Grouping and Optimization of XPath Expressions in DB2® pureXML™

Andrey Balmin
IBM Almaden Research Center
San Jose, CA, USA
abalmin@us.ibm.com

Fatma Özcan
IBM Almaden Research Center
San Jose, CA, USA
fozcan@almaden.ibm.com

Ashutosh Singh
IBM Almaden Research Center
San Jose, CA, USA
ashutosh@almaden.ibm.com

Edison Ting
IBM Silicon Valley Lab
San Jose, CA, USA
eting@us.ibm.com

ABSTRACT

Several XML DBMSs 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 that blindly combining all possible XPath expressions for concurrent execution can result in significant performance degradation in a database system. We identify two main problems with this strategy. 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, IBM® DB2® pureXML™ adopts 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. These optimization techniques have been implemented as part of IBM DB2 9.5 (pureXML).

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query Processing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

General Terms

Algorithms, Performance

Keywords

XML Query Optimization, XPath and XQuery Processing

1. INTRODUCTION

DB2 pureXML [3, 4, 22] is a hybrid relational and XML database engine. It provides native XML storage, indexing, navigation and query processing through both SQL/XML [14] and XQuery [28], using the XML data type introduced by SQL/XML. The XML navigation component of *DB2 pureXML* is based on TurboXPath [16] 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. The low-granularity approach has been used in: (1) systems that store XML as relational tables [25, 17] where each XPath step is translated into a relational join, (2) some native XML stores [10, 31], and (3) systems that closely follow the operational semantics of XQuery [8, 11, 5]. In this approach, even for simple queries the number of operators in a query execution plan is large, making it impossible to use a cost-based optimizer with exhaustive plan enumeration. Some systems employ an intermediate approach with operators that execute multiple steps of an XPath expression together [24, 6, 12]. 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 [16] 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 *DB2 pureXML*, stores pre-parsed XML documents and is able to access only the required fragments. Second, [16] supports only a small subset of XQuery, namely multiple FOR/LET bindings and simple where-clause predicates, whereas *DB2 pureXML* supports full XQuery and SQL/XML languages.

On the positive side, high-granularity approach improves perfor-

mance of two important query patterns that we frequently see in customer engagements. 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 [14] standard. XMLTable executes a set of XQuery expressions and returns its result in the form of a table. They are used widely for two reasons: First, most applications and tools still operate on relational data, and XMLTable provides the means to convert XML into relational tables. Second, SQL and XMLTable are the only means for users to run XML analytics queries, as XQuery does not include convenient grouping facilities like SQL, or complex OLAP functions. Our experimental evaluation shows significant performance improvements for both patterns, including up to 4 times speed-up in execution of individual XMLTable calls.

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, which in turn reduces the search space of plans that the optimizer needs to consider. This, in turn, enables the cost-based optimizer to apply dynamic programming plan enumeration to more complex queries. Experiments presented later in the paper show 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 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 local predicates may perform better with two or more scans over the stored document. Solving these two problems is the focus of this paper. Next, we illustrate these problems in turn.

Consider the following query, which looks for every pending order if the same customer 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>;
```

Figure 1 shows four possible query execution plans for this query. In the figure, we use a linear representation of our pattern trees, which we will describe in Section 3.1. In this notation, we use curly braces separated by comma to denote multiple next steps, and we mark steps that compute an output variable, (i.e. an extraction point) with ”→”, and also show the corresponding output variable.

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 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. Then, the optimizer produces the plans in Figure 1(c) and (d), and picks the cheaper one given the selectivity and fanout of all the expressions. Under the condition where plan (a) would outperform (b), the optimizer picks plan (d), and otherwise, the optimizer chooses plan (c), which is even faster than (b).

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:

```
  db2-fn:xmlcolumn('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` all its descendants need to be navigated 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 `customer`, and if a qualifying `status` child is found, scan all descendants of `customer` to collect the results.

If certain conditions hold, such as a `customer` fragment is sufficiently large, and the predicate is selective, but has not been already applied by an index[1], 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 the XSCAN operator, i.e. it can be made irrespective of what other expressions exist in the query.

We perform the 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. For example, Query 2 is executed by a single XSCAN that is split into two fragments:

[1]*DB2 pureXML* employs XML indexes to eliminate documents that do not satisfy XPath predicates. These indexes contain fragments of documents specified by XPath expressions to control storage and update costs.

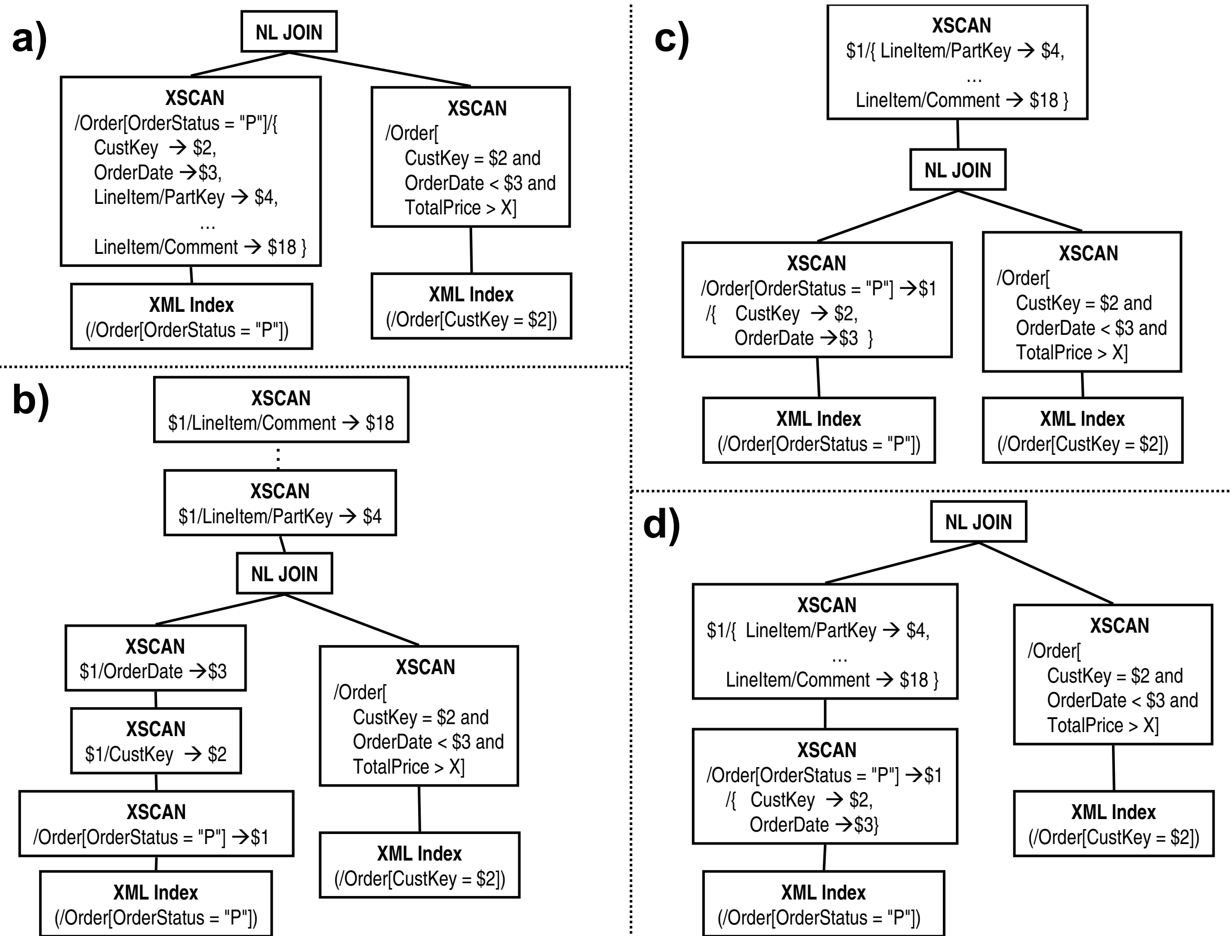


Figure 1: Query execution plans for Query 1

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

The main advantage of the global-plus-local optimization approach is the ability to optimize complex query expressions without sacrificing query planning options. For maximum flexibility, the optimizer should consider every feasible order of execution of XPath steps in the query. The Query 1 above contains 38 XPath Steps and two table accesses. Thus, optimizing this query is equivalent to optimizing a 40-way join - not a trivial or solved problem. 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.

The contributions of this paper can be summarized as follows:

- We describe the first complete framework for optimizing and processing of XQuery and SQL/XML on stored data, capable of executing multiple XPath expressions simultaneously. Our solutions have been implemented in *DB2 pureXML*, an industrial XML database system.
- We propose a combination of heuristics-based and cost-based optimization strategy to address the performance problems of the high-granularity approach. We present a heuristic-based algorithm for grouping and merging XPath expressions. We extend the cost-based optimizer of *DB2 pureXML* by providing a new cardinality estimation algorithm for complex tuple-producing expressions. We provide a local optimization algorithm that considers multi-pass execution of individual XSCAN operators. This optimization is applicable to XPath expressions that compute a single variable, but becomes even more important for merged expressions that compute multiple variable bindings.
- We provide experiments which show the effectiveness of the optimization strategy as well as the benefits of bundling multiple XPath expressions for XQuery and SQL/XML queries.

The rest of the paper is organized as follows. First, we discuss related work in Section 2. Then, we provide an overview of *DB2 pureXML*, in Section 3. In Sections 4 and 5, we present the details of the heuristic and cost-based optimization respectively. We present the results of our experimental study in Section 6. Finally, we conclude in Section 7.

2. RELATED WORK

Several XML query processing engines have been proposed both in academia and in industry [3, 34, 24, 11, 5, 8, 10, 21, 13]. A comprehensive list of public XQuery implementations and links can be found on the home page of W3C XQuery working group (http://www.w3.org/XML/Query), and a detailed discussion of each system is beyond the scope of this paper. To the best of our knowledge, none of these systems considers grouping XPath expressions

together and employs a high-granularity XPath processor which can return tuples of variable bindings.

A considerable amount of current research has focused on the efficient processing of XQuery. Recently, there has been some work on rewrite optimization of XPath expressions [19, 18, 8]. All of these approaches create a single operator for each step of an XPath expression, and hence focus on the optimized execution order of those steps. There has been also some work on translating XQuery and XPath queries into tree patterns [33, 13]. These pattern trees can be executed by either holistic XPath processors [6] or by a series of structural joins [31]. Although [33] translates all XPath expressions within a FLWOR block into a single pattern tree, it does not execute the pattern tree holistically, but translates it into structural joins, executing one step at a time. [1] has considered different granularity of algebraic operators, however their high granularity operations, called macro operators, were still implemented as trees of structural joins. None of these approaches addresses the performance problems we attack in this paper, and considers grouping a set of XPath expression for efficient streaming execution.

The SQL Server 2005 support for XML datatype is based on a pre-parsed XML representation stored in BLOBs [24]. To process queries over this representation, the BLOB based representation is internally translated into a node table format that has one row per node in the XML document. The row contains an encoding of the path, the node type and the value of the node. XQuery queries are translated to the algebra tree containing XML operators as well as traditional relational operators. During the Operator Mapping phase of the query compilation, these algebraic operators are mapped to runtime operators. An algebraic operator, XmlOp_Path, is provided for evaluation of multi-step XPath expression. Depending on the path expressions and the indexes present, this operator can be mapped to different runtime operator trees composed of selections over the node table combined with joins. Selections are used for the portions of the paths that can be evaluated by using indexes over the encoded paths in the table, as for example when the query path does not contain wildcards. Therefore it appears that in most of the cases, paths are translated to multi-operator trees.

The XML support in Oracle's DBMS uses a "hybrid" approach [34]. In this architecture, a subset of XQuery is translated into the existing relational algebra operators and system defined scalar and table functions. The new XQuery related system-defined functions perform tasks of evaluation of path expressions, aggregation, sequence operations, etc by processing bits of binary data representing XML fragments. The unsupported queries are translated into opaque operators that run an external XQuery processor. Therefore the granularity of this engine is on an XPath expression level.

Cost-based optimization for XML query processing have been employed in systems such as Lore [20], Niagara [12], TIMBER [31] and Natix [10]. The Lorel query language was OQL-based, and did not have XML sequences, which are central to the XQuery language and its optimization. Most systems that followed, adapted a one step at a time approach and optimized individual structural joins. These systems are related to our local optimization module, however they only consider single variable bindings. Also, since the optimization strategy is highly dependant on the indexing schemes and navigation operators available to the system, these approaches are not applicable to *DB2 pureXML*.

Only DB2 [2] published a description of a cost-based optimizer for an XQuery compiler and a relational-XML hybrid, which models entire XPath expressions with single variable bindings. We extend that work with cardinality estimation for tuple-returning expressions, producing multiple variable bindings.

The XML Navigation operator of *DB2 pureXML* is based on the TurboXPath [16] streaming XPath processor. There have been various proposals for evaluating XQuery/XPath over XML streams [11, 30, 7]. Most of these systems are very limited in their scope of features and only compute one variable binding at a time. BEA's streaming XQuery processor [11] is a complete XQuery implementation but it models each XPath step separately. [30] can execute multiple FOR bindings concurrently, but handles a very limited subset of XQuery. Furthermore, all these techniques are designed for a streaming environment and do not address the performance problems identified in this paper.

3. BACKGROUND

DB2 pureXML [3, 4, 22] stores XML data in columns of relational tables, as instances of the XQuery data model (XQDM) [29] in a structured type-annotated tree. By storing binary representation of type-annotated trees, *DB2 pureXML* avoids repeated parsing and validation of documents.This format preserves all the information available in the post-validation instance of XML documents. Each node is given a unique identifier [26] that gives nodes both logical and physical addressability that can be used by indexing and query evaluation. Each node also contains pointers to its attributes and ordered children nodes.

DB2 pureXML query evaluation run-time contains three major components for XML query processing: (1) XML navigation, (2) XML index run-time and (3) the XQuery function library. Additionally, several relational runtime operators have been extended to deal with XML data. The XML navigation operator evaluates path expressions over the native store, by traversing the parent-child relationships. It returns node references and atomic values to be further processed by the query runtime. XSCAN operator is described in more detail in [4] and [16]. One key feature of the XSCAN, described in [4], is *multipass processing*, which for a single query expression tree generates a set of correlated XML navigations, each evaluating a group of XPath steps using a one-pass algorithm. We make use of this feature to implement local optimization as discussed in Section 5.1.

DB2 pureXML supports value indexes defined by XPath expressions. These indexes are used to answer path expressions which contain value or general comparisons. The path expressions defining the XML index can contain wildcards and all XQuery axes, as well as kind tests.

DB2 pureXML provides both a SQL/XML and an XQuery interface. These two languages are composable: XQuery can be invoked from SQL and vice versa. *DB2 pureXML* uses a single integrated query compiler for both SQL/XML and XQuery. There is no translation from XQuery to SQL. After parsing, both SQL/XML and XQuery queries are mapped into query graph model (QGM) [23] and optimized by the hybrid query compiler [3, 4]. In its simplest form, a QGM graph consists of operations (boxes) and quantifiers (arcs) which represent the data flow between operations. More information on *DB2 pureXML* can be found in [3, 4, 22, 2].

To facilitate XML query processing, a new QGM operation, called the *ExpBox*, was introduced in [3, 4] to represent XML navigation. An ExpBox produces tuples of variable bindings, where each individual binding is an instance of the XQuery data model[29], and is either a singleton (FOR) or a sequence (LET). XML navigation is modeled using *XPath Step (XPS) trees*, which we discuss next.

3.1 XPath Representation

DEFINITION 3.1 (XPS NODE). *An* **XPS node** *represents a single XPath step and has 3 or more children* (a, t, p, N). a *is one of the six XQuery axes,* t *is the test, which is a name test, a kind test, or a wildcard test,* p *is the predicate, and* N *is a possibly empty set of next steps. Each next step is another XPS node.* p *can be arbitrarily complex, and it is the special NULL constant when there is no predicate on a step.*

XPS trees are annotated with the following flags to capture the FOR/LET semantics of FLWOR expressions.

1. **isExtraction:** This flag is set to *true*, if the XPS node is linked to an output column of the containing ExpBox and represents a variable binding, computing the result of an XPath expression.
2. **isFor:** This flag is set to *true* if the XPS node represents the last step of a FOR binding. If this flag is set to *false*, then LET is implied. An XPS node can be marked as a FOR even if it does not represent an extracted variable binding. We need to remember the last step of a FOR binding so that navigation run-time can apply the correct duplicate elimination and document order rules.
3. **EmptyOnEmpty (EOE):** This flag signals when an empty sequence needs to be created if there is no qualifying node. This flag may be set to *true* only if the XPS node is marked as extraction point. [2]

DEFINITION 3.2 (EXTRACTION POINT). *An* **extraction point** *is an XPS node whose* isExtraction *flag is set to true.*

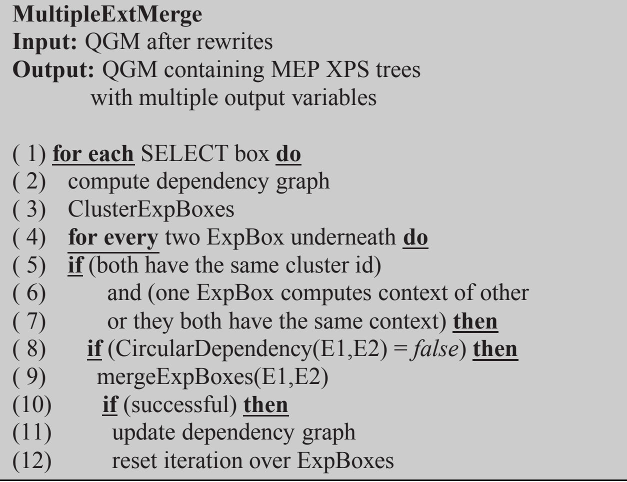
DEFINITION 3.3 (MEP XPS TREE). *A* **multiple-extraction (MEP) XPS tree** *is a tree* $T(r, E)$*, where* r *is the root of the tree and is an XPS node,* E *is a set of extraction points.*

Conceptually, XPS trees represent XPath expressions by capturing the data flow step-by-step through XPS nodes, and hence are able to model all XQuery axes, including parent, as well as any complex XQuery expressions as the predicate. Generalized tree patterns [33], on the other hand, is a structural representation of an XPath expression, and hence can only model child and descendant axes, and a limited set of predicates.

4. HEURISTIC-BASED MERGING OF XPATH EXPRESSIONS

In this section, we describe an algorithm, which takes as input the resulting QGM after the rewrites have been applied and tries to merge XPath expressions to generate XPS trees with multiple extraction points. The goal of this algorithm is twofold: First, it partitions a set of XPath expressions within a single query block that are over the same document into clusters so as not to preclude the optimizer from producing an optimal execution plan. Second, it tries to merge the XPath expression within the same cluster into an MEP XPS tree. The pseudo code for Algorithm **MultipleExtMerge** is given in Figure 2.

[2]Note that we need to model EmptyOnEmpty and FOR as two separate flags. Because *DB2 pureXML* has rewrite transformations which can convert LETs into FORs and turn off EmptyOnEmpty flag for LETs. In this paper, we are not discussing those rewrites.



MultipleExtMerge
Input: QGM after rewrites
Output: QGM containing MEP XPS trees with multiple output variables

```
( 1) for each SELECT box do
( 2)   compute dependency graph
( 3)   ClusterExpBoxes
( 4)   for every two ExpBox underneath do
( 5)   if (both have the same cluster id)
( 6)        and (one ExpBox computes context of other
( 7)        or they both have the same context) then
( 8)     if (CircularDependency(E1,E2) = false) then
( 9)       mergeExpBoxes(E1,E2)
(10)       if (successful) then
(11)         update dependency graph
(12)         reset iteration over ExpBoxes
```

Figure 2: XPath Merging Algorithm

It is important to note that what constitutes a query block is not defined by the user query, because the rewrite transformations may merge multiple query blocks, effectively unfolding nested FLWOR and SQL/XML expressions into a single query block. We define *a query block* as identified by QGM, as a SELECT box, after *all rewrite and XPath transformations* have been applied. The details of these rewrites are beyond the scope of this paper, but we note that they simplify the initial QGM considerably. As a result, we can handle complex nested FLWOR expressions with any type of return clause, although we only try to merge ExpBoxes underneath the same SELECT box.

Algorithm **MultipleExtMerge** employs a dependency graph to compute the clusters as well as to make sure the merge rewrite maintains a valid data flow in QGM. An XPS tree within an ExpBox cannot use an output variable produced by the same ExpBox.

We say that an XPS tree in an ExpBox is *dependent on* another ExpBox, if the output columns of the latter are used as input to evaluate former.

DEFINITION 4.1. *A dependency graph* $G(V, E)$ *of an SELECT box* s *is a directed acyclic graph, where*

1. $V = \{x \mid x \text{ is an ExpBox underneath } s\}$, *and*
2. $E = \{(v_1, v_2) \mid v_2 \text{ is dependent on } v_1\}$.

Algorithm **MultipleExtMerge** first computes this dependency graph for the SELECT box it is currently examining. Then, the algorithm invokes **ClusterExpBoxes** which uses the dependency graph to assign cluster numbers to individual ExpBoxes. Only ExpBoxes within the same cluster will be considered for merging.

ClusterExpBoxes starts assigning cluster numbers by identifying a set of source nodes S in the dependency graph. I.e. a set of ExpBoxes that do not depend on others. These are XPath expressions whose context is either a document root, or whose context comes from other operations outside the current SELECT box. All nodes in S get cluster number 0.

For the rest of the ExpBoxes the cluster number is assigned based on their dependencies and their participation in *external operations*. We say that an ExpBox participates in an external operation if one of its output columns is used in a subquery outside the current SELECT box, or it participates in a value-based join with either a relational column or another XML document. We also say that an

ExpBox participates in an external operation, if any of the ExpBoxes that depend on it, participates in an external operation.

Starting with nodes in S, **ClusterExpBoxes** traverses the dependency graph, and for every node v considers its dependants. If none of the dependant ExpBoxes participate in external operations, all of them get the same cluster number as v (denoted $cn(v)$). If some of the dependant ExpBoxes do participate in external operations, these ExpBoxes get cluster number $cn(v)$, while the rest of the dependants of v, get cluster number $cn(v) + 1$.

The goal of clustering is to identify groups of ExpBoxes, which operate on the same document, and do not have interaction with other operations in the query. These ExpBoxes can be merged safely because we would not be limiting the optimizer's options.

EXAMPLE 4.2. *Consider Query 1. For this query, the start set S contains two ExpBoxes, which contain the XPath expression $/Order[OrderStatus = "P"]$ and $/Order[CustKey = \$ord/CustKeyand]$. The output of ExpBoxes containing $\$ord/CustKey$ and $\$ord/OrderDate$ are used in a join operation, and hence are given the same cluster number 0 with the one that computes $ord, and all 15 LET clauses are assigned cluster number 1. In effect, we cluster ExpBoxes before the join into one group and all other ExpBoxes into another group. This way, the cost-based optimizer has the choice to decide whether to do the join before computing the 15 LET extractions, or do it afterwards.*

Once we compute the clusters for ExpBoxes, we then try to merge the ExpBoxes if they are in the same cluster. However, not every pair of ExpBoxes within a cluster can be merged. Some cannot be merged because the resulting pattern tree will violate the valid data flow in QGM, while others cannot be merged because the resulting expression cannot be expressed in a single pattern tree. Next, we describe these cases.

Before merging two ExpBoxes, **MultipleExtMerge** invokes another algorithm, **CircularDependency** (line 8), to check whether merging the two ExpBoxes would create a cycle in the dependency graph, by employing a standard cycle detection algorithm [27]. Every time we successfully merge two ExpBoxes, the dependency graph is updated to maintain a valid data flow in the query graph.

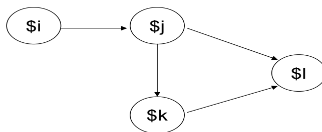


Figure 3: Example Dependency Graph

EXAMPLE 4.3. *Consider the following query:*

```
for $i in db2-fn:xmlcolumn("T.doc")//x
for $j in  $i/a/b
for $k in $j//d
for $l in $j/c[$k=d]/e
return ($k, $l)
```

The corresponding dependency graph is shown in Figure 3. Suppose that we are trying to merge the nodes corresponding to $j and $l, the resulting graph would have a cycle between $j-l and $k nodes. We can see that $k binding depends on $j, hence $j needs to be computed first. But, the newly merged $j-l XPS tree depends on $k, hence a circular dependency.

Algorithm **MultipleExtMerge** uses an XPath merge routine and tries to create the maximal XPS tree, which contains as many extractions as possible. It looks at two ExpBoxes at a time and merges them if they do not create a cycle in the dependency graph and they satisfy the necessary conditions, which we explain next. Suppose we are trying to merge two ExpBoxes, e_1 and e_2, containing XPS trees xps_1 and xps_2, respectively. We distinguish two cases:

1. **Case 1:** xps_1 computes the context of xps_2. [3]
2. **Case 2:** xps_1 and xps_2 have the same context node

The merge process proceeds as follows: In Case 1, we locate the XPS node n in xps_1, which is an extraction point and computes the context variable of xps_2. We, then, remove the first step of xps_2 ($x/a/b becomes /a/b). The first step in an XPS tree specifies the context of navigation. For example, in $x/a/b $x is the context step. Finally, we insert the remaining steps of xps_2 as next children into n. Note that due to earlier merges, the first step of xps_2 might have more than one next step. In Case 2, we remove the first step of xps_2, and insert the remaining children as next steps into the context step of xps_1. Note that the context step of xps_1 is also the context for xps_2.

In general, we need to be careful when there is a predicate on the context step of xps_2 in Case 1, and on the context step of either xps_1 or xps_2 in Case 2. We cannot simply discard the first step if there is a predicate on it. The solution is to rewrite xps_2 in Case 1 (xps_1 and xps_2 in Case 2), by injecting an extra self step, and moving the predicate to this new self step. This simple rewrite will transform an XPath expression of the form

$$\$i[pred]/optional_next_steps$$

into

$$\$i/self :: node()[pred]/optional_next_steps$$

However, this transformation is only possible, 1-) if the context variable is not a LET binding, and 2-) the predicate is not a positional predicate (a predicate which depends on context position or context sequence, such as fn:last or fn:position).

EXAMPLE 4.4. *Consider the following query fragment:*

```
let $j:= $i//a/b
for $k in $j[2]/c
```

We cannot merge these two XPath expressions, because the resulting expression cannot be expressed in a single XPS tree. It is incorrect to merge these two path expressions into the expression $i//a/b[2](→$j)/c, because this one returns the c children of the second b under every a element, whereas the original query asks for the c children of the second b under all a elements.

5. COST-BASED OPTIMIZATION

DB2 pureXML includes a cost-based optimizer, which is described in detail in [2]. 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.

The optimizer includes a number of novel features. In addition to normal relational operators, such as table scan, index access, joins, etc, our query plans also include an XSCAN operator which models the XML Navigation algorithm. An XSCAN operator is

[3] We also consider the symmetric case.

constructed for every ExpBox in the final QGM representation of a query. The cardinality estimation algorithm computes two values for every XSCAN operator: the cardinality, i.e. the number of tuples the operator is expected to return and *sequence size*, which is the average number of XML items in the sequences that are being returned. Recall, that values of XML type are XQDM instances that are always sequences of zero or more XML items. The cardinality of an XSCAN operator is the product of the input cardinality, selectivity of all the applied predicates, and the *fanout* of the navigation expression, i.e., the number of output rows produced by the navigation per average input.

The fanout computation algorithm (described in [2]) employs linear path data distribution statistics, that contain information on how many times each path occurs in the XML collection, and on distribution of data values (if any) that can be found by following the path.

To support MEP expressions, we developed a new cardinality estimation algorithm that computes (a) fanout of navigation trees with multiple next steps and extractions, and (b) sequence size for every column produced by the MEP navigation expression.

The algorithm makes a distinction between XPath Step (XPS) nodes inside predicate steps, which we call *predicate* XPS, and XPS nodes that are *not* inside any predicate of any ancestor XPS, which we call *navigation* XPS. For example, in $/a[b/c = 5]/d$, XPath Steps $/a$ and $/d$ are navigation XPS and $/b$ and $/c$ are predicate XPS nodes.

The new algorithm runs in two steps. First, it traverses the predicate XPS nodes and uses the data distribution statistics to compute cumulative selectivity $P(X)$ for a predicate subtree of every navigation XPS X. This step runs exactly as described in [2]. If X does not have a predicate, we define $P(X) = 1$.

MultipleExtFanout
Input: Tuple-extracting XPS Tree.
Output: Fanout F and sequence size SS for every navigation XPS node.

(1) **<u>for each</u>** navigation XPS node in the top-down traversal of the input XPS tree **<u>do</u>**
(2) **<u>if</u>** ($X\ isFor$)
(3) $N(X) = St(X)/St(Y)$
(4) $F(X) = N(X) * P(X)$
(5) $SS(X) = 1$
(6) **<u>else</u>**
(7) $N(X) = SS(Y) * St(X)/St(Y)$
(8) **<u>if</u>** ($N(X) > 1$)
(9) $SS(X) = N(X) * P(X)$
(10) $F(X) =$ K-th moment of $B(N(X), P(X))$
(11) **<u>else</u>**
(12) $F(X) = N(X) * P(X)$
(13) $SS(X) = 1$
(14) **<u>if</u>** ($XisEOE$)
(15) $old_F = F(X)$
(16) $F(X) = (1 - P(X))^{N(X)}$
(17) $SS(X) *= old_F / F(X)$

Figure 4: Fanout Computation Algorithm

Second, fanout F and sequence size SS of every navigation XPS X is computed in a single top-down traversal of the XPS tree by the algorithm of Figure 4.

We model each step as navigation that computes a sequence of $N(X)$ items, followed by a predicate that applies to each item uniformly and independently. For an XPS marked as FOR, the $N(X)$ is the average number of XML items the navigation will find, per parent context. On line 3, Y is the XPS parent step of X. The $St(X)$ is the number of items found by a linear path from root to X, as estimated using the data distribution statistics. If X is the root we define $St(Y) = 1$. Notice, that $F(X)$ computation for FOR steps does not take into account the sequence size of the input – it is taken into account by the parent computation, if the parent is a LET step.

For an XPS marked as LET the computation is more involved (lines 6-13), since it needs to account for the size of the context (parent) sequence. Also, sequences constructed by this XPS may need to be iterated a number of times by FOR-marked children, producing a Cartesian product for every sequence. In this case, the size of every sequence needs to be raised to the K-th power, where K is the number of FOR-marked children. In statistics, the average value of the elements of some list raised to the K-th power is called the *K-th moment*[15]. We assume that for any XPS X the probability of a predicate evaluating to $true$ is the same $P(X)$ for every element in the initial sequence of size $N(X)$ constructed by the navigation. Thus, final sequence sizes after the predicate application are distributed according to a binomial distribution with parameters $N(X)$ and $P(X)$. The K-th moment of a binomial distribution $B(N(X), P(X))$ is:

$$\sum_{i=0}^{N(X)} C_i^{N(X)} * P(X)^i * (1 - P(X))^{N(X)-i} * i^K$$

In our case, K is the number of children of X that are marked as FOR. Computationally simple closed formulas exist for small values of K, which is likely to be the case in practice. For larger K's there are well known approximation techniques [15]. The K-th moment computation does not make sense for $N(X) \leq 1$. In this case we assume that only singleton sequences are produced (lines 11-13).

To handle XPS marked as empty-on-empty (EOE), line 16 adds the number of empty sequences returned to $F(X)$. The $SS(X)$ is updated in line 17 to account for the fact that the total number of result nodes is still the same, but they are now spread over the new (larger) number of sequences.

Fanout of the navigation expression as a whole is computed after the entire XPS tree is traversed and all the node fanouts are assigned. Tree fanout is the product of all navigation node fanouts.

5.1 Local Optimization

In many cases, if an XPS tree does not contain any selective predicates, or if these predicates have already been applied by the index, or if the entire document is small enough to fit on a single page, etc., a single document traversal that skips unnecessary fragments is in fact the optimal execution strategy for an XSCAN operator. However, many queries, such as Query 2 in Section 1, may perform better with two or more passes over the document.

The task of the local optimizer is to partition the XSCAN's XPS tree into a sequence of tree fragments. The XSCAN runtime will execute each fragment in the sequence, only if the previous fragment returns a non-empty result. Fragment execution is implemented by recursive calls to the navigation runtime, utilizing the multipass processing feature of *DB2 pureXML* [3, 4].

In the worst case, the optimal fragment sequence may contain as many fragments as there are XPath Steps in the expression. However, in practice very few fragments are usually needed to achieve

near-optimal performance. The first fragment in the sequence has by far the greatest effect on the XSCAN performance.

Based on this observation, we devised an algorithm that combines greedy partitioning, with a dynamic programming fragment ordering.

The greedy partitioning algorithm works as follows. First, every *single path* fragment, i.e., some XPS node X and all its ancestors, is considered for the role of the first fragment in the sequence. The resulting XSCAN costs are estimated, assuming no further partitioning, and the fragment that results in the cheapest XSCAN is picked. If the cheapest resulting XSCAN is still more expensive than a single-pass evaluation, the algorithm terminates and a single-pass XSCAN evaluation is picked.

Next, we consider adding other paths into the first fragment as long as the addition lowers the overall XSCAN cost estimate. The most beneficial paths are added first. Once the beneficial path additions are exhausted, the first fragment is finalized. The XPS nodes of the tree, not included in the first fragment, now form one or more tree fragments. Each of these fragments could possibly be split further, so the same partitioning algorithm is recursively applied to each one of them.

The algorithm has polynomial complexity. In the worst case it will require $O(N^3)$ executions of XSCAN costing formulae, where N is the number of XPS nodes in the expression tree. Each XSCAN cost estimate is computed by a single tree traversal in at most $O(N)$ time.

The resulting partitions are ordered using standard *DB2 pureXML* join ordering algorithms, i.e., full enumeration of the ordering alternatives using dynamic programming, unless the number of fragments is too large in which case greedy join ordering is used. The join ordering algorithms respect dependencies between fragments. For instance if fragment A contains parent of the root node of fragment B, then A must be executed before B. Thus, first fragment picked by the greedy algorithm is guaranteed to remain first, since it contains the root of the XPS tree, which contains the context node.

The partitioning algorithm relies on cost estimation, which is done by modeling navigation and buffering work that needs to be performed by an XSCAN operator in order to execute a given sequence of tree fragments. The details of the cost model are beyond the scope of this paper.

6. EXPERIMENTAL RESULTS

In this section, we present experimental evaluation of the global and local optimization techniques, and the MEP XML navigation operation described in this paper in in *DB2 pureXML*. We investigate the performance of XQuery and SQL/XML queries with and without the use of MEP and our optimization strategies.

All experiments were run on a 4 processor system 1.4 GHz POWER 4 Power PC system with one of the processors dedicated to the experimental evaluation. The database bufferpool uses 40 MB of RAM.

In our experiments we used two datasets: XBench [32] and a dataset based on the TPCH database benchmark [9]. For the TPCH dataset we generated 15000 XML documents; each document contained data from a single row of the Orders table and all the corresponding rows of the LineItems table. In other words, each document contained an `Order` root element, with columns from the Order table as children as well as all the corresponding line items from the Lineitems table nested as children. For each document we randomly generated from 0 to 70 line items. Represented as text, the size of the dataset is 250MB.

Based on the TPCH database, we crafted a query with an

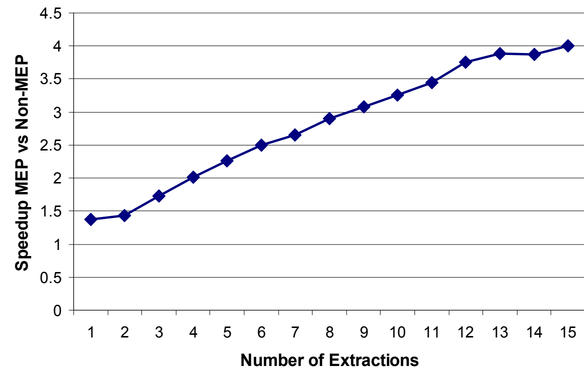


Figure 5: Speed-up of XMLTABLE query evaluation.

XMLTABLE function that extracts multiple column values from LineItem elements of XML documents. Below is a version of this query that extracts three columns.

```
select x."SuppKey"
from tpchsmall,
      xmltable('$doc/Order/LineItem'
                passing doc as "doc"
Columns
"PartKey" int PATH 'PartKey/text()',
"SuppKey" int PATH 'SuppKey/text()',
"LineNumber" int PATH 'LineNumber/text()'
) as x;
```

In order to quantify the impact of MEP, we varied the number of extracted columns, and measured the time for evaluation of the extraction portion of the query plan (ignoring result construction). The speed-up resulting from use of MEP is shown in Figure 5. The plan that does not use MEP, first extracts a reference for the line item node, and then passes it to a number of consecutive XSCAN operators that extract one column at the time. The MEP plan contains a single XSCAN operator. The experiments show that the MEP approach scales much better in terms of query evaluation time. The speedup ranges from 50% to 300% and increases almost linearly with the number of extraction points.

Figure 6 shows the graph of the query compilation times for the same XMLTABLE query set. For the 15 extraction points, the query compilation time is 0.2 timerons[4] with MEP and 5.11 without MEP. The compile time for non-MEP plans tops at 13 extraction points. Up to this point, our query optimizer used a dynamic programming algorithm to enumerate all possible execution orders of XSCANs. After this point, the plan search space became too large for the dynamic programming algorithm, and the optimizer switched to a greedy algorithm, which is more efficient, but can produce suboptimal plans. We conclude that using MEP substantially reduces the query compilation time and expands the range of queries for which we can use better optimization methods.

We study the effect of optimization strategies using Query 1. Figure 7 shows its performance as a function of selectivity of the query predicate `[TotalPrice > X]`. We compare three approaches: no MEP merging, resulting in the query plan of Figure 1 (b), full MEP merging as in Figure 1 (a), and our global optimization approach, which clusters the query XPath expressions into three XSCAN operators and allows the cost based optimizer to pick between

[4]We do not report actual compilation or running times, because our experimental evaluation was done on a commercial system.

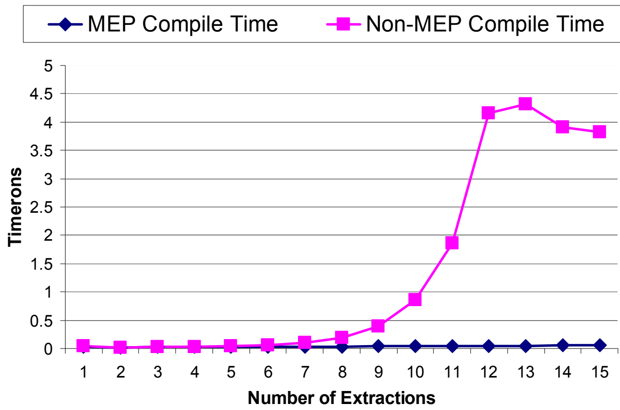


Figure 6: XMLTABLE query compilation time.

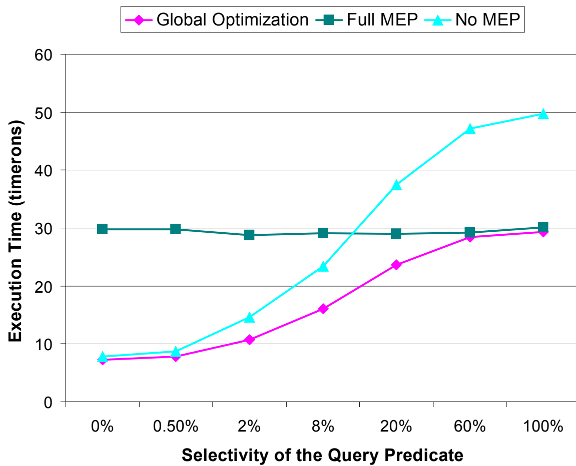


Figure 7: The execution time of Query 1 under different optimization strategies.

plans of Figure 1 (c) and (d). Local optimization is not necessary for this experiment, because indexes are used (as shown in Figure 1) to eliminate non-qualifying documents. Figure 7 shows that our global optimization indeed takes full advantage of simultaneous execution of multiple XPath expressions, without sacrificing the optimal plan.

To illustrate the need for local optimization we ran the single--document part of the XBench benchmark [32]. The queries of this benchmark do not provide many opportunities for MEP to improve performance, but they show off the danger of merging the local predicates and subsequent extractions together. To further enhance the effect, we ran the benchmark without any use of indexes. Figure 8 shows that merging multiple expressions into a single XSCAN, without doing local optimization, seriously degrades the performance. The local optimization by itself is somewhat useful in this scenario, but it becomes critical once the expressions are merged.

7. CONCLUSION

In this paper, we described the high-granularity XML query optimization in *DB2 pureXML*. To our knowledge, this is the first complete framework for optimizing complex XQuery and SQL/XML queries by utilizing a high-granularity streaming XPath processor.

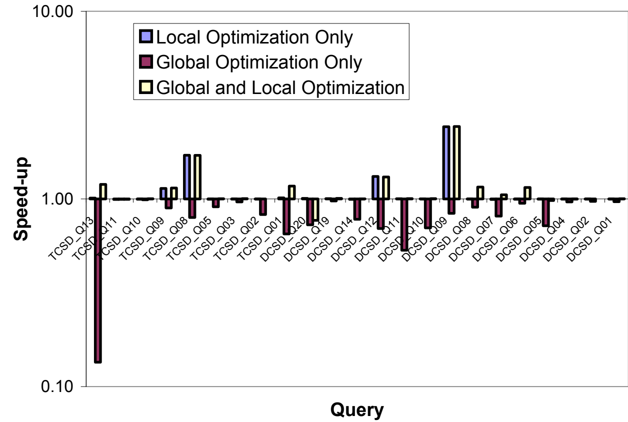


Figure 8: The speed-up of XBench benchmark queries.

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. The results of our experimental study shows that blindly combining all XPath expressions on a single document for simultaneous execution can significantly degrade query performance. However, with our optimization strategy the high-granularity paradigm allows for better query evaluation performance and reduces the query optimization complexity. We expect many XQuery and SQL/XML queries, such as those with XMLTable function, to benefit significantly from this new approach.

8. REFERENCES

[1] Shurug Al-Khalifa and H. V. Jagadish. Multi-level operator combination in xml query processing. In *CIKM*, pages 134–141, 2002.

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

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

[4] K. Beyer et al. DB2 Goes Hybrid: Integrating Native XML and XQuery with Relational Data and SQL. *IBM Systems Journal*, 45(2):271–298, 2006.

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

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

[7] C. Koch and S. Scherzinger and N. Schweikardt and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *In Proc. of ICDE*, 2004.

[8] C. Re and J. Simeon and M.F. Fernandez. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. of ICDE*, 2006.

[9] Transaction Processing Performance Council. TPC Benchmark H available at http://www.tpc.org/tpch/.

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

[11] D. Florescu et al. The BEA Streaming XQuery Processor. *VLDB Journal*, 13(3), 2004.

[12] A. Halverson et al. Mixed Mode XML Query Processing. In *VLDB*, pages 225–236, 2003.

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

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

[15] N. L. Johnson, S. Kotz, and A. W. Kemp. *Univariate Discrete Distributions*. "John Wiley and Sons", 1992.

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

[17] 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.

[18] M. Krishnaprasad et al. Query Rewrite for XML in Oracle XML DB. In *Proc. of VLDB*, pages 1122–1133, 2004.

[19] M. Brantner and S. Helmer and C.-C. Kanne and G. Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *In Proc. of ICDE*, 2005.

[20] J. McHugh et al. Lore: A database management system for semistructured data. In *SIGMOD Record*, 1997.

[21] J. F. Naughton et al. The niagara internet query system. In *IEEE Data Engineering Bulletin*, 2001.

[22] M. Nicola and B. Van der Linden. Native XML Support in DB2 Universal Database. In *Proc. of VLDB*, pages 1164–1174, 2005.

[23] 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.

[24] M. Rys. XML and relational database management systems: inside Microsoft SQL Server 2005. In *In Proc. of SIGMOD*, pages 958–962, 2005.

[25] J. Shanmugasundaram et al. A General Technique for Querying XML Documents Using a Relational Database System. *SIGMOD Record*, 30(3):20–26, September 2001.

[26] I. Tatarinov et al. Storing and Querying Ordered XML Using a Relational Database System. In *In Proc. of SIGMOD*, pages 204–215, 2002.

[27] T.H. Cormen and C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

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

[29] *XQuery 1.0 and XPath 2.0 Data Model*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/xpath-datamodel`.

[30] Y. Diao and M. J. Franklin. Query Processing for High-Volume XML Message Brokering. In *In Proc. of VLDB*, 2003.

[31] Y. Wu and J. M. Patel and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *In Proc. of ICDE*, 2003.

[32] B. Yao, T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *ICDE*, pages 621–633, 2004.

[33] Z. Chen and H.V. Jagadish and L.V.S. Lakshmanan. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. of VLDB*, 2003.

[34] Z.H. Liu and M. Krishnaprasad and V. Arora. Native Xquery processing in Oracle XMLDB. In *In Proc. of SIGMOD*, pages 828–833, 2005.

Trademarks

IBM, DB2, and pureXML are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.