WPI-CS-TR-02-24 July 2002

XAT: XML Algebra for the Rainbow System

by

Xin Zhang
Elke A. Rundensteiner

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

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

XAT: XML Algebra for the Rainbow System

Xin Zhang and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609
Tel.: (508) 831–5857, Fax: (508) 831–5776
xinz@cs.wpi.edu

April 25, 2003

Abstract

XML is the de facto standard for storing and exchanging data between different applications. More and more data is being published in the XML format and thus available for further querying and management. Recently, World Wide Web Consortium proposed a new standard XML query language, called XQuery. Several procedure-based query interpreter for XQuery have been proposed, most notably, Kweelt. We are developing Rainbow, an XML data management system, that at its core has an algebra based XQuery query engine. The XML algebra, which we have developed in the Rainbow project, is called XML Algebra Tree (XAT). This XAT has been designed with two primary goals namely querying both relational data and also querying native XML data. This algebra represents a solid foundation for query optimization, computation pushdown, query rewrite, different operator implementations, and other algebra related research tasks in Rainbow as well as in general.

Keywords: XML, Algebra, Optimization, Query, SQL, Rainbow.

1 Introduction

XML has become the de facto standard for exchange data over the Internet due to its flexibility. It can be used as a universal data representation model. This now raises the need for data management support to effectively handle large quantities of data that is becoming available in this format.

To manage such quantities of XML data, different query languages have been proposed for the XML querying. Thesis individual languages include, YaTL [4], Quilt [3], XPath [19], XSLT [10], etc. Recently, a new XML query language, called XQuery, adapted from Quilt [22] has been proposed by W3C.

Several XML data management systems have been proposed recently for handling large XML data sets including XML native approaches [13], Object-Oriented approaches [2, 15], Object-Relational approaches [17, 14], and pure relational approaches [27, 12, 11]. So far, they focus more on data storage instead of how to query the data are stored in the system. For example, they discuss how to shred the XML documents, and how to create relational tables, how to encode XML orders in relational tables.

More recently, some XQuery interpreters [16] and also several XML algebra proposals [1, 8] appeared. The XQuery interpreters are procedure based. Hence they are not as amenable for the optimization of such query evaluation is not as flexible as using algebra. For example, the FLWR expression will always be evaluated in the order of FOR and LET bindings, filtered by the WHERE clause, and then only at the end one would reconstruct in the RETURN clause and recursively called if any subqueries. Hence to evaluate say all predicates even those in a return clause as an earlier stage to improve the query performance is not considered. On the other hand the proposed XML algebra in the literature [20, 8] targets pure XML and thus cannot be as easily adapted to bridge between different storage models, namely, XML and relational.

XPERANTO [1] system instead has proposed many XML functions with its extended relational algebra. They proposed rewriting rules at the level of the XML functions, and also at the level of the relational algebra. In addition, for the query result reconstruction, they even have a set of separate algebra nodes. Though this hybrid algebra solves the problem between different storage models by translate between different sets of algebras, optimization and execution of such hybrid algebra are limited by differences between algebras. For example, it's very hard to push a select down through the result generation for the intermedia results because they are in two different sets of algebras. Hence, a unified general-purpose XML algebra is required.

We hence have been designing a system called Rainbow [7] for such purpose. The Rainbow system is used to manage and query XML data stored in heterogeneous systems, in particular, the relational format, and proposed related techniques to optimize XML query processing.

The core part of the Rainbow system is its flexible XML query algebra, called XML Algebra Tree (XAT) algebra. In this paper, we will describe the XAT algebra in detail. More precisely, this report describes:

- The XML data model for our algebra called XAT (XML Algebra Tree).
- XML algebra operators in detail with respective examples.
- Generation of the XML algebra tree (XAT) from the XQuery step by step.

As we can see, not only can the Rainbow system benefit from this general-purpose XML algebra, but it can be also used in the application of adaptive query processing, continuous query processing, system integration, data warehousing and other related areas when done in the context of the XQuery language.

Outline In the next section, we briefly review needed background knowledge of XML, DTD and XQuery, mostly in the form of a running example. Section 3 defines the data model used in XAT. An overview of the XAT operators is given in Section 4. The XML operators, special operators, and SQL operators are described in detail in Sections 5, 6, and 7 respectively. Section 8 describes the XAT generation process. We compare our work wither other related work in Section 9. We conclude this paper in Section 10.

2 Background

2.1 XML and DTD

The following DTD and XML are adapted from XML Use Cases [5]. The DTD (Figure 1) describes a price list of multiple books. Each book has one title, one publication source, and one price. An example XML document compliant to this DTD is given in Figure 2.

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

Figure 1: DTD Example.

```
<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 2: XML Data Example.

2.2 XQuery

The running example query is listed in Figure 3. The query computes the minimum price for each book in the price list, and returns the book's title with its minimum price. The query result is shown in Figure 4.

```
<result>
{
  for $t in
    distinct(document("prices.xml")/book/title)
  let $p :=
    document(``prices.xml'')/book[title=$t]/price
  return
    <minprice title=$t/text()>
      <price> min($p/text()) </price>
    </miprice>
}
</result>
```

Figure 3: XQuery Example.

```
<results>
  <minprice title=''TCP/IP Illustrated''>
    <price>65.95</price>
  </minprice>
  <minprice title=''Data on the Web''>
    </price>34.95</price>
  </minprice>
</results>
```

Figure 4: Query Result of XQuery in Figure 3

3 XAT Data Model

3.1 XAT Data Model Components

XAT data model is an order-sensitive table called XAT table. Inspired by W3C's XQuery 1.0 Formal Semantics [24], 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 [21]. This means that $t[col_n] \in XQueryDomainType$, where T is an XAT table and $t[col_n]$ is the value of t for column col_n.

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

Collection and Sequence. A collection is an unordered bag of zero or more items. Sequence is an ordered collection. 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 also 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 5: Examples of Intermediate XAT Tables for XQuery in Figure 3.

Figure 5 depicts three examples of the XAT tables. Figure 5(a) is a normal relational table. Figure 5(b) is a table of XML nodes. Figure 5(c) is a table of collections. We use {..., ...} to denotes a collection.

3.2 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 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.

3.3 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 from same parent node, then we call the sequence order the 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 in XML document.

3.4 XAT Schemas and Types

XAT uses the schemas and types defined in the W3C XML Query Algebra [21]. We briefly repeat the abstract syntax in Table 1. For example, the DTD in Figure 1 can be represented in the schema in Figure 6. As we can see the Figure 6 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 is a three other tags, i.e., `title` , `source` , and `price` ; and also for the value of each tag has their terminal types, e.g., `String` and `Float` .

type variable	y		
type	t	::= y	type variable
		| ()	empty sequence
		| ?	empty choice
		| Wild	wildcard type
		| u	unit type
		| t1 , t2	sequence, t1 followed by t2
		| t1 & t2	interleaved product, nodes in t1 interleaved with nodes in t2
		| t1 | t2	choice, t1 or t2
		| t {m, n}	repetition of t between m and n times
		| {t}	unordered forest of nodes in t
		| pic t	t interleaved with comments and processing instructions
		| mixed t	t interleaved with strings
bound	m, n	::= natural number or *	
unit type	u	::= p	atomic datatype
		| @Wild[t]	attribute with name in Wild and content in t
		| Wild[t]	element with name in Wild and content in t
		| PI	processing instruction
		| Comment	comment
		| ref (t)	reference to t
prime type	q	::= u	
		| q | q	

Table 1: Abstract Syntax for Types

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

Figure 6: Algebra Schema representing DTD in Figure 1.

In Figure 5(a), the type of the column "title" is *String* and the type of column "price" is *Float*. In Figure 5(b), the type of column "price" is $price[Float]$. In Figure 5(c), the type of column "prices" is $(price[Float])\{0, *\}$.

4 Overview of XAT Operators

The XAT algebra has two purposes. First, it is used to explicitly represent the semantics of XQuery. Second, it is used as foundation for the query engine in the Rainbow [7] system to access both XML data and relational data. Every XAT operator's input and output are XAT tables defined in Section 3. There are three kinds of XAT operators based on their purposes: XML operators, SQL operators, and special operators.

XML operators (as shown in Table 2) are used to represent the XML document related operations, e.g., navigation and construction. SQL operators together (as shown in Table 3) correspond to the relational complete subset of our algebra. Special operators (as shown in Table 4) include the operators used temporarily in different phases of optimization and the operators shared both by the class of XML operators and of SQL operators.

There are two modes of the XAT operators, e.g., order-sensitive or not. Some of the operators can only function properly in the order mode, e.g., the position() function. Depending on the nature of the data source, the *source* operator can generate two types of XAT tables, order sensitive or not. For example, the *sortby* operator will change the XAT table to become order sensitive, and the *groupby* operator usually breaks the ordering between groups (but still preserving the order within each group).

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Expose	ϵ	col	N/A	s	Expose the 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 from 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.
Composer	C	p	col	s	Construct a XML document from s according to pattern p.
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 2: XML 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	N/A	ls, rs	Join two input tables ls and rs under condition c.
Outer Join	$\overset{\circ}{\bowtie}_L, \overset{\circ}{\bowtie}_R$	c	N/A	ls, rs	Left (right) outer join two input tables ls and rs.
Distinct	δ	N/A	N/A	s	Eliminates the duplicates in the input table s.
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.

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 4: Special XAT Operators.

5 XML XAT Operators

There are six kinds of XML operators, e.g., Expose, Tagger, Navigate, Aggregate, Composer (nested tagger), and the XML set operators.

5.1 Expose Operator $\epsilon_{col}(s)$

The expose operator will expose the specified column specified as input argument into XML data in textual format. In the XAT tree composition, the expose operator can be connected as child to a source operator that would be expecting an input XML data in textual format. This is usually the root node of an XAT tree. Unlike all other operators, the expose operator will not output any XAT table.

5.2 Tagger Operator $T_p^{col}(s)$

The tagger operator will append a new column to the input table. The new column contains the new XML node created by the tagger operator for each tuple in the input table based on the pattern p (as defined below in Section 5.2.1). The output table has the same order as the input table.

The tagger operator **consumes** the columns used in the pattern p, and **produces** a new column *col* to store the results according to the pattern.

col1
65.95
34.95

(a) Input XAT Table

$T^{col2}_{<price>[col1]</price>}$

(b) Operator

col1	col2
65.95	<price>65.95</price>
34.95	<price>34.95</price>

(c) Output XAT Table

Figure 7: Example of Tagger Operator.

Figure 7 depicts an example of the Tagger operator. It takes the input column $col1$ and tags it by the `price` tag, and then puts it into the output column $col2$. Figure 8 depicts another example to tag a collection. It takes the input column *prices* and tag it by `prices` tag, then puts in the output column $col3$.

prices
{<price>65.95</price>, <price>69.95</price>}
{<price>34.95</price>, <price>39.95</price>}

(a) Input XAT Table

$T^{col3}_{<prices>[prices]</prices>}$

(b) Operator

prices	col3
{<price>65.95</price>, <price>69.95</price>}	<prices> <price>65.95</price> <price>69.95</price> </prices>
{<price>34.95</price>, <price>39.95</price>}	<prices> <price>34.95</price> <price>39.95</price> </prices>

(c) Output XAT Table

Figure 8: Example of Tagger Operator.

5.2.1 Pattern

As we have noticed, each tagger comes with a pattern. Pattern is a template of a valid XML fragment [18] with parameters being column names. The pattern p used in the Tagger can be viewed as an ordered tree composed of the following nodes:

- Root node: identify the root of the pattern.
- Tag node: identify an element tag. It contains the element name.

```
Pattern ::= RootNode, PatternTree
PatternTree ::= (TagNode | AttributeNode), PatternTree
PatternTree ::= PatternTree, PatternTree
PatternTree ::= (ColumnNode | TextNode)
```

Figure 9: Production Rule for the Patterns.

No.	pattern	valid?
(a)	$< minprice@title = [col1] >< price > [col2] < /price >< /minprice >$	yes
(b)	$< result > [col1][col2] < /result >$	yes
(c)	$< result >< price > [col1] < /price >< price > [col2] < /price >< /result >$	yes
(d)	$< price > [col1] < /price >< price > [col2] < /price >$	yes
(e)	$[col1]$	No. There is no tag.
(f)	$< result > [col1] < /price >$	No. Begin-tag and end-tag do not match.

Table 5: Cases of Valid and Invalid Patterns.

- Attribute node: identify an attribute tag. It contains the attribute name.
- Column node: identify the input column used for the content of an element or an attribute value.
- Text node: identify the constants used in the pattern.

The root of the pattern tree is always the *root node*. In the case, of multiple parallel taggers, e.g., `<a>...</a><b>...</b>` , the *root node* is the root of both *tag nodes* of `<a>` and `<b>` . The *column node*, *attribute node*, and *text node* can only be leaf nodes. Figure 9 depicts the production rules of the pattern.

Table 5 illustrates several examples of valid and invalid patterns. Figure 10 depicts a graphical illustration of the pattern (a) in Table 5.

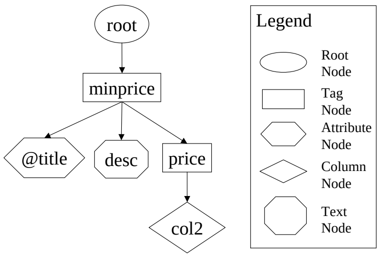

Figure 10: Example of a Pattern.

5.3 Navigate Operator $\phi(\Phi)^{col'}_{col,path}(s)$

There are two kinds of navigate operators: navigate unnest (ϕ), and navigate collection (Φ). Navigate unnest (ϕ) will unnest the parent-children relationship, and duplicate the parent values for each child. We can also call navigate unnest an **iterator**. Navigate collection (Φ) will nest the parent-children relationship, and create a collection of the children, but it would keep all the children under the same single parent value.

The navigate unnest (ϕ) operator will be used in the iteration, which is created due to a FOR binding. The Navigate collection (Φ) operator will be used in all the other places, where we need to identify a new collection, for example the LET binding and the predicates in the WHERE clause. Figure 11 shows the navigate unnest (ϕ) results in four tuples with one for each book price, while Figure 12 shows that the navigate collection (Φ) results in one tuple with a collection of four book prices.

R1
<prices> <book> ... </book> <book> ... </book> <book> ... </book> <book> ... </book> </prices>

$\phi^{col1}_{R1,book}$

R1	col1
<prices>...</prices>	<book> ... <price>65.95</price> </book>
<prices>...</prices>	<book> ... <price>69.95</price> </book>
<prices>...</prices>	<book> ... <price>34.95</price> </book>
<prices>...</prices>	<book> ... <price>39.95</price> </book>

Figure 11: Example of Navigate Unnest from an Element.

R1
<prices> <book> ... </book> <book> ... </book> <book> ... </book> <book> ... </book> </prices>

$\Phi^{col1}_{R1,book}$

R1	col1
<prices> <book>...</book> <book>...</book> <book>...</book> <book>...</book> <prices>	{<book> ... <price>65.95</price> </book> , <book>... <price>69.95</price> </book> , <book>... <price>34.95</price> </book> , <book>... <price>39.95</price> </book> }

Figure 12: Example of Navigate Collection from an Element.

For each tuple in the input XATTable s, we will get the value from the column col, and follow the path specified by $path$, and extract the children elements. The new elements will be placed into the new column col'. If there is no path that satisfies the specific $path$ parameter, an empty collection will be put into the new column col'. The tuple order in the output XAT table will be the same as the input XATTable.

If we navigate from a collection, the navigate unnest (ϕ) will generate new tuples for each value in the collection as shown in Figure 13. The Navigate collection (Φ) will not generate a new tuple for each result, instead it will create a new collection containing all the results as shown in Figure 14. If one book has more than one price, then for each price, a new output tuple will be created. In this case, the number of output tuples will always be no less than the number of input tuples.

col1
{<book>... <price>65.95</price> </book> , <book>... <price>69.95</price> </book> , <book>... <price>34.95</price> </book> , <book>... <price>39.95</price> </book> }

$\phi^{col2}_{col1,book/price}$

col1	col2
{...}	<price>65.95</price>
{...}	<price>69.95</price>
{...}	<price>34.95</price>
{...}	<price>39.95</price>

Figure 13: Example of Navigate Unnest from a Collection.

Table 6 depicts five use cases of navigate unnest (ϕ) operators. The navigation follows the semantics of path expression of XQuery [22]. Please note the difference between case 1 and case 5. Case 1 is a navigation from an ELEMENT, while case 5 is a navigation from a collection. Hence, case 1 has an

col1
{ <book>... <price>65.95 </price> </book> , <book>... <price>69.95 </price> </book> , <book>... <price>34.95 </price> </book> , <book>... <price>39.95 </price> </book> }

$\Phi^{col2}_{col1,book/price}$

col1	col2
{ <book>...</book> , <book> ... </book> , <book> ... </book> , <book> ... <book> }	{ <price>65.95 </price> , <price>69.95 </price> , <price>34.95 </price> , <price>39.95 </price> }

Figure 14: Example of Navigate Collection from a Collection.

empty result, but case 5 has element $< a > ... < /a >$ as the result.

No.	x	operator	y
1	$(< a > ...(no\ tag\ \ < a >)... < /a >)$	$\phi^{y}_{x,a}$	NULL
2	$(< a >< b > ... < /b >< /a >)$	$\phi^{y}_{x,b}$	$< b > ... < /b >$
3	$(< a > \underline{< a > ... < /a >} < /a >)$	$\phi^{y}_{x,a}$	$\underline{< a > ... < /a >}$
4	$(< a > text() < /a >)$	$\phi^{y}_{x,text()}$	text()
5	$(\{< a > ...(no\ tag\ \ < a >)... < /a >\})$	$\phi^{y}_{x,a}$	$< a > ... < /a >$

Table 6: Examples of Navigate Operators.

5.3.1 Navigation Steps

Four types of navigation steps can be used in the Navigate operators based on path expressions of XQuery [22]:

- Attribute (@): To locate an attribute. It will return the attribute with its name and value.
- Children (//, /child): To locate children. It will return the collection of the children with their tags. Please note that, the '//' will retrieve all the descendants of the current node.
- Text (text()): To locate the string value of a given element or attribute.
- Column name (col1): To denote where to get the node and elements to navigate from. The Column name is going to be evaluated based on the input XAT table.

5.4 Aggregate Operator $Agg_{col}(s)$

The aggregate operator will group all the values in the column *col* into one collection (with duplicates).

The *col* of the aggregate operator is the *focus* column, while the rest of the columns are called *context* columns. For the context columns in the input table, the values will be grouped and the duplicates will be removed. The aggregate operator is order preserving.

In the XAT generation, for the FLWR expression, the aggregate operator and the *FOR* operators are generated together. The *FOR* operator will iterate through all values in the *FOR* binding, and hence duplicates values in the context columns. While the aggregate operator will remove the duplicates in the context columns.

For example, if the following table is generated by the FOR operator, and then, we aggregate on column c, then we have the result shown in Figure 15. Please notice the focus column, which is column c, kept the duplicates, while the context columns, for example column b, removed the duplicates.

a	b	c
a1	b1	c1
a1	b2	c2
a1	b2	c2

Agg_c

a	b	c
a1	{ b1, b2}	{c1, c2, c2}

Figure 15: Example of Aggregation.

Please notice, the aggregate and *FOR* operators need a special treatment to avoid the *COUNT bug* [6]. Before the iteration of the FOR operator, an internal unique id will be assigned to each tuple, then, after the FOR operator, the aggregate operator can be implemented as a groupby operator on that internal unique id. Hence it will avoid the *COUNT bug*. Hence, the *aggregate* operator can be seen as groupby on this internal id. For details of this operator, please refer to the Groupby operator.

5.5 Composer Operator $C_p^{col}(s)$

The composer operator will only be available after the SQL generation and optimization. It is a special purpose XML construction operator which will go through the input table once and generate the whole nested XML documents in one pass.

id1	id2	id3	type	value
1	null	null	prices	null
1	2	null	book	null
1	2	3	title	TCP/IP Illustrated
1	2	4	source	www.amazon.com
1	2	5	price	65.95
1	6	null	book	null
1	6	7	title	TCP/IP Illustrated
1	6	8	source	www.bn.com
1	6	9	price	69.95
1	10	null	book	null
1	10	11	title	Data on the Web
1	10	12	source	www.amazon.com
1	10	13	price	34.95
1	14	null	book	null
1	14	15	title	Data on the Web
1	14	16	source	www.bn.com
1	14	17	price	39.95

Figure 16: Example Input Table of Composer Operator.

The composer operator requires that the input table has following schema: *(id[1..n], type, att[1..m], value)*. The $id[1..n]$ columns are used to denote the IDs for each type of XML element. The number of *id* columns (n) is the same as the number of deepest levels in the output XML document. The order in the input table is the document order in the output XML document. The hierarchical relationships between different XML elements are determined by the values in the $id[1..n]$ columns. For example, Figure 16 depicts the input table for the composer to generate the XML document in Figure 2.

5.6 XML Set Operators $\overset{x}{\cup}{}^{col}_{col1,col2}(s)$, $\overset{x}{\cap}$, $\overset{x}{-}$

There are three XML set operators, i.e., XML union ($\overset{x}{\cup}$), XML intersect ($\overset{x}{\cap}$), and XML difference ($\overset{x}{-}$). XML Union ($\overset{x}{\cup}$) is used to union multiple collections into one collection. For each tuple in the input table, it will pick the values in the two specific columns, $col1$ and $col2$. Then, it will union the two collections together and output the union to the new column col as shown in Figure 17. Please notice, single item is also can be seen as a single item collection.

title	price	result
<title>TCP/IP Illustrated</title>	<price>65.95</price>	{<title>TCP/IP Illustrated </title> , <price>65.95</price> }
<title>TCP/IP Illustrated</title>	<price>69.95</price>	{<title>TCP/IP Illustrated </title> , <price>69.95</price> }
<title>Data on the Web</title>	<price>34.95</price>	{<title>Data on the Web</title> , <price>34.95</price> }
<title>Data on the Web</title>	<price>39.95</price>	{<title>Data on the Web</title> , <price>39.95</price> }

Figure 17: Example of XML Union Operators.

The XML Intersection Operator $\overset{x}{\cap}{}^{col}_{col1,col2}(s)$ and XML Difference Operator $\overset{x}{-}{}^{col}_{col1,col2}(s)$ are similar to the XML Union Operator, except for now doing intersection and difference between two collections, respectively.

6 Special XAT Operators

Besides the XML operators proposed in the previous section, there are special XAT operators that are used in the query optimization phase and the computation pushdown phase.

6.1 SQL Statement Operator $SQL^{col[1..n]}_{stmt}$

The SQL statement operator is used to query the underlying relational database system and construct an XAT table out of the query result. This operator will contain the SQL statement to be issued to a relational database engine and also any required database connection information, e.g., address, username and password.

This operator doesn't require any input. Hence it can be used as a leaf node in the XAT tree. It will produce multiple columns, which in turn will be used in the upper level of the trees. The tuple order of the output table depends on the SQL statement stored in this operator. If there is an "orderby" clause in the statement, then the output table is ordered. Otherwise, the output table is not considered ordered,

$$SQL^{col1,col2}_{SELECT\ title,price\ FROM\ book}$$

Figure 18: Example of SQL Statement Operator.

because the order of tuples is assumed to convey no meaning. However, it can be further ordered by an extra sortby operator higher in the tree.

6.2 FUNCTION Operator $F^{col}_{param[1..n]}(s?)$

Functions are used more commonly in the XQuery language than in SQL. There are a number of categories of functions proposed in the XQuery data model standard [23]. Table 7 shows seven kinds of functions, e.g., string manipulation, aggregation, sequence, data and time, context, node, and user-defined. All of them can be generalized by a function operator which will take an arbitrary number of parameters and generate one new column with or without an input.

Type	*Examples*
String	Concat, contains, lowercase, name, starts-with, subst, trim, uppercase, ...
Aggregation	avg, count, max, min, sum, ...
Sequence	exists, empty, subsequence, union, intersect...
Date and Time	date, time, ...
Context	last, position, context-item, ...
Node	shallow, name, copy, root, ...
User Defined	foo() ...

Table 7: Example of Functions.

We can roughly divide those functions into three categories based on their structure. 1) Functions that take multiple tuples as input but only generate one tuple, e.g., the aggregation functions. 2) Functions that will generate a new value for each tuple, e.g., shallow, position, lowercase, name. 3) Functions that don't require any input but generate new data, e.g., date, time.

Function operators make the XQuery extensible and support many advanced features, e.g., recursive queries, string manipulations, etc. This also provides a convenient means in our algebra to extend our operator's expressive power in the future. At this moment, we don't explore how the function can be used in the recursive queries.

6.3 Source Operator $S^{col[1..n]}_{desc}$

The source operator will represent the data source processed in the XAT. It will generate one XAT table based on the input description. The description can be an XML document's name, database table description, an XML fragment, a view name, or other more flexible descriptions as used for example in Niagara [8].

If the source is a database table description, then the relational data will be automatically translated into a default XML view in order to be processed by the XAT. Here are examples of the possible descriptions in Table 8.

Description	Semantics
s(*)	All known XML documents.
s(url)	Any specific XML document identified by the *url*.
s(VIEW view1)	An XML view with name *view1*.
s(TABLE table1)	An RDB table identified by name *table1*.

Table 8: Example of Descriptions of Source Operator.

If the source operator identifies a relational table, the output XAT table will be the same as the relational table. If the source operator identifies one XML document, the whole document will be stored in the XAT table as one value as shown in Figure 19. If the source operator identifies a piece of the XML data, this is, an XML fragment, it will also be stored as one value in the output XAT table. If the source operator identifies a collection of documents, then multiple tuples will be generated one for each document.

$S^{R1}_{``prices.xml''}$

```
R1
<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 19: Example of Source Operator.

6.4 Name Operator $\rho_{col1,col2}(s)$

The name operators will rename a column in an XAT table [1]. Renaming is useful to avoid name conflicts and to assign semantics to the columns. The column name operator, $\rho_{col1,col2}(s)$, will rename a column

[1]We have made the decision not to rename the table name, because we assume the column names are global unique.

```
FOR $b IN document("prices.xml")/book,
    $t IN $b/title,
RETURN
  $t
```

(a)

$$
\begin{array}{l}
FOR_{\$b}(\\
\qquad \phi^{\$b}_{c1,book}(s(\text{"}prices.xml\text{"})^{c}1), \\
\qquad FOR_{\$t}(\phi^{\$t}_{\$b,title}, \phi^{\$t}_{\$t}) \\
)
\end{array}
$$

(b)

Figure 20: XQuery and XAT with two for bindings.

*col*1 into *col*2. It can also be written as expression $s.col1 \rightarrow s.col2$ in the expression parameter of the projector π. Same as in relational algebra, the $\rho_{col1,col2}(s)$ can be used with the π operator.

6.5 FOR Operator $FOR_{col}(s, sq)$

The FOR operator is used to represent the iteration of the for-bindings in the XQuery. Since most XQueries have a FOR clause, this is a very common operator in the XAT. It will iterate through a column in the input table and execute the subquery *sq* for each value in the column that is the for-binding. The output table of the FOR operator is the result of its subquery. The subquery is evaluated for each value in the binding identified as column *col* in the input table *s*. *s* corresponds to the table that contains the FOR binding. *sq* is the subquery, which is another XAT tree, executed for each value in column *col* of the *s* input table. If there are multiple FOR bindings in the XQuery, multiple FOR operators will be generated, one for each for binding.

For example, for the query in Figure 20(a) with two FOR bindings, two FOR operators will be generated. The corresponding algebra is depicted in Figure 20(b):

The FOR operators usually will be removed in the decorrelation process and replaced by outer joins and other operators, e.g., groupby operator, Cartesian operator, etc.

6.6 IF_THEN_ELSE Operator $IF_c(sq1, sq2)$

For the sake of completeness, we have this branch operator to represent the decision making in the XQuery. The condition c is evaluated before the operator decides which branch to follow. If c is true, then the subquery $sq1$ is executed, otherwise the subquery $sq2$ is executed.

In the XAT optimization, if possible, the condition can be analyzed to decide whether the condition is always true or false, hence, we can remove one of the branches.

6.7 Merge Operator $M(s[1..n])$

The merge operator will merge two tables vertically into one table by concatenating columns. Because the XAT tables are order sensitive, we can also call it "Join by Order". Hence it requires both input tables are ordered. It also requires the two input tables to have the same number of tuples. Figure 21 depicts the merging of two tables containing columns *title* and *price* respectively into one table containing both columns.

The merge operator is used to represent the sequence construction in the XQuery. It will also be used during query decorrelation to take the position of the set operators.

S1

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

S2

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

$\xrightarrow{M(s1,s2)}$

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

Figure 21: Example of Merge Operator.

The merge operator used quiet often in a sequence of XQuery statements.

Figure 22 depicts an example of using the *merge* operator. As we can see the query wants to filter out the `source` element and the outer tag `book`. The return clause is a sequence of element constructions. The first element contruction will create a `title` element, and the second one will create a `price` element. As we can see, the decorrelated XAT in Figure 22(b) uses *XML union* (line 2) and *merge* (line 3) operators to represent the sequence construction (line 3 and 5) in Figure 22(a).

7 SQL XAT Operators

The purpose of the XAT operator is to serve as the foundation for the Rainbow system which is bridging two distinct data models, namely, the XML data model and the Object-Relational data model. Hence, besides the XML and Special XAT operators, XAT also incorporates and extends as appropriate for the relational operators to the XML context in order to be relational complete. We have the following relational algebra operators [9]: project, select, Cartesian project, join, distinct, group by, order by, and set operators.

7.1 Project Operator $\pi_{col[1..n]}(s)$

Project operator is used to project out columns in the input table. The output table will only keep the columns $col[1..n]$ specified in the operator. The tuple order stays the same as in the input table.

As the extension to the column names, project operator can also have an expression as an extra

```
1: FOR $b IN document("prices.xml")/book
2: RETURN
3: {
4:   <title>$b/title</title>,
5:   <price>$b/prices</price>
6: }
```

(a) XQuery Example.

$$
\begin{array}{ll}
1: & \epsilon_{col1}(\\
 & \quad {}_{x}\,{}^{col1} \\
2: & \quad \cup_{col2,col3}(\\
3: & \qquad M(T^{col2}_{<title>col4</title>}(\ldots), \\
4: & \qquad\quad T^{col3}_{<price>col5</price>}(\ldots))
\end{array}
$$

(b) XAT after Decorrelation.

Figure 22: Example Query where the Merge Operator is used.

output column. Also, the project operator can also assign a new name to the column derived by an expression or one of the existing columns. This way has the overlapping functionality of the $\rho_{col1,col2}$ operator. We put this functionality in the π operator is for implementation convenience.

7.2 Select Operator $\sigma_c(s)$

Select operator will filter out the input table based on the condition c specified in the operator. The tuple order will be kept the same.

Expression. Both the *join* and *select* operators may specify a condition c in the format of an expression. In our algebra, we support the basic arithmetic and boolean expressions, which include negation, addition, subtraction, multiplication, division, NOT, OR, AND, and comparison. The terminals of the expression can be a string, a double value, and a column name, which will be evaluated based on the evaluation context. Table 9 gives out examples of expressions.

Expression	**Semantics**
\$a = \$b	Compare if \$a equivalent to \$b.
\$a = \$b and \$b > 4	Compare if \$a equivalent to \$b and \$b is greater than 4.
\$a = \$b + \$c	Compare is \$a is the sum of \$b and \$c.

Table 9: Examples of Expression.

7.3 Cartesian Product Operator $\times(ls, rs)$

We also have the Cartesian product operator in our algebra with essentially the same semantic as the relational algebra correspondent. It can also written as $ls \times rs$. In the order sensitive mode of this operator, it will sort with the major order of the left table first and minor order of the right table next. In the column name assignment, we make global unique column names. Hence there is no name conflict.

7.4 Join Operator $\bowtie, \mathring{\bowtie}_L, \mathring{\bowtie}_R$

There are three join operators supported by our algebra. The theta join $\bowtie$ operator and two outer join ($\mathring{\bowtie}_L$, $\mathring{\bowtie}_R$) operators.

7.4.1 Theta Join Operator $\bowtie_c (ls, rs)$

Theta Join operator is similar to its relational algebra equivalent. It can be treated as a Cartesian product operator followed by a select operator with condition c. It keeps the same order as the Cartesian product operator and the select operator. The theta join operator can be also represented as $(\sigma_c(ls \times rs))$, or $ls \bowtie_c rs$.

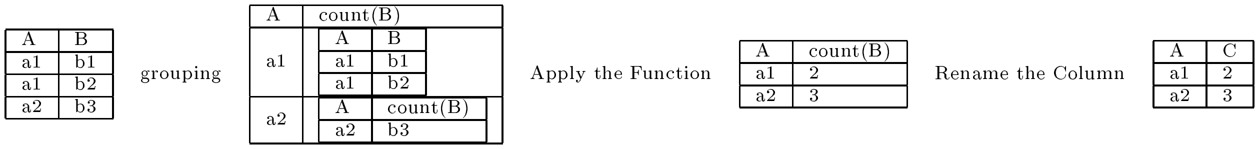

Figure 23: Example of Execution of Groupby.

7.4.2 Outer Join Operator $\overset{\circ}{\bowtie}_{Lc}(ls, rs)$, $\overset{\circ}{\bowtie}_{Rc}(ls, rs)$

Output join operator is similar to the relational algebra. The left outer join $\overset{\circ}{\bowtie}_{L}$ will join two tables together, but keep all the tuples in the left input table even if no match can be found on the right. The right outer join $\overset{\circ}{\bowtie}_{R}$ will keep all the tuples in the right input table. It can also written as $ls \overset{\circ}{\bowtie}_{Lc} rs$ and $ls \overset{\circ}{\bowtie}_{Rc} rs$.

7.5 Distinct Operator $\delta(s)$

The distinct operator eliminates the duplicates in the input table s. It will convert the whole XAT table from bag semantic into set semantic.

7.6 Groupby Operator $\gamma_{col[1..n]}(s, sq)$

Groupby operator will divide the input table s into a set of smaller tables, i.e., groups, by the distinct values of columns $col[1..n]$. Then, it will apply the subquery sq for each group. After that, the result will be merged back into one output table. The output columns are decided by the subquery sq. The groupby operator can preserve order within each group. But the order in the columns $col[1..n]$ is not preserved.

If the subquery sq has only functions, we can rewrite the groupby operator in this format: $\gamma^{col+}_{col[1..n]}(s)$, where $col+$ can have aggregation functions. For example, $\gamma^{d}_{a,b}(s, count()^{d}_{c}) \rightarrow \gamma^{count(c) \rightarrow d}_{a,b}(s)$, means group the input table on columns a and b, and count the number of c for each group, and store the count result into a new column d. Figure 23 depicts the execution of $\gamma_A(s, count()^{C}_{B})$.

7.7 Orderby Operator $\tau_{col[1..n]}(s)$

The Orderby operator is the one used to force the XAT table into a specific order. The input XAT table can be sorted by multiple columns. In the algebra tree, the operators above this operator are order sensitive. For example $position()$ function can only work properly if the input table is ordered.

7.8 Set Operators

For the completeness of our algebra, we include the three set operators in our algebra, e.g., the union, difference and intersect. The input tables are bags and their schema must be the same (or subsume each other at least) except in the case of the outer union operator. As specified in [22], duplicates are removed based on node id and also sorted by document order.

7.8.1 Union Operator $\cup(s[1..n])$

Similar to the relational union, the union operator will merge multiple XAT tables into one XAT table. It requires all the XAT tables to have the same schema.

In the order sensitive mode, the tuple order in the output table will be the first input table $s[1]$, followed by the second one $s[2]$, the third one $s[3]$, and so on.

For two input tables, the operator can be rewritten as $s1 \cup s2$.

7.8.2 Outer Union Operator $\overset{\circ}{\cup}(s[1..n])$

Outer union doesn't require that the input tables have the same schema. Instead, it will keep all the columns in all the tables. It is useful in the publishing of XML documents. Here, we only use the full outer union.

$\{x|x \in s_1 \ or \ x \in x_2 \ or \ ... \ or \ x \in x_n\}$

For two input tables, the operator can be rewritten as $s1\overset{\circ}{\cup}s2$.

a	b
a1	b1
a2	b2

a	c
a1	c1
a3	c3

$\rightarrow$

a	b	c
a1	b1	c1
a2	b2	null
a3	null	c3

Figure 24: Example of Outer Union.

7.8.3 Difference Operator $-(ls, rs)$

Difference operator will only keep the tuples in table ls, but not in table rs. It requires the ls and rs have the same schema. The output order is same as the order in left table ls. For two input tables, the operator can be rewritten as $s1 - s2$.

7.8.4 Intersection Operator $\cap(s[1..n])$

Intersection operator will only keep the common tuples in all the input tables $s[1..n]$. The output order is same as the order in left table ls. For two input tables, the operator can be rewritten as $s1 \cap s2$.

8 XAT Generation Issues

The XAT generation is also called the "Query Decomposition" step of query processing. As in the "Query Decomposition", the XAT generation includes the following steps:

- Analysis: In this stage, the query is lexically and syntactically analyzed using the parser. All the syntax errors are reported. This is done by the external XQuery parser, in our case the Kweelt engine [16]. After the analysis, the XAT will be generated. This section will focus on the algebra tree generation.
- Normalization: "The normalization stage of query processing converts the query into a normalized format that can be more easily manipulated." For example, the Navigator, Tagger, and Select operators can be normalized by applying transformation rules. Rewriting of XAT for optimization will be discussed in a separate report [26].
- Semantic analysis: "The objective of semantic analysis is to reject normalized queries that are incorrectly formulated or contradictory." For example, if $a > 4$ and $a < 2$ both appeared in the same where clause, then the where clause is meaningless.
- Simplification: "The objectives of the simplification stage are to detect redundant qualifications, eliminated common sub-expressions, and transform the query to a semantically equivalent but more easily and efficiently computed form". This is covered in the query rewriting, the computation pushdown, and schema cleanup in a separate technical report [26].

Not all the operators mentioned above will be used in the default XQuery. Only the operators listed in table 10 are used in the XQuery to XAT generation.

Type	Operators
XML	T, ϕ, Φ, $\overset{x}{\cup}$, Agg()
Special	F, S, FOR, IF
SQL	ϕ, σ, γ, Σ, $\cap$, $-$.

Table 10: XAT Operators used in XAT Generation.

Compared to SQL, XQuery is a complex query with correlated nested queries. There are a couple of issues to be considered during the XAT generation:

- Convert path expression with filters into XAT.
- Convert FLWR into XAT.

8.1 Convert Path Expression into XAT

Path expression will be represented by a combination of Navigate, Select, and Groupby operators. A path expression is composed of location steps (with axis and node test), predicates, and optional parenthesis for grouping, as further explained below. The output of a path experssion is a collection.

8.1.1 Path Expression with Only Location Steps

A path expression with only location steps can be represented as one navigation operator. If the path expression is inside the FOR binding, it will be a navigation unnest (ϕ), otherwise it will be a navigation collection (Φ). Each location step will be represented as one navigation step in the navigation operator.

To unify the case within or outside of a for binding, all the navigation steps will be navigation unnest (ϕ). At the end of the path expression an $Agg()$ will aggregate all the results into a collection. If there is a for binding, then there will be an additional navigation unnest (ϕ) to unnest the aggregated results. Hence, in the following discussion, we will only use the navigation unnest (ϕ).

In the XQuery example in Figure 3, a path expression $book/title$ from column x and bind result in column y in a for binding will be represented as $\phi^{y}_{x,book/title}$.

8.1.2 Path Expression with Predicates

There are two ways to evaluated a predicates, i.e., context sensitive evaluation or not. For example, the *position()* related predicates requires evaluation context, but for the *content* related predicates doesn't require the context.

Context Insensitive Predicates The context insensitive predicates are very easy to evaluate, because the predicates are evaluated locally. Hence, the predicates will be represented as a *select* operator on the top of the navigations.

In general, for expression $E1[E2]$, it will be translated into $E2(E1)$. For example, the $book[title ='' Data\ on\ the\ Web'']$ will be translated into: (with root denoted by r)

$\rho_x(\sigma_{y=''Data\ on\ the\ Web''}(\phi^{y}_{x,title}(\phi^{x}_{r,book})))$.

The expression means that: 1) Find all the books, 2) Find the title for each book, 3) filtered by the title's condition.

Another example for the existence of a child node, $book[title]$ will be translated into:

$\rho_x(\sigma_{y!=EMPTY}(\phi^{y}_{x,title}(\phi^{x}_{r,book})))$.

The navigate operator will keep a *book* tuple with *title* either empty or not in the result. All books without *title* will be filtered by the σ operator.

Please note that there is a *name column* ρ operator on the top of each path expression. That is used to denote which column should be the focus on the rest of the query trees. For example $book[title]$, here *book* should be the focus, which is the column x instead of the top most one, which is column y representing the *title*.

Context Sensitive Predicates For the context sensitive predicates, the context is stored as a column in input XAT tables to the condition evaluation node. For example, if a path expression generates a column y from column x, then for each tuple, the value in the column x is the evaluation context of the value in the column y.

For expression $C/E1[E2]$, the context item is C, and the context sequence is $C/E1$. Hence, in the XAT table, we need to group the sequence $C/E1$ by C, and evaluate the predicates in the nodes in which the items are sequenced in the group. It has the following pattern: $\gamma_x(C^x, E2(E1))$. If there is no C, for the context sensitive predicates $E1$, then it is handled same as the context insensitive predicates.

For example of a position predicates of a node, $root/book[2]$ will be translated into:

$\rho_x^x(\gamma_r(\phi_{s,root}^r, \sigma_{y=2}(position()^y(\phi_{r,book}^x)))))$.

Please notice in this example, we group the output of the $\phi_{r,book}^x$ by r. The reason for this is because the $position()$ function used in the predicates, which requires the context position of the context item, and the context position can be only evaluated in the higher level of the context item.

Then, in this example, $root/book[2][title ='' \ Data\ on\ the\ Web'']$, we have:

$\rho_x^x\left(\sigma_{z=''Data\ on\ the\ Web''}\left(\phi_{x,title}^z\left(\gamma_r\left(\phi_{s,root}^r, \sigma_{y=2}(position()^y(\phi_{r,book}^x))\right)\right)\right)\right)$

To simplify the complexity of the algebra expression, a couple of transformation rules between the group by operators and the select and navigate operators are proposed in a separate technical report [26].

8.1.3 Path Expression with Parenthesis

The path expression with parenthesis will change the evaluation priority, hence the evaluation context for the path expression. In this case, we have $E1/E2[E3]$, the evaluation order will be for each $E1$ evaluate $E2[E3]$, and for each $E2$ evaluate $[E3]$. If we have $(E1/E2)[E3]$, then for each $E1/E2$ we evaluate $[E3]$.

In terms of the algebra, the group by columns are changed to account for those differences in these semantics. For $E1/E2[E3]$, we have:

$\gamma_x(E1^x, \gamma_y(E2^y, E3))$.

For $(E1/E2)[E3]$, we have:

$\gamma_y(E2^y(E1^x), E3)$.

For example, $book/title[2]$ is:

$\gamma_x(\phi_{r,book}^x, \gamma_x(\phi_{x,title}^y, \sigma_{z=2}(position()^z))$.

But $(book/title)[2]$ is:

$\gamma_r(\phi_{x,title}^y(\phi_{r,book}^x), \sigma_{z=2}(position()^z))$.

8.1.4 General Path Expression

Before we translate a path expression into an algebra expression, we need to cut it into multiple simpler parts as described above. Then, we connect the pieces of the expressions together.

For example, given the path expression `root/(book[price > 30]/title)[2]` , we list the steps to translate such expression into XAT in Figure 25.

Step	XAT	Description
1	$\phi^r_{s,root}/(book[price > 30]/title[1])[2]$	Handle `root`
2	$\phi^r_{s,root}/(\phi^b_{r,book}[price > 30]/title[1])[2]$	Handle `book`
3	$\phi^r_{s,root}/(\phi^b_{r,book}[\phi^p_{b,price} > 30]/title[1])[2]$	Handle `price`
4	$\phi^r_{s,root}/(\phi^b_{r,book}[\phi^p_{b,price} > 30]/\phi^t_{b,title}[1])[2]$	Handle `title`
5	$\phi^r_{s,root}/(\sigma_{p>30}(\phi^p_{b,price}(\phi^b_{r,book}))/\phi^t_{b,title}[1])[2]$	Handle `[price>30]`
6	$\phi^r_{s,root}/(\gamma_b(\sigma_{p>30}(\phi^p_{b,price}(\phi^b_{r,book})),\sigma_{bp=1}(position()^{bp}(\phi^t_{b,title}))))[2]$	Handle `/title[1]`
7	$\rho^t_t(\gamma_r(\phi^r_{s,root},$ $\sigma_{rp=2}(position()^{rp}$ $(\gamma_b(\sigma_{p>30}(\phi^p_{b,price}(\phi^b_{r,book})),$ $\sigma_{bp=1}(position()^{bp}(\phi^t_{b,title})))))))$	Handle `/()[2]`

Figure 25: Example of Translation of a Path Expression.

8.2 Convert FLWR into XAT

There are two kinds of binding, i.e., LET binding and FOR binding, in the FLWR expression. One important task of XAT generation is to correctly represent these two bindings. The LET binding will bind a collection of data to a variable, while the FOR binding will iterate through a collection and bind each value in that collection to a variable at a time.

8.2.1 Variable Bindings

Because of the different semantics of the two types of bindings, they are translated into two different types of navigation operators.

The path expression used in the FOR binding will use the Navigate-unnest ($\phi^y_{x,p}$) operator. It will unnest all the values that can be reached by the path p, and make a new tuple for each value and also duplicate the original value in column x. For example,

```
FOR $x IN for-binding
Inner-query use $x
```

will be translated into XAT: $\mathbf{FOR}_{\$x}(\phi^{\$x}$**(for-binding), <u>Inner-query use $x</u>**).

The path expression used in the LET binding will use the Navigate-collection (Φ) operator. It will put the result of the navigation into one collection and only make it into a single tuple. The original values will not duplicated for such binding. For example,

```
LET $x := let-binding
Rest-of-query use $x
```

will be translated to XAT: **<u>Rest-of-query use $x</u>**($\Phi^{\$x}$**(let-binding)**).

Table 11 compares the differences between is the navigate operators for the FOR and LET bindings. Figure 26 gives out an example.

Binding	**XAT**
`FOR $x IN document("x.xml")/x`	$\phi^{\$x}_{R1,x}(S^{R1}_{"x.xml"})$.
`LET $x := document("x.xml")/x`	$\Phi^{\$x}_{R1,x}(S^{R1}_{"x.xml"})$.

Table 11: Different Bindings for FOR verses LET Clauses.

```
FOR $b IN /book
LET $t := $b/title
RETURN  $t
```

Step	XAT	Description
1	$FOR_{\$b}(\phi^{\$b}_{r,book},$ `LET $t:= $b/title RETURN $t`$)$	Handle `FOR $b IN /book` .
2	$FOR_{\$b}(\phi^{\$b}_{r,book},$ `RETURN $t`$(\Phi^{\$t}_{\$b,title}))$	Handle `LET $t := $b/title` .
3	$FOR_{\$b}(\phi^{\$b}_{r,book}, \epsilon_{\$t}(\Phi^{\$t}_{\$b,title}))$	Handle `RETURN $t` .

Figure 26: Example of FOR/LET Binding Translations.

8.2.2 FOR Clause

There will be one Aggregation operator $Agg()$ added for each FLWR expression. But between multiple FOR bindings, no $Agg()$ would be added. That's because of the semantics of the FLWR expression. The aggregation operator is used to aggregate the result returned by the RETURN clause into one tuple. While the multiple FOR bindings it used to iterate through the all the variables and generate the results. Hence, for one FLWR expression, there is only one $Agg()$ required at the end, and in the middle we don't need to aggregate the intermedia results. Please distinguish from the case, where there is a nested FLWR in the return clause. In this case, there are multiple FOR bindings, but they belongs to difference FLWR expressions, hence, there are more than one $Agg()$ generated, one for each FLWR expression.

Multiple FOR operators (without $Agg()$) can be combined into one FOR operator via a Cartesian product of the binding variables.

8.2.3 LET Clause

LET clauses will be converted into algebra expressions and placed at the bottom of the FOR operator's subtree. If there is no FOR operator, then the algebra expression will be placed at the bottom of the algebra tree. If there are multiple let clauses in the FLWR expression, the LET binded variables are put as a linear tree. If there are independent sources in the LET clause, the let clauses can be combined using Cartesian product operators. During the decorrelation in [25], Rainbow will produce further Cartesian products will be produced for the FOR binding variables.

8.3 A Full Example

Figure 27 depicts one example of the parsed tree generated by the Kweelt parser [16] for the XQuery in Figure 3. Figure 29 depicts the XAT algebra expression for the parsed tree in Figure 27. Figure 30 graphically illustrates the expression.

The generation algorithm will do a top-down traversal of the parsed tree. For each parsed tree node, different strategies will be used to build subtrees of XAT piece by piece, and then connected together to build the XAT tree. Figure 28 describes the relationship between the parsed tree and the XAT.

```
1: QuiltQuery(
2:  ElementConstruct(<Results>,
3:      FLWRExpression(
4:          Binding(
5:              ForBinding($t,   distinct, Nav(
6:                      FunDocument("prices.xml"),
7:                      Steps(
8:                          LocationStep(//), LocationStep(book), LocationStep(title))),
9:              LetBinding($p,  Nav(
10:                     FunDocument("prices.xml"),
11:                     Steps(
12:                         LocationStep(//),
13:                         LocationStep(book,
14:                             BinOpComp(=,
15:                                 Nav(CurrentNode,
16:                                     Steps(
17:                                         LocationStep(title))),
18:                                 Nav(Var($t), Steps(Text()))),
19:                         LocationStep(price))))),
20:         ElementConstruct(<minprice>,
21:             AttributeExpression(@title,
22:                 Nav(Var($t), Steps(Text()))),
23:             ElementConstruct(<price>,
24:                 FunMin(
25:                     Nav(Var($p, Steps(Text()))))))))))
```

Figure 27: Parsed Tree of XQuery in Figure 3.

P.T.[a]	XAT	Description
1		
2	1	One tagger per element construct.
3	2	Every FLWR expression has an aggregation function on the top.
4	3	The for binding node.
5	4	*distinct* property changed into an operator.
6	6	The document() creates a new source node.
7,8	5	The navigation node for multiple location steps.
9	11	LET binding at bottom of the tree.
10	16	The new source node created in the let binding.
11	11-15	Please see desc below for each step.
12, 13	15	Navigates to the `book` . The filter separates two navigation operators.
14	12	Creates the condition in the *select* operator.
15, 16, 17	14	Prepare for left part of expression in operator 12.
18	13	Prepare for right part of expression in operator 12.
19	11	Navigates to the `price` .
20	7	creates outmost tagger in pattern.
21	7	creates attribute part in pattern.
22	10	creates value of the attributes.
23	7	The inner tag of this operator.
24	8	The min().
25	9	Navigates to the value of var `$p` .

Figure 28: Parsed Tree and Algebra Node Mapping Table.

[a]P.T.: Parsed Tree.

1: $T_{<results>col8</result>}^{col9}($
2: $Agg($
3: $FOR_{\$t}($
4: $\delta_{col1}^{\$t}($
5: $\phi_{R1,\ //book/title}^{col1}($
6: $S_{\text{"prices.xml"}}^{R1})),$
7: $T_{<minprice\ title=[col5]><price>[col7]</price></miniprice>}^{col8}($
8: $Min_{col6}^{col7}($
9: $\Phi_{\$p,\ text()}^{col6}\ ($
10: $\Phi_{\$t,\ text()}^{col5}($
11: $\Phi_{col2,price}^{\$p}($
12: $\sigma_{col3=col4}($
13: $\Phi_{\$t,\ text()}^{col4}($
14: $\Phi_{col2,title}^{col3}($
15: $\Phi_{R2,//book}^{col2}($
16: $S_{\text{"prices.xml"}}^{R2})))))))))))$

Figure 29: XAT Algebra Expression of XQuery in Figure 3.

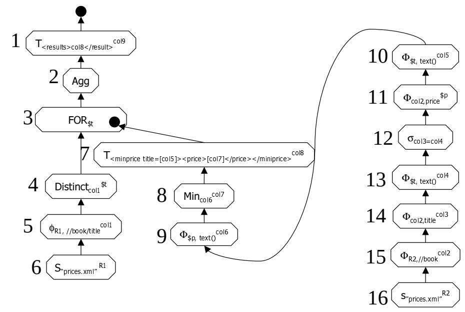

Figure 30: XAT Graph of Algebra in Figure 3.

9 Related Work

Recently, a lot of research work has been done related to the XML algebra, most notably, XPERANTO [1] and Niagara [8]. In this section we compare our work with those two algebras.

As shown in Table 12, our algebra is the first algebra that can universally represent different aspects of XQuery, e.g., navigation and construction, and also supports the translation into SQL query expressions.

10 Conclusions

In this report, we have proposed an XML query algebra called XAT. XAT is composed of three kinds of operators, e.g., XML specific operators, special operators, and SQL operators. We have also describe the translation from XQuery statemens into our XAT representation. The Rainbow system we are developing

	XPERANTO	NIAGARA	XAT
Goal	XQuery → SQL	XQuery → Algebra	XQuery → Algebra → SQL
Algebra	XQGM and Tagger Graph	XML Algebra	Universal Algebra
Data Model	Tables of a list of XML fragments	A collection of bags of vertices	Tables of collections.
Operators	10 operators with 13 functions	12 operators	27 operators
Variable Binding	Lot of temporary variables	No variables	Internal columns and variables.
Order	Sensitive	Semi-sensitive (missing orderby)	Sensitive/Insensitive
Regular Expression	No support	Support	Support
Text-in-context	No support	Support	Support
Level of abstraction	Function level (lower)	Logical level (higher)	Logical level
Transition rules	Composition rules	(ad-hoc) 1 Semantically equivalent pattern	A set of comprehensive rewriting rules.
Operation Hitory	Not maintained	Maintained	Maintained

Table 12: Comparison with other XML algebras.

is based on our XAT algebra. Query decorrelation and computation pushdown strategies for XQuery processing in Rainbow will be described in separate reports [26, 25].

References

[1] 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.

[2] B. Catania, E. Ferrari, A. Y. Levy, and A. O. Mendelzon. XML and Object Technology. In *ECOOP Workshops*, pages 191–202, 2000.

[3] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB*, pages 53–62, 2000.

[4] S. Cluet, S. Jacqmin, and J. Simeon. The New YATL: Design and Specifications. Technical report, INRIA, 1999.

[5] D. Chamberlin and P. Fankhauser and M. Marchiori and J. Robie. XML Query Use Cases. http://www.w3.org/TR/xmlquery-use-cases, April 2002.

[6] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 197–208. Morgan Kaufmann, 1987.

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

[8] L. Galanis, E. Viglas, D. J. DeWitt, J. F. Naughton, and D. Maier. Following the paths of xml data: An algebraic framework for xml query evaluation. http://www.cs.wisc.edu/niagara/papers/algebra.pdf, 2001.

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

[10] W. X. W. Group. XSL Transformations (XSLT). http://www.w3.org/TR/xslt/.

[11] G. Kappel, E. Kapsammer, S. Rausch-Schott, and W. Retschizegger. X-Ray - Towards Integrating XML and Relational Database Systems. In *International Conference on on Conceptual Modeling*, pages 339–353, October 9-12 2000.

[12] 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.

[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. Renner. XML Data and Object Databases: A Perfect Couple? In *IEEE Int. Conf. on Data Engineering*, pages 143–148, April 2001.

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

[17] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Int. Conference and Workshop on Database and Expert Systems Applications*, pages 206–217, August 1999.

[18] W3C. XML^{TM} . http://www.w3.org/XML, 1998.

[19] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, 1999.

[20] W3C. The XML Query Algebra. W3C Working Draft. http://www.w3.org/TR/query-algebra, December 2000.

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

[22] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, December 2001.

[23] W3C. XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/2001/WD-query-datamodel-20010607, June 2001.

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

[25] Xin Zhang. XML Query Decorrelation. Technical report, Worcester Polytechnic Institute, 2002. to appear.

[26] Xin Zhang and Bradford Pielech. XAT Optimization. Technical report, Worcester Polytechnic Institute, 2002. to appear.

[27] X. Zhang, G. Mitchell, W.-C. Lee, and E. A. Rundensteiner. Clock: Synchronizing internal relational storage with external xml documents. In K. Aberer and L. Liu, editors, *Eleventh International Workshop on Research Issues in Data Engineering: Document Management for Data Intensive Business and Scientific Applications, Heidelberg, Germany, 1-2 April 2001*, pages 111–118. IEEE Computer Society, 2001.