10-2002

XML Algebra Optimization

Xin Zhang
Worcester Polytechnic Institute, xinz@cs.wpi.edu

Bradford Pielech
Worcester Polytechnic Institute, winners@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute, rundenst@cs.wpi.edu

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. Three, 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 [2] 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 into one system makes a lot of sense given the demand for both reliable, persistent storage as well as for 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 in 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 [3][10][6]. XQuery's [20] expressive power can define such mapping views that wrap the relational source. Such views can limit the data exposed to an application and 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 would now need to rewrite the user query, combine it with the mapping query, and then to efficiently compute what the user desires.

That larger query would need to be translated into SQL to be executed over the relational database. However, this query rewriting and translation at the syntax level is difficult to optimize [8]. Therefore, instead we propose an XML-based algebra (XAT) representation of XQuery expressions as a basis for these query optimization and translation tasks. Our XML query engine first represents the XQuery expression as XAT algebra trees and then performs the optimization of the query plan using this XML algebra representation before SQL pushdown.

1.2 Running Example

The following XML, adapted from XML Use Cases [21], 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 and one price are given in Figure 1. Figure 2 depicts the relations of shredded XML document in Figure 1. The example XQuery in Figure 3 returns the book's title in the price list. The query result is shown in Figure 4. Figure 5 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 5, $< price >$ is not used by the user query depicted in Figure 3.

1.3 Problem Definition

This paper focuses on the challenges faced when trying to optimize the XML algebra tree (XAT) generated by merging the query tree for the user query with the query tree representing the view definition (depicted in Figures 3 and 5 in our running example respectively). The goal of this optimization step is to first optimize the XAT by rewriting it into equivalent yet more efficient query trees. In particular, our goal is to separate the tree 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 engine.

```
<prices>
  <book>
    <title>TCP/IP Illustrated</title>
    <price>65.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <price>34.95</price>
  </book>
</prices>
```

Figure 1: XML Data Example.

Book

Bid	Title
2	TCP/IP Illustrated
7	Data on the Web

Prices

Bid	Price
2	65.95
7	34.95

Figure 2: Example of Relational Tables.

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

Figure 3: User XQuery Example.

```
<result>
  <title>TCP/IP Illustrated</title>
  <title>Data on the Web</title>
</result>
```

Figure 4: Query Result of User XQuery.

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

Figure 5: View XQuery to go from Relational Database to XML

```
<!ELEMENT prices (book*)>
<!ELEMENT book (title, source, price)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT source (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Figure 6: DTD Example.

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 ultimately may not be required by the user's query. For example, the detailed information about authors and titles of the books they wrote in the view defined in Figure 5 are not of interest to the user as expressed by the user query in Figure 3. We thus are interested in developing an algorithm that assures that such data irrelevant to the user is not unnecessarily computed.

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 the XQuery language. Using this algebra, we construct an XML algebra tree that represents a logical query plan; forming the foundation for the execution of the XQuery expression. The second contribution is the set of equivalence rules for the XAT algebra. These rules define for example how two operators may be combined or swapped with one another. Our rewriting strategy optimizes the XAT logical query plan for efficient query execution either in main memory or against a relational database. For this, 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. Lastly, we describe our top-down cutting algorithm that identifies and removes unused operators. This process will shrink the XAT, resulting in a further optimized query plan. We have implemented the above proposed query model and optimization algorithms in our XML data management system Rainbow [24]. In this paper, we also present some of experiments we have conducted in order to evaluate the performance improvements achievable as well as the overhead of our algorithms in a variety of different settings.

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 and heuristics are given in Sections 4 and 5 respectively. Section 6 describes the redundant operator cutting algorithm. Section 7 reports on our experimental study. Related work and conclusions are given in Sections 8 and 9 respectively.

2 XAT Data Model

2.1 XAT Data Model Components

XAT data model is an order-sensitive table called XAT table. Inspired by W3C's XQuery 1.0 Formal Semantics [22], every entry of a tuple $t \in T$ can be:

- An atomic value.
- A node: includes XML Element, XML Document, and XML Attribute.
- A collection: which is an unordered collection of items (i.e., any mixture of nodes and atomic values).
- A sequence: which is an ordered collection of items.

Every column is denoted by a column name, which can be either a variable binding from the user-specific xquery, e.g., $\$v$, or an internally generated variable, e.g., col_n. Every column col_n is typed by an XQuery type defined in the XML Query Algebra [19]. This means that $t[col_n] \in XQueryDomainType$, where T is an XAT table, $t \in T$ and $t[col_n]$ is the value of tuple t for column col_n.

As we can see the XAT table is an extended relational table with XML domains and support for collections. As relational implementations found the multi-sets (bags) more useful than sets, we expect XAT tables to also be implemented as multi-sets.

Collection and Sequence. A collection is an unordered bag of zero or more items while a sequence is an ordered collection of zero or more items. They both have the following properties:

- A collection/sequence with one item can be treated as a singleton item, and vice versa.
- Collections/sequences cannot be nested into each other.
- A collection/sequence has a schema assigned to it. They are heterogeneous, i.e., there can be different types of items in one collection/sequence. The collection/sequence's type is the super type of the types of items in the collection/sequence.

In the following discussion, we use sequence and collection interchangeably. We only use sequence when we want to highlight the order of a given collection. Otherwise for both ordered and unordered collections we use the term $collection$ henceforth.

title	price
TCP/IP Illustrated	65.95
TCP/IP Illustrated	69.95
Data on the Web	34.95
Data on the Web	39.95

(a)

price
<price>65.95</price>
<price>69.95</price>
<price>34.95</price>
<price>39.95</price>

(b)

prices
{<price>65.95</price>, <price>69.96</price>}
{<price>34.95</price>, <price>39.95</price>}

(c)

Figure 7: Examples of Intermediate XAT Tables for XQuery in Figure 3.

Figure 7 depicts three examples of XAT tables. Figure 7(a) depicts a regular relational table. Figure 7(b) is a table of XML nodes. Figure 7(c) is a table of collections. We use $\{..., ...\}$ to denote a collection.

2.2 Comparison and Node Identity

Comparison in the XAT data model is done by values, e.g., the deep equal comparison as in the object-relational data model. A more efficient comparison can be done by node identity. If the comparison includes an atomic value on one side of the equation, then it can only be done by value comparison. The comparison

between a collection with another collection is done by comparison of each pair of items, and in the case of ordered sequences, only in the order of the sequences.

2.3 Document Order and Sequence Order

Sequence order refers to the order of the items within a given sequence. If a sequence is composed of sibling items, e.g., items with a common parent, then we say the sequence is ordered in sibling order.

Document order refers to the total order among all nodes within a given document. It is defined as the pre-order depth-first tree traversal order of all the nodes in the XML tree modeling the XML document.

2.4 XAT Schemas and Types

Rainbow's algebra makes use of the schemas and types defined in the W3C XML Query Algebra [19]. The DTD in Figure 6 can be represented as a schema in Figure 8. As we can see Figure 8 defined two types, i.e., the type $Pricelist$ and the type $Book$. The definition of the type $Pricelist$ says that it contains a `pricelist` tag and zero or more instances of type $Book$. The definition of the type $Book$ says that it contains a `book` tag, and within the `book` tag, there are three other tags, i.e., `title` and `price` . For the value of each tag has their terminal types, e.g., `String` and `Float` .

type Pricelist	= pricelist [Book {0, *}]
type Book	= book [title [String] , price [Float]]

Figure 8: Algebra Schema representing DTD in Figure 6.

In Figure 7(a), the type of the column "title" is $String$ and the type of column "price" is $Float$. In Figure 7(b), the type of column "price" is $price[Float]$. In Figure 7(c), the type of column "prices" is $(price[Float])\{0, *\}$.

3 Rainbow Framework for Mapping and Query Processing

Rainbow has been designed to exploit relational database technology to manage XML data based on a flexible mapping strategy [23]. We now will briefly describe the general concept of XML to relational mapping and XML query processing based on our system. The Rainbow system in Figuref 9 is composed of three sub-systems: a loading manager, an extraction manager, and an XML query engine.

XML and Relational Data Mapping Management. Rainbow supports flexible mapping between XML data and relational data by a mapping engine. This mapping engine [4] supports both directions of the

mapping by providing two managers, one responsible for loading and the other responsible for extraction. The loading manager can choose between multiple loading strategies to load XML data into a relational database. The mapping engine maintains an extraction view query for each loading query, i.e., they are kept as mapping pairs. Hence, the extraction manager simply selects the proper extraction view query for constructing the virtual XML document view over the relational data. This view query specifies the logic of the reconstruction of the original XML document. The system can flexibly switch between several different mapping strategies [4] by changing the XQuery mapping expressions.

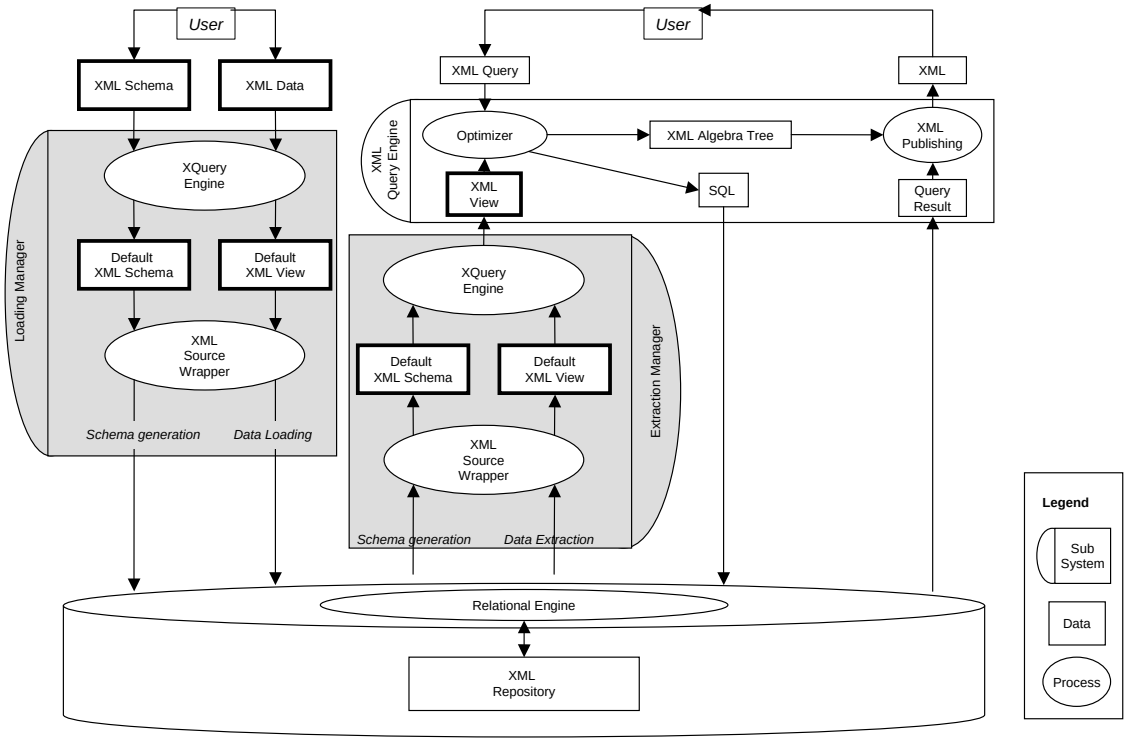


Figure 9: The Rainbow System.

Query Processing of XQuery Expressions over Relational Engines. Rainbow's query engine uses the XML algebra, called the XML Algebra Tree (XAT), explained in the previous section, for optimization and execution of queries. Similar to XPERANTO [3], Rainbow employs SQL pushdown strategies, based on algebraic equivalence rules, to extract maximal SQL expressions from the given XQuery expression. The query executor currently implements the remaining XAT operators not expressible in SQL in a native execution.

Figure 10 depicts the *Rainbow Query Engine*. Rainbow uses the *Kweelt Parser* [15] 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. See Figure 11 for the two respective XATs for our running example.

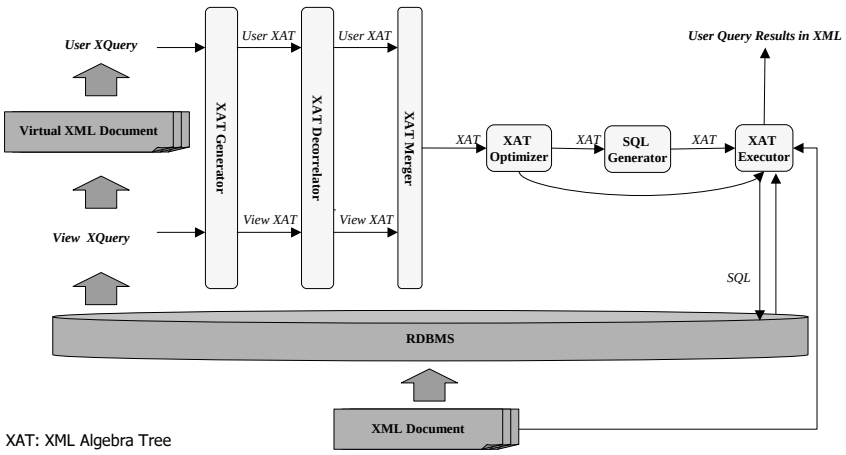


Figure 10: Architecture of Rainbow Query Engine.

The generated XATs are then unnested by the *XAT Decorrelator*. Query decorrelation is used to improve the query performance by removing the repeated evaluation of the inner query. Unnesting also often results in additional opportunities for the optimization. The decorrelation is done by creating an intermediate query space of joining the inner and outer queries together [16]. Figure 12 displays the two decorrelated XATs of our running example.

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. The query merging will match the source node(s) of the user's query with the expose node of view's query. The source and expose operators will be reduced into one rename operator that matches the two column names. Figure 13 displays the merged XAT.

The *XAT Optimizer* is composed of an *XAT Rewriter* and an *XAT Cleaner*. The *XAT Rewriter* rewrites the merged XAT so as to push as much SQL-do-able computation down to the bottom of the tree. The rules used are explained in detail in Section 4, while the overall rewriting process is outlined in Section 5. *XAT Cleaner* eliminates unused columns in the XAT tables and detects and removes unused operators. This optimized query tree is shown in Figure 14.

SQL Generator generates SQL from the bottom portions of XAT (see Figure 14). This SQL 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 the rest of this paper, we discuss the optimization of the XAT using the *XAT Rewriter* and *XAT Cleaner* in more detail.

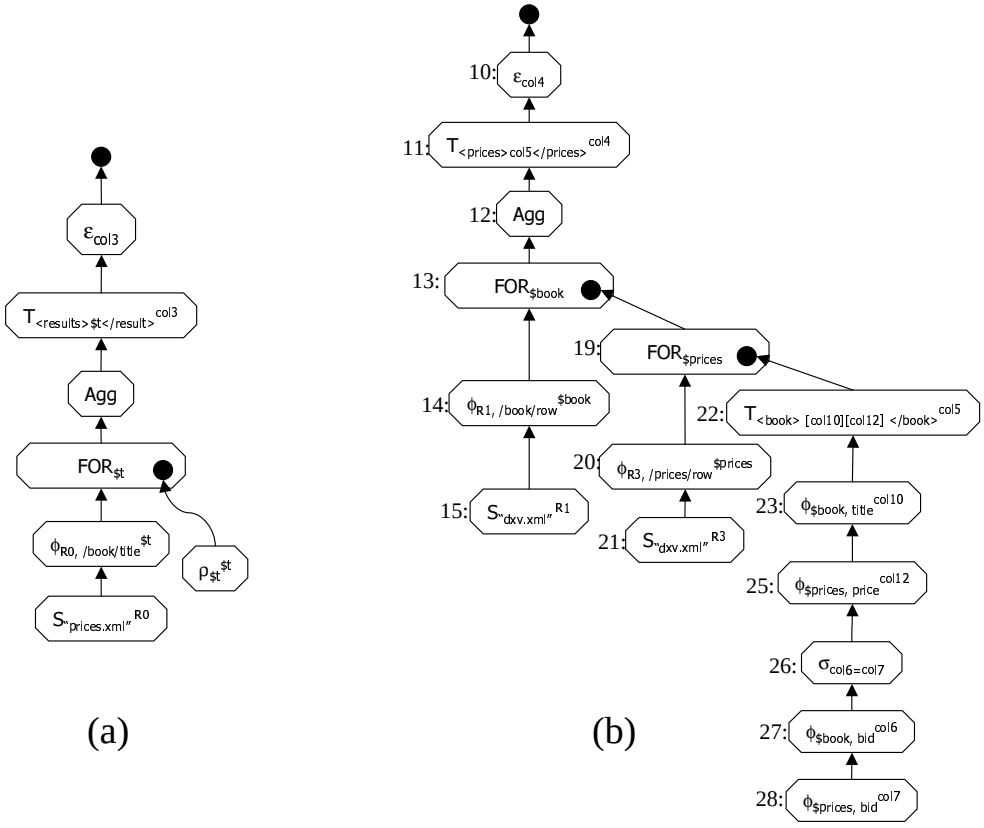


Figure 11: (a) User XAT before Decorrelation, and (b) View XAT before Decorrelation.

4 Rewriting Rules

4.1 Navigate Pushdown Rules

This section details some of the most commonly used rewrite rules [1] in Rainbow, specifically the rules that govern how a Navigate (Unnest or Collection) may be swapped with another XAT operator.

Navigate vs. Single Source Operator. The single source operator (op) includes Distinct (δ), Function ($F()$), NameColumn (ρ), Navigate (ϕ/Φ), Orderby (Σ), and Tagger (T). Column z is the output column name of that single source operator op. Intuitively, the Navigate may be swapped if the output of the op does not serve as input to the Navigate. This is demonstrated by Equation 1. The output column of the Navigate, which is y, is unaffected by this rule.

$$\Phi^{y}_{x,path}(op^{z}) \rightarrow op^{z}(\Phi^{y}_{x,path}) \tag{1}$$

The rule (Equation 2) 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 in its list of

[1] Please note in the rest of the paper, we will use the terms rules and equations interchangeably to avoid repetition.

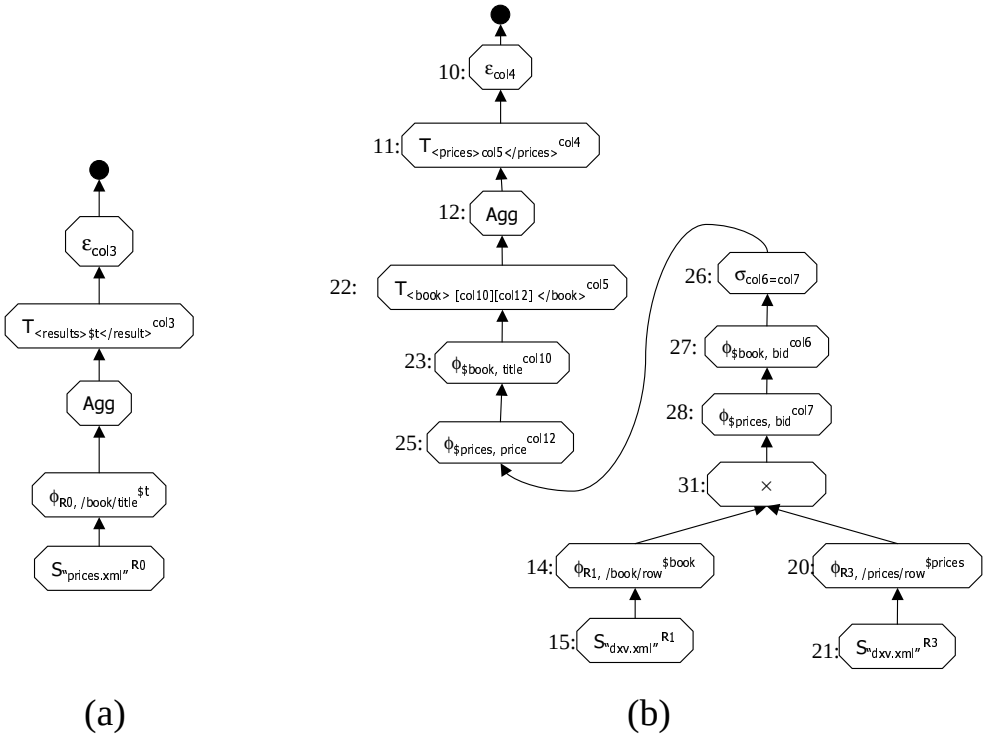


Figure 12: (a) User (b) View XAT after Decorrelation.

output columns. This is analogous to the treatment of Select and Project rewrites in relational algebra.

$$\Phi^{y}_{x,path}(\pi_z) \rightarrow \pi_{z,x}(\Phi_{x,path}) \tag{2}$$

Navigate vs. Multi-Source Operator. The multi-source operators include Cartesian Product, Join ($\bowtie$), merge (M), and set operators. Below we simply refer to them by op.

$$\begin{aligned} \Phi^{y}_{x,path}(op_p(ls, rs)) &\rightarrow op_p(\Phi^{y}_{x,path}(ls), rs) \\ \Phi^{y}_{x,path}(op_p(ls, rs)) &\rightarrow op_p(ls, \Phi^{y}_{x,path}(rs)) \end{aligned} \tag{3}$$

The navigate operator will be pushed down to the branch that contains the column it is navigating into.

Navigate vs. Navigate. Equation 4 states that a Navigate operator can be swapped with another Navigate unless there is a dependency between them. A dependency exists between two operators if the output of one operator serves as the input to another and vice versa.

$$\phi^{x2}_{x1,path1}(\phi^{y2}_{y1,path2}) \leftrightarrow \phi^{y2}_{y1,path2}(\phi^{x2}_{x1,path1}) \; [if \; x1 \neq y2 \; and \; x2 \neq y1.] \tag{4}$$

Navigate vs. Selection. 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 is captured by Equation 5.

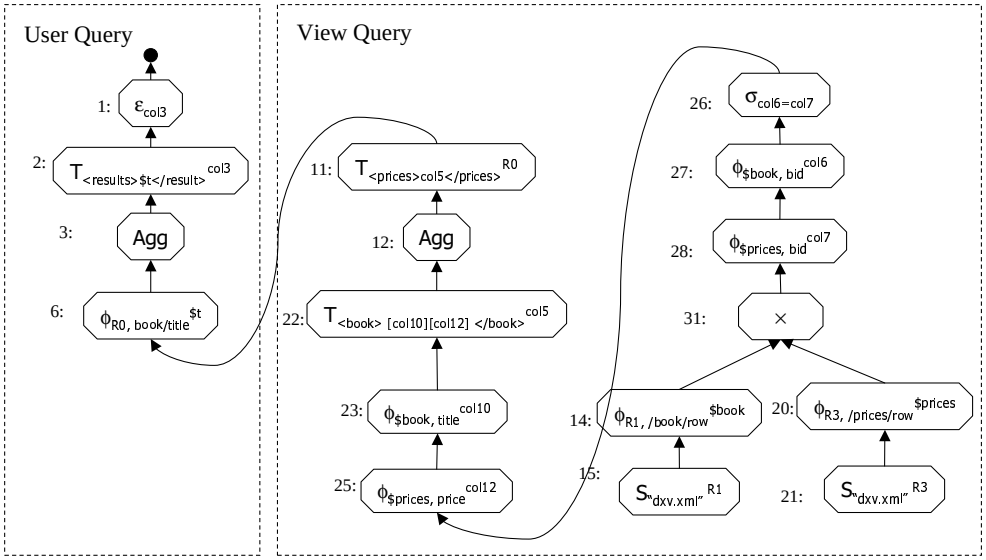


Figure 13: XAT after Merging

$$\begin{aligned} &\phi^{x2}_{x1,path1}(\sigma_c) \rightarrow \sigma_c(\phi^{x2}_{x1,path1}) \\ &\sigma_c(\phi^{x2}_{x1,path1}) \rightarrow \phi^{x2}_{x1,path1}(\sigma_c)\ [if\ x2\ not\ in\ c.] \end{aligned} \tag{5}$$

Navigate vs. Aggregate. Navigate unnest (ϕ) will navigate into a collection and turn each individual value into one separate tuple. In that case, if there is an aggregation operator before the navigate-unnest operator, the collection created by the aggregate operator will be decomposed into individual values. But if the column $x1$ is not a collection before the aggregate operator, then the aggregate operator will change the structure of the column $x1$. Hence Equation 6 only holds when the column $x1$ is only consumed (used) in the navigate operator.

$$\phi^{x2}_{x1,path}(Agg()) \rightarrow \phi^{x2}_{x1,path} \tag{6}$$

Equation 7 shows the navigate collection (Φ) can be pushed through the aggregate operator. On the left hand side, the aggregate operator will reduce the whole input table to one tuple. The the navigate-collection (Φ) will create a new collection. Hence the final result is still one collection. On the right hand side, the navigate-collection (Φ) operator can generate multiple tuples with a collection in each tuple. After the aggregation, all the collections will be merged into one collection. Therefore, Equation 7 holds.

$$\Phi^{x2}_{x1,path}(Agg()) \rightarrow Agg()(\Phi^{x2}_{x1,path}) \tag{7}$$

Navigate vs. Single Source with Internal Subquery. The single source operator with internal subqueries includes FOR (FOR), groupby (γ), and if-then-else (if) operators. Below we simply refer to any of

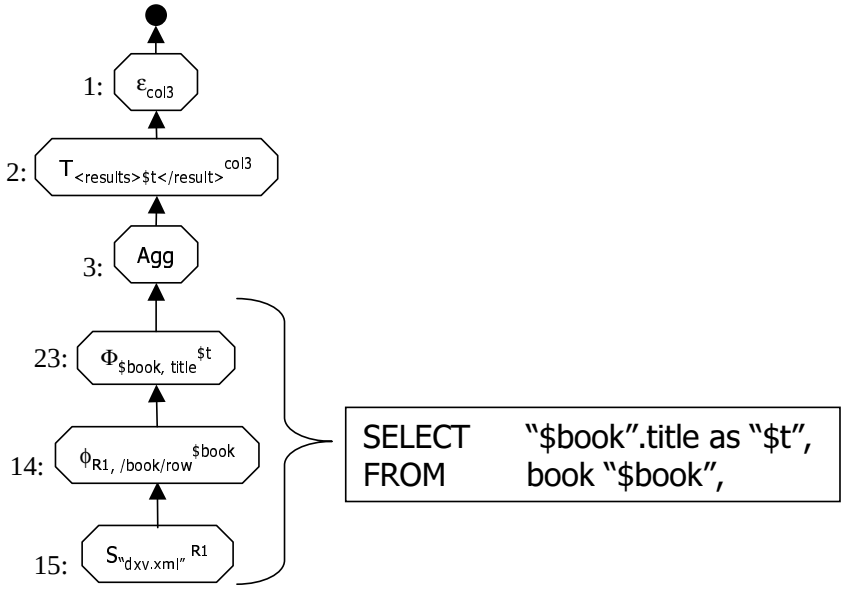


Figure 14: XAT with SQL Statement

these by op.

$$\Phi^{y}_{x,path}(op_{sq}) \rightarrow op_{sq}(\Phi^{y}_{x,path}(sq)) \tag{8}$$

4.2 Navigation Merging

There are two types of navigation operators, i.e., Navigate Unnest (ϕ) and Navigate Collection (Φ). The Navigate Unnest (ϕ) will unnest the navigated results into multiple tuples, while the Navigate Collection (Φ) will keep the result as one collection. The four rules given below dictate how to merge different navigations.

Let's use the following example to illustrate the intuition of Rules 9 to 12. Assume we have element a, which has two sub-elements $b1$ and $b2$, and element $b1$ that has sub-elements $c1$, $c2$ and $c3$, and element $b2$ that has sub-elements $c4$, $c5$, and $c6$. The tag for $b1$ and $b2$ is b, and tag for $c1$ to $c6$ is c.

Figure 15 illustrates the intuition of Equation 9. As we can see the column $/b/c$ on the left hand side is the same as the column $/b/c$ on the right. Both are individual values in multiple tuples. Figure 16 illustrates the intuition of Equation 10. As we can see the column $/b/c$ on the left is also the same as the column $/b/c$ on the right. Figure 17 illustrates the intuition of Equation 11. As we can see the column $/b/c$ on the left side is the same as the column $/b/c$ on the right side. Both are of type collection. Figure 18 illustrates the intuition of Equation 12. The column $/b/c$ is equivalent neither to the column $/b/c$ on the right side of Equation 9 as separate values nor to the column $/b/c$ on the right side of Equation 11 as a collection.

Assumption 1 *For the following four rules, we assume the* $col1$ *is only referenced by the two involved navigate operators.*

$$\phi^{col2}_{col1,path1}(\phi^{col1}_{col0,path2}) \leftrightarrow \phi^{col2}_{col0,path2/path1} \tag{9}$$

$$\phi^{col2}_{col1,path1}(\Phi^{col1}_{col0,path2}) \leftrightarrow \phi^{col2}_{col0,path2/path1} \tag{10}$$

Please notice Equation 10 has the same result as Equation 9. Therefore, there are two alternatives for the decomposition of a Navigate Unnest.

$$\Phi^{col2}_{col1,path1}(\Phi^{col1}_{col0,path2}) \leftrightarrow \Phi^{col2}_{col0,path2/path1} \tag{11}$$

$$\Phi^{col2}_{col1,path1}(\phi^{col1}_{col0,path2}) \rightarrow N/A \tag{12}$$

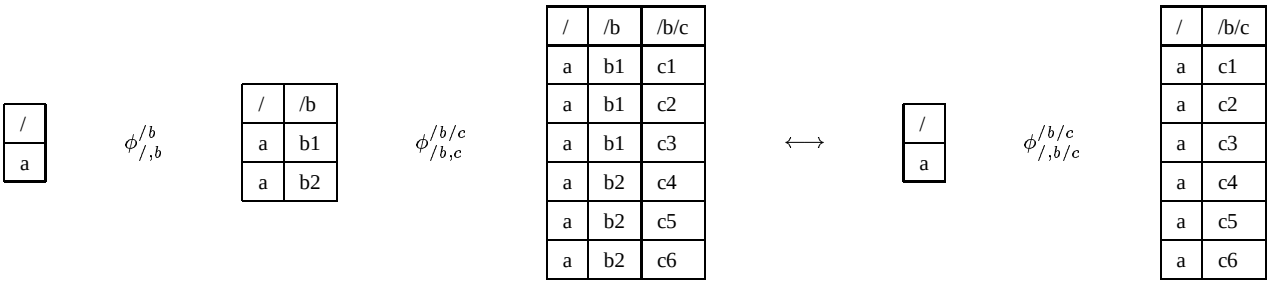


Figure 15: Example for Rule 9

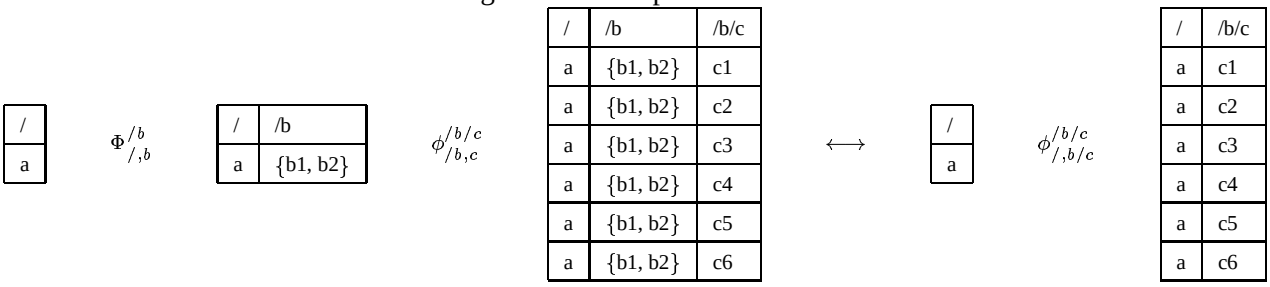


Figure 16: Example for Equation 10.

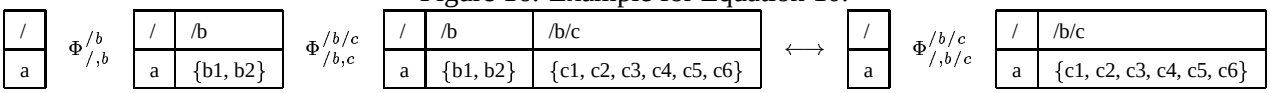


Figure 17: Example for Equation 11.

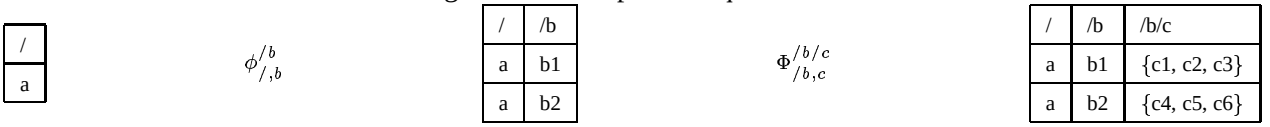


Figure 18: Example for Equation 12.

4.3 Operator Cancel Out

Due to the generality of the XQuery to XAT translation process, there may be redundant operators in the XAT Tree. Therefore, the following rules (Equations 13 to 15) are used to eliminate these redundancies. For example, the Navigate operator can be canceled with the Tagger operator, but not vice versa.

$$\phi^{y}_{x,step}(T^{x}_{<root><step>[z]</step></root>}) \rightarrow \begin{cases} T^{y}_{<step>[z]</step>} & [if\ x\ is\ not\ used\ anywhere\ else.] \\ T^{x}_{<root>[y]</root>}(T^{y}_{<step>[z]</step>}) & [if\ x\ is\ used\ somewhere\ else.] \end{cases} \quad (13)$$

$$\phi^{y}_{x,/text()}(T^{x}_{<step>[z]</step>}) \rightarrow \begin{cases} \rho^{y}_{z} & [if\ x\ is\ not\ used\ anywhere\ else.] \\ T^{x}_{<step>[y]</step>}(\rho^{y}_{z}) & [if\ x\ is\ used.] \end{cases} \quad (14)$$

As the XAT tree is generated from the parsed tree, we want to be able to keep track of how each column in the XAT table was created. To accomplish this, a binding table is created to describe the origins of each column for the operators in the entire query plan. Some examples of origin are given next. For example, 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 a function, it will keep the function name and its parameters. Making use of this binding table allows us to rewrite Equation 13 into the rule given in Equation 15.

$$\phi^{y}_{x,step}(T^{x}_{<root>[z]</root>}) \rightarrow T^{x}_{<root>[y]</root>}(\rho^{y}_{z})\ [if\ step\ is\ used\ in\ binding\ of\ z]. \quad (15)$$

Figure 19 depicts the variable dependency graph represented by the binding table (Figure 20) for XAT tree in Figure 13. The dependency graph and the binding table contain the same information. The boxes in the graph represent column names. The solid arrows mean that one column name was derived from another. The text on that edge details how it was derived from the other. For example, col3 was by adding the $< result >$ tag around the contents of column $t. A dotted line means that one column is inferred to be equivalent to another. Let us examine columns $t and col10. Column $t represents col0/title, while col0 represents book/R0 where book is the tag generated by col5. If this trace continues, it becomes apparent that these two represent the same columns. Equation 15 will find and eliminate such redundancy.

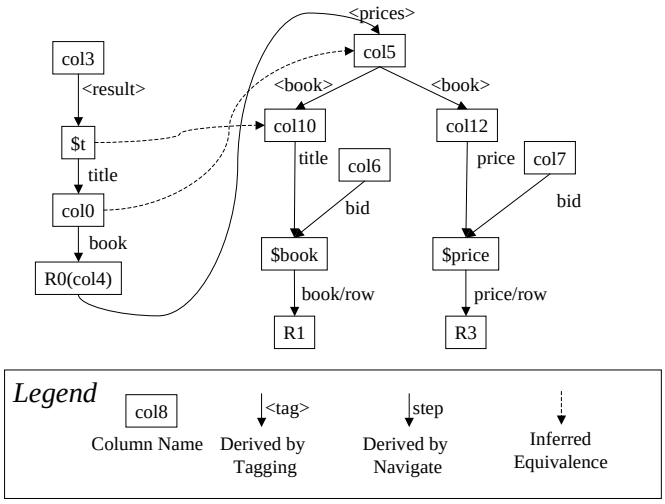


Figure 19: Variable Dependency Graph for XAT in Figure 13.

Binding	Origin	Path/Tagger
col3	col2	$< result >$
$t	col0	title
col0	book	R0
R0	col4	[EMPTY]
col4	col5	$< prices >$
col5	col10, col12	$< book >$
col10	$book	title
$book	R1	book/row
col12	$price	price
$price	R3	price/row
col6	$book	bid
col7	$price	bid
R1	data source	[EMPTY]
R3	data source	[EMPTY]

Figure 20: Binding Table.

4.4 Rename Rewriting Rules

Any time a NameColumn operator is pushed down 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.

$$\rho_{x,y}(< op >^x) \rightarrow < op >^y \tag{16}$$

$$< op >_y (\rho_{x,y}) \rightarrow < op >_y \tag{17}$$

4.5 Merge/Split Tagger Operator Rules

Finally, one Tagger operator may be split into two separate Tagger operators. The converse is also true, that is, two Taggers can be merged into one.

$$T^x_{p1}(T^y_{p2}) \leftrightarrow T^x_{p1.p2} \texttt{(If } y \texttt{ is not used besides in pattern } p1\texttt{)} \tag{18}$$

4.6 Operator Transformation Rules

After query generation, a navigation-unnest can be generated with a group-by operator with aggregate() function as its parent. This combination of three operators is equivalent to a navigation-collection operator. Following rule captures this transformation:

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

4.7 Merge/Split Select Operator Rules

These rules are general rules not specific to Rainbow's algebra. These rules are relate to merging and splitting Select operators. They are simply applications of basic commutative and associative rules.

$$\sigma_{c1}(\sigma_{c2}) \rightarrow \sigma_{c1 \; AND \; c2} \tag{20}$$

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

$$\sigma_{c1 \; OR \; c2} \rightarrow \sigma_{c1} \cup \sigma_{c2} \tag{22}$$

Other relational traditional rewriting rules [9] also applies in our algebra.

4.8 Orderby Operator and Position Function Operator Rules

The transformation rules specificly for order-sensitive query handling is out of the scope of this report, and hence described in the Master Thesis [12].

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. Figure 13 shows the XAT of our running example after decorrelation, but still before the rewriting process has been applied to this. We will now use this running example to explain the main heuristics of rewriting in a step by step fashion.

5.1 XAT Traversing

The XAT Traverser uses a bottom up approach to find an operator to rewrite. The bottom up approach is chosen 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. Thus the operators may be stopped

prematurely from being moved maximally down the tree. The Traverser iterates over the tree and applies the equivalence rules from Section 4 according to one of several of our heuristics as further explained in a later section (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 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.

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 13 notice that the Navigate operators #23, #25, #27, and #28 are in arbitrary order. Navigate #25 and #28 use $prices, and #23 and #27 use $book. By evaluting rules 4 and 5, in Section 4 we can better organize the Navigates. The tree after applying these rules can be found in 21. Notice that all the Navigates that use $price and $book are all organized in their branches.

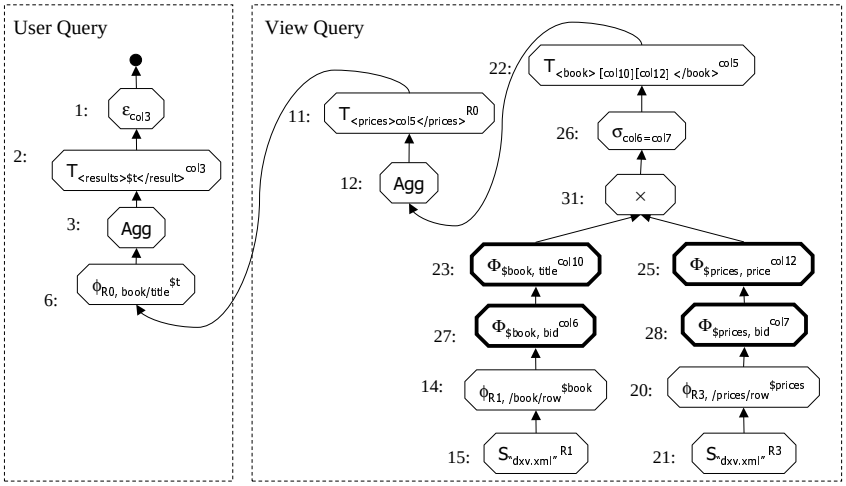


Figure 21: XAT after Navigation Pushdown.

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 makes use of Equations 13 and 14.

This heuristic is best illustrated by an example. In Figure 13, operator #23 navigates from *$book* to `title` and binds that output bag to col10. Then, operator #21 creates a `book` element out of col10, col11 and col12 (all 3 columns are elements). Using the equivalence rules for operator #23 and its child listed above, operator #23 is pushed down. Figure 23 shows the final position of operator #23 in the XAT. We then iterate over the tree looking for another Navigate to push down. After rewriting operator #6, the Traverser next chooses operator #5. This operator #5 navigates from col0 (which represents R0/book) to `title` and binds the result to $t. Operator #5 is equivalent to operator #23. The rewriter determines this by evaluating equivalence rules for #5 and its children, including for #5 and #22. The rule says that the Navigate #5 unnests the `title` element because its destination is `title`. The rule then deletes the `title` element from the Tagger operator and then deletes #5 and replaces it with a NameColumn operator. This NameColumn operator #34, not shown in the final tree due to space reasons, renames col10 into $t. The traverser then searches for another Navigate operator to push down until none can be found.

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 by the first heuristic 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 22 shows an example of a query with duplicate Navigates. The reader may notice 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 22: XQuery Example with Duplicated Navigations.

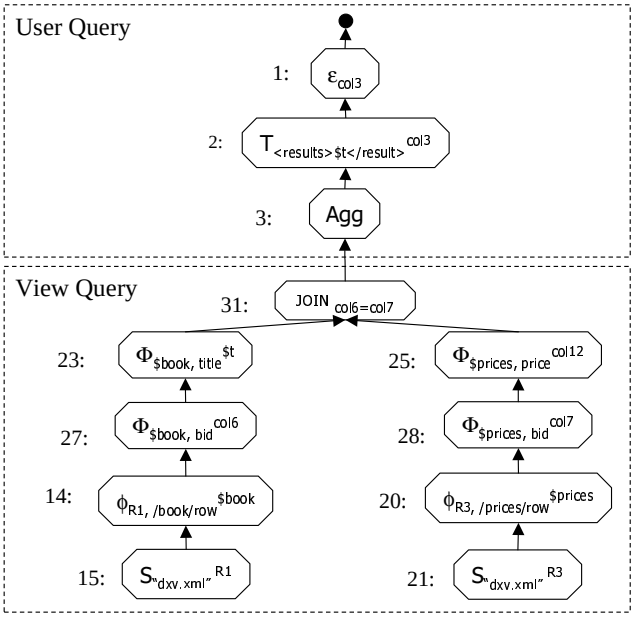


Figure 23: XAT Rewriting after is complete.

Computation Pushdown: Relational engine SQL operators such as Distinct δ, Selectσ, GroupByΓ, Join$\bowtie$, and set operators can be pushed down from the user portion of the query tree into the mapping portion by repeatedly evaluating equivalence rules. In the running example, the SQL computation that is pushed down is the Join operator (operator #31). Note that although the Navigate operators do not have an equivalent in relational algebra. This heuristic uses Equations 1 to 8. Navigates, depending on their context, can be mapped into either SELECT and FROM clauses (in SQL). Therefore, they are pushed as far down in the tree as possible.

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 Section 4.4 that will rename each operator appropriately. Notice the binding change in operator #23 in Figure 23. The NameColumn operators are deleted when they reach the bottom of the tree.

6 XQuery Optimization by 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 algebra tree, we can compute the columns actually required for each operator. Table 1 describes the *consumed*, *modified*, and *produced* columns for the operators. *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 Navigate-Collection operator will *Produce* an output column that is the result of executing the path over the document. *Modified* denotes whether an operator can affect other columns.

Operator	Consumed	Produced	Mod.
$T_p^{col}(S)$	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
$F_{prm[1..m]}^{col}(S?)$	columns in prm[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

Operator	Consumed	Produced	Mod.
$\pi_{col[1..n]}(S)$	$col[1..n]$	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
$\overset{\circ}{\bowtie}_{L\,c}(ls, rs)$	columns in condition c	N/A	Yes
$\overset{\circ}{\bowtie}_{R\,c}(ls, rs)$	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 1: Column Consumed, Produced, and Modified for Each Operator.

Figure 24 shows the schema for the example in Figure 23. Column *node* stores the node number in Figure 23. Column *parent* stores the parent of this operator. Column *produce* and *consumed* store the columns

produced and consumed by this operator as defined in Table 1, respectively. By default, every operator will simply append new columns to the end of the XAT tables. Figure 24 gives the original schema used by the XAT evaluation in column *Produced by Descendant*. In Figure 24, the nodes #2 and #3 have more than eight columns. On the other hand, the root node will only return one column.

As we can see, 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.

Node	Parent	Produced	Consumed	Produced by Descendant	Minimum Schema
1		{}	{col3}	{col3}	{col3}
2	1	{col3}	{$t}	{col3, R1, $book, col6, $t, R3, $prices, col7, col12}	{col3}
3	2	{}	{}	{R1, $book, col6, $t, R3, $prices, col7, col12}	{$t}
31	3	{}	{col6, col7}	{R1, $book, col6, $t, R3, $prices, col7, col12}	{$t}
23	31	{$t}	{$book}	{R1, $book, col6, $t}	{col6, $t}
27	23	{col6}	{$book}	{R1, $book, col6}	{$book, col6}
14	27	{$book}	{R1}	{R1, $book}	{$book}
15	14	{R1}	{}	{R1}	{R1}
25	31	{col12}	{$prices}	{R3, $prices, col7, col12}	{col7, col12}
28	25	{col7}	{$prices}	{R3, $prices, col7}	{$prices, col7}
20	28	{$prices}	{R3}	{R3, $prices}	{$prices}
21	20	{R3}	{}	{R3}	{R3}

Figure 24: Original Schema before Schema Cleanup.

Intuitively, the minimal schema should include all columns, which are produced by this operator or its descendants and consumed by the operator's ancestors, but exclude all the columns produced by the operator's ancestors. Hence, we have the following formula to define the minimal schema of an operator as:

$$MS := P_{descendant} \cap (C_{ancestor} - P_{ancestor}) \cup P \tag{23}$$

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, by the operator's ancestors. $P_{descendant}$ denotes the columns produced by the operator's descendants.

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

$$\begin{aligned} MS &:= P(\texttt{for root operator}) \\ MS &:= P_{descendant} \cap (MS_p - P_p) \cup C_p \cup P \end{aligned} \tag{24}$$

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.

Figure 24 shows the schema for the example in Figure 23, using the formula depicted above. The column *minimum schema* stores the computed minimum schema of this operator. Please notice the node #23 that has a binary operator parent node #31, so the minimum schema of node #23 does not include the column $col7$, 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 we call 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 cuttable operator.

#	Parent()	col3	$t	col6	col7	$book	R1	col12	$prices	R3	Cut?
1		R									
2	1	P	C								
3	2	-	M	-	-	-	-	-	-	-	
31*	3			C	C						
23	31		P			C					
27	23			P		C					
14	27					P	C				
15	14						P				
25	31							P	C		
28	25				P				C		
20	28								P	C	
21	20									P	

*We assume Join didn't modify $t. Otherwise, only node 25 will be deleted.

Figure 25: Matrix for Nodes Cleanup.

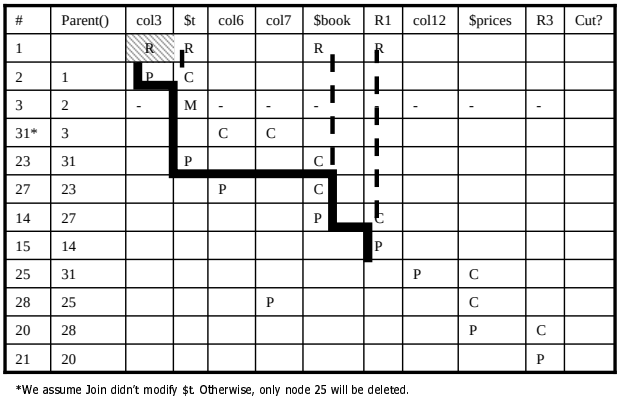


Figure 26: Analysis of Required Columns.

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 cell at location $(1, col3)$ in Figure 25 contains an r, meaning $col3$ is a required column for operator $\#1$.

Before we discuss the rules, we 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 25, *Parent(1,2)* is true. *Ancestor(node1, node2)* is defined recursively as follows:

```
Ancestor(A, B) if Parent(A, B)
Ancestor(A, C) if Parent(A, B) and Ancestor(B, C)
```
(25)

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* to show it is modified. For example, the cell $(3, \$t)$ in Figure 25 means that the operator #3 modified $\$t$, because $\$t$ was produced by operator $\#23$ ($\#3$'s descendant) and consumed by operator $\#2$ ($\#3$'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)
```
(26)

$S(A, x)$ denotes the content of operator A column x of operator in the matrix.

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 25. The rules to compute r (means *required*) 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)
```
(27)

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 $S(2, \$t) = c$ and $S(2, col3) = p$ and $S(1, col3) = r$ and operator #1 is operator #2's parent, the column $\$t$ is required. For the same reason, columns $book$ and $R1$ 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 cuttable 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
```
(28)

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 27 for the XAT depicted in Figure 23. As we can see, the operators #31, #27, #25, #28, #20, and #21 are deemed unnecessary (by our formula) and can safely be removed from the tree. For example, the operator #25 produced columns $col12$ but $col12$ is never used. Hence the operator #25 can be cut. After the cutting, the XAT is simplified as shown in Figure 28 (b). Let's compare the SQL statements generated for these two trees. Figure 28 (c) depicts the SQL statement corresponding to the original XAT in Figure 28 (a). Figure 28 (d) depicts the SQL statement corresponding

#	Parent()	col3	$t	col6	col7	$book	R1	col12	$prices	R3	Cut?
1		R	R			R	R				
2	1	P	C								
3	2	-	M	-	-	-	-	-	-	-	
31*	3			C	C						X
23	31		P			C					
27	23			P		C					X
14	27					P	C				
15	14						P				
25	31							P	C		X
28	25				P				C		X
20	28								P	C	X
21	20									P	X

*We assume Join didn't modify $t. Otherwise, only node 25 will be deleted.

Figure 27: Analysis of Cuttable Nodes.

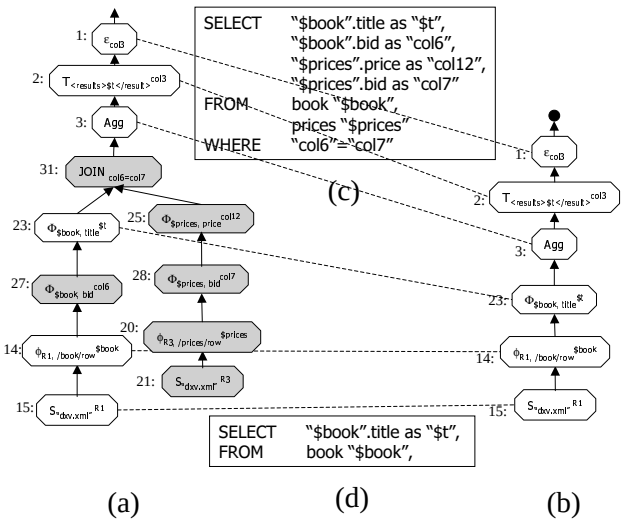


Figure 28: Cutted XAT and SQLs Generated from Each.

to reduced algebra representation in Figure 28 (b). We can see that there is a major difference between the tree and the final optimized query tree (Figure 28).

7 Preliminary Evaluation of Rainbow

We have implemented the Rainbow system in Java JDK1.2 using Xerces as DOM Parser and the Kweelt Engine [15] 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. Schema clean up, SQL translation and also query processing are also all implemented in Java.

7.1 Experimental 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 Celeron 1.2G machine with 384 MB memory under Windows 2000 Professional.

7.2 Performance Gain

7.2.1 Native Execution with Fixed Loading

In the first experiment, we compare three kinds of query execution performances. In Figure 29, the x-axis displays different data sizes in terms of the number of elements, while the y-axis shows the total execution time for the queries.

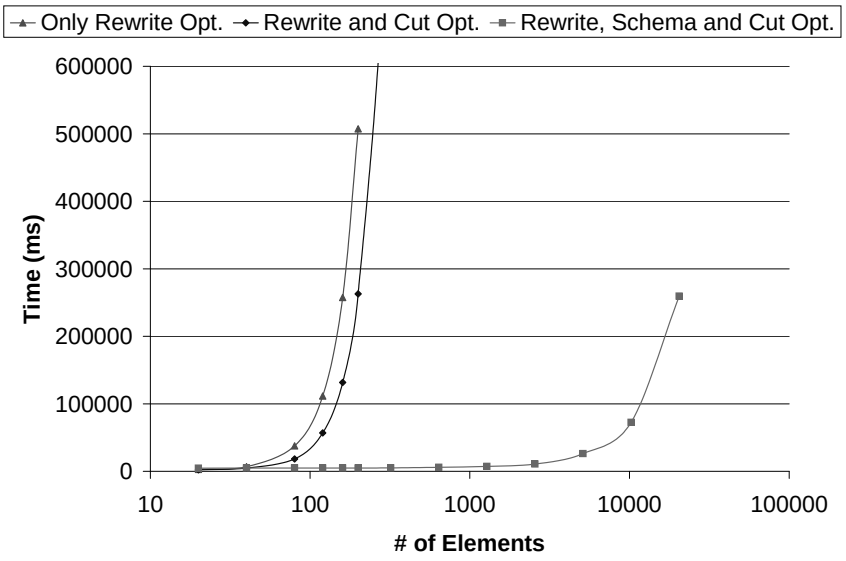


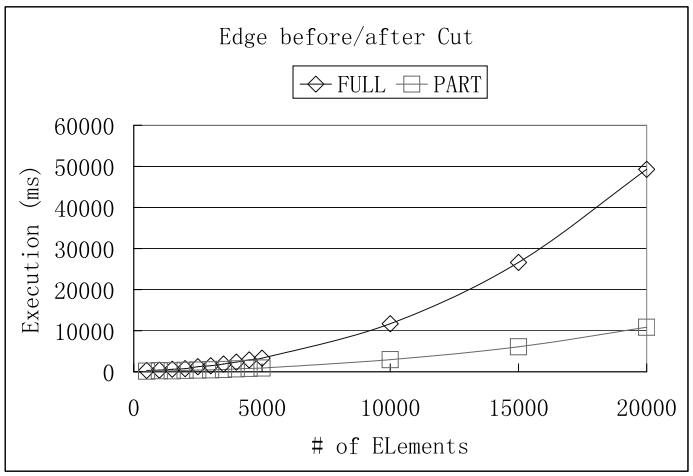
Figure 29: Performance Comparison.

The query before operator cutting and schema cleanup scales the worst. After the cutting the useless operators, the performance increased by 50 percent. The performance with schema clean and cutting scales the best, which now is capable to handle up to 20,000 elements.

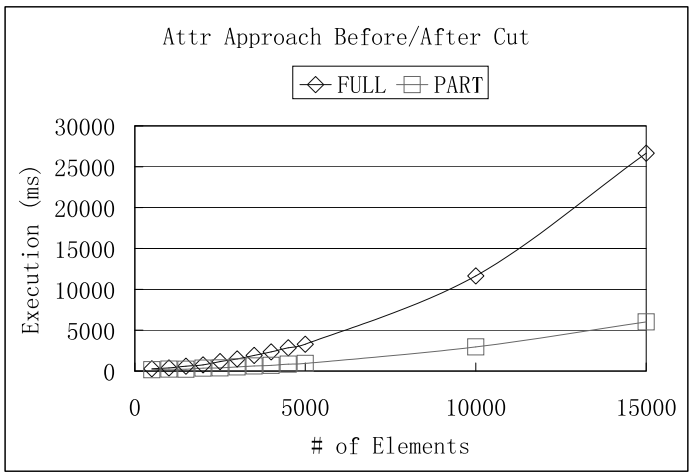
7.2.2 Computation Pushdown and Flexible Loading

In this experiment, we have explored how different mappings from XML to the relational data store can affect the performance of our query optimization strategies. We have chosen three different kinds of loading approaches, namely, edge approach [7], the attribute approach [7], and the maximally shredding approach [23].

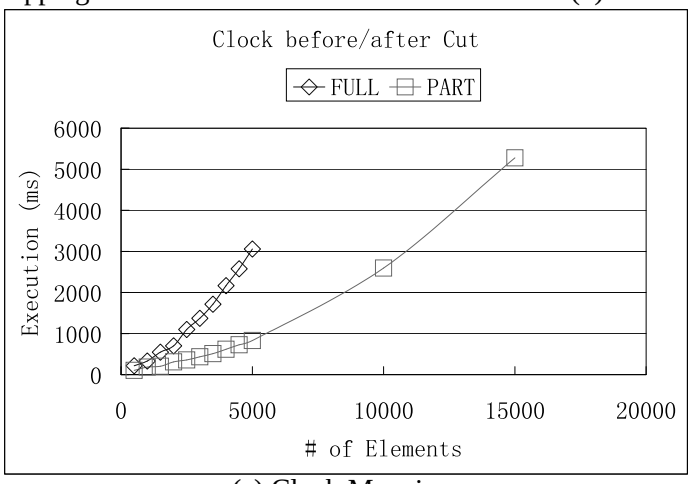
In all three charts in Figure 30, the x-axis denotes the number of XML elements that have been loaded into the relational database. The y-axis measures the execution time of the optimized XAT in terms of milliseconds. There are two lines in each chart. The line annotated by the diamond symbol denotes the performance before cutting, called FULL. The line annotated with the square symbol denotes the performance after cutting, called PART. As we can see, for all three mapping approaches after the XAT cleanup and schema cleanup, the performance improved dramatically compared to the performance before the schema cleanup. The reason for this is that in all three approaches, the particular query would have required a join operation similar to the query in our running example in this paper. For the edge mapping, this meant for example a self-join. When used with schema and operator clean up, all the unnecessary joins were removed, and hence



(a) Edge Mapping.



(b) Attribute Mapping.



(c) Clock Mapping.

Figure 30: Performance Enhancement of Multiple Mappings.

the performance increased dramatically.

7.3 Optimizer Overhead

In this experiment we study the overhead of the query optimization. Figure 31 depicts the time for different stages of the query processing. From the largest to the smallest, generation (2,834 ms), decorrelation (411 ms), rewrite (280 ms), schema cleanup (211 ms), cutting (80 ms), and merge (13 ms).

In Figure 31, we first notice the total time of processing a query before the execution can take up to 4 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. The overhead of preparing the algebra tree of merged queries composed of 44 operators can take up to 4 seconds. Hence, for large queries a prepared statement is a good next step to avoid the compile time. The schema clean up is very efficient,

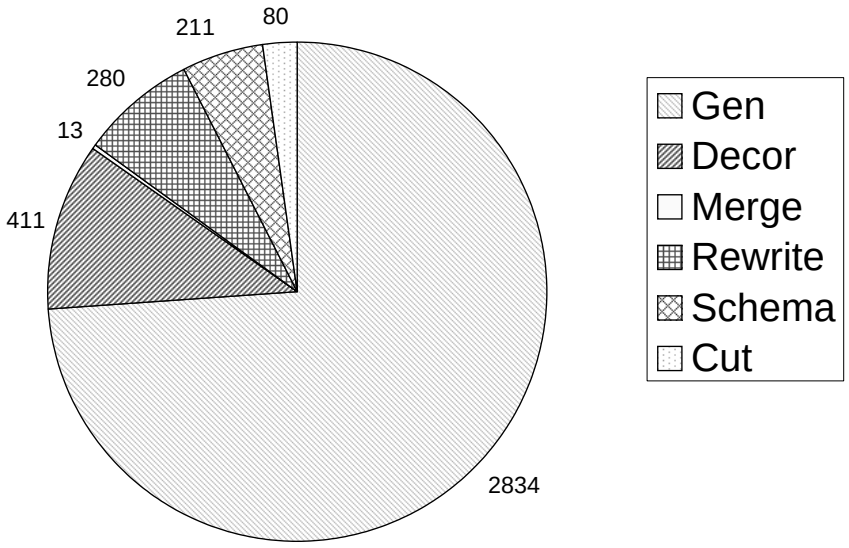


Figure 31: Overhead of Schema Cleanup.

it roughly takes 0.2 seconds. Hence, we found schema-cleanup to be an important technique in our system as at a neglectible cost for optimization it could then offer significantly improvements in query execution. Though the rewriting heuristics are complex, the rewriting time of the algebra tree in our system still was significantly shorter than the total generation time.

Currently, our Rainbow prototype does not contain any special code optimizations nor any sophisticated memory manager to carefully manage the Java objects. Instead, we rely on JAVA's object management and garbage collection facilities, which at times is of course not in tune with high-performance query processing. Our main goal has been to explore strategies for XAT optimization and to test them out, which we believe has been done successfully. Now, further effort can go into making the query engine robust and scalable.

8 Related Work

8.1 XML Mapping

To be useful, any XML data management system including XPERANTO [3] or Rainbow must provide some convenient way for XML data to be loaded into the database. The proposed solution in XPERANTO as reported in [18] is to hand-code one algorithm for every possible loading strategy such as universal, edge, inline, attribute, and the fully shredded approaches. Each such algorithm could then generate for example an XQuery expression for loading some given XML document into the database. On the other hand, Rainbow's mapping approach is to directly provide generic XQuery expression pairs, namely, a loading and an extraction expression, that by operating on XML Schema knowledge can load as well as extract any arbitrary XML

documents or XML schema without having to develop any code. Our solution (see Section 3, [4] contains more details) is thus one level more general than what XPERANTO provides for this purpose.

Bohannon et. al. [1] proposed a heuristic search strategy for finding the most cost-effective relational schema for a given query workload based on incremental schema modifications. A modified XML Schema is used to include statistical information about the XML data in the decision process of how to represent the particular XML structure with relational tables. Unlike the Rainbow mapping approach, they do not focus on how to explicitly model mappings to drive the schema generation, the data loading and query processing process

8.2 XML Query Processing

Most commercial DBMS systems also provide some level of XML querying capabilities. Oracle's XMLSQL utility [14] (XSU) issues queries against the stored XML objects using XSQL, an extension of SQL that supports XPath expressions. Oracle relies on an additional external XSLT script to transform the returned XML from the default encoding into a view desired by the user. Clearly, it thus hinges on the XSLT tool used if this is scalable, and cursor-type of access to query results would not be possible in such a scheme. This element construction task is instead directly incorporated into the core Rainbow engine due to Rainbow's support of XQuery processing. SQL Server 2000 [11] is capable of evaluating XPath expressions over XML views, but not joins or nested queries. IBM's DB2 Data Access Definition (DAD) [5] 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.

XPERANTO [3] [17] defines XML views on top of relational data using XQuery and supports user querying of those views. The user query is composed with the view query and, after composition rules are applied, the XML result is returned to the user. This approach is similar to that of the Rainbow system, but there are several key differences. First, the data model for XPERANTO, the XML Query Graph Model (XQGM), is more procedural in nature whereas Rainbow's XAT is more declarative. XQGM does not support collections and is not order-sensitive; element construction and navigation are modeled using different functions in the project operator. This is different than Rainbow's use of explicit logical operators to model those functionalities.

Query optimization in XPERANTO is also different than that of Rainbow. XPERANTO recursively applies composition rules to the XQGM that compose navigate functions in the user query with their counterparts in the view query. Select and Join predicates are then pushed down to the relational engine. In the final step, XPERANTO iterates over the tree and separates the "tagger operations ... from the SQL operations

[17].” Our system differs in that the three XPERANTO steps are combined into one computation pushdown step. This combined approach may result in a smaller number of iterations over the tree. XPERANTO also uses function-composition for cancel out in the XQGM, whereas we propose rule-based optimization for the XAT.

Silkroute [6], another research system, is similar to Rainbow. They use the same concept of composing views as Rainbow and XPERANTO, although Silkroute uses a view forest as the intermediate representation. The key difference between the two approaches is that, after separating the queries in two halves, Silkroute focuses on generating efficient SQL queries that move all computation to the relational engine. One example is the exploitation of one to one functional dependencies to reduce the number of sorted outer union queries that are generated because these are more expensive than inlining. The translation depends on a query representation that separates the structure of the output XML document from the computation that produces the document’s content. In contrast, Rainbow focuses on optimization directed at the XML algebra-level before the generation of SQL queries. The cutting algorithm is one example of this algebra-level focus of Rainbow.

The two approaches have their trade-offs: Rainbow has not focussed on generating optimized SQL statements while, by choosing the syntax level, Silkroute may miss some of the optimization opportunities that are present at the algebraic level. Furthermore, due to the generality of the translation, the SQL queries generated by Silkroute tend to have redundant predicates and joins. Some of these redundancies can be removed and the query further optimized, but [6] has showed this problem to be NP-complete.

AGORA [10] translates XQuery queries over XML views into equivalence SQL queries by rewriting at the XQuery syntax level, not using an algebra. This approach has some merit, but it seems to be difficult to optimize. Niagara [13] 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 processing of XML queries over virtual XML views of relational data. We have described our XML algebra that captures the semantics of XQuery. Using this algebra, we have proposed a two-fold optimization process at the logical XML algebra level. First, equivalence rules are applied according to some 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 XML algebra query plan that then can be converted into compact SQL queries. The proposed techniques have all been incorporated into an XML data management prototype system, called Rainbow. Preliminary experimental results conducted using the Rainbow system confirm that the query execution time was significantly improved by these optimization strategies.

References

[1] P. Bohannon, J. Freire, P. Roy, and J. Simon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE*, 2002.

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

[3] 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.

[4] S. Christ and E. A. Rundensteiner. X-Cube: A fleXible XML Mapping System Powered by XQuery. Technical Report WPI-CS-TR-02-18, Worcester Polytechnic Institute, 2002.

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

[6] M. Fernndez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. Silkroute: A framework for publishing relational data in xml. *ACM Transactions on Database Systems (TODS)*, 27(4):438–493, 2002.

[7] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.

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

[9] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems - The Complete Book*. Prentice Hall, 2002.

[10] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with xml and relational. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 623–626. Morgan Kaufmann, 2000.

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

[12] B. Murphy. Order-Sensitive XML Query Execution Over Relational Sources. Master's thesis, Worcester Polytechnic Institute, 2003.

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

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

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

[16] P. Seshadri, H. Pirahesh, and T. Y. C. Leung.

[17] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *The VLDB Journal*, pages 261–270, 2001.

[18] J. Shanmugasundaram, E. J. Shekita, J. Kiernan, R. Krishnamurthy, S. Viglas, J. F. Naughton, and I. Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30(3):20–26, 2001.

[19] W3C. The XML Query Algebra. http://www.w3.org/TR/query-algebra/, February 2001.

[20] W3C. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery/, February 2001.

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

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

[23] 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.

[24] 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.