Effective and Efficient Update of XML in RDBMS

Zhen Hua Liu, Muralidhar Krishnaprasad, James W. Warner, Rohan Angrish, Vikas Arora
Oracle Corporation
400, Oracle Parkway, Redwood Shores, CA 94065, USA
{zhen.liu, muralidhar.krishnaprasad, jim.warner, rohan.angrish, vikas.arora}@oracle.com

ABSTRACT
Querying XML effectively and efficiently using declarative languages such as XQuery and XPath has been widely studied in both academic and industrial settings. Most RDBMS vendors now support XML as a native data type with SQL/XML and XQuery support over it. However, the problem of updating XML is still the subject of ongoing effort. Several SQL/XML update extensions have been implemented and an XQuery Update Facility is in the proposal phase to add an update capability to XQuery. There are a lot of challenges involved in updating XML, particularly identifying and updating partial fragments of XML while maintaining concurrency, transactional semantics and validity of the document. In this paper, we illustrate the XML update functionality provided by Oracle XML DB within the context of SQL/XML. This functionality has been developed and optimized based on actual customer use cases of querying and updating XML. We discuss our design philosophy, optimization details for providing capability of updating XML and compare it with the current XQuery Update Facility proposal with the goal of providing insight into incorporating the XQuery Update Facility in the SQL/XML standard in the future.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *Relational databases, transaction processing.*

General Terms: Algorithms, Management, Performance, Design, Standardization, Languages.

Keywords: XML, XQuery, update.

1. INTRODUCTION
Querying XML using XQuery/XPath [11] has been widely studied in both academic and industrial settings. Relational database systems have now implemented XML support based on the SQL/XML[14] standard which adds support for XML as a native data type, as well as XML query support using XQuery/XPath embedded in SQL functions and constructs such as XMLQuery(), XMLExists(), XMLCast(), XMLTABLE and other vendor specific extension functions [4,5,8,9].

Efforts to effectively and efficiently update XML are ongoing in research settings as well. The paper 'Updating XML' [3] proposed a set of XML node level deletion, update, insertion, replace expressions in XQuery and the FLWU (FOR..LET..WHERE..UPDATE) extension to XQuery. The paper compared some of the implementation strategies of supporting such language extensions for updating XML views over relational data. Similar ideas and implementation strategies have also been discussed in papers [6, 7]. XQuery![12], on the other hand, proposed a full semantic framework that extends XQuery with side-effect operations while preserving the benefits of XQuery's declarative semantics whenever possible. It enhances the current XQuery language [11] to allow side-effect operations in any context and shows the significance of such an approach when building web service applications using XQuery with both query and update capabilities.

In parallel, the W3C working group has completed the XQuery Update Facility [10] proposal. It supports XML updates at the node level and has semantic constraints to restrict the placement of update expressions (side-effect expressions). The proposal extends XQuery with 5 new expressions – insert, delete, replace, rename, transform. The first 4 expressions take a target expression in the syntax, which is the target of the update. The transform expression provides syntax to create a copy of an expression and update the copy, so that the original XML value is not mutated.

These XML update language proposals have yet to be incorporated into the SQL/XML standard to address the issue of updating XML values inside the RDBMS. To build an industrial strength XML application, providing the XML update capability within the framework of an RDBMS is crucial. Oracle has introduced a set of XML update functions, such as *deleteXML*(), *updateXML(), insertXML()*, as extensions to SQL/XML functions in Oracle 10g[9] to address the need for declaratively updating XML at the node level. These functions provide transactional, snapshot semantics for updating XML similar to any other data type supported in the RDBMS. Based on actual customer use cases of updating XML, we find that these XML update functions, along with other SQL/XML query functions, can *effectively* allow users to declaratively update XML nodes and fragments for both XML transformation and persistent modification of stored XML. Furthermore, the SQL/XML capability used in the context of Oracle PL/SQL enables user to build sophisticated applications involving XML query and update. This approach has the added advantage that the learning curve for users is smooth as they have already been familiar with SQL procedural languages, such as Oracle PL/SQL [24. 25] or the standard SQL PSM [15] that have provided full fledged procedural control logic for executing SQL. Furthermore, XML node level updates can also be supported *efficiently* by using the **XML update rewrite** technique described in section 4 of this paper.

In this paper, we will go over the design philosophy and implementation challenges related to supporting these XML update functions in the Oracle database system, compare them with the current XQuery Update Facility [10], and discuss how they may be leveraged to support future SQL/XML extensions that incorporate the XQuery Update Facility within SQL.

SIGMOD'07, June 12–14, 2007, Beijing, China.

The main contributions of this paper are:

- We show that these XML update functions can be used for both updating persistent XML and transforming XML in a declarative fashion, and compare them with the current XQuery Update Facility proposal [10].
- We show that these XML update functions can be used with XQuery/XPath embedded in SQL/XML functions, such as *XMLExists*(), and embedded in PL/SQL to provide full and complete control of updating XML with procedural control flow, exception handling, variable assignment, snapshot semantics, transactional semantics – all of which are essential to build XML based applications in an RDBMS. We compare this approach with the XQuery! [12] and XQuery-P [13] proposals.
- We discuss the optimization challenges in supporting these XML update functions and present the **XML update rewrite** technique. In particular, we show how the XML update rewrite technique has handled cases that the existing approaches of updating XML views over relational data [3, 6, 7] have not fully explored. We also discuss some specific challenges of this approach including handling cases of updating XML with recursive schemas, preserving XML DOM and order fidelity, and providing partial XML update capability.
- We also show how these XML update functions may be leveraged to support the XQuery Update Facility in the future, and its possible influence on the XML update capability in future extensions to SQL/XML.

The rest of this paper is organized as follows. Section 2 describes Oracle XML DB's XML update functionality and its semantics in detail with examples comparing with the current XQuery Update Facility proposal [10]. Section 3 presents our rationale and design philosophy in providing users with the capability of effectively updating XML. Section 4 describes the XML update rewrite technique to implement efficient updates, including the algorithm and addressing issues that are not explored with existing approaches of updating XML view over relational data. Section 5 describes the performance evaluation of XML update rewrites. Section 6 describes related work. Section 7 discusses future direction. Section 8 concludes the paper with acknowledgements.

2. XML UPDATE SQL/XML FUNCTIONS

2.1 XML Update Function Descriptions

Oracle supports the following set of XML update functions as extensions to the current SQL/XML standard: *deleteXML(), updateXML(), insertXMLBefore(), appendChildXML(), insertChildXML()*. These functions have the following commonalities:

- They all accept an XML instance as input and return a new XML instance as the output. The new XML instance is logically a copy of the input XML instance with the change specified by the update functions applied.
- They all can be used to either transform the input XML, typically used in a SQL SELECT statement, or to modify a stored XMLType column typically used in a SQL update statement. The update of the persistent XML column is done by replacing the original XML column instance by the new copy with the necessary changes applied.
- They all take XPath as input and compute a sequence of nodes as the targets for change by applying the XPath on the copy of the input XML instance. Note the allowed XPath is the common sequence of path steps with axies, which is equivalent to the Path expression [18] defined in XQuery [11]. The optional namespace string parameter declares the bindings between any namespace URLs and namespace prefixes used in the XPath parameter.

The individual XML update functions are described below.

deleteXML:
Syntax: deleteXML(input_XML_instance,
XPath_string [, namespace_string])
deleteXML() deletes a sequence of nodes matched by the XPath expression applied on the copy of the input XML instance and returns the copied instance with these nodes deleted. This function is analogous to the 'do delete' expression in the XQuery Update Facility proposal [10].

updateXML:
Syntax: updateXML (input_XML_instance,
XPath_string, value_expression [, namespace_string])
updateXML() takes as arguments an input XML instance and an XPath-value pair, and returns a new XML instance with nodes or their value changed. The node for updating is identified by applying XPath on the copy of the input XML instance. If the node matched by the XPath is an XML element, then the corresponding *value_expression* must be an XML element node. If the node matched by the XPath is an attribute, text, PI or comment node, then the *value_expression* can be the corresponding node or SQL scalar type. For the case of SQL scalar type, the corresponding attribute node content, or text, PI or comment node content, is replaced by the *value_expression*. This function is analogous to the 'do replace' expression in the XQuery Update Facility proposal [10].

insertXMLBefore:
Syntax: insertXMLBefore(input_XML_instance, XPath_String,
source_XML_value_expression, [namespace_string])
insertXMLBefore() inserts the source XML value expression, which is evaluated to be a sequence of nodes, before the node matched by the XPath expression applied on the copy of the input XML instance, and returns the copied instance with the new nodes inserted. This function is analogous to the 'do insert before' expression in the XQuery Update Facility proposal [10].

appendChildXML:
Syntax: appendChildXML(input_XML_instance, XPath_string,
source_XML_value_expression [, namesapce_string])
AppendChildXML() inserts the source XML value expression, which is evaluated to be a sequence of nodes, as the last child of the node matched by the XPath expression applied on the copy of the input XML instance and returns the copied instance with new nodes appended. This function is analogous to the 'do insert as last' expression in the XQuery Update Facility proposal [10].

insertChildXML:
Syntax: insertChildXML(input_XML_instance, XPath_string,
target_element_or_attribute_name,
source_XML_value_expression [, namesapce_string])

insertChildXML() inserts the source XML value expression, which is evaluated to be a sequence of nodes, as the child of the node matched by the XPath expression applied on the input XML instance. For schema based input XML instance, the position to insert is determined by the XML schema. This function is analogous to the 'do insert into' expression in XQuery Update Facility proposal [10] except that the additional parameter *target_element_or_attribute_name* specifies the name of the element or attribute node in the node sequence of the source XML value expression. This can facilitate the XML update rewrite technique discussed in section 4.2. For non-schema based XML instance, the position to insert is implementation defined just as the 'do insert into' expression in the XQuery Facility Update proposal [10].

As a natural extension, we can also add new SQL/XML update functions, such as *InsertXMLAfter*() and *PrefixChildXML*() , which correspond to the 'do insert after' and 'do insert as first' expressions respectively in the current XQuery Update Facility proposal [10].

2.2 XML Update Function Examples

All examples use the following two tables *purchaseOrder* and *purchaseOrder2* whose DDL statements are shown in Table 1. Note *purchaseOrder* is an XMLType table whereas *purchaseOrder2* is a table consisting of two XMLType columns.

```
-- an XML table
create table purchaseOrder of XMLType;

-- a table with two XML columns: col1, col2
create table purchaseOrder2 (col1  XMLType, col2  XMLType);
```

Table 1 - purchaseOrder & purchaseOrder2 tables

A sample XML document stored in these tables is shown in Table 2. Each *purchaseOrder* document has a collection of *lineItem* elements and each *lineItem* has a collection of *part* elements, which form a classical master detail hierarchical relationship.

2.2.1 XML update function for Transformation

The first example shows the use of the deleteXML() function to transform an XML value without modifying the persisted XML document.

Example 1: Using deleteXML() for transformation

```
select deleteXML(value(po),
        '/purchaseOrder/lineItem/Parts')
from purchaseOrder po
```

The transformation is to delete all the *Parts* element nodes under '*/purchaseOrder/lineItem*'. This SQL/XML query is equivalent to the XQuery using the 'transform copy' and 'do delete' expressions in the XQuery Update Facility proposal [10] as shown in example 2.

Example 2: Equivalent of example 1 using XQuery Update Facility [10].

```
for $po in ora:view("purchaseOrder")
return
   transform
      copy $cpo := $po
      modify do delete
          $cpo/purchaseOrder/lineItem/Parts
      return $cpo
```

Note ora:view() is an Oracle XQuery extension that returns the XML document stored in each row of an XMLType table as an XQuery sequence of XML document nodes. It is conceptually the same as *fn:collection('purchaseOrder')* function which returns a sequence of XML document nodes stored under a *purchaseOrder* folder.

```
<purchaseOrder>
<reference>GHB3567</reference>
  <shipAddrCity>Oakland</shipAddrCity>
  <billAddrCity>Berkeley</billAddrCity>
 <lineItem itemNo ="34">
    <itemName>Furniture Set</itemName>
    <itemDescription> .... </itemDescription>
    <itemQuantity>3</itemQuantity>
    <itemUnitPrice>323.45</itemUnitPrice>
    <itemDiscount>0.15</itemDiscount>
          <Parts>
              <Part>
                <partName>Queen Size Bed</partName>
                <partQuantity>1</partQuantity>
                <partDescripton>.....</partDescripton>
              <Part>
              <Part>
                ...
              <Part>
             </Parts>
   <comments>This item discount rate is <b>approved</b>
by store manager</comments>
   <picture encode="bin.base64">GPAOP…</picture>
 </lineItem>
   ...
 </lineItem>
</purchaseOrder>
```

Table 2 – A sample Purchase Order XML document

2.2.2 XML update function for modifying persistent XML data

The next example shows a SQL update statement where the actual persisted XML document is updated.

Example 3: Use updateXML() to persistently change an XML instance.

```
update purchaseOrder  po
set value(po) = updateXML(value(po),
'/purchaseOrder/lineItem
 [itemNo =  "34"]/Parts/Part
 [partName = "Queen Size Bed"]/partDescription',
   XML('<partDescription>This is new style
                   bed</partDescription>')
)
where XMLExists('$doc/purchaseOrder
  [reference = "GHB3567"]'
passing by value value(po) as "doc")
```

Example 3 is a SQL update statement that persistently changes the XML documents stored in the *purchaseOrder* table. It uses the *XMLExists*() function in SQL/XML to identify the XML

documents to be updated, and then uses the *updateXML*() with an XPath expression to locate the node to update within each XML document. This update statement can be expressed using the new XQuery Update Facility [10] as shown in example 4 below.

Example 4: Equivalent of example 2 using XQuery Update Facility [10].

```
for $po in ora:view("purchaseOrder")
where $po/purchaseOrder[reference = "GHB3567"]
return
    do replace
          $po/purchaseOrder/
              lineItem[itemNo ="34"]/Parts/
              Part[partName = "Queen Size Bed"]
              /partDescription
    with
          <partDescription>This is new style
               Bed</partDescription>
```

When XML update functions are used for transformation, they can be fully composed because each XML update function returns a new XML instance that can be passed as an input to the next XML update function. Example 5 below illustrates this by composing the deletion of the *Parts* element node under */purchaseOrder/lineItem* in an XML document, followed by a deletion of the *reference* element node under */purchaseOrder*, using nested deleteXML() calls.

2.2.3 Composability of XML update function for transformation

Example 5: Composability of XML update SQL/XML functions.

```
select deleteXML(
         deleteXML(value(po),
         '/purchaseOrder/lineItem/Parts'),
      '/purchaseOrder/reference')
from purchaseOrder po
```

This SQL/XML query can be expressed using XQuery Update Facility [10] as shown in example 6 using the nested transform expression.

Example 6: Equivalent of example 5 using XQuery Update Facility

```
for $po in ora:view("purchaseOrder")
return
   transform
   copy $rpo : =
         (
          transform
          copy $cpo := $po
          modify
             do delete
              $cpo/purchaseOrder/lineItem/Parts
          return $cpo
         )
   modify do delete /purchaseOrder/reference
   return $rpo
```

2.3 XMLDiff () and XMLPatch() Function

For certain users of XML, especially for those who are using XML as a format for documents, they do not need a programmatic way to update the XML data in their database. Instead, they may download documents to the client, change them in a document editor, and transfer the changed version back to the database. In this case, users would like to be able to send just the changes to the document, instead of the full new document. So we provide two functions, XMLDiff() and XMLPatch().These functions are designed in the spirit of UNIX diff and patch utility programs on text files. XMLDiff() takes two XML instances and computes the diff between the two. XMLPatch() takes an XML diff format, which can be computed by XMLDiff() function, and applies the diff format to an XML instance to generate a new XML instance.

Consider the following example which uses XMLPatch() to apply XMLDiff to the document instances stored in *purchaseOrder* table:

Example 7: Example of using XMLPath to apply XMLDiff

```
declare diff_doc xmltype;
Set diff_doc = XMLType(
'<?xml version="1.0" encoding="UTF-8"?>
<xd:xdiff
xmlns:xd=http://xmlns.oracle.com/xdb/xdiff.xsd
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-
instance
 xsi:schemaLocation="http://xmlns.oracle.com/xdb/x
diff.xsd http://xmlns.oracle.com/xdb/xdiff.xsd">
<xd:update-node xpath="/PurchaseOrder/User/text()"
node-type="text">
 <xd:content>SKING</xd:content>
<xd:update-node>
</xd:xdiff>');

UPDATE purchaseorder po
set value(po) = XMLPatch(value(po), diff_doc)
WHERE xmlexists(
'$doc/PurchaseOrder[Reference="SBELL-
2002100912333601PDT"]' passing by value(po) as
"doc" ) ;
```

In this example, XMLPath() is a function which takes two input XML instances. It applies the second input value, which is an XML document describing the XMLDiff, to the first input value, which is the *purchaseOrder* document stored in *purchaseOrder* table. The update statement with XMLPatch() is equivalent to the following update statement without using XMLPatch():

```
UPDATE purchaseorder po
SET value(po) = updateXML(value(po),
'/PurchaseOrder/User/text()','SKING')
WHERE xmlexists(
'$doc/PurchaseOrder[Reference="SBELL-
2002100912333601PDT"]' passing by value(po) as
"doc" ) ;
```

In fact, internally, we rewrite XMLPatch() function into our XML update functions, such as updateXML(), whenever possible, to efficiently implement XMLPatch(). There is no equivalent of such facility existing in the current XML update proposal. The details of XMLDiff() and XMLPatch(), however, are beyond the scope of this paper.

Having illustrated the basic XML update functions, we will next show how these XML update functions may be used in the context of PL/SQL. Using these functions in PL/SQL provides the capability to update XML coupled with control flow, exception handling, providing snapshot semantics and transactions. Business logic is frequently predicated on these capabilities and is vital to building reasonably sophisticated XML applications.

3. EFFECTIVE XML UPDATE

The design of the XML update functions reflects decisions based on Oracle XML DB customer user cases for both XML query as well as update. Some of these design decisions are highlighted below where we describe how XML updates can be effectively implemented using these functions in SQL and PL/SQL.

3.1 Node Level update and transformation

A critical aspect of supporting XML update is to provide node level updates, deletes, and inserts, for the purpose of transforming XML documents as well as making persistent updates to stored XML documents. This design idea is consistent with the new transform expression in the XQuery Update Facility proposal [10]. Additionally, we allow XML update functions for transformation to be fully composable with any other SQL/XML function that return an XML instance similar to the full composibility of XQuery non-updating expressions. This was demonstrated in examples 1 and 5 in section 2 of this paper. However, to have these functions update the stored XML instances persistently, we leverage the SQL UPDATE statement to assign the result of the XML update function to a persistent XML column as shown in example 3. By using a SQL UPDATE statement for updating stored XML documents, we *conceptually update the entire stored XML document*. However, optimizations can be done in the actual implementation so that *only nodes of the original XML documents affected by the update functions are physically updated.* Such optimizations, which are referred to as XML update rewrite, will be discussed in section 4.

3.2 Declarative Capability for identifying documents and nodes for update

In RDBMS, XML document instances are stored in various XML columns of relational tables or XML tables. This is analogous to that of XML documents stored in various document collections for pure XML databases. An XML update operation involves identifying both the XML documents to be updated using some selection criteria, as well as the nodes to be updated within these XML documents based on additional selection criteria.

To locate XML documents to be updated, we use the *XMLExists*() function which embeds XQuery/XPath expressions, in the WHERE clause of the SQL statement. This is demonstrated in example 3 of section 2.

To locate nodes within each XML document to be updated, we use only XPath path expressions [18] to locate the nodes instead of relying on the full XQuery language. Though this may appear restrictive, in practice it is sufficient to identify the target nodes within the XML documents for update. This approach would also avoid the problem of having to define update semantics on nodes that are constructed by XQuery constructors had we allowed an arbitrary XQuery expression to locate nodes to be updated. This is because it is not semantically clear to define updates on constructed nodes which containing the copy of the persisted nodes or containing values computed from multiple persistent nodes. This is analogous to the study of update semantics of relational views [23] where views of computed value from the base tables are not updateable.
These XPath expressions are passed in as parameters to the XML update functions to locate persistent nodes to be updated in SQL update statement.

3.3 Snapshot, Concurrency and Transactional Semantics

To provide update snapshot semantics [10], that is determining the scope within which all expressions are evaluated before any updates are applied, we leverage the conventional snapshot semantics defined in SQL and PL/SQL.

PL/SQL consists of a sequence of statements. A statement can be a SQL statement or other procedural control flow statement. Each SQL statement defines a snapshot. That is, each SQL statement sees the side effects from all the previously executed SQL statements. However, within a SQL statement, side effects are not visible. That is, evaluation of WHERE clause of a SQL UPATE statement does not see the side effect from the current UPDATE statement. See the example below for a detailed description of these semantics. This statement based snapshot semantics naturally and clearly demarcates where side effects are visible. This semantics is not only used in PL/SQL but is also used in the context of embedded SQL in other hosted programming languages. The block expression along with sequential execution mode proposed in XQuery-P [13] is in principle the same as that of a PL/SQL statement, The equivalent statement concept and semantics also exists in SQL/PSM [15].

The PL/SQL logic in example 8 shown below uses a conditional expression to decide to execute UPDATE statement 1 on *purchaseOrder* table or UPATE statement 2 on *purchaseOrder2* table. Both table definitions are in table 1. After the update, it executes two SELECT statements to compute the document counts satisfying certain criteria and then returns the sum of the counts.
Example 8: Example of using XML update functions in PL/SQL.

```
Create or replace function
   opPODocuemnt(refid IN NUMBER, x IN XML)
Begin
  Declare totalItems number;
  declare totalItems2 number;
  Begin
    if
(x.existsNode('/purchaseOrder[count(lineItem) >
3]') = 1) then
-- UPDATE statement 1
   update purchaseOrder po
     SET VALUE(po) =
       deleteXML( insertChildXML(value(p0),
    '/purchaseOrder/lineItem', 'lineItem', x),
    '//comment/b')
       where
 XMLExists('$doc/purchaseOrder[reference = $rd]'
  passing by value value(po) as "doc",
    refid as "rd");
   else
-- UPDATE statement 2
```

```
      update purchaseOrder2 po2
      set po2.col1 =
      insertChildXML
        ('/purchaseOrder/lineItem',
         'lineItem', col1),
          po2.col2 = po2.col1
      where
      XMLExists('$doc/purchaseOrder
                   [reference = $rd]'
       Passing by value po2.col1 as "doc",
        refid as "rd");
■       end if;
    -- SELECT statement 3
    select count(*) into totalItems
    from purchaseOrder po
    where
     XMLExists('$doc/purchaseOrder/lineItem'
      passing BY VALUE VALUE(po) as "doc");

    -- SELECT statement 4
    select count(*) into totalItems2
    from purchaseOrder2 po2
    where
      XMLExists('$doc/purchaseOrder/lineItem'
        passing by value po2.col1 as "doc");

    commit;

    return totalItems + totalItems2;

  exception
   when others
    then
      rollback;
      dbms_output.putline('Errors occured ' ||
               to_char(SQLCODE));
  end;
end opPODocuemnt;
```

In this example, the side effect of the first two SQL UPDATE statements (UPDATE statement 1 and UPDATE statement 2) is visible to the following two SELECT statements (SELECT statement 3 and SELECT statement 4). However, the evaluation of the expression in the WHERE clause and the UPDATE target list for each SQL UPDATE statement do NOT see the side effects from the update.

In Example 8, all the XML column values passing into the *XMLExists*() functions in the WHERE clause of both UPDATE statement are the values **prior to any update being made**. The same is also true for the XML column value passed into the *updateXML*() function in the SET target list. Moreover, for the second expression '*po2.col2 = po2.col1*' in the UPATE target list of UPDATE statement 2, the *po2.col1* value on the right hand side of the assignment is still **the value prior to the update**.

These snapshot semantics during the SQL UPDATE statement are provided consistently for all data types including XML by the SQL system. The SQL system actually keeps track of both the old values prior to any updates being performed, and the new values post update so that a trigger fired as a result of the UPDATE statement can access both values.

We choose these SQL and PL/SQL snapshot semantics for XML as they are very natural to a large base of RDBMS users who are already familiar with the SQL and PL/SQL, PSM [15] semantics. In fact, there is no need to provide additional snapshot semantics for XML. Also it allows us to avoid the need for defining conflict detection and resolution semantics at the XML node level, as would have been the case if we had defined fine grained snapshot semantics for XMLType only as in XQuery ![12].

For the concurrency control of the XML column, we stay at the column level instead of defining fine-grained concurrency control at the node level within an XML column. This means we can leverage the classical row level locking model provided by the underlying SQL system. Defining concurrency at the node level is a challenge and merits future research.

For transaction control, we use the SQL transaction semantics with COMMIT and ROLLBACK statements to control the transaction boundary. In PL/SQL, a transaction is either implicitly started by a SQL statement or explicitly started by BEGIN WORK statement unless there has already been a transaction started. A transaction is explicitly committed or rolled back via explicit COMMIT or ROLLBACK statement. The change of DML statements in a transaction is not visible by other sessions until the transaction is committed.

3.4 Providing procedural logic using PL/SQL

It is inevitable that users want to leverage procedural logic, such as variable assignment, conditional statements, loop and exception handling logic, to build their XML applications in the database. We use PL/SQL assignment statements, conditional statements, loop statements and exception handling blocks to provide this capability. An example of using PL/SQL to do this has been shown in Example 7. Note the usage of 'if then else end if' conditional statement and the 'begin exception end' exception handling statement in example 7.

The current XQuery [11] and XQuery Update Facility [10] proposals do not provide the capability to write equivalent business logic as in Example 7, without having to embed XQuery and XQuery Update into another general purpose hosted programming language. This can however be achieved using the capabilities described in the XQuery-P [13] proposal. We will discuss this in detail in section 6.

3.5 Triggers and Constraints

Triggers and constraints are extremely useful to ensure consistency and for performing various actions such as auditing, security etc. during updates. Since we utilize the Oracle kernel APIs for handling updates, user level triggers and constraints can be defined over the XML columns just as for any other relational column, and they will be executed during the update. Customers have used these triggers to do schema validation of XML document instances, to ensure consistency of the XML values with respect to other SQL tables and to add new values to the XML documents not supplied by the input instance – for example to add a count of the number of items in a list.

Instead-of triggers also allow updating XML values created through views. This allows the user to write the actual logic of performing the update on the view in the trigger body that is a group of PL/SQL statements.
Such trigger mechanisms do not yet exist in the XQuery Update Facility Language [10]. Currently, the trigger body does not have any special means of identifying which specific nodes in the tree have been updated. This is a subject for future research.

3.6 Comparison of XQuery Update and Oracle SQL/XML Update function

In summary, this is the comparison of the XQuery Update Facility with Oracle SQL/XML update functions extension.

- Both provide for locating and updating XML nodes in a declarative manner for either transformation or modification. The primitive from both approaches are similar.
- Oracle SQL/XML update functions support using XPath PathExpression to locate XML nodes for update.
- Oracle SQL/XML update functions do not support rename expression that changes the name of an element or attribute node.

Oracle SQL/XML update function support diff and patch style of transforming and updating XML that XQuery Update Facility does not support.

4 EFFICIENT XML UPDATE

Having discussed the functionality of effective XML updates in the previous section, we will now discuss how to provide this functionality efficiently.

4.1 XML Update Rewrite Concept

Oracle XML DB supports various XML storage and indexing methods to support the diversity of XML application characteristics. XML content in Oracle XML DB may be stored in the text form in a CLOB column, as objects in object-relational tables, in a binary XML form in a BLOB column, or in a hybrid form in which certain structured parts are stored object relationally and non-structured parts stored in a LOB [1, 5]. As XML covers both structured and un-structure data, there is **no one-size-fits-all solution**. Figure below shows a diagram of the various XML storage and indexing mode [27].

Document \ **Parts of the document**	*unstructured*	*structured*
unstructured	BLOB-Tree/ Path-Value Index	BLOB-Tree/ XMLTable Index
structured	ObjectRelational -BLOB-Tree/ Path-Value Index	Object-Relational

The **functional evaluation** of an XML update function constructs the stored XML into a DOM tree and then applies the specific changes on the tree. The SQL update then takes the modified tree and then replaces the target XML column with it. This method is presently used for XML stored in a CLOB. For other storage models, the XML update functions can be optimized to update only the components that are the targets of the update functions, instead of replacing the entire tree as described above. This is referred to as the **XML update rewrite** technique that allows for very efficient updates of XML. Our challenge is to *deliver XML update rewrite solutions for different XML storage and indexing models*.

4.2 XML update rewrite for Object Relational Storage

4.2.1 Basic Algorithm

In cases where there is an XML schema associated with the XML content, Oracle XML DB allows for the XML content to be stored object relationally. In this case, the underlying object relational storage table meta-data information is known during SQL statement compilation time. This information can be exploited to rewrite the update of the XML content based on the XPath in the XML update functions, so that only the underlying storage tables associated with the targets of the update are modified.

Consider the example 3 in which a DML statement is applied to the XML table with *purchaseorder* XML documents stored object relationally. In the object relational storage, collection items are typically stored as a **nested collection table** forming a primary key - foreign key relationship with the parent table.

Conceptually, we execute this XML update DML statement in two steps: we evaluate an outer query with the WHERE clause of the original DML statement to locate the *purchaseOrder* table rows that needs to be updated, in effect identifying the XML documents that need to updated. This query can be efficiently executed by rewriting XQuery/XPath operations in the WHERE expression as described in [2,19].

Then for each row in the *purchaseOrder* table identified as above, we recursively run another DML statement that actually updates the underlying storage tables for the components that belong to this *purchaseOrder* XML document row. This DML statement may itself recursively invoke other DML statements to perform the update in a cascading fashion.

The outer query shown below identifies all the primary key column values for all affecting *purchaseOrder* table rows. Note the original *XMLExists*() operator has been implemented as a SQL predicate on the *reference* column using the technique described in [2, 19].

```
select purchaseOrder.pkeyCol
from purchaseOrder po
where po.reference = 'GHB3567'
```

For each *purchaseOrder.pkeyCol* value from the outer query, we pass the value as a bind variable to the following DML statement that actually updates the underlying storage table *partTab* for *Part* elements through the intermediate storage table *lineItemTab* storing all *lineItem* elements.

```
update partTab pt
set pt.partDescrtiption = 'This is new style bed'
where pt.partName = 'Queen Size Bed'
and
exists
 (select 1
  from lineItemTab li
  where li.pkeyCol =  pt.fkeyCol
  and li itemNo = '34'
   and li.fkeyCol = :po.pkeyCol
                    -- bind variable passed in
   )
```

The key to generating the above DML statement is to break the original XPath in the XML update function into the **target component** along with multiple **intermediate collection component**s, and then to traverse these components in the reverse order starting from the target component to generate the DML statement.

The target component of the XPath, in this case, is the '*Part/partDescription*', which identifies the target table *partTab* and its column *partDescription* for update. Each intermediate collection component of the XPath, in this case, the '*lineItem*', maps to an EXISTS subquery joining on the foreign key columns in the nested collection table with the primary key column of the parent table. The predicate for each component can be formulated using the *XMLExists*() operator and is then implemented by applying SQL predicates on the object relational tables using the techniques described in [2,19]. Traversing these intermediate components in the reverse way makes the join all the way to the root table in which the root primary key column values for the updated documents are computed by the outer query.

In general, given an XPath */s1/c1[pred1]/c2[pred2]/c3[pred3]* used in XML update function, where *ci* means collection element access and *si* means scalar element access and *CI_table* is the nested storage table for the collection element *CI*, we formulate the SQL DML statement using the following template:

```
<DML action> ON C3_table
WHERE <SQL predicate for pred3>
AND
EXISTS
(SELECT 1
 FROM C2_table
 WHERE <SQL predicate for pred2>
 AND C2_table.pkey = C3_table.fkey
 AND
 EXISTS
 (SELECT 1
  FROM C1_table
  WHERE <SQL predicate for pred1>
  AND C1_table.pkey = C2_table.fkey
  AND C1_table.fkey = :root_table.pkey – passed in
as bind variable from the outer query))
```

The <DML action> is INSERT, DELETE, UPDATE SQL DML statements depending on if the XML DML functions is *insertChildXML*(), *deleteXML*() or *updateXML*() respectively. Note that although there are multiple EXISTS subqueries in the generated DML statement, the relational optimizer is smart enough to un-nest a subquery into a semi-join so that different join strategies or even join orders can be explored to speed up the DML statements [16].

The other key point is that the DML action on the target table itself may recursively run its cascading DML statements. For example, deleting the *lineItem* elements from the *lineItemTab* causes the cascading deletion of all the parts elements stored in *partTab*. The invocations of the outer query, the DML statement and all of its cascading DML statements are all done through internal Oracle kernel APIs for DMLs. This is not only much faster than using user level triggers but also allows users to still use user level triggers to execute their application specific logics. Had we used trigger mechanisms for implementing XML update, then we would have to distinguish system level triggers and user level triggers.

Although there are similarities between updating XML views over relational tables [3][6] and updating XML stored object relationally as we described, there are some **key differences** and issues that are not addressed by updating XML views over relational tables. We explain each of them below in turn.

4.2.2 Preservation of XML fidelity & Order

The object relational storage of XML provided by Oracle XML DB has the option to provide *XML fidelity* instead of just the relational fidelity that classical XML to relational mapping supports. We internally create binary columns called **positional descriptors** to store information: processing instructions, comments, namespaces, the element content model, the empty content of an element (to be distinguished with the case of absences of the element), for each XML schema type. These binary columns contain an encoding of the information and are updated and maintained through internal SQL executed along with the updates performed during the XML update rewrite process. Furthermore, when the collection element is stored with *order fidelity*, we internally create an array index column for the table containing collection elements. We allow decimal values on the array index columns so that when collection elements are deleted or inserted in the middle, we don't need to always compact or shift elements to delete positions or to make new positions. The concept of the positional descriptor is the key to supporting XML document order and fidelity with Oracle object relational storage of XML.

4.2.3 Support for Recursive Schemas

Many XML applications require the handling of XML with recursive schema elements. We support object relational storage of XML with recursive elements by using the object id and object reference concepts in Object Relational SQL [22]. This is done by storing all recursive elements with the same element name into the same XML table. Each row of the XML table has an object id that can be stored and referenced as an **XML reference**. We then store the parent and child relationship among these recursive elements in a nested collection table that stores the XML reference instead of the actual collection content.

Consider the following recursive XML documents shown in table 3.

```
<section>
   <c> c1 </c>
  <section>
    <c>c 2 </c>
  </section>
   <section>
     <c> c3 </c>
        <section>
           <c> c4</c>
        </section>
    </section>
 </section>
```

Table 3 - XML with recursive element

There are two object relational tables generated to store this XML: the XML table storing all the section elements at any level and the nested collection table which keeps track of the parent and child relationships among these recursive elements. The content of these two tables is shown in tables 4 and 5.

Object Id	C	Pkey
oid1	c1	P1
oid2	c2	P2
oid3	c3	P3
oid4	c4	P4

Table 4 - XML table for storing All Section Element

Fkey	XML Reference
P1	oid2
P1	oid3
P3	oid4

Table 5 – Nested Collection Table storing XML references

Then the same XML update rewrite algorithm can be used to rewrite XML update functions as for the case of non-recursive elements. The difference is that the join is made between the recursive element table and the nested collection table through the join key *section_table.ObjectId = LinkTable.XMLReference*.

4.2.4 Partial XML Update Rewrite

For XML that is stored in hybrid mode, the non-structured part of the XML is stored in a LOB datatype. For example, the *comment* element with mixed content in the *purchaseOrder* document may be stored in a LOB. Users may then want to run the following deleteXML() statement to delete all the *b* elements within the *comment* element:

```
UPDATE purchaseOrder po
SET VALUE(po) = deleteXML(VALUE(po),
'/purchaseOrder/lineItem[@itemNo =
"34"]/comments//b')
WHERE XMLEXISTS(VALUE(PO),
'/purchaseOrder[reference = "GHB3567"]'
```

Since the structure of the *comment* element is opaque and stored in a LOB, we have to use functional evaluation to update the XML value. However, instead of updating the entire XML document via functional evaluation, we use the technique called "*partial XML update rewrite.*" The key idea in this technique is to utilize the rewrite technology to update the smallest number of nodes.

Observe that the initial components are stored object relationally, so we split the XPath used in the *deleteXML*() function into a sequence of steps that are rewritable based on the object relational structural types, and steps that are functionally executed as they lead into a LOB column.

In this example, the initial path '*/purchaseOrder/lineItem [@itemNo="34"]*' is rewritable but the remaining path '*comments//b*' is not. The idea of the partial XML update rewrite is to compute the XML components that the rewritable part of the XPath selected efficiently through XML query rewrite and then update the rest using functional evaluation. The internal executed DML statement via partial XML update rewrite is shown below:

```
UPDATE lineItemTab li
SET li.commentLobCol = XmlToLOB(
     DELETEXML(
      ( SELECT XMLAgg(VALUE(lir) )
        FROM lineItemTab lir
         WHERE lir.itemNo = "34"
          AND lir.fkeyCol = :po.pkeyCol
                    -- bind variable passed from
                    -- the outer query ),
     'lineItem/comment//b'))
```

Note that in this DML statement, only the *commentLobCol* LOB column of table *lineItemTab* is updated. The *deleteXML*() function needs to only evaluate the non-rewritable XPath, which is applied to the XML constructed by the XMLAgg() scalar subquery that constructs the *lineItems* with itemNo = "34". The new XML comment element content is converted into a LOB via *XMLToLOB*() internal operator and this new value replaces the old *commentLobCol* value.

4.3 XML Update rewrite for Binary XML storage

4.3.1 Efficient piece-wise BLOB update

XML can be encoded in binary form and stored in a BLOB column. The binary form of XML is based on tag tokenization and the base data value is stored in native form when the XML schema is available [5], both of which leads to data compression. One challenge for updating binary XML stored in BLOB column is to provide scalable and efficient piece-wise data update for BLOB column. Traditionally when data is stored as BLOB, an update of a portion of the BLOB data results in a update of the full BLOB column which can be very costly and expensive. Therefore, *a new BLOB infrastructure* is provided for binary XML so that an efficient piece-wise update of BLOB column is feasible. In the new BLOB infrastructure, the BLOB offset and length used to access the piece-wise BLOB data is no longer a physical offset. It is rather a logical offset whose physical offset is computed from the internal mapping tables tracked by the new LOB infrastructure. When a piece of BLOB data is updated, the new data value is inserted at the end of the BLOB with a new physical offset, the new physical offset is entered into the mapping table for the original offset which is used to access the updated data. The original offset now becomes a logical offset. This strategy effectively makes the logical BLOB organized physically like a linked BLOB data fragments. The new BLOB infrastructure can do re-organization when necessary so as to reduce internal fragmentation. This strategy makes update of XML data scalable and efficient.

4.3.2 XML update with XMLIndex

XMLIndex [5] can be created on binary XML. The logical XMLIndex is physically implemented as a *path table* with each row storing information of a node in XML. Each row keeps track of the following information for a node: its document id, the ordered key representing hierarchical position of the node in XML, the path identifier corresponding the named path from the root to the node. If the node is the leaf element node or attribute node, its value is stored in the column of the path table. If the node is complex, its value stored in the column of the path table is *the underlying BLOB locator* that can be used to fetch the node within the BLOB efficiently. When XML data is updated, the XMLIndex needs to be updated as well. However, since the BLOB locator contains the logical offset instead of the physical offset from the new underlying BLOB infrastructure, therefore, the need to massively update the BLOB locator is reduced drastically during XMLIndex maintenance.

The XML update process still has an outer query that identifies the rows of XML documents that need to be updated. The *XMLExists*() in the WHERE clause can be efficiently evaluated by using the XMLIndex. Once the XML document is identified, we use the XMLIndex or streaming evaluation of XPath to identify the nodes within each XML document on which modification need to happen. Each XML update function results in a set of piece-wise BLOB update operations consisting of an offset and length of the old data to be replaced, along with the new value and its length. The underlying new BLOB infrastructure described in section 4.3.1 supports piece wise update of the BLOB without requiring a full replacement of the original BLOB. After the stored XML BLOB update, the XMLIndex is also maintained to keep the index consistent with the underlying data. This may result in deletion of rows from the path table when nodes are deleted or insertion of new rows into the path table when new nodes are added. However, updating BLOB locators is kept as minimal as discussed above.

Having looked at the performance aspects of updating XML persistently, we now discuss the issue of providing efficient support for transforming XML with the XML update function.

4.4 Challenge of Efficient XML Update for XML Transformation

Although transforming XML does not involve actual XML modification persistently, optimizing XML transform queries is challenging. The functional evaluation of the XML update function for transformation, by applying the basic update operations on the materialized XML DOM tree is not efficient. Similar to the XQuery/XPath Query rewrite strategy on top of relational storage of XML or XML view over relational data [2, 19], we need to push down the XML update operations while constructing the XML from its underlying object relational tables.
For example, the functional evaluation of example 1 is to construct the XML DOM tree from the underlying object relational storage tables and then apply the *deleteXML('/purchaseOrder/lineItem/Part')* operator. However, intuitively this *deleteXML*() implies that we need **not** select the content in the *partTab* storage tables. Therefore, the most efficient way of executing example 1 is to rewrite it as the following query:

```
select (select xmlelement("purchaseOrder",
       Xmlforest(reference),
         (select xmlagg(xmlelement("lineItem",
                  xmlattributes(itemNo
                              as "itemNo"),
                   xmlforest(itemName,
                             comments, picture))
          from lineItemTab li
          where li.fkeyCol = po.pkeyCol)
from purchaseOrder po
```

Note that the scalar *XMLAgg*() subquery to construct the *part* elements is NOT present, as the *deleteXML*() operation requires the deletion of the *part* element. Although this simple example can be worked out this way, to support such rewrites for full nesting of XML update functions would be a challenge. Additional rewrite algebra rules need to be established for these cases. The design of such algebra rules requires future work.

5. PERFORMANCE EVALUATION

To demonstrate that XML update rewrite performs much faster than functional evaluation, we have performed the following experiments that compare *deleteXML*() and *insertChildXML*() DML statements on the *purchaseOrder* table which is stored object relationally.

Figure 1 shows the performance comparison of XML update rewrite vs. functional evaluation on the following *deleteXML*() DML statement:

```
UPDATE purchaseOrder po
SET VALUE(po) = deleteXML(VALUE(po),
'/purchaseOrder/lineItem[@itemNo = "34"]')
WHERE XMLEXISTS(VALUE(po),
'/purchaseOrder[reference = "GHB3567"]')
```

Figure 2 shows the performance comparison of XML update rewrite Vs. functional evaluation on the following *insertChildXML*() DML statement:

```
UPDATE purchaseOrder po
SET VALUE(po) = insertChildXML(VALUE(po),
'/purchaseOrder/lineItem', 'lineItem',
XML('<lineItem>…</lineItem>'))
WHERE XMLEXISTS(VALUE(po),
'/purchaseOrder[reference = "GHB3567"]')
```

The experiments are performed with the number of *lineItems* in the nested collection table as 10, 25 and 100. Both Figure 1 and Figure 2 show that the XML DML rewrite consistently performs better than functional evaluation. Furthermore, the performance gained from DML rewrite scales linearly in response to the number of *lineItems* whereas functional evaluation does not. This is understandable because the XML update rewrite is implemented as insertion or deletion on the underlying target collection tables and thus inherits the scalability of the underlying relational storage. Functional evaluation, on the other hand, does not scale as the size of XML documents increases.

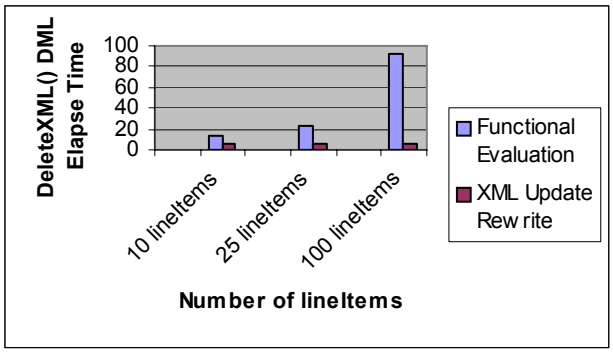

Figure 1 - DeleteXML() Functional Evaluation Vs. XML Update Rewrite

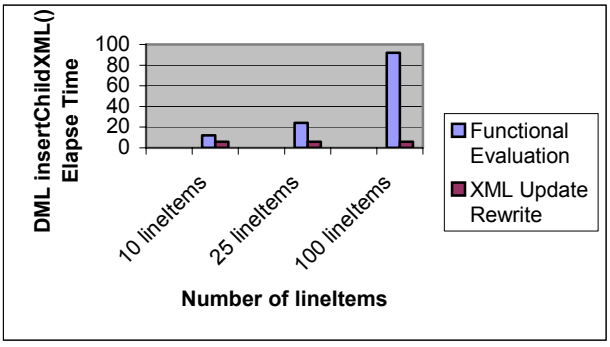

Figure 2 - insertChildXML() Functional Evaluation Vs. XML Update Rewrite

6. RELATED WORK COMPARISON

The XML update functions provided by Oracle XML DB is conceptually analogous to the new basic updating expression proposed by the XQuery Update Facility [10] to effectively support node level XML transformation and update. We currently do not support the *rename* expression, which is proposed by both 'updating XML' paper [3] and the XQuery Update Facility [10].

However, to build sophisticated XML applications, besides the current XQuery and its update facility proposals [10], users definitely need basic programming facilities, such as ordered execution, variable assignment, full control flow, exception handling, transactional semantics, triggers and constraints. We have selected PL/SQL as the natural RDBMS procedural language to provide these facilities and integrate our SQL/XML support tightly with PL/SQL.

XQuery-P [13] adds the sequential execution mode, explicit atomic block expression, while expression etc, all of which are in principle aligned with PL/SQL and PSM [14]. The XQuery![12] proposal, which allows side-effect expressions to be used everywhere, also adds ordered semantic mode as one of the modes in addition to the non-deterministic semantic mode and conflict-detection semantic mode. We have purposefully chosen to not to implement side-effect expressions and non-ordered execution mode, as we feel that this will actually confuse the application users and force them to write many non-optimizable programs.

Analogous to SQL, the XQuery Update Facility proposal [10] restricts side-effect expressions in certain XQuery expressions so as to achieve the proper balance between theoretical elegance and practicality for XML update.

If we consider industrial products, Microsoft Yukon SQL Server [20] has provided the capability of XML node level update with the single XML *modify*() method embedding the new XQuery update expressions. IBM DB2 Viper [21] does not appear to support node level updates yet.

There are papers [3, 6, 7] which address updating XML views over relational data. However, all of the approaches use the RDBMS as a black box and the XML to relational data mapping discussed there supports only relational fidelity. Paper [6] supports XML node updates occurring at leaf nodes only. Our XML update rewrite method is fully integrated inside the RDBMS kernel without relying on user level cursors and user level triggers and thus delivers superior DML performance. This is one of the key performance advantages that user level XML shredding solutions could not achieve. We support node update at non-leaf nodes level. Furthermore, we leverage constructs from object relational databases to fully support XML fidelity beyond just the relational fidelity. We also address the efficient XML update issue for hybrid XML storage model combining both the object relational and BLOB storage through partial XML update rewrite and new BLOB infrastructure.

7. FUTURE DIRECTIONS

As the XQuery Update Facility [10] proposal becomes a recommendation, it is important to integrate it with the future SQL/XML standard so that users can do XML node level update and transform in SQL/XML. Our thoughts are to leave the current *XMLQuery*() and *XMLExists*() functions to only include non-updating XQuery expressions along with the new transform expression in XQuery to perform transformation. For updating persistently stored XML values, we propose a new *XMLUpdate()* SQL function, which includes either basic updating expression or updating expression, to update input XML. We feel that this would be sufficiently powerful and flexible to support updating XML values within the RDBMS.

Furthermore, although SQL/XML with PL/SQL and PSM can provide all the facilities to build XML applications from the perspective of SQL/XML in RDBMS, native XML users may prefer to build XML applications without SQL. This can be achieved by using XQuery, XQuery Update Facility and XQuery-P [13, 26] all together. For those users who do not want to learn or use SQL, the RDBMS can be easily enhanced to support **XQuery stored procedures and functions** whose bodies are coded in pure XQuery-P without reference to SQL. Relational data can be accessed as XML in the XQuery stored procedures using functions such as *ora:view*(). This function is already used today in Oracle XML DB to get an XML view of the relational data within XQuery expressions [19]. This achieves the best XQuery and SQL duality in the RDBMS.

8. CONCLUSION

In this paper, we have presented the node level XML update and transformation functionality supported by Oracle XML DB and how they compare with the past research on XML update and the current XQuery update facility proposal. Furthermore, we illustrate the necessity of providing full procedural logic support in an XML programming language in order to build XML applications and show how we achieve this through the integration of SQL/XML with PL/SQL. We have also presented the XML update rewrite technique to efficiently support XML node level modification. With the introduction of XML update, there will be much work to be done both on SQL/XML side and XQuery side to make SQL/XML-PSM and XQuery-P truly a programming model and language for enterprise XML computing.

9. ACKNOWLEDGEMENTS

We gratefully acknowledge the contributions of all the members of the Oracle XML DB development and product management teams.

10. REFERENCES

[1] R. Murthy, S. Banerjee: XML Schemas in Oracle XML DB. VLDB 2003

[2] M. Krishnaprasad, Z. Hua Liu, A. Manikutty, J. Warner, V. Arora, S. Kotsovolos: Query Rewrite for XML in Oracle XML DB,VLDB 2004

[3] I. Tatarinov, Z. G. Ives, A. Y. Halevy, D. S. Weld: Updating XML, SIGMOD 2001

[4] F. Ozcan, R. Cochrane , H. Pirahesh, J. Kleewein, K. Beyer, V. Josifovski , C. Zhang: System RX: One Part Relational, One Part XML, SIGMDO 2005

[5] 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

[6] V. P. Braganholo, S. B. Davidson, C. A. Heuser : From XML view updates to relational view updates: old solutions to a new problem, VLDB 2004

[7] V. P. Braganholo, S. B. Davidson, C. A. Heuser: UXQuery: Building Updatable XML Views over Relational Databases

[8] M. Rys: XML and relational database management systems: inside Microsoft SQL Server 2005.

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

[10] XQuery Update Facility: http://www.w3.org/TR/xqupdate/

[11] XQuery: http://www.w3.org/TR/XQuery/

[12] G. Gheli, C. Re, J. Simeon: XQuery! : An XML query language with side effects, http://XQuerybang.cs.washington.edu/papers/XQueryBang.pdf

[13] D. Chamberlin, M. Carey, D. Florescu, D. Kossmann, J. Robie : XQueryP: Programming with XQuery, XIME-P 2006

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

[15] Database language – SQL – Part4: Persistent Stored Modules (SQL/PSM). ANSI/ISO/IEC 9075-4-1999.

[16] W. Kim. "On Optimizing an SQL-Like Nested Query", ACM TODS, September, 1982.

[17] http://www.w3.org/TR/xpath20/

[18] http://www.w3.org/TR/XQuery/#id-path-expressions

[19] Z. Hua Liu, M. Krishnaprasad, V. Arora: Native XQuery Processing in Oracle XML DB. SIGMOD 2005

[20] http://www.microsoft.com/sql/default.mspx

[21] http://www128.ibm.com/developerworks/db2/library/techarticle/dm-0602saracco/

[22] M. Stonebraker, P. Brown, D. Moore: Object-Relational DBMSs, Second Edition Morgan Kaufmann 1998

[23] F. Bancilhon, No. Spyratos.: Update semantics of relational views. ACM Transactions on Database Systems, 6(4), Dec. 1981

[24] http://www.oracleplsqlprogramming.com/

[25] http://www.unix.org.ua/orelly/oracle/prog2/index.htm

[26] D. Chamberlin, M.J. Carey, M. Fernandez, D. Florescu, G. Ghelli, D. Kossmann, J. Robie: XqueryP: AnXML application Development Language http://2006.xmlconference.org/proceedings/38/presentation.pdf

[27] Z. Hua Liu, Muralidhar Krishnaprasad, Hui J. Chang, Vikas Arora: XMLTable Index - An Efficient Way of Indexing and Querying XML Property Data, ICDE 2007.