WPI-CS-TR-02-25 Oct 2002

XML Algebra Optimization.

by

Xin Zhang
Bradford Pielech
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

XML Algebra Optimization.

Xin Zhang, Bradford Pielech and Elke A. Rundesnteiner
Department of Computer Science
Worcester Polytechnic Institute Worcester, MA 01609
(xinz|winners|rundenst)@cs.wpi.edu

Abstract

Mapping of XML data into and out of relational database systems, including query processing over such virtual XML views that wrap relational sources, has become a topic of critical importance recently. The Rainbow XML data management system, being developed at WPI, focuses on the processing and optimization of XQuery queries against XML views of that relational data. For this, Rainbow's query model, XML Algebra Tree (XAT), has been designed. Because the XML formatting operators are interleaved with the computation operators, this XAT must then be optimized before being translated into SQL. For this, our computation pushdown technology splits the XAT into the XML-specific and SQL-doable operators with the later then being converted into SQL statements. However, the XAT after computation pushdown may contain unreferenced columns or unused operators. We show that these unneeded operators cannot be discovered by the relational engine after SQL generation. Leaving these operators in the tree would create unnecessarily large SQL statements that will slow down the overall execution.

Our main contributions to XML query processing, described in this paper, are threefold. One, we describe the XAT algebra for modeling XQuery expressions. Two, we propose rewriting rules to optimize XQueries by cancelling operators. Lastly, we describe a cutting algorithm that removes unreferenced columns and operators from the XATs. We have implemented the techniques discussed in this paper in the Rainbow system. Our experimental study compares the performance of execution before and after operator cancel out and before and after cutting.

Keywords: XML, query processing, algebra, XQuery, XML to relational, query optimization

1 Introduction

1.1 Motivation

XML [1] has emerged as a popular choice for exchanging structured data between web applications. Because of its balance of power and flexibility, it is quickly becoming the standard for data representation on the Web. Combining these strengths of the XML data model with the maturity of relational database technology in one larger system makes a lot of sense given the demand for both reliable, persistent storage as well as a flexible means to exchange data between applications.

Combining these technologies creates major challenges that must be tackled. One is to load, model, and extract XML data into and out of the relational backend system without losing the semantics of the data. Creating an XML view of the relational data has emerged as the leading architectural design to bridge the gap between these two different data models [2, 7, 5] because of the expressive power of XQuery to define such mapping views that wrap the relational source. Such views can limit the data exposed to an application. They also wrap the data in any XML format desired by the applications.

The second challenge is to efficiently process queries against these XML views. The end user can specify queries against such XML views using their favorite XML query language. The XML query engine then composes the user's query with the mapping query (view definition) to compute what the user desires. The difficulty with this scenario arises due to the mismatch since the language to query the XML views and the language to query the underlying relational database are different.

Using the same language to define and query the XML views allows us to merge the query that defines the view with the user query over that view to form one larger query. That larger query then needs to be translated into SQL to be executed over the relational database. However translation at the syntax level is difficult to control and even more difficult to optimize [6]. Therefore, instead we propose an XML-based algebra tree (XAT) representation of queries as a basis for this query optimization and translation process. Our XML query engine represents one XQuery expression as XAT algebra tree, handles the merging of the user and view definition queries and the optimization of the query plan before SQL pushdown.

1.2 Running Example

The following XML, adapted from XML Use Cases [12], will be used as the running example throughout this paper. The XML document describes a price list of multiple books where each book has one title, one publication source, and one price are given in Figure 1. The example XQuery in Figure 2 returns the distinct book's title in the price list. The query result is shown in Figure 3. Figure 4 depicts the view query built on top of the relational tables that virtually restores the original XML document view according to Figure 1. The user queries are specified against this view. Please notice that in Figure 4, $< source >$ and $< price >$ are not used by the user query depicted in Figure 2.

1.3 Problem Definition

This paper focuses on the challenges faced when trying to optimize the XML algebra tree (XAT) generated by merging the view definition and the user query (depicted in Figures 4 and 2 in our running example respectively). The goal of this optimization step is rewrite the XAT into two portions such that the top of the query contains the XML-specific operators (such as those that create XML fragments) and the bottom half contains the SQL-doable operators (such as Select and Join). This separation ensures that the bulk of the computation is pushed down to the relational engine and only computation not typically doable by a standard relational engine is performed by the Rainbow execution engine.

```
<prices>
  <book>
    <title>TCP/IP Illustrated</title>
    <source>www.amazon.com</source>
    <price>65.95</price>
  </book>
  <book>
    <title>TCP/IP Illustrated</title>
    <source>www.bn.com</source>
    <price>69.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <source>www.amazon.com</source>
    <price>34.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <source>www.bn.com</source>
    <price>39.95</price>
  </book>
</prices>
```

Figure 1: XML Data Example.

```
<result> {
  FOR $t IN
      distinct(document("prices.xml")/book/title)
  RETURN
    <booktitle>
      $t/text()
    </booktitle>
}</result>
```

Figure 2: User XQuery Example.

```
<results>
  <booktitle>TCP/IP Illustrated</booktitle>
  <booktitle>Data on the Web</booktitle>
</results>
```

Figure 3: Query Result of User XQuery.

```
<prices>
  FOR $book IN document("dxv.xml")/book/row,
      $store IN document("dxv.xml")/store/row,
      $prices IN document("dxv.xml")/prices/row
  WHERE $book/bid = $prices/bid AND
        $source/sid = $prices/sid
  RETURN
    <book>
      $book/title,
      $store/source,
      $prices/price
    </book>
</prices>
```

Figure 4: View XQuery to go from Relational Database to XML

However, just attempting to maximally push all SQL-like computation down does not necessarily correspond to an optimal execution plan. Once SQL is to be generated in the lower portion of the XML algebra tree (XAT), note that data may be extracted from the relational database that would not be required by the user's query. For example, the detailed information about `source` and `price` of the books they wrote in the view defined in Figure 4 are not of interest to the user as expressed by the user query in Figure 2.

1.4 Contributions

The core contributions of this paper are three fold. First, we design an XQuery algebra, called XAT, which is capable of capturing the core features of XQuery. Using this algebra, we build an XAT that represents a logical plan for execution of the XQuery. The second contribution are heuristic-based rewriting rules for the XAT algebra which define how two operators may be combined or swapped with one another. Lastly, we describe our two-part XAT rewriting strategy that optimizes the XAT logical query plan for efficient query execution either in main memory or against a relational database. First, the rewriter iteratively applies our rewriting rules in a bottom up fashion to remove intermediate XML fragment construction and to push computation down to the relational engine. Second, it applies our top-down cutting algorithm to identify and remove unused operators. This process will shrink the XAT, resulting in a quicker execution as supported by our experimental results, reported in this paper and our system demonstrated at ACM-SIGMOD 2002 [15].

1.5 Outline

In the next section we briefly review the data model and operators of our XAT algebra. Section 3 introduces the Rainbow framework, in particular we review XAT generation, decorrelation and query merging. Query rewriting rules are given in Section 5. Section 6 describes the redundant operator cutting algorithm. Section 7 reports the experimental results. Related work and conclusions are given in Sections 8 and 9 respectively.

2 XML Algebra Tree (XAT)

2.1 XAT Data Model Components

The operators in the XAT must pass intermediate results to one another, which are ordered collections of related XML fragments. For this reason, the XAT data model is a tuple based ordered table called XAT table. Inspired by W3C's XQuery 1.0 Formal Semantics [13], every cell of a tuple is one of the following types:

- An atomic value
- A node including XML Element, XML Document, and XML Attribute
- An unordered or ordered collection of items (i.e, any mixture of nodes and atomic values).

Every column is denoted by a column name, which can be either a variable binding from the user-specific XQuery, e.g., $\$v$, or otherwise an internally generated variable, e.g., col_n. Every column is typed by an XQuery type defined by W3C [13].

2.2 XAT Operators

The XAT algebra has two purposes. First, it is used to explicitly represent the semantics of XQuery by composing well-defined building blocks, namely algebra operators. Second, it is used as foundation for the query engine in the Rainbow [4] system to allow for query optimization and processing. Every XAT operator's input and output are XAT tables defined in Section 2.1.

The XAT operators are classified into three categories: XML operators, SQL operators, and special operators. XML operators (as shown in Table 1) are used to represent the XML document related operations, e.g., navigation and construction. They are expose, tagger, navigate unnesting, navigate collection, aggregate, and XML set operators. Special operators (as shown in Table 2) include operators used temporarily in different phases of optimization and operators shared by the class of XML and of SQL operators. They are SQLstmt,

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Expose	ϵ	*col*	N/A	s	Expose column col as XML documents or fragments.
Tagger	T	p	*col*	s	Taggering s according to list pattern p.
Navigate	ϕ/Φ	$col1, path$	$col2$	s	Navigate into column col of s based on path $path$.
Aggregate	Agg	*col*	N/A	s	Make a collection for each column. col is the focus column name.
XML Union	$\overset{x}{\cup}$	$col+$	*col*	s	Union multiple columns into one.
XML Intersect	$\overset{x}{\cap}$	$col+$	*col*	s	Intersect multiple columns into one.
XML Difference	$\overset{x}{-}$	$col+$	*col*	s	Compute the difference between two columns.

Table 1: XML XAT Operators.

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
SQLstmt	SQL	$stmt$	$col+$	N/A	Execute a SQL query statement $stmt$ to underlying database.
Function	$\{F\}$	$param+$	*col*	$s?$	XML or user defined function over optional input table with given parameters.
Source	S	$desc$	$col+$	N/A	Identify a data source by description $desc$. It could be a piece of XML fragment, an XML document, or a relational table.
Name	$\rho, \rightarrow$	$col1, col2$	N/A	s	Rename column $col1$ of source s into $col2$.
Name	$\rho, \rightarrow$	ns	N/A	s	Rename table s into new name ns.
FOR	FOR	*col*	N/A	s, sq	FOR operator iterate over s and execute subquery sq with the variable binding column.
IF_THEN_ELSE	IF	c	N/A	sq_1, sq_2	If condition c is true, then execute subquery sq_1, else execute subquery sq_2.
Merge	M	N/A	N/A	$s+$	Merge multiple tables into one table based on tuple order.

Table 2: Special XAT Operators.

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Project	π	$col+$	N/A	s	Project out multiple columns from input table s.
Select	σ	c	N/A	s	filter input table s by condition c.
Cartesian Product	$\times$	N/A	N/A	s_1, s_2	Cartesian product of the results of two input tables, s_1 and s_2.
Theta Join	$\bowtie$	c	$col+$	ls, rs	Join two input tables ls and rs under condition c, and output selected columns.
Outer Join	$\overset{\circ}{\bowtie}_L, \overset{\circ}{\bowtie}_R$	c	$col+$	ls, rs	Left (right) outer join two input tables ls and rs, and output selected columns.
Distinct	δ	$col+$	N/A	s	Eliminates the duplicates in the columns $col+$ of input table s by grouping.
Groupby	γ	$col+$	N/A	s, sq_g	Making temporary groups by multiple columns from input table s, then evaluate subquery sq_g for each group, then merge the evaluated results back.
Orderby	τ	$col+$	N/A	s	Sort input table s by multiple columns.
Union	$\cup$	N/A	N/A	$s+$	Union multiple sources together.
Outer Union	$\overset{\circ}{\cup}$	N/A	N/A	$s+$	Outer union multiple sources together.
Difference	$-$	N/A	N/A	ls, rs	Difference between two sources.
Intersect	$\cap$	N/A	N/A	$s+$	Intersect multiple sources.

Table 3: SQL XAT Operators.

function, source, name, FOR, if-then-else, and merge operators. SQL operators (as shown in Table 3) are those operators with relational algebra counterparts including project, select, Cartesian product, joins (theta, outer), distinct, groupby, orderby, and set operators.

Each table describes the operator's name, its symbol, its parameters, the notion of the output columns, and its sources of data and subqueries, with their description. A detailed discussion of the algebra along with its semantics and numerous examples can be found in [16].

3 Framework of Rainbow

3.1 Loading and Extracting of XML Data

The XML management system, Rainbow, has been designed to exploit relational database technology to manage XML data based on a flexible mapping strategy [14]. The Rainbow system is composed of three sub-systems, e.g., a loading manager, a mapping manager, and an XML query engine build on top of a relational database. Rainbow supports flexible mapping between XML data and relational data in the loading manager. The system can flexibly switch between several different mapping strategies by changing the XQuery loading expression. The mapping manager provides an extraction view query for each loading query. This view query specifies the logic of the reconstruction of the original XML document.

3.2 Query Engine

The *Rainbow Query Engine*, the core part of the *Rainbow* system, processes XML queries. Rainbow's query engine uses the XML algebra, XAT, introduced in Section 2, for optimization and execution of queries. Below we briefly describe the XQuery to SQL translation procedure, while Sections 5 and 6 describe the rewrite engine and cutting algorithm, respectively.

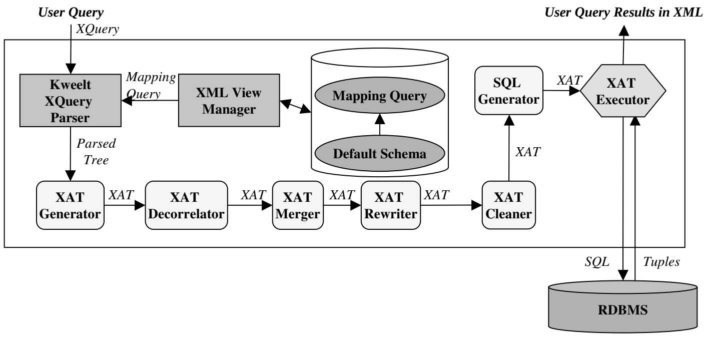

Figure 5: Architecture of Rainbow Query Engine.

Figure 5 depicts the *Rainbow Query Engine* that uses the *Kweelt Parser* [11] to parse XQuery expressions. The *XAT Generator* analyzes the parsed tree and generates an *XML Algebra Tree* (XAT) for both the user and mapping queries. The generated XATs are then unnested by the *XAT Decorrelator*. The two XATs are merged by the *XAT Merger*, generating one final merged XAT that has the user's query on top of the mapping query. See Figure 6 for the XAT at this point. The *XAT Rewriter* rewrites the merged XAT so as to push as much translatable into SQL computation down to the bottom of the tree (see Figure 16). *XAT Cleaner* eliminates unused columns in the XAT tables and detects and removes unused operators (see Figure 18). *SQL Generator* generates SQL from the bottom portions of XAT, which is then executed against the underlying relational database by the *XAT Executor*. The tuples returned by the SQL engine are then tagged into XML elements using construction information from XML specific operators at the top in XAT. Finally, the results in the form of an XML document are returned back to the user. In this paper, we will focus on the *XAT Rewriting* and *XAT Cleaner*.

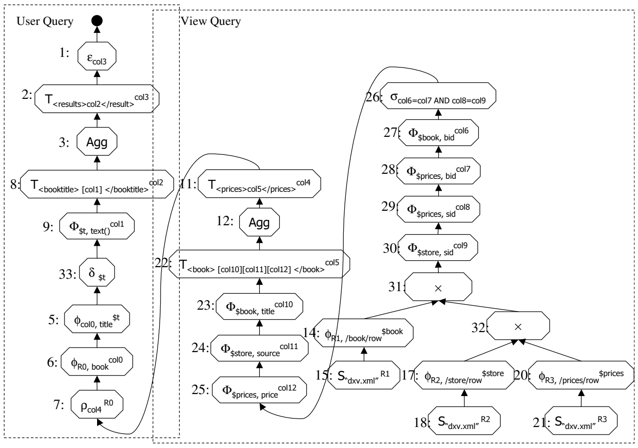

Figure 6: XAT after Generation, Decorrelation, and Merging

4 Rewriting Rules

4.1 Navigation Merging

There are two types of navigation operators, i.e., navigate unnest (ϕ) and navigate collection (Φ). The following four rules dictate how to merge different navigations.

$$\phi^{col2}_{col1,path1}(\phi^{col1}_{col0,path2}) \leftrightarrow \phi^{col2}_{col0,path2/path1} \tag{1}$$

$$\phi^{col2}_{col1,path1}(\Phi^{col1}_{col0,path2}) \leftrightarrow \phi^{col2}_{col0,path2/path1} \tag{2}$$

$$\Phi^{col2}_{col1,path1}(\Phi^{col1}_{col0,path2}) \leftrightarrow \Phi^{col2}_{col0,path2/path1} \tag{3}$$

$$\Phi^{col2}_{col1,path1}(\phi^{col1}_{col0,path2}) \rightarrow N/A \tag{4}$$

Assumption 1 *For the above four rules, we assume the $col1$ is not reference anywhere, otherthan the two involved navigate operators.*

Let's use following example to illustrate the intuition of the four rules. Assume we have element a, which has two sub-elements $b1$ and $b2$, and element $b1$ has sub-elements $c1$, $c2$ and $c3$, and element $b2$ has sub-elements $c4$, $c5$, and $c6$. The tag for $b1$ and $b2$ is b, and tag for $c1 - c6$ is c.

/
a

$\phi^{/b}_{/,b}$

/	/b
a	b1
a	b2

$\phi^{/b/c}_{/b,c}$

/	/b	/b/c
a	b1	c1
a	b1	c2
a	b1	c3
a	b2	c4
a	b2	c5
a	b2	c6

$\rightarrow$

/
a

$\phi^{/b/c}_{/,b/c}$

/	/b/c
a	c1
a	c2
a	c3
a	c4
a	c5
a	c6

Figure 7: Example for Rule 1

/
a

$\Phi^{/b}_{/,b}$

/	/b
a	{b1, b2}

$\phi^{/b/c}_{/b,c}$

/	/b	/b/c
a	{b1, b2}	c1
a	{b1, b2}	c2
a	{b1, b2}	c3
a	{b1, b2}	c4
a	{b1, b2}	c5
a	{b1, b2}	c6

$\rightarrow$

/
a

$\phi^{/b/c}_{/,b/c}$

/	/b/c
a	c1
a	c2
a	c3
a	c4
a	c5
a	c6

Figure 8: Example for Rule 2.

/
a

$\Phi^{/b}_{/,b}$

/	/b
a	{b1, b2}

$\Phi^{/b/c}_{/b,c}$

/	/b	/b/c
a	{b1, b2}	{c1, c2, c3, c4, c5, c6}

$\rightarrow$

/
a

$\Phi^{/b/c}_{/,b/c}$

/	/b/c
a	{c1, c2, c3, c4, c5, c6}

Figure 9: Example for Rules 3.

/
a

$\phi^{/b}_{/,b}$

/	/b
a	b1
a	b2

$\Phi^{/b/c}_{/b,c}$

/	/b	/b/c
a	b1	{c1, c2, c3}
a	b2	{c4, c5, c6}

? $\rightarrow$?

Figure 10: Example for Rule 4.

Figure 8 illustrates the intuition of the Rule 2. As we can see the column $/b/c$ on the left is the same as the column $/b/c$ on the right, which are both separate values. Figure 9 illustrates the intuition of the Rule 3. As we can see the column $/b/c$ on the left side is the same as the column $/b/c$ on the right side, which are both collections. Figure 7 illustrates the intuition of the Rule 1. As we can see the column $/b/c$ on the left is the same as the column $/b/c$ on the right, which are both separate values. Figure 10 illustrates the intuition of the Rule 4. As we can see the column $/b/c$ on the left side not equivalent either to the column $/b/c$ on the right side of Rule 1 as separate values, or to the column $/b/c$ on the right side of Rule 3 as a collection.

4.2 Navigate Pushdown Rules

This section details some of the most commonly used rewrite rules in Rainbow. Rule 5 states that a Navigate operator can be swapped with another Navigate unless there is a dependency on its child. Rule 6 says that a Navigate may always be pushed through a Select, but a Select can only be swapped with a Navigate if the output column of the Navigate is not in the expression of the Select. The final Navigate Pushdown rule applies to a Navigate and either a Cartesian Product, any of the Joins, and the Union operators. This states that the Navigate is moved to the side of the binary operator whose output table contains the entry point of the Navigate.

$$\phi^{x2}_{x1,path1}(\phi^{y2}_{y1,path2}) \leftrightarrow \phi^{y2}_{y1,path2}(\phi^{x2}_{x1,path1})\ [if\ x1! = y1\ and\ x2! = y2.] \tag{5}$$

$$\begin{aligned} &\phi^{x2}_{x1,path1}(\sigma_c) \rightarrow \sigma_c(\phi^{x2}_{x1,path1}) \\ &\sigma_c(\phi^{x2}_{x1,path1}) \rightarrow \phi^{x2}_{x1,path1}(\sigma_c)\ [if\ x2\ not\ in\ c.] \end{aligned} \tag{6}$$

$$\phi^{x2}_{x1,path}(\times(t1,t2)) \rightarrow \begin{cases} \times(\phi^{x2}_{x1,path}(t1),t2) & (if\ x1\ in\ DM(t1)) \\ \times(t1,\phi^{x2}_{x1,path}(t2)) & (if\ x1\ in\ DM(t2)) \end{cases} \tag{7}$$

Navigate vs. Aggregate

$$\phi^{x2}_{x1,path}(Agg()) \rightarrow \phi^{x2}_{x1,path} \tag{8}$$

Same on $x2$. But, different effect on $x1$.

$$\Phi^{x2}_{x1,path}(Agg()) \rightarrow Agg()(\Phi^{x2}_{x1,path}) \tag{9}$$

Same on $x2$. But, different effect on $x1$.

Navigate vs. Single Source Operator The single source operator includes distinct (δ), function ($F()$), name (ρ), navigate (ϕ/Φ), orderby (Σ), and tagger (T).

$$\Phi^{y}_{x,path}(Op^z) \rightarrow Op^z(\Phi_{x,path}) \tag{10}$$

The rule between navigate(ϕ/Φ) and project (π) is different, because the project operator will remove columns, hence after the swapping, the project should include the navigate column into the output column list.

$$\Phi^{y}_{x,path}(\pi_z) \rightarrow \pi_{z,x}(\Phi_{x,path}) \tag{11}$$

Navigate vs. Multi-Source Operator The multi-source operators include join ($\bowtie$), merge (M), and set operators.

$$\begin{aligned} \Phi^y_{x,path}(Op(ls,rs)) &\rightarrow Op(\Phi^y_{x,path}(ls),rs) \\ \Phi^y_{x,path}(Op(ls,rs)) &\rightarrow Op(ls,\Phi^y_{x,path}(rs)) \end{aligned} \tag{12}$$

Navigate vs. Single Source with Internal Subqueries The single source operator with internal subqueries includes FOR (FOR), groupby (γ), and if-then-else (if) operators.

$$\Phi^y_{x,path}(Op_{sq}) \rightarrow Op_{\Phi^y_{x,path}(sq)} \tag{13}$$

4.3 Operator Cancel Out

During the query rewriting process, redundant operators may be canceled out. For example, the Navigate operator can be canceled with the Tagger operator, but not vice versa.

$$\begin{aligned} &\phi^y_{x,step}(T^x_{<root><step>[z]</step></root>}) \rightarrow \\ &\begin{cases} T^y_{<step>[z]</step>} & [if\ x\ is\ not\ used\ anywhere\ else.] \\ T^x_{<root>[y]</root>}(T^y_{<step>[z]</step>}) & [if\ x\ is\ used\ somewhere\ else.] \end{cases} \end{aligned} \tag{14}$$

$$\begin{aligned} &\phi^y_{x,/text()}(T^x_{<step>[z]</step>}) \rightarrow \\ &\begin{cases} \rho^y_z & [if\ x\ is\ not\ used\ anywhere\ else.] \\ T^x_{<step>[y]</step>}(\rho^y_z) & [if\ x\ is\ used.] \end{cases} \end{aligned} \tag{15}$$

If we make use of the binding table, the rule 14 can be written as:

$$\phi^y_{x,step}(T^x_{<root>[z]</root>}) \rightarrow T^x_{<root>[y]</root>}(\rho^y_z)\ [if\ step\ is\ used\ in\ binding\ of\ z]. \tag{16}$$

As the XAT tree generated from the parsed tree, a binding table is created to describe how each column has been created. If the column is generated by a navigation operator, it will keep the path for the navigates; if the column is generated by a tagger operator, it will keep the pattern of the tagger; if the column is generated by a source operator, it will keep the source description; if the column is generated by the function, it will keep the function name and its parameters.

Figure 11 depicts the variable dependency graph represented by the binding table for XAT tree in Figure 6.

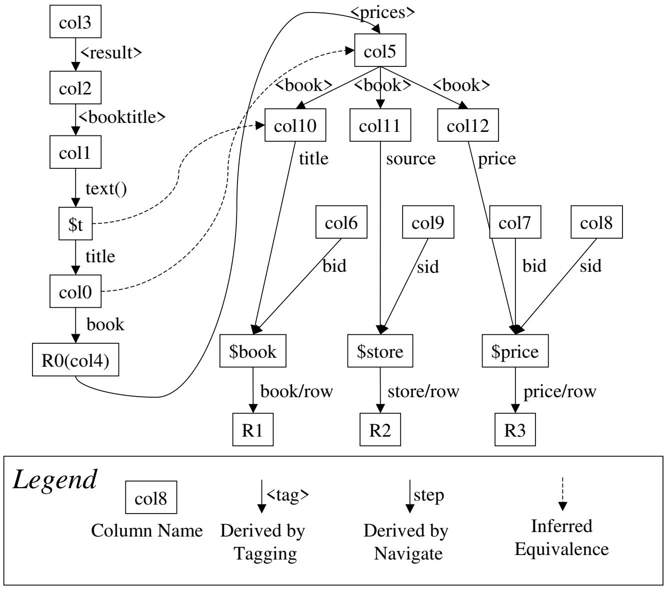

Figure 11: Variable Dependency Graph for XAT in Figure 6.

Binding	Origin	Path/Tagger
col3	col2	$< result >$
col2	col1	$< booktitle >$
col1	$t	text()
$t	col0	title
col0	book	R0
R0	col4	
col4	col5	$< prices >$
col5	col10, col11, col12	$< book >$
col10	$book	title
$book	R1	book/row
col11	$store	source
$store	R2	store/row
col12	$price	price
$price	R3	price/row
col6	$book	bid
col9	$store	sid
col7	$price	bid
col8	$price	sid

Figure 12: Binding Tables.

4.4 Rename Rewriting Rules

Anytime a NameColumn operator is pushed down in the tree, the child of that NameColumn will have its output column renamed if it corresponds to the old name of the NameColumn. The NameColumn can also rename the entry point and destinations of Navigate operators, expressions in the Select operators, or column names in Tagger operators.

$$\rho_{x,y}(< Op >^{x}) \rightarrow < Op >^{y} \tag{17}$$

$$< Op >_{y} (\rho_{x,y}) \rightarrow < Op >_{y} \tag{18}$$

4.5 Operator Transformation Rules

$$\Phi^{y}_{x,path} \leftrightarrow \gamma_{x}(Agg_{y}(\phi^{y}_{x,path}))\ [if\ x\ is\ unique.] \tag{19}$$

4.6 Merge/Split Tagger Operator Rules

$$T^{x}_{p1}(T^{y}_{p2}) \rightarrow T^{x}_{p1.p2}\ \texttt{(If } y \texttt{ is not used besides in pattern } p1\texttt{)} \tag{20}$$

4.7 Orderby Operator Rules

4.8 Position Function Operator Rules

4.9 Merge/Split Select Operator Rules

$$\sigma_{c1}(\sigma_{c2}) \rightarrow \sigma_{c1\ AND\ c2} \tag{21}$$

$$\sigma_{c1\ AND\ c2} \rightarrow \sigma_{c1}(\sigma_{c2}) \tag{22}$$

$$\sigma_{c1\ OR\ c2} \rightarrow \sigma_{c1} \cup \sigma_{c2} \tag{23}$$

5 Query ReWriting

Query rewriting in Rainbow refers to the process of evaluating equivalence rules on an XML algebra tree (XAT) to optimize the tree for efficiency and create a version of the tree where SQL queries can easily be generated. The administrator can choose from several possible rule application heuristics in order to optimize the tree, otherwise default heuristics are chosen.

5.1 XAT Traversing

The XAT Traverser uses a bottom up approach to find an operator to rewrite. The bottom up approach is necessary because operators are moved downwards in the tree. If top down were used, the Traverser would try to push down operators before their dependencies have been rewritten. The operators would be stopped prematurely from being moved maximally down the tree. The Traverser iterates over the tree and applies equivalence rules according to one of several heuristics (Section 5.2). During rewriting, the engine only examines the local operator it is pushing down (x1) and its child (x2). The appropriate rule is chosen based on the types of x1 and x2 and the heuristic. After evaluating the rule, x1 and x2 are typically swapped in the tree, this has the affect of pushing x1 further down in the tree.

5.2 ReWrite Heuristics

Several heuristics can be chosen to rewrite XAT into the normalized form where all XML operators are on the top and all SQL-doable are at the bottom. The order that the heuristics are applied can affect the resulting tree.

1. **Reorganize Navigate operators into logical groups:** After XAT merging the Navigate operators are in arbitrary order. Reorganizing both the NavUnnest and NavCollection operators into logical execution groups makes the execution run more optimially. Pushing the Navigates through the Cartesian Product operators will reduce the cost of executing the Cartesian because it will reduce the result set. In Figure 6 notice that the Navigate operators from 23-20 (excluding the Select operator #26) are in arbitrary order. Navigate #24 and 30 relate to $store, #25, 28, and 29 use $prices, and #23 and 27 use $book. By evaluting rules 5, 6, and 7 in Section 4 we can better organize the Navigates. The tree after applying these rules can be found in 13. Notice that all the Navigates that use $price, $book, and $store are all organized in a row.

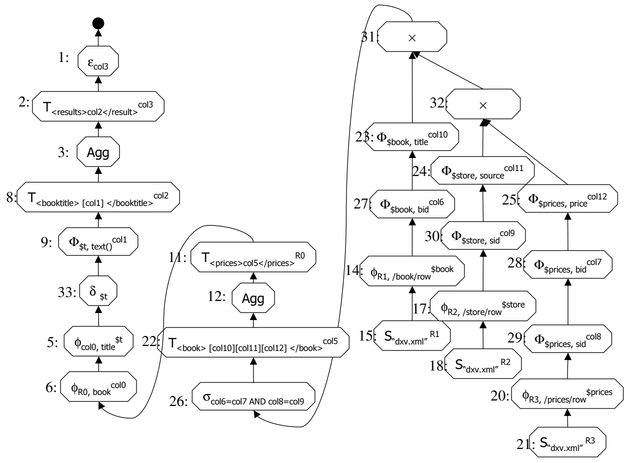

Figure 13: XAT after Navigation Pushdown.

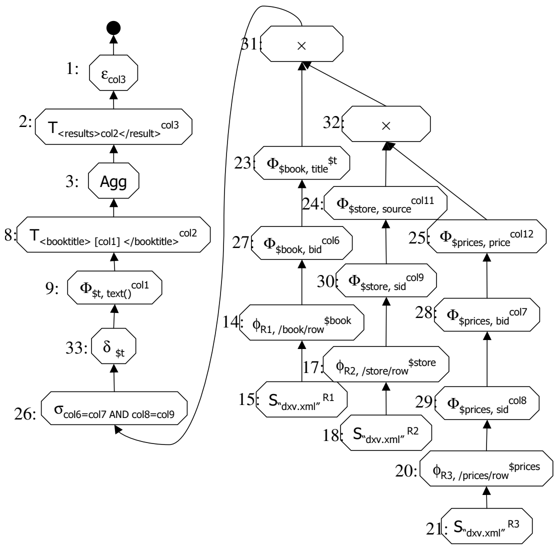

Figure 14: XAT after the Tagger Cancel Out.

2. **Removal of the construction of intermediate XML fragments:** The mapping query generates bags of XML data using the Navigate operators and then XML fragments are formed later from those bags by the Tagger operators. The user query then navigates into said constructed fragments and builds additional bags with Navigate operators and then more bags containing XML fragments with more Taggers operators. This process is redundant as the same information is stored multiple times in intermediate result sets. This redundancy can be eliminated by combining the different Navigate and Tagger operators from the user and mapping query. Once the Navigate pairs have been identified, we can rename the mapping query Navigates to what is specified in the user query and then remove the user query Navigates.

   This heuristic is best illustrated by an example. After applying the above heuristic, we want to identify which intermediate XML fragments and associated Navigate operators we can remove. In Figure 13, operator #23 navigates from $book to title and binds that output bag to col10. Then, operator #22 creates a book element out of col10, col11 and col12 (all 3 columns are elements). Operator #6, Navigates from R0 (which is bound to the output column of the Tagger operator #11) t0 book. Evaluating the rules in 4.3, we can see that we can cancel the Tagger #11 because Navigate #6 ”undoes” it. The binding of Tagger #22 is renamed to col0 because that now serves as the input to the Navigates above. Navigate #5 now references the Tagger #22 because the Navigates entry point matches the output of the Tagger. Navigate #5 navigates from that entry point to title. Looking at the columns in Tagger #22 that were wrapped by the book element, we can see that col10 represents title. Therefore, we can create a NameColumn operator that will rename col1 into $t and delete the Navigate #5 because it is the same as the Navigate #23. Tagger #22 is removed by Schema Cleanup because the other two columns in the Tagger are not referenced above.

The XAT after applying this heuristic can be found in 14.

3. **Propagate Renames:** Some of the above heuristics will create NameColumn operators in the tree. These operators are elminated by pushing them down in the tree and changing the bindings of the corresponding Navigate and Select operators. In the running example, after canceling Navigate and Tagger operators, there are several NameColumn operators in the tree. The traverser starts from the bottom of the tree and moves up until it finds the first NameColumn to push down. Then it evaluates the rules in 4.4 that will rename each operator appropriately. Notice the binding change in operator #23 in Figure 16. The NameColumn operators are deleted when they reach the bottom of the tree.

4. **Cancel Duplicate Navigate Operators:** An XAT may contain one or more equivalent Navigates. These duplicates may exist in both the user and the mapping query portion of the tree and are not necessarily removed in Heuristic 1 because they do not relate to the construction of intermediate XML fragments. The ReWrite engine identifies these operators by comparing the Navigates. If the entry point and destination match, then the top most Navigate is replaced with a NameColumn operator. This NameColumn will rename the output column from the lower Navigate into that of the upper. Figure 15 shows an example of a query with duplicate Navigates. Noticed the $t/text() in the WHERE clause and in the RETURN clause. These will each create a Navigate operator with $t/text() as the path.

```
<result>{
  FOR $t IN distinct(document("prices.xml")/book/title)
  WHERE $t/text() = "Data on the Web"
  RETURN <booktitle>$t/text()</booktitle>
}</result>
```

Figure 15: XQuery Example with Duplicated Navigations.

5. **Computation Pushdown:** Relational engine SQL operators such as Distinct, Select, GroupBy, Join, and Union can be pushed down from the user portion of the query tree into the mapping portion by

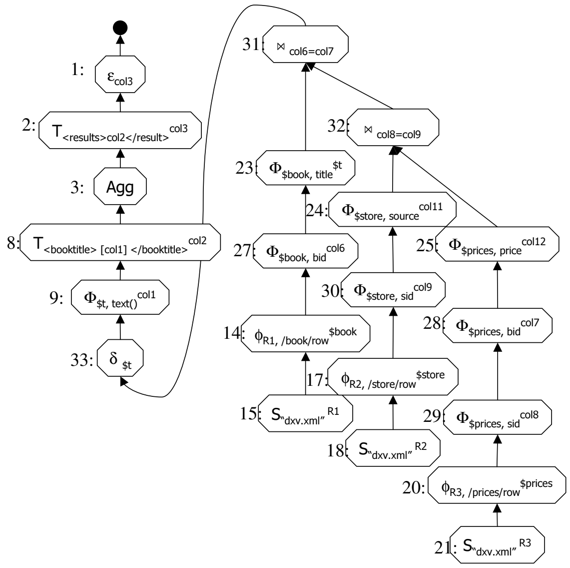

Figure 16: XAT after Select Cartesian Product Merge.

repeatedly evaluating equivalence rules. In the running example, the Select operator #26 can be merged with the Cartesian Product operators #31 and #32 to form Joins. See Figure 16 for the XAT after this merging.

6 XAT Cleanup

XAT cleanup eliminates unused columns in the XAT tables and detects and removes unused operators. This cutting must be done before the SQL generation is performed because the relational database has no way of knowing which columns are used later in the remaining XAT.

6.1 Schema Cleanup

XAT Schema cleanup is used to optimize the size of the intermediate results generated by each operator, effectively reducing the overall space used during XAT execution.

Every operator in the XAT consumes (uses) and modifies some existing columns and produces (creates) other new columns. By recursively analyzing each operator in the XAT, we can compute the columns actually required for each operator. *Consumed* refers to the columns that an operator uses during its execution, such as a Select operator consumes the columns in its predicate. *Produced* is the list of columns created by the execution of an operator. An example of this is a NavigateCollection operator will *Produce* an output column that is the result of executing the path over the document. *Modified* is whether an operator can affect other columns. Table 4 describes the *consumed*, *modified*, and *produced* columns for the operators.

Operator	Consumed	Produced	Modified
$T_p^{col}(S)$	Columns in pattern p	col	No
$C_p^{col}(S)$	Columns in pattern p	col	No
$\phi_{col,path}^{col'}(S)$	col	col'	Yes
$\Phi_{col,path}^{col'}(S)$	col	col'	No
$Agg(S)$	N/A	N/A	Yes
${}^{x\,col'}\cup_{col[1..n]}(S)$	$col[1..n]$	col'	No
${}^{x\,col'}\cap_{col[1..n]}(S)$	$col[1..n]$	col'	No
${}^{x\,col'}-_{col[1..n]}(S)$	$col[1..n]$	col'	No
δ_{col}	col	N/A	Yes
$SQL_{stmt}^{col[1..m]}$	N/A	depends on stmt	No
$F_{param[1..m]}^{col}(S?)$	columns in param[1..m]	col	No
$S_{desc}^{col[1..n]}$	N/A	$col[1..n]$	No
$\rho_{col1,col2}(S)$	$col1$	$col2$	No
$\rho_{ns}(S)$	N/A	N/A	No
$M(s[1..n])$	N/A	N/A	No
$\pi_{col[1..n]}(S)$	$col[1..n]$	N/A	No
$\sigma_c(S)$	columns in condition c	N/A	No
$\times(ls, rs)$	N/A	N/A	Yes
$\bowtie_c (ls, rs)$	columns in condition c	N/A	Yes
$\overset{\circ}{\bowtie}_{Lc}(ls, rs)$	columns in condition c	N/A	Yes
$\overset{\circ}{\bowtie}_{Rc}(ls, rs)$	columns in condition c	N/A	Yes
$\gamma_{col[1..n]}(S, sq)$	$col[1..n]$ and columns in sq	columns generated in sq	No
$\sigma_{col[1..n]}(S)$	$col[1..n]$	N/A	No
$\cup_(S[1..n])$	N/A	N/A	Yes
$\overset{\circ}{\cup}(S[1..n])$	N/A	N/A	Yes
$\cap(S[1..n])$	N/A	N/A	Yes
$-(ls, rs)$	N/A	N/A	Yes
$\epsilon_{desc}^{col[1..n]}$	$col[1..n]$	$col[1..n]$	No

Table 4: Column Consumed, Produced, and Modified for Each Operator.

By default, every operator will simply append new columns at the end of the XAT tables. However, not all the columns will be used later in the XAT. Hence a lot of space and computation time may be wasted. We define the *minimum schema* of an operator as the columns that will be used later by its ancestors.

Intuitively, the minimal schema should include all columns, which produced by this operator and its descendants, that are consumed by the operator's ancestors, but exclude all the columns produced by the operator's ancestors. Hence, we have the following:

$$MS := P_{descendant} \cap (C_{ancestor} - P_{ancestor}) \cup P \tag{24}$$

where MS represents the minimum schema of this operator. P represents the columns produced by the operator whose MS we wish to calculate. $P_{ancester}$ and $C_{ancester}$ are the set of columns produced and consumed, respectively, for the operator's ancestors. $P_{descendant}$ denotes the columns produced by the operator's descendants.

The minimum schema for each operator can be computed step by step by traversing the XAT tree top-down by the following:

Node	Parent	Produced	Consumed	Schema Before	Minimum Schema
1		{}	{col3}	{$col3}	{col3}
2	1	{col3}	{col2}	{R1, $book, col6, R2, $store, col9, R3, $prices, col8, col7, col12, col11, $t, col1, col2, col3}	{col3}
...	...	...	...	...	...
31	33	{}	{col6, col7}	{R1, $book, col6, $t, R2, $store, col9, col11, R3, $prices, col8, col7, $col12}	{$t }
23	31	{$t}	{$book}	{R1, $book col6, $t}	{$t, col6}
...	...	...	...	...	...
32	31	{}	{col9, col8}	{R2, $store, col9, col11, R3, $prices, col8, col7, col12}	{$col7}
...	...	...	...	...	...

Table 5: Schema Cleanup for the XAT in Figure 16.

$$\begin{aligned} &MS := P\,\texttt{(for root operator)} \\ &MS := P_{descendant} \cap (MS_p - P_p) \cup C_p \cup P \end{aligned} \tag{25}$$

where MS_p is the minimal schema for the operator's parent. P_p and C_p are the set of columns produced and consumed, respectively, for the operator's parent.

Table 5 shows the schema for the example in Figure 16, using the formula depicted above. Column *node* stores the node number in Figure 16. Column *parent* stores the parent of this operator. Column *produce* and *consumed* stores the columns produced and consumed by this operator as defined in Table 4. Column *schema before* stores the columns computed before the schema cleanup, which is all the columns produced by the descendents and itself. And column *minimum schema* stores the computed minimum schema of this operator. Please notice the node #23 that its parent node #31 is a binary operator, so the minimum schema of node #23 does not includes the $col7$, which is produced by node #32.

During execution, each operator looks up its minimal schema in the *MS* table (generated before execution for the XAT), and sets its output schema accordingly. An alternative to using this schema cleanup technique would be to insert a Project operator after each operator. This Project operator will ensure that only the columns consumed or modified later in the XAT are kept in the schema.

6.2 Unused Operator Cutting

After the above schema cleanup of the XAT, some operators may produce data that will not be consumed (used) by the operators above it in the XAT. Therefore, those operators are not necessary for the final query result. If we identify those operators, then we can cut them to improve the query execution performance.

Our strategy uses what is called the **cutting matrix**. Our *cutting matrix* is composed of operator identifications (for the row header) and column names (for the column headers). There are six symbols, *r, c, p, m, x,* _, used in the matrix, where *r* represents a required column, *c* a consumed column, *p* a produced column, *m* a modified column, _ a possibly affected column, and *x* denotes a cutable operator.

We first fill the *cutting matrix* with *p, c, and* _ to show which column names each operator produced, consumed and possibly affected. Note that the root of the tree is at the top of the matrix. For example, the

#	parent()	r1	$book	col6	$t	r2	$store	col9	c11	r3	$prices	col8	col7	col12	col1	col2	col3
1																	c
2	1															c	p
3	2	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
8	3														c	p	
9	8				c										p		
33	9	_	_	_	c	_	_	_	_	_	_	_	_	_	_	_	_
31	33	_	_	c	_	_	_	_	_	_	_	_	c	_	_	_	_
23	31		c		p												
27	23		c	p													
14	27	c	p	_	_	_	_	_	_	_	_	_	_	_	_	_	_
15	14	p															
32	31	_	_	_	_	_	_	c	_	_	_	c	_	_	_	_	_
24	32						c		p								
30	24						c	p									
17	30	_	_	_		c	p	_	_	_	_	_	_	_	_	_	_
18	17					p											
25	32										c			p			
28	25										c		p				
29	28										c	p					
20	29	_	_	_	_	_	_	_	_	c	p	_	_	_	_	_	_
21	20									p							

(a) Before Compute Required.

#	parent()	r1	$book	col6	$title	r2	$store	col9	c11	r3	$prices	col8	col7	col12	col1	col2	col3	cut?
1		r	r		r										r	r	r	
2	1															c	p	
3	2	_	_	_	_	_	_	_	_	_	_	_	_	_	_	m	_	
8	3														c	p		
9	8				c										p			
33	9	_	_	_	c	_	_	_	_	_	_	_	_	_	_	_	_	
31	33	_	_	c	m	_	_	_	_	_	_	_	c	_	_	_	_	x
23	31		c		p													
27	23		c	p														
14	27	c	p	_	_	_	_	_	_	_	_	_	m	_	_	_	_	
15	14	p																
32	31	_	_	_	_	_	_	c	_	_	_	c	m	_	_	_	_	x
24	32						c		p									x
30	24						c	p										x
17	30	_	_	_	_	c	p	_	_	_	_	m	m	_	_	_	_	x
18	17					p												x
25	32										c			p				x
28	25										c		p					x
29	28										c	p						x
20	29	_	_	_	_	_	_	_	_	c	p	_	_	_	_	_	_	x
21	20									p								x

(b) After Compute Required and Cut.

Figure 17: Operator Cutting Matrix after Computation.

cell at location $(1, r1)$ in Figure 17(b) contains an r, meaning $r1$ is a required column for operator $\#1$.

Before we discus the rules, we'd like to define two relationships, called *Parent(node1, node2)* and *Ancestor(node1, node2)*. If an operator A's parent is B, then, *Parent(B, A)* is true, otherwise, false. For example, in Figure 17(a), *Parent(1,2)* is true. The *Ancestor(node1, node2)* is defined as follow:

```
Ancestor(A, B) if Parent(A, B)
Ancestor(A, C) if Parent(A, B) and Ancestor(B, C)
```
(26)

For each operator, if a column produced lower in the tree is used higher in the tree, and that column is possibly affected by the current operator, we put an *m* there to show it is modified. For example, the cell $(31, \$title)$ in Figure 17(b) means that the operator #31 modified $\$title$, because $\$title$ was produced by operator $\#23$ ($\#31$'s child) and consumed by operator $\#33$ ($\#31$'s parent).

```
S(A, x)=m if Ancestor(B, A) and S(B, x)=c
 and Ancestor(A, C) and ( S(C, x)=p or S(C, x)=m)
```
(27)

where $S(A, x)$ denotes the schema at operator A column x.

We then update the first row of the matrix with the required columns needed to generate the final result, found in cell $(1, col3)$ in Figure 17. The rules to compute r are as follows:

```
S(1,y)=r
IF S(B,y)=c and (S(B,y)=p or S(B,y)=m) and S(1,y)=r and Ancestor(1, B)
```
(28)

In other words, if the required column is modified or produced by another operator, then all other consumed columns of that operator are required. In our case, we set cell $(1, col3) = r$ as the initial condition. Because, $S(2, col2) = c$ and $S(2, col1) = p$ and $S(1, col1) = r$ and operator #1 is operator #2's parent, the column $col2$ is required. For the same reason, columns $col1$, $title$, $book$ are all *required.*

The last step is to compute the columns that can be cut (removed) from the XAT using the following rule:

```
Operator A is cutable IF Ancestor(1, A) and
for any X in (S(A, X)=p or S(A, X)=m) there is no S(1,X)=r
```
(29)

This means that none of the produced or modified columns of that operator are used in producing the final result. We compute the cutting matrix in Figure 17 for the XAT depicted in Figure 16. As we can see, the operators #31 #32 #24 #30 #17 #18 #25 #28 #20 #21, are deemed unnecessary (by our formula) and can safely be removed from the tree. For example, the operator #32 modified columns $col7$ but $col7$ is not required. Hence the operator #32 can be cut. After the cutting, the XAT is simplified as shown in Figure 18.

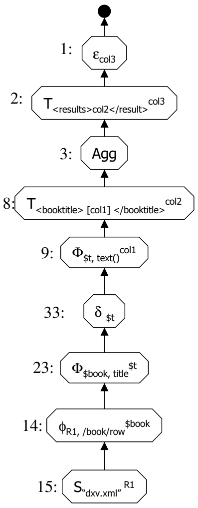

Figure 18: Cutted XAT in Figure 16

7 Preliminary Experiments

We have implemented the Rainbow system in Java JDK1.2 using Xerces for DOM Parser, and Kweelt Engine [11] to generate the parsed tree for each XQuery expression. The rewriting rules are written in Java and can be loaded dynamically, depending on the situation.

Experiment Setup. We use the XQuery statements described in Kweelt's test cases, which in turn were derived from the W3C's test cases. We also designed synthetic data sets with different sizes. All experiments are run on a PIII800 machine with 256 MB memory under Windows 2000 Professional.

Performance Gain. In the first experiment, we compare four kinds of query execution performances. In Figure 19, the x-axis shows increasing data sizes in terms of the number of elements, while the y-axis shows the total execution time for the queries. We use m to denote the number of operators and n the number of tuples. If we assume the XAT is a linear tree, the leaf node keeps only one column, and the root node keeps

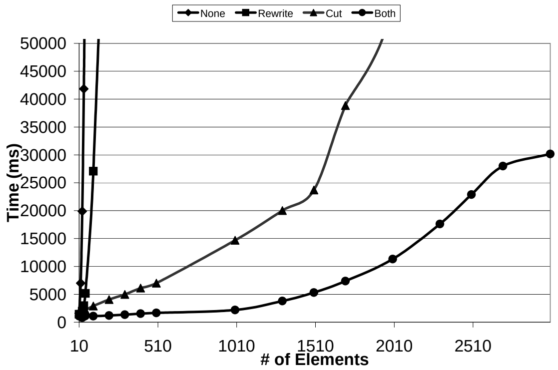

Figure 19: Performance Comparison

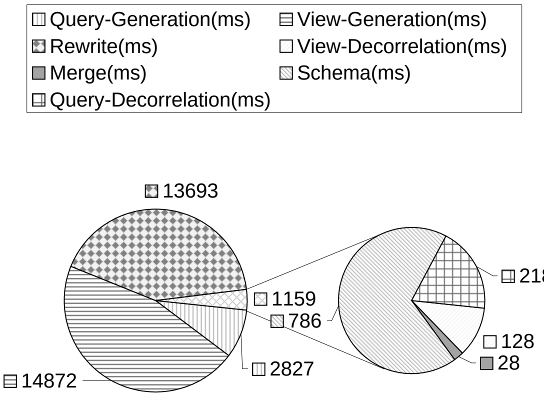

Figure 20: Overhead of Schema Cleanup

m columns, then the total space consumed by the whole XAT in terms of number of values is:

$$n + 2n + 3n + ... + m \times n = \frac{(1+m) \times m}{2} \times n \tag{30}$$

As you can see, the performance without any optimization reaches the memory limits fast ($O(m^2 \times n)$). Thus the performance decreased dramatically because of the memory swapping. Rewriting only can reduce the number of nodes, say k, in the XAT somewhat when such useless operators exist. Therefore, the XAT can handle more data within the same amount of memory. However, there may still be redundant columns. Hence, it's still ($O(k^2 \times n)$).

The performance with schema clean up only scales much better. If we assume that after the schema clean up, every operator only keeps one column. Then the total space consumed by the whole XAT in terms of the number of values is: $n \times m$. As expected, the performance with both the schema clean up and cutting is the best.

Optimizer Overhead. In this experiment we study the overhead of the schema cleanup and the rewriting. Figure 20 depicts the time for different stages of the query processing. The left pie shows the three most expensive steps, i.e., view generation (14,872 ms), rewrite (13,693 ms), and query generation (2,827 ms). The right pie shows the rest four steps, i.e., the schema cleanup and cutting (786 ms), query decorrelation (218 ms), view decorrelation (128 ms), and merge (28 ms). In Figure 20, first we noticed the total time of processing a query before the execution can take up to 30 seconds in our prototyping system. Hence, a mechanism of keeping the optimized query plan in the database instead of compiling it every time for execution can save a lot of time. For the generation of a user tree which in our case is composed of 9 operators only took 3 seconds, while to generate a large view tree of 39 operators can take up to 15 seconds. Hence, for large queries materialization is really a good next step to avoid the compile time and also can save a lot on the execution. The schema clean up is really quick, namely 0.7 seconds, but can significantly improve the performance. Though the rewriting heuristics are complex, the rewriting time of the algebra tree is still shorter than the total generation time.

8 Related Work

Commercial relational systems including Oracle, SQL Server, and DB2 have or are continuing to add support for XML data. Oracle's XMLSQL utility [10] is capable of storing XML data, using a fixed mapping, in its object-relational engine and then queries may be issued against these objects using XSQL, an extension of SQL that supports XPath expressions. Oracle relies on an additional XSLT script to generate the return XML while this element construction is explicit in XQuery. Microsoft's SQL Server 2000 [8] is capable of evaluating XPath expressions over XML views, but not joins or nested queries. IBM's DB2 Data Access Definition (DAD) [3] language generates an XML document using arbitrary selections and joins on existing relational tables. However, the language lacks support for user-defined functions and nested queries.

Similar to Rainbow, both XPERANTO [2] and SilkRoute [5] also process XQueries over virtual XML views of relational data using a library of rewrite rules that push computation down to the relational engine. However, key differences include the data model for XPERANTO, which is the XML Query Graph Model XQGM that does not support collections and is not order-sensitive. XQGM models element construction and navigation using different functions in the project operator. XQGM uses function-composition for cancel out, whereas we propose rule-based optimization for the XAT. Lastly, Rainbow proposes algorithms for redundant join operator cutting while XPERANTO provides only the traditional predicate cutting. Silkroute [5] focuses on generating efficient SQL queries that exploit one-to-one functional dependencies to reduce the number of sorted outer-union SQL queries. In contrast, Rainbow focuses on optimization, such as cutting, at the XML algebra-level before the generation of SQL queries.

AGORA [7] translates XQuery queries over XML views into equivalence SQL queries by rewriting at the XQuery syntax level, not using an algebra. Niagara [9] proposes an XML algebra for the purpose of efficiently processing queries over XML data, not over relational systems. Niagara also proposes a set of equivalence rules, some of which were adopted in the creation of Rainbow's rule library.

9 Conclusions

This paper explains Rainbow's approach to efficient processing of XML queries over virtual XML views of relational data. We describe our XML algebra that captures the semantics of XQuery. Using this algebra, we propose a 2-fold optimization process at the logical XML algebra level. First, equivalence rules are applied according to an array of heuristics in order to push as much computation down to the SQL. Second an algorithm for cleaning the schema for each operator and the cutting of unused operators has been designed. These two techniques together result in a more efficient XAT that can then be converted into compact SQL queries. Preliminary experimental results confirm that the query execution time was significantly quicker after optimization.

References

[1] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensaible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml, Feburary 1998.

[2] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[3] DB2 UDB XML Extender. XML Extender Administration and Programming. http://www-4.ibm.com/software/data/db2/extenders/xmlext/library.html, December 1999.

[4] DSRG. Rainbow: Relational Database Auto-Tuning for Efficient XML Query Processing. http://davis.wpi.edu/dsrg/rainbow, 2002.

[5] M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[6] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice-Hall, 2000.

[7] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with xml and relational. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 623–626. Morgan Kaufmann, 2000.

[8] Microsoft Corp. Microsoft sql server. http://www.microsoft.com.

[9] J. Naughton, D. DeWitt, D. Maier, and J. C. etc. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[10] Oracle Technologies Network. Using XML in Oracle Database Applications. http://technet.oracle.com/tech/xml/htdocs/about_oracle_xml_products.htm, November 1999.

[11] A. Sahuguet. Kweelt: More than just "yet another framework to query xml!". In *Demo Session Proceedings of SIGMOD'01*, page 602, 2001.

[12] W3C. XML Query Use Cases. http://www.w3.org/TR/xmlquery-use-cases, 2002.

[13] W3C. XQuery 1.0 Formal Semantics. http://www.w3.org/TR/query-semantics, 2002.

[14] X. Zhang, G. Mitchell, W.-C. Lee, and E. A. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *RIDE-DM*, pages 111–118, April 2001.

[15] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and E. A. Rundensteiner. Rainbow: Mapping-Driven XQuery Processing System. In *Demo Session Proceedings of SIGMOD'02*, page 614, 2002.

[16] X. Zhang and E. A. Rundensteiner. XAT: XML Algebra for the Rainbow System. Technical Report WPI-CS-TR-02-24, Worcester Polytechnic Institute, July 2002.