Cost-based optimization in DB2 XML

A. Balmin
T. Eliaz
J. Hornibrook
L. Lim
G. M. Lohman
D. Simmen
M. Wang
C. Zhang

DB2 XML is a hybrid database system that combines the relational capabilities of DB2 Universal Database™ (UDB) with comprehensive native XML support. DB2 XML augments DB2® UDB with a native XML store, XML indexes, and query processing capabilities for both XQuery and SQL/XML that are integrated with those of SQL. This paper presents the extensions made to the DB2 UDB compiler, and especially its cost-based query optimizer, to support XQuery and SQL/XML queries, using much of the same infrastructure developed for relational data queried by SQL. It describes the challenges to the relational infrastructure that supporting XQuery and SQL/XML poses and provides the rationale for the extensions that were made to the three main parts of the optimizer: the plan operators, the cardinality and cost model, and statistics collection.

INTRODUCTION

As XML has been increasingly accepted by the information technology industry as a ubiquitous language for data interchange, there has been a concomitant increase in the need for repositories that natively store, update, and query XML documents. Together with extensions to SQL (Structured Query Language) for formatting relational rows into XML documents and for querying them, called SQL/XML,[1] XQuery has emerged as the primary language for querying XML documents, with the publication in 2005 of a draft standard for the language.[2] XQuery combines many of the declarative features of SQL and the document navigational features of XPath,[3] but subsumes neither.

Despite the ascendancy of XML, SQL/XML, and XQuery, the huge investment in relational database technology over the last three decades is unlikely to be supplanted abruptly. Hence the XML "revolution" is more likely to be a gradual evolution, in which XML documents will be stored in relational tables and queried interchangeably by either SQL or XQuery for the foreseeable future.

Accordingly, IBM has developed DB2* XML, a hybrid database system that combines the relational capabilities of DB2 Universal Database* for Linux**, Unix**, and Windows** with comprehensive native XML support. This means that XML is supported as a native data format alongside relational tables, and XQuery is supported as a second query language alongside SQL. Moreover, the hybrid nature of DB2 XML makes it much easier to fully support the SQL/XML language, which allows arbitrary nesting of SQL and XQuery statements within each other.

To support query processing in the XQuery and SQL/XML languages on the native XML store, the DB2 XML team has extended existing components in DB2 UDB with support for XQuery and added new components. The overall rationale and architecture of DB2 XML, as well as an overview of the design of its major components, are described in Reference 4, including the new native XML store, XML indexes, query modeling, and query processing. In this paper we describe in more detail the extensions made to the cost-based optimizer portion of the DB2 UDB query processor to efficiently support XQuery and SQL/XML.

XQuery language

In explaining the role of the DB2 XML optimizer, we first review how DB2 XML stores XML data and how XQuery queries it. In DB2 XML, a new native XML type is introduced to represent XML data. Tables can be created having one or more columns of this XML type, with each row in any XML column containing an XML document, or, more precisely, an instance of the XML Query Data Model.[5] As with other column types, the contents of XML columns can be indexed optionally by one or more indexes. Example 1 shows the creation of a table with an XML column, the insertion of an XML document into that column of the table, and the creation of two XML indexes on that column.

Example 1

```
create table Product (
      pid varchar(10) not null primary key,
      Description xml
);
insert into Product values(
'100-100-01',
xmlparse(document
   '<product pid="100-100-01">
      <description>
         <name>Snow Shovel, Basic 22"</name>
         <details>
              Basic Snow Shovel, 22" wide,
              straight handle with D-Grip
         </details>
         <price>9.99</price>
         <weight>1 kg</weight>
      </description>
      <category>Tools</category>
    </product>'
preserve whitespace)
);
create index I_PRICE
      on Product(Description)
      generate key using xmlpattern
      '//price' as sql double;
create index I_CATEGORY
      on Product(Description)
      generate key using xmlpattern
      '/product/category' as sql varchar(10);
```

In this example, `//price` and `/product/category` are XPath patterns. The last two statements in Example 1 define indexes `I_PRICE` and `I_CATEGORY` that contain references to only those nodes in "Description" documents whose root-to-node paths match these XPath patterns, organized by the values of such nodes. The "//" notation in the first XPath pattern permits any number of nodes between the root node of each document and an instance of a `price` node.

XQuery resembles SQL in that it is largely declarative; that is, it specifies what data is desired, not how to access that data. Each XQuery statement contains a FLWOR (for, let, where, order by, and return, pronounced "flower") expression, which specifies (1) zero or more FOR and LET clauses that describe the data to be accessed, (2) an optional WHERE clause that defines conditions on that data, (3) an optional ORDER BY clause for ordering the result, and (4) a RETURN clause that specifies the structure of the data returned by that query. The FOR and LET clauses can optionally assign intermediate results to variable names, denoted by a preceding '$'.

The FOR clause can be thought of as an iterator that accesses items from XML data, creating one row per item. The LET clause effectively arranges those data items into a sequence in one row. This mapping in DB2 of XQuery items to rows and XQuery FOR clauses to the iterators used in processing relational rows is crucial for exploiting much of the existing infrastructure of DB2, as we explain in the section "Plan generation."

Example 2 shows a sample XQuery that returns all products having a price less than 100 and a category of "Tools." The FOR clause iterates over the product nodes in all documents of PRODUCT.DESCRIPTION that match the given XPath pattern, assigning each to the variable $i. Those whose category is "Tools" survive the filtration of the WHERE clause and are RETURNed to the user. This query has no LET clause.

Example 2

```
for $i in
db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')
     //product[.//price < 100]
where $i/category = 'Tools'
return $i;
```

Although the semantics of the XQuery language require that results be returned in the order specified by any nested FOR clauses, the requirement does not mandate the strategy for evaluating those clauses by an optimizer, and many aspects of XQuery, such as nested FOR loops and XPath navigation, partially restrict the order in which it should be processed. XQuery has enough alternative execution choices to need cost-based optimization in the same way that SQL queries do. The above example illustrates that even simple XQuery queries require many of the same optimization decisions required for SQL queries. Because a DB2 XML user can define multiple XML indexes on an XML column, as well as a traditional index on any combination of relational columns, the optimizer must decide which of these alternative access paths, either individually or in combination, to exploit in evaluating a query.

A *plan* is defined as an ordered set of steps used to access information. Alternative plans for the query in this example might exploit the I_PRICE index, the I_CATEGORY index, both indexes (ANDed together), or neither. The choice of the optimal plan depends on the characteristics of the database and may have enormous impact on the query's performance. For example, if most of the products fall into the "Tools" category, but very few of them have a price less than 100, then the plan that uses the I_PRICE index will be much more efficient than one that scans the entire table or one that accesses both I_PRICE and I_CATEGORY indexes and intersects the result.

Although our simple example did not illustrate it, XQuery permits join predicates, that is, WHERE clauses or XPath predicates that relate the values of multiple columns, or nodes from documents in multiple XML columns. Similar to relational predicates that were proven to be commutative and associative by using relational algebra, XQuery predicates may largely be reordered in a similar way, potentially providing huge performance benefits. Hence, the DB2 XML optimizer still needs to determine the best way to order those joins and the best join method (algorithm) to accomplish each join. This is the major contributor to complexity in SQL optimizers. These and other considerations offer many opportunities for optimization of XQuery queries.

Challenges

Why then is it not possible to reuse the results of three decades of research and experience with relational query optimization to optimize XQuery queries? It is in fact possible for the most part, but XQuery introduces several major new challenges. First and foremost of these is heterogeneity. SQL optimization was significantly aided by the simple homogeneity of rows in relational tables having identical "flat" schemas. In contrast, the XML data model is inherently heterogeneous and hierarchical. For a given XML schema, one or more elements may be missing in any XML document, and there is no requirement for explicit NULL values for these elements. LET clauses effectively construct varying-length rows containing sequences of elements whose number is difficult to estimate and may vary from row to row; a FOR loop over such a sequence removes the nesting of that sequence and changes it into as many rows as there were elements in a single row. We further postulate that XML schemas themselves are likely to change frequently from document to document, or even be unavailable or unknown for a given XML document, leading to schema heterogeneity within even a single table containing a single XML column.

Another major challenge was engineering a hybrid optimizer that facilitates combining XML and relational access paths and is able to interchange and optimize both SQL and XQuery operations in a unified framework on an equal footing, even when combined within the same SQL/XML statement. A unified optimizer is crucial for enabling the DB2 XML system to efficiently support mixed relational and XML workloads. Our approach to query optimization in DB2 XML, therefore, reuses the existing infrastructure in DB2 UDB for relational query optimization, extending it where needed to meet the new challenges posed by XQuery and XML documents. Because XQuery requires the same fundamental choices of access path, join order, and join method as in relational optimization, it makes sense to largely reuse the DB2 UDB infrastructure that generates alternative query execution plans (QEPs) and estimates an execution cost for each by first estimating the size (or "cardinality") of each intermediate result, using statistics on the database that are collected beforehand. Exploiting the relational infrastructure permits DB2 XML to inherit the existing functionality, scalability, and robustness of DB2 UDB and reduce its time to market.

Organization

The paper is organized as follows. We describe related work in the second section. In the third section we give an overview of the DB2 optimizer and point out specific challenges that had to be addressed in DB2 XML. In the following three sections, we describe the design and XML extensions of each of the three main parts of the optimizer; namely, plan generation, cardinality and cost modeling, and statistics collection. Then we list directions for future work, and finally, we summarize our conclusions.

RELATED WORK

In recent years, many different approaches have been proposed for XML data management. Approaches that reuse the relational DBMS (database management system) and translate XML queries to SQL do not require changes to the relational optimizer, whereas approaches that store XML data natively benefit from an XML-specific optimizer. Although numerous papers on XML query processing have been published, only a few have addressed cost-based optimization of XQuery queries. Most of these adapt or extend relational optimization techniques. Major native XML data management systems that employ cost-based optimization include Lore,[6] Niagara,[7] TIMBER,[8] Natix,[9] and ToX.[10] Most of these systems translate XML queries into some logical algebra first, whereas DB2 XML represents an XQuery query in its internal entity-relationship representation, called the query graph model (QGM).

At the time of this writing, both Microsoft[11] and Oracle[12] have released or announced support for a wide spectrum of XML and XQuery functionality in their database products. However, to our knowledge, no prior work describes in detail a cost-based optimizer for an XQuery compiler, let alone a relational-XML hybrid. Because the runtime operators supported by the preceding systems have different capabilities and complexity, they differ in the way that they evaluate the plans constructed with those operators.

The techniques used in these systems for evaluating path expressions can roughly be divided into two categories: *structural joins* that process path expressions in small steps at a time and then join the results, and *holistic algorithms* that process complex path expressions in one operator. DB2 XML takes the holistic approach, evaluating an entire path expression with a single operator by invoking an adaptation of the stream-based TurboXPath algorithm.[13] The holistic operator makes use of efficient algorithms and optimization heuristics to compute a complete path expression in one pass through the document, without generating large intermediate results. Furthermore, the holistic approach reduces the number of plans generated in comparison to systems using structural joins[7,8] and does not necessarily sacrifice plan quality.

The first cost-based optimizer for an XML query system was implemented by the Lore system.[14] The Lore query language was based on the Object Query Language (OQL) and did not have XML sequences, which are central to the XQuery language. Lore had a number of indexes supporting various operations, and it enumerated various plans with different access methods, much as relational optimizers do. To estimate the cardinality of path expressions embedded in a query, the Lore optimizer used a Markov model technique.

This approach was extended by the Markov table method of Aboulnaga et al.[15] and further improved by XPathLearner.[16] Lore also proposed a "DataGuides" data structure for compactly summarizing the structure of each semistructured document. This idea was later extended with correlated subpath trees,[17] XSketch,[18,19] and TreeSketch structures[20] that were designed for structural join cardinality estimation. Other work on XML cardinality estimation includes StatiX,[21] position histograms,[22] Bloom histograms,[23] and CXHist.[24] These cardinality estimation techniques differ in whether they are online or offline algorithms, whether they handle subtree queries or linear path queries only, whether they handle leaf values, and whether the leaf values are assumed to be strings or numbers.

OVERVIEW OF DB2 XML COST-BASED OPTIMIZATION

To understand the extensions to the DB2 optimizer needed to support XQuery, it is first necessary to understand the context in which those extensions were made. This section therefore gives a brief overview of the DB2 optimizer's approach to relational query optimization.

The DB2 cost-based query optimizer is responsible for determining the most efficient evaluation strategy for an SQL query. There are typically a large number of alternative evaluation strategies for a given query. These alternatives may differ significantly in their use of system resources or response time. A cost-based query optimizer uses a sophisticated enumeration engine (i.e., an engine that enumerates the search space of access and join plans) to efficiently generate a profusion of alternative query-evaluation strategies and a detailed model of execution cost to choose among those alternative strategies.

There is an extensive body of work on query-evaluation strategies and cost-based query optimization for relational query languages such as SQL. The IBM research and development community has made significant contributions in both of these areas over the past three decades.[25–30] This legacy continues with System RX[4] and DB2 XML. This section provides a high-level overview of the DB2 XML cost-based optimization architecture, highlighting the key aspects of the architecture that were modified or extended in support of the XQuery and SQL/XML languages. The remaining sections of the paper provide more detailed descriptions of each of these key aspects.

DB2 XML query compilation

Cost-based optimization is part of a multistep query compilation process, as illustrated by ***Figure 1***. Language-specific parsers first map SQL or XQuery to an internal representation, called the *query graph model* (QGM). The QGM represents the entities of the query and their relationships in a way that captures the semantics of both languages. Next, the query rewrite phase employs heuristics to transform the QGM into a more optimization-friendly representation. It eliminates unnecessary operations and reorders or merges other operations to provide the cost-based optimizer with more options for accessing tables and reordering joins.

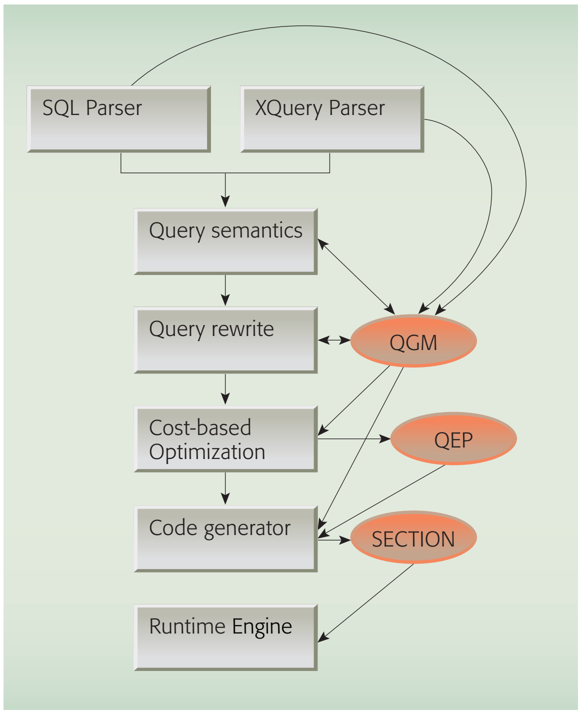

Figure 1
DB2 XML query compilation

For this transformed QGM, the cost-based optimizer then determines the most efficient evaluation strategy, called a *query execution plan* (QEP). The code-generation phase then maps the QEP to a sequence of execution engine calls, called a *section*. The section is stored in the database and interpreted by the runtime engine whenever this query is executed.

Generation of alternative query execution plans

The number of alternative QEPs for a given query is typically vast. This stems from both the large number of equivalent logical query representations it might have, due mainly to the commutative and associative nature of relational join operations, as well as from the number of possible implementations for each logical representation. For example, the equivalent logical join sequences `JOIN(JOIN(CUSTOMER, ORDERS),LINEITEM)` and `JOIN(JOIN(LINEITEM, ORDERS), CUSTOMERS)` are valid for the query in Example 3, which is taken from the TPC-H[31] industry-standard benchmark. Moreover, either of these logical join sequences can have many implementations, depending upon available table access methods, join methods, and so forth.

Example 3

```
SELECT *
FROM LINEITEM L, ORDERS O, CUSTOMER C
WHERE C.CUSTKEY = O.CUSTKEY AND
O.ORDERKEY=L.ORDERKEY AND
C.NAME = 'Acme' AND
L.PRICE > 1,000
ORDER BY O.ORDERDATE
```

As each partial QEP is generated, its execution cost is estimated and compared to other QEPs producing equivalent results, so that the more expensive QEPs can be pruned. The DB2 XML optimizer descends directly from Starburst[27–29] and thus shares the following key aspects of its QEP generation architecture:

- *A set of QEP building blocks called operators*—Each operator corresponds to a query processing primitive implemented by the runtime engine. All QEP operators consume and produce a common object: a table. The QEP produced by the optimizer is essentially a nested sequence of operators. The operator-oriented nature of the QEP makes it easy to model extensions to the runtime engine.
- *A set of plan generation rules*—These rules define how operators may be combined into QEPs. They determine how each logical representation of the query is mapped to its various alternative implementations. The optimizer's repertoire of alternatives can be extended simply by adding new rules for composing operators into QEPs. Moreover, the alternatives considered for a given query can be expanded or contracted by enabling or disabling these rules.
- *A configurable enumeration engine*—The enumeration engine drives the QEP generation process by progressively invoking the plan generation rules. The most important task of the enumeration engine is to determine which logical sequences of binary joins to evaluate. Configuration parameters allow the user to control the topology of binary join trees considered and other aspects of the process.[32]

The enumeration engine drives the plan generation process in a "bottom-up" fashion, one query block at a time. Alternative QEPs for accessing the tables and subqueries referenced in the query block are generated first, the former by invoking the rules responsible for constructing alternative table access implementations, and the latter by calling the enumeration engine recursively. Logical join sequences are then progressively enumerated starting with joins of size two, size three, and so on. Partial QEPs generated in previous steps are reused in the construction of alternatives for the current step. In addition to driving the table access and join rules, the enumeration engine invokes rules that build plans for other query clauses, such as GROUP BY, ORDER BY, DISTINCT, UNION, and so forth.

As each QEP is constructed, the optimizer computes the properties and cost of each operator in that QEP. These properties characterize key aspects of the partial query result produced by the operator, such as its estimated result size, its order, the predicates applied, and so on. At each step of the enumeration process, the optimizer prunes alternatives that are inferior to another alternative in terms of estimated execution cost and other properties, as described next.

We were able to make a straightforward extension to the existing QEP generation architecture in support of XQuery by using familiar table metaphors to represent XPath evaluation and sequences. The internal representation of the query was extended so that XPath navigation, as expressed in XQuery's FOR or LET clauses, is represented to the optimizer as an invocation of a special *table function*.[4] A table function is a QGM entity that represents any generic computation and produces a stream of rows. This new table function essentially takes an XML column and an XPath expression as input and produces a table in which each result row contains a sequence of XML items satisfying the XPath expression. Using table references to represent XPath navigation allowed us to use the existing enumeration engine "as is" to generate alternative sequences for evaluating XPath and relational expressions. We added new QEP operators to model runtime primitives for XPath navigation and XML indexes to gain direct access to documents satisfying XPath predicates. We also added rules for combining the new operators into alternative QEPs for evaluating XPath expressions. Because we represent the results of these new operators as tables, we can use existing relational operators to join, combine, and perform other types of manipulations on their output. The details of these extensions are described in the section "Plan generation."

Modeling cost and cardinality

Each QEP operator maintains a running total of the projected I/O, CPU, and (in distributed environments) communications resources required to produce its result. How these components of cost are combined into a total cost figure is operator-independent and depends upon whether the system is optimizing to maximize throughput or to minimize response time. An operator estimates its contribution to each of these cost components by using a detailed model of its execution behavior. The model must take into account detailed aspects of execution, such as the algorithmic behavior of the operator, its memory requirements, its interaction with the I/O subsystem, and so on.

The most critical input to an operator's cost model is the number of records that it processes, which is first estimated by the cardinality model. Starting with statistics about the database, such as the number of rows in each table and the number of distinct values in each column, the cardinality model estimates the filtering effect of operations such as predicate application or aggregation. DB2's cardinality model is largely based on the probabilistic model proposed in System R.[26]

Each filtering operation is assigned a *selectivity*, which represents the probability that a given row qualifies for the filtering operation. Selectivity estimates are derived from statistics that characterize the value distribution of the columns referenced in the filtering operation. Uniform distributions might be characterized simply by using the number of distinct column values and the range of values. Nonuniform column distributions require more detailed statistics, such as frequent values or histograms.[33] Cardinality estimation occurs incrementally, by progressively multiplying the cardinality of base tables by the selectivity of each filtering operation applied as a QEP is constructed. Adjustments to these cardinality estimates are applied if statistics exist indicating the extent to which the columns referenced in multiple filtering operations are not independent.

Building an accurate cost and cardinality estimation model is a complex and tedious process. Even algorithmically straightforward operations such as accessing a B-tree index (a type of index using a balanced tree structure) must take into account details such as the physical layout of the index, how the index keys are clustered relative to the data pages, the amount of memory available for buffering disk pages, and so on. The new XML operators are inherently more complex due to the heterogeneous and hierarchical nature of XML document collections. Consequently, building an accurate cost model for operators that manipulate XML data is far more difficult. (Details on estimating the cost for the new XML operators can be found in the section "Cost estimation.")

Moreover, the heterogeneous and hierarchical nature of XML complicates the process of cardinality estimation. For example, determining the number of items satisfying an XPath expression such as `/customer[name="Acme"]/order[lineitem/price > 1,000]` must not only take into account the selectivities of the individual predicates `/customer[name="Acme"]` and `/customer/order[lineitem/price > 1,000]`, but also the structural relationship between nodes that might satisfy those predicates; that is, nodes satisfying the individual predicates must descend from the same customer node.

If one were to make an analogy to relational cardinality estimation, estimating the number of items that satisfy an XPath expression involves many of the same complexities as estimating cardinality after a series of join operations. In fact, determining the number of nodes reached by the XPath expression `/customer[name="Acme"]/order[lineitem/price > 1,000]` is equivalent to estimating the cardinality of the TPC-H query in Example 3.

The extensions made to the cardinality estimation model in support of XPath are discussed in the section "Cardinality and cost estimation." We introduce a new metric called *fanout*, which is used to determine the number of items that can be reached by XPath navigation. Fanout is used in conjunction with the traditional notion of selectivity in determining the cardinality of XPath navigation. The section "Statistics collection" describes the set of statistics used to make XML cost and cardinality estimates and discusses some of the challenges involved in making the XML statistics collection process efficient.

PLAN GENERATION

The addition of new runtime operations for processing XML documents required corresponding extensions to generate new plans that would invoke these operations in the DB2 optimizer. First, we needed to define three new optimizer operators to represent each of the corresponding runtime operations. Second, we added new plan generation rules to assemble these operators into valid plans whose cost would then be estimated. The heterogeneity of XML documents and schemas, as well as the anticipated volume of XML repositories, significantly increased the need for highly effective indexing schemes; thus, significantly more sophisticated exploitation of the XML indexes was required. Finally, we considered whether extensions were needed to the plan properties required for distinguishing one plan from another for pruning purposes, or to the enumeration of alternative plans. Fortunately, the approach we took for representing XPath expressions as table functions limited the changes needed to properties and enumeration. This section describes each of these changes in more detail.

New operators for XML

In DB2 XML, new runtime algorithms were devised to process path expressions operating on the native XML store and on XML indexes. Correspondingly, we introduced three new operators in the optimizer: XSCAN, XISCAN, and XANDOR. Each is described in more detail next. The encapsulation in DB2 of each runtime function as a QEP operator allows these new operators to be added quite easily in the optimizer and to be interleaved with existing operators to form new plans.

The XSCAN operator

The new XSCAN operator enables scanning and navigating through XML data to evaluate a single XPath expression. It takes references to XML nodes as input, and these are used as starting points for navigation; it returns references to XML items that satisfy the path expression. The runtime operation that XSCAN invokes utilizes an adaptation of the stream-based TurboXPath algorithm[13] on preparsed XML documents stored as paged trees. The XSCAN runtime algorithm is described in Reference 4. The algorithm processes the path query by traversing the document tree and is able to skip portions of the document that are not relevant to the query evaluation. The path expressions supported by the XSCAN runtime are quite powerful; the expressions contain various axes (i.e., direction-setting query components) and wild cards, including '//' (descendant and self), '..' (parent), '*' (any node), and value predicates.

The XISCAN operator

The new XISCAN operator enables direct access to a subset of documents of interest through an XML index that is limited by an index expression and is analogous to a relational index scan. It takes as input an index expression that consists of a linear XPath, a comparison operator, and a value (e.g., `//price` $<$ `100`), and returns the row identifiers (RIDs) of documents that contain matching nodes.

Before plan generation is begun, it is decided whether an XML index can be used to process a query by a sophisticated index-matching process[34] that tries to find subexpressions of the original query which match the XPath expression in the definition for each XML index. If there is a match, it produces an index expression that associates the matched portion of the path expression with the matching index. See Reference 4 for a detailed description of XML indexes and how they are used for processing queries.

The XANDOR operator

XANDOR (XML index ANDing and ORing) is a new n-ary operator that simultaneously combines ANDing and ORing of multiple XML index accesses, effectively performing an n-way merge of its inputs, which are scans on individual indexes. The XANDOR operator invokes an algorithm similar to holistic twig joins.[35] Having the n-way context permits the XANDOR, when opening parallel scans on multiple indexes, to use the node identifiers it reads from one index to skip forward in the scan of other indexes past regions where results are impossible. This not only improves performance but also effectively makes the ordering of its inputs dynamic. As a result, the XANDOR is far more robust to any estimation errors in the optimizer's compile-time decisions (concerning which inputs to include and how to order them) than traditional index ANDing and ORing.[36] The initial implementation of the XANDOR only supports ANDing and is limited to equality predicates. We expect to relax these restrictions in the future.

Extensions for generating plans

In the main algorithm for generating plans in DB2, access plans are first generated for accessing each table individually; these are then combined in different sequences by the join enumeration algorithm. Internally, the individual access plans are first generated by the invocation of generic *access rules* for each base table. Join plans are then generated by a separate set of generic *join rules* for each set of two or more tables that is generated by the join enumerator, in successively larger sets. In the following, we describe the changes to the access rules which are needed to construct appropriate plans using the new XML operators just described. Rather than detailing the exact rules, we show instead the plans that those rules produce.

Extensions to access rules

The extensions to the access rules were fairly straightforward for XSCANs, as illustrated by Example 4 and its resulting XSCAN plan, which is shown in *Figure 2*.

Example 4

```
for $i in
db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')
//product[.//price < 100]
return $i;
```

For this query, a straightforward plan is to access the `DESCRIPTION` documents one at a time from the `PRODUCT` table, and then evaluate the path expression `//product[.//price < 100]` for each document. This is analogous to applying a predicate in a table scan in a relational system, but the path expression can be significantly more complex. This plan is shown in Figure 2. The XSCAN operator evaluates the path expression and returns references to qualifying product nodes.

Generation of XML index plans

Plans for XML index scans are quite a bit more complicated to generate than plans for XSCANs. We illustrate this with the following example.

Example 5

```
for $i in
db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')
//product[.//price < 100]/@id
return $i;
```

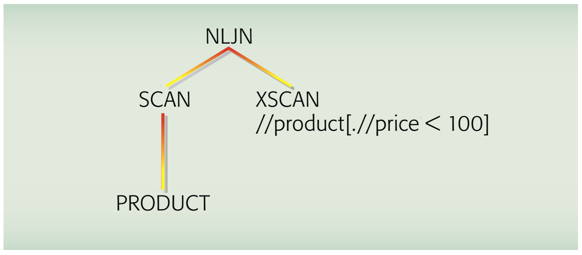

Figure 2
Simple XSCAN on PRODUCT

Figure 3 shows a possible plan with an XISCAN operator that exploits the XML index on `PRODUCT.I_PRICE` to evaluate the index expression `//product//price < 100`, thereby quickly identifying only those documents having products with prices less than 100. For each node in the XML index that satisfies the index expression, XISCAN returns the RID of the document containing that node. Since each document may contain many such nodes, duplicate RIDs may qualify. The `SORT (distinct)` operator then eliminates these duplicates, and the `FETCH` operator retrieves the `DESCRIPTION` documents on which XSCAN does its evaluation. The XSCAN operator then applies the path expression `//product[.//price < 100]/@id` on each of the documents found by the index. We represent this with a nested-loop join (NLJN) operator, in which each RID is passed from the outer (left) stream to the XSCAN on the inner (right) stream.

Currently, an XSCAN is always necessary after an index access to eliminate any false positives due to

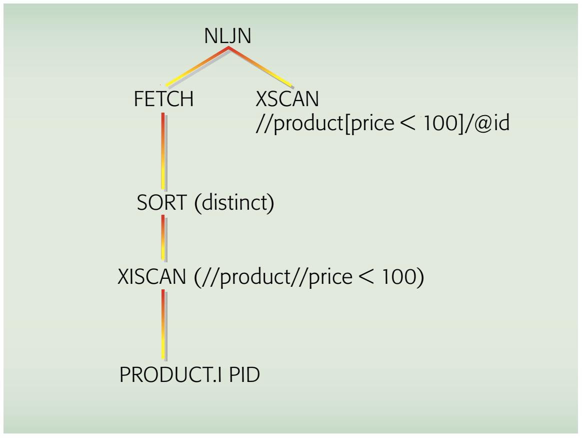

Figure 3
Simple XISCAN on index I.PRICE

type casting and value rounding or truncation done by the index. It also extracts the final results, which are often different from the nodes filtered by the index expression; for example, in the plan of Figure 3, XISCAN finds qualifying price elements, and the XSCAN operator produces the resulting @id nodes.

In Examples 4 and 5, the XSCANs operate on `DESCRIPTION` documents from the base table `PRODUCT`. In general, XSCANs can operate on any XML data instances, including intermediate results. For complex queries containing multiple path expressions, it is common to have a query execution plan containing several XSCAN operators, with the output of one XSCAN operator being the input of another; however, for compatibility with existing infrastructure, these are represented in plans as a sequence of nested-loop joins, each having a different XSCAN operator as the inner table (right branch) of the join.

Generation of index ANDing plans

In addition to the XML index plans described in the section "The XISCAN operator," DB2 XML generates three types of access plans that use multiple indexes in concert: index ANDing, index ORing, and XANDOR (XML index ANDing and ORing). Index ANDing is a technique enabling the use of multiple indexes to answer conjunctive predicates and is described for relational database systems in Reference 36, as shown in Example 6.

Example 6

```
SELECT P.*
FROM PRODUCT P
WHERE P.PRICE > 100 AND
      P.CATEGORY = 'Tools'
```

If the `Product` table had indexes for the price and category columns, an index ANDing plan would be generated that intersects the RIDs which qualify from the index access on price with those resulting from the index access on category. The ANDing plan may be superior in both I/O and CPU cost to a plan that uses only a single index.[36] Analogously, we extended the access rules to construct index ANDing plans for conjunctive XML predicates that match indexes. The comparable XQuery is given in Example 7. If indexes are defined for `//product//price` and `//product/category`, the DB2 XML optimizer constructs an XML index ANDing plan that intersects the results of the XML index scan (XISCAN) on the price index with those of the XISCAN on the category index.

Example 7

```
for $i in
db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')
/inventory/product[.//price < 100 and
./category = "Tools"]
return $i;
```

Relational index ANDing intersects the RIDs of each of the input index scans by using a Bloom filter scheme that hashes the RIDs. We have extended this mechanism in DB2 XML to compute the intersections of results at the lowest common ancestor of the nodes returned by the index. In Example 7, a node-based intersection is performed at the product node level, assuring that an `inventory/product` node (i.e., a product node under a root inventory node) qualifies only if both its price and category descendants meet the criteria.

Generation of index ORing plans

Index ORing is a technique enabling the use of multiple indexes to answer disjunctive predicates and is described for relational database systems in Reference 36. Traditionally, when all disjunctive predicates of an expression can be evaluated by using indexes, DB2 constructs an index ORing plan that unions the resulting RIDs from multiple index scans with an OR operator. DB2 XML does not alter the basic behavior of index ORing, but the conditions under which an ORing plan is considered must be relaxed.

Due to the expected high cost of evaluating XML expressions within XSCAN, we aggressively produce plans that use indexes to filter documents. In DB2 XML, eligibility for ORing is extended to include sets of indexes that are ensured to produce a superset of the required results. (An index is *eligible* for a query if it contains a superset of the results required for the query and all information required to execute the index scan is available.) This can be done correctly due to the mandatory reapplication of index expressions with XSCAN, described in the section "The XISCAN operator." These types of index combinations are likely to be quite useful in XML query processing.

XPath expressions often have a mix of predicates and next steps. The impact of such expressions on the eligibility of an ORing plan is illustrated by the following example:

Example 8

```
for $i in
db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')
/inventory/product[.//price < 100 or
./available[@validated="true"]/starting
= '2009-10-04']
return $i;
```

In this example, suppose that indexes are defined on `//product//price` (index 1), `//product/available/@validate` (index 2), and `//starting` (index 3). Although it appears that an index ORing plan could be considered for this example, the path expression from the child node `./available` poses some complications. This step can be thought of as an implicit conjunction between the predicate on `validated` and the next step, `starting`. The nodes returned should be only those that are `available` and have both a proper `validated` attribute and a `starting` child matching the predicate.

Traditional index ORing would not consider this expression eligible, because a union of all results from our three indexes would yield a superset of the correct answers (e.g., some `./available[@validated="true"]` nodes that lack a starting child).

In the case of Example 8, the DB2 XML optimizer would generate two alternative plans that constitute the union of the results of index 1 with either index 2 or index 3, but not both. Including all three indexes in the union would produce no additional correct results and could produce further incorrect results.

Generation of XANDOR plans

Although index ANDing and ORing provide powerful ways to combine indexes, they are inherently limited in how they interact with each other when represented separately. By combining all the index ANDing and ORing steps into a single operator, the DB2 XML runtime for the XANDOR operator is able to dynamically skip the processing of some of its inputs, based upon the node identifier with the highest ID value from any input. Although the plans with a single XANDOR operator look simpler and are easier to construct than the corresponding nesting of index ANDing and ORing operators, it is nevertheless necessary to extend the plan generation rules to produce XANDOR plans in addition to the ANDing and ORing plans. Which of these alternative

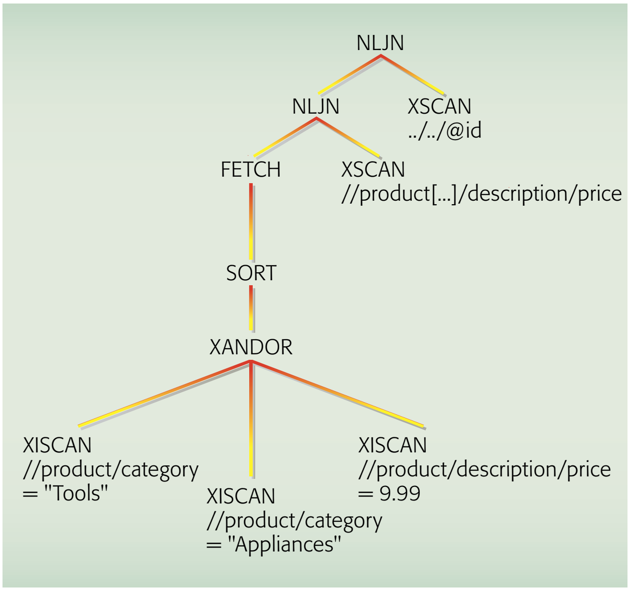
NLJN
NLJN
XSCAN ../../@id
FETCH
XSCAN //product[...]/description/price
SORT
XANDOR
XISCAN //product/category = "Tools"
XISCAN //product/category = "Appliances"
XISCAN //product/description/price = 9.99

Figure 4
XANDOR using indexes I.CATEGORY and I.PRICE

plans is actually used at runtime is decided by their estimated costs. As with other XML index plans, it is still necessary to add a `SORT` to remove duplicate documents and a `FETCH` to retrieve the needed columns from the table and then perform an NLJN with an XSCAN that reapplies the XPath expression and extracts the nodes. For example, the query of Example 9 could exploit the XML indexes on `I_CATEGORY` and `I_PRICE` in an XANDOR plan such as that shown in ***Figure 4***.

Example 9

```
for $i in
db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')
//product[category="Tools" or category="Appliances"]
./description/price
where $i = 9.99
return $i/../../@id;
```

Each XISCAN operator evaluates one of three index expressions: `//product/category="Tools"`, `//product/category="Appliances"`, and `//product/description/price=9.99`, returning RIDs of `DESCRIPTION` documents containing such expressions. The single XANDOR operator unions the RIDs returned by the first two XISCANs and then intersects them with the RIDs returned by the third XISCAN. As described earlier for plans using only a single XML index, the `SORT` operator is required after the XANDOR to remove duplicate RIDs, followed by the FETCH operator to retrieve the documents that survive the filtering by the three index expressions, and finally, followed by the XSCAN to reevaluate the expression //product[...]/@id on the filtered documents and extract the required nodes.

Properties, pruning, and enumeration of plans

DB2 associates *properties* with each subplan to track the cumulative work that the subplan has accomplished. Examples of relational properties include the tables accessed and predicates applied. A plan having properties identical to another plan but greater estimated cost may be pruned to limit the number of candidate plans.

Similarly, in DB2 XML we needed to track the work done by XPath expressions in various operators of a plan. Internally representing XPath expressions as a table function enabled the reuse of existing properties that track which relational tables have been accessed thus far. No additional property was required for tracking XPath expressions. Another even bigger benefit of this representation was that DB2's mechanism for enumerating different orders for joins would generate various orders for intermingling XPath expressions and table accesses, without major changes to the fundamental join enumeration algorithm or the conditions upon semantically legal orders (e.g., any expression, whether relational or XPath, clearly cannot reference a column, including an XML column, until the table containing that column has been referenced). Our decision to represent XPath expressions as a table function abstraction therefore saved a significant amount of implementation effort.

CARDINALITY AND COST ESTIMATION

A relational optimizer uses cost estimation to choose the least expensive of all the alternative QEPs produced by the plan generation algorithm. The operator costs are estimated based on cardinality, configuration parameters, and machine configurations, in addition to sophisticated modeling of the environment and execution-time processing algorithms. The *cardinality* of each operator in the QEP is an estimate of the number of results the operator will produce during plan execution. Cardinality greatly affects operator cost and is notoriously hard to estimate accurately. Precise data distribution statistics and sophisticated algorithms for processing these statistics are needed to produce accurate cardinality estimates. In DB2 XML, cardinality and cost estimation are implemented by extending and adapting the current DB2-optimizer cardinality and cost-estimation infrastructure. This allows us to take advantage of the current infrastructure to support relational, XML, and mixed workloads.

The modularized architecture of the DB2 UDB optimizer allows us to localize the changes to three general areas. First, we generalize predicate selectivity estimation to support XPath predicates and navigation expressions. In addition to selectivity, we compute fanout, the number of outputs produced for a single input. Second, cardinality estimation is extended to handle the new XML operators XSCAN, XISCAN, and XANDOR. Finally, new costing algorithms have been designed for the new operators, and some of the existing costing modules have been modified in order to deal with the new XML data type flowing through the system. In the remainder of this section, we describe each of these steps. We discuss the challenges and solutions involved in their adoption in DB2 XML.

Selectivity and fanout of XPath expressions

In traditional relational optimizers, the cardinality of predicate-applying operators such as SCAN is computed based on predicate *selectivity*. The selectivities are computed before the construction of alternative plans, using data distribution statistics, because the selectivity depends only on the predicate semantics and not on the operator in which the predicate is applied. XPath expressions in a query may act as predicates because they filter out input rows for which no results are produced. However, these expressions do more than simple predicates because they produce new result rows. To estimate the cardinality of XSCANs, which apply XPath expressions, we introduce the concept of XPath expression fanout. The *fanout* of an XPath expression is defined as the average number of result XML items produced per input (context) XML item.

The XPath expression /customer[name="Acme"]/order in a TPC-H schema would be roughly equivalent to the following SQL query:

```
SELECT *
FROM ORDERS O, CUSTOMER C
WHERE C.CUSTKEY = O.CUSTKEY AND
C.NAME = 'Acme'
```

The cardinality of this SQL query can be computed as a product of the customer table cardinality, the selectivity of the "Acme" local predicate, and a certain *expansion factor*, which captures the number of orders of a single customer. This expansion factor can be thought of as the join predicate selectivity, with two notable exceptions:

1. Its value may (or may not) be greater than one (e.g., there may be more orders than customers.)
2. Join predicates extract new values (e.g., orders of a customer), instead of *selecting* values from the set, as done by local predicates.

In XPath expressions, the line between local and join predicates is blurred because navigations and predicates can be arbitrarily nested within each other. Thus, we combine the selectivity and the expansion factor into a single variable, the fanout.

The following query finds the names of products with a price of less than 10 American dollars. (The example assumes that a product can have different names and prices in different markets.)

Example 10

```
for $i in db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')
//product[10 > .//price[@currency="USD"]]
let $j = $i//name
return <result> {$i/@id} {$j} </result>;
```

The same XPath expression `//product[10 > .//price[@currency="USD"]]` can both increase and decrease the cardinality. A single document may contain many product elements, which will increase the XSCAN cardinality. But it also contains predicates and the expression `[@currency="USD"]` that reduce the XSCAN cardinality.

Suppose that data distribution statistics tell us that this collection contains a total of 1000 documents, which contain 200 product elements with a qualifying "price" descendant. These 200 `products` have among them 500 "name" descendants, and each `product` has an `id` attribute. The fanouts of the three XPath expressions in the query are shown in *Table 1*.

Note that very elaborate statistics are required to find the exact fanout of the XPath expression. In the above example, ideally we need the count of name descendants of product elements that satisfy some predicate. We may also need information on statistical correlation for any combination of any number of arbitrary XPath predicates. Collecting, storing, and accessing such large amounts of statistical information is not practical. DB2 XML collects only the most essential statistics, as discussed in the section "Statistics collection." In order to estimate the fanout, we make the following two uniformity assumptions about the data distribution:

Table 1 Fanouts of XPath expressions

XPath expression	Fanout computation
//product[. . .]	$200/1000 = 0.2$
$i//name	$500/200 = 2.5$
$i/@id	1

1. *Fanout uniformity*—For any two XPath steps, *A* and *B*, where *A* is an ancestor of *B* in the XPath expression tree, XML data items that bind to *B* are uniformly distributed among XML fragments rooted at elements that bind to *A*. For example, for an XPath expression `//X/Y`, any two *X* results will have the same number of *Y* children.

2. *Predicate uniformity*—For any XPath step with a predicate (i.e., `/axisX::testX[Y]`), XML data items that bind to *X* and satisfy *Y* are uniformly distributed among all items that bind to *X*.

Selectivity and fanout of index expressions

The selectivity of an index expression (IE) is the fraction of documents in the collection that will be returned by an XISCAN with this IE. An XISCAN operator returns both XML nodes and the documents in which they occur. Currently, DB2 XML only uses XML indexes to prefilter the documents on which the XSCAN is applied. Thus, each XISCAN is followed by the `SORT` operator, which eliminates duplicate document IDs, as shown in Figure 3.

For indexable XML predicates, we compute both the selectivity and the fanout. IE fanout is used to estimate the number of XML items returned by the index access, which, in turn, is used to estimate the cost of the XISCAN operator and the subsequent `SORT`. The IE selectivity is needed to estimate the cardinality of the subsequent `SORT`. To facilitate accurate estimation of IE selectivity and fanout, DB2 XML maintains both document-count and node-count statistics for frequent path-value pairs and all paths in an XML column, as detailed in the section "Statistics collection." The document counts are

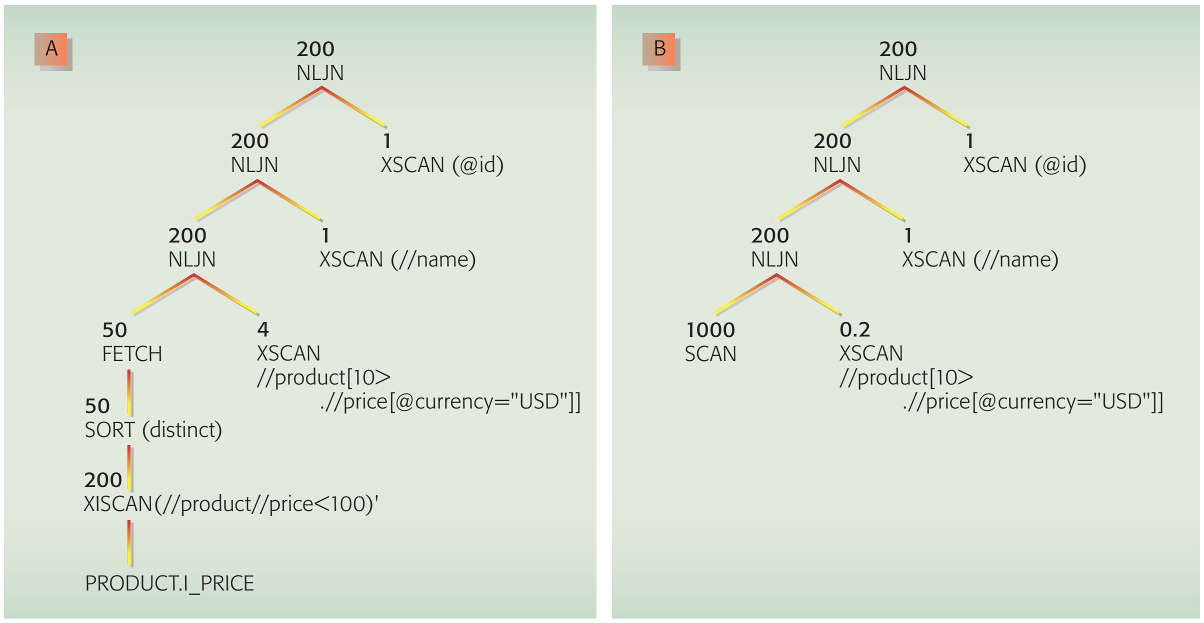

Figure 5
XISCAN and SCAN plan alternatives: (A) plan using an index to find specific documents; (B) plan scanning all documents

used to compute the IE selectivity, and the node counts are used to estimate IE fanout.

Cardinality estimation

For each QEP operator, DB2 XML needs to track the expected number of produced rows, as well as the number of XML items in each sequence that the row contains. Because the sequences may need to be sorted or filtered, their size is also important in cost estimation. To address this challenge, we estimate and record internally the average *sequence size* for each column in the query, including derived results. The sequence size of an XML column in a query plan is defined as the average number of XML items per XML sequence flowing through this column. The sequence size of a column produced by a FOR extraction is equal to 1. The sequence size of a column produced by a LET extraction can be any value greater than or equal to zero.

For example, in the XQuery of Example 10, every sequence size is 1, except for the column that corresponds to `$j`. Its sequence size is 2.5, according to the fanout estimation of Table 1. The cardinality of each operator is computed by a "bottom-up" traversal of each plan tree and depends on the operator and its input. Note that the cardinality of the inner row of the join is always estimated per outer row.

As with relational index scans, the cardinality of the XSCAN operator is estimated to be a product of the fanout of its XPath expression, the selectivity of all predicates applied by XSCAN, and the sequence size of the input (context) column. The sequence size term is needed in this computation in case the input to XSCAN is a sequence of XML items created by an earlier LET extraction. The cardinality of the XISCAN operator is the product of the base table's cardinality and the selectivity of the index expression. Each XISCAN operator has to be followed by a join with an XSCAN operator that finishes the XPath expression computation (see the section "Generation of XML index plans"). A plan that joins an XISCAN operator and the corresponding XSCAN operator computes the same expression as a plan that performs an NLJN of a table scan and the same XSCAN operator. The cardinalities of these two plans have to be identical because they compute the same result.

Figure 5 shows two possible plans for the query in Example 10. Plan A uses an index on price elements to find only those documents having a product price less than 100, and plan B scans all documents. The estimated cardinality of each operator is shown in boldface, next to the operators. If all 200 resulting "product" elements are found in 50 documents, the index expression selectivity is 50/1000 = 0.05. Note that in plan B, the cardinality of the first XSCAN operator is 0.2, which is the fanout of the XPath expression to which this XSCAN operator applies, as computed in Table 1. This means that, for each document that the XSCAN operator takes as an input, it will produce, on average, 0.2 output rows. However, in plan A, the cardinality of the same XSCAN operator is different, because the input documents to the XSCAN operator have been prefiltered by XISCAN. For each document output by XISCAN, XSCAN will produce an average of four result rows because 50 documents returned by XISCAN contain 200 product elements that XSCAN is looking for.

To ensure consistent cardinality estimates, we adjust the cardinality of each XSCAN operator that applies XPath expressions associated with index expressions applied earlier in the plan. The cardinality of such an XSCAN operator is divided by the combined selectivity of all these IEs, to compensate for the prefiltering already done by the index accesses. Without this adjustment, an XISCAN operator and a SCAN plan having the same result would have different cardinality estimates.

XANDOR cardinality

When estimating the cardinality of index ANDing and ORing operators, we have to take special care to account for statistical correlations between expression tree nodes that may be implicit in the tree structure. We estimate the combined selectivity of multiple index expressions by dividing the product of all IE selectivities by the selectivity of all lowest common ancestor (LCA) steps in the XPath expression tree. Because the existence of a node implies the existence of its parent, selectivities of two branching paths are not independent; both of them include the selectivity of the LCA.

Example 11

Consider the query `db2-fn:xmlcolumn('T.DOC') /a[b]/c` where `/a` occurs in 100 of 1000 documents of table T, `/a/b` occurs in 50, and `/a/c` occurs in 10. Given two index expressions on `/a/b` and `/a/c`, with selectivities S(`/a/b`) = 50/1000 = 0.05 and S(`/a/c`) = 10/1000 = 0.01, the combined selectivity of the two IEs is S(`/a[b]/c`) = S(`/a/b`) * S(`/a/c`) / S(`/a`). The last term avoids double-counting the selectivity of S(`/a`), which is implicitly included in both S(`/a/b`) and S(`/a/c`). In this case, the index ANDing cardinality is: Card(T) * S(`/a[b]/c`) = 5.

Cost estimation

The DB2 UDB cost estimation infrastructure is modularized by operators. For DB2 XML, we added cost estimation for the three new operators and modified the cost models of any existing operators, such as SORT and FILTER, that were extended to support the XML datatype.

Cost modeling for SQL/XML and XQuery on XML data is much more complex than for relational data, due to the semantic differences and the complexity of the operators, as well as the versatility and complexity of XML data; for example, the hierarchical data model, its schema flexibility, predicates on a mix of structure and values, and documents with heterogeneous structure and characteristics. DB2 XML was specifically designed to address the schema evolution scenario, in which multiple, possibly conflicting schemas coexist in a document collection. This schema heterogeneity complicates the use of schema information for cost estimation. We also support querying documents that do not have schemas, while exploiting schema information when it is available.

To illustrate the difficulty of cost estimation, consider an XSCAN operator that evaluates the path expression `/*/product[.//price < 100]/@id` on a collection of XML documents without a fixed schema. The XSCAN algorithm will need to find the IDs of all second-level product elements in all XML documents, such that there exists a descendant price element whose value is less than 100. The XSCAN navigation algorithm will read once through each document, skipping all second-level elements that are not named "product." For each product element, the scan of its descendants will be terminated as soon as the first qualifying price element is found.

The XSCAN algorithm makes its navigation decisions based on the data that it is processing. Thus the navigation pattern is different from one XML document in the collection to the next, and even within fragments of the same document. To accurately model the cost of the navigation, we need to know how many product elements each document contains in its second level, how many of them contain a price descendant with a value less than 100, and how many pages of other descendants will be read before the first such price is found. It is unreasonable to expect all these values to be available from some auxiliary data structure, such as data distribution statistics. The size of such a

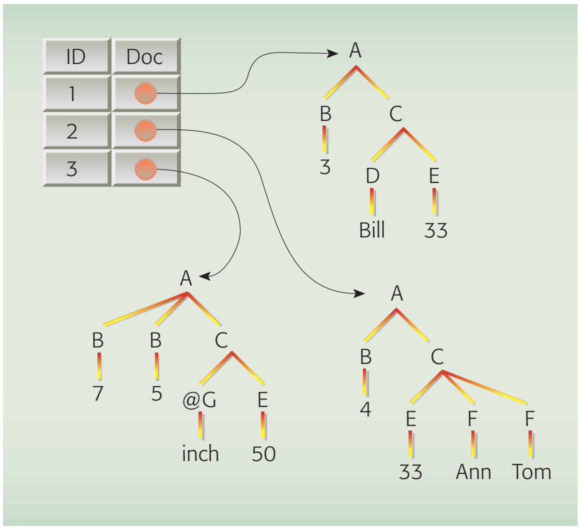

Figure 6
Example of table with one XML column

structure would be comparable to the size of the database itself, and the estimation would probably take longer than the query execution. Thus we have to trade accuracy of the estimation for its performance.

The problem of modeling XML operators is further compounded by the need for integration with the existing cost-based optimization for SQL queries. The power of the DB2 XML system lies in its ability to mix relational and XML predicates in a single query. Thus, the same query could, for example, be executed by using an XML index access (XISCAN) or a relational one (ISCAN). The cost models of the two operators have to be consistent in their level of detail and in the factors being considered in order to avoid any bias toward either of the operators in the model.

STATISTICS COLLECTION

In this section, we describe the type of XML statistics that DB2 XML collects and the challenges involved in the collection procedure.

Statistics collection in DB2 is performed by a utility called `runstats`, which collects two types of statistics for relational data: table statistics and index statistics, which are collected by scanning each row of each table and each index. After `runstats` scans the table and its indexes, the statistics are summarized and written to the system catalog tables. For DB2 XML, we extend the `runstats` utility to collect XML statistics when it encounters columns of type XML during the table scan. In the following, we focus our discussion on table statistics for XML columns.

XML column statistics

For relational data, the default column statistics consist of aggregate information that roughly characterizes the distribution of the values that occur in that column. These statistics are used for computing selectivities of predicates. For XML columns, cardinality estimation must include computing fanout, which requires occurrence counts for each linear rooted path in that column. A linear rooted path is one that has no predicates and starts at the root of the document.

We distinguish two types of linear paths, each having distinct statistics: *simple paths* are paths that end in element nodes only, whereas *path-value pairs* could end either in an attribute value or a text value. We also distinguish between two types of occurrence counts: node counts and document counts. For each XML column, we collect node-count and document-count statistics for each distinct path and path-value pair. For the XML column in the table shown in ***Figure 6***, the node counts and document counts of all the paths and path-value pairs in the column are shown in ***Table 2***.

Because XML documents with different structural characteristics could be stored in different rows of the same XML column, it is possible that the number of distinct paths in an XML column could be very large and infeasible to store. We therefore adopt the same strategy used in relational distribution statistics: keep counts for a specified number of the most frequently occurring values and one "bucket" that aggregates counts for all the values for computing averages and interpolation. For each of the most frequent values, we collect the (simple) path node counts, path document counts, path-value node counts, and path-value document counts.

We also collect several aggregates for each XML column, to capture statistics on paths that do not occur sufficiently frequently; namely, the number of distinct simple paths, the sum of node counts over each distinct simple path, and the sum of document counts over each distinct simple path. For path-value pairs, we have one path-value-bucket data structure for each distinct path that leads to an attribute or text value within a column, with

Table 2 Node and document counts for paths and path-value pairs

Path	Node count	Document count
/A	3	3
/A/B	4	3
/A/C	3	3
/A/C/D	1	1
/A/C/E	3	3
/A/C/F	2	1
/A/C/@G	1	1
/A/B=3	1	1
/A/B=4	1	1
/A/B=5	1	1
/A/B=7	1	1
/A/C/D=Bill	1	1
/A/C/E=33	2	2
/A/C/E=50	1	1
/A/C/@G=inch	1	1
/A/C/F=Ann	1	1
/A/C/F=Tom	1	1

Table 3 XML statistics

Frequent values	
Path node count	*(/A/B, 4), (/A, 3)*
Path document count	*(/A/B, 3), (/A, 3)*
Path-value node count	*(/A/C/E=33,2)*
Path-value document count	*(/A/C/E=33,2)*
Path bucket	
Distinct paths	7
Sum of node counts	17
Sum of document counts	15
Path value bucket for /A/B	
Distinct paths	4
2nd highest value	5
2nd lowest value	4
Sum of node counts	4
Sum of document counts	4
Path value bucket for /A/C/D	
. . .	
Path value bucket for /A/C/E	
. . .	
Path value bucket for /A/C/F	
. . .	
Path value bucket for /A/C/@G	
. . .	

statistics such as the number of distinct values. *Table 3* summarizes all the statistics associated with an XML column for an example with the number of paths and the number of path-value pairs, namely, $k_p = k_{pv} = 2$ and the table shown in Figure 6. Conceptually, these statistics are an aggregation and summarization of the counts in Table 2.

Implementation

Even though the XML statistics that we collect are quite basic, extending the relational `runstats` infrastructure to collect XML statistics in a single scan of the table significantly complicates the estimation of its memory requirements. For relational data, it is easy to estimate the memory required because the number of columns in the table are known beforehand from catalog information, and each column on which statistics are to be collected is of an atomic type. For an XML column, however, each row is an XML document with many nodes, many distinct paths, and many distinct path-value pairs.

For example, in order to allocate enough path-value buckets when collecting path-value-bucket statistics, we need to know the number of distinct paths that lead to a specific attribute or text value. The problem is that the number of such paths is not maintained anywhere, nor is it easily computable without an additional scan of the table. Our solution is to perform an index scan on the column path index to count the number of paths that lead to attribute or text values. In DB2 XML, a column path index is automatically created and maintained for each XML column. This index stores all the paths that occur in the XML column and their corresponding numeric path identifiers. Collecting document-count statistics requires remembering what paths have been seen before in that XML document; we use a fixed-size Bloom filter[37] to remember the distinct paths in a document, an approximate solution that allows us to cap memory utilization.

The space required for frequent value statistics similarly requires memory that grows with the number of distinct simple (or path-value pair) paths, which is neither known beforehand nor scalable. *Reservoir sampling*[38] provides a solution that yields good estimates in a fixed amount of memory and a single access of the data. Logically, reservoir sampling is equivalent to sampling using a uniform distribution, but algorithmically, reservoir sampling is a one-pass algorithm that does not require knowledge beforehand of the size of the data.

FUTURE WORK

Query processing on XML data is still in its infancy. This paper describes the first wave of extensions to the DB2 optimizer in support of XQuery and SQL/XML. This section outlines some anticipated future work.

XML index exploitation

A scan of an XML index returns references to individual nodes that satisfy an index expression, but today, DB2 XML does not fully exploit this granularity. We currently use the result of an XML index scan to qualify entire documents that satisfy an XML index expression. Subsequent navigation from the root of each qualifying document determines the exact set of nodes that satisfies the full XPath expression. This strategy of filtering documents works well when a few small documents are qualified by the predicate, as navigation of a few small documents is fairly efficient. If our presumption about the size of documents fails to hold in practice, we will need to add evaluation strategies that navigate from each of the individual nodes qualified by the index. Such strategies would allow us to use XML indexes to pinpoint relevant sections of large documents.

Deferred XPath evaluation

One possible enhancement of our current join enumeration algorithm would be to consider additional plans that would defer the XSCANs used to evaluate a full XPath expression following an XML index access. Currently, the XSCAN is always (nested-loop) joined immediately with each XISCAN plan, as shown in Figure 3. Deferring the XSCAN until after performing other joins might reduce the number of documents that have to be navigated, but would significantly increase the number of alternative plans.

Index ANDing heuristics

There can be many alternative index ANDing plans. Heuristics are often employed to reduce the number of alternatives considered. For example, an alternative that orders the inputs by decreasing selectivity so that more filtering is done early might be considered along with another that orders the inputs by decreasing cost so that cheap accesses are done early. It is possible, however, that the simple heuristics typically used today would fail to yield the most efficient XML index ANDing plan, as they do not take into account the effect of intersecting two indexes whose lowest common ancestor is high in the XML hierarchy. One of our future investigations will be to extend the current index ANDing heuristics to take this XML-specific dimension of the problem into account.

Cardinality estimation and statistics

We anticipate the need to extend our statistics and cardinality estimation model to take into account structural relationships between predicates. For example, our linear path statistics suffice to estimate the number of matches for each path expression `/product/price` and `/product/@id`, but give no information on how many of the matches are related through a common `/product` node for the path expression `/product[price]/@id`. Moreover, in order to accurately estimate the cardinalities of path expressions involving numeric or timestamp values such as `/product[price < 100.00]`, we will need to collect statistics that are data type specific. Currently, our XML statistics collection routines treat all values as strings, in the absence of schema information, because we do not know how a value should be treated until it is referenced in a query.

Cost estimation

The cost models for our new XML operators were developed using the static modeling approach typically used in the development of cost models for relational operators. This approach involves having an optimizer cost-model expert perform extensive analysis to determine a set of formulas that accurately models the algorithmic behavior of an operator, given any possible input or system configuration.

This human-intensive process must be repeated whenever the implementation of the operator is changed or extended. The behavior of XML operators is inherently more complex than relational operators due to the hierarchical nature of the XML data on which they operate; consequently, using a static modeling approach to develop and maintain a cost model for these operators is a daunting task. In the future, we hope to use more automated techniques to develop operator cost models. For example, the COMET prototype[39] proposes the use of statistical learning techniques to automatically build a cost model for the equivalent of the XSCAN operator.

Order optimization

In XQuery, the fundamental construct is an ordered sequence. The semantics of XQuery usually require the output to preserve both the "bind order," which is essentially the order in which FORs and LETs are nested, as well as the original "document order," which is the order of nodes within documents, and even between documents. We plan to extend the optimizer's order optimization architecture[40] so that it can effectively deal with these implicit order requirements in addition to the usual value-based orderings that it currently supports.

CONCLUSION

Overall, reusing the relational infrastructure of DB2 UDB has facilitated robust and scalable support for XQuery and SQL/XML far faster than starting fresh, even though the heterogeneous and semi-procedural aspects of XQuery required some challenging extensions to the plan generation, cardinality and costing, and statistics components. The reuse was made possible by (1) the introduction of an XML column type that can hold variable-length sequences of XML items, (2) uniform internal modeling of SQL, XQuery, and SQL/XML queries, and (3) representing XPath expressions as table functions. The plan generation extensions included new QEP operators and plan enumeration rules. We extended the cardinality and selectivity concepts to support heterogeneous XML sequences and XPath expressions that combine predicates and navigation. Furthermore, new data distribution statistics were collected to capture the hierarchical nature of the XML data.

We are convinced that XQuery opens a new and very challenging chapter in query optimization for declarative database languages, one that—like SQL—will mature over the years to come.

ACKNOWLEDGMENT

We would like to thank the entire DB2 XML team for their support of this work.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Linus Torvalds, The Open Group, or Microsoft Corporation in the United States, other countries, or both.

CITED REFERENCES

1. *Information Technology-Database Languages—SQL-Part 14: XML-Related Specifications (SQL/XML), ISO/IEC 9075-14:2003/Cor 1:2005*, International Organization for Standardization (2005), http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=41905&ICS1=35&ICS2=60&ICS3=&scopelist=.
2. *XQuery 1.0: An XML Query Language*, S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, Editors, W3C Candidate Recommendation, (November 2005), http://www.w3.org/TR/xquery.
3. *XML Path Language (XPath) Version 2.0*, A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon, Editors, W3C Candidate Recommendation, (November 2005), http://www.w3.org/TR/xpath20.
4. K. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. Truong, B. Van Der Linden, B. Vickery, and C. Zhang, "System RX: One Part Relational, One Part XML," *Proceedings of the 2005 ACM International Conference on the Management of Data (SIGMOD 2005)* ACM Press, New York (2005), pp. 347–358.
5. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, Editors, W3C Candidate Recommendation, (November 2005), http://www.w3.org/TR/xpath-datamodel.
6. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, "Lore: A Database Management System for Semi-structured Data," *ACM SIGMOD Record* **26**, No. 3, 54–66 (1997).
7. J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnagaa, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen, "The Niagara Internet Query System," *IEEE Data Engineering Bulletin* **24**, No. 2, 27–33 (2001).
8. 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. 4, 274–291 (2002).

9. T. Fiebig, S. Helmer, C.-C. Kanne, J. Mildenberger, G. Moerkotte, R. Schiele, and T. Westmann, "Anatomy of a Native XML Base Management System," *VLDB Journal* **11**, No. 4, 292–314 (2002).

10. D. Barbosa, A. Barta, A. Mendelzon, G. Mihaila, F. Rizzolo, and P. Rodriguez-Gianolli, "ToX—the Toronto XML Engine," University of Toronto, Department of Computer Science (2001), http://www.cs.toronto.edu/tox/papers/wiiw.pdf.

11. M. Rys, "XML and Relational Database Management Systems: Inside Microsoft SQL Server 2005," *Proceedings of the ACM SIGMOD Conference on the Management of Data*, ACM Press, New York (2005), pp. 958–962, http://portal.acm.org/citation.cfm?doid=1066301.

12. R. Murthy, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A.-T. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, and V. Krishnamurthy, "Toward an Enterprise XML Architecture," *Proceedings of the ACM SIGMOD Conference on the Management of Data,* ACM Press, New York (2005), pp. 953–957.

13. V. Josifovski, M. Fontoura, and A. Barta, "Querying XML Streams," *VLDB Journal* **14**, No. 2, 197–210 (2005).

14. J. McHugh and J. Widom, "Query Optimization for XML," *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB 1999),* Morgan Kaufmann Publishers, San Francisco, CA (1999), pp. 315–326.

15. A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton, "Estimating the Selectivity of XML Path Expressions for Internet Scale Applications," *Proceedings of the International Conference on Very Large Data Bases (VLDB '01),* Morgan Kaufmann Publishers, San Francisco, CA (2001), pp. 591–600.

16. L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr, "XPath-Learner: An On-line Self-tuning Markov Histogram for XML Path Selectivity Estimation," *Proceedings of the International Conference on Very Large Data Bases (VLDB '02),* Morgan Kaufmann Publishers, San Francisco, CA (2002), pp. 442–453.

17. Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava, "Counting Twig Matches in a Tree," *Proceedings of the 17th International Conference on Data Engineering (ICDE 2001),* IEEE Computer Society, Los Alamitos, CA (2001), pp. 595–604.

18. N. Polyzotis and M. N. Garofalakis, "Statistical Synopses for Graph-Structured XML Databases," *Proceedings of the ACM SIGMOD Conference on the Management of Data,* ACM Press, New York (2002), pp. 358–369.

19. N. Polyzotis and M. N. Garofalakis, "Structure and Value Synopses for XML Data Graphs," *Proceedings of the 28th International Conference on Very Large Data Bases,* Morgan Kaufmann Publishers, San Francisco, CA (2002), pp. 466–477.

20. N. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis, "Approximate XML Query Answers," *Proceedings of the ACM SIGMOD Conference on the Management of Data,* ACM Press, New York (2004), pp. 263–274.

21. J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simeon, "StatiX: Making XML Count," *Proceedings of the 2002 ACM International Conference on the Management of Data (SIGMOD 2002),* ACM Press, New York (2002), pp. 181–191.

22. Y. Wu, J. M. Patel, and H. V. Jagadish, "Estimating Answer Sizes for XML Queries," *Proceedings of the 8th International Conference on Extending Database Technology (EDBT 2002),* LNCS 2287, Springer-Verlag, New York (2002), pp. 590–608.

23. W. Wang, H. Jiang, H. Lu, and J. X. Yu, "Bloom Histogram: Path Selectivity Estimation for XML Data with Updates," *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004),* Morgan Kaufmann Publishers, San Francisco, CA (2004), pp. 240–251.

24. L. Lim, M. Wang, and J. S. Vitter, "CXHist: An On-line Classification-Based Histogram for XML String Selectivity Estimation," *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005),* ACM Press, New York (2005), pp. 1187–1198.

25. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD Conference on the Management of Data,* ACM Press, New York (1979), pp. 23–24.

26. G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger, and P. F. Wilms, "Query Processing in R*," in *Query Processing in Database Systems,*W. Kim, D. Reiner, and D. Batory, Editors, Springer-Verlag, New York (1985), pp. 31–47.

27. L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita, "Starburst Mid-flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering* **2**, No. 1, 143–160, (1990).

28. G. M. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives," *Proceedings of the ACM SIGMOD Conference on the Management of Data,* ACM Press, New York (1988), pp. 18–27.

29. H. Pirahesh, J. Hellerstein, and W. Hasan, "Extensible/Rule-based Query Rewrite Optimization in Starburst," *Proceedings of the ACM SIGMOD Conference on the Management of Data,* ACM Press, New York (1992), pp. 39–48.

30. P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang, "Query Optimization in the IBM DB2 Family," *IEEE Data Engineering Bulletin* **16**, No. 4, 4–18 (1993).

31. TPC-H, Transaction Processing Performance Council, http://www.tpc.org/tpch/.

32. K. Ono and G. M. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization," *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB 1990),* Morgan Kaufmann Publishers, San Francisco, CA (1990), pp. 314–325.

33. V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proceedings of the ACM SIGMOD Conference on the Management of Data,* ACM Press, New York (1996), pp. 294–305.

34. A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh, "A Framework for Using Materialized XPath Views in XML Query Processing," *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04),* Morgan Kaufmann Publishers, San Francisco, CA (2004), pp. 60–71.

35. N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," *Proceedings of the 2002 ACM International Conference on the Management of Data (SIGMOD 2002),* ACM Press, New York (2002), pp. 310–321.

36. C. Mohan, D. J. Haderle, Y. Wang, and J. M. Cheng, "Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques," *Proceedings of the International Conference on Extending Data Base Technology (EDBT '90)* LNCS 416, Springer-Verlag, New York (1990), pp. 29–43.

37. B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM* **13**, No. 7, 422–426 (1970).

38. J. S. Vitter, "Random Sampling with a Reservoir," *ACM Transactions on Mathematical Software* **11**, No. 1, 37–57 (1985).

39. N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang, "Statistical Learning Techniques for Costing XML Queries," *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, ACM Press, New York (2005), pp. 289–300.

40. D. Simmen, E. Shekita, and T. Malkemus, "Fundamental Techniques for Order Optimization," *Proceedings of the ACM SIGMOD Conference on the Management of Data*, ACM Press, New York (1996), pp. 625–628.

Accepted for publication January 12, 2006.
Published online May 18, 2006.

Andrey Balmin
IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (abalmin@us.ibm.com). Dr. Balmin is a research staff member in the Exploratory Database department at IBM's Almaden Research Center. He is a member of the cross-divisional team that extended DB2 Universal Database for Linux, UNIX, and Windows to provide native XML and XQuery support. Within this project, he is responsible for XML index eligibility and XML cardinality and cost estimation modules. His research interests include querying and management of semistructured data, as well as integration of structured and unstructured search technologies. He received a Ph.D. degree from the University of California at San Diego.

Tom Eliaz
IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (teliaz@us.ibm.com). Mr. Eliaz is a software engineer in IBM's Silicon Valley Lab. He is a member of the DB2 Universal Database for Linux, UNIX, and Windows query compiler team and is focused on extending the DB2 optimizer with native XML and XQuery support. He graduated from the University of Pennsylvania with a B.S. degree in computer science and has worked for IBM on the query optimizer since 2002.

John Hornibrook
IBM Software Group, 8200 Warden Ave., Markham Ontario L6G 1C7, Canada (jhornibr@ca.ibm.com). Mr. Hornibrook is the development manager for the optimizer, *runstats*, and *explain* components of DB2 Universal Database for Linux, UNIX, and Windows. He has been a DB2 developer for 14 years, working on various DB2 engine components, including Relational Data Services and the SQL Compiler.

Lipyeow Lim
IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (liplim@us.ibm.com). Dr. Lim is a research staff member at the Watson Research Center. He is part of the team that prototyped and developed the XML *runstats* component in DB2. His research interests lie in the field of data management, in particular, statistics collection and storage for database query optimization.

Guy M. Lohman
IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (lohman@almaden.ibm.com). Dr. Lohman is the manager of Advanced Optimization in the Advanced Database Solutions department at IBM's Almaden Research Center. He has over 23 years of experience in relational query optimization. He is the architect of the optimizer of the DB2 Universal Data Base (UDB) for Linux, UNIX, and Windows, and was responsible for its development in Versions 2 and 5, as well as the invention and prototyping of Visual Explain. During that period, Dr. Lohman also managed the overall effort to incorporate into the DB2 UDB product the Starburst compiler technology that was prototyped at the Almaden Research Center. More recently, he was a co-inventor and designer of the DB2 Index Advisor (now part of the Design Advisor) and co-founder of the DB2 Autonomic Computing project. In 2002, Dr. Lohman was elected to the IBM Academy of Technology. His current research interests involve query optimization, self-managing database systems, and problem determination.

David Simmen
IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (simmen@us.ibm.com). Mr. Simmen is a Senior Technical Staff Member in the Exploratory Database department at IBM's Almaden Research Center. He was formerly the technical lead of the DB2 Universal Database for Linux, UNIX, and Windows query optimization team and managed the Silicon Valley Laboratory branch of the team. He was a member of the cross-divisional team that developed the DB2 optimizer from a research prototype and spent a large portion of his IBM career leading efforts to extend that technology. His research interests include query processing and the integration of content, database, and text-processing systems. He joined IBM after receiving a B.S. degree in applied mathematics from Carnegie Mellon University in 1989.

Min Wang
IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (min@us.ibm.com). Dr. Wang received a Ph.D. degree in computer science from Duke University in 1999. She is currently a research staff member at the Watson Research Center in Hawthorne, New York. Her research interests include query optimization, approximate query processing, data mining, electronic commerce, and database security.

Chun Zhang
Cosmix Corporation, 444 Castro Street, Suite 700, Mountain View, California 94041 (yazhangchun@yahoo.com). Dr. Zhang is currently a member of the technical staff at Cosmix Corporation in Mountain View, California. The work described in this paper was done while she was a research staff member at IBM's Almaden Research Center and part of the DB2 XML optimizer team focusing on the design and development of the query optimizer. She has extensive experience in structured, semistructured, and unstructured data management. Her current work involves text-mining and search-engine technologies. She is also interested in storage, query processing, and optimization. Dr. Zhang received a Ph.D. degree from the University of Wisconsin at Madison. ■