Query Rewrite for XML in Oracle XML DB

Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikutty, James W. Warner, Vikas Arora, Susan Kotsovolos

Oracle Corporation
400, Oracle Parkway
Redwood Shores, CA 94065
USA

{Muralidhar.Krishnaprasad, Zhen.Liu, Anand.Manikutty, Jim.Warner, Vikas.Arora, Susan.Kotsovolos}@oracle.com

Abstract

Oracle XML DB integrates XML storage and querying using the Oracle relational and object relational framework. It has the capability to physically store XML documents by shredding them as relational or object relational data, and creating logical XML documents using SQL/XML publishing functions. However, querying XML in a relational or object relational database poses several challenges. The biggest challenge is to efficiently process queries against XML in a database whose fundamental storage is table-based and whose fundamental query engine is tuple- oriented. In this paper, we present the '**XML Query Rewrite**' technique used in Oracle XML DB. This technique integrates querying XML using XPath embedded inside SQL operators and SQL/XML publishing functions with the object relational and relational algebra. A common set of **algebraic rules** is used to reduce both XML and object queries into their relational equivalent. This enables a large class of XML queries over XML type tables and views to be transformed into their semantically equivalent relational or object relational queries. These queries are then amenable to classical relational optimisations yielding XML query performance comparable to relational. Furthermore, this rewrite technique lays out a foundation to enable rewrite of XQuery [1] over XML.

1. Introduction

XML processing in the Oracle XML DB is based on the XMLType datatype. This is a native datatype introduced in Oracle 8i [4]. Built using the object relational infrastructure, the XMLType is similar to other built-in datatypes such as number and character. Various XML related operations are supported on the XMLType. These are useful for shredding the XML data to relational and object relational data, for traversing the XML content using XPath notation, or for generating XML from relational and object relational data. [2]

In particular, users can create tables of XMLType storing all XML document instances conforming to a particular XMLSchema [11] registered in Oracle XML DB. This is referred to as XMLSchema-based storage where the XMLType instances are stored in a shredded form in object-relational columns. The object relational types and columns are generated when the XML schema is registered [5]. **Collection elements** (elements with maximum occurrences greater than one) can be stored inline as a variable array (VARRAY) of object instances or stored in a separate collection table.

Users can also convert existing relational data into logical XML values using views. Oracle XML DB supports the SQL/XML standard [3,12,13] which allows users to leverage a set of SQL/XML publishing functions, such as *XMLElement*, *XMLAgg*, *XMLForest*, and *XMLConcat* to flexibly generate XMLType instances from the relational data. *XMLAgg* is an important aggregate function for concatenating a rowset of XML values through a relational query.

To query XML tables or views, Oracle XML DB provides a set of functions, such as *Extract*, *ExistsNode*, and *Extractvalue* that use XPath to locate and extract data from XML documents [2]. Users can then use SQL/XML publishing functions to construct new XML nodes. Collection of XML values can be un-nested using the table function, *XMLSequence* that converts a list of XML elements into multiple rows.

A simple approach to querying XML in a database is to first materialize the entire XMLType instance as a DOM and then use an XPath engine to traverse the DOM. However, this solution is very expensive due to the cost of materializing the entire XML, and is foreign for a

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

relational engine, which is designed to process tuple-oriented data instead of tree-oriented XML data.

A key observation is that since the XML is constructed logically through the underlying relational data in Oracle XML DB, it is feasible to rewrite the XPath navigation to select just the underlying data that is needed for the construction of the result. Further, XPath predicates can be transformed to predicates on the underlying relational tables. Thus, the original query over XML can be logically rewritten into an equivalent relational query. This can be then optimized by a standard relational optimizer, for instance, by picking the best join order and index on the underlying tables.

This approach, referred to as '**XML Query Rewrite**', enables Oracle XML DB to transform XML queries into their equivalent relational queries. The theme of this paper is to show the framework that we use in Oracle XMLDB to perform such rewrites.

The rest of this paper is organized as follows. Section 2 provides examples of querying XML in Oracle XDB. Section 3 gives an overview of XML query rewrite. Section 4 discusses XPath transformation to SQL operators. Section 5 discusses operator tree optimizations with SQL/XML normalization and optimization algebra rules. Section 6 discusses the integration with view merging. Section 7 discusses performance experiments. Section 8 discusses related work. Section 9 discusses future direction and section 10 concludes the paper.

2. XML Query Motivating Examples

2.1 Querying an XML view constructed via SQL/XML function

Oracle XML DB enables users to create a view of XML type instances via SQL/XML publishing functions over relational tables. Consider a classic case of the *dept* and *emp* tables. The *dept* table keeps track of all the departments and *emp* table keeps track of all employees. The *deptno* column of the *emp* table is a foreign key referencing the *deptno* column of the *dept* table. The content of the *dept* and *emp* tables are shown in table 1 and table 2.

Deptno	Dname	loc
10	*ACCOUNTING*	*NEW YORK*
40	*OPERATIONS*	*BOSTON*

Table 1 - content of dept table

empno	ename	job	sal	deptno
7782	*CLARK*	*MANAGER*	*2450*	*10*
7839	*KING*	*PRESIDENT*	*5000*	*10*
7934	*MILLER*	*CLERK*	*1300*	*10*
7954	*SMITH*	*VP*	*4900*	*40*

Table 2 - content of emp table

To generate XML from the relational tables *dept* and *emp*, we create a view *dept_xmlview* as shown in table 3:

```
CREATE VIEW dept_xmlview AS
SELECT XMLElement("Department",
            XMLAttributes(deptno as "Deptno"),
   XMLElement("DeptInfo",
      XMLForest(dname as "DepartName", loc as "Location")),
    (SELECT XMLAgg(XMLElement("Employee",
                XMLAttributes(empno as "Empid"),
       XMLForest(ename as "EmpName", job as "Job, sal as "Salary")))
    FROM emp e
    WHERE e.deptno = d.deptno)) AS department
FROM dept d
```

Table 3- SQL/XML constructed XML view dept_xmlview

This view generates two rows of XMLType instances as shown in table 4. For each row in the *dept* table, it uses the SQL/XML standard publishing functions to construct an XMLType instance. The SQL query containing *XMLAgg* is a correlated scalar subquery that aggregates the XML information from the *emp* table. Thus, for each *dept* row, the relevant *emp* rows are retrieved and converted into a collection of *employee* elements.

```
<Department Deptno="10">
  <DeptInfo>
    <DepartName>ACCOUNTING</DepartName>
    <Location>NEW YORK</Location>
    </DeptInfo>
  <Employee Empid="7782">
    <EmpName>CLARK</EmpName>
    <Job>MANAGER</Job>
    <Salary>2450</Salary>
  </Employee>
  <Employee Empid="7839">
    <EmpName>KING</EmpName>
    <Job>PRESIDENT</Job>
    <Salary>5000</Salary>
  </Employee>
  <Employee Empid="7934">
    <EmpName>MILLER</EmpName>
    <Job>CLERK</Job>
    <Salary>1300</Salary>
  </Employee>
</Department>
```

```
<Department Deptno="40">
  <DeptInfo>
    <DepartName>OPERATIONS</DepartName>
    <Location>BOSTON</Location>
  </DeptInfo>
   <Employee Empid=7954>
    <EmpName>SMITH</EmpName>
   <Job>VP</Job>
   <Salary>4900</Salary>
  </Employee>
</Department>
```

Table 4 - Two rows of XMLType instances from *dept_xmlview*

Example 1: Consider an XML query shown in table 5 that finds all the rows in *dept_xmlview* where the Deptno

attribute is 10. This can be expressed in XPath as '*/Department[@Deptno=10*]'. The query further extracts the DeptInfo element for each such Department.

```
select Extract(department, '/Department/DeptInfo')
from dept_xmlview v
where ExistsNode(department, '/Department[@Deptno= 10]') = 1
```

Table 5 - Query dept_xmlview example 1

This query uses the *Extract* and *ExistsNode* functions with XPath to query the XML instances from *dept_xmlview*. These are Oracle XMLDB specific SQL functions that allow querying XML instances. *ExistsNode* checks for the existence of the node(s) targeted by the XPath expression and returns 1 or 0 depending on whether the XPath identifies any nodes in the document. The *Extract* function returns an XMLType consisting of the resultant nodes from applying the XPath expression on the source XML instance. It returns NULL if there are no nodes found by the XPath expression. The advantage of *ExistsNode* is that it allows short circuit evaluation if a node is found, whereas the *Extract* function has to find and return all matching nodes. The above query returns one row containing XMLType instance as shown in table 6:

```
<DeptInfo>
  <DepartName>ACCOUNTING</DepartName>
  <Location>NEW YORK</Location>
</DeptInfo>
```

Table 6 - Result from query example 1

A straightforward evaluation of the query first materializes the contents of *dept_xmlview* by constructing the XMLType instances and then evaluates the *ExistsNode* function by evaluating the XPath *'/Department[@Deptno=10]'*. If the *ExistsNode* returns 1, then it evaluates the *Extract* function by evaluating the XPath *'/Department/DeptInfo'*. An optimal evaluation plan, however, exploits the fact that the XPath *'/Department[@Deptno=10]'* maps to a predicate on the underlying *deptno* column of the *dept* table and the XPath *'/Department/DeptInfo'* maps to the XML node constructed by the function *XMLElement("DeptInfo", …).* The query can then be evaluated using a simple relational query shown in table 7. Note that this rewritten query does not contain any XPath functions.

```
select XMLElement(("DeptInfo",
      XMLForest(dname as "DepartName", loc as "Location"))
from dept
where deptno = 10
```

Table 7 - Rewritten query for query example 1

Oracle XML DB uses this optimal evaluation plan by rewriting the query in Example 1 to the query shown in Table 7 using the **XML Query Rewrite** technique. The rewritten query in table 7 is a relational query on the relational table and the standard relational optimizer can select the index on the *deptno* column of the *dept* table to speed up the query. The execution plan for example 1 query is shown below in table 8.

```
>SELECT STATEMENT
  >TABLE ACCESS BY INDEX ROWID  (DEPT)
    >INDEX UNIQUE SCAN   (INDEX ON DEPT.DEPTNO)
      >- access("DEPTNO"=10)
```

Table 8 - Execution Plan for query example 1

Example 1 query only involves XPath traversal on non-collection elements. Here we show another rewrite example where the XPath traversal involves a collection element *Employee*.

Example 2: Consider the XML query shown in table 9. It finds all XML instances in *dept_xmlview* where there exists a node satisfying the XPath expression *'/Department/Employee[@Empid=7839]')*. For such XML instances, it extracts the *'/Department/DeptInfo/Location'* node and attaches it to a new *Department* element node constructed by the *XMLElement* function.

```
select XMLElement("Department",
      Extract(v.department, '/Department/DeptInfo/Location'))
 from dept_xmlview v
where ExistsNode(v.department, '/Department/Employee[@Empid=
7839]') =1
```

Table 9 - Query dept_xmlview example 2

This XML query returns one row containing the XMLType instance shown in table 10:

```
<Department>
   <Location>NEW YORK</Location>
</Department>
```

Table 10 - Result from query example 2

Again, the optimal way to evaluate the query is to map the XPath predicate as a SQL predicate on the *empid* column of the *emp* table and map the XPath *'/Department/DeptInfo/Location'* to the *XMLForest* node on the *loc* column of the *dept* table. Oracle XMLDB also rewrites this query and evaluates it as a relational query as shown in table 11.

```
select XMLElement("DeptInfo",
   XMLForest(loc as "Location"))
from emp, dept
where empno = 7839 and emp.deptno = dept.deptno
```

Table 11 - Rewritten query for query example 2

We now show how to convert a forest of collection element nodes into a virtual SQL table using the *XMLSequence* table function and how they get rewritten.

Example 3: Here is an XML query to list the department name and the names of the employees that are in that department.

```
select extractValue(v.department, '/Department/DeptInfo/DepartName'),
      extractValue(value(v2), '/Employee/EmpName')
 from dept_xmlview v,
```

```
table(XMLSequence(Extract(v.department, '/Department/Employee'))) v2
```

Table 12 - Query dept_xmlview example 3

This query returns the set of rows shown in table 13.

ACCOUNTING	*CLARK*
ACCOUTING	*KING*
ACCOUNTING	*MILLER*
OPERATIONS	*SMITH*

Table 13 - Result from example 3 query

The ***Extract*value** function is similar to ***Extract*** but returns only scalar values. This is a type aware function and returns the appropriate type (NUMBER, DATE etc.) based on the XMLSchema type information, or the underlying SQL expression information, if available, at query compilation time. In this particular example all the title elements are simple strings, so the result is of type VARCHAR2.

XMLSequence is a SQL function that takes an XMLType containing a forest of XML element nodes and transforms it to a collection of XMLType instances. The TABLE function can then be used to convert the collection of XML into multiple rows. Multiple correlated TABLE functions containing XMLSequence and *Extract* expressions can be used to un-nest the hierarchical information contained in the XML document. The query is optimized into a simple relational query (shown in table 14) over the underlying relational tables contained in the *dept_xmlview*.

```
select dname, ename
from dept d , emp e
where d.deptno = e.deptno
```

Table 14 - Rewritten query for example 3 query

All previous query examples are on an XML view constructed via SQL/XML functions. We now show the examples of XMLType table storing XMLType instances conforming to an XML schema.

2.2 Querying schema based XML table examples

We first define an XML schema '*http://www.oracle.com/dept.xsd*' as shown in table 15 and have registered it with the Oracle XML DB. Then we create an XMLType table *dept_xmltab* containing XMLType instances that conform to the '*http://www.oracle.com/dept.xsd*' schema as shown in table 16. The table *dept_xmltab* table now contains the hidden columns designated as *deptid_hc, deptname_hc, deptloc_hc* for storing the *department id, name, location* values and the collection of employees are stored as a separate storage table *emp_col_tab*. The object relational infrastructure is used to create object types, collection types and subtypes to reflect the various XML Schema constructs. See reference [5] for more details on schema-based storage.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.oracle.com/dept.xsd" version="1.0"
xmlns:xdb="http://xmlns.oracle.com/xdb"
 elementFormDefault="qualified">
 <element name "Department">
  <complexType>
   <sequence>
    <element name = "DeptInfo">
     <complexType>
      <sequence>
       <element name = "DepartName" type = "string"/>
       <element name = "Location" type = "string"/>
      </sequence>
     </complexType>
    </element>
    <element name = "Employee" maxOccurs = "unbounded">
     <complexType>
      <sequence>
       <element name = "EmpName" type = "string"/>
       <element name = "Job" type = "string"/>
       <element name = "Salary" type = "positiveInteger"/>
      </sequence>
      <attribute name="Empid" type="positiveInteger"/>
     </complexType>
    </element>
   </sequence>
   <attribute name="Deptno" type="positiveInteger"/>
  </complexType>
 </element>
</schema>
```

Table 15 - dept.xsd XML schema

```
create table dept_xmltab of xmltype
    xmltype store as object relational
   xmlschema "http://www.oracle.com/dept.xsd" element "Department"
```

Table 16 - dept_xmltab schema based XML table

When XMLType instances are inserted into the *dept_xmltab* table, they are appropriately shredded and the values are inserted into the underlying hidden columns of *dept_xmltab* and *emp_col_tab*. The collection table internally has a hidden column *nested_table_id* to join with the hidden *set_id* column of the parent table *dept_xmltab*.

XML queries on the XMLType table *dept_xmltab* are optimized through the rewrite technique similar to those on *dept_xmlview* view as shown in examples 1, 2 and 3.

Example 4: The equivalent of the query in example 1 on the *dept_xmltab* and its rewritten counterpart are shown in table 17. For the rewritten query, *mkxml* is a SQL primitive operator converting an object type instance constructed by the object constructor (*ocons*) into XML.

```
select Extract(value(t), '/Department/DeptInfo',
  'xmlns="http://www.oracle.com/dept.xsd"')
from dept_xmltab t
where ExistsNode(value(t), '/Department[@Deptno = 10]',
  'xmlns="http://www.oracle.com/dept.xsd"') = 1
```

```
select  mkxml(ocons('deptInfo', t.deptname_hc, t.deptloc_hc),…)
from dept_xmltab t
```

```
where t.deptno_hc = 10
```

Table 17 – Query on dept_xmltab table example 4

Example 5: The equivalent of the query in example 2 on the *dept_xmltab* and its rewritten counterpart are shown in table 18.

```
select XMLElement("Department",
   Extract(value(t), '/Department/DeptInfo/Location',
      'xmlns="http://www.oracle.com/dept.xsd"'))
from dept_xmltab t
where ExistsNode(value(t), '/Department/Employee[@Empid = 7839]',
   'xmlns="http://www.oracle.com/dept.xsd"') =1
```

```
select  XMLElement("Department", mkxml(deptloc_hc,...))
from emp_col_tab col, dept_xmltab as t
where col. empid = 7839 and col.nested_tabl_id=t.set_id
```

Table 18 – Query on dept_xmltab table example 5

Example 6:. Equivalent of example 3 query on *dept_xmltab* and its rewritten query are shown in table 19.

```
select extractValue(value(t), '/Department/DeptInfo/DepartName',
       'xmlns="http://www.oracle.com/dept.xsd"'),
      extractValue(value(t2), '/Employee/EmpName',
       'xmlns="http://www.oracle.com/dept.xsd"')
from dept_xmltab t,
   table(XMLSequence(Extract(value(t),
                '/Department/Employee',
                'xmlns="http://www.oracle.com/dept.xsd"'))) t2
```

```
 select t.deptname_hc, col.empname
from dept_xmltab t, emp_col_tab col
where col.nested_table_id = t.set_id
```

Table 19 – Query on dept_xmltab table example 6

3. XML Query Rewrite Overview

As shown in the previous examples, the XML Query Rewrite technique rewrites XPath operations to XML data, which is physically stored relationally, to directly operate on the underlying data. This enables further optimizations by the classic relational optimizer in terms of optimal index access methods and join-order, and thus avoids the need to physically materialize the XML in memory.

The query rewrite happens at query compilation time. After a query passes through the parser, semantic analyzer and pre-type checking phases, it is internally represented as a query tree composed of query blocks and operator trees. We walk through all the query blocks to identify XPaths in every *Extract*, *Extractvalue* and *ExistsNode* function and convert them into SQL operator trees with possible subquery blocks. The query tree is then further optimized through view merging and subquery to join conversion and operator tree optimization. The resultant query tree is then given to the relational optimizer, which generates the execution plan for the execution engine. Figure 1 shows the logic flow of rewrite during the query compilation phases.

The key to implementing this query rewrite idea is to develop a new set of primitive SQL operators for XPath navigation and SQL/XML publishing functions. Some of the operators are directly exposed to the user while others are not. We have also developed a new set of algebra rules to optimize those primitive SQL operators. The new operators and their algebra rules are directly integrated with the existing relational and object relational algebra in Oracle. Since the backbone storage of the XML is relational or object relational, this approach leverages the existing relational and object relational algebra framework.

The XML Query rewrite technique consists of the following key modules:

- XML input analysis
- XPath Expansion based on input XML meta-data
- XPath step meta-data annotation
- XPath transformation to the SQL operator tree
- SQL/XML publishing function normalization
- Operator tree optimization based on a new set of XML operator algebra rules.

The technique also integrates with relational algebra rules for view merging, object relational algebra rules for object construction and attribute access, and collection view merging.

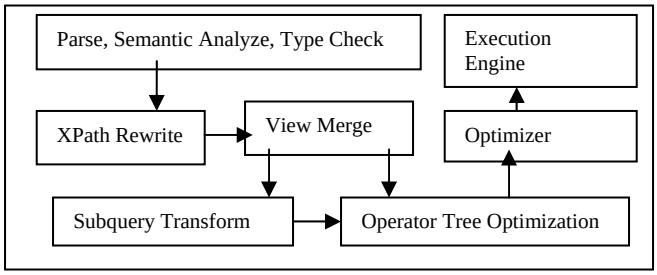

Figure 1 - Query Rewrite Logic Flow

4. XPath Rewrite

4.1 XML Input Analysis

During the compilation of a query, we examine the XML input to each XML Query function, *Extract*, *ExistsNode* and *Extract*value, and rewrite them if the underlying XML input data satisfies one of the following criteria:

- The input XML is stored in a schema-based XMLType table or column.
- The input XML is generated by an object view using an XML generation function that relies on the default mapping of object-relational data to XML.
- The input XML is generated from standard SQL/XML publishing functions, such as *XMLElement*, *XMLAgg*, and *XMLForest*.
- The input XML is from a view column, which is constructed on top of the above three cases.

We do not do rewrite if the underlying XML input structure is opaque to the query compiler, such as when the input XML is CLOB based storage or generated from arbitrary user defined functions returning XMLType. The idea is that we need to know the meta-data that describes the structure of the input XML in order to optimize the XPath operation on top of it. The **input XML meta-data** itself, on the other hand, can be very flexible - it can be an XML schema annotated with shredding information of the mapping object meta-data and object relational table, or merely object meta-data descriptor, or an arbitrary SQL/XML operator tree constructing the XML input. For an XML input whose structure is opaque, the user can still use a functional index or a text index to do query optimization. However, the discussion of optimization using a functional or text index is beyond the scope of this paper.

4.2 XPath Expansion based on input XML meta-data

For each XPath that is a compile time constant used in the *Extract*, *ExistsNode* and *Extractvalue* functions, we expand the XPath if it contains wildcard character matching and double slash abbreviation based on the meta-data information of the input XML that we gathered from the input XML analysis . We expand the double slash into its constituents. For example, an XPath *'/a//d'* can be rewritten as *'/a/b/c/d'* if we know from the meta-data that all the possible elements between element '*a*' and element '*d*' is element '*b*' followed by element '*c*'. We also expand wildcard character '*' based on meta-data information. For example, the XPath *'/a/b/*/d'* is rewritten to *'/a/b/c/d'*.

4.3 XPath Step Meta-data Annotation

After the expansion of the XPath, we annotate each step of an XPath with the meta-data from the underlying input XML meta-data. The meta-data of an XPath element may contain the object attribute meta-data to which this element is mapped or a sub-tree of the SQL/XML operator tree from which this element is constructed.

The XPath step meta-data also indicates whether this step is feasible. For example, for an XPath *'a/b'*, if '*b*' is not a possible child element of element '*a*' based on the meta-data, then the element '*b*' step is marked as a not feasible step. A non-feasible step is transformed to the SQL NULL constant in the subsequent 'XPath Transformation to SQL operator tree' section.

We annotate each element step with the element cardinality information. This information can be used by other XPath transform functions to determine if the element is a collection or scalar element. For example, for an XPath step that matches the element '*employee*', if the maximum occurrences property for that element in the XML schema is more than 1 or it maps to a collection object type, then element '*employee*' is a collection element. With neither an XML schema nor an object descriptor, we can derive this information from the input SQL/XML operator tree. For example, if the '*employee*' element is constructed by applying *XMLAgg* to a set of rows as a query *'select XMLAgg(XMLElement("employee"…)) from emp'* or is constructed from an *XMLConcat* expression, such as *'XMLConcat(XMLElement("employee" ,'Miller'), XMLElement("employee", 'Smith'))'*, then the element is a collection element.

4.4 XPath Transformation to SQL operator tree

After we finish the annotation of meta-data of each XPath step, we transform the entire XPath into a SQL operator tree. The transformation engine walks through the input XPath expression and recursively applies the following rewrite functions depending on the node type of the XPath expression tree. The rewrite functions are

- Rewrite of XPath steps
- Rewrite of XPath predicates
- Rewrite of XPath expressions, such as relation operators, logical operators, arithmetic operators and XPath built-in functions.

4.4.1 Rewrite of XPath Steps

We convert each step of element or attribute extraction into a primitive SQL operator that extracts that node out of the underlying construction XML operator. If the element is a collection element, we further expand the result of the extraction into a subquery block, which represents the selection of a logical table formed by the conversion of collection elements into relational rows. The purpose of transforming XPath steps into a SQL operator tree along with subquery blocks is that when they are applied to the input XML construction SQL operator tree by the operator tree optimization and view merge module, many operators can be cancelled so that an optimal operator tree is derived.

The SQL extraction operators are primitive SQL operators, *XATG* and *OATG*. If the element is constructed from a SQL/XML operator tree, then *XATG* is used. For example, *Extract(XMLElement("foo", 3), './foo')* is transformed into *XATG(XMLElement("foo", 3), '.', 'foo')*, which can be further optimized by the operator tree optimization module into just *XMLElement("foo", 3)*. *XATG* is a primitive SQL operator not exposed to the user. Semantically, *XATG(xmltype, '.', 'A')* is equivalent to *Extract(xmltype,'A')* and *XATG(XATG(xmltype, '.', 'A'), 'A', 'B')* is equivalent to *Extract(xmltype, A/B')*.

If the element is constructed from object relational data, then the *OATG* operator is used. For example,
Extract(MKXML(OCONS('obj', attr1Val, attr2Val)), 'attr2')
is transformed into
(MKXML(OATG(OCONS('obj', attr1Val, attr2Val), 'attr2')).

This can be further optimized by the operator tree optimization module into just *MKXML(attr2Val)*. Here the *OCONS* is an object constructor SQL operator which constructs an object of type 'obj' containing two attributes 'attr1' and 'attr2' and the *MKXML* is a SQL operator which converts an object value instance into an XMLType instance value. Details of these SQL operators, such as *MKXML*, *OCONS* and *OATG*, are discussed in Section 5.4.

For the extraction of collection elements, we also form a subquery block that represents a selection from a logical table. For schema based XML or XML formed by object relational data, the logical table is either the physical storage table for the collection object if the collection object is stored out of line or constructed via conversion of a collection object type instances into a tuple stream if the collection object is stored inline with the main table. For SQL/XML generated XML, the logical table is mapped to the result of the SQL query that contains the *XMLAgg*. This subquery block can then be view merged into the parent query block recursively through the relational view merge and collection view merge process. Consequently, this logical table does not have to be materialized at run time.

4.4.2 Rewrite of XPath Predicates

XPath predicates are transformed into SQL predicates. In the case of predicates on collection elements, the SQL predicate created is attached to the subquery created corresponding to the collection element. This effectively pushes down the predicate to the source table so that the relational optimizer can do its optimization job.

XPath predicates on scalar elements are transformed into SQL CASE expressions where are further optimized using algebra rules that are applied by the operator tree optimization module.

4.4.3 Rewrite of XPath Expressions

We transform XPath operators, such as logical operators, arithmetic operators, relational operators, etc, into their equivalent SQL operators. We also transform XPath built-in functions into their equivalent SQL functions. Most of these transformations are straightforward. The principle here is that if we do not find an equivalent SQL operator, we can create one that implements the semantics required by the XPath. This is an advantage of rewriting these XML queries inside the database server since we can always extend it with new SQL operators. It also enables us to expose PL/SQL and other SQL functions as XPath functions.

The XPath comparison operators, such as equality, greater-than, lesser-than etc., have existence semantics if either of their arguments are collection elements. From the XPath step meta-data, we know if the step results in a collection element or not. So, for an XPath predicate containing the comparison operator, such as '*[b = c]*', we transform it differently depending on the element cardinality.

- If both element '*b*' and element '*c*' are scalar elements, then this is transformed into the equivalent SQL '=' operator.
- If both element '*b*' and element '*c*' are collection elements, then this is transformed into an EXISTS SQL operator with a subquery block testing the existence relationship. So it is transformed into

```
EXISTS (select null   from  collection_b
        where EXISTS (
           select null from collection_c
           where collection_b.column_value =
                 collection_c.column_value).
```

- If one element is a scalar and the other element is a collection, (say *'b'* is scalar element), then it is transformed into

```
EXISTS(select null from collection_c
       where collection_c.column_value = b).
```

For input to logical operators, such as AND and OR, we again convert the subquery input which represents collection elements into *EXISTS* SQL operator representing existential check.

Rewriting into exists subquery blocks enables further transformation into semi-joins by the regular relational subquery transformations.

After an XPath is transformed into SQL operators for the *Extract* function, the rewrite is done. For *ExistsNode*, we apply a final SQL operator to compute the effective boolean value of the result XML. If the result XML is a collection element node, then we apply *EXISTS* SQL operator, otherwise, we apply the SQL *CASE* operator. For *ExtractValue*, we apply a final SQL operator that atomizes the node by extracting the text value and cast it into an appropriate SQL type.

We also handle the rewrite of query over XML with XML schema having Choice, substitution groups and inheritance construct by exploiting object type inheritance, SQL CASE and TREAT operators. However, the discussion of such is beyond the scope of this paper.

5. Operator Tree Optimization

Algorithmically, operator tree optimization is an algebraic transformation of the operator tree by applying a set of algebraic rules for each operator node in the tree. We do this recursively, from bottom up, so that an optimal operator tree is derived. Optimization arises from the elimination of unnecessary operator nodes and overall

simplification in the operator tree. The extraction and generation operators can be cancelled when the two operators are the inverse of each other. Operator nodes of the XML generation tree that are not selected by the XPath are also eliminated. Consequently, the operator tree is reduced down to a minimal set of operator nodes that are needed to compute the query result.

For each SQL operator, we define a set of algebraic rules to optimize the operator. The algebraic rules fall into the following categories:

- **Nullification rule:** The nullification rule is cost-reductive because it simplifies the operator if any argument to the operator is SQL NULL. The nullification rule usually simplifies the operator by replacing it with NULL, if all the arguments are NULL, or by eliminating any arguments that are NULL.
- **Elimination rule:** This rule is cost-reductive because it eliminates unnecessary intermediate operators.
- **Distribution rule**: This rule applies an operator *f* to another operator *g* by distributing operator *f* to all the children of operator *g*.
- **Cancellation rule:** This rule cancels two operators that are the inverse of each other.
- **Normalization rule:** Normalization does not reduce the number of SQL operators in the operator tree, however it canonicalizes all the syntax exposed to the user to the low-level operators so that fewer algebra rules need to be added into the system. Thus, the operator tree optimization needs to deal with fewer possible cases of input operators. This is especially useful when a language provides a large number of high-level syntactic variations to the users.

5.1 SQL/XML publishing function normalization

We normalize SQL/XML publishing functions into a set of primitive SQL operators that generate XML. Normalization is to transform user-exposed functions, such as a SQL/XML publishing function, into an operator tree composed of primitive SQL operators. The higher-level operators can be viewed effectively as syntactic sugar for the primitive operators.

Consider the SQL/XML function *XMLForest*. *XMLForest* examines each input argument expression, and if the argument expression is not null, then a new *XMLElement* node is formed with the input tag name, otherwise, the result is null. The rest is *XMLConcat* applied on each of the new *XMLElement* nodes. For example, the expression *XMLForest(name as "DeptName", location as "DeptLocation")* generates the following XML fragment if *name* column value is '*Engineering*' but *location* column value is NULL: *<DeptName>Engineering</DeptName>*.

The above *XMLForest* expression can be syntactically transformed into the following equivalent:

```
XMLConcat(
  CASE WHEN name IS NOT NULL, THEN
    XMLElement("DeptName", name) ELSE NULL END,
  CASE WHEN location IS NOT NULL THEN
    XMLElement("DeptLocation", location) ELSE NULL END)
```

5.2 New SQL/XML Algebra Rules

In the examples below, we designate *XE*, *XC*, *XAG*, *CS*, and *INN* to represent the SQL functions *XMLElement*, *XMLConcat*, *XMLAgg*, *CASE WHEN*, and *IS NOT NULL* respectively. We use *"T" and "U"* to represent XML element tag name, *e* to represent an expression, and *c* to represent conditional expression. For example, $XMLElement("T", e_1, .., e_n)$ is represented as $XE("T", e_1, .., e_n)$. $CASE\ WHEN\ c_1\ THEN\ e_1\ WHEN\ c_2\ THEN\ e_2\ ELSE\ e_d\ END$ is represented as $CS(c_1, e_1, c_2, e_2, e_d)$.

Note that the list in this section is not exhaustive – it is a subset, and is meant to provide a flavor of the algebraic rules used in Oracle XML DB.

The second column of the tables of algebraic rules uses the following abbreviations for the classification of algebraic rules: N: Nullification Rule; E: Elimination Rule; D: Distribution Rule; O: Normalization Rule

5.2.2 Algebraic rules for *XMLElement*

$XE("T", NULL) = XE(T)$	N1	*If the input argument to XMLElement is SQL NULL itself, then it merely creates an element with empty content*
$XE("T", e_1, .., NULL, .. e_n) = XE("T", e_1, \ldots e_n)$	N2	*If any of the input argument to XMLElement is NULL, then that argument can be eliminated from the input*
$XE("T", e_1, .., XC(e_i, .., e_j), . e_n) = XE("T", e_1, .., e_i, .., e_j, .. e_n)$	E1	*If any input argument to XMLElement is XMLConcat, then all the arguments to XMLConcat can be merged into the parent XMLElement as its argument. The intermediate XMLConcat operator is eliminated.*

5.2.3 Algebraic rules for *XMLForest*

$XMLFOREST(e_1 \text{ as } "T_1", .., e_n \text{ as } "T_n") = XC(CS(INN(e_1), XE("T_1", e_1), NULL), .., CS(INN(e_n), XE("T_n", e_{n0}), NULL))$	O1	*Normalize XMLForest into XMLConcat of CASE of XMLElement*

5.2.4 Algebraic rules for *XMLConcat*

$XC(NULL) = NULL$	N1	*If the input argument t to XMLConcat is SQL NULL itself, then the output is SQL NULL.*
$XC(e_1, .., NULL, .. e_n) = XC(e_1, .., e_n)$	N2	*If any of the input arguments to XMLConcat is NULL, then that argument can be eliminated from the input*
$XC(e_1, .., XC(e_i, .., e_j), .., e_n) = XC(e_1, .., e_i, .., e_j, .., e_n)$	E1	*If any input argument to XMLConcat is XMLConcat, then all the arguments to XMLConcat can be merged into the parent XMLConcat as its argument. The intermediate XMLConcat function is eliminated.*
$XC(e) = e$	E2	*XC with single input argument can be*

		eliminated.

5.2.5 Algebraic rules for the XATG operator

XATG(NULL, "T", "U") = NULL	N1	*If the input argument t to XATG is SQL NULL itself, then the output is SQL NULL.*
XATG(XE("T", e), '.', 'T') = XE("T", e)	E1	*This rule eliminates the XATG operator.*
XATG(XE("T_1", e), 'T_1', 'T_2') = XATG(e, '.', 'T_2')	E2	*This rule eliminates the XMLElement function. "e" is assumed to be an XMLType*
XATG(XE("T", e), '.', 'T_1'= NULL	E3	*This rule eliminates the XATG and XE operators because tag 'T' and 'T_1' does not match*
XATG(XC(e_1, .., e_n), 'T', 'U') = XC(XATG(e_1, 'T', 'U'), .., XATG(e_n, 'T', 'U'))	D1	*This rule distributes XATG operator to all the arguments to XMLConcat function.*
XATG(CS(c_1, e_1, .., c_n, e_n, e_d), 'T', 'U') = CS(c_1, XATG(e_1, 'T', 'U'), .., c_n, XATG(e_n, 'T', 'U'), XATG(e_d, 'T', 'U'))	D2	*This rule distributes XATG operator to all the branches of a CASE operator.*
XATG(XAGG(e), 'T', 'U') = XAGG(XATG(e, 'T', 'U'))	D3	*This rule distributes XATG operator to the argument of XAGG.*

5.2.6 Algebraic rules for IS NOT NULL (INN) expression

INN(NULL) = false	N1	*If the input argument t to INN is SQL NULL itself, then the output is false.*
INN(not_null_e) = true	E2	*If the input argument to INN is a not null expression (such as a non-nullable column), then the output is true.*

5.2.7 Algebraic rules for CASE expressions (CS)

CS(true, e_1, e_2) = e_1	E1	*Eliminates the CS operator when the case value is known to be true.*
CS(false, e_1, e_2) = e_2	E2	*Eliminates the CS operator when the case value is known to be false.*
CS(c, e, e) = e	E3	*Eliminates the CS operator when the branched expressions are equivalent.*

5.3 Example of operator tree optimization by application of algebraic rules

Assume *colb* is a nullable column and *colc* is a non-nullable column. The following is an example to show how *Extract(XMLElement("a", XMLForest(colb as "b", colc as "c")), './a/c')* is optimized to just *XMLElement("c", colc)* by applying the algebraic rules.

- Applying normalization of *XMLForest* and transformation of XPath into XATG operators yields:
  XATG(XATG(XE("a", XC(CS(INN(colb), XE("b", colb), NULL), CS(INN(colc), XE("c", colc), NULL))), '.', 'a'), 'a', 'c')
- Applying INN-E2 and CS-E1 (note *colc* is not a nullable column) yields:

  XATG(XATG(XE("a", XC(CS(INN(colb), XE("b", colb), NULL), XE("c", colc))), '.', 'a'), 'a', 'c')
- Applying XATG-E1 to eliminate the inner XATG yields:

  XATG(XE("a", XC(CS(INN(colb), XE("b", colb), NULL), XE("c", colc))), 'a', 'c').
- Applying XATG-E2 to eliminate the outer XE yields:

  XATG(XC(CS(INN(colb), XE("b", colb), NULL), XE("c", colc)), '.', 'c')
- Applying XATG distribution rules XATG-D1 and XATG-D2, and XATG-N1 yields:

  XC(CS(INN(colb), XATG(XE("b", colb), '.', 'c'), NULL), XATG(XE("c", colc), '.', 'c'))
- Applying XATG-E3 to eliminate the first XATG and XATG E1 rule to eliminate the second XATG yields:

  XC(CS(INN(colb), NULL, NULL), XE("c", colc))
- Applying CS-E3 yields:

  XC(NULL, XE("c", colc))
- Applying XC-N2 yields the final optimal tree:

  XE("c", colc)

5.4 Integration with Object Relational Algebra Rules

Oracle uses a set of algebraic rules to optimize object operations. These rules can be used seamlessly with the XML algebraic rules to perform XML optimizations in the presence of object operands, and object optimizations in the presence of XML.

Object operators in Oracle include OCONS for object construction, and OATG for attribute access. As an example of object algebraic optimization, consider the following rule :
OATG(OCONS('obj', attr1Val, attr2Val), 'attr2') = attr2Val .
This rule states that to get '*attr2*' attribute of an object constructed with *attr1* having value *attr1Val* and *attr2* having value *attr2Val*, the result is just *attr2Val*.

In order to integrate with XML, two new operators *MKXML* and *UMKXML* have been developed. *MKXML* converts an object instance into an XMLType instance and *UMKXML* converts an XMLType instance back into an object instance. *MKXML* and *UMKXML* are the inverses of each other.

Algebraic rules are used to specify the transformation of XPath steps over XML constructed from object instances – this will results in a tree with *OATG* over *OCONS*. Consider the following SQL expression:
extractValue(MKXML(OCONS('obj', attr1Val, attr2Val)), 'attr2'),
This is transformed into an equivalent SQL expression :
OATG(UMKXML(MKXML(OCONS('obj', attr1Val, attr2Val))), 'attr2'),
This is optimized by the operator tree optimization into just the simple *attr2Val* SQL expression..

6. Integration with relational view Merging

Relational view merge merges a query or a view definition in the *FROM* clause into the main query. For

example, the following query *select * from (select * from t)* may be optimized into *select * from t.*

In an object-relational system, an XPath query over a collection column, with or without a predicate, (e.g. *'/Department/Employee[EmpName = 'CLARK']'*)
is converted into a subquery selecting from a logical table:

```
select * from table(cast(multiset(
    select * from table where pred) as collectionType))
```

Here *tab* is the underlying storage table for the instances of *collectionType*. The predicate *pred* is present only if the XPath query has a predicate. Collection view merge cancels the *table* function with the *cast(multiset(query) as collectionType)* operation, leaving the query as

```
select * from (select * from tab where pred).
```

This is then further optimized via relational view merge into the simple query :

```
select * from tab where pred.
```

Alternatively, in a relational system, collections may be constructed using the *XMLAgg* function to aggregate XMLType values. In this case, the rewritten query for an XPath such as *'/Department/Employee[EmpName='JO']* is of the form

```
select xmlagg(v.column_value)
 from table(XMLSequence(
   select xmlagg(XMLElement(..)) from t where pred))) v
```

This query can be effectively transformed into the following form:

```
select xmlagg(v.column_value)
 from table(cast(multiset(select XMLElement(...) from t where pred)
    as xmlsequenceType) v,
```

Note that the *XMLSequence* over *XMLAgg* has been transformed into a *cast(multiset()).* Here, *XMLSequenceType* represents an array of XMLType. Collection view merge then optimizes the query to :

```
select xmlagg(v.column_value) from(select XMLElement(…) from t
where pred) v
```

Relational view merge then optimizes the query to:

```
select xmlagg(XMLElement(…)) from t where pred.
```

Through the relational and collection view merge, the query over the underlying storage table or view constructing the collection elements is folded into the parent query. The predicates on the collection elements automatically become the predicates on the underlying collection storage table or view. This effectively pushes the predicate down, and various access methods can be better exploited. No run-time materialization of the collection elements is needed.

7. Performance

To measure the performance of XML query rewrite over SQL/XML viewed over relational data, we create the SQLX-Bucky benchmark based on Bucky[10] benchmark. We use SQL/XML publishing functions to create XMLType views over relational tables. To measure the performance of XML query rewrite over schema based XML table, we use XMark[14]. In both benchmarks, we express the query using *Extract, ExistsNode, ExtractValue,* XMLSequence and SQL/XML publishing functions.

The performance objectives are two-fold. The first is to compare the performance of rewritten XML queries with the performance of the same query without rewrite. Without query rewrite, XML needs to be materialized followed by XPath evaluation. The performance of rewritten queries, however, scales gracefully similar to that of relational queries. Rewritten queries are *orders of magnitude* faster than non-rewritten queries since they can use indexes.

Our second objective is to compare the performance of the XML queries against their semantically equivalent object relational or relational queries combined with SQL/XML publishing functions. We find that the performance of the two is comparable for both XMark and SQLX-Bucky benchmark. Figure 2 and Figure 3 show the ratio of the query performance using query rewrite to the semantically equivalent relational or object relational query written directly over the underlying storage tables. This demonstrates that query over XML combined with rewrite yields performance comparable to that of queries directly on the underlying data.

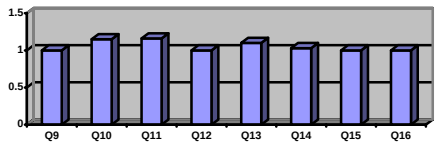

Figure 2 – Query Speed Ratio for SQLX-Bucky

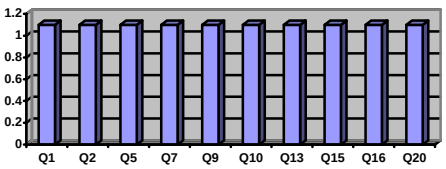

Figure 3 – Query Speed Ratio for XMark

8. Related Work

Many of the concepts presented in this paper have been studied in other contexts. XML algebra [6][7] and optimizing queries of XML views of relational data in the middleware [8][9], in particular, have been the subject of much research.

Our work is unique in the following respects. First, our query optimization rules are based on optimizing XPath expressions over SQL/XML and object relational SQL. Second, query processing is performed inside a popular, commercial database server, as opposed to non-integrated mid-tier solutions. Our solution does not materialize large

volumes of XML in the middleware. We primarily optimize queries over XML whose underlying storage is relational and object-relational. Third, our algebraic rules for XML processing and optimization is tightly integrated with existing relational and object-relational rules. This enables optimization involving a mix of relational and XML queries.

Since the majority of business data is stored in relational and object-relational database, Oracle XML DB focuses on a practical subset of XML querying problems that, we believe, are the most useful for customers. We bridge the relational and XML worlds within Oracle XML DB by leveraging the relational and object relational algebra, and its optimization infrastructure.

9. Future Direction

As XQuery [1] becomes the standard to query XML, and the SQL/XML standard embraces XQuery functionality, Oracle XML DB will optimize XQuery over XML data. The XML Query Rewrite techniques presented in this paper has laid out the groundwork to fully optimize XQuery over XML stored object-relationally or generated by SQL/XML functions from relational data. We will discuss this in our future paper

XML data can be recursive. Such XML can be constructed using the Oracle *CONNECT BY* expression and hierarchical XML generation methods. The rewrite of queries over such recursive constructs has scope for future investigation.

10. Conclusion

In this paper, we have focused on a technique of optimizing queries on XML whose underlying storage is relational or object-relational. The idea is to transparently transform the XML query into its equivalent relational or object-relational equivalent through query rewrite techniques at compile time, so that a classic optimizer can further optimize it and a tuple-oriented execution engine can efficiently execute it. We create a set of new SQL operators for XPath navigation, and incorporate a new set of algebra rules for SQL/XML operators with existing relational and object relational algebra rules in the Oracle database server. Our experience has shown that this technique enables customers to leverage their existing relational and object relational systems, and to provide interoperability between XML and their existing data and applications.

Acknowledements

We gratefully acknowledge the contributions of all the members of the Oracle XML DB development and product management teams. We would especially like to thank *Ravi Murthy* and *Muralidhar Subramanian*, who have given us valuable and insightful ideas for query rewrite.

References

[1] World Wide Web Consortium, “XQuery 1.0: An XML Query Language”, W3C Working Draft, November 2003.

[2] Oracle XML DB Developer’s Guide: Oracle 9iR2. See http://otn.oracle.com/tech/xml/xmldb

[3] Database Languages – SQL - Part 14: XML Related Specifications (SQL/XML) – Aug 2003

[4] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, Ravi Murhty: “Oracle 8i – The XML Enabled Data Management System”, ICDE 2000.

[5] Ravi Murthy, Sandeepan Banerjee: “XML Schemas in Oracle XML DB”. VLDB 2003

[6] Flavius Frasincar, Geert-Jan Houben, Cristian Pau: “XAL: an Algebra for XML Query Optimization”. In ADC 2002, Melbourne, Australia, 2002, ACS

[7] H.V. Jagadish, Laks V.S. Lakshmanan, Divesh Srivastava, Keith Thompson: “TAX A Tree Algebra for XML”. In DBPL 2001.

[8] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene Shekita, Catalina Fan, John Funderburk: “Querying XML Views of Relational Data”. VLDB 2001.

[9] Mary Fernandez, Atsuyuki Morishima, Dan Suciu: “Efficient Evaluation of XML Middleware Queries”, SIGMOD Conf., May 2001

[10] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes E. Gehreke, Dhaval N. Shah: “The Bucky Object Relational Benchmark”,0 http://www.cs.wisc.edu/~naughton/bucky.html.

[11] The W World Wide Web Consortium, “XML Schema Standard”, see http://www.w3.org/XML/Schema

[12] Andrew Eisenberg and Jim Melton: SQL/XML and the SQLX Informal Group of Companies, , ACM SIGMOD Record, Vol. 30 No. 3, Sept. 2001, http://www.acm.org/sigmod/record/issues/0109/standards.pdf

[13] Andrew Eisenberg and Jim Melton: SQL/XML Is Making Good Progress” http://www.acm.org/sigmod/record/issues/0206/standard.pdf

[14] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioanna Manolescu, Ralph Busse: “Xmark: A Benchmark for XML Data Management” http://www.csd.uch.gr/~hy561/Papers/XMark-vldb02.pdf

Appendix Benchmark Query

For reference, we list a sample of XMark queries and SQLX-Bucky queries that we use for our performance experiments.

XMark Benchmark

The XMLdata is stored in a schema based XMLType table *site_tab*, with the XML schema derived from the XMark Internet auction site.

Q1: Return the name of the person with ID ‘person0’:

```
select extract(value(v), '/person/name')
from site_tab v0,
    table(xmlsequence(extract(value(v0), '/site/people/person'))) v
where extractValue(value(v), '/person/@id') = 'person0'
```

Q5: How many sold items cost more than 40:

```
select count(*)
from (select extract(value(v), '/closed_auction/price')
     from site_tab v0, table(xmlsequence(extract(value(v0),
                  '/site/closed_auctions/closed_auction'))) v
 where extractValue(value(v), '/closed_auction/price') >=40 )v
```

Q7: How many pieces of prose are in our database ?:

```
select xmlelement("cnt",
  (select count(*)
    from table(xmlsequence(extract(value(v), '/site//description')))) +
  (select count(*)
    from table(xmlsequence(extract(value(v), '/site//annotation')))) +
    (select count(*)
     from table(xmlsequence(extract(value(v), '/site//email'))))
from site_tab v0, table(xmlsequence(extract(value(v0), '/site'))) v
```

Q9: List the names of persons and the names of the items they bought in Europe:

```
select xmlelement("person",
    xmlattributes(extractValue(value(p), '/person/name') as "name"),
    (select xmlagg(xmlelement("item",
         (select xmlagg(extract(value(t2), '/item/name'))
          from site_tab v000,
           table(xmlsequence(extract(value(v000),
                  '/site/regions/europe/item'))) t2
         where extractValue(value(t),
                 '/closed_auction/itemref/@item')=
              extractValue(value(t2), '/item/@id') )))
     from site_tab v00, table(xmlsequence(extract(value(v00),
                  '/site/closed_auctions/closed_auction'))) t
     where extractValue(value(p), '/person/@id' )=
            extractValue(value(t),'/item/buyer/@person') ) )
from site_tab v0,
    table(xmlsequence(extract(value(v0), '/site/people/person'))) p
```

Q13: List the names of items registered in Australia along with their descriptions:

```
select xmlelement("item",
    xmlattributes(extractValue(value(i), '/item/name/text()') as "name"),
      extract(value(i), '/description') )
from site_tab v0,  table(xmlsequence(extract(value(v0),
                  '/site/regions/australia/item'))) i
```

Q20: Group customers by their income and output the cardinality of each group:

```
select xmlelement("result",
    xmlelement("preferred",
      (select count(*) from site_tab v,
          table(xmlsequence( extract(value(v),
            '/site/people/profile[@income >= 100000]'))))),
    xmlelement("standard",
        (select count(*) from site_tab v,
           table(xmlsequence( extract(value(v),
                  '/site/people/profile[@income < 100000
           and @income >= 30000]'))))),
    xmlelement("challenge",
        (select count(*) from site_tab v,
           table(xmlsequence( extract(value(v),
            '/site/people/profile[@income < 30000]'))))),
    xmlelement("na",
        (select count(*) from site_tab v
         where existsNode(value(v), 'site/people/person/@income') = 0
              )))   from dual
```

SQLX-Bucky Benchmark

Relational tables are created to hold base data and SQL/XML views are created on the relational tables.

Q1: Find the address of the staff member with id 6966:

```
select extractvalue(staff, '/ROW/NAME') name,
       extractvalue(staff, '/ROW/ADDRESS/STREET') street,
       extractvalue(staff, '/ROW/ADDRESS/CITY') city,
       extractvalue(staff, '/ROW/ADDRESS/STATE') state,
       extractvalue(staff, '/ROW/ADDRESS/ZIPCODE') zip
from Staff_sqlxv e
where extractvalue(staff, '/ROW/SSN') = 6966;
```

The *Staff_sqlxv* is a SQL/XML view created on top of the *rf_person* table as*:*

```
Create  View Staff_sqlxv AS
    SELECT XmlElement("ROW",
      XmlElement("SSN", id), XmlElement("NAME", name),
      XmlElement("ADDRESS",    XmlElement("STREET", street),
        XmlElement("CITY", city),   XmlElement("STATE", state),
        XmlElement("ZIPCODE", zipcode)),
      XmlElement("BIRTHDATE", birthdate),
      XmlElement("KIDNAMES",
       (select XMLAgg(XmlElement("CHLDNAME", kidname))
        from rf_Kids k      where k.id = p.id)),
      XmlElement("PICTURE", picture),
      XmlElement("PLACE", XmlElement("LATITUDE", latitude),
                XmlElement("LONGITUDE", longitude)),
                XmlElement("DATEHIRED", DateHired),
                XmlElement("STATUS", status),
               XmlElement("WORKSIN", worksin),
              XmlElement("ANNUALSALARY", annualSalary)) as staff
    FROM rf_PersonFlat p
    WHERE p.type=10;– type code for staff in table rf_person.
```

Q8: Find all staff whose children are named “girl16” and “boy16”:

```
select distinct extractvalue(e.staff, '/ROW/NAME') name,
       extractvalue(e.staff, '/ROW/ADDRESS/STREET') street,
       extractvalue(e.staff, '/ROW/ADDRESS/CITY') city,
       extractvalue(e.staff, '/ROW/ADDRESS/STATE') state,
       extractvalue(e.staff, '/ROW/ADDRESS/ZIPCODE') zip
from Staff_sqlxv e,
   TABLE(xmlsequence(extract(e.staff,
         '/ROW/KIDNAMES/CHLDNAME'))) k1,
    TABLE(xmlsequence(extract(e.staff,
         '/ROW/KIDNAMES/CHLDNAME'))) k2
where  extractvalue(value(k1), '/CHLDNAME') = 'girl16'
and extractvalue(value(k2), '/CHLDNAME') = 'boy16'
```