Native XQuery Processing in Oracle XMLDB

Zhen Hua Liu Muralidhar Krishnaprasad Vikas Arora

Oracle Corporation
400 Oracle Parkway
Redwood Shores, CA 94065
U.S.A

{zhen.liu,
muralidhar.krishnaprasad,
vikas.aora}@oracle.com

ABSTRACT

As XQuery is becoming the standard language for querying XML and relational SQL platform has been 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 natively supporting these XQuery operations using SQL/XML standard functions. In this paper, we present in detail, how we integrate the XQuery processing logic into the ORDBMS kernel and build upon the SQL/XML infrastructure and XPath querying capability available since Oracle 9i so as to utilize the industrial strength relational query optimizer.

1. Introduction

With the introduction of 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 XMLType columns. 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 XML type 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 challenges for the RDBMS engine. A straightforward approach referred to as the *coprocessor* approach, is to simply embed an 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 problems.

In this paper, we propose a **native XQuery compilation, optimization and execution** approach in the RDBMS by rewriting XQuery into SQL with XML extension operators and constructs, which are then amenable for 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 deliver 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 example of XQuery rewrite into SQL. In section 3, we describe related work and comparison with our approach. In section 4, we describe the XQuery native compilation process. In section 5, we describe the rewrite to SQL technique and the algebra optimization. In section 6, we discuss the performance experiment. In section 7 and 8, we draw the conclusion with acknowledgement.

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 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 problems due to the following reasons:

Storage Optimization: Since the XQuery processor is completely opaque to the rest of the RDBMS engine, it may not be able to take advantage of how the XML input data is stored physically. 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 separate from the SQL engine may prevent using standard relational

optimization technology such as constant folding, view merging, sub query optimization, distributed query processing, parallel query, partition pruning and common sub query elimination within the XQuery functions. These may have to 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 passing 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 of supporting XMLQuery() and XMLTable, is to rewrite the embedded XQuery, which is a static string, into SQL with XML extension operators so that the entire SQL statement that includes the XMLQuery() functions and XMLTable can be optimized as a whole. This approach fits naturally inside an RDBMS environment.

As SQL is a compiled language, during SQL compilation, it makes sense to do static analysis of the XQuery for each XMLQuery() and XMLTable invocation during SQL compilation time. They 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. These strategies works gracefully within the model of view expansion and merge process in the relational system as it enables the pushdown of the predicates and optimizes the result by eliminating unnecessary intermediate materialization of XML values.

Though native compilation is the optimal approach to XQuery processing inside SQL systems, some XQuery constructs and semantics require enhancing the RDBMS kernel which is an inherently complex task.

2.3 Hybrid Approach:

Therefore, we use a hybrid approach where we use the native XQuery optimization and execution as a primary strategy and use the co-processor approach for the 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 RDBMS kernel to eventually process all XQuery constructs natively.

2.4 Example:

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

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

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

XMLTable is defined as a syntactic transformation built on top of XMLQuery(). That is, first, the XMLQuery() is invoked by passing the input XQuery text to XMLTable. The result of the XMLQuery() is a sequence of items and each item becomes the input to construct a row of the result virtual relational table. An XPath embedded in each column specification is applied to the row item and the XPath result is atomized to an atomic value and converted to the correspodning SQL type.

```
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 queries for examples shown in table 2 and table 3 are listed in table 4 and table 5. (Note that the SQL table *dual* defined in Table 4 is a single row, single column pre-defined table available in Oracle) As it can be seen, the result of this XQuery rewrite process is to convert the original SQL statement into its semantically equivalent relational query using SQL/XML publishing functions to construct the result XML. The rewritten query can 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.

2.5 SQL Translation versus Native Compilation:

There is a fundamental difference between an XQuery *translation to SQL* as described in [11][13][14][18][22] versus our *native compilation*. A translation to SQL essentially creates a SQL string and then sends it to the server for compilation as a regular SQL statement. Native compilation on the other hand involves creating the internal SQL data structures, such as sub query blocks and operators directly from the XQuery expression. The advantage with native compilation is that SQL becomes simply a language syntax and both XQuery and SQL get compiled in to the same underlying data structures. The disadvantage in the case of the translation approach is that we first need to parse and compile the XQuery expression, generate a SQL string from it and then go through the SQL parsing and compilation process. Also, any new XML operations that are needed have to be introduced at the SQL syntax level.

3. Related Work Survey and Comparison

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]. For the cases that XQuery engines communicating with the SQL backend, the XQuery is typically translated into classical relational SQL statements which are then sent to the backend RDBMS for execution. Our approach is different that we build XQuery framework directly inside ORDBMS kernel and tightly integrates the XQuery engine with the SQL engine. We rewrite XQuery into SQL with SQL/XML extensions so that we can deliver SQL/XQuery duality and interoperability using SQL/XML infrastructure. There are tremendous performance advantage of using this approach and enables ORDBMS to support XQuery processing natively.

Our approach is based on the following principles:

1. We leverage the fact that SQL/XML has defined a new XML type as the first class datatype in SQL system and Oracle XMLDB has built XML type infrastructure which can be enhanced to natively support the XQuery data model [2] inside the relational engine. Based on the native XML type, we can build internal primitive SQL operators that can consume and generate XML document, XML document fragment, XQuery data model instance and use them for the implementation of XQuery operations that are foreign to the SQL system. Since we work inside ORDBMS kernel, we have the advantage of developing these primitive SQL operators to rewrite these XQuery operations whereas a typical middleware based solution treating SQL database as a black box is not able to do so [11] [18]. Paper [18] observed that the relational engine needs to add primitives to construct XML document fragments to perform these operations. This in principle agrees with our direction to support XQuery processing natively inside ORDBMS via building XMLType as native datatype inside database kernel.

2. We leverage the SQL/XML publishing functions as the basis to translate XQuery constructors to these publishing functions. Again most of the middleware solutions usually build the XML tagging layer in the middleware itself. Our approach allows us to leverage top-down stream evaluation of SQL/XML publishing functions [17] that were built for SQL/XML infrastructure to support XQuery constructor evaluation in streaming fashion.

3. We can control the quality of SQL that is built from the rewrite of XQuery. Although the rewritten SQL has many extensions that might appear to be exotic to the pure relational users, it is indeed a natural extension 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 laid out a foundation to enable the optimization of the SQL translated from the XQuery. Again since our approach is directly working in the ORDBMS kernel, we can enhance the SQL query transformation and rewrite modules, such as view merge, subquery unfolding, operator tree algebraic optimizations, to handle the complete optimization of these rewritten SQL with SQL/XML extensions and thus yield the XQuery performance orders of magnitude faster inside database server than otherwise executed in the middleware.

4. We further leverage the fact that since we compile the XQuery during SQL compilation, we can compute the static type of the XQuery expression, utilizing the types of the input SQL expressions and generate appropriate conversion operators for optimal performance.

Furthermore, this approach indeed opens an opportunity for middleware XQuery engine to push down the XQuery into the backend RDBMS engine via the XMLQuery() function or XMLTable construct. Since the Oracle XML DB supports XML stored as XML type tables and view and offers infrastructure to support a XML file repository, then XQuery middleware performance can be improved significantly via the technique of XQuery push down.

4. XQuery Compilation

4.1 Architectural Overview

The rewrite of XMLQuery() function and XMLTable construct occurs during the SQL query compilation time. After SQL parser, 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 rewrite of each XMLQuery() function in the SQL statement. The rewrite driver parses the static XQuery string, does a static analysis and type checking on the XQuery and rewrites it into native SQL data structures with XML extension operators and replaces the original XMLQuery() function by the result of this native compilation. If the XQuery expression can not be rewritten into SQL, then we leave the XMLQuery() function intact. After the rewrite phase, the SQL goes through various query transformations, such as operator tree optimization, view merge, subquery unnesting, and then goes to the optimizer which generates an optimal plan for execution.

Figure 1, shows how this hybrid strategy works. Note that if there are multiple XMLQuery expressions in a SQL statement, some of them may be rewritten while the rest may go through the coprocessor execution.

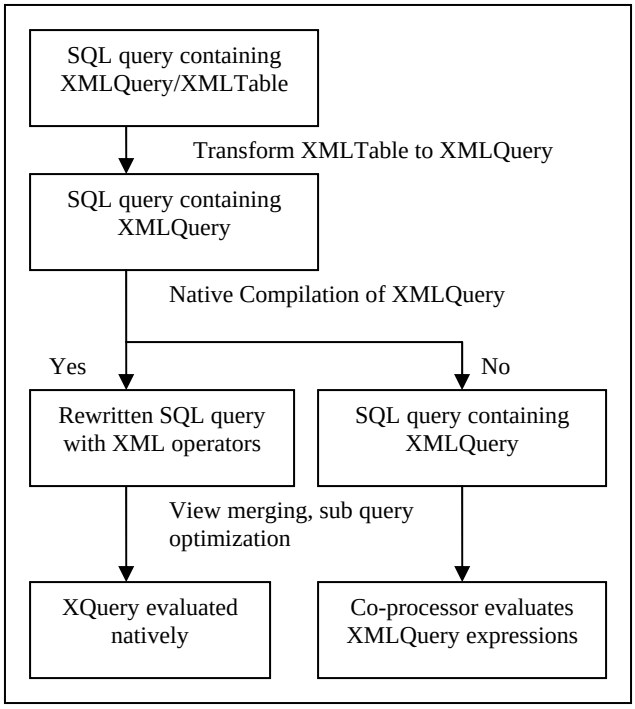

Figure 1 – XQuery hybrid evaluation strategy

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 parser, a standard XML parser is called to construct DOM tree from XQueryX and the XQuery compiler works on the DOM tree to construct the XQuery expression tree.

The XQuery expression tree is designed with a base expression class having multiple different kinds of XQuery expression classes inheriting from it. The kinds of the XQuery expression are, for example, FLWOR expression, path expression, sequence expression, constructor expression etc. The base expression class contains information common to all XQuery expressions, for example, the kind of the XQuery expression and the static type information for the XQuery expression.

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 XQuery F&O calls.

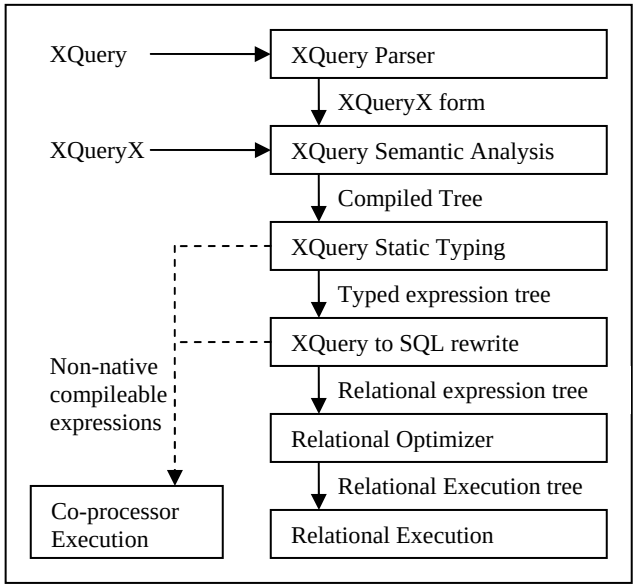

Figure 2 – XQuery Compilation Engine

After semantic analysis, the typecheck phase performs static type checking and annotates the expression tree with the static type information. Later the SQL rewrite phase uses this static type information to convert the expressions into the appropriate set of SQL operators and subquery blocks.

4.3 XQuery Static Type Checking

4.3.1 Goal of XQuery static type checking

XQuery formal semantics specification [3] has defined XQuery static type checking feature. XQuery static type checking is very useful when the input XML structure is known during compile time. The feature itself enables early error recovery. We do optimistic instead of pessimistic type checking and we leverage static type checking phase to gather information needed to guide the subsequent XQuery to SQL rewrite phase because we view static type checking as an important opportunity for XQuery optimization. We accomplish the following goals based on the static type checking:

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 figure out the cardinality of XML element or attribute access and convert general comparison expression into value comparison expression.

4. Since we do optimistic static type checking, we annotate the XQuery expression tree that fails on conservative static type checking so that the rewrite can generate operators which do run time occurrence check 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 expression based on the static type information.
6. We prune unnecessary validate expression if the input XML is proven to be valid based on the static type information.

4.3.2 Type Tree and Type Manager

We use tree representation to represent the static type of an XQuery expression and we extend this mechanism further to associate the type tree for each SQL expression returning an XML type value. We have developed a type manager which provides the type tree construction, manipulation and type computation on the type tree so that the rest of the system just needs to interact with the type manager.

4.3.3 Typing SQL expression with XQuery static type information

The next version of SQL/XML standard [9] will use an XML type modifier to describe more finer grained typing of XML value at the SQL level. For example, XML(Sequence) describes an arbitrary XQuery data model instance and XML(Content) describes an XML document fragment having a single document node with possibly many element nodes as its children. We go one step further and internally associate the corresponding XQuery static type information for each SQL expression returning XML type. This is crucial for XQuery type checking module to do a better static type analysis for SQL functions which query XML, such as the XMLQuery() function.

The XQuery context item and variables referenced in XMLQuery() function are passed in as an arbitrary SQL expression. The interesting SQL input expressions are those whose type is XML and there can be variety of them as Oracle XMLDB enables user to use SQL expression returning XML in many ways. 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 DBMS kernel. Our XML type image is flexible enough to support atomic values, node references etc 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 rewrite process actually return an XML(Sequence) type.

5. XQuery Rewrite to SQL

5.1 New SQL Operators and Rewrite Logic

The XQuery rewrite driver starts at the top of the XQuery expression tree. Recall that each kind of XQuery expression has a corresponding SQL expression that it can be compiled into. The rewrite driver recursively rewrites each child of the XQuery expression and then rewrites the XQuery expression itself. Each expression is converted into a new SQL operator or operator tree or a sub-query block. For example, the FLWOR XQuery expression is rewritten into a scalar SQL subquery and the constructor XQuery expression is rewritten into SQL/XML publishing function etc.

Table 6 shows a list of XML/XQuery internal SQL operators. Note the list is not exhaustive and it is meant to provide a flavor of the SQL operators that we've created for XQuery rewrite.

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 expression, 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 clasped during collection view merge process.

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

SQL OPERATORS	DESCRIPTION
XQCONCAT	*constructs an XML(Sequence) image by assembling all the input xquery items into one sequence*
XQAGG	*A SQL aggregate function. It constructs an XML(Sequence) image by aggregating the XQuery items generated from each row of the SQL query into an XQuery sequence type image.*
XQSEQ	*constructs an ordered collection (varray) of individual XML(Sequence). Primary used together with table() function as the inverse operation to xqagg().*
XQEXTRACT	*applys an xpath to an XML(Sequence) image and returns the result as XML(Sequence)*
XQRANGE	*constructs an XML(Sequence) image representing a sequence of integers between two bounded values without enumerating all the in-between integer values in the image.*
MKXQFROMSQL	*constructs an XML(Sequence) of atomic value image from a SQL scalar value*
UMKXQTOSQL	*Inverse of mkxqfromsql(). Extracts the SQL scalar value from an XML(Sequence) of atomic value*

	image
XQTYPCHK	*Do run time type check of an XML(Sequence) image*
XQSEQ2CONT	*converts an XML(Sequence) image to an XML(Content) image*
XQCON2SEQ	*converts an XML(Content) image to XML(Sequence) image by treating each child of the top document node as an item in the result sequence*
Polymorphic arithmetic and comparison operators	*Dispatch to the appropriate arithmetic and comparison operators depending on the run time type.*

Table 6 - SQL operators for XQuery Rewrite

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 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 it into a SQL literal and then wrap the result with *MkXQFromSQL*().

Rewrite of Conditional Expression – we construct a SQL CASE operator as the rewrite result.

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, wrapped with *MkXQFromSQL*() as the rewrite result.

Rewrite of XQuery Sequence Construction – we construct a new *XQConcat*() operator as the rewrite result.

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*() operator as the rewrite result.

Rewrite of Cast and constructor function – we use low level SQL casting functions and wrap the result with *MkXQFromSQL*() function.

Rewrite of Sequence Type Expression – we construct the *XQTypCheck*() operator with SQL CASE operator as the rewrite result.

Validate Expression – we construct the internal XMLValidate() operator as the rewrite result.

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 rewrite 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 table and view or to convert a relational view into XML via SQL/XML publishing function automatically. The rewrite of *ora:view()* is the underlying SQL query block defining the underlying XMLtype table or view.

User Defined XQuery function – we rewrite them into Oracle PL/SQL function.

To support the returning content option for XMLQuery(), we apply the *XQSeq2Cont*() operator on the XQuery rewrite result.

5.2 Algebra Optimization

The syntactic transformation of XMLTable construct and the subsequent rewrite of XMLQuery() function into SQL often result in a complicated SQL statement as each XMLQuery() essentially becomes an expansion of a set of nested subquery blocks with a large operator trees. We then leverage the operator tree optimization, subquery un-nesting and view merge mechanism [16] to simplify the result SQL statement. We enhance our current algebra rules to handle the new SQL functions and operators created for the rewrite of XQuery. With algebra cancellation rule in mind, we often develop a SQL operator along with its inverse operator. New SQL operators are distributed to the branches of SQL CASE expression and are usually pushed into the *XQAgg()* based scalar subquery block. 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 merge step [16]. Due to lack of space, we don't list all the algebra rules here.

Our experience with this algebra optimization system has been quite 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 and many queries end up with its final optimal form which is amendable for the relational optimizer to consume. Our experience with the algebra system is very close to the Query rewrite optimization in Starburst [21].

6. Performance Experiment

We use XMark [23] to verify the performance gain using the native XQuery compilation approach versus the coprocessor approach. The XMark XMLSchema is registered into Oracle XDB and the *auction.xml* data is stored object relationally with the top level table named as *SITE_TAB*. All XMark queries are

expressed using the XMLQuery() function and XMLTable construct with ora:view() XQuery function on *SITE_TAB* table.

For example, Q1 is expressed as the SQL statement shown in as the first row of table 7.

The performance experiment runs the XMARK Queries in two modes. In the first mode we disable the XQuery rewrite so that the entire XMLQuery() invocation is handled by the coprocessor. In the other mode we enable the XQuery rewrite so that the invocation of XMLQuery() and XMLTable construct is natively compiled into SQL/XML constructs and results in a highly optimized SQL statement. The equivalent optimized SQL statement for the Q1 in table 7 is shown in the second row of table 7. As one can see, the final SQL query can leverage relational index on the underlying relational tables whereas the coprocessor approach, which treats XQuery as a black box, cannot do so. Table 8 and Table 9 show XMark Q12 and Q13 queries and their equivalent rewritten relational SQL query.

The performance speed up achieved by the XQuery rewrite approach is tremendous and it is orders of magnitude faster than the coprocessor approach as the final optimized relational query can leverage all the mature optimization and execution relational technologies that the coprocessor approach can not leverage. Figure 3 shows the speed ration of the execution time for XQuery rewrite approach Vs the coprocessor approach.

```
SELECT XMLQuery(
    'for $b in ora:view("SITE_TAB")/site/people/person
     where $b/@id = "person0"
     return $b/name' returning content)
FROM dual;
```

```
SELECT   (  SELECT XMLAgg(XMLElement("name", p.name))
             FROM  SITE_TAB s,PERSON_TAB p
            WHERE p.id ='person0' AND
                  p.NESTED_TABLE_ID=s."SYS_NC0004700048$"
          )
FROM dual;
```

Table 7 - XMark Q1 and its native SQL via Oracle XQuery rewrite technique

```
SELECT x.* FROM XMLTABLE('
      for $p in ora:view("SITE_TAB")/site/people/person
    let $l :=
  for $i in ora:view("SITE_TAB")/site/open_auctions/open_auction/initial
       where $p/profile/@income > (5000 * $i/text())
       return $i
where  $p/profile/@income > 50000
return <items person="{$p/profile/@income}"> {count ($l)} </items>') x
```

```
SELECT XMLElement("items",
      XMLAttributes("PERSON_TAB"."income" AS "person"),
      (SELECT   COUNT(
                  XMLForest("OPEN_AUCTION_TAB"."initial") )
      FROM    SITE_TAB, OPEN_AUCTION_TAB
      WHERE PERSON_TAB."income" >
                      5000 * "OPEN_AUCTION_TAB"."initial"
        AND "OPEN_AUCTION_TAB"."NESTED_TABLE_ID"=
                              "SITE_TAB"."SYS_NC0005000051$")
```

```
      )
FROM SITE_TAB ,PERSON_TAB
WHERE PERSON_TAB."income" > 50000
   AND PERSON_TAB.NESTED_TABLE_ID
            = SITE_TAB."SYS_NC0004700048$"
```

Table 8 – XMark Q12 and its native SQL via Oracle XQuery rewrite technique

```
SELECT x.* FROM XMLTABLE('
for $i in ora:view("SITE_TAB")/site/regions/australia/item
return <item name="{$i/name/text()}"> {$i/description} </item>') x
```

```
SELECT XMLElement("item",
      XMLAttributes("AUSTRALIA_ITEM_TAB"."name" AS "name"),
      SYS_MAKEXML("AUSTRALIA_ITEM_TAB"."DESCRIPTION"))
FROM SITE_TAB,AUSTRALIA_ITEM_TAB
WHERE AUSTRALIA_ITEM_TAB.NESTED_TABLE_ID=
            SITE_TAB."SYS_NC0002300024$"
```

Table 9- XMark Q13 and its native SQL via Oracle XQuery rewrite technique

7. 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, Stephen Buxton 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.

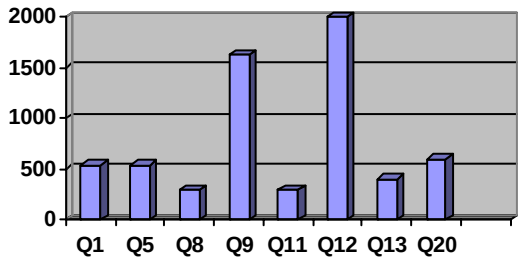

Figure 3: Query Speed Ratio-XQuery Rewrite Vs Coprocessor

8. Conclusions

This paper illustrates the native XQuery support in Oracle XMLDB by rewriting XQuery into SQL with XML extensions so that the rewritten SQL can be optimized and executed efficiently by the underlying ORDBMS engine. This approach allows us to leverage the solid object relational engine to process XQuery natively and results in 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.

9. 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.