Native XQuery Processing in Oracle XMLDB

Zhen Hua Liu Muralidhar Krishnaprasad Vikas Arora

zhen.liu@oracle.com muralidhar.krishnaprasad@oracle.com vikas.arora@oracle.com

ABSTRACT

With XQuery becoming the standard language for querying XML, and the relational SQL platform being recognized as an important platform to store and process XML, the SQL/XML standard is integrating XML query capability into the SQL system by introducing new SQL functions and constructs such as XMLQuery() and XMLTable. This paper discusses the Oracle XMLDB XQuery architecture for supporting XQuery in the Oracle ORDBMS kernel which has the XQuery processing tightly integrated with the SQL/XML engine using native XQuery compilation, optimization and execution techniques.

1. Introduction

With the introduction of the XML datatype for typing XML data in SQL via SQL/XML standard[6][7][8], Oracle XMLDB enables users to store XML natively in object relational DBMS via the use of XMLType tables and views. Furthermore, users can convert their relational data into XMLType views using SQL/XML publishing functions, such as XMLElement(), XMLConcat() etc., defined by SQL:2003 [8] standard. There is ongoing work in the SQL/XML committee to provide XML querying capabilities in the next version of the SQL standard using XQuery. A new SQL function called *XMLQuery*() and a from-clause construct - *XMLTable* have been proposed in this regard [9]. XMLQuery() function allows an arbitrary XQuery [1] to be embedded directly in SQL to query and construct XML data. XMLTable construct, on the other hand, enables the users to convert the result of XQuery into a virtual relational table.

Supporting XMLQuery() and XMLTable construct in SQL imposes new challenges on the RDBMS engine. A straightforward approach, referred to as the *coprocessor* approach, is to simply embed an off-the-shelf XQuery processor and treat the XQuery related functions as a black-box - sending the queries over to the embedded processor and getting the results. Although this approach is conceptually clean and easy to implement, it does not leverage the full potential of RDBMS as a query optimization and execution engine and suffers from intrinsic performance limitations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD2005, June 14-16, 2005, Baltimore, Maryland, USA
Copyright 2005 ACM 1-59593-060-4/05/06…$5.00.

In this paper, we propose a **native XQuery compilation, optimization and execution** approach in the RDBMS by rewriting XQuery into SQL operators and constructs with XML extensions, which are then amenable to optimization by the underlying relational optimizer and efficiently executable by the underlying relational execution engine. This approach enables us to tightly integrate XQuery and SQL/XML support within the ORDBMS kernel and delivers performance that is orders of magnitude faster than the coprocessor approach. This also enables us to utilize standard indexes that are present on the underlying data and enable relational performance optimization techniques such as parallel query and partitioning on XML queries.

The rest of the paper is organized as follows. In section 2, we illustrate the key concepts of native XQuery compilation optimization and execution with examples. In section 3, we describe related work and comparisons with our approach. In section 4, we describe the XQuery native compilation process. In section 5, we look at the native XQuery compilation technique and the algebra optimizations. In sections 6 and 7, we draw the conclusion with acknowledgements.

2. Key Concepts – native XQuery compilation, optimization and execution

2.1 Co-processor Approach

The coprocessor approach logically represents the semantics of running XQuery inside the RDBMS. In this approach, the evaluation of the XMLQuery() function is done by invoking the XQuery processor to execute the XQuery. This approach has inherent performance and scalability limitations due to the following reasons:

Storage Optimization: Because the XQuery processor is completely opaque to the rest of the RDBMS engine, it may not be able to take advantage of the physical storage of the XML input data. In many cases, the XML data as input to the XQuery processor may have to be constructed by the SQL processor at run time even though the underlying storage of the XML data may have been shredded into object relational tables or may have been defined as an XMLType view over the relational data.

Intra-Query Optimization: Having a separate processor distinct from the SQL engine may prevent the use of standard relational optimization technology such as constant folding, view merging, subquery optimizations, distributed query processing, parallel query, partition pruning and common sub-expression elimination within the XQuery functions. These may have to be re-implemented as part of the XQuery processing separately.

Inter-Query Optimization: Furthermore, even if the embedded XQuery processor is able to optimize the single XQuery passed to it, it may not be able to optimize the XQuery in the global context

of the original SQL statement which invokes the XMLQuery function. A SQL statement can invoke multiple XMLQuery() functions and the output of one XMLQuery() can become the input of another XMLQuery() in the same SQL statement. This occurs naturally in the presence of views in relational system where one view may use XMLQuery() to query result from another view which in turn uses XMLQuery() to query another XMLType tables, views or columns. In the case of supporting XMLTable construct, the output of the XMLQuery() computing the row XML value is passed as input to the multiple invocations of XMLQuery() to compute each column value.

2.2 Native Compilation

The optimal strategy for supporting XMLQuery() and XMLTable, is to compile the embedded XQuery into SQL constructs and operators with XML extensions as needed, so that the entire SQL statement that includes the XMLQuery() and XMLTable constructs can be optimized as a whole. This approach fits naturally inside an RDBMS environment.

As SQL is a compiled language, it makes sense to do the static type analysis of the XQuery for each XMLQuery() and XMLTable invocation during SQL compilation . As a result, XQueries can then be compiled into a set of subquery blocks and operators that can be algebraically optimized in the context of the global SQL statement. This strategy works gracefully within the model of view expansion and merge techniques in relational systems as it enables the pushdown of predicates and optimizes the result by eliminating unnecessary intermediate materializations of XML values.

2.3 Example

Consider the following example: Table 1 shows an XML view *purchaseOrderXML* which is constructed using SQL/XML publishing functions over the relational tables *purchaseorder* and *lineitems* constituting the classical master detail relationship.

Table 2 shows an example of a SQL statement with XQuery embedded in the XMLQuery() function to find the *ShippingAddress* of all the *purchaseOrder* XML instances that have purchased the '*CPU*' item.

Table 3 shows an example to convert the XML document instances into relational tables via XMLTable construct.

```
CREATE VIEW purchaseOrderXml  AS
SELECT XMLElement("PurchaseOrder",
                XMLAttributes(pono AS "pono"),
           XMLElement("ShipAddr",
         XMLForest(street AS  "Street", city AS  "City", state AS "State")),
          ( SELECT XMLAgg(
              XMLElement ("LineItem",XMLAttributes(lino as "lineno"),
                     XMLElement("liname", liname)))
           FROM  lineitems l WHERE  l.pono = p.pono)
         ) AS  po
FROM purchaseorder p
```

Table 1 - purchaseOrderXML view

```
SELECT XMLQuery(
        'for $i in ./PurchaseOrder
        where $i/LineItem/liname = "CPU"
        return $i/ShipAddr' PASSING  BY VALUE p.po
           RETURNING CONTENT)
FROM  purchaseOrderXml p
```

Table 2 – XMLQuery() example

```
SELECT  xt.lineno, xt.liname
FROM  purchaseOrderXml p,
 XMLTABLE(  'for $i in ./PurchaseOrder/LineItem return $i'
      PASSING p.po
     COLUMNS
          lineno NUMBER PATH '/LineItem/@lineno',
          liname VARCHAR(20) PATH '/LineItem/liname'
 ) xt;
```

Table 3 – XMLTable() example

```
SELECT
   ( SELECT XMLElement("ShipAddr",
        XMLFOREST(street AS "Street", city AS "City", state AS "State"))
    FROM  dual
    WHERE EXISTS (
       SELECT NULL
       FROM lineitems l
      WHERE l.liname='CPU' AND l.pono = p.pono)
 )
FROM  purchaseorder p
```

Table 4 - Rewritten Query for XMLQuery in table 1

```
SELECT l.lino AS "LINENO", l.liname AS "LINAME"
FROM purchaseorder p, lineitems l
WHERE l.pono = p.pono
```

Table 5 - Rewritten Query for XMLTable in table 2

The SQL equivalent for the natively compiled XQueries in tables 2 and 3 are shown in tables 4 and 5 respectively. This example illustrates the XQuery native compilation process to convert the original SQL statement into a semantically equivalent relational query using SQL/XML publishing functions to construct the result XML. The equivalent query can then be optimized by a classical relational optimizer and executed natively by tuple oriented relational execution engine. This strategy enables us to leverage the mature object relational technology and SQL/XML infrastructure inside Oracle XMLDB to support native XQuery execution.

3. Related Work Survey and Comparison

SQL Translation versus Native Compilation

There are many published papers on XQuery implementation [10][11][12][13][14][18][22]. Most of them build an XQuery engine in the middleware interacting with relational DBMS in the backend [11][13][14][18][22]. In cases where the XQuery engine needs to communicate with the SQL backend, the XQuery is typically translated into a set of classical relational SQL statements that are then sent to the backend RDBMS for execution. A translation to SQL essentially creates a SQL string and then sends it to the server for compilation as a regular SQL statement. Our native compilation approach on the other hand compiles XQueries into the same internal data structures as SQL such as sub query blocks and SQL operators with extensions as needed. The advantage of native compilation is that SQL,

SQL/XML, XQuery merely become language syntaxes, all of which are converted into the same underlying structures for compilation, optimization and execution. The Oracle XMLDB has the XQuery framework built directly in the ORDBMS kernel and delivers SQL/XQuery duality and interoperability using the SQL/XML infrastructure.

Our approach is based on the following key principles:

1. **XQuery Data Model based XMLType:** We leverage the fact that SQL/XML has defined a new XML type as a first class datatype in the SQL type system. The Oracle XMLDB provides the XML type infrastructure which can natively support the XQuery data model [2] inside the relational engine.
2. **XQuery SQL Operators:** From XQuery data model based XML type, we can develop new SQL operators that can consume and generate XQuery data model instances and use them for the implementation of XQuery operations that are foreign to the SQL system. For example, we create internal SQL operators that can do XQuery Sequence type matching expressions. This is different from a direct translation of XQueries to SQL which may not always be feasible without extending the SQL language to add new primitives. For example, paper [18] observed that relational engines need to add primitives to support construction of XML document fragments.
3. **Efficient XML construction:** We leverage the SQL/XML publishing functions as the basis for XQuery constructors by compiling XQuery constructors into the same operators underlying the SQL/XML publishing functions. This approach allows us to leverage optimizations implemented for efficient execution of SQL/XML publishing functions such as top-down stream evaluation [17]. Again, this is different from most middleware solutions which build the XML tagging layer in the middleware itself.
4. **Enhancing relational query transformations:** Although the SQL constructs that an XQuery is compiled to may have many extensions that might appear to be exotic to pure relational users, these are indeed natural extensions from the perspective of SQL/XML users. The Oracle SQL extension functions, such as *extract(), existsNode(), extractValue()* and *XMLSequence()* table function and their rewrite optimization [16] had provided a foundation to enable the optimization of the SQL/XML from XQuery. Native support for XQuery in Oracle XMLDB has been implemented by extending the SQL query transformation and rewrite modules, such as view merging, subquery unfolding, and operator tree algebraic optimizations to handle the complete optimization of the SQL constructs and operators underlying XQuery. This yields XQuery performance that is orders of magnitude faster in the database server as compared to execution in middleware.
5. **Static type checking for SQL/XML expression:** We further leverage the static type information to generate appropriate operators for optimal performance. Since the XQuery static type analysis is done at the time of SQL compilation, the types of XQuery expressions can be derived based on the SQL types of the underlying constructs.

Furthermore, this approach indeed opens an opportunity for middleware XQuery engines to push down XQuery into the backend RDBMS engine via the XMLQuery() function or XMLTable construct when needed. This is particularly beneficial for XQuery middleware performance with XML content stored in Oracle XMLDB in the form of XML type tables and views, or in the XML file repository.

4. XQuery Compilation

4.1 Architectural Overview

The processing of XMLQuery() and XMLTable() functions occurs during SQL query compilation time. After SQL parsing, we syntactically transform XMLTable into an XMLQuery() function within the built-in *XQSeq()* table function. Then after the SQL semantic analysis, type checking and view expansion process, we start the processing of each XMLQuery() function in the SQL statement. The XQuery native compilation driver parses the static XQuery string, does static analysis and type checking of the XQuery and compiles it into native SQL data structures with XML extension operators. In cases where the XQuery expression can not be compiled into SQL and SQL/XML constructs, we leave the XMLQuery() function intact. This is thus a hybrid approach. After this phase, the generated SQL structure goes through various query transformations, such as operator tree optimizations, view merging, subquery unnesting, etc and then goes to the optimizer which generates an optimal plan for execution.

Figure 1 shows the hybrid approach where we use the native XQuery optimization and execution as a primary strategy and use the co-processor approach for cases where we are unable to perform the native compilation. This allows us to deliver the full functionality of XQuery in the server while continuously enhancing the ORDBMS kernel to eventually process all XQuery constructs natively.

Figure 2 shows the XQuery compilation engine.

4.2 XQuery Parser and Semantic Analyzer

The parsing modules take in the XQuery text and convert it into an XQueryX [4] representation. This is different from a traditional parser which constructs an abstract syntax tree directly. The intermediate XQueryX form helps us to isolate the parser from the rest of the XQuery expression tree structure changes and allows us to effectively support XQueryX as an alternative language to XQuery. After the parsing, a standard XML parser is called to construct a DOM tree from the XQueryX representation, and the XQuery compiler subsequently works on the DOM tree to construct the XQuery expression tree.

The semantic analyzer performs the semantic analysis of the XQuery. It maintains various lists of namespace declarations, variable declarations, schema imports and function definitions with variable declaration associate with a lexical scope. The list is used to resolve variable and QName references, and to resolve XQuery Functions & Operators calls.

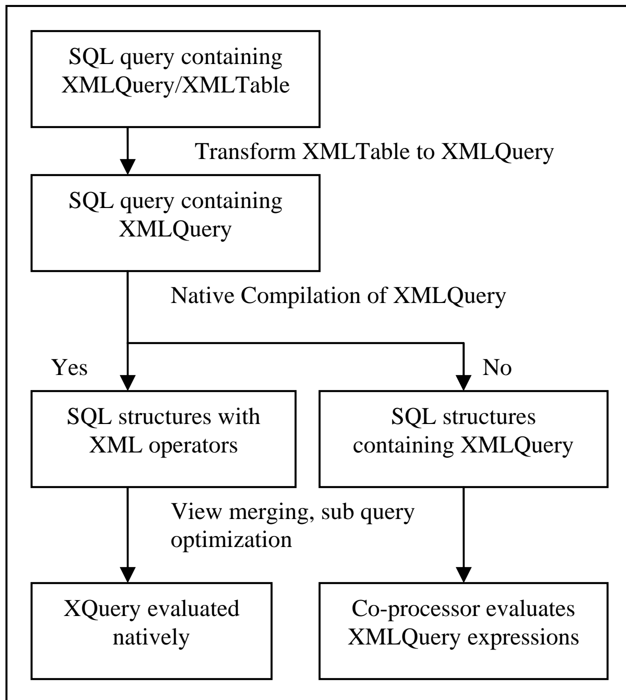

Figure 1 – XQuery hybrid evaluation strategy

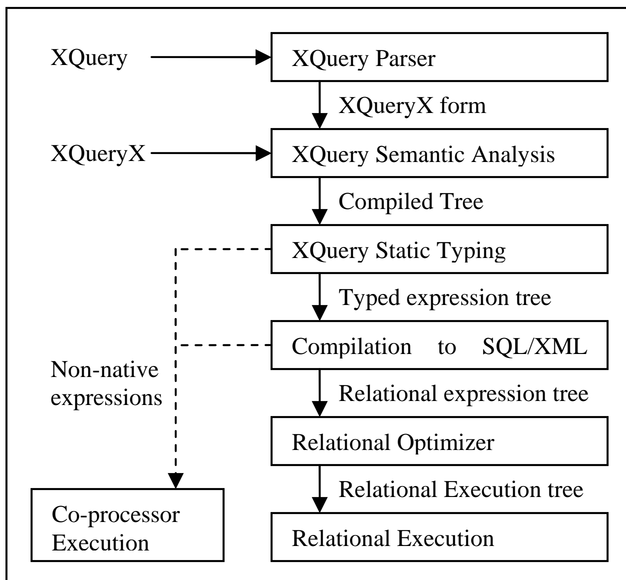

Figure 2 – XQuery Compilation Engine

4.3 XQuery Static Type Checking

4.3.1 Goal of XQuery static type checking

The XQuery Formal Semantics specification [3] has defined the rules of XQuery static type checking. XQuery static type checking is very useful for XQuery optimization. We take an optimistic type checking approach instead of a pessimistic one. We leverage the information gathered during the static type checking phase to guide the subsequent XQuery native compilation because we view static type checking as an important opportunity for XQuery optimization. The following is a list of sample optimizations we do based on static type checking and analysis:

1. We annotate each XQuery expression tree with static type information.
2. We expand wildcard XPath step and // XPath step based on the static type information. This is conceptually the same as that of *Compile-Time Path Expansion* idea in Lore [20].
3. We determine the cardinality of XML element or attribute access and convert general comparison expressions into value comparison expressions.
4. Since we do optimistic static type checking, we annotate the XQuery expression tree that fails on conservative static type checking so that the compilation can generate operators which do run time occurrence checks and type verification for such XQuery expressions.
5. We prune non-feasible branches of XQuery conditional expression, the where clause of FLWOR expression, type-switch clause of sequence type expressions, etc based on the static type information.
6. We prune unnecessary validate expressions if the input XML is proven to be valid based on the static type information.

4.3.2 Typing SQL expression with XQuery static type information

We use a tree representation to represent the static type of an XQuery expression and we extend this mechanism further to associate a type tree with each SQL expression returning an XML type value. We have developed a type manager that provides the type tree construction, manipulation and type computation operations on the type tree so that the rest of the system only needs to interact with the type manager. This is crucial for the XQuery type checking module to do better static type analysis for SQL functions which query XML, such as the XMLQuery() function.

The XQuery context item and variables referenced in the XMLQuery() function are passed in as an arbitrary SQL expression. Our system can build an XMLType tree for an arbitrary SQL expression tree and does XQuery static type checking based on the XML type tree.

4.4 Native support of XQuery Data Model

We have enhanced the current Oracle XML type image [17] to accommodate the XQuery data model so that we can natively support an XQuery data model based XML type value inside the ORDBMS kernel. Our XML type image is flexible enough to support atomic values, node references, etc as required by the XQuery data model. This is crucial as each XQuery expression returns an XQuery data model instance, which is modeled at the SQL/XML type level as an XML(Sequence) type. All the internal SQL operators created by the XQuery expression compilation process actually return an XML(Sequence) type.

5. XQuery Rewrite to SQL/XML

5.1 New SQL Operators and Rewrite Logic

Each XQuery expression is converted into a SQL operator or operator tree or a sub-query block. Due to space limitations, we

do not list all the internal SQL operators to support all XQuery constructs.

Rewrite of FLWOR expression – we construct a SQL select scalar subquery as the rewrite result. The for-clause of the FLWOR is converted into from-clause of the SQL with table function. The where-clause is rewritten into the SQL where-clause. The order-by clause is rewritten to the SQL order-by clause. The return clause is rewritten into the SQL select list. The entire select list is wrapped with the *XQAgg()* aggregate function so that the resulting SQL becomes a scalar subquery. For nested FLWOR expressions, the *XQAgg()* based scalar subquery is expanded with *XQSequence()* in table function used in the outer from clause which can then be view merged and algebraically collasped during the collection view merge process.

LET clause is handled by rewriting the XQuery expression for the variable definition into a SQL expression and binding the XQuery variable with the rewritten SQL expression. This is then used for the processing of **XQuery variable references** by substituting each variable reference with the SQL expression binding for that variable.

Rewrite of Constructors – we construct a SQL operator tree consisting of SQL/XML publishing functions, such as XMLElement(), XMLAttributes(), XMLPI(), XMLComment(), as the result of the rewrite. We internally enhance these publishing functions to handle tag names whose value is only available at run time as this is required for the rewrite of the computed constructors.

Rewrite of Path Expression – we construct *XQExtract*() SQL operator which evaluates the XPath on XML inputs and returns the result as XML(Sequence). Then we do further XPath rewrite on the *XQExtract*() operator into SQL/XML and object relational primitive operators leveraging the XPath rewrite framework that were built in [16].

Rewrite of literals – we rewrite each literal into a SQL literal and then wrap the result with an operator that converts the scalar value into an XML(Sequence) of atomic value.

Rewrite of Conditional Expression – we construct a SQL CASE operator.

Rewrite of Quantified Expression – we construct a SQL EXISTS/NOT EXISTS subquery.

Rewrite of Aggregate Expression – we construct the corresponding SQL aggregate functions, such as min(), max(), count() etc.

Rewrite of XQuery Sequence Construction – we construct a new *XQConcat*() SQL operator.

Rewrite of Arithmetic/Logical/Comparsion – we construct the corresponding SQL arithmetic, logical and comparison operators as the rewrite result. For general comparison, we rewrite them into EXISTS subquery as illustrated in [16]. Since the XQuery allows overloading of basic arithmetic and comparison functions with multiple built-in types, we need to construct polymorphic SQL arithmetic and comparison operators if the input type is determined to be a choice of different built-in types during the static type checking phase.

Rewrite of Range Expression – we construct the *XQRange*() SQL operator.

Rewrite of Cast and constructor function – we use low level SQL casting functions and operators.

Rewrite of Sequence Type Expression – we construct the *XQTypMatch*() operator with SQL CASE operator.

Validate Expression – we construct the internal XMLValidate() SQL operator.

XQuery functions/operators - we map them into existing SQL functions/operators. For certain XQuery functions/operators that do not have equivalent SQL functions/operators, new SQL operators are created in the RDBMS engine to implement the semantics of the corresponding XQuery operators.

For *fn:doc()* and *fn:collection()* function, we compile them into the underlying SQL query block that selects from the Oracle XMLDB repository tables. We also introduce Oracle extension function *ora:view()* which enables users to directly query XMLType tables and views or to convert a relational view into XML via SQL/XML publishing function automatically. The *ora:view()* functions are converted into an SQL query block that defines the underlying XMLtype table or view.

User Defined XQuery functions – we compile them into Oracle PL/SQL functions.

5.2 Algebra Optimization

The syntactic transformation of XMLTable construct and the subsequent compilation of the XMLQuery() function into a SQL native form often results in a complicated SQL construct as each XMLQuery() essentially becomes an expansion of a set of nested subquery blocks with a large set of operator trees. We then leverage the operator tree optimization, subquery un-nesting and view merging mechanisms [16] to simplify the resulting SQL structure. We have enhanced our current algebra rules to handle the new SQL functions and operators created during XQuery compilation. With algebra cancellation rules in mind, we often develop a SQL operator along with its inverse operator. New SQL operators are distributed to the branches of SQL CASE expressions and are usually pushed into the *XQAgg()* based scalar subquery blocks. This often leads to the subsequent application of cancellation rules for the SQL operator. The collapsing of *XQAgg()* with *XQSeq()* table function is carried out during the collection view merging step [16]. Due to space limitations, we do not list all the algebra rules here.

Our experience with this algebra optimization system has been positive. It is very easy to add new algebra rules for new SQL operators and enhance algebra rules for existing SQL operators. We have developed internal debugging tools for us to trace rewritten SQL query at various stages of algebra optimizations so that we know what new algebra rules to develop based on the final SQL statement. Since each algebra rule application always yields a valid SQL statement whose performance is not worse than the previous one, we always end up with a better performing query. Many queries end up with a final optimal form which is amendable to relational optimizations. Our experience with the algebra system is very close to the Query rewrite optimization in Starburst [21].

6. Acknowledgements

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

teams. We thank Vishu Krishnamurthy and Susan Kotsovolos for their managerial support of XQuery and SQL/XML, Sandeepan Banerjee, Geoff Lee, Stephen Buxton, Mark Drake for their XQuery product management support, Hui X. Zhang, Karuna Muthiah, Ying Lu, Qin Yu, Anand Manikutty and James W. Warner for their great XQuery and SQL/XML project development effort.

7. Conclusions

This paper illustrates native XQuery support in Oracle XMLDB by compiling XQuery into SQL constructs with XML extensions that can be optimized and executed efficiently by the underlying ORDBMS engine. This approach allows us to leverage the solid industrial strength engine to process XQuery natively and results in a tremendous performance improvement over the approach of embedding an off-the-shelf XQuery engine as a coprocessor.

As both XQuery and SQL/XML become the final recommendation and standard, there is much work remaining to develop new SQL operators, algebraic optimizations and execution methods so that all of the XQuery constructs can be natively compiled and the coprocessor approach can be completely eliminated. The merit of our native approach of integrating XQuery infrastructure on top of SQL/XML infrastructure enables Oracle XML DB to support both the SQL and XQuery syntaxes while utilizing the same underlying optimizer and execution engine to make it a truly industrial strength XML processing platform.

8. REFERENCES

[1] Scott Boag,Don Chamberlin,Mary F. Fernández,Daniela Florescu,Jonathan Robie, Jérôme Siméon. XQuery 1.0: An XML Query Language http://www.w3.org/TR/xquery/.

[2] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model http://www.w3c.org/TR/xpath-datamodel/

[3] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics http://www.w3c.org/TR/xquery-semantics/.

[4] Ashok Malhotra,Jim Melton,Jonathan Robie,Michael Rys.XML Syntax for XQuery 1.0 (XQueryX) http://www.w3.org/TR/2003/WD-xqueryx-20031219/

[5] World Wide Web Consortium, "XML Schema Standard" http://www.w3c.org/XML/Schema

[6] The international Committee for Information Technology Standard H2.3 Task Group, http://www.sqlx.org

[7] Andrew Eisenberg, Jim Melton. SQL/XML Is Making Good Progress. SIGMOD Record Vol 31, No 2, June 2002.

[8] SQL/XML 2003, the first edition of the SQL/XML standard published by the ISO as part 14 of the SQL standard: ISO/IEC 9075-14:2003.

[9] Andrew Eisenberg, Jim Melton . SQL/XML Advancements. http://www.sigmod.org/sigmod/record/issues/0409/11.JimMelton.pdf.

[10] Mary Fernández,Jérôme Siméon, Byron Choi, Amelie Marian, Gargi Sur. Implementing XQuery 1.0: The Galax Experience. VLDB 2003.

[11] Ioana Manolescu,Daniela Florescu,Donald Kossmann. Answering XML Queries over Hetergenous Data Sources. VLDB 2001.

[12] Daniela Florescu, Chris Hillery, Donald Kossman, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, Arvind Sundararajan. The BEA/XQRL Streaming XQuery Processor. VLDB 2003.

[13] Xin Zhang, Elke A. Rundesnteiner. Honey, I Shrunk the XQuery! - An XML Algebra Optimization Approach. WIDM'02 Nov, 2002, McLean, Virginia, USA.

[14] David DeHaan, David Toman, Mariano P. Consens, M. Tamer Ozsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. SIGMOD 2003.

[15] Ravi Murthy, Sandeepan Banerjee. XML Schemas in Oracle XML DB. VLDB 2003.

[16] Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikutty, Jim Warner, Vikas Arora, Susan Kotsovolos. Query rewrite for XML in Oracle XMLDB. VLDB 2004.

[17] Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikutty, Jim Warner, VikasArora. Towards an industrial strength SQL/XML infrastructure. ICDE 2005.

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

[19] Jayavel Shanmugasundaram, Eugene Shekita, Rimon Barr, Michael Carey, Bruce Lindsay, Hamid Pirahesh, Berthold Reinwald. Efficiently Publishing Relational Data as XML Documents. VLDB2000

[20] Jason McHugh, Jennifer Widom. Compile-Time Path Expansion in Lore. http://www-db.stanford.edu/lore/pubs/re.pdf

[21] Hamid Pirahesh, Joseph M. Hellerstein, Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. SIGMOD 1992.

[22] Torsten Grust,Sherif Sakr,Jens Teubner. XQuery on SQL Hosts. VLDB 2004

[23] Albrecht Schmidt, Florian Wass, Martin Kersten, Michael J. Carey, Ioana Manolescu, Ralph Busse. Xmark: A Benchmark for XML Data Management. VLDB 2002.