Grouping and Optimization of XPath Expressions in System RX.

Andrey Balmin #, Fatma Özcan #, Ashutosh Singh #, Edison Ting *

#IBM Almaden Research Center
*IBM Silicon Valley Lab
{abalmin,fozcan,ashutosh,eting}@us.ibm.com

***Abstract*— Several XML DBMS support XQuery and/or SQL/XML languages, which are based on navigational primitives in the form of XPath expressions. Typically, these systems either model each XPath step as a separate query plan operator, or employ holistic approaches that can evaluate multiple steps of a single XPath expression. There have also been proposals to execute as many XPath expressions as possible within a single FLWOR block simultaneously in a *data streaming* context.**

We observe in our System-RX prototype that blindly combining all possible XPath expressions for concurrent execution can result in significant performance degradation. We identify two main problems. First, the simple strategy of grouping all XPath expressions on a single document does not always work if the query involves more than one data source or has nested query blocks. Second, merging XPath expressions may result in unnecessary execution of branches that can be filtered by predicates in other branches or elsewhere in the query. To rectify these problems, we develop a combination of heuristic-based rewrite transformations, to decide which XPath expressions should be grouped for concurrent evaluation, and cost-based optimization to globally order the groups within the query execution plan, and locally order the branches within individual groups. Experimental evaluation confirms that selectively grouping multiple XPath expressions allows for better query evaluation performance and reduces the query optimization complexity.

I. Introduction

System RX [2] is a hybrid relational and XML database engine. It provides native XML storage, indexing, navigation and query processing through both SQL/XML [7] and XQuery [11], using the XML data type introduced by SQL/XML. The XML navigation component of *System RX* is based on TurboXPath [8] streaming engine, capable of executing multiple correlated XPath expressions in a single traversal of an XML fragment.

This high-granularity query processing is in contrast to most XML query processing engines today that model and execute each XPath step as a separate operator [9], [5], [6], [3]. With this approach, the query execution plans of even simple queries contain large numbers of operators, making it impossible to use a cost-based optimizer with exhaustive plan enumeration. Some systems [4], [2] employ an intermediate approach with operators that execute multiple steps of an XPath expression together. This approach has the benefit of smaller overhead and can be used when an external XPath engine is used to evaluate XPath over the XML data.

It has been shown in [8] that bundling together as many XPath expressions as possible for single-pass execution provides significant performance improvements in a *streaming environment*. However, we observe, that blindly applying the same idea to a database system produces mixed results, for two reasons. First, in a streaming system the whole document is always scanned during query processing. On the contrary, an XML database system, such as *System RX*, stores pre-parsed XML documents and is able to access only the required fragments. Second, [8] supports only a small subset of XQuery, namely multiple FOR/LET bindings and simple where-clause predicates, whereas *System RX* supports full XQuery and SQL/XML languages.

On the positive side, the high-granularity approach improves performance of two important query patterns that we frequently see in customer scenarios. First pattern includes XML restructuring queries that extract multiple fields from XML fragments to construct new XML documents. The second pattern is the XMLTable function of the SQL/XML [7] standard. XMLTable executes a set of XQuery expressions and returns its result in the form of a table. XMLTable provides the means to dynamically generate a relational view of XML data. Our experimental evaluation shows significant performance improvements for both patterns, including up to 4 times speed-up in execution of individual XMLTable calls.[1].

Another benefit of combining multiple XPath expressions into a single expression tree is the reduction in the number of XML navigation operators in the query execution plan. With fewer operators, the search space of plans is much smaller. This enables the cost-based optimizer to apply dynamic programming plan enumeration to more complex queries. We observed up to 50 times improvement in compile time of complex queries due to XPath merging.

Unfortunately, reducing the search space carries the risk of eliminating the optimal query execution plan from consideration. We identify two main causes for potential performance degradations. First, complex XQuery and SQL/XML queries often contain value-based joins and nested subqueries that can filter the results. In this case merging XPath expressions may lead to unnecessary work in navigating fragments that will be filtered out later in the query. Second, even simple XPath expressions with only local predicates may perform better with two or more scans over the stored document. We

[1]Due to limited space, we are not able to report the details of our experiments

978-1-4244-1837-4/08/$25.00 © 2008 IEEE

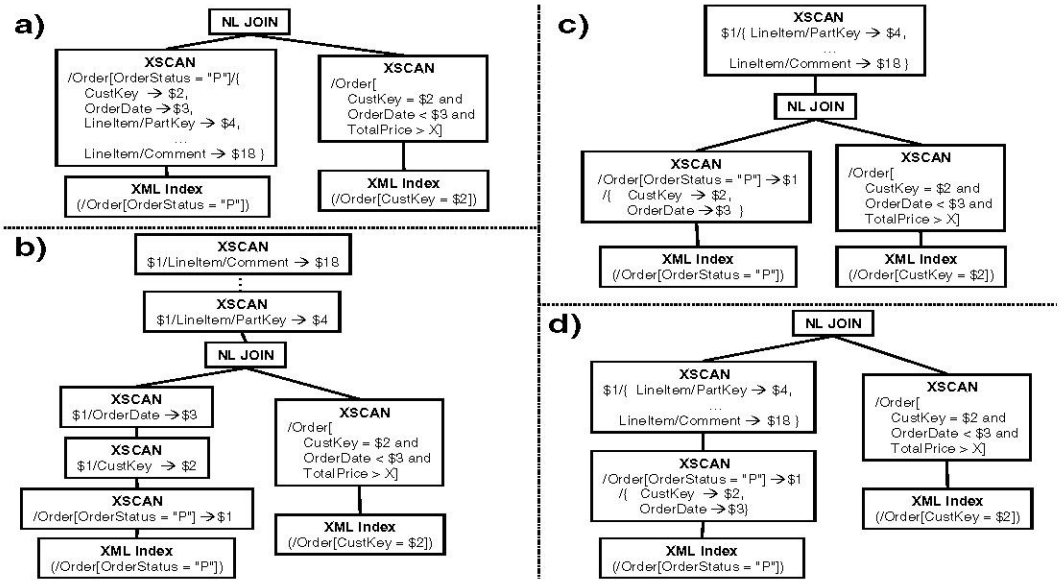

Fig. 1. Query execution plans for Query 1

address these two problems using a combination of *global* and *local* optimization strategies, which we have prototyped in *System RX*, an industrial-strength XML database system. *System RX* uses a single integrated query compiler for both SQL/XML and XQuery. After parsing, both SQL/XML and XQuery queries are mapped onto query graph model (QGM) [10] and optimized by the hybrid query compiler [2].

II. Global Optimization

Consider the following query, which looks for pending orders of customers who previously made expensive orders.

Query 1:

```
for $ord in db2-fn:xmlcolumn('TPCH.DOC')/Order[
                          OrderStatus = "P"]
let
 $c1 := $ord/LineItem/PartKey,
    ...
 $c15 := $ord/LineItem/L_Comment
where exists( db2-fn:xmlcolumn('TPCH.DOC')/Order[
                CustKey = $ord/CustKey and
                OrderDate < $ord/OrderDate and
                TotalPrice > X] )
return <res>...</res>;
```

This query extracts all 15 sub-elements of the lineitems of the pending order and constructs a result object out of them. If we merge all possible XPath expressions in the query, the only choice of the optimizer is the plan of Figure 1(a), which has just two XML navigation (XSCAN) operators. On the other hand, if we do not merge any expressions, then the optimizer could generate the plan in Figure 1(b), where the 15 LET clauses are executed separately after the join. This query will benefit from merging the 15 LET clauses with the first FOR clause (plan (a)), if the $[TotalPrice > X]$ predicate is not selective. However, if it is selective, the LET clauses do not need to be executed for the $Order$ elements eliminated by the join. Note that all 15 LET clauses can always be executed together. The problem is deciding whether they should be computed before or after the join.

We address this problem in two steps. First, we use heuristics to partition XPath expressions into clusters that can each be safely executed by a single XSCAN operator without loss of performance. Second, we use cost-based optimization to decide the execution order of XSCANs and other operators in the query.

For the first step, we provide an algorithm which takes as input the resulting QGM after the rewrites have been applied and tries to merge XPath expressions. The algorithm first computes a dependency graph among the XPath expressions in a query block. We say that an XPath expression is *dependent on* another expression, if the output columns of the latter are used as input to evaluate the former. Next, the algorithm partitions the set of XPath expressions within the same query block that are over the same document into clusters, without sacrificing an optimal execution plan. Finally, it merges the XPath expression within the same cluster, as long as the resulting dependency graph is acyclic. For example, for Query 1 our heuristics generate two expression trees. One for `/Order[OrderStatus="P"]/{CustKey,OrderDate}`, which produces the values needed for the join, and the other for all 15 LET clauses.

The second step is done by the cost based optimizer of *System RX*, which is described in detail in [1]. The optimizer uses XML and relational data distribution statistics to estimate cardinality and execution cost of alternative execution plans for pure XML and hybrid XML-relational queries, and picks the cheapest.

To support merged expressions, we developed a new XSCAN cardinality estimation algorithm that computes (a) cardinality of navigation trees with multiple next steps and extractions, and (b) the average number of XML items in the sequences that are being returned, for every output column of XSCAN. For Query 1, the optimizer produces the plans in Figure 1(c) and (d), and picks the cheapest.

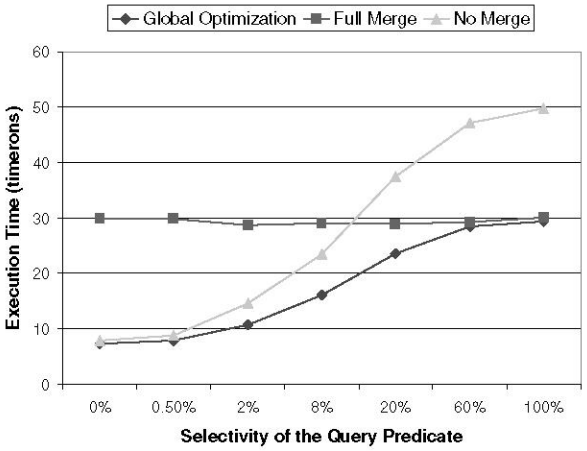

Fig. 2. The execution time of Query 1 under different optimization strategies.

Figure 2 shows *System RX* performance on Query 1 as a function of selectivity of the query predicate `[TotalPrice > X]`. We compare three approaches: no merging, resulting in the query plan of Figure 1 (b), full merging as in Figure 1 (a), and our global optimization approach, that allows the optimizer to pick between plans of Figure 1 (c) and (d). Figure 2 shows that our global optimization indeed takes full advantage of simultaneous execution of multiple XPath expressions, without sacrificing the optimal plan.

III. Local Optimization

To illustrate the second problem, consider the following example:

Query 2:

```
for $cust in db2-fn:xmlcolumn('CUST.DOC')/customer
where $cust/status = "I"
return
  <contact>{cust/name, $cust//phone}</contact>
```

This query could be merged into a single expression tree

```
('CUST.DOC')/customer(FOR)[status = "I"]{
   /name(LET),
   //phone(LET)}
```

Executing this tree in one-pass streamed navigation would entail a full scan of every document. For each `customer` element, all its descendants need to be accessed to collect `name` and `phone` sequences. Since the qualifying `status` element could be the last child of a `customer` element, there is no opportunity to short-circuit the computation for customers that do not satisfy the predicate.

An alternative two-pass strategy may be to scan the children of a `customer` element, and only if a qualifying `status` child is found, scan all descendants of `customer` to collect the results.

A two-pass execution strategy may outperform the single-pass execution by orders of magnitude. The choice between the strategies depends on data characteristics and should be made by a cost-based optimizer, utilizing data distribution statistics. Fortunately, this choice is *local* to an XSCAN operator, i.e. it can be made irrespective of what other expressions exist in the query.

We perform local cost-based optimization for every XSCAN operator in the query execution plan produced by the global query optimization stage. The local optimizer instructs the XML navigation algorithm to split the XSCAN expression into a pipeline of one or more fragments, with each fragment executed only after the previous one succeeds. Fragment execution is implemented by recursive calls to the navigation runtime, utilizing the multi-pass processing feature of *System RX* [2]. For example, Query 2 is executed by a single XSCAN that is split into two fragments: `('CUST.DOC')/customer(FOR)[status = "I"]`, and `{/name(LET), //phone(LET)}`.

Local optimization opportunities frequently arise due to merging of expression by the global optimization process. Our experimental evaluation suggest that while local optimization is useful even with the original medium granularity approach of *System RX*, it becomes critical once the global optimization is employed.

IV. Conclusion

We described the first complete framework for optimizing complex XQuery and SQL/XML queries by utilizing a high-granularity XPath processor, on pre-parsed and stored XML data. We proposed heuristic-based rewrites to decide which XPath expressions should be grouped for concurrent evaluation, and cost-based optimization methods to globally order the groups within the query execution plan, and locally order the branches within individual groups. We implemented these techniques in *System RX* [2], an industrial-strength XML database system. By applying heuristics and partitioning the problem into pieces that can be solved in isolation, we are able to successfully optimize complex queries with many large XPath expressions.

References

[1] A. Balmin et al. Cost-based optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–320, 2006.

[2] K Beyer et al. System RX: One Part Relational, One Part XML. In *In Proc. of SIGMOD*, pages 347–358, 2005.

[3] P. Boncz et al. MonetDB/XQuery: a Fast XQuery Processor Powered by a Relational Engine. In *Proc. of SIGMOD*, pages 479–490, 2006.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *In Proc. of SIGMOD*, pages 310–321, 2002.

[5] T. Fiebig et al. Anatomy of a Native XML Base Management System. *VLDB Journal*, 2002.

[6] H.V. Jagadish et al. TIMBER: A Native XML Database. *VLDB Journal*, 11(1):225–236, 2002.

[7] International Organization for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML), ANSI/ISO/IEC 9075-14:2006.

[8] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *The VLDB Journal*, 14(2):197–210, 2005.

[9] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *Proc. of XSym*, 2003.

[10] H. Pirahesh, J.M. Hellerstein, and W. Hasan. Extensible Rule Based Query rewrite Optimization in Starburst. In *Proc. of ACM SIGMOD*, pages 39–48, 1992.

[11] *XQuery 1.0: An XML Query Language*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/xquery`.