Towards a Physical XML independent XQuery/SQL/XML Engine

Zhen Hua Liu, Sivasankaran Chandrasekar, Thomas Baby, Hui J. Chang

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065, USA

{zhen.liu, sivasankaran.chandrasekar, thomas.baby, hui.x.zhang}@oracle.com

ABSTRACT

There has been a lot of research and industrial effort on building XQuery engines with different kinds of XML storage and index models. However, most of these efforts focus on building either an efficient XQuery engine with one kind of XML storage, index, view model in mind or a general XQuery engine without any consideration of the underlying XML storage, index and view model. We need an underlying framework to build an XQuery engine that can work with and provide optimization for different XML storage, index and view models. Besides XQuery, RDBMSs also support SQL/XML, a standard language that integrates XML and relational processing. There are industrial efforts for building hybrid XQuery and SQL/XML engines that support both languages so that users can manage and query both relational and XML data on one platform. However, we need a theoretical framework to optimize both SQL/XML and XQuery languages in one RDBMS. In this paper, we show our industrial work of building a combined XQuery and SQL/XML engine that is able to work and provide optimization for different kinds of XML storage and index models in Oracle XMLDB. This work is based on **XML extended relational algebra** as the underlying tuple-based logical algebra and incorporates tree and automata based physical algebra into the logical tuple-based algebra so as to provide optimization for different physical XML formulations. This results in **logical and physical rewrite** techniques to optimize XQuery and SQL/XML over a variety of physical XML storage, index and view models, including schema aware object relational XML storage with relational indexes, binary XML storage with schema agnostic path-value-order key XMLIndex, SQL/XML view over relational data and relational view over XML. Furthermore, we show the approach of leveraging **cost based XML physical rewrite** strategy to evaluate different physical rewrite plans.

1. INTRODUCTION

With XML becoming a universal data model to represent structured, semi-structured and unstructured data and declarative XML processing languages, such as XQuery and SQL/XML [10], becoming standardized, there has been substantial research and industrial efforts on building XQuery engines on different platforms with different XML storage, index and view models. One approach is to build a native XML database using XML tree as the physical storage model [1,40, 41] with XPath/XQuery as the only query languages. The other approach is to build a hybrid XML/SQL database. Major RDBMS vendors have built hybrid XQuery and SQL/XML engines in their RDBMS products so that relational data and XML data can be managed and queried on one platform [7,8,9,30] with both XQuery and SQL/XML as query languages.

However, IBM and Microsoft support one XML storage model in their respective XQuery engines, which are optimized for their corresponding storage and index model. IBM uses XML tree as the physical storage model with a combined path and value index [8,48] whereas Microsoft uses binary XML as the physical storage model with distinct path, value indexes [9,27]. Monet DB [33] uses range encoding to store XML documents in relational tables. Oracle XML DB, however, concludes based on its customer XML use-cases, that XML is an abstract data type and its optimal physical storage and index models are use-case driven. We find that there is no "one-size-fits-all" solution for physical XML storage and indexing because XML is used to represent data with a wide variety of characteristics. Consequently, the first ***requirement*** for the Oracle XQuery engine is that it must be XML storage, index and view model independent and yet be able to choose the best physical optimization strategies when working with the underlying XML physical model.

In hybrid SQL and XML RDBMS use-cases, both XQuery and SQL/XML are used to query XML. Furthermore, there are SQL queries to query relational views over XML using the XMLTable construct defined in SQL/XML. Therefore, the second ***requirement*** for the Oracle XQuery engine is that it must provide better interoperability with the SQL engine so that cross-language optimizations between XQuery and SQL/XML are feasible. The resulting XQuery/SQL engine is independent of any particular XML query language.

Although SQL and XQuery are different query languages, they share commonalities. One of the key similarities is that they are both set-based, declarative languages so that iterator-based (stream-based), lazy evaluation strategies, which use as little data materialization as possible, can be applied to process both languages [2,26]. Furthermore, both XQuery and SQL have the concept of join, selection, projection, and sort algebra. It is natural that an XQuery engine integrated into RDBMS should

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

share the same physical iterator-based execution infrastructure as that of the SQL engine and share the same physical data representation. Therefore, the third ***requirement*** for the Oracle XQuery engine is that it leverage existing mature SQL infrastructures as much as possible. Despite their similarities, SQL and XQuery have differences too. SQL is statically typed whereas XQuery is dynamically typed. A Relational set is unordered whereas XQDM (Xquery Data Model) is ordered. Therefore, one challenge that needs to be addressed is bridging these semantic gaps between XQuery and SQL if they are integrated as one engine in an RDBMS.

In this paper, we show our work and experience of building such an XQuery/SQL-XML engine optimized for different XML storage, index, and view models. The main contributions of the paper are as follows:

- We create **XML Extended Relational Algebra (XERA)** as the logical, tuple based algebra into which both XQuery and SQL semantics are compiled. With the theoretical foundation of SQL extensibility and object relational SQL [13], we show that the XERA that we have created and implemented in Oracle XMLDB is complete and is derived from a direct application of the principles of SQL extensibility and object relational SQL in the domain of XML. XERA is essentially SQL query graph extended with XML constructs and operators. We also incorporate tree algebra pattern in the form of *XPath navigation tree pattern* and *XPath with branching predicate twig pattern*, and *automata-based streaming evaluation algebra* pattern into the tuple based algebra so that both tree and automata physical optimizations over different physical XML formulations are feasible.

- We show physical rewrite that enables the XQuery/SQL engine to work with different physical XML formulations: (a) XML schema aware structured object-relational storage, (b) path, value, XMLTable based XML indexes on schema agnostic binary XML storage, (c) SQL/XML view over relational data. (d) XMLTable relational view over XML. To our knowledge, this is the first industrial XQuery/SQL/XML engine that is designed to work and optimize with different XML storage, index and view models.

- We show opportunities for using a cost based approach to prune different physical XML rewrite algebra plans given the same logical plan.

Outline Of The Paper: Section 2 discusses the various XML storage, index and view models that Oracle XMLDB supports and the architecture overview of the unified XQuery and SQL/XML engine. Section 3 discusses an example to show the multi-phase XML rewrite transformation. Section 4 discusses XML extended relational algebra. Section 5 discusses optimisations for the XML extended relational algebra – the logical rewrite independent of physical XML storage, index, and view models. Section 6 discusses the physical optimizations for different XML storage, index and view models, with cost based pruning techniques. Section 7 discusses performance evaluation. Section 8 discusses the related work comparison and section 9 concludes the paper with acknowledgement.

2. Overview of XML Storage and Index in Oracle XMLDB

Oracle XMLDB addresses both data centric and document centric XML [7] use cases. On one end of the spectrum, data centric XML is very structured and bound with a rigid XML Schema. In such use-cases, modelling XML as a hierarchical view of relational data with XML schema aware decomposition of XML gives much better query performance and offers the best interoperability of relational and XML data [5,6,17,19]. On the other end of the spectrum, document centric XML may not have an XML schema or may have a very flexible XML schema. In such use cases, a schema-agnostic aggregated XML storage, such as tree or binary XML storage, with path-value index gives much better query performance.

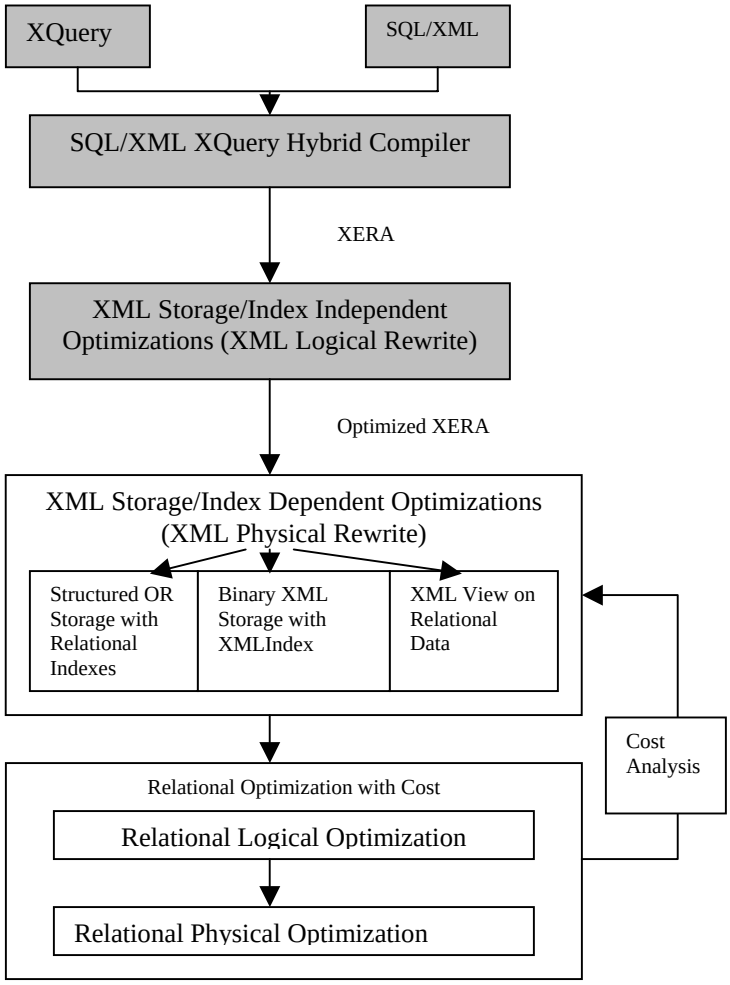

Figure 1 – Architecture of Unified XQuery/SQL/XML Engine in Oracle XMLDB for different XML physical formulations

Oracle XMLDB provides both XML schema aware object relational storage with relational index for structured XML data [5] and binary XML storage with XMLIndex [7] for semi-structured or unstructured XML data. There are three types of indexing strategies in Oracle XMLIndex: (a) **path-value-order key based** indexing strategies for ad-hoc XPath and value

search, (b) **structured, XMLTable-based** indexing strategies for querying structured components, and (c) **full-text extensions** to path-value-order key-based indexing strategies for doing full-text search within a document fragment using the *ora:contains()* Oracle XQuery extension function. The path-value-order key-based indexing is conceptually similar to path-based indexing technique described in [20] with ordered Dewey key [18]. The structured XMLTable based index is discussed in [12].

In addition, Oracle XMLDB provides a SQL/XML view over relational data using SQL/XML generation functions so that relational data can be queried as XML [6]. Also Oracle XMLDB provides a relational view over XML using XMLTable construct so that XML data can be queried relationally as if they were physically stored in a tabular form.

The XQuery and SQL/XML query rewrite architecture is multi-tiered as shown in Figure 1. Both XQuery and SQL/XML are first compiled into the same XERA and optimized without considering the actual physical XML storage/index. Then, the XERA logical algebra tree goes through physical rewrite with different XML storage, index and view models. Queries generated by physical rewrite are optimized by the relational query engine to produce physical-algebra query plans. For a given XML storage/index, if multiple physical, storage-dependent rewrite strategies exist, the relational optimizer is invoked for each physical rewritten query to determine the best physical rewrite strategy in terms of cost. An example of this multi-phase rewrite process is illustrated in Section 3. Note the SQL text shown here is for ease of presentation. The actual rewrite process does NOT generate SQL text but rather does rewrite transformation on query graph structures that represent these XML extended SQL queries.

3. Motivating Example

Consider an example of a table *xmlt* that stores XML documents and a SQL/XML query Q1 that finds the number of XML documents satisfying the XQuery in *XMLEXISTS()* SQL/XML operator.

```
select count(*)
from xmlt v
where xmlexists('$x/a/b[c="cv"][d="dv"]' passing value(v) as "x")
```

Q1- SQL/XML Query

3.1 XML Storage/Index Independent Logical Rewrite

User query Q1 is first compiled into the SQL extended with XML operators (XERA) as Q2. Then multiple *EXISTS* sub-queries are optimized into semi-joins as shown in Q3 (semi-join is not in the standard SQL, it is an existence join). However, *table(xqpath())* functions capturing the node tuple iteration in Q3 remain because the logical rewrite is not aware of the physical XML storage/index properties.

```
select count(*)
from xmlt v
where exists(
  select 1
  from table(xs(xqpath('$x/a/b') passing value(v) as "x")) v1
  where
    exists
      (select 1
       from table(xs(xqpath('$v1/c') passing value(v1))) v2
       where xqexval(value(v2)) = "cv")
   and
    exists
      (select 1
       from table(xs(xqpath('$v1/d') passing value(v1))) v3
       where xqexval(value(v3)) = "dv")
 )
```

Q2 - Post XQuery/SQL Hybrid Compilation

```
select count(*)
from xmlt v,
   semi-join lateral(table(xs(xqpath('$x/a/b') passing value(v) as "x"))) v1,
   semi-join lateral(table(xs(xqpath('$v1/c') passing value(v1) as "v1"))) v2,
   semi-join lateral(table(xs(xqpath('$v1/d') passing value(v1) as "v1"))) v3
where xqexval(value(v2)) = 'cv' and xqexval(value(v3)) = 'dv'
```

Q3 – Post Logical Rewrite

3.2 XML Storage/Index Dependent Physical Rewrite

3.2.1 Multiple physical rewrite strategies with binary XML

If *xmlt* is binary XML storage with XMLIndex., then Q3 can be rewritten using different physical XML rewrite strategies (S1,S2,S3). S1 uses the XMLIndex physical rewrite discussed in section 6.3. S2 uses the binary XML stream evaluation rewrite discussed in section 6.4. S3 uses a combination of XMLIndex and binary XML stream evaluation.

S1: Q4 is derived from physical rewrite using XMLIndex.. Note the *table(xs(xqpath()))* function is optimized into the selection from the XMLIndex storage table *pathtable*. Conceptually this query does a join based on node dewey order keys and docids among three different path-value probes of pathtable. Q4 is then optimized by the relational optimizer that determines the best physical join plan among pathtables and the best indexes to use to probe the pathtables using statistics collected on the pathtables and *xmlt* table.

```
select count(*)
from xmlt v, semi-join pathtable p1,
   semi-join pathtable p2, semi-join pathtable p3
where v.docid = p1.docid  and p1.pid = pid('/a/b') and
    p2.pid = pid('/a/b/c') and p2.docid = p1.docid and p2.value = "cv"
    and parent_key(p2.orderkey) = p1.orderkey
    and p3.pid = pid('/a/b/d') and p3.docid = p1.docid and p3.value = "dv"
    and parent_key(p3.orderkey) = p1.orderkey)
```

Q4 - Post Physical Rewrite with XMLIndex

S2: The *table(xs(xqpath()))* expression is rewritten using a binary XML specific xpathtable physical operator with streaming evaluation of XPath on the binary XML token stream. This results in Q5.

```
select count(*)
from xmlt v,
semi-join  lateral(xpathtable('$x/a/b', passing value(v) as "x") ) v1,
semi-join  lateral(xpathtable('$v1/c', passing v1.xvalue as "v1")) v2,
semi-join  lateral(xpathtable('$v1/d', passing v1.xvalue as "v1")) v3
where v2.value = 'cv' and v3.value = 'dv'
```

Q5 – Post Physical Rewrite with Binary Stream Evaluation

S3: This strategy uses an XMLIndex to evaluate '*/a/b*' and then uses binary stream evaluation to compute the two separate XPaths '*c*' and '*d*'. This results in Q6. The *mkini()* in Q6 is a

physical operator that constructs XQDM from the LOB locator stored in pathtable that locates the XML fragment.

```
select count(*)
from xmlt v, semi-join (select mkini(p1) as v1
                        from pathtable p1
                        where p1.pid =pid( '/a/b')
                        and p1.docid = v.docid) pv,
   semi-join lateral(xpathtab('$v1/c' passing pv.v1 as "v1"))  v2,
   semi-join lateral(xptahtab('$v1/d' passing pv.v1 as "v1")) v3
where  v2.value = 'cv' and v3.value = 'dv'
```

Q6 –Post Physical Rewrite with XMLIndex & Binary Stream Evaluation

Costs from the relational optimizer for Q4, Q5 and Q6 are then compared to choose the best physical rewrite strategy. Note that we can compare and cost among CPU operations for binary XML stream evaluation and pathtable I/O. The detail is beyond the scope of this paper though.

3.2.2 Physical rewrite for Object Relational Storage

If *xmlt* is OR storage with elements for '*/a/b*' stored in *tab_b*, elements '*c*' stored in *tab_c* and elements '*d*' stored in *tab_d*, then after physical rewrite and relational optimisation, Q3 is rewritten into Q7. The relational optimizer then chooses the optimal classical relational physical algebra: physical join plan and B+tree or bitmap index to access the storage tables.

```
select count(*)
from xmlt v,
      semi-join tab_b b, semi-join tab_c c, semi-jon tab_d d
where  v.docid = b.docid and  b.nid = c.nid and b.nid2 = d.nid
and c.val = 'cv' and d.val = 'dv'
```

Q7 - Post Physical Rewrite with OR storage

Note that although OR storage requires that all XML documents stored in the table have uniform schema, query performance for Q7 is conceptually better than that of Q4 for typical use cases because different xpath match of the pathtable that need to be done during run time for Q4 has been pre-compiled into different physical table access in Q7 during compile time. The joins in Q7 are primary-key foreign key joins instead of range-based Dewey order key joins. The value search in Q7 is targeted for specific columns of specific tables instead of one bloated value column capturing all values in the pathtable.

4. XML Extended Relational Algebra (XERA)

Although SQL and XQuery have similarities, they have a number of critical differences that necessitate the use of Object Relational SQL and Extensibility SQL, which are initiated in POSTGRES [47,13] and are widely supported by RDBMS products [14,15,16], as the theoretical foundation to create the XML extended relational algebra into which both XQuery and SQL is compiled. Extensibility SQL allows SQL to be extended with new datatypes; new functions and type comparison operators in SQL as if they were built-in SQL functions and built-in type comparison operators that can be used in SQL *ORDER BY* clause, *UNION, INTERSECT, EXCEPT* set operations; new aggregation functions as if they were built-in aggregation function (such as *sum(), avg(), min(), max()*); new indexing methods as if there were built-in index methods. Object relational SQL allows SQL to be extended with collection type and table function to nest and un-nest collection type.

We create new operators, new table function, new aggregation functions, to be discussed below, in Oracle XMLDB to formulate the XML extended relational algebra which bridges the semantics gap between XQuery and SQL.

4.1 Principles of XERA

1. Incorporate XQDM as a Datatype in SQL: XQuery Data Model (XQDM) is added as a new datatype into SQL. An XQDM can be a single XQItem, which is either an atomic value with its type, or a single XML node reference. It can also be an ordered set (sequence) of such XQItems (each XQItem is implicitly associated with an ordinal position indicating the item position in the sequence). Note that an XQuery atomic value is different from a SQL atomic value because an XQuery atomic value has its type associated with it. The new XQDM is actually a collection of items that is in principle similar to the collection type in the object extension of SQL [13].
2. Incorporate XQDM operators in SQL: We add a number of operators that manipulate the XQDM added into SQL. We refer to them as *XQSQL operators*. These operators take XQDM as inputs and return XQDM as output and are closed with respect to XQDM. They implement the semantics of XQuery expressions that SQL does not have.
3. UNNEST XQDM as relational set: We add a new table function *XS()* that converts an XQDM, which is a sequence of XQItems, into a set of relational tuples. The significance of this table function is that it models the concept of iteration of each XQItem of an XQDM as iteration of each row of a virtual table in SQL. There are two columns in each relational tuple. The first column *pos* is of type integer that is the ordinal position of the XQItem in the sequence, the second column *value* is an XQItem. The position column is used to answer *position()* and XPath positional index expressions in XQuery.
4. NEST relational set into XQDM: We add a new aggregator function *XQAgg()* that aggregates a set of relational tuples of XQItems into an XQDM. It is the opposite of the table function *XS()* and flattens a virtual table into a collection. Here is the **algebra identity rule for NEST/UNEST**: *SELECT XQAgg(tt.value ORDER BY tt.pos) FROM TABLE(XS(XQDM)) tt = XQDM*. The *XQAgg()* has an *ORDER BY* clause to aggregate all of the XQItems based on its ordinal position. The application of the table function *XS()* and the aggregate function *XQAgg()* enables the compilation of XQuery expressions into SQL with proper application of NEST and UNNEST operators. The combination of table function *XS()* and aggregate function *XQAgg()* allows us to unflatten and flatten XQDM at proper places and algebraically simplify XQuery expressions after expression inline, variable substitution and FOR clause merge.
5. Ordered and lateral Join construct: Joins in XQuery are ordered and have left join dependency whereas a pure relational SQL join is unordered and has no left join dependency. However, SQL has the concept of a lateral join which models left join dependency semantics. So join in XQuery is compiled into *ordered lateral join construct* that the underlying Oracle SQL engine supports. The access of iterator variable of the for clause of FLWOR is compiled into the access of value column of each row of the table function *XS()* over the for clause input. However,

to compile a let clause, a table with one row is used as the anchor point. See example 2 described below.

4.2 Examples of compiling XQuery into XERA

In Example 1, first, the table function *XS()* is used to iterate through each item from a sequence and *XQAgg()* is used to flatten tuple results back into a sequence. This makes the resulting SQL query a scalar query (query returns one row and one column output). Second, since '>' is a general comparison, we have it compiled as a SQL *EXISTS* sub-query that again uses table function *XS()* to iterate through each item of the sequence. Third, note that the *tti.pos* column is used for the position *at* clause in FLWOR. Also, note the usage of *XQElem()* operator to construct element node, *XQConcat()* operator to construct XQuery sequence, and *XQPath()* operator to apply an XPath expression on the input.

```
declare $x external;
for $i at $j in $x/a/b
where $i/c > 3
return  <rslt>{($j, $i/d/e, <newf>{$i/d/f}</newf>)}</rslt>
```

```
SELECT XQAgg(
   XQElem("rslt",
        XQConcat(XQAtomic(tti.pos), XQPath(tti.value, 'd/e'),
          XQElem("newf", XQPath(tti.value, 'd/f')))) ORDER BY
tti.pos)
FROM TABLE(XS(XQPath(XQBindVGet($x), 'a/b'))) tti
WHERE EXISTS(
              SELECT 1
               FROM TABLE(XS(XQPath(tti.value, 'c'))) tt2
               WHERE XQPolyVGT(tt2.value, XQConstruct(3,
xs:integer)) )
```

Example 1 - FWR compilation

```
declare $x external;
for $i  in $x/a/b
let $j :=  $i/d/e
where count($j) > 3
return  ($i, $j)
```

```
SELECT XQAgg(
            XQConcat(tti.value, ttj.value) ORDER BY tti.pos)
FROM
TABLE(XS(XQPath(XQBindVGet($x), 'a/b'))) tti, LATERAL(SELECT
XQPath(tti.value, 'd/e')
                                        FROM ONE_ROW_TAB)
ttj
WHERE (SELECT COUNT(*)
          FROM TABLE(XS(ttj.value))) > 3
```

Example 2 – *let* Compilation

In Example 2, the let clause is compiled into selection from a special table *ONE_ROW_TAB* that has only one row. Since there is a join dependency between a *for* clause and a *let* clause (because $j computation depends on $i), we use the LATERAL join construct of SQL. The XQuery *fn:count()* function is compiled into SQL count.

4.3 Compilation of XQuery into XERA

In [11], we presented how to compile XQuery into extended SQL with XML operators. Here we present the new improvements so that the XQuery rewrite logic is complete without the dependence on a co-processor evaluation.

Handle Dynamic Typing: In contrast to statically typed SQL, an XQuery expression can be dynamically typed if the implementation does not support static typing. In the presence of static typing of XQuery, we may be able to map certain XQuery expressions, for example, arithmetic expressions and value comparing expressions to their equivalent SQL operators where static typing is needed to map to proper SQL operators. However, in general we have to create new XML specific type polymorphic operators to handle XQDM whose type information may not be available during query compile time. Therefore, all the new XML specific SQL operators that we create in XERA take XQDM as input and return XQDM as output so that the actual sequence type information of the input XQDM can be obtained during run time in these type-polymorphic operators that do XQuery semantics based on the dynamic typing rules of XQuery.

These new polymorphic operators include *XQPolyAdd(), XQPolySub(), XQPolyMul(), XQPolyDiv(), XQPolyIDiv() and XQPolyMod()* XQSQL operators for XQuery arithmetic expressions.

These new polymorphic operators *XQPolyVGT(), XQPolyVGE, XQPolyEQ(), XQPolyLT(), XQPolyLE()* are XQSQL operators for XQDM value comparison. Note these operators have additional flag parameters to indicate what comparison context they are invoked in. This is critical as comparison semantics are different in different contexts. For example, although the semantics of general comparison is existential qualification and can be compiled into nested *EXISTS* sub-query. *$x > $y* is compiled into *EXISTS (SELECT 1 FROM TABLE(XS($x)) ttx WHERE EXISTS(SELECT 1 FROM TABLE(XS($y)) WHERE XQPolyVGT(ttx.value, tty.value, **GC_FLAG**))*. Note XQPolyVGT() is passed with a third parameter GC_FLAG to indicate that it is called with a general comparison context as the XQuery type casting rule semantics is different for general comparison and value comparison.

Aggregate based XQuery F&O functions, *fn:count(), fn:min(), fn:max(),fn:sum(),fn:avg(), fn:distinct-value()* are compiled into the corresponding SQL aggregate functions *count, min(), max(),sum(), avg()* and *DISTINCT* construct in SQL using these type polymorphic comparison and arithmetic functions.

Handle Node Comparison: Node comparison is compiled into *XQNodeCmp()* XQSQL operators. These operators take XQDM node reference as input and determine their relationship based on their node identities. When nodes are combined using XQuery union,intersect,except functions, they are compiled into SQL *UNION, INTERSECT, EXCEPT* query using *XQNodeCmp()* to do node comparison based on node identity. The physical node identity representation varies depending on the physical XML storage structures.

XPath Compilation: Instead of normalizing path expression into nested FLWOR expression at each step expression level as defined in XQuery formal semantics [43], we normalize the step expression into FWOR expression when the step has predicates. If each step of a path expression is a sequence of axis step expression without any predicates, it is directly compiled into ***XQPath()* XQSQL operator.**

For example, *$x/a/b/c* is compiled into *XQPath(XQBindVGet($x), 'a/b/c')*.

A path expression with predicate, for example, *$x/a[. gt "v1"][. lt "w2"]/b/c/d* is compiled into the following **Scalar Subquery with Predicates (SSP)** as shown in XQP1.

```
SELECT XQAgg(XQPath(tt.value, 'b/c/d'))
FROM TABLE(XS(XQPath(XQBindVGet($x), 'a'))) tt
WHERE  XQPolyVGT(XQExVal(tt.value),  "v1")  and
XQPolyVLT(XQExVal (tt.value), "w2")
```

XQP1

SSP can handle multiple predicates as long as each predicate does not access context position and it is statically determined

that the result of the predicate is not a numeric type. Otherwise, it is compiled into nested SSP.

For example, *$x/a[. gt "v1"][position() > 4]* is compiled into the following XQP2.

```
SELECT XQAgg(value(v))
FROM (SELECT /*+ NO_MERGE */ tt.value
    FROM TABLE(XS(XQPath(XQBindVGet($x), 'a'))) tt
    WHERE XQPolyVGT(XQExVal(tt.value), "v1"))  v
WHERE v.pos > 4
```

XQP2

Note that this is compiled into an inline view without merging as the inner view generates context position value that is filtered by the outer query.

As another example, if we don't know the static type of *$y*, then *$x/a[. gt "v1"][$y]* is compiled into the following XQP3.

```
SELECT XQAgg(value(v))
FROM (SELECT /*+ NO_MERGE */ tt.value
    FROM TABLE(XS(XQPath(XQBindVGet($x), 'a'))) tt
    WHERE XQPolyVGT(XQExVal(tt.value), "v1"))  v
WHERE XQPredTruth(v.pos, XQBindGet($y))
```

XQP3

The *XQPredTruth()* XQSQL operator checks the dynamic type of *$y*. If it is a numeric type, it does context comparison with its first input which is the context position and returns the result of the comparison. Otherwise, it returns the effective boolean value of *$y*.

4.4 Compilation of SQL/XML functions into XERA

SQL/XML generation functions: they are compiled into the same set of operators as the ones that XQuery constructor expressions are compiled into.

XMLAgg() aggregation function: it is compiled into XQAgg().

XMLQuery() and XMLExists() functions: The XQuery expression body in these functions are compiled as described above. For XMLExists(), the extra compilation of fn:exists() is added on top of the XQuery expression body.

XMLTable Construct: it is compiled into a **XQTab SQL query** using an *table(XS())* table function to iterate through each item from the row XQuery expression and projects out column values from the column XQuery expression. All XQuery expressions in row and column of XQTab are compiled into XML extended algebra as discussed above.

5. Optimization of XERA

5.1 Optimistic Static Type check

A **structured type tree** is used to do the static type inference [11,24]. A structured type tree is built from the bottom up while traversing the XQuery expression tree. The bottom portion of the structured type tree is constructed from the input variables to the query. There, the XML schema information or SQL/XML functions with SQL schema that constructs the input variables are used to build the structured type tree. If the input schema information is not available, a structured type tree representing *item()** sequence type is built.

We use static type inference to catch type errors that would otherwise be raised as dynamic type errors if not caught statically. However, we would not raise type errors statically if they may succeed during run time. For example, *3 + 'abc'* results in static type error. However, *declare $x external; 3 + (if ($x) then 4 else 'abc')* does not result in static type error because whether the dynamic type error is raised or not depends on the value of *$x* during run time. Such optimistic static type checking is feasible because the underlying XML extended relational algebra operators, such as *XQPolyAdd()*, are able to verify type from the input XQDM during run time. Note that this is different from pessimistic static typing defined in XQuery formal semantics [43] and what is implemented in Microsoft SQL Server [28].

5.2 Optimisation based on static type inference

Simplify arithmetic and comparison expression: arithmetic expressions and value comparison expressions can be simplified to their corresponding static SQL operators using static type inference.

Simplify Boolean based expressions: if *fn:boolean()* input type is one or more nodes, then *fn:boolean()* is true. If *fn:exists()* input type is one more item, then *fn:exists()* is true. If the input type to *fn:boolean()* or *fn:exists()* is an empty sequence, then the result is false. Based on these rules, we can simplify xquery expressions that are based on *fn:boolean()* value, such as condition expression, logical expression, where clause of FLWOR expression, quantifier expression etc, and *fn:exists()* function in a cascading fashion.

Sequence type based expressions: Expressions, such as instance of, treat as, typeswitch, castable, cast, are optimized based on static type inference.

5.3 Data Flow Analysis based Algebraic Optimization

Join Optimization: Left dependency join in XML extended relational algebra from XQuery is optimized into SQL join algebra if there is no dependency among join variables. Ordered join in XML extended relational algebra from XQuery is optimized into SQL join algebra if either unordered expression is used or the order of the items in the sequence is determined to be irrelevant. For example, when an expression is used as input in a boolean context, such as *fn:exists()* or *fn:boolean()* if the input is of static type *node()**, the order is not relevant so that unordered optimization can be used. Other optimizations of elimination of order in XPath [57,58] context can also be used. Once the join is optimized as unordered SQL join, hash and merge join from the physical algebra can be used to optimize join processing.

Existence Optimization: XQuery expressions, such as quantified expression, general comparison, *fn:exists()*, *fn:boolean()* whose input is of static type node()*, are compiled into *EXISTS()* and *NOT EXISTS()* subquries so that subquery un-nesting into semi-join and anti-join optimization in SQL [3] can be applied.

Nested FWR Block Merge: A FWR can be merged with its parent FWR clause. For example, xquery *for $p in (for $q in $x where $x > 3 return $q) return $p* can be optimized into *for $q in $x where $x > 3 return $q*. Since FWR is compiled into *SELECT FROM WHERE* query so that view merge (folding) technique in SQL [4] can be applied.

Ordinal positional number Elimination: positional number is not generated from *Table(XS())* table function if the resulting sequence is not used in positional predicate computation and the *at* clause of FLWOR construct is absent.

Cancellation & Merge Algebraic Reduction: Path navigation operator *XQPath()* can be cancelled with the node constructor operator *XQElem()* to eliminate unnecessary node construction. Two *XQPath()* operators can also be merged into one. For example, *XQPath(XQPath($x, 'a/b'), 'c/d') = XQPath($x, 'a/b/c/d')*.

Merging *XQPath()* operator enables the growth of tree pattern that can be facilitated for physical rewrite discussed in section 6.

XQPath Push Down: This is the distributive rule for *XQPath()*. For example, an *XQPath()* can be distributed to each child of a *XQConcat()* operator and each branch of a *CASE* operator. Such push down of *XQPath()* to its source enables XQPath cancellation and merge algebraic reductions.

Variable Inlining and Factoring: Let clause variable, prolog variable and non-recursive XQuery functions are inlined with their reference when the semantics is allowed. The result of inlining can create opportunities to apply composition & de-composition reduction. For example, xquery expression *let $x:= (<a>34<c>56</c><d>76</d>) return ($x/b, $x/d)* is optimized as *(34, <d>76</d>)*. However, if such elimination is not feasible, then it is better to compute let clause variable value once and avoid re-computation if the variable is referenced in multiple places [25].

Node reference Analysis: Static node reference analysis is performed to check if node identity generation is needed for each expression that may return nodes. For example, generating node identity for constructor is expensive. Furthermore, cancellation of XPath navigation with node construction requires that the node identity be irrelevant. For example: *let $x := (<a>34<c>56</c><d>76</d>) return ($x/b << $x/d)*. Since << requires the node identity generation for the constructor, so cancellation of xpath navigation with node constructor is not applied here and node identity generation is needed for the constructor.

6. Optimization with physical XML Storage, Index, View – Physical Rewrite

Here we show how Q3 in section 3 is optimized into Q4,Q5,Q7 for different physical rewrites. The physical rewrite occurs from inner query -blocks to outer ones. For each physical rewrite, we show below how query blocks *v1* and *v2* of Query 3 are optimized. In each optimization step, we highlight the operator or set of operators that are optimized.

```
table(xs(xqpath('$x/a/b' passing value(v) as "x"))) v1
```

Q3 –v1

```
table(xs(xqpath('$v1/c' passing value(v) as "v1"))) v2
where xqexval(value(v2)) = 'cv'
```

Q3 –v2

6.1 Physical Rewrite for OR Storage and SQL/XML view

When the XML schema is available, an XML instance can be decomposed and stored into a set of object relational tables for the XML schema. This corresponds to a schema aware XML storage model. However, Oracle XMLDB object relational storage is beyond simple 'shredding' methods because it can encode and store non-relational XML contents, such as PI, comment node information, node order, schema type information, into binary columns of the storage tables. Construction of XML from relational data and creation of XML views, using SQL/XML generation functions, are also supported.

ORPW-v1 shows three intermediate steps in the process of optimizing query Q3-v1. ORPW-v1.S1 shows how the input *$x* is constructed from *XQSQL* generation operator *XE()*. *XE()* operator constructs element node. The *XQAgg()* within *XE()* constructs collection elements from the underlying relational storage tables. ORPW-v1.S2 shows the result of the cancellation of *XQPath()* with input *XE()*. Finally, ORPW-v1.S3 shows the result of cancelling *table(XS())* with *XQAgg()*, after application of the NEST/UNEST algebra rule in section 4.1.

S1	*table(xs(**xqpath**(**XE**("a", select xqagg(**XE**("b", select xagg(XE("c", c.val)) from tab_c where c.nid = b.nid, select xqagg(XE("d", d.val)) from tab_d d where d.nid = b.id)) from tab_b b where tab_b.docid = v.docid), 'a/b'))) v1*
S2	***table(xs(** select **xqagg**(XE("b", select xagg(XE("c", c.val)) from tab_c c where c.nid = b.nid, select xqagg(XE("d", d.val)) from tab_d d where d.nid = b.id)) from tab_b b where tab_b.docid = v.docid)) v1*
S3	*select XE("b", select xagg(XE("c", c.val)) from tab_c c where c.nid = b.nid, select xqagg(XE("d", d.val)) from tab_d d where d.nid = b.id) from tab_b b where tab_b.docid = v.docid*

ORPW -v1

ORPW- v2 shows three intermediate steps in the process of optimizing query block Q3-v2. The input to *v2* is *v1*, the result of ORPW-v1. The optimizations of ORPW-v2.S1 and ORPW-v2.S2 are the same as those of ORPW-v1.S1 and ORPW-v1.S2. ORPW-v2.S3 shows how *xqexval()* is optimized into the underlying relational storage column.

S1	*table(xs(**xqpath**((select **XE**("b", select xagg(XE("c", c.val)) from tab_c c where c.nid = b.nid, select xqagg(XE("d", d.val)) from tab_d d where d.nid = b.id), from tab_b b where tab_b.docid = v.docid), 'c'))) v2 where xqexval(value(v2)) = 'cv'*
S2	***table(xs(** select **xagg**(XE("c", c.val)) from tab_c c where c.nid = b.nid)) v2 where **xqexval**(value(v2)) = 'cv'*
S3	*select XE("c", c.val) from tab_c c where c.nid = b.nid, and c.val = 'cv'*

ORPW -v2

The same OR physical rewrite of Q3-v2 is also applied to Q3-v3. All of these steps rewrite Q3 into ORPW-pre-Q7 below. Then, ORPW-pre-Q7 is optimized into Q7 via relational view merge. Since select lists of *v1*,*v2*,*v3* are not referenced in the top query block, they do not appear in the final query Q7.

```
select count(*)
from xmlt v,
semi-join lateral (select XE("b", select xagg(XE("c", c.val))
from tab_c c where c.nid = b.nid, select xqagg(XE("d", d.val))
from tab_d d where d.nid = b.id)
 from tab_b b where tab_b.nid = v.nid),
semi-join lateral (select XE("c", c.val)  from tab_c c
where c.nid = b.nid, and c.val = 'cv'),
semi-join lateral (select XE("d", d.val)  from tab_c d where
d.nid = b.nid,  and d.val = 'dv')
```

ORPW –pre-Q7

6.2 Physical Rewrite for XMLIndex

In the absence of an XML schema or in cases where XML schema flexibility is critical, an XMLIndex offers fast value-based and path-based searches. Physically, an XMLIndex consists of a path table that stores one row for each node in an

XML document. The path table stores for each node, an identifier for the document containing the node, a Dewey style order key [18] that captures the hierarchical and sibling relationships among nodes, an identifier for the concatenation of Qnames of nodes along the path from the root to the indexed node, and the atomized value of the node [7]. The mapping from the concatenation of Qnames of nodes to its identifier is stored in system-wide token tables.

XIPW-v1 shows two intermediate steps in the process of optimizing Q3-v1. *XQPath()* with input *$x* is rewritten into selection from pathtable with pid equals to *'/a/b'* path (internally we use binary pathid comparison). Then, *table(xs())* is optimized away with *XQAgg()*.

S1	***table(xs(*** *select* ***xqagg(*** *mkini(p1.loc)) from pathtable p1 where p1.pid = pid('/a/b') and p1.docid = v.docid)) v1*
S2	*select mkini(p1.loc) from pathtable p1 where p1.pid = pid('/a/b') and p1.docid = v.docid*

XIPW -v1

XIPW- v2 shows two intermediate steps in the process of optimizing query block Q3-v2. The input to *v2* is *v1*, which is the result of rewrite step XIPW-v1. *XQPath('$v1/c')* is rewritten into a selection from pathtable p2. Nodes selected from p2 should have path *'/a/b/c'*, which is obtained by concatenating the path for p1 (i.e., *'/a/b'*) with the path specified in Q3-v2 (i.e., *'c'*). They should also be direct children of nodes selected from pathtable p1, and this constraint is enforced using a check on the order keys of nodes from p1 and p2.

S1	***table(xs(*** *select* ***xqagg(****mkini(p2.loc)) from pathtable p2 where p2.pid = pid('/a/b/c') and p2.doicid = p1.docid and parent_key(p2.orderkey) = p1.orderkey)) v2 where xqexval(value(v2)) = 'cv'*
S2	*select mkini(p2.loc) from pathtable p2 where p2.pid = pid('/a/b/c') and p2.doicid = p1.docid and parent_key(p2.orderkey) = p1.orderkey and p2.value = 'cv'*

XIPW -v2

The same XI physical rewrite of Q3-v2 is also applied to Q3-v3. All of these steps rewrite Q3 into XIPW-pre-Q4 below. Then, XIPW-pre-Q4 is then optimized into Q4 via relational view merge.

select count(*)
from xmlt v,
semi-join lateral (select ***mkini(p1.loc)*** *from pathtable p1 where p1.pid = pid('/a/b') and p1.docid = v.docid),*
semi-join lateral (select ***mkini(p2.loc)*** *from pathtable p2 where p2.pid = pid('/a/b/c') and p2.doicid = p1.docid and parent_key(p2.orderkey) = p1.orderkey and p2.value = 'cv'),*
semi-join lateral (select ***mkini(p3.loc)*** *from pathtable p3 where p3.pid = pid('/a/b/d' and p3.doicid = p1.docid and parent_key(p3.orderkey) = p1.orderkey and p3.value = 'dv')*

XIPW –pre-Q4

6.3 Physical Rewrite for Binary XML Streaming Evaluation

Binary XML storage provides a compact post-parsed representation of an XML document. It can be viewed as a serialized form of a SAX stream over XML. The tags in XML are tokenised. In addition, if the XML is schema based, then content is stored in native format by making use of type information form the schema.

The main access pattern for identifying pieces from an encoded binary XML storage is to use a finite-state automaton based approach [54]. A single scan of the input binary-encoded document can identify nodes matching one or more XPaths. We refer to this approach as *binary XML streaming evaluation*.

Each node can be uniquely identified by means of a locator, which also serves as a Node Identifier. This identifier contains information about the location of the node in the binary encoded stream along with its QName, an optional type id and associated information. When the physical rewrite is applied to binary XML storage, query evaluation proceeds by first identifying matching nodes using streaming evaluation. These nodes themselves are represented using Node Identifiers, which make it possible to perform further stream evaluation on them.

SEBPW-v1 shows result of rewrite of Q3-v1 to build an xpathtable row source which performs automata evaluation of XPath *'/a/b'*. SEBPW-v2 shows the result of rewriting Q3-v2 to build an xpathtable row source that evaluates xpath '*c*.' This row source takes input from xpathtable column v1.xvalue and performs an automaton-based evaluation of xpath *'c.'* All of these steps rewrite Q3 into SEBPW-pre-Q5, which is then view merged into Q5.

select v1.xvalue
from xpathtable('$x/a/b' passing value(v) as "x") v1

SEBPW -v1

select v2.xvalue
from xpathtable('$v1/c' passing v1.xvalue))) v2
where v2.value = 'cv'

SEBPW -v2

select count(*)
from xmlt v, semi-join lateral(select ***v1.xvalue*** *from xpathtable('$x/a/b' passing value(v) as "x") v1), semi-join lateral (select* ***v2.xvalue*** *from xpathtable('$v1/c' passing v1.xvalue) v2 where v2.value = 'cv'), lateral (select* ***v2.xvalue*** *from xpathtable('$v1/d' passing v1.xvalue) v3 where v3.value = 'dv')*

SEBPW – pre-Q5

6.4 Cost-Based Evaluation of Physical Rewrite Strategies

For binary XML, there are several physical rewrite strategies:

- Evaluate master-detail twig tree pattern using structured XMLTable based XMLIndex or path-value-order key XMLIndex.
- Evaluate descendant XPath navigation by using the join of two sub-query probes of the path tables or by expanding a descendant XPath using token tables and then using expanded XPaths.
- Evaluate XQTab query construct using path index or using streaming evaluation or using path index for row expression of XQTab and streaming evaluation for the columns of XQTab as illustrated in section 3.

Since there are multiple physical rewrite strategies, our physical XML rewrite driver is cost based. We perform different physical rewrite strategies and call the relational optimizer to compute the cost of each query plan to determine which physical rewrite produces the cheapest plan.

7. Performance Evaluations & Observations

7.1 XMark

No-one-size-fits-all: We use the Xmark benchmark [59] for our performance experiments to evaluate the XQuery engine using both schema based object relational storage (OR) with relational

B+ tree indexes and binary XML storage with path-value-order key based XMLIndex (BINXI). All 20 XMark queries can be optimized fully by the Oracle XMLDB XQuery engine at the level of physical rewrite with OR and BINXI storages. However, the performance of queries comparing the two storage and index models is different depending on the type of queries.

Value-Predicate-Qry: For Q1 and Q5 that use XPath value predicate, OR out-performs BINXI as shown in Figure 2. This is expected because the value index in XMLIndex indexes the leaf values for all the nodes in one table whereas OR B+ tree index indexes leaf value for different nodes in different tables. Therefore, OR can precisely determine which leaf value column of the table it needs to search and thus significantly reduces the number of leaf values needed to be searched.

Positional-Predicate-Qry: For Q2 and Q3 that use XPath positional predicate, OR outperforms BINXI as shown in Figure 3. This is expected because OR can use the ordinal number of OCT (described in section 6.2) to compare the position quickly whereas BINXI needs to rank the nodes using order key to determine the position.

Count-Sequential-Qry: For Q6 and Q7 that count all nodes with a particular path, OR and BINXI achieve relatively the same performance as shown in Figure 4. This is expected because both queries compute the count without predicates, and this essentially involves table scans to count the rows.

Long-XPath-Qry: For Q15 and Q16 that use very lengthy XPath. BINXI outperforms OR as shown in Figure 5. This is expected because Q15 and Q16 have very long XPaths that can be answered using path index, which directly returns the locators to the corresponding XML fragments.

These performance observations from XMark queries are explainable from the underlying strength or weakness of each XML storage/index model. It demonstrates that the choice of the right XML storage and indexing strategies is use-case driven and depends upon the type of XML data and the type of queries. There is no 'one size fits all' solution to determine how to store and index XML.

Scaling: Q11 – Q12 are the time consuming XMark queries as they involve joins. Figure 6 shows that we get quadratic scaling with document size of 100MB and 200MB. For Q6 and Q7, we get linear to sub-linear scaling because Q6 and Q7 compute *count()* without predicates and therefore are equivalent to table scans. These results are consistent with the experimental results from Monet DB [33] that an XQuery system is bound to exhibit quadratic scaling with document size on XMark query Q11-Q12. Q6 and Q7 show sub-linear scaling for Monet DB.

7.2 TPOX

Need for cost based physical XML rewrite: XMark uses single document scaling with document size. However, in practice, we have seen that a more realistic data centric XML use-case is that of a large collection of moderately sized XML documents. TPOX [60] models such XML use-cases. TPOX queries can be optimized fully by the Oracle XMLDB XQuery engine using a structured XMLIndex [12] (XTXI), path-value-order based XMLIndex (PVXI) or binary XML stream evaluation (SEB). However, there are performance differences among them.

In TPOX queries, using XTXI to qualify XML documents among large collections of XML documents provides better performance than using PVXI as shown in Figure 7. This is expected because the master-detail-detail twig pattern used in selection can be answered by querying the pivoted XMLTable without requiring XPath searching during execution time. Furthermore, it is very common for a user to query a relational view over XML using XMLTable construct. TXQ shows such XMLTable query using TPOX schema. For such a query, SEB yields better performance than PVXI as the number of projected columns of XMLTable increases as shown in Figure 8. This is expected because PVXI needs to compute each XPath projected column using a scalar sub-query over the path table whereas SEB can evaluate each XPath for a projected column from the common row fragment in a streaming fashion.

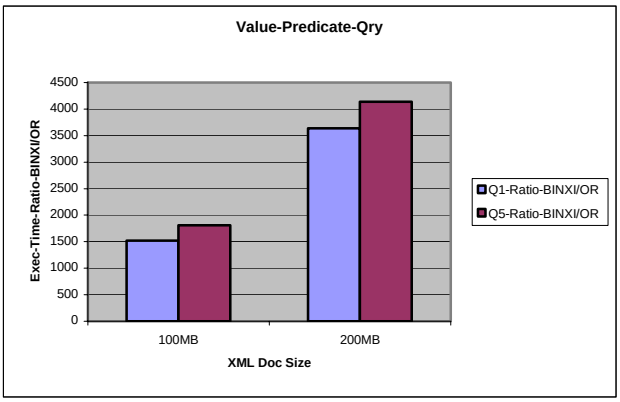

Figure 2 - OR outperforms BINXI for value-predicate-Qry

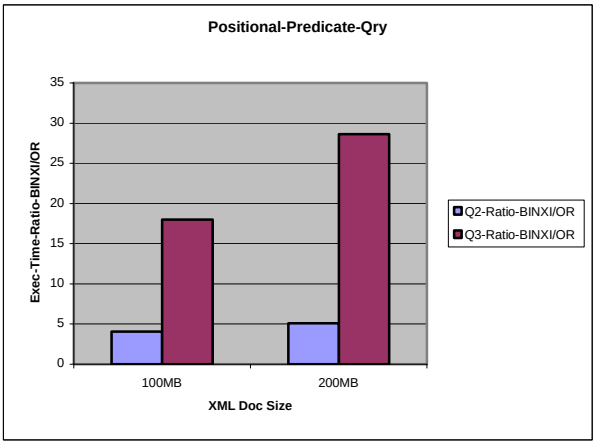

Figure 3 - OR outperforms BINXI for positional-predicate-Qry

Experiments from TPOX shows that different physical rewrite strategies yield different performance for the same query and therefore it is important to have a framework where we can cost different physical rewrite plans during compile time and to develop a costing model for different physical XML evaluation strategies. This is what we had discussed in Section 6.4.

8. Rationale & Related Work Comparison

The amount of work on XQuery in the database community during the last decade is enormous. There are basically three approaches to XQuery/XPath processing in the database community. The first approach is to use relational-like, tuple-based algebra as the logical algebra. This includes early work of translating XQuery to SQL [37][38][39]. However, *XQuery to SQL translation is not theoretically complete without the theoretical framework from object relational SQL and SQL*

extensibility. Then, various ways of incorporating XML specific operators into relational algebra have been proposed [29,31,32,33,34,35,36,43]. The second approach is to use tree-based algebra - the entire XPath and the XQuery FOR clause is folded into a pattern tree, which forms the basic unit [41,42,44,45,46]. The third approach is to use automata based algebra working with XML token streams [53,54,55,56].

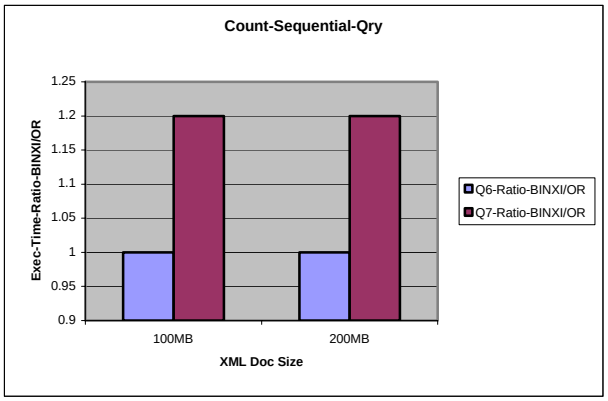

Figure 4 - OR,BINXI same for count-sequential-Qry

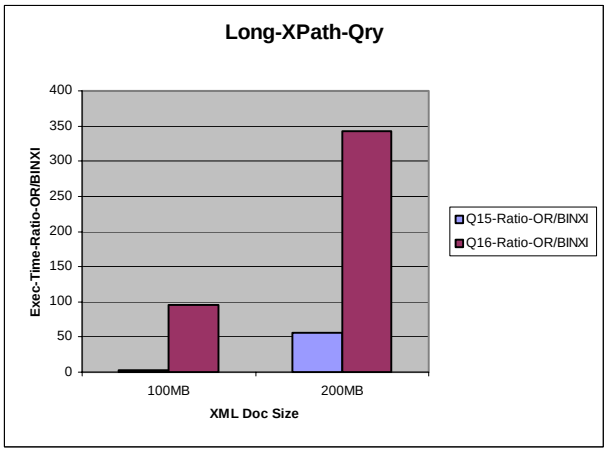

Figure 5 - BINXI outperforms OR for long-Xpath Qry

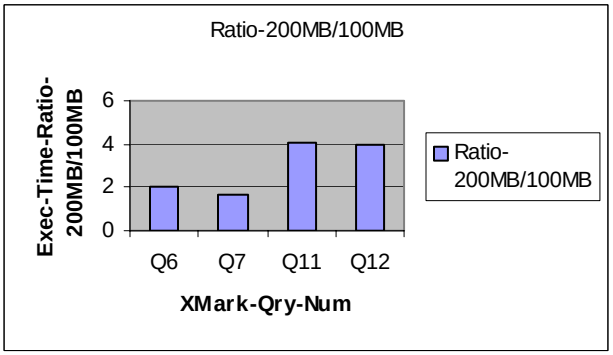

Figure 6 - XMark Scaling

```
SELECT v.*
FROM security_tab s,
  XMLTable(
XMLNamespaces (default 'http://tpox-benchmark.com/security' ),
'$doc/Security'
    passing s.sdoc as "doc"
  columns
    symbol varchar2(20) path 'Symbol',
    Name varchar2(20) path 'Name',
    SecurityType varchar2(20) path 'SecurityType',
    sector varchar2(20) path 'SecurityInformation//Sector',
    PE number path 'PE',
    Yield number path 'Yield') v) v
```

TXQ- TPOX XMLTable-Qry

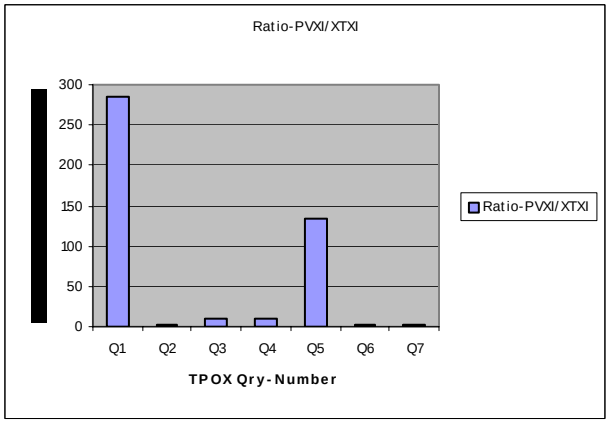

Figure 7 - XTXI outperforms PVXI for TPOXQ

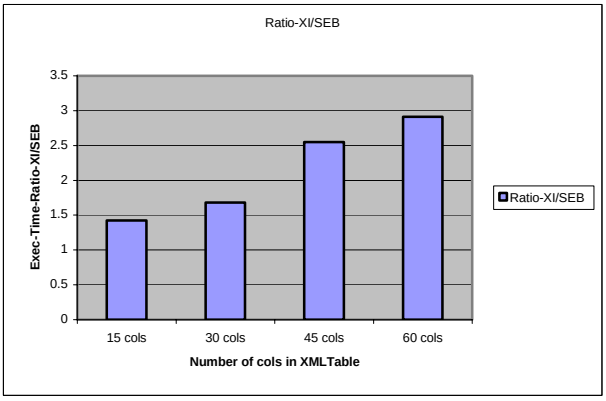

Figure 8 - SEB outperforms PVXI for XMLTable Qry

Our approach is to integrate the strengths of these algebraic approaches into one XQuery engine. We use XML extended relational algebra - a tuple-based algebra, as the main algebra. Our reasoning is that this algebra is theoretically complete as it can handle arbitrarily complex XQuery expressions. It is also practically adaptive to our relational-algebra-based RDBMS platform with its support for SQL extensibility and object-relational SQL framework [13]. We incorporate the common *XPath navigation tree pattern* and *XPath with branching predicate twig tree pattern* as high-level operators into XML extended relational algebra so that they can be used as a logical unit for physical rewrite. We also incorporate the automata based algebra as the physical algebra for evaluating XPath tree with binary XML storage.

As discussed previously, IBM DB2, Microsoft SQL Server and MonetDB work with one XML storage and index model. IBM DB2 Viper uses tree storage (with a mixture of tuple and tree based algebra) and schema-agnostic path-value index [8, 48]. Microsoft uses binary XML storage with path, value, order, property index [27] and its XQuery engine is hardwired to work with this XML storage and index model [28]. Monet DB [33]

shreds XML documents using range-based encoding and leverages relational engine to process SQL translated from XQuery on the encoding tables. Oracle's early work of XQuery/XPath XQuery is primarily designed to work with structured XML using object relational storage and XML view over relational data generated using SQL/XML [6,11]. XPERANTO [23] XQuery system works with XML view over relational data. This paper shows the approach of an XQuery engine based on the complete XML extended relational algebra. This engine works with different XML storage, index and view models and combines tuple, tree and automata algebra together. The idea of abstracting out tree based logical operators for different XML storage, index and view-models is in principle closer to the XAM (XML Access Module) idea proposed in [51]. However, we have demonstrated the set of XAMs we use and how this set can be efficiently supported on both schema aware structured XML storage and schema-agnostic XML storage. Furthermore, we show the idea of using cost based physical rewrite strategy to weigh XAMs, a strategy that distinguishes us from [51].

9. Conclusion & Future work

In this paper, we present our work on building a combined XQuery and SQL/XML engine that can work with and optimize for different XML storage, index and view models in RDBMS. To our knowledge, this is the first industrial XQuery engine that can work with a variety of physical XML storage and index models. We define an XML-extended-relational algebra as the logical algebra to optimize both XQuery and SQL/XML into the same underlying logical algebra presentation. This algebra is based on the theoretical framework of object-relational SQL and SQL extensibility. This achieves a physical XML independent XQuery-SQL/XML engine. Then, we optimize specific tree based algebraic operators - such as XPath navigation pattern, XPath with predicate branching pattern, specific master-detail twig pattern, and automata-based streaming evaluation - based on the underlying XML storage, index and view models. Our future work will include support for additional kinds of XAM patterns using XQuery/XPath materialized views.

10. ACKNOWLEDGEMENTS

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

11. REFERENCES

[1] LORE: http://infolab.stanford.edu/lore/

[2] G. Graefe. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 25(2):73–170, 1993.

[3] W.Kim: On Optimizing an SQL-like Nested Query. ACM TODS, Sep 7, 1982.

[4] M. Stonebraker: Implementation of Integrity Constraints and Views by Query Modification. SIGMOD Conference 1975: 65-78

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

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

[7] R. Murthy, Z. H. 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

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

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

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

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

[12] Z. H. Liu, M. Krishnaprasad, H. J. Chang, V. Arora: XMLTable Index - An Efficient Way of Indexing and Querying XML Property Data, ICDE 2007

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

[14] Z. H. Liu. "Object-Relational Features in Informix Internet Foundation."Informix technical notes. 9.4(Q4 1999):77-95.

[15] V. Krishnamurthy, S. Banerjee, A. Nori: Bringing Object-Relational Technology to Mainstream. SIGMOD Conference 1999: 513-514

[16] M. J. Carey, N. M. Mattos, A. Nori: Object-Relational Database Systems: Principles, Products, and Challenges (Tutorial). SIGMOD Conference 1997: 502

[17] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton: Relational Databases for Querying XML documents: Limitations and Opportunities, VLDB 1999

[18] I. Tatarinov, E. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita: Storing and Querying Ordered XML Using a Relational Database System: SIGMOD 2002

[19] F. Tian, D. DeWitt, J. Chen, C. Zhang: The Design and Performance Evaluation of Alternatives of Storage Strategies: SIGMOD Record, Vol 31, No 1, Mar 2002.

[20] M. Yoshikawa, T. Amagasa, T. Shimura, S. Uemura: Xrel: A Path-Based Approach to Storage and Retrieval of XML documents Using Relational Databases

[21] H. Jiang, H. Lu, Wei Wang, Jeffrey Xu Yu: Path Materialization Revisited: An Efficient Storage Model for XML Data. Australasian Database Conference 2002

[22] D. Florescu, D. Kossmann: Storing and Querying XML Data using an RDMBS. IEEE Data Eng. Bull. 22(3): 27-34 (1999)

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

[24] V. R. Borkar, M. J. Carey, D. Lychagin, T. Westmann, D. Engovatov, N. Onose: Query Processing in the AquaLogic Data Services Platform. VLDB 2006: 1037-1048

[25] Y. Diao, D. Florescu, D. Kossmann, M. J. Carey, M. J. Franklin: Implementing Memoization in a Streaming XQuery Processor. XSym 2004: 35-50

[26] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan: The BEA streaming XQuery processor. VLDB J. 13(3): 294-315 (2004)

[27] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, V. V.Zolotov: Indexing XML Data Stored in a Relational Database. VLDB 2004: 1134-1145

[28] S. Pal, I. Cseri, O. Seeliger, M. Rys, Gideon Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, E. Kogan: XQuery Implementation in a Relational Database System. VLDB 2005: 1175-1186

[29] C. Re, J. Siméon, M. F. Fernández: A Complete and Efficient Algebraic Compiler for XQuery. ICDE 2006: 14

[30] M. Rys, D. D. Chamberlin, D. Florescu: XML and relational database management systems: the inside story. SIGMOD Conference 2005: 945-947

[31] A. Deutsch, Y. Papakonstantinou, Y. Xu: The NEXT Logical Framework for XQuery. VLDB 2004: 168-179

[32] X. Zhang, B. Pielech, E. A. Rundensteiner: Honey, I shrunk the XQuery!: an XML algebra optimization approach. WIDM 2002: 15-22

[33] P. A. Boncz, T. Grust, M. Keulen, S. Manegold, J. Rittinger, J. Teubner: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. SIGMOD Conference 2006: 479-490

[34] N.May, S. Helmer, G. Moerkotte: Nested queries and quantifiers in an ordered context.: ICDE 239-250, Mar 2004

[35] R. A. Kader: XQuery Optimization in Relational Database Systems: http://arvo.ifi.uzh.ch/dbtg/vldbphd2007/Camera-Ready%20Papers/Paper%206/XQuery_Optimization.pdf

[36] M. Grinev, S. Kuznetsov: Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience. http://www.ispras.ru/~grinev/mypapers/rewriting-extended.pdf

[37] R. Krishnamurthy, R. Kaushik, J. Naughton: XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. http://homepages.inf.ed.ac.uk/wenfei/qsx/reading/xmltosqlsurvey.pdf

[38] T. Grust, S. Sakr, J. Teubner: XQuery on SQL Hosts. VLDB 2004: 252-263

[39] I. Manolescu, D. Florescu, D. Kossmann: Answering XML Queries over Heterogeneous Data Sources. BDA 2001

[40] C. Kanne, G. Moerkotte: Efficient Storage of XML Data. ICDE 2000: 198

[41] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M.Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A Native XML Database," VLDB Journal 11, No. 1, 274–291 (2002)

[42] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, Keith Thompson: TAX: A Tree Algebra for XML. DBPL 2001: 149-164

[43] XQuery 1.0 and Xpath 2.0 Formal Semantics: http://www.w3.org/TR/xquery-semantics/

[44] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, S. Paparizos: From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. VLDB 2003: 237-248

[45] P. Michiels, G. A. Mihaila, J. Siméon: Put a Tree Pattern in Your Algebra. ICDE 2007: 246-255

[46] N. Bruno, N. Koudas, D. Srivastava: Holistic Twig Joins: Optimal XML Pattern Matching. SIGMOD 2002

[47] M. Stonebraker: Inclusion of New Types in Relational Data Base Systems. ICDE 1986: 262-269

[48] A. Balmin, F. Ozcan, K.S. Beyer, R.J. Cochrane, H. Pirahesh: A Framework for Using Materialized Xpath Views in XML Query Processing. VLDB 2004.

[49] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, D. Srivastava: Index structures for matching XML twigs using relational query processors. Data Knowl. Eng. 60(2): 283-302 (2007)

[50] W. Xu, Z. Meral Özsoyoglu: Rewriting XPath Queries Using Materialized Views. VLDB 2005: 121-132

[51] A. Arion, V. Benzaken, I. Manolescu: XML Access Modules: Towards Physical Data Independence in XML Databases. XIME-P 2005

[52] A. Arion, V. Benzaken, I.Manolescu, Y. Papakonstantinou: Structured Materialized Views for XML Queries. VLDB 2007: 87-98

[53] M. Lee, B. Chin Chua, W. Hsu, K. Tan: Efficient evaluation of multiple queries on streaming XML data. CIKM 2002: 118-125

[54] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, P. M. Fischer: Path sharing and predicate evaluation for high-performance XML filtering. ACM Trans. Database Syst. 28(4): 467-516 (2003)

[55] H. Su, E. A. Rundensteiner, M. Mani: Automaton meets algebra: A hybrid paradigm for XML stream processing. Data Knowl. Eng. 59(3): 576-602 (2006)

[56] M. F. Fernández, P. Michiels, J. Siméon, M. Stark: XQuery Streaming à la Carte. ICDE 2007: 256-265

[57] J. Hidders, P. Michiels: Avoiding Unnecessary Ordering Operations in XPath. DBPL 2003: 54-70

[58] T. Grust, J. Rittinger, J. Teubner: eXrQuy: Order Indifference in XQuery. ICDE 2007: 226-235

[59] A. Schmidt, F. Waas, M.L.Kersten, M.J. Carey, Manolescu, R. Busse: "XMark: A Benchmark for XML Data Management" pp974-985 VLDB 2002

[60] M. Nicola, I. Kogan, B. Schiefer: An XML Transaction Processing Benchmark, SIGMOD 2007.

[61] A. Baras, C. A. Galindo-Legaria, T. Grabs, B. Krishnaswamy, S. Pal: Optimizing Similar Scalar Subqueries for XML Processing in Microsoft SQL Server. ICDE 2007 1164-1173