

# Efficient XSLT Processing in Relational Database System

Zhen Hua Liu, Agnel Novoselsky

Oracle Corporation

400, Oracle Parkway

Redwood Shores, CA 94065

U.S.A

{Zhen.Liu , Anguel.Novoselsky}@oracle.com

## ABSTRACT

Efficient processing of XQuery, XPath and SQL/XML on XML documents stored and managed in RDBMS has been widely studied. However, much less of such type of work has been done for efficient XSLT processing of XML documents stored and managed by the database. This is partially due to the observation that the rule based template driven XSLT execution model does not fit nicely with the traditional declarative query language processing model which leverages index probing and iterator based pull mode that can be scaled to handle large size data. In this paper, we share our experience of efficient processing of XSLT in Oracle XML DB. We present the technique of processing XSLT efficiently in database by *rewriting XSLT stylesheets* into highly efficient XQuery through *partially evaluating* XSLT over the XML documents structural information. Consequently, we can leverage all the work done for efficient XQuery/XPath processing in database to achieve combined optimisations of XSLT with XQuery/XPath and SQL/XML in Oracle XMLDB. This effectively makes XSLT processing scale to large size XML documents using classical declarative query processing techniques in DBMS.

## 1. Introduction

XMLType has become a native data type in RDBMS. Users can create XMLType tables or XMLType columns to store XML documents. XMLType values can be generated from relational data via SQL/XML standard generation functions (such as XMLElement(), XMLAgg()) so that XMLType views over relational data can be created. XMLType can be queried using XQuery/XPath embedded in SQL/XML standard query functions (such as XMLQuery(), XMLExists()) and extract(), existsNode()

---

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12–15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

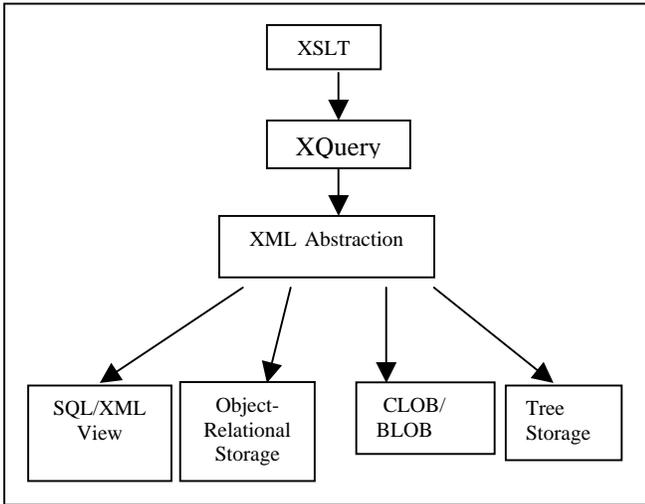
and extractValue() extension functions from Oracle XMLDB[1,2]. Furthermore, efficient processing of XQuery and XPath in RDBMS through XPath/XQuery native rewrite and indexing techniques has been well studied and applied in industrial settings [3, 4, 11, 12, 17, 18]. However, little of such type of effort has been applied to XSLT [5] transformation on XML documents stored in or generated from database.

Since the release of Oracle 9i, Oracle XMLDB has also supported XMLTransform() SQL/XML function that enables user to apply XSLT transformation on XMLType values [2, 16]. However, currently XSLT processing in XMLTransform() is evaluated functionally. That is, the XSLT processor views the input XMLType document as a DOM tree and uses the XMLType DOM API to perform the transformation, without taking advantage of how the input XML document is stored, indexed or generated in the database, nor does it take advantage of schema or DTD information to which the input XML documents conform. Intuitively, if we know how the XMLType is stored, indexed or generated in the database during query compilation time, then we should be able to use a similar XQuery/XPath native query rewrite and compilation technique to efficiently process XSLT without functionally evaluating XMLTransform().

The technique we will discuss in this paper we call *XSLT rewrite*. That is, we rewrite XSLT stylesheet into highly efficient XQuery query by *partially evaluating* [14, 15] XSLT over the input XML document structural information. Then we leverage the XQuery/XPath *native rewrite* techniques [3, 4, 12] to efficiently execute the result XQuery/Xpath query with the input XMLType values. That way, the XSLT transformation can be done efficiently in RDBMS by fully leveraging the underlying storage and index structures of the input XMLType values.

One of the main design philosophies of Oracle XMLDB query processing is to treat XQuery, XPath, XSLT, SQL/XML as different XML processing languages, which however, are compiled into the same internal representation (SQL extended with XML operators) and which are executed on the underlying Oracle *SQL/XML engine*. Thus we promote the feasibility of *cross language global*

*optimisations* among these XML query and transformation languages [4, 11]. This approach enables us to achieve global optimisations crossing all XML query and transformation languages over variety of XMLType physical storage in a *systematic* way. Figure 1 shows the architectural diagram of XSLT is transformed into XQuery which is further optimised over variety of XMLType physical storage and index models.



**Figure 1 - XSLT/XQuery Optimization Over XMLType Abstraction**

The rest of this paper is organized as follows. Section 2 provides motivating examples of using XSLT to transform XML in Oracle XDB and their optimisation results. Section 3 discusses XSLT to XQuery rewrite general techniques and various optimisation techniques to generate highly efficient XQuery based on the structural information of the input XML information. Section 4 discusses the underlying partial evaluation. Section 5 discusses performance experiments. Section 6 discusses related work comparison. Section 7 discusses future direction and section 8 concludes the paper with acknowledgement.

## 2. XSLT Transformation Motivating Examples

### 2.1 Optimisation of XSLT transformation over XML generated from relational tables

Oracle XML DB enables users to create a view of XML type instances via SQL/XML publishing functions over relational tables. Consider a classical case of the *dept* and *emp* tables forming master-detail relationship. 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 - Table “dept” content**

empno	ename	Job	Sal	deptno
7782	CLARK	MANAGER	2450	10
7934	MILLER	CLERK	1300	10
7954	SMITH	VP	4900	40

**Table 2 - Table “emp” content**

To generate XML from the relational tables *dept* and *emp*, we create a view *dept\_emp* (Table 3). 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 *employees* elements.

```

CREATE VIEW dept_emp
AS
SELECT
  XMLElement("dept",
    XMLElement("dname", dname),
    XMLElement("loc", loc),
    XMLElement("employees",
      (SELECT XMLAgg(XMLElement("emp",
        XMLElement("empno", empno),
        XMLElement("ename", ename),
        XMLElement("sal", sal)))
      FROM emp
      WHERE emp.deptno = dept.deptno)) as dept_content
FROM dept
  
```

**Table 3- SQL/XML constructed XML view dept\_emp**

**Example 1:** Consider the example of using XMLTransform function to generate an HTML document from each XML document row from *dept\_emp* view using XSLT stylesheet shown Table 5.

In this example, a SELECT query runs on the *dept\_emp* view, which has a *dept\_content* XMLType column. For each row of *dept\_emp*, it fetches an XML document from the *dept\_content* XMLType column and then it calls the XMLTransform() function to generate a new XMLType value. XMLTransform() is Oracle SQL/XML extension function, which applies a stylesheet on an XML document and returns the XSLT transformation result. The XSLT stysheet, in this case, generates HTML, which displays highly paid employees (employees whose semi monthly salary is more than 2000) in a department as show in Table 6.

```

<dept>
  <lname>ACCOUNTING</lname>
  <loc>NEW YORK</loc>
  <employees>
    <emp>
      <empno>7782</empno>
      <ename>CLARK</ename>
      <sal>2450</sal>
    </emp>
    <emp>
      <empno>7934</empno>
      <ename>MILLER</ename>
      <sal>1300</sal>
    </emp>
  </employees>
</dept>
<dept>
  <lname>OPERATIONS</lname>
  <loc>BOSTON</loc>
  <employees>
    <emp>
      <empno>7954</empno>
      <ename>SMITH</ename>
      <sal>4900</sal>
    </emp>
  </employees>
</dept>

```

**Table 4 - Two rows of XMLType instances from dept\_emp**

A straightforward functional evaluation of the query first materializes the XML contents of *dept\_xml* by constructing an XMLType instance from the relational data and then applies the XSLT transformation on it. This functional evaluation is sub-optimal because large input XML data has to be materialized before the actual XSLT transformation can be performed. Another key observation, however, is that the optimal evaluation plan should exploit the fact that one the heavily computed XPath expression:

*/emp[sal > 2000]*

actually maps to a predicate on the underlying *sal* column of the *emp* table. This fact can potentially promote index usage to filter all the rows that do not contribute to the final result. Also, XSLT stylesheet template bodies can be inlined all together to construct a single query that builds the result HTML document from the relational column data.

```

SELECT
XMLTransform(dept_emp.dept_content,
'<?xml version="1.0"?><xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="dept">
    <H1>HIGHLY PAID DEPT EMPLOYEES</H1>
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="lname">
    <H2>Department name: <xsl:value-of select="."/></H2>
  </xsl:template>
  <xsl:template match="loc">
    <H2>Department location: <xsl:value-of select="."/></H2>
  </xsl:template>
  <xsl:template match="employees">
    <H2>Employees Table</H2>
    <table border="2">
      <tr>
        <td><b>EmpNo</b></td>
        <td><b>Name</b></td>
        <td><b>Weekly Salary</b></td>
      </tr>
      <xsl:apply-templates select="emp[sal > 2000]"/>
    </table>
  </xsl:template>
  <xsl:template match = "emp">
    <tr>
      <td><xsl:value-of select="empno"/></td>
      <td><xsl:value-of select="ename"/></td>
      <td><xsl:value-of select="sal"/></td>
    </tr>
  </xsl:template>
  <xsl:template match="text()">
    <xsl:value-of select="."/>
  </xsl:template>
</xsl:stylesheet>')
FROM dept_emp;

```

**Table 5 - XSLT XMLTransform() example 1**

To realize this intuition, we use the **XSLT rewrite** technique to rewrite the original user stylesheet with the XSLT transformation into a SQL/XML query shown in Table 7.

Note that this rewritten query merely consists of SQL/XML generation functions, such as XMLConcat(), XMLElement(), XMLAgg() to construct the resultant XML from the underlying relational column data. It does not contain any XSLT or XPath operators at all. The rewritten query is a relational query on the relational table and the standard relational optimizer can select the index on the *sal* column of the *emp* table to speed up the query. The XSLT rewrite technique is accomplished in two steps.

First we rewrite the XSLT stylesheet into an equivalent XQuery using the input document structural information. The rewritten XQuery for the stylesheet from Table 5 is shown in Table 8.

```

<H1>HIGHLY PAID DEPT EMPLOYEES</H1>
<H2>Department name: ACCOUNTING</H2>
<H2>Department location: NEW YORK</H2>
<H2>Employees Table</H2>
<table border="2">
  <td><b>EmpNo</b></td>
  <td><b>Name</b></td>
  <td><b>Weekly Salary</b></td>
<tr>
  <td>7782</td>
  <td>CLARK</td>
  <td>2450</td>
</tr>
</table>

```

```

<H1>HIGHLY PAID DEPT EMPLOYEES</H1>
<H2>Department name: OPERATIONS</H2>
<H2>Department location: BOSTON</H2>
<H2>Employees Table</H2>
<table border="2">
  <td> <b>EmpNo</b></td>
  <td><b>Name</b></td>
  <td><b>Weekly Salary</b></td>
<tr>
  <td>7954</td>
  <td>SMITH</td>
  <td>4900</td>
</tr>
</table>

```

**Table 6 - Result of XSLT transformation from example 1**

```

SELECT XMLConcat(
  XMLElement( "H1", 'HIGHLY PAID DEPT EMPLOYEES'),
  XMLElement( "H2", 'Department name: '
  ||"SYS_ALIAS_4"."DNAME"),
  XMLElement( "H2", 'Department location: '
  ||"SYS_ALIAS_4"."LOC"),
  XMLElement( "H2", 'Employees Table'),
  XMLElement( "table", XMLAttributes('2' AS "border"),
  XMLElement( "td",
  XMLElement( "b", 'EmpNo')),
  XMLElement( "td", XMLElement( "b", 'Name')),
  XMLElement( "td", XMLElement( "b", 'Weekly Salary')),
  (SELECT XMLAGG(
  XMLElement( "tr",
  XMLElement( "td", "EMP"."EMPNO"),
  XMLElement( "td", "EMP"."ENAME"),
  XMLElement( "td", "EMP"."SAL")))
  FROM EMP
  WHERE SAL > 2000
  AND DEPTNO=DEPT.DEPTNO)))
FROM DEPT

```

**Table 7 - Rewritten query for query example 1**

```

SELECT XMLQuery(
  'declare variable $var000 := .;
  (: builtin template :)
  (
  let $var002 := $var000/dept
  return
  (: <xsl:template match="dept"> :)
  (
  <H1>HIGHLY PAID DEPT EMPLOYEES</H1>,
  (
  let $var003 := $var002/dname
  return
  (: <xsl:template match="dname"> :)
  <H2>{fn:concat("Department name: ", fn:string($var003))}</H2>,
  let $var003 := $var002/loc
  return
  (: <xsl:template match="loc"> :)
  <H2>{fn:concat("Department location: ", fn:string($var003))}</H2>,
  let $var003 := $var002/employees
  return
  (: <xsl:template match="employees"> :)
  (
  <H2>Employees Table</H2>,
  <table border="2">
  {
  <td><b>EmpNo</b></td>,
  <td><b>Name</b></td>,
  <td><b>Weekly Salary</b></td>,
  (
  for $var005 in ($var003/emp[sal > 2000])
  return
  (: <xsl:template match="emp"> :)
  <tr>
  <td>{fn:string($var005/empno)}</td>
  <td>{fn:string($var005/ename)}</td>
  <td>{fn:string($var005/sal)}</td>
  </tr>
  )
  )
  }
  </table>
  )
  )
  )
  )' PASSING dept_emp.dept_content RETURNING CONTENT)
FROM dept_emp

```

**Table 8 - XQuery resultant from XSLT rewrite example 1**

Next, we further rewrite XQuery into a SQL/XML query with the input XML construction function using XQuery

rewrite techniques [3,4] to result in the final SQL/XML query shown in Table 7. The resultant query is very efficient, because it does not fetch or compute any unnecessary data that do not contribute to the final transformation result and because it uses B-tree index to compute the predicate.

## 2.2 Combined optimisation of XQuery/XPath optimisation over XSLT transformation

Since the output of XMLTransform() is another XMLType value that can be further queried or transformed through XQuery/XPath or XSLT, we can combine the XSLT optimization with the next step XQuery/XPath optimization.

**Example 2:** We create an XSLT view *XSLT\_VU* shown in Table 9, which wraps the XSLT transformation from Example 1.

```
-- wrap XMLTransform() example 1 as an XSLT_VU.
CREATE VIEW xslt_yu AS
-- XMLTransform() text from example 1 in Table 5
SELECT XMLTransform(dept_emp.dept_content,
'.....') AS xslt_rslt
FROM dept_emp
```

**Table 9 - XSLT View**

After this, we query *XSLT\_VU* via another FLWOR XQuery using XMLQuery() operator as shown in Table 10.

```
SELECT
XMLQuery( 'for $tr in ./table/tr return $tr'
PASSING xslt_yu.Xslt_rslt RETURNING CONTENT)
FROM XSLT_VU
```

**Table 10 -XQuery query on the result from XSLT**

The combined optimisation that applies XSLT rewrite to XQuery and XQuery rewrite to SQL/XML recursively optimises the query from Table 10. The final optimal query from XSLT and XQuery rewrite is shown in Table 11.

```
SELECT
(SELECT XMLAgg(XMLElement( "tr",
XMLElement( "id",empno),
XMLElement( "id",ename),
XMLElement( "id", sal)))
FROM emp
WHERE sal > 2000
AND deptno = d.deptno)
FROM dept d
```

**Table 11 - Optimal SQL/XML Query from combined optimization of XSLT, XQuery, SQL/XML.**

## 3. XSLT to XQuery Rewrite

As shown in the previous examples, the key step is to rewrite XSLTstylesheet into an equivalent XQuery. XSLT and XQuery share the same XPath and many functions and

operators as a common core. They both have similar language XML node creating constructs, iterations with sort, conditional testing and variable access. So the translation between these constructs is straightforward. The main difference is however, that XSLT templates are activated as a result of dynamic pattern matching while XQuery functions are invoked explicitly. This is the biggest challenge of rewriting XSLT into XQuery.

### 3.1 General XSLT Rewrite to XQuery Technique

Generally speaking, an XSLT stylesheet is composed of a collection of templates provided by the user and the default built-in template. Each template can be translated into an XQuery user defined function and each XSLT instruction in the template body can be converted into its corresponding XQuery expression. The challenging aspect here is to how to translate *<apply-templates/>* instruction, which implicitly demands the template pattern matching. The idea proposed in [9] is to compile XSLT *<apply-templates/>* instruction into a combination of XQuery's conditional expressions where the expression conditions literally model the template pattern matching and the expression bodies contain function calls that invoke the corresponding XQuery function that translated from the XSLT template. This approach essentially converts the pattern matching and template selection normally carried by the XSLT processor to explicit XQuery conditional expressions executed by the XQuery processor. However, this **straightforward compilation approach** usually results in an inefficient execution. This is because the XSLT processor might internally provide aggressive optimisations of locating the right template (via internal hash tables for example), whereas the translated XQuery query uses a large number of conditional expressions to sequentially test which template to instantiate [9]. Therefore, we have to apply aggressive optimisations in order to get an efficient query. In the absence of the input XMLType structural information, this straightforward compilation approach is appropriate. However, in the context of RDBMS, we can derive that structural information and use that to rewrite XSLT into XQuery. In particular, using the partial evaluation technique to create a specialized XQuery query from XSLT stylesheet can result in highly efficient XSLT transformation.

### 3.2 Exploiting XML Structural Information

In the database environment, we can typically obtain the XMLType structural information from the database meta-data information.

- If the input XMLType is from XMLType table or columns with XML schema or DTD information, we can use XML schema and DTD to get the XML structural information.

- If the input XMLType is generated from relational or object-relational data as example 1, we can get the XML structural information from the underlying relational or object relational schema.
- If the input XMLType is computed from another XQuery/XPath, then we can derive the structural information based on the static typing result of the XQuery.
- If the input XMLType is computed from another XSLT transform as in example 2, we rewrite the XSLT into XQuery recursively first and then derive the structural information of the XSLT result based on the static typing result of the equivalent XQuery query.
- If the input XMLType is from view column, we can trace the view column that is built on top of the above cases.

We use the XML structural information during the XSLT to XQuery translation when we apply *optimal XSLT to XQuery rewrite techniques* such as: *template instantiation inlining; children template instantiation with leveraging children model group and cardinality information; removing unnecessary backward XPath testing; removing unused templates*, all of which are discussed in sections 3.3 to 3.7. This results in an XQuery that is significantly simpler than the one from the straightforward approach proposed in [9]. We will go over each rewrite technique in following sections. The key point is that although each one of the rewrite technique alone is quite simple, however their combined optimisation effect is drastic.

In summary, our approach attempts to generate efficient XQuery by aggressively exploiting the structural information of the input XMLType and does optimisations as much as possible based on that information. This approach makes sense because XSLT transformation in database is usually applied to a set of large number of input XML documents, each of which is either stored as a row in an XMLType table or column conforming to one or a collection of XML schemas that are registered to RDBMS or generated from object relational data. One common such use case is that XSLT transformation is used to transform a set of XML documents conforming to schema *S1* to another XML documents conforming to schema *S2* due to non-compatible XML schema (*S1* and *S2* are not compatible as they are defined by different organizations).

In the case that the set of XML documents conform to a collection of XML schemas (say *S1*, *S2*,... *Sn*), however, we can rewrite the XSLT transformations into multiple optimal SQL/XML queries based on *S1*, *S2*,... *Sn* respectively, during query compile-time. Then during run time, depending on the actual target XML schema, we execute its corresponding optimal SQL/XML query plan.

### 3.3 Template instantiation inline

This technique inlines the body of the activated templates directly in the caller, which either calls the template explicitly through `<call-template/>` instruction or calls the template implicitly through `<apply-templates/>` instruction. For example, there are two `<apply-templates/>` instructions in the XSLT templates shown in Table 5. The first one matches "dept" element and the other one matches "employees" element:

```
<xsl:template match="dept">
...
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="employees">
...
  <xsl:apply-templates select="emp[sal > 2000]"/>
...
</xsl:template>
```

As a result of the partial evaluation step we know that the *current nodes* for the first `<apply-templates>` instruction are the elements "dname", "loc" and "employees", while for the second `<apply-templates>` instruction it is the element "emp". We replace the first `<apply-templates>` with the template bodies activated by "dname", "loc" and "employees". For the second template, which is the activated template for element (*employees*), we recursively apply the same algorithm but with "emp" as a current element. The inlining effect is shown in Table 8.

### 3.4 Children template instantiation based on model group and cardinality information

This technique utilizes the children model group information in order to explicitly arrange the inline template body. For example, the XML schema specifies that the children model group can be one of the following: *sequence*, *choice*, or *all*. Model information allows us to inline the XQuery expressions for children elements more efficiently, especially for the most common models of *sequence* and *choice*.

In example 1, if we only knew that there are three children elements "dname", "loc" and "employees" under element "dept", but we did not know in what order they appear, then we would have to rewrite the XSLT into XQuery as shown in Table 12.

However, if we knew that the children model group is "choice", that is, "dept" element has only one child element, which can be either "dname", or "loc" or "employee" element, then we could inline them by removing the explicit FLWOR for each node() access as shown in Table 13.

Finally, if the three children elements model group is "sequence", that is, "dept" element has "dname", "loc" and "employee" child elements in order, then we can inline

them by removing all the conditional tests completely as shown in Table 14.

Another optimisation we can do is to generate FLWOR clause based on the cardinality information of the child element. For example, we use FOR clause to iterate through each 'emp' element because it has multiple occurrences under 'employee' element. We use LET clause to access 'dname' element because it has at most one occurrence under 'dept' element. This is shown in Table 15.

```
declare variable $var000 := .;
(
  let $var002 := $var000/dept
  return
  for $var003 in $var002/node()
  return
  (
    if ($var003 instance of element(dname)) then
      inlined xquery from template for "dname",
    else if ($var003 instance of element(loc)) then
      inlined xquery from template for "loc",
    else if ($var003 instance of element(employee)) then
      inlined xquery from template for "employee"
  )
)
```

**Table 12 - inline templates for the “all” model group**

```
declare variable $var000 := .;
(
  let $var002 := $var000/dept
  return
  if ($var002/dname) then
    inlined xquery from template for "dname"
  else if ($var002/loc) then
    inlined xquery from template for "loc"
  else if ($var002/employee) then
    inlined xquery from template for "employee"
)
```

**Table 13 - inlined templates for the “choice” model group**

```
declare variable $var000 := .;
(
  let $var002 := $var000/dept
  return
  (
    let $var003 := $var002/dname
    inlined xquery from template for "dname",
    let $var003 := $var002/loc then
    inlined xquery from template for "loc"
    let $var003 := $var002/employee then
    inlined xquery from template for "employee"
  )
)
```

**Table 14 - inline templates for the “sequence” model group**

```
let $var005 := $var003/dname
inlined xquery from template for "dname"

let $var003 := $var002/employees
return
for $var005 in ($var003/emp[sal > 2000])
inlined xquery from template for "emp"
```

**Table 15 - inline based on cardinality**

### 3.5 Removing unnecessary backward XPath Step Testing

The XPath P used in the pattern matching of a template can have multiple XPath steps with predicates. The conceptual definition of pattern matching implies finding the existence of parent node such that when evaluating XPath P with that parent node as a context node yields the result containing the testing node to which template is applicable. Because the pattern matching based on the conceptual definition is quite inefficient, an efficient XSLT processor usually evaluates the XPath expression by reversing the pattern as proposed in [6] and illustrated in [9]. However, with the XML structural information, we can remove some of the unnecessary backward XPath testing in the translated XQuery.

Consider the template in Table 16, which uses Xpath with multiple steps for matching condition

```
<xsl:template match = "emp/empno">
```

**Table 16 - multi-step XPath in matching condition**

Without knowing that "empno" element has only one parent element "emp", we would have to generate using XQuery conditional expression testing shown in Table 17.

```
((($var instance of element(empno)) and fn:exists($var/parent::emp) )
```

**Table 17 - Parent Axis XPath test**

However, from the SQL/XML generation function, we can infer that "empno" element has only one "emp" parent element, this fn:exists() testing with parent axis of "emp" element can be eliminated.

On the other hand, if we have two templates, one of them with an XPath predicate, as shown in Table 18.

```
<xsl:template match = "emp/empno[. = 3456]">
...
<xsl:template match = "emp/empno">
...
```

**Table 18 – multi-step XPath having predicate**

then we can not remove the conditional expression testing. However, we can still simplify the pattern by removing the parent axis check as shown in Table 19.

```
if ($var instance of element(empno)) and (empno[. = 3456]) then
...
else if ($var instance of element(empno)) then
...
```

**Table 19 - Parent Axis Removed**

### 3.6 Built-in template Only Optimisations

If we find that all nodes of an XML document (it can be the entire XML document) use the built-in templates, then we can generate a more compact XQuery code without explicitly iterating through each node of the subtree from top to bottom. For example, if the XSLT template applied in example is shown in Table 20, then we can just generate the a compact Xquery shown in Table 21.

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

**Table 20 – Empty stylesheet**

In this case, the template applies built-in template for each node of the XML document, the translated XQuery uses // to select all the self and descendant text nodes and concatenate all the *fn:string()* value of each text node together.

```
declare variable $var000 := .;
(: builtin template :)
fn:string-join(
  for $var002 in $var000/text()
  return fn:string($var002), " ")
```

**Table 21 - Compact XQuery with built-in template only**

### 3.7 Removed non-instantiated template

We also keep track of all the templates that might be instantiated based on the input XML structural information. For those templates, which are not instantiated, we don't generate corresponding XQuery for it.

## 4. XSLT Partial Evaluation

### 4.1 Rationale of using partial evaluation

To be able to use various techniques discussed above to generate efficient XQuery, we have leveraged partial evaluation techniques. Although one can use data flow analysis to optimise the straightforward translated XQuery, we found that it is more efficient and easier to generate optimised XQuery by partially evaluating the XML structural information. In fact, many XQuery optimisations based on static type analysis can be conceptually modelled as if doing XQuery partial evaluation on the input XML static type tree.

Partial evaluation technique is not new and has been well studied in the past [14,15]. An application can benefit from partial evaluation if:

*The application computation can be described as a function  $F(X,Y)$ , where  $X$  changes less frequently than  $Y$ , and where a significant part of  $F$ 's computation depends only on  $X$ .*

The key observation is that: if we let the function  $F$  to be the XSLT stylesheet itself and if we decompose the input XML document into a **structural information part** ( $X$ ) and an **actual content data** ( $Y$ ), then we can see that partial evaluation has a perfect application in the context of XSLT. Since pattern matching in a typical XSLT stylesheet is primarily on the input XML document structure and the actual content data testing is usually in the XPath predicate, so we can significantly simplify the XSLT based on the

partial evaluation on the structural information of the input XML. The expression that depends on the actual content data, such as XPath predicate, is then left in the residual XQuery and can typically be efficiently processed via index probe in the database environment. Furthermore, the default built-in template is solely based on input XML document structure and can be inlined multiple times via partial evaluation with different nodes in the document as input parameters instead of being recursively called as a function via straightforward translation of XSLT to XQuery approach.

When applying the partial evaluation in the context of XSLT processing, we first construct a special **sample XML document**, which captures all the structural information from the input XMLType but not the actual content values. Then we invoke the XSLTVM [13], enhanced with special trace instructions to get the **execution trace information**, such as, the list of actually instantiated templates for each `<apply-template/>` instruction etc. We also build a **template call execution graph**, which models the exact sequence of template activations.

Note that since we don't know the actual value of the text node, we have to be conservative during the partial evaluation and assume that the result of matching pattern with a predicate, such as: *emp[empno = 3456]* is always true for an 'emp' element node with a child element "empno".

### 4.2 Sample XML document generation

Based on the input XML schema, DTD, or relational schemas, we can generate a sample XML document that captures the structural information.

To denote XML schema model groups, such as choice, all, or data type information, we annotate elements with a special attribute belonging to predefined Oracle XDB namespace.

### 4.3 Partial Evaluation

The partial evaluation step consists of two phases. During the first one we compile the stylesheet into XSLTVM bytecode along with the special '**trace-instructions**' for collecting the run time information. We build a **template-table** for each template listed in the stylesheet. It contains important template information, including the template formal parameters (if any). We also build a **trace-table**, where each table-entry maps to an `<apply-templates>` instruction used in the stylesheet. The entry has a **trace-call-list** with the run-time instantiated template and the actual parameters, along with the matching XML nodes that cause the template activation.

Next, the XSLTVM is invoked to transform the sample XML document. The *trace-instructions*, when executed, send the actual information to the Execution Graph Builder, which builds the template **execution graph**. Each template instantiation creates a new graph state (unless there is a

recursion), which corresponds to the activated template and a transition arch representing the current node. The first graph transition corresponds to the first template activation with the sample XML document root node.

#### 4.4 XQuery Generation

Based on the template execution graph and the trace-call-list, we can generate the target XQuery using the efficient XQuery translation techniques we have discussed in section 3.3 to 3.7. Currently, we support two rewrite modes: **non-inline mode** and **inline mode**.

If the template execution graph contains at least one recursion, then we are in non-inline mode. In that case we generate the result query as a list of XQuery functions, each one corresponding to an invoked template traced from partial evaluation. We scan the *template-table* and compile each instantiated template into an XQuery function. The main query merely calls the first instantiated template. For `<xsl:apply-templates/>` instruction, we scan the trace-call-list and generate a conditional function call for each call-entry.

If the template call execution graph is not recursive, then this is the inline mode. We generate only a main query without any functions. All stylesheet template bodies are inlined into the main query body. We scan the template call execution graph states and for each state, we inline the state-template body. In this mode each `<xsl:apply-templates>` element is equivalent to a state transition. We scan the `<xsl:apply-templates>` trace-call-list and generate a conditional function call only for *call-entries*, whose *current node* is the same as the *transition-node*. Then we shift to the new state and inline the new state-template body recursively.

Although we currently have only two modes: inline or non-inline, that is, we use non-inline mode as soon as we see one recursive function call, we can always enhance it to do partial inline mode as well. In our experimental assessment discussed below, however, we find that even with just the current approach, more than 50% of the XSLT cases can benefit from the full inline mode.

### 5. Experimental Assessment

We used XSLTMark [19] tests to measure the effectiveness of XSLT rewrite. It is an XSLT benchmark suite, which uses forty test cases designed to assess important functional areas of an XSLT processors. We store XML documents object relationally with various indexes created for efficient predicate evaluation.

The first performance objective is to compare the performance of XSLTquery going through the rewrite versus not going through the rewrite. The XSLT rewrite

approach rewrites the XSLT into XQuery and the XQuery is further rewritten into SQL/XML query over the underlying object relational storage tables with index access. The XSLT no rewrite approach constructs the XML documents from the storage tables as DOM object and then XSLT runs on top of the DOM object.

Figure 2 shows performance comparison between XSLT rewrite versus XSLT no rewrite for 'dbonerow' test case from XSLTMark. 'dbonerow' XSLT uses XPath value predicate to select one qualified node. The X-axis indicates the size of XML documents to which XSLT is applied while the Y-axis indicates the time taken to perform the XSLT transformation on the XML documents with and without rewrite. We measured four cases with size of XML document as 8M, 16M, 32M and 64M. As Figure 2 shows, the XSLT rewrite approach performs much better than XSLT without rewrite approach consistently. Furthermore, the time taken for XSLT no rewrite approach increases quickly in response to the increase size of XML document whereas the time taken for the XSLT rewrite approach grows slowly due to the use of B-tree index to process the predicate.

Figure 3 shows the performance comparison between XSLT rewrite and XSLT no rewrite for the 'avts', 'chart', 'metric', 'total' test cases from XSLTMark. These are test cases that there is no XPath value predicate so that there is no value index to be used to filter nodes. The X-axis shows the test cases and the Y-axis indicates the time taken to perform the XSLT transformation for each case with or without XSLT rewrite. Again the XSLT optimised through rewrite outperforms those without XSLT rewrite.

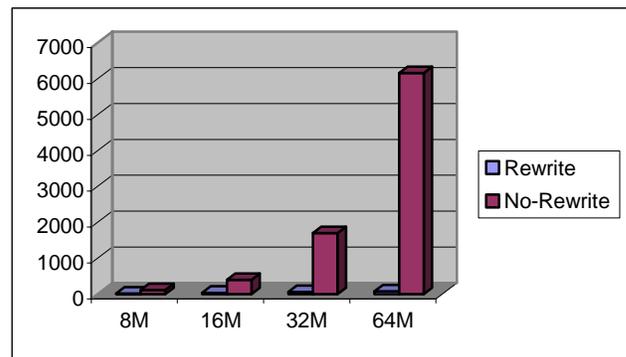
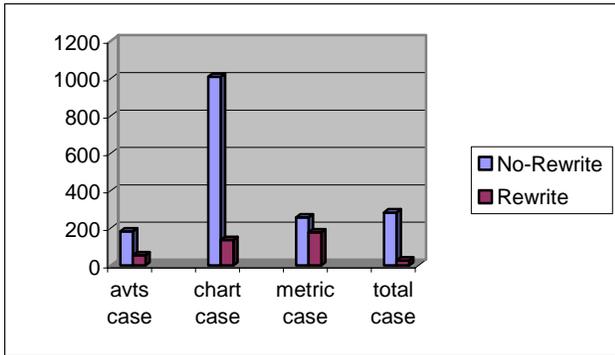


Figure 2 - Performance comparison between XSLT rewrite versus XSLT no rewrite for few nodes selection

For both 'avts' and 'metric' cases, the XSLT essentially constructs new XML nodes with conditional expression. The resultant SQL/XML query from XSLT rewrite can be efficiently executed by the top-stream evaluation of SQL/XML publishing function [2]. For 'chart' and 'total' test cases from XSLTMark, the XSLT uses count() and sum() aggregate function. The resultant SQL/XML query from XSLT rewrite has these common aggregate functions

which can be efficiently executed by the underlying RDBMS aggregation process in parallel manner.



**Figure 3 - XSLT rewrite Vs XSLT no-rewrite Performance Comparison**

Our second objective was to measure how effective the XSLT to XQuery inline generation mode is. We found that 23 out of 40 XSLTMark test cases can be completely inlined into an efficient XQuery query without any function calls. So more than 50% of XSLT use cases in the benchmark can benefit from inline translation of XSLT to XQuery without containing any XQuery function calls.

## 6. Related Work Comparisons

Ideas of efficient processing of XSLT in the database context have been proposed in the past. G. Moerkotte in [6] has proposed to compile XSLT into an internal algebra that can be integrated with the rest of the relational engine. The paper [6] concluded that future research would be on combined optimisations of queries constructing XML documents and XSLT processing them and also alternative ways to incorporate XSLT processing into database engines. Instead of rewriting XSLT into some private algebra, our work shows that one of the appealing alternatives of incorporating XSLT processing into RDBMS is to use XQuery as an intermediate language into which XSLT is translated and then further optimise the XQuery by taking advantage of how the XML is stored, indexed or generated in RDBMS. This approach can leverage all the XQuery and SQL/XML processing and optimisations work done so far because processing of XQuery and SQL/XML on database engine have been far more widely studied than that of XSLT. Furthermore, this XSLT to XQuery rewrite approach enables cross-language global optimisations of combined XQuery, XSLT and SQL/XML invocations in one query as shown in example 2 of section 2.2.

S. Jain etc [7] and Chengkai Li etc [8] have proposed algorithms of translating XSLT into efficient SQL over input XMLType view generated from relational tables. They describe optimisation strategies of XSLT processing of XML publishing views over relational data. This essentially falls into the category of combined

optimisations of XSLT processing queries constructed XML documents. Their goal is achieved via extensive XSLT stylesheet analysis, followed by special optimisations and resulting into a generation of efficient SQL query.

We believe that our approach of using XQuery as an XML storage independent intermediate language has advantages over ideas discussed in paper [7] and [8] because it clearly divides the XML language processing into language specific and storage specific optimisation phases. During the first phase we use partial evaluation to compile and optimise the input XSLT stylesheets into XQuery. Next, we use XQuery as the common language and leverage XQuery rewrite framework to perform XML storage and index model specific optimisations in RDBMS.

Rewriting XSLT into XQuery has been recently proposed in [9]. However, it is a straightforward translation and the resultant query is full of recursive functions and large conditional branch expressions. It is inefficient and may not yield any performance benefits compared with just evaluating the original XSLT stylesheet directly. In order to obtain optimal performance the result query has to be aggressively optimised but along with the structural information of the input XML document.

Our approach, on the other hand, leverages the fact that we can obtain the input XMLType structural meta-data information, such as the XML Schema, DTD or the underlying relational schema, so we can generate a highly efficient XQuery from XSLT. By partially evaluating the input XMLType structural information during query compilation time, we can inline XSLT templates and eliminate many unnecessary XQuery conditional expressions. That results in an efficient and easily optimisable query. This is not surprising because the partial evaluation can help us to do aggressive optimisations of XSLT through combination of constant folding, function and variable inlining and cross-function optimisation propagation applied to the input XML nodes. This is what is genuinely needed for XSLT optimisation in general.

Furthermore, our approach can deal with XML schema evolution as the XSLT can be recompiled based on the new version of XML schema. In fact, this recompilation process is automated because the XSLT query has dependency on the XML schema whose change is tracked by the database system.

## 7. Open Issues and Future Directions

### 7.1 XSLT constructs

Certain XSLT constructs are hard to translate into XQuery. In particular, handling XSLT 2.0 [21] grouping construct, `<xsl:for-each-group>`, will be challenging as the group by construct needs to be part of the XQuery as illustrated in [20].

## 7.2 Partial Evaluation

For XSLT partial evaluation, we currently do not handle recursive XML document structure. We will need to add special attribute annotation to denote recursive structures in the sample XML document and enhance the partial evaluation to handle recursive case. For XQuery compilation from XSLT, we will work on partial inline mode.

## 7.3 XML Schema Evolution

The other direction is to study the XSLT to XQuery rewrite based on a set of XML schemas that are similar, for example, evolving from the same parent schema. This is common for XMLType storage table with evolving XML schemas.

## 7.4 XML storage Model

Furthermore, we will need to study the XSLT performance for different physical XML storage and index models (object relational storage, CLOB or BLOB storage with path/value index, tree storage with path/value index) through XSLT to XQuery rewrite so that we know what type of storage is ideal for what type of XSLT query.

## 8. Conclusion

In this paper, we show that optimising XSLT transformation over XML documents stored in or generated from the database is feasible despite the fact that XSLT is a highly declarative template-based language. Our approach is to rewrite XSLT stylesheets into optimised XQuery and then apply XQuery database optimisations. However, it is crucial that we take into account the input XML document structural information by partially evaluating it. The end result shows that XSLT processing over large XMLType documents managed by RDBMS can enjoy the same performance benefits as that of XQuery or SQL/XML query. The classical declarative query language processing techniques, such as indexing probe, iterator based execution model [10] with parallel aggregation and sorting can be fully leveraged and applied to the XSLT transformations as well.

## 9. Acknowledgements

We gratefully acknowledge the contributions of all the members of the Oracle XML DB development and product management teams. In particular, we thank Muralidhar Krishnaprasad who have provided ideas and insights for this project.

## 10. References

[1] I.O. for Standardization (ISO). Information Technology- Database Language SQL-Part 14: XML-Related Specifications (SQL/XML)

[2] M. Krishnaprasad, Z. H. Liu, A. Manikutty, J. Warner, V. Arora. "Towards an industrial strength SQL/XML infrastructure". ICDE 2005

[3] M. Krishnaprasad, Z.H. Liu, A. Manikutty, J. Warner, V. Arora, S. Kotsovolos. "Query rewrite for XML in Oracle XMLDB". VLDB 2004

[4] Z.H.Liu, M.Krishnaprasad, V.Arora, "Native XQuery Processing in Oracle XMLDB". SIGMOD 2005

[5] XSLT 1.0: <http://www.w3.org/TR/xslt>.

[6] G. Moerkotte, "Incorporating XSL Processing Into Database Engines", VLDB 2002

[7] Jain, R. Mahajan, D. Suci, "Translating XSLT Programs to Efficient SQL Queries", In Proc of the Eleventh Int'l Conference on World Wide Web, pages 616-626, 2002.

[8] C. Li, P. Bohannon, H. F. Korth, P.P.S. Narayan, "Composing XSL Transformations with XML Publishing Views", SIGMOD 2003.

[9] A Fokoue, K Rose, J. Simeon, L. Villard, "Compiling XSLT 2.0 into XQuery 1.0", Proceedings of the 14<sup>th</sup> international conference on Word Wide Web Publishing, May 2005.

[10] G. Graefe. "Query evaluation techniques for large databases". ACM Computing Surveys. 25(2), June 1993

[11] R. Murthy, Z. Hua Liu, M. Krishnaprasad, S. Chandrasekar, A. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, V. Krishnamurthy: Towards An Enterprise XML Architecture , SIGMOD 2005

[12] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk. "Querying XML Views of Relational Data." VLDB 2001

[13] A Novoselsky, "The Oracle XSLT Virtual Machine", XTech 2005, Amsterdam, Netherlands. <http://www.idealliance.org/proceedings/xtech05/papers/04-02-01/>

[14] C. Consel and O. Danvy. "Tutorial Notes on Partial Evaluation". In ACM Symposium on Principles of Programming Languages, pages 493-501, 1993.

[15] N. Jones, C. Gomard, and P. Sestoft. Partial Evaluation and Automatic Program Generation. Englewood Cliffs, NJ: Prentice Hall, 1999.

[16] Oracle XML DB Developer's Guide: Oracle 9iR2. See <http://otn.oracle.com/tech/xml/xmlldb>

[17] K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. Truong, B. Linden, B. Vickery, C. Zhang: "System RX: One Part Relational, One Part XML". In SIGMOD 2005.

[18] S. Pal, I. CSeri, O. Seeliger, M. Rys, G. Schaller, P. Kukol, W. Yu, D. Tomic, A. Baras, C. Kowalczyk, B. Berg, D. Churin, E. Kogan: "XQuery Implementation in a relational database system". In VLDB 2005.

[19] XSLT benchmark tests <http://www.datapower.com/xmldev/xsltmark.html>

[20] K.Beyer, D. Chamberlin, L. Colby, F. Ozcan , H. Pirahesh , Y. Xu: Extending XQuery for Analytics, SIGMOD 2005.

[21] XSLT 2.0: <http://www.w3.org/TR/xslt20>