in Database Technology, Proc. 8th Int. Conf. on Extending Database Technology, 2002, pp. 477–495.
8. XML Path Language (XPath) 2.0. W3C Recommendation. Available at: http://www.w3.org/TR/xpath20/
9. XQuery 1.0: An XML Query Language. W3C Recommendation. Available at: http://www.w3.org/TR/xquery/
10. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation. Available at: http://www.w3.org/TR/xpath-datamodel/
11. XQuery 1.0 and XPath 2.0 Full-Text 1.0. W3C Working Draft. Available at: http://www.w3.org/TR/xpath-full-text-10/
12. XQuery 1.0 and XPath 2.0 Full-Text 1.0 Requirements. W3C Working Draft. Available at: http://www.w3.org/TR/xpath-full-text-10-requirements/
13. XQuery 1.0 and XPath 2.0 Full-Text 1.0 Use Cases. W3C Working Draft. Available at: http://www.w3.org/TR/xpath-full-text-10-use-cases/

XQuery Interpreter

► XQuery Processors

XQuery Processors

Torsten Grust[1], H. V. Jagadish[2], Fatma Özcan[3], Cong Yu[4]
[1]University of Tübingen, Tübingen, Germany
[2]University of Michigan, Ann Arbor, MI, USA
[3]IBM Almaden Research Center, San Jose, CA, USA
[4]Yahoo! Research, New York, NY, USA

Synonyms

XML database system; XQuery compiler; XQuery interpreter

Definition

XQuery processors are systems for efficient storage and retrieval of XML data using XML queries written in the XQuery language. A typical XQuery processor includes the data model, which dictates the storage component; the query model, which defines how queries are processed; and the optimization modules, which leverage various algorithmic and indexing techniques to improve the performance of query processing.

Historical Background

The first W3C working draft of XQuery was published in early 2001 by a group of industrial experts. It is heavily influenced by several earlier XML query languages including Lorel, Quilt, XML-QL, and XQL. XQuery is a strongly-typed functional language, whose basic principals include simplicity, compositionality, closure, schema conformance, XPath compatibility, generality and completeness. Its type system is based on XML schema, and it contains XPath language as a subset. Over the years, several software vendors have developed products based on XQuery in varying degrees of conformance. A current list of XQuery implementations is maintained on the W3 XML Query working group's homepage (http://www.w3.org/XML/XQuery). There are three main approaches of XQuery processors. The first one is to leverage existing relational database systems as much as possible, and evaluate XQuery in a purely relational way by translating queries into SQL queries, evaluating them using SQL database engine, and reformatting the output tuples back into XML results. The second approach is to retain the native structure of the XML data both in the storage component and during the query evaluation process. One native XQuery processor is Michael Kay's Saxon, which provides one of the most complete and conforming implementations of the language available at the time of writing. Compared with the first approach, this native approach avoids the overhead of translating back and forth between XML and relational structures, but it also faces the significant challenge of designing new indexing and evaluation techniques. Finally, the third approach is a hybrid style that integrates native XML storage and XPath navigation techniques with existing relational techniques for query processing.

Foundations

Pathfinder: Purely Relational XQuery

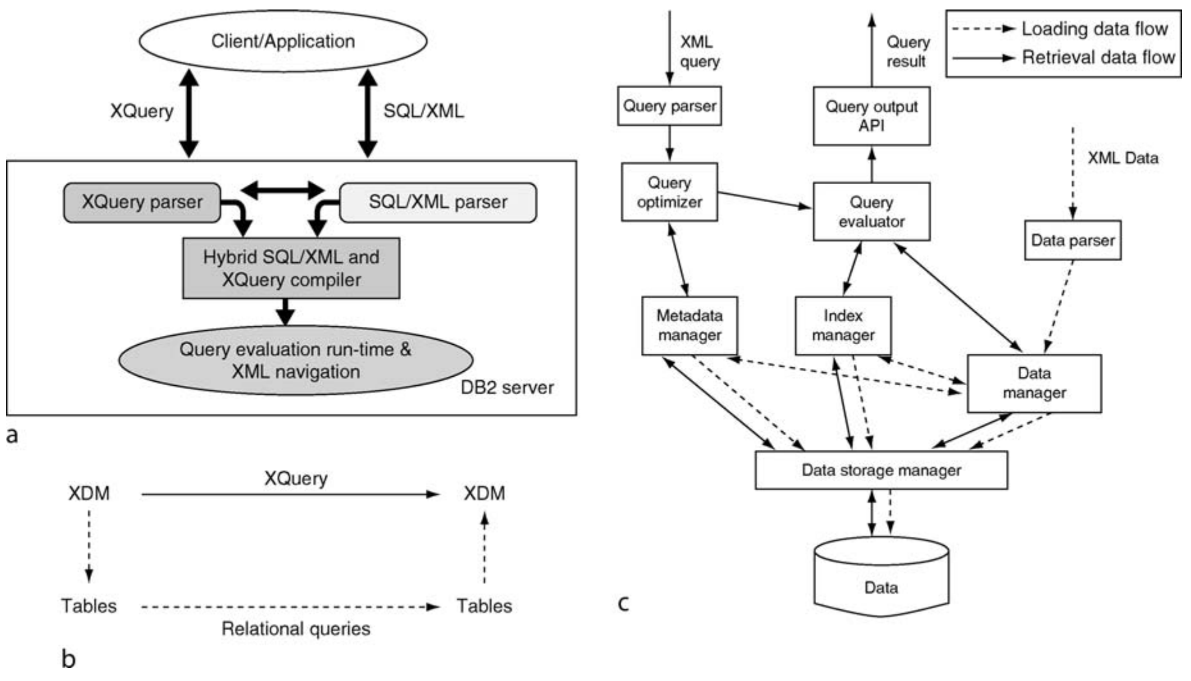
The *Pathfinder* XQuery compiler has been developed under the main hypothesis that the well-understood relational database kernels also make for efficient XQuery processors. Such a relational account of XQuery processing can indeed yield scalable XQuery implementations – provided that the system exploits suitable relational encodings of both, (i) the XQuery Data Model (XDM), *i.e.* tree fragments as well as ordered item sequences, and (ii) the dynamic semantics of XQuery that allow the database back-end to play its trump: set-oriented evaluation. *Pathfinder* determinedly implements this approach, effectively realizing the dashed path in Fig. 1b. Any relational database system may assume the role of *Pathfinder*'s back-end database; the compiler does not rely on XQuery-specific builtin functionality and requires no changes to the underlying database kernel.

Internal XQuery and Data Model Representation. To represent XML fragments, *i.e.* ordered unranked trees of XML nodes, *Pathfinder* can operate with any node-level tabular encoding of trees (node $\hat{=}$ row) that – along other XDM specifics like tag name, node kind, *etc.* – preserves node identity and document order. A variety of such encodings are available, among these are variations of *pre/post* region encodings [8] or ORDPATH identifiers [15]. Ordered sequences of items are mapped into tables in which a dedicated column preserves sequence order.

Pathfinder compiles incoming XQuery expression into plans of relational algebra operators. To actually operate the database back-end in a set-oriented fashion, *Pathfinder* draws the necessary amount of independent work from XQuery's `for`-loops. The evaluation of a subexpression $\in$ in the scope of a `for`-loop yields an ordered sequence of zero or more items in each loop iteration. In *Pathfinder*s relational encoding, these items are laid out in a *single table for all loop iterations*, one item per row. A plan consuming this *loop-lifted* or "unrolled" representation of $\in$ may, effectively, process the results of the individual iterated evaluations of $\in$ in any order it sees fit – or in parallel [10]. In some sense, such a plan is the embodiment of the independence of the individual evaluations of an XQuery `for`-loop body.

Exploitation of Type and Schema Information. Pathfinder's node-level tree encoding is schema-oblivious and does not depend on XML Schema information to represent XML documents or fragments. If the DTD of the XML documents consumed by an expression is available, the compiler annotates its plans with node location and fan-out information that assists XPath location path evaluation but is also used to reduce a query's runtime effort invested in node construction and atomization. *Pathfinder* additionally infers static type information for a query to further prune plans, *e.g.* to control the impact of polymorphic item sequences which present a challenge for the strictly typed table column content in standard relational database systems.

Query Runtime. Internally, *Pathfinder* analyzes the data flow between the algebraic operators of the



XQuery Processors. Figure 1. (a) *DB2 XML* system architecture. (b) Pathfinder: purely relational XQuery on top of vanilla database back-ends. (c) Timber architecture overview [11].

generated plan to derive a series of operator annotations (keys, multi-valued dependencies, *etc.*) that drive plan simplification and reshaping. A number of XQuery-specific optimization problems, *e.g.* the stable detection of value-based joins or XPath twigs and the exploitation of local order indifference, may be approached with simple or well-known relational query processing techniques.

The *Pathfinder* XQuery compiler is retargetable: its internal table algebra has been designed to match the processing capabilities of modern SQL database systems. Standard B-trees provide excellent index support. Code generators exist that emit sequences of SQL:1999 statements [9] (no SQL/XML functionality is used but the code benefits if OLAP primitives like `DENSE_RANK` are available). Bundled with its code generator targeting the extensible column store kernel *MonetDB*, *Pathfinder* constitutes XQuery technology that processes XML documents in the Gigabyte-range in interactive time [7].

Timber: A Native XML Database System

Timber [11] is an XML database system which manages XML data natively: i.e. the XML data instances are stored in their actual format and the XQueries are processed directly. Because of this native representation, there is no overhead for converting the data between the XML format and the relational representation or for translating queries from the XQuery format into the SQL format. While many components of the traditional database can be reused (for example, the transaction management facilities), other components need to be modified to accommodate the new data model and query language. The key contribution of the Timber system is a comprehensive set-at-a-time query evaluation engine based on an XML manipulation algebra, which incorporates novel access methods and algebraic rewriting and cost based optimizations. An architecture overview of Timber is shown in Fig. 1c, as presented in [11].

XML Data Model Representation. When XML documents are loaded into the system, Timber automatically assigns each XML node with four labels $\langle D, S, E, L\rangle$, where D indicates which document the node belongs to, and S, E, L represents the *start key*, *end key*, and *level* of the node, respectively. These labels allow quick detection of relationships between nodes. For example, a node $\langle d_1,s_1,e_1,l_1\rangle$ is an ancestor of another node $\langle d_1,s_2,e_2,l_2\rangle$ iff $s_1 < s_2 \wedge e_1 > e_2$. Each node, along with the labels, are stored natively, and in the order of their start keys (which correspond to their document order), into the Timber storage backend.

XQuery Representation. A central concept in the Timber system is the TLC (Tree Logical Class) algebra [12,18,17], which manipulates sets (or sequences) of heterogeneous, ordered, labeled trees. Each operator in the TLC algebra takes as input one or more sets (or sequences) of trees and produces as output a set (sequence) of trees. The main operators in TLC include *filter*, *select*, *project*, *join*, *reordering*, *duplicate-elimination*, *grouping*, *construct*, *flatten*, *shadow/illuminate*. There are several important features of the TLC algebra. First, like the relational algebra, TLC is a "proper" algebra with compositionality and closure. Second, TLC efficiently manages the heterogeneity arising in XML query processing. In particular, heterogenous input trees are reduced to homogenous sets for bulk manipulation through tree pattern matching. This tree pattern matching mechanism is also useful in selecting portions of interest in a large XML tree. Third, TLC can efficiently handle both set and sequence semantics, as well as a hybrid semantics, where part of the input collection of trees is ordered [17]. Finally, the TLC algebra covers a large fragment of XQuery, including nested FLWOR expressions. Each incoming XQuery is parsed and compiled into the TLC algebra representation (i.e. logical query plans) before being evaluated against the data.

The Timber system, and its underlying algebra, has been extended to deal with text manipulation [2] and probabilistic data [14]. A central challenge with querying text is that exact match retrieval is too crude to be satisfactory. In the field of information retrieval, it is standard practice to use scoring functions and provide ranked retrieval. The TIX algebra [2] shows how to compute and propagate scores during XML query evaluation in Timber. Traditionally, databases have only stored facts, which by definition are certain. Recently, there has been considerable interest in the management of uncertain information in a database. The bulk of this work has been in the relational context, where it is easy to speak of the probability of a tuple being in a relation. ProTDB develops a model for storing and manipulating probabilistic data efficiently in XML [14]. The probability of occurrence of an element in a tree (at that position) is recorded conditioned on the probability of its parent's occurrence.

Query Runtime: Evaluation and Optimization. The heart of the Timber system is the query evaluation engine, which compiles logical query plans into the physical algebra representation (i.e. physical query plans) and evaluates those query plans against the stored XML data to produce XML results. It includes two main subcomponents: the *query optimizer* and the *query evaluator.*

The query evaluator executes the physical query plan. The separation between the logical algebra and the physical algebra is greater in XML databases than in relational databases, because the logical algebra here manipulates trees while the physical algebra manipulates "nodes" – data are accessed and indexed at the granularity of nodes. This requires the design of several new physical operators. For example, for each XML element, the query may need to access the element node itself, its child sub-elements, or even its entire descendant subtree. This requires the *node materialization* physical operator, which takes a (set of) node identifier(s) and returns a (set of) XML tree(s) that correspond to the identifier(s). When and how to materialize the nodes affects the overall efficiency of the physical query plan and it is the job of the optimizer to make the right decision. *Structural join* is another physical operator that is essential for the efficient retrieval of data nodes that satisfy certain structural constraints. Given a parent-child or ancestor-descendant relationship condition, the structural join operator retrieve all pairs of nodes that satisfy the condition. Multiple structural join evaluations are typically required to process a single tree-pattern match. In Timber, a whole stack-based family of algorithms has been developed to efficiently process structural joins, and they are at the core of query evaluation in Timber [1].

The query optimizer attempts to find the most efficient physical query plan that corresponds to the logical query plan. Every pattern match in Timber is computed as a sequence of structural joins and the order in which these are computed makes a substantial difference to the evaluation cost. As a result, *join order selection* is the predominant task of the optimizer. Heuristics developed for relational systems often do not work well for XML query optimization [20]. Timber employs a dynamic programming algorithm to enumerate a subset of all the possible join plans and picks the plan with the lowest cost. The cost is calculated by the *result size estimator,* which relies on the *position histogram* [19] to estimate the lower and upper bounds of each structural join.

DB2 XML: A Hybrid Relational and XML DBMS

DB2 XML (DB2 is a trademark of IBM Corporation.) is a hybrid relational and XML database management system, which unifies new native XML storage, indexing and query processing technologies with existing relational storage, indexing and query processing. A *DB2 XML* application can access XML data using either SQL/XML or XQuery [6,16]. The general system architecture is shown in Fig. 1a. It builds on the premises that (i) relational and XML data will co-exist and complement each other in enterprise information management solutions, and (ii) XML data are different enough that it requires its own storage and processor.

Internal XQuery and Data Model Representation. At the heart of *DB2 XML*'s native XML support is the XML data type, introduced by SQL/XML. *DB2 XML* introduces a new native XML storage format to store XML data as instances of the XQuery Data Model in a structured, type-annotated tree. By storing the binary representation of type-annotated XML trees, *DB2 XML* avoids repeated parsing and validation of documents.

In *DB2 XML*, XQuery is not translated into SQL, but rather mapped directly onto an internal query graph model (QGM) [6], which is a semantic network used to represent the data flow in a query. Several QGM entities are re-used to represent various set operations, such as iteration, join and sorting, while new entities are introduced to represent path expressions and to deal with sequences. The most important new operator is the one that captures XPath expressions. *DB2 XML* does not normalize XPath expressions into FLWOR blocks, where iteration between steps and within predicates is expressed explicitly. Instead, XPath expressions that consist of solely navigational steps are expressed as a single operator. This allows *DB2 XML* to apply rewrite and cost-based optimization [4] to complex XQueries, as the focus is not on ordering steps of an XPath expression.

Exploitation of Type and Schema Information. DB2 XML provides an XML Schema repository (XSR) to register and maintain XML schemas and uses those schemas to validate XML documents. An important feature of *DB2 XML* is that it does not require an XML schema to be associated with an XML column. An XML column can store documents validated according to many different and evolving schemas, as well as schema-less documents. Hence, the association between schemas and XML documents is on per document basis, providing maximum flexibility.

As *DB2 XML* has been targeted to address schema evolution [5], it does not support schema import or static typing features of XQuery. These two features are too restrictive because they do not allow conflicting schemas and each document insertion or schema update may result in recompilation of applications. Hence, *DB2 XML* does not exploit XML schema information in query compilation. However, it uses simple data type information for optimization, such as selection of indexes.

Query Runtime. DB2 XML query evaluation runtime contains three major components for XML query processing:

1. *XQuery Function Library: DB2 XML* supports several XQuery functions and operators on XML schema data types using native implementations.
2. *XML Index Runtime: DB2 XML* supports indexes defined by particular XML path expressions, which can contain wildcards, and descendant axis navigation, as well as kind tests. Under the covers, an XML index is implemented with two B+Trees: a *path index*, which maps distinct reverse paths to generated path identifiers, and a value index that contains path identifiers, values, and node identifiers for each node that satisfy the defining XPath expression. As indexes are defined via complex XPath expressions, *DB2 XML* employs the XPath containment algorithm of [3] to identify the indexes that are applicable to a query.
3. *XML Navigation:* XNAV operator evaluates multiple XPath expressions and predicate constraints over the native XML store by traversing parent-child relationship between the nodes [13]. It returns node references (logical node identifiers) and atomic values to be further manipulated by other runtime operators.

Key Applications

Scalable systems for XML data storage and XML query processing are essential to effectively manage increasing amount of XML data on the web.

URL to Code

Pathfinder

The open-source retargetable Relational XQuery compiler *Pathfinder* is available and documented at www.pathfinder-xquery.org. *MonetDB/XQuery – Pathfinder* bundled with the relational database back-end *MonetDB* – is available at www.monetdb-xquery.org.

Timber

Timber is available and documented at www.eecs.umich.edu/db/timber.

Cross-references

► XML Benchmark
► XML Indexing
► XML Query Processing and XML Algebra
► XML Storage
► XPath/XQuery

Recommended Reading

1. Al-Khalifa S., Jagadish H.V., Patel J.M., Wu Y., Koudas N., and Srivastava D. Structural joins: a primitive for efficient XML query pattern matching. In Proc. 18th Int. Conf. on Data Engineering, 2002, pp. 141–152
2. Al-Khalifa S., Yu C., and Jagadish H.V. Querying structured text in an XML database. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 4–15
3. Balmin A., Özcan F., Beyer K.S., Cochrane R.J., and Pirahesh H. A Framework for using materialized XPath views in XML query processing. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004, p. 6071.
4. Balmin A. et al. Integration cost-based optimization in DB2 XML. IBM Syst. J., 45(2):299–230, 2006.
5. Beyer K.S. and Özcan F. et al. System RX: one part relational, one part XML. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2005, pp. 347–358.
6. Beyer K.S., Siaprasad S., and van der Linden B. DB2/XML: designing for evolution. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2005, pp. 948–952.
7. Boncz P.A., Grust T., van Keulen M., Manegold S., Rittinger J., and Teubner J. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2006, pp. 479–490.
8. Grust T. Accelerating XPath location steps. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 109–220.
9. Grust T., Mayr M., Rittinger J., Sakr S., and Teubner J. A SQL: 1999 code generator for the pathfinder XQuery compiler. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2007, pp. 1162–1164.
10. Grust T., Sakr S., and Teubner J. XQuery on SQL hosts. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004, pp. 252–263.
11. Jagadish H.V., Al-Khalifa S., Chapman A., Lakshmanan L.V.S., Nierman A., Paparizos S., Patel J., Srivastava D., Wiwatwattana N., Wu Y., and Yu C. TIMBER: a native XML database. VLDB J., 11:274–291, 2002.
12. Jagadish H.V., Lakshmanan L.V.S., Srivastava D., and Thompson K. TAX: a tree algebra for XML. In Proc. 8th Int. Workshop on Database Programming Languages, 2001, pp. 149–164.
13. Josifovski V., Fontoura M., and Barta A. Querying XML streams. VLDB J., 14(2):197–210, 2005.

14. Nierman A. and Jagadish H.V. ProTDB: probabilistic data in XML. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 646–657.
15. O'Neil P., O'Neil E., Pal S., Cseri I., Schaller G., and Westburg N. ORDPATHs: insert-friendly XML node labels. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 903–908.
16. Özcan F., Chamberlin D., Kulkarni K.G., and Michels J.-E. Integration of SQL and XQuery in IBM DB2. IBM Syst. J., 45 (2):245–270, 2006.
17. Paparizos S. and Jagadish H.V. Pattern tree algebras: sets or sequences? In Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 349–360.
18. Paparizos S., Wu Y., Lakshmanan L.V.S., and Jagadish H.V. Tree logical classes for efficient evaluation of XQuery. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 71–82.
19. Wu Y., Patel J.M., and Jagadish H.V. Estimating answer sizes for XML queries. In Advances in Database Technology, Proc. 8th Int. Conf. on Extending Database Technology, 2002, pp. 590–608.
20. Wu Y., Patel J.M., and Jagadish H.V. Structural join order selection for XML query optimization. In Proc. 19th Int. Conf. on Data Engineering, 2003, pp. 443–454.

XSL Formatting Objects

► XSL/XSLT

XSL/XSLT

Bernd Amann
Pierre Marie Curie University (upmc), Paris, France

Synonyms

eXtensible Stylesheet Language; eXtensible Stylesheet Language transformations; XSL-FO; XSL formatting objects

Definition

XSL (eXtensible Stylesheet Language) is a family of W3C recommendations for specifying XML document transformations and typesettings. XSL is composed of three separate parts:

- XSLT (eXtensible Stylesheet Language Transformations): a template-rule based language for the structural transformation of XML documents.
- XPath (XML Path Language): a structured query language for the pattern, type and value-based selection of XML document nodes.
- XSL-FO (XML Formatting Objects): an XML vocabulary for the paper document oriented typesetting of XML documents.

Historical Background

The development of XSL was mainly motivated by the need for an open typesetting standard for displaying and printing XML documents. Its conception was strongly influenced by the DSSSL (Document Style Semantics and Specification Language) ISO standard (ISO/IEC 10179:1996) for SGML documents. Like DSSL, XSL separates the document typesetting task into a *transformation* task and a *formatting* task. Both languages are also based on *structural recursion* for defining transformation rules, but whereas DSSL applies a functional programming paradigm, XSL uses XML-template rules and XPath pattern matching for defining document transformations.

The W3C working group on XSL was created in December 1997 and a first working draft was released in August 1998. XSLT 1.0 and XPath 1.0 became W3C recommendations in November 1999, and XSL-FO reached recommendation status in October 2001. During the succeeding development of XQuery, both the XQuery and XSLT Working Groups shared responsibility for the revision of XPath, which became the core language of XQuery. XSLT 2.0, XPath 2.0 and XQuery 1.0 achieved W3C recommendation status in January 2007.

Foundations

XSLT Programming

XSLT programming consists in defining collections of *transformation rules* that can be applied to different classes of document nodes. Each rule is composed of a *matching pattern* and a possibly empty *XML template.* The matching pattern is used for dynamically binding rules to nodes according to their local (name, attributes, attribute values) and structural (document position) properties. Rule templates are XML expressions composed of static XML output fragments and dynamic XSLT instructions generating new XML fragments from the input data.

The following example illustrates the usage of XSLT template rules for implementing some simple