(x) \land \text{parent}(y) = \text{parent}(x) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>following-sibling</td>
<td>( \text{pre}(y) &gt; \text{pre}(x) \land \text{parent}(y) = \text{parent}(x) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>preceding</td>
<td>( \text{pre}(y) + \text{size}(y) &lt; \text{pre}(x) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td>( \text{pre}(y) &lt; \text{pre}(x) \land \text{parent}(y) = \text{parent}(x) \land \text{kind}(y) \neq \text{att} )</td>
</tr>
</tbody>
</table>

Table 1. XPath axes evaluation conditions

Translating the XPath expressions into SQL Queries is a straightforward process by using the available knowledge of the encoded XML document trees in the XPath Accelerator document table and the defined XPath axes conditions. An XPath expression with a series of location steps represented as \( S_1/S_2/.../S_n \) is converted into a series of pipelined compositional \( n \) region queries where the node sequence output by axis step \( S_i \) is the context node sequence for the subsequent step \( S_{i+1} \). Figure 3 shows an example for the evaluation of the XPath expression \( v/\text{preceding}/\text{ancestor} \). In the shown example, the referenced document table represents the relational table representation of the \( (\text{pre}/\text{size}/\text{level}) \) encoding of the target XML document (an example of this document table was presented in Figure 2 (c)). The referenced context table represents the sequence of the input XML context nodes. The evaluation condition of the \( \text{preceding} \) axis is represented in line 4 and the evaluation conditions of the \( \text{ancestor} \) axis are represented in lines 5 and 6 of the SQL Query. Since the XPath semantics (Draper et al., 2006) requires that the result of an XPath expression to be duplicate-free and represented in the document order, we had to use the SQL constructs of \text{DISTINCT} \ and \text{ORDER BY} \ in the result SQL evaluation query.

```
1 SELECT DISTINCT doc3.*
2 FROM context AS ctx, document AS doc1, document AS doc2, document AS doc3
3 WHERE ctx.pre=doc1.pre
4 AND doc2.pre + doc2.size < doc1.pre
5 AND doc3.pre < doc2.pre AND doc2.pre <= doc3.pre + doc3.size
6 AND doc3.kind <> 'Att'
7 ORDER BY doc3.pre
```

Fig. 3. An example for the SQL evaluation for an XPath expression based on the XPath accelerator encoding scheme.
3 XQuery Algebraic Compilation

Relational algebra has been a main component in relational database systems, and has played an important role in their success for gaining widespread usage. A corresponding XML algebra would have the same importance and substantial role for XML query processing. In our context, having an adequate algebraic compilation for XQuery expressions provides us with the solid infrastructure for predicitng the cardinality of the main XQuery expressions and its sub-expressions in a very convenient and accurate way. Many proposals for an algebra for XML query processing have been introduced (Brantner, Helmer, Kanne, & Moerkotte, 2005), (Chen, Jagadish, Lakshmanan, & Paparizos, 2003), (Sartiani & Albano, 2002), (Jagadish, Lakshmanan, Srivastava, & Thompson, 2001), (Paparizos, Al-Khalifa, Jagadish, Niermann, & Wu, 2002), (H.Zhang & F.W.Tompa, 2003). According to (Re, Siméon, & Fernández, 2006) existing algebras for XQuery fall into two classes:

- **Tuple-based algebra**: this class of algebra tries to facilitate the use of relational optimization techniques as well as the whole relational query processing framework (theory, compilation, optimization, execution).
- **Tree-based algebra**: this class of algebra provides more natural support for novel XML-specific optimizations and manipulates XML data modelled as forests of labelled ordered trees.

The *Pathfinder* project has a special module for compiling XQuery expressions into its own dialect of tuple-based algebra producing equivalent relational query plans. The *Pathfinder* algebra is quite primitive such that it can efficiently fit within the capabilities of SQL-based systems. *Pathfinder* compiles the XQuery core dialect listed in Table 2 into relational query plans using the set of algebraic operators listed in Table 3.

<table>
<thead>
<tr>
<th>atomic literals</th>
<th>document order (e1 &lt;&lt; e2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequences(e1,e2)</td>
<td>node identity (e1 is e2)</td>
</tr>
<tr>
<td>variables</td>
<td>arithmetics (+,-,*,...)</td>
</tr>
<tr>
<td>let $v:=e1 return e2</td>
<td>comparisons (=, &lt;, &gt;, ...)</td>
</tr>
<tr>
<td>for $v [at$p] in e1 return e2</td>
<td>Boolean connectives (and, or)</td>
</tr>
<tr>
<td>if (e1) then e2 else e3</td>
<td>user-defined functions</td>
</tr>
<tr>
<td>e1 order by e2,...,en</td>
<td>fn:doc(.), fn:root(.), fn:data(.)</td>
</tr>
<tr>
<td>unordered {e}</td>
<td>fn:id(.), fn:idref(.)</td>
</tr>
<tr>
<td>element {e1},{e2}</td>
<td>fn:distinct-values(.)</td>
</tr>
<tr>
<td>attribute {e1},{e2}</td>
<td>op:union(.), op:intersect(.), op:difference(.)</td>
</tr>
<tr>
<td>text {e}</td>
<td>fn:count(.), fn:sum(.), fn:max(.)</td>
</tr>
<tr>
<td>XPath Location Steps</td>
<td>fn:position(.), fn:last(.)</td>
</tr>
</tbody>
</table>

Table 2. Pathfinder supported XQuery dialect
### 3.1 Pathfinder Algebraic Operators

The *Pathfinder* algebraic operators are designed to receive one or more inputs and produce one or more outputs. These inputs and outputs are in the form of sets of tuples. Most of the operators perform quite simple and standard relational operations which allow the query optimizer to employ the usual relational algebraic optimization techniques such as the pushdown of the selection operator. In this section, we will first provide a detailed description for the operators and then in Section 4.2, we define their associated SQL translation templates. Figure 4 illustrates the behaviour for some of the *Pathfinder* algebraic operators.

- **Projection / Attribute Renaming:** The project operator ($\pi_{a_1:b_1,\ldots,a_n:b_n}$) receives a relational table $R$ and a list of projection attributes ($a_1, a_2, \ldots, a_n$). The operator returns the tuples of the relation $R$ with a schema filtered to the list of the projected attributes. An optional attribute renaming operation could be done during the projection operation by changing the attribute names of the schema from ($a_1, a_2, \ldots, a_n$) to ($b_1, b_2, \ldots, b_n$) according to the specification of each renaming pair($a_1 : b_1$). Due to the design of the *Pathfinder* compilation rules, the project operator has a specific property that it does not need to perform any duplicates removal operation.

- **Attachment:** The attachment operator ($\oplus_{a:v}$) receives a relational table $R$ and a list of attaching attributes ($a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n$). It appends one or more new attributes $a_i$ to the tuples of the input relation $R$. The values for the new attributes are assigned the value $v_i$.

- **Selection:** The selection operator ($\sigma_a$) receives a relational table $R$ and a Boolean selection attribute $a$. It returns a new relation where the tuples of $R$ which have the value of the attribute $a$ equal to $false$ are missing. The Boolean selection attributes ($a$) are usually produced using the comparison operator ($\oplus$).

- **Disjoint Union:** The union operator ($\cup$) is a common relational operator. It receives two input relations $R$ and $S$. It ensures that the input relations are union compatible and returns one relation containing the tuples of the two input relations. In Pathfinder algebraic plans, the two input relations of
the union operator are guaranteed to be always disjoint due to the design of the Pathfinder compilation rules.

- **Difference**: The difference operator (\) receives two relations \( R \) and \( S \) and returns all tuples of the first input relation \( R \) that have no matching tuples in the second relation \( S \).

- **Cartesian Product**: The Cartesian product operator (\( \times \)) receives two input relations \( R \) and \( S \). It returns one relation which combines each tuple of the first relation \( R \) with all the tuples of the second relation \( S \). The output relation has a schema which concatenates the schemes of the two input relations.

- **Equi-Join**: The equi-Join operator (\( \setminus \{ a = b \} \)) is a well-known standard relational operator. It receives two input relations \( R \) and \( S \) as well as a predicate \( P \) (In Pathfinder algebraic plans, the predicate \( P \) is always an equality predicate of the form \( a = b \)). It evaluates the predicate \( P \) over each pair of tuples \((t_1, t_2) \in (R \times S)\), returning only the pairs satisfying \( P \). The primary usage of the Equi-Join operator in Pathfinder’s algebraic plan is the representation of the iteration concept for the FLWOR expression as it will be shown in Section 3.2

- **Duplicate Elimination**: The duplicate elimination operator (\( \delta \)) is used to remove the duplicate tuples in the required places inside the relational plan. This operator is a specific operator that is unique to the Pathfinder algebra and thus there is no equivalent operator in the conventional relational algebra.

- **Unsorted Row Numbering**: The unsorted row numbering operator (\( \#_a \)) and the sorted row numbering operator (\( \varrho_a : (o_1, \ldots, o_n) / p \)) are two of the most important Pathfinder’s algebraic operators. Together, they are responsible of preserving the order concept defined by the XQuery/XPath specifications (Fernández, Malhotra, Marsh, Nagy, & Walsh, 2006). The unsorted row numbering operator (\( \#_a \)) receives an input relation \( R \) and returns the same relation extended with a new consecutive numbering attribute \( a \) from 1 to \( n \) where \( n \) is the cardinality of the input relation \( R \).

- **Sorted Row Numbering**: The sorted row numbering operator (\( \varrho_a : (o_1, \ldots, o_n) / p \)) receives an input relation \( R \), an ordering attribute list \( (o_1, \ldots, o_n) \) and an optional partitioning attribute \( p \). It returns the same relation extended with a new consecutive numbering attribute \( a \). The numbering attribute \( a \) respects the tuple sorting of relation \( R \) defined by the order specification \( (o_1, \ldots, o_n) \) and restarts the numbering from 1 for each partition defined by the optional partitioning attribute \( p \).

- **Arithmetic**: The arithmetic operator (\( \odot_a : (b,c) \)) represents the standard arithmetic operations (+, −, *, \( \backslash \), mod, div). It receives a relational table \( R \) and a pair of argument attributes \( (b,c) \). The output relation is the input relation extended with one more attribute \( a \) representing the result of applying the arithmetic operation \( \odot \) over the two argument attributes \( (b,c) \).

- **Comparison**: The comparison operator (\( \ominus_a : (b,c) \)) represents the standard comparison operations (<, \( \leq \), =, \( \neq \), >, \( \geq \)). It receives a relational table \( R \) and a pair of comparison attributes \( (b,c) \). The output relation is the input relation extended with one more attribute \( a \) representing the result of applying the comparison operation \( \ominus \) over the two argument attributes \( (b,c) \).
Fig. 4. Illustrating Examples for the behavior of some Pathfinder algebraic operators.
relation extended with one more attribute (\(a\)) that represents the result of applying the comparison operation (\(\circ\)) over the two comparing attributes (\(b, c\)).

- **Negation**: The negation operator (\(\neg_{\alpha}\)) receives a relational table \(R\) and a Boolean attribute (\(b\)). The output relation is the input relation extended with one more attribute (\(a\)) representing the negations of the Boolean values of the attribute (\(b\)).

- **Aggregation**: The aggregation operator (\(Agg_{\alpha/p}\)) represents the standard aggregation operations (Count, Sum, Min, Max, Average). The Pathfinder aggregate operator appears in Pathfinder algebraic plans in one of the following two possible versions:
  1. The first version, receives a relation \(R\) and an aggregate attribute (\(a\)). It returns a single value which is computed by applying the aggregate function \(Agg\) on the values of the aggregate attribute (\(a\)) for the input tuples.
  2. The second version, receives a relation \(R\), an argument attribute (\(a\)) and an grouping attribute (\(p\)). In this version, the aggregate operator returns a binary relation \((p,a)\) which is computed by applying the aggregate function \(Agg\) on the values of the aggregate attribute \(a\) for the input tuples grouped by the values of the attribute \(p\). Usually, the grouping attribute of an input relation \(R\) represents its iterating attribute \(iter\).

- **Tables**: The tables operator is normally used in Pathfinder algebraic plan for the representation of literals. It receives no input, is represented as a relational table storing the values of the literals and is usually preceded with an attachment (@) operator.

- **Document Access**: The document access operator (\(\Delta_{nvalue:item}\)) receives three inputs:
  1. A context relation (\(ctx\)).
  2. An node identification attribute (\(item\)).
  3. An encoded XPath accelerator relation for the associated live nodes fragment (\(\Gamma\)) (usually represented by the encoded XPath accelerator document table or transient nodes for XML document fragments constructed at the runtime using the construction operators (\(\varepsilon/\tau\))).

The document access operator applies a join operation between the context relation and the live nodes fragment relation (\(ctx \bowtie_{(item=pre)} \Gamma\)) and returns the input relation \(ctx\) extended with one more node values attribute (\(nvalue\)). The values of the new attribute are computed by retrieving the (\(value\)) attribute from the relation representing the input live nodes fragment (\(\Gamma\)) for the tuples with the pre-order values contained in the node identification attribute (\(item\)) of the input relation (\(ctx\)).

- **XPath Location Step Evaluator**: The XPath location step evaluator operator (\(\xi_{item:(a,n)}\)) is another very important and special Pathfinder algebraic operator. This operator is responsible for evaluating XPath expressions. In the general case, the operator is independent of the relational document encoding used for storing the XML documents and the technique used
for evaluating the XPath expressions. Hence, in our context, it is based on our use of the *XPath accelerator* encoding scheme and the mechanism of evaluating XPath expression described in Section 2.2. The XPath Evaluator operator (eval) receives three inputs:

1. A context relation (ctx) which stores the node identifiers of the input context nodes.
2. An encoded XPath accelerator relation for the associated live nodes fragment (Γ).
3. The evaluated XPath step (ctx/α :: n), where α represents the XPath location step’s axis and n represents the location step’s node test (n).

The XPath Evaluator operator applies the evaluation conditions of the XPath step (α,n) over the input context nodes (ctx) using the information of the live nodes fragment(Γ) and returns the result as a sequence of tuples representing the node identifiers of the resulting nodes. The resulting nodes from applying the XPath Evaluator operator are: duplicate free, preserving the document order, and maintaining both of the iteration and fragment information of the input context nodes relation (ctx).

**Element / Text Construction:** The XQuery language provides the capability of creating transient nodes and document fragments during the runtime execution of the XQuery expressions. The element / text Construction operator (ε/τ(e₁,e₂)) is responsible for implementing such functionality in the Pathfinder algebra. The implementation of the element construction operator is a bit more complex and is thus not as straightforward as the rest of the operators. It receives three inputs:

1. An encoded XPath accelerator relation for the associated live nodes fragment (Γᵢₕ) (Figure 5 (a)).
2. An element names relation (T) that stores the tag names of the newly constructed node (one tuple per iteration) (Figure 5 (b)).
3. A context relation (ctx) that stores the node identifiers of the input context nodes (Figure 5 (c)).

The Element Construction operator (ε) processes its input relations and returns two output relations as follows:

1. An output live nodes relation (Γᵢₜ) (Figure 5 (d)). This relation extends the input live nodes relation (Γᵢₗ) with the tuples representing the newly created transient nodes. The new tuples of the transient nodes are created as follows:
   - For each tuple in the element names relation (T), we insert a new tuple into the output live nodes relation (Γᵢₜ) representing the root node of the new subtree.
   - For each tuple with node identifier in the context relation (ctx), we insert a new tuple into the output live nodes relation (Γᵢₜ) and copy the tuples from the input live nodes relation (Γᵢₗ) representing its associated subtree.
   - Each iteration in the element names relation (T) with no context nodes in the context relation (ctx) will be represented with only a single tuple for the root node with the size of 0 for an empty subtree.
Fig. 5. An illustrating example for the behavior of the element construction operator.
Fig. 6. An illustrating example for the behavior of the text construction operator.
• The pre-order values of the new tuples (transient nodes) are generated in a consecutive manner starting from the maximum pre-order value of the input live nodes relation (Γ_{in}) + 1.
• Each group of a new root node and its associated copied subtree for the same iteration is represented as a new fragment.

2. An output context relation (Figure 5 (e)). This relation stores the pre-order values of the newly constructed transient root nodes and is used as an input context relation for the immediate parent operator in the relational plan.

Figure 5 shows an example describing the behavior of the element construction operator.

In principal, the text construction operator (\( \tau(e_1,e_2) \)) is responsible of converting the string values resulting from the valuation of XQuery expression \( e_2 \) into text nodes with tag names specified by the XQuery expression \( e_1 \). Hence, the behavior of the text construction is very similar to the element construction operator with the following two differences:

(i) In the input context relation (ctx), the item column stores the strings evaluating the expression \( e_2 \) and which will be converted to text nodes instead of storing the node identifiers of the nodes to be constructed in the case of the element construction operator.
(ii) For each tuple with string value in the context relation (ctx), we only insert a new tuple into the output live nodes relation (Γ_{out}) with no need for subtrees copying (in element construction we need).

Figure 6 shows an example describing the behavior of the text construction operator.

3.2 Compiling XQuery into Relational Algebra

*Pathfinder* compiles the XQuery core dialect listed in Table 2 into relational query plans using the algebraic operators described in Section 3.1. In this section, we give an overview of the *Pathfinder loop-lifting* technique, which is considered to be the heart of this compilation process. For a detailed and complete description of this technique and its translation rules we refer to (Teubner, 2006),(Grust, Sakr, & Teubner, 2004),(Grust, 2005).

In a nutshell, loop-lifting is the key technique used to compile XQuery iterations into efficient bulk style application of algebraic operators. The principal idea behind the compilation scheme is that every XQuery expression occurs in the scope of an iteration. The iterations of each scope are encoded by a column \( \text{iter} \) in the associated relational representation. In order to clarify the loop-lifting idea, we will show some examples of *Pathfinder loop-lifted* translations for some XQuery expressions into their associated *Pathfinder* algebraic relational representation.

**Sequences** The XQuery language is designed to operate over ordered, finite sequences of items as its principal data type. The evaluation of any XQuery expression yields an ordered sequence of \( n \geq 0 \) items. These items can be either
atomic values (integers, strings, ..., etc) or XML tree nodes. An XQuery item sequence \((x_1, ..., x_n)\) is encoded with the following relational table:

<table>
<thead>
<tr>
<th>pos</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(x_1)</td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(n)</td>
<td>(x_n)</td>
</tr>
</tbody>
</table>

The column \(pos\) is used to preserve the order information between the items inside the target sequence. The \(item\) column is a polymorphic column. In the case of atomic items, it stores the values of the encoded atomic items (1,"A",...) and in the other case of XML tree nodes, it stores the pre-order ranks of the encoded nodes. The RDBMS supports the representation of such polymorphic columns using the \(Variant\) data type. The empty sequence () is encoded with an empty table with the same schema \((pos,item)\).

**FLWOR Expressions** The FLWOR expression is one of the main features provided by the XQuery language. It is used for representing iterations and for the binding of variables to intermediate results. It is also used for computing joins between two or more sequences and for restructuring data. A loop of \(n\) iterations is represented by a relation \(loop\) with a single column \(iter\) of \(n\) values (1,2,...,\(n\)).

<table>
<thead>
<tr>
<th>iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>(\vdots)</td>
</tr>
<tr>
<td>(n)</td>
</tr>
</tbody>
</table>

Based on the relational representation for the sequence \((x_1, ..., x_n)\) we presented in the previous section, we can now represent the compilation of variables bound in the iterations of FLWOR expression using the following XQuery for-loop example:

```xquery
for $v$ in \((x_1, ..., x_n)\) return e.
```

This example expression binds each \(x_i\) item to variable \($v$\) and evaluates the loop body \(e\) for each iteration. The relational encoding of the variable \($v$\) has the following form:

<table>
<thead>
<tr>
<th>iter pos item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 (x_1)</td>
</tr>
<tr>
<td>2 1 (x_2)</td>
</tr>
<tr>
<td>(\vdots) (\vdots) (\vdots)</td>
</tr>
<tr>
<td>(n) 1 (x_n)</td>
</tr>
</tbody>
</table>

This translation encodes all bindings of \($v$\) in a single relation. In general, each tuple of the encoding relation \((i,p,x)\) indicates that for the \(i\)-th iteration, the item at position \(p\) stores the value \(x\).
**Path Steps** The compilation of path steps is represented in *Pathfinder* algebraic plans using the XP evaluator operator ($\langle\rangle$). It takes a context relation (iter, item) as input, where the item column stores the node identifiers of the input context nodes. It uses the relation of the current live nodes $\Gamma$(*pre*, *size*, *level*, *kind*, *name*, *value*, *frag*) to evaluate the path step ($\alpha$, *n*) and returns a new relation with the same schema (iter, item). In the output relation, the item column stores the node identifiers of the resulting nodes. Figure 7 illustrates an example for the loop-lifted compilation of the path step where Figure 7(a) represents the input context nodes, Figure 7(b) represents the sample source XML document and Figure 7(c) represents the output context nodes from applying the path step *child::c* over input context nodes and the sample source XML document.

![Figure 7](image_url)

**Arithmetic and Comparison Expressions** Given the relational representation $R_1$(*iter1*, *item1*) and $R_2$(*iter2*, *item2*) of two XQuery expression $e_1$ and $e_2$, the design of the loop-lifting compilation of the arithmetic and comparison expressions requires that the schema of the relational representation ($R_i$) of each argument expression ($e_i$) must have a single node identifier column (*item_i*) for each iteration (*iter_i*) with no order preserving column (*pos*). The arithmetic expression $e_1 \circ e_2$ is evaluated by joining $R_1$ and $R_2$ over their iterations attribute *iter*, for each tuple of the result we apply the arithmetic operator ($\circ$) over the two arguments columns *item1* and *item2*, and store the result in a new column *res*. The algebraic representation of the arithmetic expression ($e_1 \circ e_2$) is defined as follows:

$$e_1 \circ e_2 \Rightarrow \pi_{iter1, iter2, res}(\circ_{res}(\cdot\cdot\cdot(iter1, item1)\cdot\cdot\cdot)\cdot\cdot\cdot(iter2, item2)\cdot\cdot\cdot)\cdot\cdot\cdot)$$

where $R_1$ and $R_2$ are consequentially representing the relational representation of the two expressions $e_1$ and $e_2$. 
Similarly, the comparison expression \( e_1 \otimes e_2 \) is evaluated by joining \( R_1 \) and \( R_2 \) over their iterations attribute \( \text{iter} \), for each tuple of the result we apply the comparison operator \((\circ)\) over the two argument attributes \( \text{item}_1 \) and \( \text{item}_2 \), and store the result in a new attribute \( \text{res} \). The comparison operators are normally followed by a selection operator to filter the tuples satisfying the comparison condition. The algebraic representation of the comparison expression \((e_1 \otimes e_2)\) is defined as follows:

\[
e_1 \circ e_2 \Rightarrow \sigma_{\text{res}}(\pi_{\text{iter}_1:\text{iter},\text{res}}(\bowtie_{\text{res}:\langle \text{item}_1,\text{item}_2 \rangle}(R_1 \bowdle\text{iter}_1=\text{iter}_2 \bowdle R_2)))
\]

4 Cardinality Aware XQuery-to-SQL Translation

4.1 XQuery-to-SQL Framework

Figure 8 represents the framework of Pathifier as a relational XQuery engine. The query engine translates its input of XQuery expressions into SQL evaluation scripts through the following steps:

1. The source XML documents that are processed by the engine have to be mapped (shredded) into the XPath Accelerator relational scheme using the \((\text{pre}/\text{size}/\text{level})\) encoding introduced in Section 2.1.

Fig. 8. The framework of Pathfinder relational XQuery engine.
2. Based on the list of *Pathfinder* algebraic operators and the loop-lifting compilation technique introduced in Section 3, the *Pathfinder* algebraic translation module translates the input XQuery expression into its equivalent relational algebraic plan.

3. The SQL generator module receives the algebraic plans produced in step two as well as the SQL translation templates (Section 4.2) to generate the equivalent SQL script for evaluating the results of the input XQuery expression.

4. The SQL script generated in step three is using a standard SQL: 1999 code which can be executed on any conventional RDBMS such as IBM DB2, Microsoft SQL server and Oracle.

5. An XML serialization module receives the relational tuples resulting from the evaluation process of the SQL script using the conventional RDBMS and serializes them into an equivalent output of the XML model. We refer to (Grust, Keulen, & Teubner, 2004) for a detailed description of this serialization algorithm.

### 4.2 Translation Templates

For each of the *Pathfinder* algebraic operators described in Section 3.1, we define a corresponding SQL translation template. During the SQL translations, the parameters for the algebraic operator is properly transformed from one operator to another according to the DAG plan. Figures 9 and 10 illustrate the inference rules for translating *Pathfinder* algebraic operators into SQL code. Remarks about the inference rules of SQL translation is given as follows:

- A sample interpretation of the inference rule Trans-1 representing the translation of *Pathfinder* algebraic project operator ($\pi$) is:

  Given the information that the relational object *(Table-View)* $q$ represents the SQL evaluation of the expression $e$, the translation of the *Pathfinder* algebraic project operator ($\pi_{a_1,a_2:b_2,...,a_n(e)}$) is defined using the following SQL code:

  ```sql
  CREATE VIEW op_i AS
  SELECT a_1, a_2 AS b_2, ..., a_n FROM q;
  ```

  where $i$ represents the operator number of the translated project operator in the associated algebraic plan.

- In the inference rule Trans-5 for translating the difference operator ($\setminus$), the notation $q_1.Schema : (f_1, f_2)$ means that the schema of the relational object $q$ consists of the columns $(f_1, f_2)$.

- In the inference rule Trans-14 for translating the tables operator $\downarrow$, the referenced relational table *OneRowTable* represents a standard data dictionary table for any RDBMS that consists exactly of one column and one record.
In the inference rule Trans-18 for translating the XPath location step evaluator operator \( (\xi_{item:\alpha.(\alpha,n)}(e)) \), the referenced function XPathEvaluationConditions (\( d_{\alpha.(\alpha,n)} \)) represents the evaluation conditions of the different XPath location steps which was described earlier in Section 2.2.

Due to the complexity of the element / Text construction operators (\( \varepsilon/\tau \)), we do not use the inference rule notation for representing its translation. A detailed explanation of the translation steps of these operators will be presented as follows:

Element Construction Translation

Input Parameters:
1. The live nodes fragment: \( OpLiveNodes \).

<table>
<thead>
<tr>
<th>Pre</th>
<th>Size</th>
<th>Level</th>
<th>Kind</th>
<th>Name</th>
<th>Value</th>
<th>Frag</th>
</tr>
</thead>
</table>

2. The relation of the element names: \( OpElementNames \).

<table>
<thead>
<tr>
<th>Iter</th>
<th>Name</th>
</tr>
</thead>
</table>

3. The input context relation: \( OpContext \).

<table>
<thead>
<tr>
<th>Iter</th>
<th>Item</th>
<th>Pos</th>
</tr>
</thead>
</table>

Step 1: Creating the output live nodes fragments.

```sql
CREATE VIEW OpOutputLiveNodes AS
SELECT MaxPre.pre +
    ROW_NUMBER() OVER (ORDER BY iter,pos,
        ElementConstruction.pre) AS pre,
    ElementConstruction.size, ElementConstruction.level,
    ElementConstruction.kind, ElementConstruction.name,
    ElementConstruction.value,
    MaxPre.frag + DENSE_RANK() OVER (ORDER BY iter) AS frag
FROM (SELECT OPElementNames.iter, 0 AS pos, -2 AS pre,
    COALESCE(SUM(size + 1), 0) AS size,
    0 AS level, 'Elem' AS kind,
    OpElementNames.item AS name,
    NULL AS value
    FROM OpContext INNER JOIN OpLiveNodes
    ON OpContext.item = OpLiveNodes.pre
    RIGHT OUTER JOIN OpElementNames
    ON OpElementNames.iter = OpContext.iter
GROUP BY OpElementNames.iter,OpElementNames.item
UNION ALL
SELECT OpContext.iter, OpContext.pos,
```

```sql
```
Fig. 9. Inference rules of SQL translation (1).
\[
\text{SQL}(e) : q
\]

\[
\text{SQL}(\neg a \land b(e)) \Rightarrow
\begin{align*}
&\text{CREATE VIEW } \text{opi} \text{ AS} \\
&\text{SELECT } q.*, 'True' \text{ AS a} \\
&\text{FROM } q \\
&\text{WHERE } b = 'True' \\
&\text{UNION ALL} \\
&\text{SELECT } q.*, 'False' \text{ AS a} \\
&\text{FROM } q \\
&\text{WHERE } b = 'False';
\end{align*}
\] (TRANS-13)

\[
\text{SQL}(\exists c) \Rightarrow
\begin{align*}
&\text{CREATE VIEW } \text{opi} \text{ AS} \\
&\text{SELECT } v \text{ AS c} \\
&\text{FROM OneRowTable;}
\end{align*}
\] (TRANS-14)

\[
\text{SQL}(e) : q \quad \text{Agg} \in \{\text{count, max, min, sum, avg}\}
\]

\[
\text{SQL}(\text{Agg}_{v:a}(e)) \Rightarrow
\begin{align*}
&\text{CREATE VIEW } \text{opi} \text{ AS} \\
&\text{SELECT } \text{Agg}(a) \text{ AS v} \\
&\text{FROM } q;
\end{align*}
\] (TRANS-15)

\[
\text{SQL}(e) : q \quad \text{Agg} \in \{\text{count, max, min, sum, avg}\}
\]

\[
\text{SQL}(\text{Agg}_{v:a/p}(e)) \Rightarrow
\begin{align*}
&\text{CREATE VIEW } \text{opi} \text{ AS} \\
&\text{SELECT } p, \text{Agg}(a) \text{ AS v} \\
&\text{FROM } q \\
&\text{GROUP BY } p;
\end{align*}
\] (TRANS-16)

\[
\text{SQL}(e) : q \quad \text{SQL}(\Gamma) : \text{OpLiveNodes}
\]

\[
\text{OpLiveNodes.Schema : } (\text{pre, size, parent, kind, level, name, value, frag})
\]

\[
\text{SQL}(\Delta_v(i(e)) \Rightarrow
\begin{align*}
&\text{CREATE VIEW } \text{opi} \text{ AS} \\
&\text{SELECT } q.*, \text{OpLiveNodes.value} \text{ AS v} \\
&\text{FROM } q \\
&\text{WHERE } q.i = \text{OpLiveNodes.pre;}
\end{align*}
\] (TRANS-17)

\[
\text{SQL}(e) : q \quad \text{SQL}(\Gamma) : \text{OpLiveNodes}
\]

\[
\text{OpLiveNodes.Schema : } (\text{pre, size, parent, kind, level, name, value, frag})
\]

\[
\text{SQL}(d_{\text{item},(a,n)}(e)) \Rightarrow
\begin{align*}
&\text{CREATE VIEW } \text{opi} \text{ AS} \\
&\text{SELECT DISTINCT } q.\text{iter}, d2.\text{pre} \text{ AS item} \\
&\text{FROM } q, \text{OpLiveNodes AS d1, OpLiveNodes AS d2} \\
&\text{WHERE } q.\text{item} = d1.\text{pre} \\
&\text{AND XPathEvaluationConditions(d2,}\alpha,\text{n}) \\
&\text{ORDER BY d2.}\text{pre;}
\end{align*}
\] (TRANS-18)

Fig. 10. Inference rules of SQL translation (2).
d2.pre AS pre, d2.size, d2.level - d1.level + 1 AS level,
d2.kind, d2.name, d2.value AS value
FROM OpContext, OpLiveNodes AS d1, OpLiveNodes AS d2
WHERE d1.pre = OpContext.item
AND d2.pre >= d1.pre
AND d2.pre <= d1.pre + d1.size) AS ElementConstruction,
(SELECT MAX(pre) AS pre,Max(frag) AS frag FROM OpLiveNodes ) AS MaxPre;

(i) The lines 2 and 3 generate the new pre-order ranks for the newly con-
structed nodes.
(ii) The line 7 generates the new identifier for the newly created transient
XML document fragments.
(iii) The lines from 8 to 17 create the new root nodes for the constructed
sub-tree (one root node per iteration).
(iv) The lines from 21 to 27 copies the required sub-trees from the input live
node fragment.
(v) The lines 28 and 29 specify the maximum pre-order rank and maximum
fragment identifier of the input live node fragment. The maximum pre-
order rank + 1) is used as the starting pre-order rank in step (i) while
the (maximum fragment identifier + 1) is used as the starting fragment
number in step (ii).

– Step 2 : Creating the output context nodes.

CREATE VIEW OpOutputContext AS
SELECT ElementConstruction.Iter,
MaxPre.pre + ROW_NUMBER() OVER (ORDER BY iter,pos,
ElementConstruction.pre) AS Item
FROM (SELECT OPElementNames.iter, 0 AS pos, -2 AS pre
FROM OpContext INNER JOIN OpLiveNodes
ON OpContext.item = OpLiveNodes.pre
RIGHT OUTER JOIN OPElementNames
ON OPElementNames.iter = OpContext.iter
GROUP BY OpElementNames.iter
UNION ALL
SELECT OpContext.iter, OpContext.pos, d2.pre AS pre
FROM OpContext, OpLiveNodes AS d1, OpLiveNodes AS d2
WHERE d1.pre = OpContext.item
AND d2.pre >= d1.pre
AND d2.pre <= d1.pre + d1.size) AS ElementConstruction,
(SELECT MAX(pre) AS pre FROM OpLiveNodes ) AS MaxPre
WHERE ElementConstruction.Level = 0;

The SQL script for creating the output context nodes is similar to the SQL script
for creating the output live nodes fragment with minimal changes. It deals only
with constructing the pre-order ranks of the newly constructed root nodes as a subset from the information maintained by the step of creating the output live nodes fragments.

(i) The lines 2 and 3 generate the new pre-order ranks for the newly constructed nodes.
(ii) The lines from 4 to 9 create the new root nodes for the constructed sub-tree (one root node per iteration).
(iii) The lines from 13 to 17 copy the required sub-trees from the input live node fragment.
(iv) Line 18 specifies the maximum pre-order rank of the input live node fragment. The (maximum pre-order rank + 1) is used as the starting pre-order rank for generation step (i).
(v) Line 19 filters the tuples to be stored in the output context relation. The stored tuples are only the tuples representing the newly constructed root nodes.

Figure 5 illustrated an example describing the behavior of the element construction operator.

**Text Construction Translation** As we discussed in Section 3.1, the behavior of the text construction operator is very similar to the behavior of the element construction operator with minimal changes. The input parameters of the text construction operator is the same as the input parameters of the element construction operator with only one difference. In the element construction operator, the item column of the input context relation OpContext stores the node identifiers of the newly constructed nodes while in the case of the text construction operator, this item column stores the strings of the newly constructed text nodes. The SQL translation template of the text construction operator is defined as follows:

- Step 1: Creating the output live nodes fragments.
  
  ```sql
  CREATE VIEW OpOutputLiveNodes AS
  SELECT MaxPre.pre +
    ROW_NUMBER() OVER (ORDER BY iter,pos,pre) AS pre,
    TextConstruction.size, TextConstruction.level,
    TextConstruction.kind, TextConstruction.name,
    TextConstruction.value,
    MaxPre.frag + DENSE_RANK() OVER (ORDER BY iter) AS frag
  FROM (SELECT OPElementNames.iter, 0 AS pos, 0 AS pre,
    1 AS size, 0 AS level, 'Elem' AS kind,
    OpElementNames.item AS name, null AS value
  FROM OpContext
  RIGHT OUTER JOIN OpElementNames
  ON OpElementNames.iter = OpContext.iter)
  UNION ALL
  ```
SELECT OpContext.iter, OpContext.pos,
  1 AS pre, 0 AS size , 1 AS level, 'Text' as kind,
  OpElementNames.item as name, OpContext.item AS value
FROM OpContext, OpElementNames
WHERE OpContext.iter = OpElementNames.iter) AS TextConstruction,
(SELECT MAX(pre) AS pre,Max(frag) AS frag FROM OpLiveNodes ) AS MaxPre;

– Step 2: Creating the output context nodes.

CREATE VIEW OpOutputContext AS
SELECT TextConstruction.Iter,
  MaxPre.pre + ROW_NUMBER() OVER (ORDER BY iter,pos,pre) AS Item
FROM (SELECT OPElementNames.iter, 0 AS pos, 0 AS pre
  FROM OpContext
  RIGHT OUTER JOIN OpElementNames
  ON OpElementNames.iter = OpContext.iter
  UNION ALL
  SELECT OpContext.iter, OpContext.pos, 1 AS pre
  FROM OpContext) AS TextConstruction,
  (SELECT MAX(pre) AS pre FROM OpLiveNodes ) AS MaxPre
WHERE TextConstruction.Level = 0;

As we discussed in Section 3.1, the element construction operator require sub-
trees copying for the input context nodes from the input live node fragments
relation OpLiveNodes while the text construction operator does not involve
any sub-tree copying for the newly constructed text nodes. Figure 6 illustrated
an example describing the behavior of the text construction operator.

4.3 Translation Approaches

The SQL code generator uses three different approaches for generating the eval-
uation SQL scripts (Grust, Mayr, Rittinger, Sakr, & Teubner, 2007):

1. View-Based Approach: In this approach, each operator of the associated
relational algebraic plan is compiled into an SQL:1999 view definition using
the SQL translation templates defined in the previous Subsection 4.2. The
view definition that represents the SQL translation of an algebraic operator
is represented in the following form:

Create View OP(i) AS ...;

where i represents the operator number in the associated algebraic plan.
The generated SQL scripts using any of the three different approaches always
end up with an additional single query to retrieve the results of the evaluation
script. This additional evaluation step is represented as follows:
SELECT * FROM OP1;

where \textit{OP1} is a relational object (\textit{View - Table}) which stores the result of the final (\textit{root}) operator of the algebraic plan.

The structure of the general form of the generated \textit{view-based} translation scripts can be illustrated as follows:

\begin{verbatim}
CREATE VIEW OP(n) AS ...;
CREATE VIEW OP(n-1) AS ...;
... \ldots
CREATE VIEW OP1 AS ...;
SELECT * FROM OP1;
\end{verbatim}

where \textit{n} represents the number of algebraic operators in the associated algebraic plan.

Although the performance comparisons between the three different approaches (Section 5.1) shows that the \textit{view-based} translation scripts usually performs the best for most of the XMark queries, this approach suffers from two disadvantages. These disadvantages are:

(a) The resulting query plans for the final query statement:

\begin{verbatim}
SELECT * FROM OP1;
\end{verbatim}

for some scripts are more complex and nested such that the query optimizer of some conventional RDBMS are unable to process them. In this case, the SQL code generator must break the nesting of the view definitions by materializing the results of an intermediate relation using a temporary table object. For example, by running the \textit{view-based} SQL translation script for the XMark query \textit{Q8} using Microsoft SQL Server 2005, the system gives the message that "\textit{it is not possible to execute the required statement because the level of nesting is too deep}".

(b) The XQuery language provides the facility of constructing \textit{nested} (\textit{hierarchical}) element structures as a result of an XQuery expression. An example of using this facility is the following expression:

\begin{verbatim}
for $x$ in //person, $y$ in //department return
  <ParentItem>
    <ChildItem1>{$x/@id}</ChildItem1>
    <ChildItem2>{$y/@id}</ChildItem2>
  </ParentItem>
\end{verbatim}

The process of evaluating the generated SQL scripts for such expressions using the \textit{view-based} approach is relatively inefficient. The main reason behind this inefficiency is that the resulting transient nodes created during the runtime evaluation of the first element construction step
has no index information. This constraint of the absence of the index information makes the evaluation of the second element construction step very expensive and inefficient especially if the size of the intermediate fragments are big.

2. **Table-Based Approach:** In this approach, each operator of the associated relational algebraic plan is compiled into a temporary relational table object using the same predefined SQL translation templates with minimal modifications. In general, the *CREATE VIEW* statement of each translation template is replaced with a *CREATE TABLE* statement. The content of each temporary table is specified using the corresponding *view definition* of each operator. The general form of the *Table-Based* version of the SQL translation template can be illustrated as follows:

```
CREATE TABLE OP(i)....;
INSERT INTO TABLE OP(i)
...
[View Definition Of The View-Based Translation]
...;
```

where *i* represents the operator number in the associated algebraic plan.

For the sake of addressing the performance issues, the *table-based* translation needs to instantiate some *CREATE INDEX* statements for the intermediate created temporary table. These *CREATE INDEX* statements are added to the *table-based* version of the translation templates for some of the algebraic operators such as the join operator (*⊗*), document access operator (*Δ*) and element construction operator (*ε*). For example, in the table-based translation template of the join operator we firstly index the join attributes before applying the join operation itself to accelerate its evaluation time (Rule 1). Examples of these *Table-Based* modified templates are illustrated in Figure 11. As illustrated in the Rule 1, the SQL code generator’s decision to utilize the *UNIQUE* feature of each created index influenced by the inferred *key* properties of the algebraic operator (Grust, 2005).

Consequently, The structure of the general form of the generated *table-based* translation scripts can be illustrated as follows:

```
CREATE TABLE OP(n)....;
INSERT INTO TABLE OP(n)....;
CREATE TABLE OP(n-1)....;
INSERT INTO TABLE OP(n-1)....;
...
CREATE INDEX IOP(n-i) on TABLE OP(n-i)([Indexing Attributes]);
...
CREATE TABLE OP1....;
INSERT INTO TABLE OP1....;
SELECT * FROM OP1;
```
where \( n \) represents the number of algebraic operators in the associated algebraic plan.

Using the table-based translation approach has two different sides to it. On the one hand, it solves the two mentioned problems of the view-based approach and executes the evaluation scripts of the queries with nested element construction in a more efficient way. And on the other hand, the performance comparisons between the three different approaches (Section 5.1) shows that the execution time of the table-based translation scripts is mostly longer than the execution time of the view-based translation scripts for the set of XMark queries when no nested element construction expressions are used.

\[
\frac{SQL(e_1) : q_1}{SQL(e_2) : q_2} \quad \frac{a \in e_1.\text{key}}{b \notin e_2.\text{key}}
\]

(1)

\[
SQL(\varrho_{_{\alpha_1,\ldots,\alpha_n/p}}(e)) \Rightarrow
\begin{align*}
\text{CREATE INDEX } & q_{\text{Sort ON}} q (o_1,\ldots,o_n); \\
\text{CREATE INDEX } & q_{\text{Partition ON}} q (p); \\
\text{CREATE TABLE } & \text{opi}(a \text{ int,}[\text{OpInput Attributes}]); \\
\text{INSERT INTO } & \text{opi} \\
\end{align*}
\]

(2)

\[
SQL(e) : q
\]

(3)

Fig. 11. Examples of the inference rules of table-based SQL translation.

3. Hybrid-Based Approach: The hybrid approach attempts to combine the advantages of the two other approaches in an elegant way. The hybrid-based
translation follows in standard the view-based translation for translating the algebraic operator of the associated plan (therefore, it has the high performance advantages associated with this approach) and follows the table-based translation in situations where the view-based approach suffers from the already mentioned performance problems (either in the case where the view definitions will lead to a too nested or complex query plan or in the case where the occurrences of nested element constructions steps are detected in the translated algebraic plans).

The structure of the general form of the generated hybrid-based translation scripts can be illustrated as follows:

```
CREATE VIEW OP(n) AS ...;
CREATE VIEW OP(n-1) AS ...;
...
CREATE TABLE OP(n-i)....;
INSERT INTO Table OP(n-i)....;
CREATE INDEX IOP(n-i) on Table OP(n-i)([Indexing Attributes]);
...
CREATE VIEW OP1 AS ...;
SELECT * FROM OP1;
```

where \( n \) represents the number of algebraic operators in the associated algebraic plan.

Clearly, there is a trade-off between the number of materialization points and the query processing time. In our context, we are using the available cardinality information during the SQL translation process (Sakr, 2007) to select the materialization points in a more intelligent and efficient way. We are applying a conditional intermediate results materialization mechanism where the estimated size of the intermediate results is passed to the SQL code generator which uses these estimates to evaluate the relative benefits from materializing these intermediate results and picks some of them for materialization with the goal of minimizing the total query costs. The main rule of thumb of this mechanism is: if the estimated size of the intermediate result is expected to be large, then we decide to materialize them in the following two situations:

- Before applying any binary or ternary algebraic operators (join operator \( \times \), difference operator \( \setminus \), document access operator \( \Delta \), XPath evaluator operator \( \exists \), element / text construction operators \( \varepsilon/\tau \) ).
- In the case when the estimated large intermediate results is used by more than one operator, a decision to materialize is to be made.

Actually, the size of large intermediate results is a vague concept. As such an intermediate result is defined as being large by a system parameter which can be specified by the system administrator according to the available main memory size. To illustrate our mechanism let us consider the two XMark queries (Schmidt et al., 2002) Q8 and Q20. Although both of them use nested element
construction, the hybrid translation approach for Q20 does not decide to ma-
erialize any of the input intermediate results at any element construction step
because in each of them only one new element need to be constructed independent
of the size of the source XML document. However, the hybrid translation
approach for Q8 - in case of processing large documents - decides to materialize
the input intermediate results before applying the element construction steps
because the size of the input relations is estimated to be large. The experiments
of Section 5.1 support the idea of our mechanism. In these experiments, mate-
rializing the intermediate results for the XMark query Q8 before applying the
nested element construction steps was efficient because the size of these inter-
mediate results was relatively large while for Q20 materializing the intermediate
results was inefficient because for this query the size of the intermediate results
for the element construction operations are consisting of only one tuple. In this
case, materializing this small intermediate results yields to extra non-required
overhead costs.

4.4 Translation Patterns

As discussed, the direct translation of relational algebraic plans into its equiv-
alent SQL scripts is achieved by traversing the associated DAG in a bottom up
fashion and then translating each algebraic operator into its equivalent SQL eval-
uation step using the defined SQL translation templates. Therefore, the direct
translation approach leads to a number of SQL evaluation steps that is equal
to the number of the algebraic operators in the associated algebraic plan. Con-
sequently, the SQL scripts resulting from this direct translation approach tends
to be cumbersome, lengthy and suffers from performance degradation. The per-
formance degradation is a result of the consequence execution of huge number
of detailed evaluation steps especially in the case of the table-based translation
approach where extra non-required intermediate tables are created and extra
intermediate tuples are inserted.

To avoid these limitations, the SQL code generator uses a Pattern-Based Trans-
lation Approach for merging and rewriting the translation of a group of operators
(Pattern) into a single SQL evaluation step. The Pattern-Based Translation Ap-
proach used by the SQL code generator is very similar to the tree matching
approach for code generation described in (Aho, Ganapathi, & Tjiang, 1989).
The use of this Pattern-Based Translation Approach yields a more efficient,
amenable and compact SQL script. The number of intermediate evaluation steps
for the generated scripts using the Pattern-Based Translation Approach is ap-
proximately equal to \(\frac{1}{3}\) of the number of the evaluation steps for scripts generated
using the direct translation approach. The experiments detailed in Section 5.3
will present a comparison between the two approaches in terms of the number of
evaluation steps and the execution time of the resulting SQL scripts. Practically,
the SQL code generator stores a library of algebraic patterns that varies from
simple patterns consisting of two operators (for example: Project(\(\pi\))/Select(\(\sigma\)) -
Project(\(\pi\))/Union(\(\cup\)) - Attach(\(\oplus\))/Project(\(\pi\)),...etc) to complex patterns merg-
ing 6 or 7 algebraic operators. During the translation process, the code generator
PatternBasedSQLTranslation (AlgebraicPlan G)
{
  BEGIN
    TranslationScript t;
    Pattern p;
    InstanceArray instances;
    int counter;
    Node v;
    Node n;

    G.setTranslatePropertyForAllNodesEqualToTrue();
    G.setPatternPropertyForAllNodesEqualToFalse();
    p=getNextTranslationPattern()
    DO WHILE (p != null)
      instances=detectPatternInstance(G,p);
      FOR counter = 1 to instances.count
        FOR EACH v IN instances(i).subTree
          v.translate = False;
        NEXT
        instances(i).root.pattern= True;
        instances(i).root.patternKind = p.name;
      NEXT
    LOOP

    n = G.getNextPostorderNode();
    DO WHILE (n != null)
      IF (n.translate == True) THEN
        if (n.pattern == True) THEN
          t.add(translatePattern(n));
        ELSE
          t.add(translateOperator(n));
        END IF
      END IF
    n = G.getNextPostorderNode();
  LOOP
  RETURN t;
}

Fig. 12. Pattern-based XQuery-to-SQL translation algorithm
detects the occurrences of its defined patterns and uses them for generating more elegant and efficient SQL scripts. Figure 12 illustrates the pattern-based algorithm for translating the relational algebraic plans into equivalent SQL scripts. Remarks about the translation algorithm are given as follows:

– The lines from 11 to 23 are representing a pre-processing step for detecting the occurrences of the stored translation patterns in the processed algebraic plan.
– The function call $G.setTranslatePropertyForAllNodesEqualToTrue()$ in line number 11 initiates the translate property of all operators in the processed algebraic plan to the Boolean value $True$. The value of the translate property for each operator will be used in line number 27 to decide if the SQL translation of this operator will be added to the resulting translation script or if it will be combined with the translation of other operators in a pattern translation.
– The function call $G.setPatternPropertyForAllNodesEqualToFalse()$ in line number 12 initiates the pattern property of all operators in the processed algebraic plan to the Boolean value $False$. The value of the pattern property of each operator will be used in line number 28 to decide the translation template of this operator if it is pattern-based or direct based (in case of the translate property of this operator is equal to true).
– The function call $getNextTranslationPattern()$ in line number 13 is used for retrieving the stored patterns from the SQL code generator library of patterns one by one and in a deterministic order according to their size and generality to avoid the problem of overlapping patterns. For example the pattern ($@ π \backslash b$) will be retrieved before the pattern ($π \backslash b$).
– The function call $detectPatternInstance(G, p)$ detects the occurrences of the stored patterns in the processed algebraic plan and returns a list of the detected instances. The function $detectPatternInstance$ operates only over the nodes where the translate property is equal to $True$ and the pattern property is equal to $False$. If the translate property is equal to $False$ or the pattern property is equal to $True$, then it means that this operator is detected inside the occurrence of previously retrieved pattern which avoids the problem of overlapping patterns. Each operator in the detected instance of the pattern in the processed plan must have only a single parent operator otherwise the existence of the pattern is ignored from the returning instances.
– For each operator in the sub-tree of detected pattern, the $False$ value is assigned to the translate property. For the root operator of the pattern, the $True$ value is assigned to the pattern property and the pattern name is stored in the patternKind property.
– The lines from 25 to 35 are performing the SQL translation process by traversing the operators of the processed algebraic plan in a postorder fashion. In the translation process, the operators with the translate property equal to $False$ are ignored and for the operators with the translate property equal to $True$, the adequate translation templates are chosen based on the information of the pattern and the patternKind properties.
To illustrate, we will represent an example for the *Translation Patterns* idea used by the SQL code generator and in Section 4.5, we will present an example of translating the relational algebraic plans into equivalent SQL evaluation scripts.

**Combined XPath Steps** Figure 13 illustrates an example of the *Combined XPath Steps* translation pattern. The annotation numbers within the symbol ⬇️ represents the operator identifier numbers in the algebraic plan. In this pattern, a sequence of XPath Evaluator algebraic operators (Ctx) appears in the algebraic plan in a consecutive manner where the output of one operator is used as an input for the following operator. Applying the *direct* translation approach over the example of combined XPath steps pattern illustrated in figure 13 will evaluate each operator in a separate step as follows:

```sql
1. CREATE UNIQUE INDEX CtxItem ON Ctx(item);
2. CREATE TABLE OP9 (iter int,item int);
3. INSERT INTO OP9
   4. SELECT DISTINCT ctx.iter,d2.Pre
      5. FROM ctx, document AS d1,document AS d2
      6. WHERE ctx.item=d1.pre
         7. AND d2.pre > d1.pre AND d2.pre <= d1.pre + d1.size
```

![Diagram](image)  
**Fig. 13.** An example of a translation pattern of combined XPath steps.
AND d2.level=d1.level +1 AND d2.kind = 'Elem'
AND d2.name='site'
ORDER BY d2.pre;

CREATE UNIQUE INDEX OP9Item ON OP9(item);
CREATE TABLE OP8 (iter int,item int);
INSERT INTO OP8
  SELECT DISTINCT ctx.iter,d2.Pre
  FROM OP9, document AS d1,document AS d2
  WHERE ctx.item=d1.pre
  AND d2.pre > d1.pre AND d2.pre <= d1.pre + d1.size
  AND d2.level=d1.level +1 AND d2.kind = 'Elem'
  AND d2.name='people'
  ORDER BY d2.pre;

CREATE UNIQUE INDEX OP8Item on OP8(item);
CREATE TABLE OP7 (iter int,item int);
INSERT INTO OP7
  SELECT DISTINCT ctx.iter,d2.Pre
  FROM OP8, document AS d1,document AS d2
  WHERE ctx.item=d1.pre
  AND d2.pre > d1.pre AND d2.pre <= d1.pre + d1.size
  AND d2.level=d1.level +1 AND d2.kind = 'Elem'
  AND d2.name='person'
  ORDER BY d2.pre;

CREATE UNIQUE INDEX OP7Item on OP7(item);
CREATE TABLE OP6 (iter int,item int);
INSERT INTO OP6
  SELECT DISTINCT ctx.iter,d2.Pre
  FROM OP7, document AS d1,document AS d2
  WHERE ctx.item=d1.pre
  AND d2.pre > d1.pre AND d2.pre <= d1.pre + d1.size
  AND d2.level=d1.level +1 AND d2.kind = 'Elem'
  AND d2.name='person'
  ORDER BY d2.pre;

The lines from 1 to 10 represents the evaluation of the XPath Evaluator operator \((d\!l)\) in the algebraic plan with the path step \((child::site)\). The lines from 12 to 21 represents the evaluation of the XPath Evaluator operator \((d\!l)\) in the algebraic plan with the path step \((child::people)\). The lines from 23 to 32 represents the evaluation of the XPath Evaluator operator \((d\!l)\) in the algebraic plan with the path step \((child::person)\). The lines from 34 to 43 represents the evaluation of the XPath Evaluator operator \((d\!l)\) in the algebraic plan with the path step \((child::name)\).
The pattern translation will combine the evaluation of the four XPath Evaluator algebraic operators in a single step as follows:

```
CREATE UNIQUE INDEX CtxItem on Ctx(item);
CREATE TABLE OP6 (iter int, item int);
INSERT INTO OP6
    SELECT DISTINCT ctx.iter,d5.Pre
    FROM ctx, document AS d1,document AS d2,document AS d3,
    document AS d4,document AS d5
    WHERE ctx.item=d1.pre
    AND d2.pre > d1.pre AND d2.pre <= d1.pre + d1.size
    AND d2.level=d1.level +1 AND d2.kind = 'Elem'
    AND d2.name='site'
    AND d3.pre > d2.pre AND d3.pre <= d2.pre + d2.size
    AND d3.level=d2.level +1 AND d3.kind = 'Elem'
    AND d3.name='people'
    AND d4.pre > d3.pre AND d4.pre <= d3.pre + d3.size
    AND d4.level=d3.level +1 AND d4.kind = 'Elem'
    AND d4.name='person'
    AND d5.pre > d4.pre AND d5.pre <= d4.pre + d4.size
    AND d5.level=d4.level +1 AND d5.kind = 'Elem'
    AND d5.name='name'
ORDER BY d5.pre;
```

The lines from 8 to 10 represents the evaluation conditions of the path step (child::site) 9. The lines from 11 to 13 represents the evaluation conditions of the path step (child::people) 9. The lines from 14 to 16 represents the evaluation conditions of the path step (child::person) 9. The lines from 17 to 19 represents the evaluation conditions of the path step (child::name) 9.

Comparing the resulting two scripts of the direct translation and pattern translation reveals the improved efficiency provided by the pattern translation in terms of the reduction in the number of evaluation steps. The pattern translation combines the evaluation steps of the operators 9, 8, 7, 6 in a single step while the direct translation uses four separate evaluation steps. From a performance perspective, the pattern translation approach generates SQL scripts that are clearly more efficient than those generated by direct translation especially in the context of the table based translation approach because it avoids the creation of additional structures and tuples insertion in a larger number of the intermediate results.

4.5 Translation Examples

In this section, we represent an example for translating the Pathfinder intermediate algebraic plans into SQL scripts.
Fig. 14. The Pathfinder algebraic plan for XQuery Q1.
This example is based on the "auction.xml" XML document from the XMark Benchmark project (Schmidt et al., 2002). In the SQL script, we refer to the `document` table as an equivalent for the XPath accelerator relational encoding for the XML document "auction.xml". Figure 14 illustrates the Pathfinder relational algebraic plan for the XQuery query Q1. The SQL translation of the illustrated algebraic plan using the View Based approach is defined as follows:

```sql
-- Step 1 (Path Expression)
CREATE VIEW OP16 AS
  SELECT d3.pre AS item1
  FROM document AS d1, document AS d2, document AS d3
  WHERE d1.kind = 'Doc'
  AND d2.pre > d1.pre AND d2.pre <= d1.pre + d1.size
  AND d2.name = 'price'
  AND d3.pre > d2.pre AND d3.pre <= d2.pre + d2.size
  AND d3.level = d2.level + 1 AND d3.kind = 'Text';

-- Step 2 (Row Numbering)
CREATE VIEW OP14 AS
  SELECT ROW_NUMBER() OVER () AS iter,
        ROW_NUMBER() OVER (ORDER BY item1) AS pos, item1
  FROM OP16;

-- Step 3 (Document Access + Attachment)
CREATE VIEW OP12 AS
  SELECT OP14.iter, OP14.pos, OP14.item1,
        d.value AS item2, 40 AS item3
  FROM OP14, document AS d
  WHERE OP14.item1 = d.pre;

-- Step 4 (Comparison Operation)
CREATE VIEW OP10 AS
  SELECT iter, item1
  FROM OP12
  WHERE item2 < item3;

-- Step 5 (Attachment + Projection)
CREATE VIEW OP8 AS
  SELECT iter, 'item' AS item
  FROM OP10;
```
-- Step 6 (Attachment + Projection)
CREATE VIEW OP6 AS
SELECT iter, item AS item, 1 AS pos
FROM OP10;

-- Step 7 (Element Construction)
CREATE VIEW OP5 AS
SELECT MaxValues.Mpre +
    ROW_NUMBER() OVER (ORDER BY iter,pos,pre) AS pre,
    ElementConstruction.size, ElementConstruction.level,
    ElementConstruction.kind, ElementConstruction.name,
    ElementConstruction.value,
    MaxValues.frag + DENSE_RANK() OVER (ORDER BY iter) AS frag
FROM (SELECT OP8.iter, 0 AS pos, -2 AS pre,
    COALESCE(SUM(size + 1), 0) AS size,
    0 AS level, 'Elem' AS kind,
    OP8.item AS name,
    null AS value
    FROM OP6 INNER JOIN document
    ON OP6.item = document.pre
    RIGHT OUTER JOIN OP8
    ON OP8.iter = OP6.iter
    GROUP BY OP8.iter,OP8.item
UNION ALL
SELECT OP6.iter, OP6.pos,
    d2.pre AS pre, d2.size, d2.level - d1.level + 1 AS level,
    d2.kind, d2.name, d2.value AS value
FROM OP6, document AS d1, document AS d2
WHERE d1.pre = OP6.item
AND d2.pre >= d1.pre
AND d2.pre <= d1.pre + d1.size) AS ElementConstruction,
(SELECT MAX(pre) AS Mpre,MAX(frag) AS frag
FROM Document ) AS MaxValues;

CREATE VIEW OP4 AS
SELECT ElementConstruction.Iter,
    MaxValues.Mpre + ROW_NUMBER()
    OVER (ORDER BY iter,pos,pre) AS Item
FROM (SELECT OP8.iter, 0 AS pos, -2 AS pre, 0 as level
    FROM OP6 INNER JOIN document
    ON OP6.item = document.pre
    RIGHT OUTER JOIN OP8
    ON OP8.iter = OP6.iter
    GROUP BY OP8.iter)
UNION ALL
SELECT OP6.iter, OP6.pos, d2.pre AS pre, 
    d2.level - d1.level + 1 AS level
FROM OP6, document AS d1, document AS d2
WHERE d1.pre = OP6.item
    AND d2.pre >= d1.pre
    AND d2.pre <= d1.pre + d1.size) AS ElementConstruction,
(SELECT MAX(pre) AS Mpre FROM document ) AS MaxValues
WHERE ElementConstruction.Level = 0;

-- Step 8 (Project + Join)
CREATE VIEW OP1 AS
    SELECT OP4.item, OP14.pos
FROM OP4, OP14
WHERE OP4.iter = OP14.iter;

SELECT * from Op1;

The SQL translation script of the algebraic plans seems to be very lengthy. However, using the relational database infrastructure such as the query optimization techniques and the indexing mechanisms yields to very efficient execution times. The experiments of Section 5 will give the evidences which are supporting our claim.

4.6 XPath Optimization in A Relational XQuery Engine

Partitioned B-tree Indexes Path expressions are the basic building block for the XQuery language. They are used inside the XQuery expression for retrieving the target data from the XML tree. Optimizing the evaluation of path expression is therefore a crucial step in order to optimize the evaluation of XQuery expressions. In Section 2.2, we described the SQL evaluation of XPath expression using the XPath accelerator encoding scheme. As discussed, the evaluation of XPath expression with a series of location steps $S_1/S_2/.../S_n$ is translated into a series of compositional $n$ region queries where the node sequence output by axis step $S_i$ is the context node sequence for the subsequent step $S_{i+1}$. Consequently, the evaluation of XPath expressions requires multiple self-joins ($n$) for the relational encoding of the target XML document and the number of these self-join operations is equal to the number of the location steps in the path expression. Generally, the relational encoding schemes for storing the XML documents lack of the understanding of the tree nature of the underlying XML data. To avoid this limitation, several join algorithms have been proposed for efficient processing of the XPath axes over the relational storage schemes (Al-Khalifa et al., 2002), (Bruno, Koudas, & Srivastava, 2002), (Chien, Vagena, Zhang, Tsotras, & Zaniolo, 2002), (Jiang, Wang, Lu, & Yu, 2003). In our context, Grust have proposed in (Grust et al., 2003) the staircase join algorithm based on XPath accelerator storage scheme. This algorithm was designed to speed up the SQL-based evaluation of XPath expressions by incorporating the specific knowledge about
the XML tree and the (pre/size/level) encoding. Although the work of (Mayer, Grust, Keulen, & Teubner, 2004) has shown the possibility of incorporating the staircase join into the open source conventional RDBMS PostgreSQL. We are, however, not able to exploit such optimized join algorithm in our approach because it requires modifications to the internals of the underlying RDBMS kernel which is not easily applicable in commercial RDBMS systems (e.g., IBM DB2, Microsoft SQL Server, Oracle, etc).

Relational database indexes has proven to be a very efficient technique for enhancing the performance of evaluating the SQL expressions over the stored relational tables. The R-trees indexing data structure proved its efficiency for indexing multi-dimensional information and in (Grust, 2002), Grust has shown the effectiveness of using R-tree indexing for supporting the processing of the evaluation condition (range conditions) of the XPath axes using the XPath accelerator encoding scheme. However, we are not able to depend on this indexing technique in our approach because the R-tree indexing technique is not commonly supported by many of the RDBMS systems where B-tree indexing is the still the most common used technique.

Since we represent our work as a purely relational implementation of XQuery, we decided to use the B-tree indexing as this is commonly available database technique for accelerating the processing of the XPath evaluation conditions. Specifically, we use a slight variant of the B-tree indexing structure called partitioned B-tree. The idea of the partitioned B-trees is represented by Graefe in (Graefe, 2003) where he recommended using of low-selectivity leading columns to maintain the partitions within the associated B-tree. In (Grust, Rittinger, & Teubner, 2007), Grust has demonstrated the possibility of achieving comparable XPath performance by exploiting existing database functionality and purely relational means. One of these demonstrated techniques is partitioned B-trees indexes. Examples for our used partitioned B-trees indexes are given as follows:

- Supporting the processing of the evaluation condition of the descendant, descendant-or-self, ancestor and ancestor-or-self XPath axes can be achieved in terms of (kind, pre) and (kind, name, pre) indexes.
- Supporting the processing of the evaluation condition of the child and parent XPath axes can be achieved in terms of (kind, level, pre) and (kind, name, level, pre) indexes.
- Supporting the processing of the evaluation condition of the attribute, following-sibling and preceding-sibling XPath axes can be achieved in terms of (kind, parent, pre) and (kind, name, parent, pre) indexes.

The experiments detailed in Section 5.4 aims to prove the effectiveness of our proposed indexes for the evaluation of the XPath and consequently XQuery expressions.

**Guide Node SQL Scripts** Materialized views is another well-known physical structures that can significantly accelerate the performance of the evaluation of SQL queries. There has been a lot of work done in the general problem of rewriting SQL queries using materialized views (Agrawal, Chaudhuri, & Narasayya,
and on the specific area of rewriting XML queries using materialized XPath views (Xu & Meral, 2005), (Balmin, zcan, Beyer, Cochrane, & Pirahesh, 2004), (Barta, Consens, & Mendelzon, 2004).

In our context, during the shredding process of the source XML document we build a summarized tree structure, Statistical Guide, of the source XML document (Sakr, 2007). The Statistical Guide represents an implementations for the Data Guide summary tree structure presented in (Goldman & Widom, 1997) and is very similar to the path tree summary structure presented by Aboulnaga in (Aboulnaga, Alameldeen, & Naughton, 2001). Every node in the Statistical Guide is a Guide Node which is representing a correspondent group of nodes in the source XML document which are sharing the same rooted path starting from the root node. Each Guide Node with its associated pre-order rank represents a form of Path_ID for their correspondent nodes in the source XML documents (Sakr, 2007). To make use of this point, we extended our encoding relation (document table) of the source XML document with an additional attribute to store the Path_ID information for each correspondent node. Additionally, we build a partitioned B-tree index over the (path_id,pre) attributes to form the basis of a materialized view which establishes the link between each original XML node and its correspondent Guide Node.

During the compilations process, we are able to infer the Guide Node property of the XPath evaluator operators in the algebraic plan (Sakr, 2007). Our mechanism is based on the observation that we can rewrite the SQL translation of the rooted XPath expressions in the algebraic plan using the Guide Node information of the last path step in the rooted sequence. To illustrate let us consider the following example:

```xml
S1
for $x in doc("auction.xml")/site/open_auctions/open_auction
return $x
```

Figure 15 illustrates the algebraic plan of the XQuery expressions S1. The conventional SQL translation of the combined XPath steps translation pattern of the operators (7),(6),(5),(4) is:

```sql
SELECT d4.pre as item
FROM document AS d1,document AS d2,document AS d3,document AS d4
WHERE d1.kind = 'Doc' and d1.name='auction.xml'
AND d2.pre >= d1.pre AND d2.pre <=d1.pre + d1.size
AND d2.level=d1.level+1 AND d2.name='site' AND d2.kind='Elem'
AND d3.pre >= d2.pre AND d3.pre <=d2.pre + d2.size
AND d3.level=d2.level+1 AND d3.name='open_auctions' AND d3.kind='Elem'
AND d4. pre >= d3.pre AND d4.pre <=d3.pre + d3.size
AND d4.level=d3.level+1 AND d4.name='open_auction' AND d4.kind='Elem';
```
where line number 3 represents the evaluation conditions of the operator $\text{7}$, lines 5 and 6 represent the evaluation conditions of the operator $\text{6}$, lines 8 and 9 represent the evaluation conditions of the operator $\text{5}$, lines 11 and 12 represent the evaluation conditions of the operator $\text{4}$.

Using the Guide Node information of the Pathfinder algebraic operators, we can discard the operators $\text{7}$, $\text{6}$, $\text{5}$ and use the Guide Node property of the algebraic operator $\text{4}$ to rewrite the SQL translation of the same combined XPath steps translation pattern as follows:

```
SELECT pre AS item
FROM document
WHERE path_Id = 41;
```

This rewriting mechanism using the Path_ID materialized view has shown to be very efficient especially in the case of rewriting long rooted XPath expressions with high evaluation costs for the conventional mechanisms as well as in the case of processing large XML documents.

In addition, rewriting the SQL evaluation of XPath expression using the guide node information could be considered as a form of schema aware optimization. Using the information of the statistical guide and the guide node annotation mechanism we could avoid the evaluation of XPath expressions which yield to an empty sequences of context nodes (Barta, Consens, & Mendelzon, 2005).
Applying the path steps of such path expression over the statistical guide will yield to an algebraic operator with an empty set of guide node annotations. Such instances of the algebraic operators could be translated into a very cheap SELECT statement from an empty table instead of using the relatively expensive conventional SQL translation.

4.7 Selectivity Influences

The DBMS query optimizers are responsible for determining the most efficient evaluation strategy for every given SQL query. For any given SQL query, there are a large number of alternative execution plans. These alternative execution plans may differ significantly in their use of system resources or response time and usually this difference can be in orders of magnitude difference in performance between the best and worst plans which makes selecting the right plan a very important task (Reddy & Haritsa, 2005).

Optimizing the evaluation of SQL queries depends crucially on the ability to obtain effective compile-time estimates for the selectivity of the referenced "WHERE" conditions over the underlying stored relational tuples and attribute values. Sometimes query optimizers are not able to select the most optimal execution plan for the input queries because of the unavailability or the inaccuracy of the required statistical information. SQL is a declarative query language which enables the user to define "WHICH" data need to be accessed but not "HOW" this access should be done. To solve this problem, modern RDBMS such as IBM DB2 and Oracle give the users the ability to give hints to influence the query optimizers by providing additional selectivity information for the individual predicates of the given SQL queries. In this context, selectivity values are ranging between the values 0 and 1 for each individual predicate. A lower selectivity value (close to 0) will inform the query optimizer that the associated predicate will return fewer result rows while a higher selectivity value (close to 1) will inform the query optimizer that the associated predicate will return a larger number of result rows. In the case of the correctness of this additional selectivity information hinted to the query optimizers, it will help the query optimizers to make the right decisions in several situations such as: selecting the right order of the join operations and selecting the most suitable indexes. On the other side, hinting the query optimizer with incorrect selectivity information will lead the query optimizer to incorrect decisions and consequently to inefficient execution plans. Figure 16 illustrates the effect of assigning different selectivity values for a specific predicate ($S1$) on the response time of the following SQL query:

```sql
SELECT d3.pre
FROM document as d1,document as d2,document as d3
WHERE d1.kind = 'Doc' SELECTIVITY $S1$
AND d2.pre >= d1.pre
AND d2.pre <= d1.pre + d1.size
AND d2.level=d1.level + 1
AND d2.name = 'site';
```
In our implementation, we used the available cardinality information from the XQuery estimation module to provide the query optimizers with selectivity values hints of the used predicates during the SQL translation process. Remarks about the computation of these selectivity values are given as follows:

- The SQL translation templates for the algebraic operators described in Section 4.2 implement the predicates using the "SQL WHERE" conditions in the translation templates of the following Pathfinder algebraic operators: the selection operator ($\sigma$), the join operator ($\Join$), the comparison operators ($\equiv$), the document access operator ($\Delta$) and the XPath evaluator operator ($\mathcal{E}$).

- The translation templates of the selection operator ($\sigma$) and the comparison operators ($\equiv$) use a single predicate which are represented using a single "WHERE" conditions in each of them. For these two operators, the selectivity of the associated "WHERE" condition is computed by dividing the estimated cardinality of the resulting relation by the estimated cardinality of the input relation.

- The translation templates of the join operator ($\Join$) and the document access operator ($\Delta$) are also using a single predicate. The only difference here is these operators are binary operators which are receiving two input relations while the selection operator ($\sigma$) and the comparison operators ($\equiv$) are unary operators which are receiving a single relation. Hence, for these two operators, the selectivity of the associated "WHERE" condition is computed by
dividing the estimated cardinality of the resulting relation by the resulting value from multiplying estimated cardinalities of the two input relations.

– The translation template of XPath evaluator operator (ε) is using multiple predicates represented with the multiple conjunctive "WHERE" conditions for the different evaluation conditions of the different path steps which are previously described in Section 2.2. Although, the XQuery estimation module is able to estimate the cardinality of the resulting relation from applying the associated path step over the input context relation, this information is not useful in this context because the query optimizers accept to receive hints for the selectivity information of each individual predicate separately and not for a group of predicates. In this case, the selectivity of each predicate is computed using the information of the Statistical Guide and the associated guide node information.

5 Experiments

In this section, we present a performance study of the Pathfinder as a purely relational XQuery engine. In our experiments we are using the data generated by the XMark benchmark (Schmidt et al., 2002). We generated XML documents using the XMark benchmark for three scaling factors, 0.009 (1 Mb), 0.09 (10 Mb), and 0.9 (100 Mb). The experiments of this article are performed on Linux based server with two 3,2 GHZ Intel Xeon processors, 8 GB main memory storage and 280 GB SCSI secondary storage. We verified the correctness of the execution strategies by comparing the output of the SQL translation scripts of each query over different XQuery engines using a text comparison tool. In principle, our experiments have the following goals:

– To compare the difference in performance characteristics between the different SQL translation approaches (view-based, table-based, hybrid).
– To test the quality of our XQuery-to-SQL translation approaches with regards to leveraging the RDBMS technology by measuring their performance efficiency and scalability.
– To test the effectiveness of the various optimization techniques used by our approach (pattern translation, partitioned B-tree indexing).
– To demonstrate the efficiency of our purely relational approach for implementing XQuery processor in comparison to the native XML/XQuery functionality supported by DB2 version 9.

All reported numbers are the average of five executions with the highest and the lowest values removed. In our five readings for each query, we noticed that the first reading is always expensively inconsistent with the other readings. This is because the relational database uses buffer pools as a caching mechanism, The initial period when the database spends its time loading pages into the buffer pools is known as the warm up period. During this period the response time of the database declines with respect to the normal response time. For efficient testing for the important and most time consuming feature of XQuery,
nested element constructions, we have made slight changes to the XMark queries. We replaced the attribute construction expressions into an element construction expressions of a child element. For example the original version of XMark query Q8 is defined as follows:

```xml
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
    where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

while our modified version is defined as follows:

```xml
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
    where $t/buyer/@person = $p/@id
  return $t
return <item>
  <person>{$p/name/text()}</person>
  <count>{count($a)}</count>
</item>
```

The same modification has been applied for Q3, Q9, Q11, Q12, Q13 and Q19.

### 5.1 View-Based vs. Table-Based vs. Hybrid

In Section 4.3, we described three approaches (view-based, table-based, hybrid) for translating the intermediate Pathfinder algebraic plan into SQL scripts. Figure 17 illustrates a comparison between the execution times of the SQL translation scripts of the 20 XMark queries using the three different approaches. This experiment have used an instance of XMark document which has the size of 100 MB and contains around 3 million nodes. Remarks about the results of this figure are given as follows:

– The view-based approach has demonstrated its effectiveness over the table-based approach for the queries which are not using nested element construction expressions (Q1, Q4, Q5, Q6, Q7, Q14, Q15, Q16, Q17, Q18) while the table-based approach has demonstrated its effectiveness for the queries which are using nested element construction expression (Q3, Q8, Q9, Q10, Q11, Q12, Q13, Q19). As discussed in Section 4.3, the main reason behind this is that by evaluating the upper element construction step, the view-based approach suffers from the limitation of missing the indexing information for the resulting transient nodes created during the runtime evaluation of the child element construction step.
The hybrid approach uses the view-based approach as the standard approach and the table-based approach when nested element construction expressions are used (see Secion 4.3). The results of Figure 17 have shown that the hybrid approach is the most efficient in all of the 20 XMark queries except for Q20. In Q20, although nested element construction expressions are used, materializing the intermediate results of these expressions is not efficient because each intermediate result consists of only one tuple which do not have any side effect on the nested element construction evaluation step. Hence, the hybrid approach in this case suffers from the extra overhead of inefficient materialization steps.

The queries Q8, Q9, Q10, Q11 and Q12 have the longest execution times. For queries Q8, Q9, Q11 and Q12, this is due to XQuery joins that produce substantial intermediate XML results which hurts the evaluation of the nested element construction steps. For Q10 the reason is different as several required nested element construction evaluation steps are executed in addition to the deeper nesting level of the element construction steps. The view-based translation scripts for these queries could not complete the evaluation process within the time frame of three hours.

![Fig. 17. Comparison between view - table - hybrid SQL translation approaches. Execution times for the 20 XMark queries ran against 100 MB XMark document instance hosted by DB2.](image-url)
5.2 Scalability

One of the main advantages of using a relational database to store and process XML documents is to exploit their well-known scalability feature. To assess the scalability of our approach, Figure 18 illustrates the execution times for the SQL translations scripts for the 20 XMark queries over the encoding relations of three XMark documents with sizes of 1 MB, 10 MB and 100 MB. The SQL scripts of these experiments are generated using the hybrid approach. The figure shows that the execution times of our system scales in a near linear fashion with respect to the document size.

Fig. 18. Pathfinder scalability. Execution times for three XMark document instances with sizes 1Mb, 10 MB and 100 MB.

5.3 Direct Translation vs. Pattern Translation

In Section 4.4, we presented our Pattern-Based translation approach which combines and rewrites the translation of a group of algebraic operators (Pattern) of the processed Pathfinder algebraic plan into a single SQL evaluation step. Using the pattern-based translation approach has a significant effect on the generated SQL evaluation scripts in terms of reducing the number of the evaluation steps and consequently accelerating the execution time. Figure 19(a) illustrates the comparison between direct translation and the pattern translation approaches
in terms of the number of the evaluation steps for the XMark queries. On average, the number of the evaluation steps by using the pattern-based translation approach is equal to $\frac{1}{4}$ the number of the evaluation steps of the direct approach. Figure 19(b) illustrates the comparison between the two approaches in terms of their execution times for the SQL translation scripts for the XMark queries against the relational encoding of XMark document with size of 1 MB (25000 nodes) hosted by DB2. On average, the execution time of the SQL scripts using the pattern-based translation approach is equal to 70% of the execution time of the execution time of the SQL scripts using the direct approach. The SQL scripts of this experiment are generated using the table-based approach where the pattern translation approach is more effective in terms of execution time because of its avoidance of extra creation for intermediate results.

5.4 Indexes Effectiveness

The performance of queries evaluation in relational database systems is very sensitive to the defined indexes structures over the data of the source tables. Using relational indexes can accelerate the performance of queries evaluation in several ways (Valentin, Zuliani, Zilio, Lohman, & Skelley, 2000). For example, by applying predicates, it can limit the data that must be accessed to only those rows that satisfy those predicates. In addition, query evaluations can be achieved using index-only access and save the necessity to access the data pages by providing all the columns needed for the query evaluation. In Section 4.6, we presented our mechanism of using the partitioned B-tree indexing mechanism for accelerating the SQL evaluation of XPath expressions and consequently accelerating the evaluation of XQuery expressions. Leveraging our purely relational approach for storing and querying the XML documents, we are able to use ready made tools provided by the RDBMSs to propose the candidate indexes that are effective for accelerating our queries work loads. However, similar approaches and tools are still not available by the native XML support of these systems. In this experiment, we used the db2advis tool provided by the DB2 engine to recommend the suitable index structure for our query workload. Using this tool significantly improves the quality of our designed indexes and speeds up the evaluation of our queries by reducing the number of the calls to the database engine. Figure 20 illustrates the substantial effect of using the Partitioned B-tree indexing mechanism and the index structures proposed by db2advis on the execution time of the SQL translation scripts of XMark queries. We have compared between the execution times of the SQL translation scripts of XMark queries with and without the indexes information. The experiments of this section uses an XMark document instance with a size of 10 MB and use the hybrid approach for generating the SQL translation scripts for XMark Queries. On average, the execution time of the SQL translation scripts with the index information is equal to $\frac{1}{3}$ the execution time without the index information.
(a) Comparison between the direct translation and the pattern translation approaches in terms of the number of the evaluation steps.

(b) Comparison between direct translation and the pattern translation approaches in terms of execution times. The execution times are for the 20 XMark queries ran against 1 MB XMark document instance hosted by DB2.

Fig. 19. Direct translation vs. pattern translation.
5.5 DB2 Native XML vs. DB2 XQuery/SQL

Figure 21 illustrates a comparison between the execution times of our purely relational implementation of an XQuery processor and the native XML/XQuery functionality supported by DB2 pureXML. The execution times of these experiment are collected by running the XMark queries against an XML document instances with a size of 100 MB. The results of this experiment confirms the efficiency advantage of our purely relational approach over DB2 pureXML with respect to the evaluation of all XMark queries. DB2 pureXML could not complete the evaluation of queries Q8, Q9, Q10, Q11 and Q12 within the time frame of two hours. Excluding the uncompleted queries by DB2 pureXML, the execution time of our purely relational approach is on average equal to 15% of the execution time of DB2 pureXML. This experiment indicates that our purely relational approach for implementing XQuery processor deserves to be pursued further.

6 Related Work

The design and implementations of different XQuery processors have been presented in the literature of recent years (Boncz et al., 2005), (Fernández, Siméon, Choi, Marian, & Sur, 2003), (Fiebig et al., 2003), (X-Hive., n.d.), (Paparizos et al., 2003), (Beyer et al., 2005). In principle, these proposed XQuery engines can
be classified into three main classes: Native XML/XQuery Processors, Streaming XQuery Processors and Relational XQuery Processors.

6.1 Native XML/XQuery Processors

The implementations of this approach make use of storage models, indexing and querying mechanisms that have been designed specifically for XML data.

**Timber** TIMBER (Paparizos et al., 2003) is a native XML database which is able to store and query XML documents. In TIMBER, XML data is stored directly in its natural tree structure. It uses the tree-based query algebra (TAX) (Jagadish et al., 2001) which considers collections of ordered labelled trees as the basic unit of manipulation that means each operator on this algebra would take one or more sets of trees as input and produce a set of trees as output. The evaluation of XQuery expressions are achieved through the following steps:

a) The XQuery expressions are parsed into an algebraic operator tree by the query parser.

b) The query optimizer reorganizes this tree and based on a set of rules it performs the required mapping from logical to physical operators (Paparizos et al., 2002).
c) The resulting query plan tree is evaluated by the query evaluator and pipelined one operator at a time.

The query execution in TIMBER heavily depends on structural joins. Hence, the authors have developed efficient structural join algorithms (Al-Khalifa et al., 2002) as well as structural join order algorithms (Wu, Patel, & Jagadish, 2003a) to achieve acceptable performance results. Additionally, query optimization in TIMBER involve estimating costs of all promising sets of evaluation plans (Wu, Patel, & Jagadish, 2003b) before selecting the best one.

Natix

Natix (Fiebig et al., 2003) is another native XML database which clusters subtrees of XML documents into physical records of limited size. The XML data tree is partitioned into small subtrees and each subtree is stored into a data page. The Natix architecture consists of three main components: Storage Layer, Service Layer and Binding Layer. The bottommost layer is the storage layer which manages all persistent data structures. On top of it, the service layer provides all DBMS functionality required in addition to simple storage and retrieval. The binding layer consists of the modules that map application data and requests from other APIs to the Natix engine and vice versa. The two components responsible for query processing in Natix are the Query Compiler and the Query Execution Engine. The query compiler in Natix follows the following steps:

1) The parser module generates an abstract syntax tree for the input query.
2) The NSFT module performs Normalization, Semantic analysis, Factorization of common sub-expressions and Translation into an internal representation. This internal representation is a mixture of Natix algebra (Brantner et al., 2005) and a calculus representation.
3) Some query rewriting rules are applied such as queries unnesting.
4) The plan generator module replaces the calculus representation of query blocks with algebraic expressions.
5) The code generator module generates the code for the query evaluation plan.

The query execution engine consists of an iterator-based implementation of algebraic operators which process ordered sequences of tuples. Tuple attributes either hold base type values such as strings, numbers and tree node references, or ordered sequences.

DB2/System RX

In (Nicola & Linden, 2005), Nicola and Linden have described the native XML support and XQuery implementation in IBM DB2. In this work, DB2 introduces the new XML data type which can be used like any other SQL type. A column of type XML can hold one well-formed XML document for every row of the table while the NULL value is used to indicate the absence of an XML document. Relational and XML data are stored differently, while the relational columns are stored in traditional row structures, the XML data is stored in hierarchical structures. An XML column can hold
schema-less documents as well as documents for many different or evolving XML
schemas. Schema validation is optional on a per-document basis i.e. the asso-
ciation between schemas and documents is per document and not per column,
which provides maximum flexibility. Every node contains pointers to its parent
and children to support efficient navigational queries. Thus, path expressions are
evaluated directly over the native format on buffered pages without copying or
transforming the data. Additionally, the store also supports direct access to a
node, which avoids the top-down traversal through every node from the root to
the target node. DB2 support three classes of XML indexes:

1) *Structural Indexes* which map distinct node names, paths, or tag-based path
expressions to all matching node instances.
2) *Value Indexes* which allow quick retrieval of nodes based upon the nodes data
value.
3) *Full-text Indexes* which map tokens to the nodes that contain the token.

DB2 supports interfaces for both SQL/XML (Eisenberg & Melton, 2002), (Eisenberg
& Melton, 2004) and XQuery (Boag et al., 2006) as the primary languages for
querying XML data in an integrated and unified query model. Different parsers
are used to read SQL/XML and XQuery queries, after which a single compiler
is used for both languages where no translation from XQuery to SQL is done.
The queries compilation is done using the following three steps:

1) The query statement is compiled into an internal query graph model (QGM)
(Pirahesh, Hellerstein, & Hasan, 1992), which is a semantic network used to
represent the data flow in a query.
2) Rewrite transformations are applied to normalize, simplify, and optimize the
data flow.
3) The optimizer uses this graph to generate a physical plan, which is translated
into executable code by the process of code generation.

The optimizer scans a QGM graph and produces alternative execution plans.
The optimizer utilizes data statistics to build a cardinality model, which is then
used to estimate costs for the execution plans. However, intermediate plans can
be pruned based on costs and plan properties such as the order of input data
after which the cheapest cost is chosen for execution. In fact, the DB2 XQuery
implementation does not implement static typing and does not normalize the
XPath expression into explicit FLWOR blocks, where iteration between steps
and within predicates is expressed explicitly.

### 6.2 Streaming XQuery Processors

The implementations of this approach receive the XML data in the form of
continuous streams of tokens and applies on-the-fly the query processing func-
tionalities over them.
**BEA/XQRL** In (Florescu et al., 2003) Florescu et al. have described the design and the implementation of the BEA/XQRL streaming XQuery processor. The processor is a central component of the 8.1 release of BEAs WebLogic Integration (WLI) product (BEA WebLogic Integration., n.d.) and was designed to provide very high performance for message processing applications. In BEA/XQRL, XML data is represented as a stream of tokens which minimizes the memory requirements of the engine and allows the lazy evaluation of queries. At runtime, each runtime operator consumes its input a token at a time and input data that is not required is simply discarded. The query engine is implemented entirely as a library, so it is can be embedded in any application that might need to manipulate XML data. The XQuery compiler is composed of three managers: the **Expression Manager**, the **Context Manager** and the **Operation Manager**. In addition, there are three functional components: the **Query Parser**, the **Query Optimizer** and the **Code Generator**. The expression manager holds the internal representation for all kinds of XQuery expressions and implements various functionalities required for query optimization like variable and substitution management, type derivation, semantic properties derivation, copying, sub-expression cut and paste, etc. The operation manager holds all the information about the first-order functions and operators available to the query engine such as the operator names and signatures, semantic properties, pointers to the class implementing each operator and to the Java code for type derivation of polymorphic operators. The context manager holds the context information passed through all query processing phases in the form of a variety of environmental properties. The task of the parser component is to translate the input XQuery string into the corresponding internal representation. Subsequently, the task of the optimizer component is to translate the expression generated by the parser into an equivalent expression that is cheaper to evaluate. The task of the code generator component is to translate the internal representation into an executable plan represented as a tree of token iterators. Finally, the **Runtime System** interprets the query execution plan using a library of iterators containing implementations for all functions and operators of XQuery. Unfortunately, the BEA/XQRL XQuery process can not process larger XML documents. Additionally, it does not handle aggregate functions as well as its query optimizer only makes use of heuristics instead of using a cost-based model.

**FluXQuery** In (Koch, Scherzinger, Schweikardt, & Stegmaier, 2004a), Koch et al. have presented the FluXQuery streaming XQuery processor that is based on an internal query language called FluX. FluX extends the main structures of XQuery by introducing a construct for event-based query processing. It firstly translates the input XQuery expressions into its internal query language (Koch, Scherzinger, Schweikardt, & Stegmaier, 2004b). The buffer size is then optimized by analyzing the schema constraints derived from the DTD information as well as the query syntax. The optimized FluX queries are then transformed into physical query plans which are translated into executable JAVA code or interpreted and executed using the **Streamed Query Evaluator**. Although, the FluXQuery...
XQuery processor is designed with a strong emphasis on buffer-conscious query processing on structured data streams, it suffers from the same limitations of the BEA/XQRL processor.

6.3 Relational XQuery Processors

The implementations of this approach makes use of the relational indexing and querying mechanisms for querying the source XML data. Although there are large body of research work have been done on the domain of relational XPath evaluation, it is surprising that very few approaches have tried to leverage the relational systems in the XQuery domain. To date and to the best of our knowledge, the work of this article is the first instance of a purely relational implementation of an XQuery processor that can reside in any conventional RDBMS and exhibits the efficiency, scalability and the well-know maturity of the relational infrastructure.

In (Manolescu, Florescu, & Kossmann, 2001), Manolescu et al. have presented the first attempt for translating XQuery expression into SQL queries. The approach of this work is implemented in the Agora data integration system (Manolescu, Florescu, Kossmann, Xhumari, & Olteanu, 2000). In Agora, relational and XML data are defined as a view over the global schema (Levy, 1999). Agora employs XML as the user interface format. When queries posed to the system, all data flows inside the query processor consists of relational tuples and the results are formatted as XML making the underlying relational engine transparent to the user. In Agora, the source XML documents are stored using edge-based relation schema similar to the one presented in (Florescu & Kossmann, 1999) as a fully normalized version of the hierarchical structure of an XML documents. Unfortunately, this approach only supports the translation of a very limited subset of the XQuery language. Additionally, the use of edge-based encoding does not capture any information on the document order. This means that the results of the translation are limited to nested loops as an attempt to preserve the document order.

In (DeHaan, Toman, Consens, & Özsu, 2003), DeHaan et al. have presented an approach for translating XQuery expressions into a single equivalent SQL query statement using a dynamic interval encoding scheme for the source XML documents. In principle, the used dynamic interval encoding in this work is very similar to the XPath accelerator mapping scheme (Grust, 2002) we are using in our work. In this work, the authors have proposed a compositional translation of a subset of the XQuery language into SQL that supports arbitrary combinations and nesting of basic functions and FLWOR expressions without the need of using a general purpose programming language. The translation of XQuery expressions to relational queries is done by translating the XQuery expressions into a basic set of operations on XML forests and then using a set of SQL templates for fragments of queries that are composed to produce the final query. In fact, the work of (DeHaan et al., 2003) is similar to our approach described in this article. However, our approach supports a larger subset of the XQuery language. Additionally, DeHaan’s work suffers from a major limitation in
In order to solve this problem the authors have proposed some modification to the relational engine to achieve an acceptable performance while our proposed approach are purely relational and does not require any changes to the underlying DBMS.

The Pathfinder/MonetDB XQuery processor (Boncz et al., 2006), as its name implies, consists of two main parts: the Pathfinder XQuery Compiler and MonetDB, a relational main memory database system (Boncz, 2002). MonetDB is a main memory database system which is mainly focusing on exploiting CPU caches for query optimization. The Pathfinder/MonetDB XQuery processor encodes the source XML documents using the XPath Accelerator relational mapping scheme (Grust, 2002). Since MonetDB only supports the use of Binary Association Tables (BATs) for storing the relational data, every relation in MonetDB is vertically fragmented into BATs. Each BAT consists of two columns, a head, often containing a unique identifier and a tail for storing attribute data. Hence, to get a full table view of a relation, its BAT fragments have to be joined together. The choice of using a data storage model with binary tables only was with the aim to minimize main memory access. The Pathfinder XQuery compiler translates XQuery expression into an algebra very similar to the conventional relational algebra (Grust & Teubner, 2004). The intermediate algebraic plans are translated into scripts of the Monet Interpreter Language (MIL) (Boncz & Kersten, 1999) which is then passed to and -can be only-processed by the MonetDB’s MIL interpreter (Rittinger, 2005). Although the approach of Pathfinder/MonetDB XQuery processor has been shown to be one of the fastest XQuery processors, it is very tightly coupling to the Monet DBMS and it requires huge main memory sizes to store large XML documents.

7 Conclusion

This article presented Pathfinder as a purely relational implementation of an XQuery processor. In Pathfinder, firstly the source XML documents are encoded using the XPath accelerator relational encoding scheme. Secondly, using the loop-lifting technique, XQuery expressions are translated into intermediate algebraic plans. Thirdly, Pathfinder algebraic plans are then optimized and annotated with their own special properties for the algebraic operators. Finally, Pathfinder uses a certain group of well-defined SQL translation templates and utilize the cardinality information inferred by relational cost model to translate the intermediate algebraic plans into equivalent enhanced cardinality-aware SQL scripts and to influence the RDBMS query optimizers for a better selection for the SQL execution plans. The experiments of this article demonstrated the efficiency and scalability of our purely relational approach in comparison to the native XML/XQuery functionality supported by conventional RDBMSs and has shown that our purely relational approach for implementing XQuery processor deserves to be pursued further.
Acknowledgments

The author of this work would like to thank the Pathfinder people at TUM (Technical University of Munich) for their support, valuable comments and providing the required infrastructure for achieving the work of this article.
References


