X

XA Standard

► Two-Phase Commit Protocol

XMI

► XML Metadata Interchange

XML

Michael Rys
Microsoft Corporation, Sammamish, WA, USA

Synonyms

Extensible markup language; XML 1.0

Definition

The Extensible Markup Language or XML for short is a markup definition language defined by a World Wide Web Consortium Recommendation that allows annotating textual data with tags to convey additional semantic information. It is extensible by allowing users to define the tags themselves.

Historical Background

XML was developed as a simplification of the ISO Standard General Markup Language (SGML) in the mid 1990s under the auspices of the World Wide Web Consortium (W3C). Some of the primary contributors were Jon Bosak of Sun Microsystems (the working group chair), Tim Bray (then working at Textuality and Netscape), Jean Paoli of Microsoft, and C. Michael Sperberg-McQueen of then the University of Chicago. Initially released as a version 1.0 W3C recommendation on 10 Feb. 1998, XML has undergone several revisions since then. The latest XML 1.0 recommendation edition is the fourth edition as of this writing. A fifth edition is currently undergoing review. The fifth edition is adding some functionality into XML 1.0 that was part of the XML 1.1 recommendation, that has achieved very little adoption.

Based on the XML recommendation, a whole set of related technologies have been developed, both at the W3C and other places. Some technologies are augmenting the core syntactic XML recommendation such as the XML Namespaces and XML Information Set recommendations, others are building additional infrastructure on it such as the XML Schema recommendation or the XSLT, XPath and XQuery family of recommendations. Since XML itself is being used to define markup vocabularies, it also forms the basis for vertical industry standards in manufacturing, finance and other areas, including standard document formats such as XHTML, DocBook, Open Document Format (ODF) and Office Open XML (OOXML) and forms the foundation of the web services infrastructure.

Foundations

XML's markup format is based on the notion of defining well-formed documents and is geared towards human-readability and international usability (by basing it on Unicode). Among its design goals were ease of implementation by means of a simple specification, especially compared to its predecessor, the SGML specification, and to make it available on a royalty-free basis to both implementers and users.

Well-formed documents basically contain markup elements that have a begin tag and an end tag as in the example below:

< tag > character data < /tag >

Element tags can have attributes associated with it that provide information about the tag, without interfering with the flow of the textual character data that is being marked up:

<tag attribute1 = "value" attribute2 = "42">
character data </tag>

Processing instructions can be added to convey processing information to XML processors and comments can be added. And comments can be added for commenting and documentation purposes. = They follow different syntactic forms than element tags and can appear anywhere in a document, except within the begin tag and end tag tokens themselves:

```
<?xml-stylesheet type="application/xslt + xml"
                 href="#style1"" ? >
<!- This is a comment –>
```

A well-formed document must also have exactly one top-level XML element, and can contain several processing instructions and comments on the top-level next to the element.

The order among these elements is information-bearing, since they are meant to mark up an existing document flow. Thus, the following two well-formed XML documents are not the same:

```
<doc> <element> value1 </element>
<element> value2 </element> </doc>
 <doc> <element> value2 </element>
<element> value1 </element> </doc>
```

The XML information set recommendation defines an abstract data model for these syntactic components, introducing the notion of document information items for a document, element information items for element tags, attribute information items for their attributes, character information items for the marked up character data etc.

The XML namespace recommendation adds the ability to scope an element tag name to a namespace URI, to provide the ability to scope markup vocabularies to a domain identifier.

Besides defining an extensible markup format, the XML recommendation also provides a mechanism to constrain the XML document markup to follow a certain grammar by restricting the allowed tag names and composition of element tags and attributes with document type declarations (DTDs). Documents that follow the structure given by DTDs are not only well-formed but also valid. Note that the XML Schema recommendation provides another mechanism to constrain XML documents.

Finally, XML also provides mechanisms to reuse parts of a document (down to the character level) using so called entities.

For more information about XML, please refer to the recommended reading section.

Key Applications

While XML was originally designed as an extensible document markup format, it has quickly taken over tasks in other areas due to the wide-availability of free XML parsers, its readability and flexibility. Besides the use for document markup, two of the key application scenarios for XML are the use for interoperable data interchange in loosely-coupled systems and for adhoc modeling of semi-structured data.

XML's first major commercial applications actually have been to describe the data and, with DTDs or XML schema formats, structures of messages that are being exchanged between different computer systems in application to application data exchange scenarios and web services. XML is not only being used to describe the message infrastructure format and information such as the SOAP protocol, RSS or Atom formats, but also to describe the structure and data of the message payloads. Often, XML is also used in more adhoc micro-formats for more REST-ful web services.

At the same time that XML was being developed, several researcher groups were looking into data models that were less strict than relational, entity-relationship and object-oriented models, by allowing instance based properties and heterogeneous structures. XML's tree model provides a good fit to represent such semi-structured, hierarchical properties, and its flexible format is well–suited to model the sparse properties and rapidly changing structures that are often occurring in semi-structured data. Therefore, XML has often been used to model semi-structured data in data modeling.

XML support has been added to databases on form of either pure XML databases or by extending existing database platforms such as relational database systems to enable databases to manage XML documents serving all these three application scenarios.

Cross-references

- ▶ XML Attribute
- ▶ XML Document
- ▶ XML Element
- ▶ XML Schema
- ▶ XPath/XQuery
- ▶ XSL/XSLT

Recommended Reading

1. Namespaces in XML 1.0, latest edition. Available at: http://www.w3.org/TR/xml-names
2. Wikipedia entry for XML. Available at: http://en.wikipedia.org/wiki/XML
3. XML 1.0 information Set, latest edition. Available at: http://www.w3.org/TR/xml-infoset
4. XML 1.0 recommendation, latest edition. Available at: http://www.w3.org/TR/xml
5. XML 1.1 recommendation, latest edition. Available at: http://www.w3.org/TR/xml11

XML (Almost)

► Semi-structured Data

XML 1.0

► XML

XML Access Control

Dongwon Lee[1], Ting Yu[2]
[1]The Pennsylvania State University, University Park, PA, USA
[2]North Carolina State University, Raleigh, NC, USA

Definition

XML access control refers to the practice of limiting access to (parts of) XML data to only authorized users. Similar to access control over other types of data and resources, XML access control is centered around two key problems: (i) the development of formal models for the specification of access control policies over XML data; and (ii) techniques for efficient enforcement of access control policies over XML data.

Historical Background

Access control is one of the fundamental security mechanisms in information systems. It is concerned with who can access which information under what circumstances. The need for access control arises naturally when a multi-user system offers selective access to shared information. As one of the oldest problems in security, access control has been studied extensively in a variety of contexts, including operating systems, databases, and computer networks.

The most influential policy models today are discretional access control (DAC), mandatory access control (MAC), and role-based access control (RBAC) models. In DAC, the owner of an object (e.g., a file or database table) solely determines which subjects can access that object, and whether such privileges can be further delegated to other subjects. In MAC, whether a subject can access an object or not is determined by their security classes, not by the owner of the object. In RBAC, privileges are associated with roles. Users are assigned to roles, and thus can only exercise access privileges characterized by their roles.

Typical implementations of access control are in the form of access control lists (ACLs) and capabilities. In ACLs, a system maintains a list for each object of subjects who have access to that object. In capabilities, each subject is associated with a list that indicates those objects to which it has access. ACLs and capabilities are suitable to enforce coarse-grained access control over objects with simple structures (e.g., file systems or table-level access control in relational databases). They are often not efficient for fine-grained access control over objects with complex structures (e.g., element-level access control in XML, or row-level and cell-level access in relational databases).

Foundations

XML access control is fine-grained in nature. Instead of controlling access to the whole XML database or document, it is often required to limit a user's access to some substructures of an XML document (e.g., some subtrees or some individual elements).

An XML access control policy can be typically modeled as a set of access control rules. Each rule is a 5-tuple (*subject*, *object*, *action*, *sign*, *type*), where (i) *subject* defines a set of subjects; (ii) *object* defines a set of elements or attributes of an XML document; (iii) *action* denotes the actions that can be performed on XML (e.g., read, write, and update); (iv) *sign* $\in\{+, -\}$ indicates whether this rule is a grant rule or a deny rule; and (v) *type* $\in\{LC, RC\}$ refers to either *local check* or *global check*. Intuitively, an access control rule specifies that subjects in *subject* can (when *sign*=+) or cannot (when *sign*=−) perform action specified by *action* on those elements or attributes in *objects*. When *type*=*RC*, this authorization decision also applies to the descendants of those in *object*.

Object is usually specified using some XML query languages such as XPath expressions. There are multiple ways to identify *subject*. Following RBAC, *subject* can be specified as one or several roles (e.g., student and faculty) [4]. It may also follow *attribute-based access control*, where each user features a set of attributes and *subject* is defined based on the values of those attributes (e.g., those subjects whose age is over 21). Some access control policy in the literature also follows MAC, where *subject* refers to security classes.

Example 1. Consider two access control rules for the subject of "admin" as follows:

R_1: (*admin, /people/person/name, read, −, LC*)
R_2: (*admin, /people//address//*, read/update, +, RC*)

R_1 Indicates that users belonging to the admin role cannot read textual and attribute data of XML node <name>, the child of <person> that is the child of the root node <people>. On the other hand, R_2 specifies that the same users of the admin role can read and even update any XML data that are descendents of XML node if they are descendents of <address> under <people>.

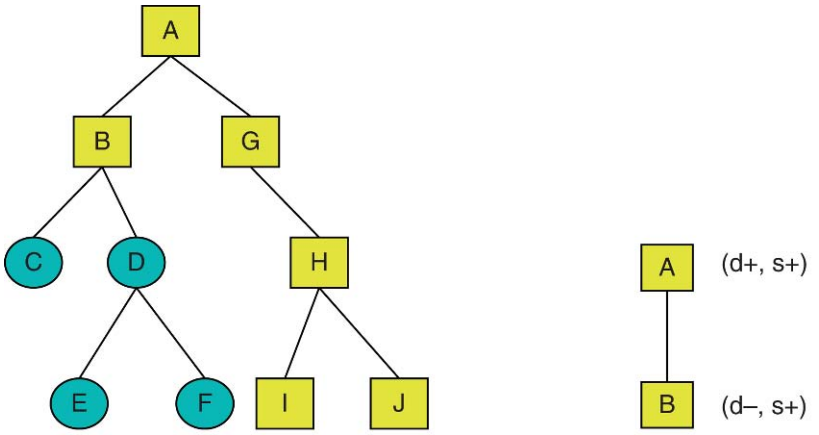
Once an access control policy is specified, there are two problems that need to be addressed. First, given any access request, one needs to determine whether or not a rule exists that applies to the request. If so, the policy is said to be *complete*. Most access control systems adopt a closed world assumption. That is, if no rules apply to a request, the request is denied. Second, multiple rules with different authorization decisions may apply to a request. One typical way to resolve such conflicts is to let denial override permit. For XML access control, it may also be solved by having more explicit rules override less explicit ones. For instance, an LC rule may override an RC rule. For two RC rules, the one that applies to a node's nearest ancestor usually dominates.

Languages for specifying access control policy are proposed in such efforts as XACL by IBM [7]. Therefore, it is also possible to use much more expressive languages to specify access control policy within XML access control models. Finally, the use of authorization priorities with propagation and overriding are related to similar techniques studied in object-oriented databases.

Once XML access control is specified in a given model, it can be enforced in a variety of ways. By and large, most of the existing XML access control methods are either *view-based* or rely on the XML engine to enforce access control at the node-level of XML trees. The idea of view-based enforcement (e.g., [5,11]) is to create and maintain a separate view for each user (or role) who is authorized to access a specific portion of an XML data [4]. The view contains exactly the set of data nodes that the user is authorized to access. After views are constructed, during run time, users can simply run their queries against the views without worrying about access control issues. Although views can be prepared offline, in general, view-based enforcement schemes suffer from high maintenance and storage costs, especially for a large number of roles: (i) a virtual or physical view is created and stored for each role; (ii) whenever a user prompts *update* operation on the data, all views that contain the corresponding data need to be synchronized. To tackle this problem, people proposed a method using compressed XML views to support access controls [11]. The basic idea is based on the observation of accessibility locality, i.e., elements close to each other tend to have the same accessibility for a subject. Based on this observation, a compressed accessibility map only maintains some "crucial" nodes in the view. With simple annotation on those crucial nodes, other nodes' accessibility can be efficiently inferred instead of explicitly stored. Each node in the compressed view is associated with a label (*desc*, *self*), where *desc* can be either d+ or d−, indicating whether its descendants are accessible or not, and *self* can be either s+ or s−, indicating whether the node itself is accessible. Given any node in an XML tree, by its relationship to those labeled nodes in the compressed view, we can infer its accessibility.

Example 2. Consider the XML tree in Fig. 1a with squares and circles denoting accessible and inaccessible nodes, respectively. The corresponding compressed view is shown in Fig. 1b. Note that since node C is a descendant of B and B is labeled (d−, s+), C can be inferred to be inaccessible.

In the non view-based XML access control techniques (e.g., [2,9,10]), an XML query is pre-classified against the given model to be "entirely" authorized, "entirely" prohibited, or "partially" authorized before being submitted to an XML engine. Therefore, without using pre-built views, those entirely authorized or prohibited queries can be quickly processed. Furthermore, those XML queries that are partially authorized are re-written using state machines such that they request for only data that are granted to the users or roles.



a XML document as a tree b Compressed view

Example 3. Consider three access control rules for the security role "admin." Furthermore, an administrator "Bob" requested three queries, Q_1 to Q_3 in XPath as follows:

R_1: (admin, /people/person/name, read, −, LC)
R_2: (admin, /people//address//, read/update, +, RC)*
R_3: (admin, /regions/namerica/item/name, read, +, LC)
Q_1: /people/person/address/street
Q_2:/people/person/creditcard
*Q_3:/regions//**

Then, Q_1 by Bob can be entirely authorized by both R_1 and R_2, but entirely denied by R_3. Similarly, Q_2 is entirely authorized by R_1, entirely denied by both R_2 and R_3. Finally, Q_3 is entirely accepted by R_1, entirely denied by R_2, and partially authorized by R_3 and needs to be re-written to /regions/namerica/item/name in order not to be conflicted with R_3.

Key Applications

As XML has been increasingly used not only as a data exchange format but as a data storage format, the problem of controlling selective access over XML is indispensable to data security. Therefore, XML access control issues have been tightly associated with secure query processing techniques in relational and XML databases (e.g., [3,5]). The access control policy model of XML can also be extended to express security requirements of other semi-structured data such as LDAP and object-oriented databases. The aforementioned access control enforcement techniques can be further extended to protect privacy during the exchange of XML data in distributed information sharing system. For instance, in PPIB system [8], XML access control is used to hide what query content is or where data objects are located, etc. In an environment where XML data are stored in a distributed fashion and users may ask sensitive queries whose privacy must be kept to is utmost extent (e.g., HIV related queries in health information network), XML access control techniques can be used, along with XML content-based routing techniques [6].

Cross-references

► Relational Access Control
► Secure XML Query Processing

Recommended Reading

1. Bertino E. and Ferrari E. Secure and selective dissemination of XML documents. ACM Trans. Inform. Syst. Secur., 5(3):290–331, 2002.
2. Bouganim L., Ngoc F.D., and Pucheral P. Client-based access control management for XML documents. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004, pp. 84–95.
3. Cho S., Amer-Yahia S., Lakshmanan L.V.S., and Srivastava D. Optimizing the secure evaluation of twig queries. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 490–501.
4. Damiani E., Vimercati S., Paraboschi S., and Samarati P. A fine-grained Access Control System for XML Documents. ACM Trans. Inform. Syst. Secur., 5(2):169–202, 2002.
5. Fan W., Chan C.-Y., and Garofalakis M. Secure XML querying with security views. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 587–598.
6. Koudas N., Rabinovich M., Srivastava D., and Yu T. Routing XML queries. In Proc. 20th Int. Conf. on Data Engineering, 2004, p. 844.
7. Kudo M. and Hada S. XML document security based on provisional authorization. In Proc. 7th ACM Conf. on Computer and Communications Security, 2002, pp. 87–96.
8. Li F., Luo B., Liu P., Lee D., and Chu C.H. Automaton segmentation: a new approach to preserve privacy in XML information brokering. In Proc. 14th ACM Conf. on Computer and Communications Security, 2007, pp. 508–518.
9. Luo B., Lee D., Lee W.C., and Liu P. QFilter: fine-grained runtime XML access control via NFA-based query rewriting.

In Proc. Int. Conf. on Information and knowledge Management, 2004, pp. 543–552.
10. Murata M., Tozawa A., and Kudo M. XML access control using static analysis. In Proc. 10th ACM Conf. on Computer and Communication Security, 2003, pp. 73–84.
11. Yu T., Srivastava D., Lakshmanan L.V.S., and Jagadish H.V. A compressed accessibility map for XML. ACM Trans. Database Syst., 29(2):363–402, 2004.

XML Algebra

► XML Tuple Algebra

XML Application Development

► XML Programming

XML Attribute

Michael Rys
Microsoft Corporation, Sammamish, WA, USA

Synonyms

XML attribute; XML attribute node

Definition

An XML attribute is used to represent additional information on an XML element in the W3C XML recommendation [2].

Key Points

The name of an XML attribute has to be unique for a given element. Therefore, the following XML element is not allowed

<e a1 = "v1" a1 = "v2"/>
while the following is well-formed
<e1 a1 = "v1"> <e2 a1 = "v2"/> </e1>

An XML attribute has a value that follows the attribute value normalization rules outlined in section 3.3.3 of [2]. This means that several whitespace characters do not get preserved in attribute values, unless they are explicitly represented with a character entity (e.g.,
 for carriage return).

Attributes can be constrained and typed by schema languages such as XML DTDs [2] or XML Schema [3].

An XML attribute is represented as an XML attribute information item in the XML Information Set [1] and an XML attribute node in the XPath and XQuery data model [4].

Cross-references

► XML
► XML Document
► XML Element

Recommended Reading

1. XML 1.0 Information Set, latest edition available online at: http://www.w3.org/TR/xml-infoset.
2. XML 1.0 Recommendation, latest edition available online at: http://www.w3.org/TR/xml.
3. XML Schema Part 0: Primer, latest edition available online at: http://www.w3.org/TR/xmlschema-0/.
4. XQuery 1.0 and XPath 2.0 Data Model (XDM), latest edition available online at: http://www.w3.org/TR/xpath-datamodel/.

XML Attribute Node

► XML Attribute

XML Benchmarks

Denilson Barbosa[1], Ioana Manolescu[2], Jeffrey Xu Yu[3]
[1]University of Alberta, Edmonton, AB, Canada
[2]INRIA, Saclay–Île-de-France, Orsay, France
[3]The Chinese University of Hong Kong, Hong Kong, China

Definition

An XML benchmark is a specification of a set of meaningful and relevant tasks, intended to assess the functionality and/or performance of an XML processing tool or system. The benchmark must specify the following: (i) a deterministic workload, consisting of a set of XML documents and/or a procedure for obtaining these and a set of operations to be performed;

(ii) detailed rules for executing the workload and making the measurements; (iii) the metrics used to report the results of the benchmark; and (iv) standard ways of interpreting the results.

Historical Background

XML has quickly become the preferred format for representing and exchanging data on the Web age. The level of acceptance of XML is astonishing, especially when one considers that this technology was introduced only in 1997. XML is an enabling technology with applications in virtually all domains of information processing. At the time of writing, XML is widely used in content distribution on the Web (e.g., RSS feeds), as the foundation of large initiatives such as the Semantic Web and Web Services, and is the basis for routinely used productivity tools, such as text editors and spreadsheets.

Such complexity led to the development of a large number of fairly narrowly-scoped benchmarks for XML. Moreover, there is still no clear understanding of what an XML benchmark should look like.

The first XML benchmarks were developed in academia for testing specific processing tasks and/or relatively narrow applications. In fact, some of these earlier benchmarks are categorized as micro-benchmarks. For example, XMach emulated the scenario of an XML file servers using a large number of simple XML files, while XMark modeled an online auction application using a single complex XML document. Over time, XML benchmarks evolved to contemplate more realistic and complex application scenarios, in an attempt to emulate the typical workload of larger applications. For instance, XBench modeled four scenarios, resulting from the combination of two factors: (i) the nature of the documents (data-centric vs. document-centric); and (ii) the number of documents in the workload (single-document vs. multi-document).

The first XML Benchmark developed entirely by industry was TPoX (Transaction Processing over XML), which simulates a financial application in a multi-user environment with concurrent access to the XML data. TPoX is intended for testing all aspects of a relational-based XML storage and processing system.

Another development worth mentioning concerns the declarative synthetic data generators for XML. While, strictly speaking, these are not benchmarks, these tools help in obtaining appropriate testing data with reasonably low effort and high enough customizability.

Foundations

Benchmarks should be simple, portable, scale to different workload sizes, and allow the objective comparisons of competing systems and tools [4]. It should be noted that the diversity and complexity of its applications, developing meaningful and realistic benchmarks for XML is a truly herculean task. Also, XML processing tools fall into many categories, from simple storage services to sophisticated query processors, thus adding to the complexity of developing relevant and *realistic* XML benchmarks.

The two factors above have led to the development of benchmarks that are relatively narrow in scope, focusing on very specific tasks. Moreover, the lack of universally accepted, comprehensive XML benchmarks, resulted in the development of general-purpose synthetic data generators, which allow the user to obtain customized test data with relatively low effort. This is significant, as such tools were not popular until the advent of XML.

XML Microbenchmarks

A micro-benchmark is a narrowly-defined benchmark aimed at testing very specific aspects of a tool and/or system.

The Michigan Benchmark The Michigan Benchmark is an XML Micro-benchmark developed at the University of Michigan [8]. It uses a single synthetic document that does not resemble a typical document from any real world application domain. Instead, it is carefully designed to allow the testing of the following query processing operations: matching attributes by value; selecting elements by name; evaluation of positional predicates; selection of nodes based on predicates over their parent, children, ancestors or descendants; join operations; computing aggregate functions; and processing updates. The authors applied the benchmark to three database systems: two native XML DBMSs, and a commercial ORDBMS.

XML Application Benchmarks

An application benchmark is a comprehensive set of tasks that approximates the workload of a typical application in the respective domain. Four important

XML application benchmarks are XMach-1 [2], XMark [9], XBench [10], and TPOX [7].

XMach-1 XMach-1 (*X*ML Data *Ma*nagement Ben*ch*-mark) is a multi-user benchmark developed at the University of Leipzig, Germany [2]. Unlike most existing XML benchmarks that are designed to test the query processors of database management systems, XMach-1 is designed to test database management systems which include a query processor as well as the other key components. In terms of measurement, XMach-1 evaluates systems based on throughput performance (XML queries per second) instead of response time for user-given XML queries.

XMach-1 considers a system architecture to support web applications, which consists of four main components, namely, XML database, application servers, loaders and browser clients. In the XML database, there are multiple schemas, and there are between 2 and 100 documents per schema. Each document is generated using 10,000 most frequent English words, and occupies between 2 and 100 KB of storage. The workload contains eight queries and three update operations. Some evaluation results can be found in [6].

XMark XMark is the result of the XML benchmark project, led by a team at CWI [9]. It models an Internet auctioning application. The workload consists of a large database, in a single XML document, containing: (i) items for auction in geographically dispersed areas; (ii) bidders and their interests; and (iii) detailed information about existing auctions, which can be open or closed. XMark's workload includes 20 queries that cover the following broad kinds of operations: simple node selections; document queries for which order information is relevant; navigational queries; and computing aggregate functions.

The XMark data generator employs several kinds of probability distributions and uses several real data values (e.g., names of countries and people) to produce realistic data; also, the textual content uses real words of the English language. XMark is by far the most widely used XML benchmark at the time of writing.

XBench XBench is a benchmark suite developed at the University of Waterloo [10]. XBench defines application benchmarks categorized according to two criteria: single-document versus multi-document and *data-centric* versus *text-centric* domains. The latter criterion distinguishes data management and exchange scenarios from content management applications (e.g., electronic publishing). Document collections in XBench range in size from a few kilobytes to several gigabytes, and its workload consists of bulk-loading as well as various query and text-based search operations. Results of an evaluation of four different systems, comprising both native XML stores as well as relational-based stores, are provided in [10].

TPoX TPoX (Transaction Processing over XML) is a comprehensive application benchmark developed jointly by IBM and Intel [7]. TPoX simulates a financial application domain (security trading) and is based on the industry-standard XML Schema specification FIXML [3]. The testing environment in TPoX covers several aspects of XML management inside DBMS, including the use of XQuery, SQL/XML, updates, and concurrent access to the data. The authors report on an experimental evaluation of the IBM DB2 product for storing and processing XML data [7].

Synthetic Data Generators

Synthetic data have other applications besides benchmarking, such as testing specific components of a complex system or application. In this setting, an important requirement for a data generator, besides generating *realistic* data (i.e., synthetic data whose characteristics match those of typical real data in the application), is the the ability of easily customizing the test data (e.g., its structure).

Declarative synthetic data generators, on the other hand, are tools that produce synthetic data according to specifications that describe *what* data to generate, as opposed to *how* to generate such data, thus facilitating the generation of synthetic data. Declarative data generators are intended for easing the burden in obtaining test data, unlike the data generators of standardized benchmarks, which have the characteristics of the data they produce embedded in their source code.

Declarative data generators rely on formalisms providing higher levels of abstraction than programming languages, such as conceptual schema languages annotated with probabilistic information (for describing the characteristics of the intended data). Such probabilistic information are needed because schema languages specify only what content is allowed in valid document instances. A realistic data generator must allow the specification of the characteristics of

typical documents as well. For example, while a schema formalism will specify that a book element may contain between 1 and 10 authors, a realistic data generator will allow one to define a probability distribution for the number of authors in the test data. Another desirable feature of realistic synthetic data is that it satisfies integrity and referential constraints. For instance, an XML document describing a book review should refer to an existing book in the test data.

Two examples of declarative XML data generators are the IBM XML Generator [3], whose data specifications are based on Document Type Definitions, and ToXgene [1], which relies on XML Schema specifications. Both tools allow the specification of skewed probability distributions for elements, attributes, and textual nodes. ToXgene, being based on XML Schema, supports different data types, as well as key and referential constraints. ToXgene also offers a simple declarative query language that allows one to model relatively complex dependencies among elements and attributes involving arithmetic and string manipulation operations. For instance, it allows one to model that the total price of an invoice should be the sum of the individual prices of the items in that invoice multiplied by the appropriate tax rates. Finally, ToXgene offers support for generating recursive XML documents.

Key Applications

Meaningful benchmarks are essential for the development of new technologies, as they allow developers to assess progress and understand intrinsic limitations of their tools. Applications include functionality testing, performance evaluation, and system comparisons.

Recommended Reading

1. Barbosa D. and Mendelzon A.O. Declarative generation of synthetic XML data. Softw. Pract. Exper. 36(10):1051–1079, 2006.
2. Böhme T. and Rahm E. XMach-1: a benchmark for XML data management. In Proc. German Database Conference. Springer, Berlin, 2001, pp. 264–273; Multi-user evaluation of XML data Management Systems with XMach-1. LNCS, Vol. 2590, 2003, pp. 148–159.
3. Financial Information Exchange Protocol. FIXML 4.4 Schema Version Guide. Available at: http://www.fixprotocol.org.
4. Gray J. (ed.). The Benchmark Handbook for Database and Transaction Systems (2nd edn.). Morgan Kaufmann, San Francisco, CA, USA, 1993, ISBN 1-55860-292-5.
5. IBM XML Generator. Available at: http://www.alphaworks.ibm.com/tech/xmlgenerator, 2007.
6. Lu H., Yu J.X., Wang G., Zheng S., Jiang H., Yu G., and Zhou A. What makes the differences: benchmarking XML database implementations. ACM Trans. Internet Technol., 5(1):154–194, 2005.
7. Nicola M., Kogan I., and Schiefer B. An XML transaction processing benchmark. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2007, pp. 937–948.
8. Runapongsa K., Patel J.M., Jagadish H.V., Chen Y., and Al-Khalifa S. The Michigan benchmark: towards XML query performance diagnostics. Inf. Syst., 31(2):73–97, 2006.
9. Schmidt A., Waas F., Kersten M.L., Carey M.J., Manolescu I., Busse R. XMark: a benchmark for XML data management. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 974–985.
10. Yao B.B., Özsu M.T., and Khandelwal N. XBench Benchmark and Performance Testing of XML DBMSs. In Proc. 20th Int. Conf. on Data Engineering, 2004, pp. 621–633.

XML Cardinality Estimation

► XML Selectivity Estimation

XML Compression

Jayant R. Haritsa[1], Dan Suciu[2]
[1]Indian Institute of Science, Bangalore, India
[2]University of Washington, Seattle, WA, USA

Definition

XML is an extremely verbose data format, with a high degree of redundant information, due to the same tags being repeated over and over for multiple data items, and due to both tags and data values being represented as strings. Viewed in relational database terms, XML stores the "schema" with each and every "record" in the repository. The size increase incurred by publishing data in XML format is estimated to be as much as 400% [14], making it a prime target for compression. While standard general-purpose compressors, such as `zip`, `gzip` or `bzip`, typically compress XML data reasonably well, specialized XML compressors have been developed over the last decade that exploit the specific structural aspects of XML data. These new techniques fall into two classes: (i) *Compression-oriented*, where the goal is to maximize the compression ratio of the data, typically up to a factor of two

better than the general-purpose compressors; and (ii) *Query-oriented*, where the goal is to integrate the compression strategy with an XPath query processor such that queries can be processed directly on the compressed data, selectively decompressing only the data relevant to the query result.

Historical Background

Research into XML compression was initiated with Liefke and Suciu's development in 2000 of a compressor called `XMill` [6]. It is based on three principles: separating the structure from the content of the XML document, bucketing the content based on their tags, and compressing the individual buckets separately. `XMill` is a compression-oriented technique, focusing solely on achieving high compression ratios and fast compression/decompression, ignoring the query processing aspects. Other compression-oriented schemes that appeared around the same time include Millau [4], designed for efficient encoding and streaming of XML structures; and XMLPPM, which implements an extended SAX parser for online processing of documents [2]. There are also several commercial offerings that have been featured on the Internet (e.g., [11,13,15]).

Subsequently, the focus shifted to the development of query-oriented techniques intended to support query processing directly on the compressed data. This stream of research began with Tolani and Haritsa [9] presenting in 2002 a system called XGrind, where compression is carried out at the granularity of individual element/attribute values using a simple context-free compression scheme – tags are encoded by integers while textual content is compressed using non-adaptive Huffman (or Arithmetic) coding. XGrind consciously maintains a *homomorphic encoding*, that is, the compressed document is still in XML format, the intention being that all existing XML-related tools (such as parsers, indexes, schema-checkers, etc.) could continue to be used on the compressed document.

In 2003, Min et al. [7] proposed a compressor called XPRESS that extended the homomorphic compression approach of XGrind to include effective evaluation of query path expressions and range queries on numerical attributes. Their scheme uses a reverse arithmetic path-encoding that encodes each path as an interval of real numbers between 0 and 1. The following year produced the XQueC system [1] from Arion et al., which supports cost-based tradeoffs between compact storage and efficient processing.

An excellent survey of the state-of-the-art in XML compressors is available in [1].

Foundations

The basic principles for compressing XML documents are the following:

Separate structure from data: The *structure* consists of XML tags and attributes, organized as a tree. The *data* consists of a sequence of items (strings) representing element text contents and attribute values. The structure and the data are compressed separately.

Group data items with related meaning: Data items are logically or physically grouped into *containers*, and each container is compressed separately. By exploiting similarities between the values in a container, the compression improves substantially. Typically, data items are grouped based on the element type, but some systems (e.g., [1]) chose more elaborate grouping criteria.

Apply specialized compressors to containers: Some data items are plain-text, while others are numbers, dates, etc., and for each of these different domains, the system uses a specialized compressor.

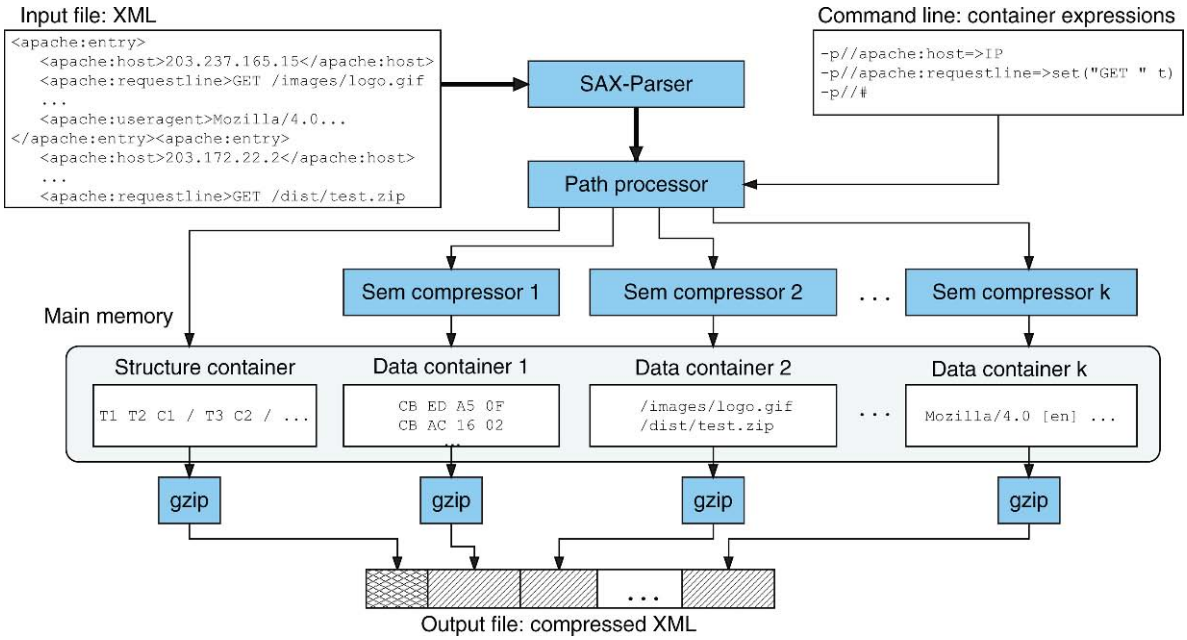
Compression-Oriented XML Compressors

The architecture of the `XMill` compressor [6], which is typical of several XML compressors, is depicted in Fig. 1. The XML file is parsed by a SAX parser that sends tokens to the *path processor*. The purpose of the path processor is to separate the structure from the data, and to further separate the data items according to their semantics.

Next, the structure container and all data containers are compressed with `gzip`, then written to disk. Optionally, the data items in certain containers may be compressed with a user-defined semantic compressor. For example, numerical values or IP addresses can be binary encoded, dates can be represented using specialized data structures, etc.

By default, `XMill` groups data items based on their innermost element type. Users, however, can override this, by providing *container expressions* on the command line. The path processor uses these expressions to determine in which container to store each data item. Path expressions also determine which semantic compressor to apply (if any).

The amount of main memory holding all containers is fixed. When the limit is exhausted all containers are `gzip`-ed, written to disk, as one logical block, then the compression resumes. In effect, this partitions the



XML Compression. Figure 1. Architecture of the XMill compressor [4].

input XML file into logical blocks that are compressed independently.

The decompressor, XDemill, is similar, but proceeds in reverse. It reads one block at a time in main memory, decompresses every container, then merges the XML tags with the data values to produce the XML output.

Example. To illustrate the working of XMill, consider the following snippet of Web-server log data, where each entry represents one HTTP request:

```
<apache:entry>
   <apache:host> 202.239.238.16 </host>
   <apache:requestLine>GET / HTTP/1.0
   </apache:requestLine>
   <apache:contentType> text/html
   </apache:contentType>
   <apache:statusCode> 200
   </apache:statusCode>
   <apache:date> 1997/10/01-00:00:02
   </apache:date>
   <apache:byteCount> 4478
   </apache:byteCount>
   <apache:referer>
     http://www.so-net.jp/
   </apache:referer>
   <apache:userAgent> Mozilla/3.0 [ja]
   </apache:userAgent>
</apache:entry>
```

After the document is parsed, the path processor separates it into the *structure* and the *content*, and further separates the content into different containers. The structure is obtained by removing all text values and attribute values and replacing them with their container number. Start-tags are dictionary-encoded, i.e., assigned an integer value, while all end-tags are replaced by the same, unique token. For illustration purposes, start-tags are denoted with T1, T2,..., the unique end-tag with /, and container numbers with C1, C2,.... In this example the structure of the entry element is:

```
   T1 T2 C1 / T3 C2 / T4 C3 / T5 C4 /
T8 C7 / T8 C7 / T11 C10 / T12 C11 / /
```

Here T1 = apache:entry, T2 = apache:host, and so on, while / represents any end tag. Internally, each token is encoded as an integer: tags are positive, container numbers are negative, and \ is 0. Numbers between (− 64, 63) take one byte, while numbers outside this range take two or four bytes; the example string is overall coded in 26 bytes. The structure is compressed using gzip, which is based on Ziv-Lempel's LZ77 algorithm [10]. This results in excellent compression, because LZ77 exploits very well the frequent repetitions in the structure container: the compressed structure usually amounts to only 1–3% of the compressed file.

Next, data items are partitioned into containers, then compressed. Each container is associated with

an XPath expression that defines which data items are stored in that container, and an optional semantic compressor for the items in that container. By default there is a container for each tag `tag` occurring in the XML file, the associated XPath expression is `//tag`, and there is no associated semantic compressor. Users may override this on the command line. For example:

```
xmill -p //shipping/address -p
//billing/address file.xml
```

creates two separate containers for `address` elements: one for those occurring under `shipping`, and one for those occurring under `billing`. If there exist `address` elements occurring under elements other than `shipping` or `billing`, then their content is stored in a third container. Note that the container expressions only need to be specified to the compressor, not to the decompressor. Overriding the default grouping of data items usually results in only modest improvements in the compression ratio. Much better improvements are achieved, however, with semantic compressors.

Query-Oriented XML Compressors

XGrind. The technique described in [9] is intended to simultaneously provide efficient query-processing performance and reasonable compression ratios. Basic requirements to achieve the former objective are (i) fine-grained compression at the element/attribute granularity of query predicates, and (ii) context-free compression assigning codes to data items independent of their location in the document. Algorithms such as LZ77 are not context-free, and therefore XGrind uses non-adaptive Huffman (or Arithmetic) coding, in which two passes are made over the XML document – the first to collect the statistics and the second to do the actual encoding. A separate character-frequency distribution table is used for each element and non-enumerated attribute, resulting in fine-grained characterization. The DTD is used to identify enumerated-type attributes and their values are encoded using a simple binary encoding scheme, while the compression of XML tags is similar to that of `XMill`. With this scheme, exact-match and prefix-match queries can be completely carried out directly on the compressed document, while range or partial-match queries only require on-the-fly decompression of the element/attribute values that feature in the query predicates.

A distinguishing feature of the `XGrind` compressor is that it ensures *homomorphic* compression – that is, its output, like its input, is semi-structured in nature. In fact, the compressed XML document can be viewed as the original XML document with its tags and element/attribute values replaced by their corresponding encodings. The advantage of doing so is that the variety of efficient techniques available for parsing/querying XML documents can also be used to process the compressed document. Second, indexes can be built on the compressed document similar to those built on regular XML documents. Third, updates to the XML document can be directly executed on the compressed version. Finally, a compressed document can be directly checked for validity against the compressed version of its DTD.

As a specific example of the utility of homomorphic compression, consider repositories of genomic data (e.g., [12]), which allow registered users to upload new genetic information to their archives. With homomorphic compression, such information could be compressed by the user, then uploaded, checked for validity, and integrated with the existing archives, all operations taking place completely in the compressed domain.

To illustrate the working of XGrind, consider the XML student document fragment along with its DTD shown in Figs. 2 and 3. An abstract view of its XGrind compressed version is shown in Fig. 4, in which *nahuff*(s) denotes the output of the Huffman-Compressor for an input data value `s`, while *enum*(s) denotes the output of the Enum-Encoder for an input data value `s`, which is an enumerated attribute. As is evident from Fig. 4, the compressed document output in the second pass is semi-structured in nature, and maintains the property of validity with respect to the compressed DTD.

The compressed-domain query processing engine consists of a lexical analyzer that emits tokens for encoded tags, attributes, and data values, and a parser built on top of this lexical analyzer does the matching and dumping of the matched tree fragments. The parser maintains information about its current path location in the XML document and the contents of the set of XML nodes that it is currently processing. For exact-match or prefix-match queries, the query path and the query predicate are converted to the compressed-domain equivalents. During parsing of the compressed XML document, when the parser detects that the current path matches the query path, and that the compressed data value matches the compressed

```
<!- student.xml -->
<STUDENT rollno = "604100418">
    <NAME>Edgar Codd</NAME>
    <YEAR>2000</YEAR>
    <PROG>Master of Science</PROG>
    <DEPT name = "Computer_Science">
</STUDENT>
```

XML Compression. Figure 2. Fragment of student database.

```
<!- DTD for the Student database -->
<!ELEMENT STUDENT (NAME, YEAR, PROG, DEPT)>
<!ATTLIST STUDENT rollno CDATA #REQUIRED>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT YEAR (#PCDATA)>
<!ELEMENT PROG (#PCDATA)>
<!ELEMENT DEPT EMPTY>
<!ATTLIST DEPT name (Computer_Science
    | Electrical_Engineering
        ⋮
    | Physics | Chemistry)
>
```

XML Compression. Figure 3. DTD of student database.

```
T0 A0 nahuff(604100418)
    T1 nahuff(Edgar Codd) /
    T2 nahuff(2000) /
    T3 nahuff(Master of Science) /
    T4A1 enum(Computer_Science) /
/
```

XML Compression. Figure 4. Abstract view of compressed XGrind database.

query predicate, it outputs the matched XML fragment. An interesting side-effect is that the matching is more efficient in the compressed domain as compared to the original domain, since the number of bytes to be processed have considerably decreased.

XPRESS. While maintaining the homomorphic feature of XGrind, XPRESS [7] significantly extends its scope by supporting both path expressions and range queries (on numeric element types) directly on the compressed data. Here, instead of representing the tag of each element with a single identifier, the element label path is encoded as a distinct interval in [0.0, 1.0). The specific process, called reverse arithmetic encoding, is as follows: First, the entire interval [0.0, 1.0) is partitioned into disjoint sub-intervals, one for each distinct element. The size of the interval is proportional to the normalized frequency of the element in the data. In the second step, these element intervals are reduced by encoding the path leading to this element in a depth-first tree traversal from the root. An example from [7] is shown in Table 1, where the element intervals are computed from the first partitioning, and the corresponding reduced intervals by following the path labels from the root.

XML Compression. Table 1. XPRESS interval scheme

Element tag	Path label	Element interval	Path interval
book	*book*	[0.0, 0.1)	[0.0, 0.1)
section	*book.section*	[0.3, 0.6)	[0.3, 0.33)
subsection	*book.section.subsection*	[0.6, 0.9)	[0.69, 0.699)

For each node in the tree, the associated interval is incrementally computed from the parent node. The intervals generated by reverse arithmetic encoding guarantee that *"If a path P is represented by the interval I, then all intervals for suffixes of P contain I."* Therefore, the interval for *subsection* which is [0.6, 0.9) contains the interval [0.69, 0.699) for the path *book.section.subsection*. Another feature of XPRESS is that it infers the data types of elements and for those that turn out to be numbers over large domains, the values are compressed by first converting them to binary and then using differential encoding, instead of the default string encoding. Recently, XPRESS has been extended in [8] to handle updates such as insertions or deletions of XML fragments.

Finally, an index-based compression approach that improves on both the compression ratio and the querry processing speed has been recently proposed in [3].

Key Applications

Data archiving, data exchange, query processing.

Cross-references

- ▶ Data Compression in Sensor Networks
- ▶ Indexing Compressed Text
- ▶ Lossless Data Compression
- ▶ Managing Compressed Structured Text
- ▶ Text Compression
- ▶ XML
- ▶ XPath/XQuery

Recommended Reading

1. Arion A., Bonifati A., Manolescu I., and Pugliese A. XQueC: a query-conscious compressed XML database. ACM Trans. Internet Techn., 7(2):1–35, 2007.
2. Cheney J. Compressing XML with multiplexed hierarchical PPM models. In Proc. Data Compression Conference, 2001, pp. 163–172.
3. Ferragina P., Luccio F., Manzini G., and Muthukrishnan M. Compressing and Searching XML Data Via Two Zips. In Proc. 15th Int. World Wide Web Conference, 2006, pp. 751–760.
4. Girardot M. and Sundaresan N. Millau: an encoding format for efficient representation and exchange of XML over the Web. In Proc. 9th Int. World Wide Web Conference, 2000.
5. Liefke H. and Suciu D. An extensible compressor for XML data. ACM SIGMOD Rec., 29(1):57–62, 2000.
6. Liefke H. and Suciu D. XMill: an efficent compressor for XML data. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2000, pp. 153–164.
7. Min J.K., Park M., and Chung C. XPRESS: a queriable compression for XML data. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 122–133.
8. Min J.K., Park M., and Chung C. XPRESS: a compressor for effective archiving, retrieval, and update of XML documents. ACM Trans. Internet Techn., 6(3):223–258, 2006.
9. Tolani P. and Haritsa J. XGRIND: a query-friendly XML compressor. In Proc. 18th Int. Conf. on Data Engineering, 2002, pp. 225–235.
10. Ziv J. and Lempel A. A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory, 23(3):337–343, 1977.
11. www.dbxml.com
12. www.ebi.ac.uk
13. www.ictcompress.com
14. www.ictcompress.com/xml.html
15. www.xmlzip.com

XML Data Dependencies

► XML Integrity Constraints

XML Data Integration

► XML Information Integration

XML Database

► XML Storage

XML Database Design

► Semi-structured database design

XML Database System

► XQuery Processors

XML Document

Michael Rys
Microsoft Corporation, Sammamish, WA, USA

Synonyms

XML document

Definition

An XML document is a text document that follows the W3C XML recommendation [3].

Key Points

An XML document is considered well-formed, if it satisfies the XML 1.0 well-formedness constraints of having exactly one top-level XML element, balanced begin- and end-tags, and all the other rules outlined in [3].

It is considered valid, if it validates against a schema language such as the XML DTDs [3] or XML Schema [1].

An XML document is represented as an XML document information item in the XML Information Set [2] and an XML document node in the XPath and XQuery data model [4].

Cross-references

► XML
► XML Attribute
► XML Element

Recommended Reading

1. Schema Part 0: Primer, latest edition available online at: http://www.w3.org/TR/xmlschema-0/.
2. XML 1.0 Information Set, latest edition available online at: http://www.w3.org/TR/xml-infoset.

3. XML 1.0 Recommendation, latest edition available online at: http://www.w3.org/TR/xml.
4. XQuery 1.0 and XPath 2.0 Data Model (XDM), latest edition available online at: http://www.w3.org/TR/xpath-datamodel/.

XML Element

Michael Rys
Microsoft Corporation, Sammamish, WA, USA

Synonyms
XML element

Definition
An XML element is a markup element in the W3C XML recommendation [3] that consists of a beginning markup tag and an end tag with optional content and optional attributes.

Key Points
An XML element is a markup element in the W3C XML recommendation [3] that consists of a beginning markup tag and an end tag.

It has to have balanced begin- and end-tags as in this example

<element>content</element>.

XML elements can be nested as in

<e1>some optional text <e2>nested element </e2> some optional text</e1>

They can have XML attributes as in

<element attribute="value">content</element>

And they can be empty, meaning have no content as in

<element/>.

The structure of elements is free within the grammatical rules of [3], but elements can be constrained and typed by schema languages such as XML DTDs [3] or XML Schema [1].

An XML element is represented as an XML element information item in the XML Information Set [2] and an XML element node in the XPath and XQuery data model [4].

Cross-references
► XML
► XML Attribute
► XML Document

Recommended Reading
1. Schema Part 0: Primer, latest edition available online at: http://www.w3.org/TR/xmlschema-0/.
2. XML 1.0 Information Set, latest edition available online at: http://www.w3.org/TR/xml-infoset.
3. XML 1.0 Recommendation, latest edition available online at: http://www.w3.org/TR/xml.
4. XQuery 1.0 and XPath 2.0 Data Model (XDM), latest edition available online at: http://www.w3.org/TR/xpath-datamodel/.

XML Enterprise Information Integration

► XML Information Integration

XML Export

► XML Publishing

XML Filtering

► XML Publish/Subscribe

XML Indexing

Xin Luna Dong, Divesh Srivastava
AT&T Labs–Research, Florham Park, NJ, USA

Definition
XML employs an ordered, tree-structured model for representing data. Queries in XML languages like XQuery employ twig queries to match relevant portions of data in an XML database. An XML Index is a data structure that is used to efficiently look up all matches of a fragment of the twig query, where some of the twig query fragment nodes may have been mapped to specific nodes in the XML database.

Historical Background

XML path indexing is related to the problem of join indexing in relational database systems [15] and path indexing in object-oriented database systems (see, e.g., [1,9]). These index structures assume that the schema is homogeneous and known; these assumptions do not hold in general for XML data. The DataGuide [7] was the first path index designed specifically for XML data, where the schema may be heterogeneous and may not even be known.

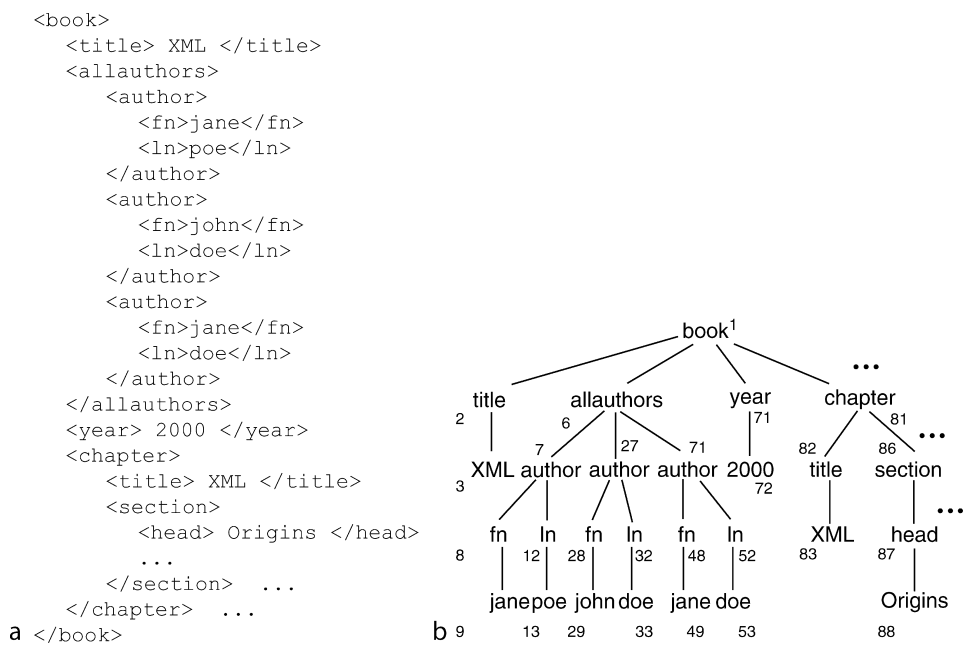
Foundations

Notation

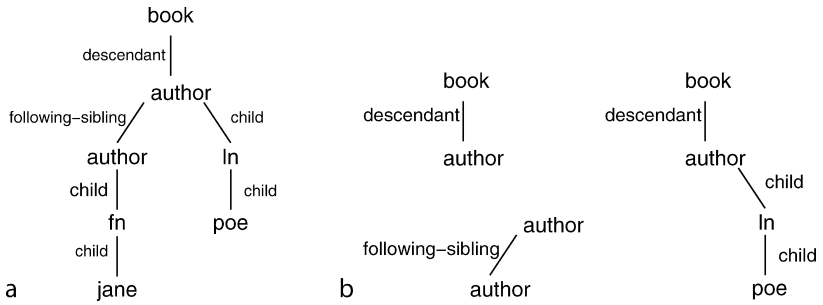
An XML document d is a rooted, ordered, node-labeled tree, where (i) each node corresponds to an XML element, an XML attribute, or a value; and (ii) each edge corresponds to an element-subelement, element-attribute, element-value, or an attribute-value relationship. Non-leaf nodes in d correspond to XML elements and attributes, and are labeled by the element tags or attribute names, while leaf nodes in d correspond to values. For the example XML document of Fig. 1a, its tree representation is shown in Fig. 1b. Each node is associated with a unique number referred to as its Id, as depicted in Fig. 1b. An XML database D is a set of XML documents.

Queries in XML languages like XPath and XQuery make fundamental use of twig queries to match relevant portions of data in an XML database. A twig query Q is a node- and edge-labeled tree, where (i) nodes are labeled by element tags, attribute names, or values; (ii) edges are labeled by an XPath axis step, e.g., `child`, `descendant`, or `following-sibling`. For example, the twig query in Fig. 2a corresponds to the path expression `/book[./descendant::author[./following-sibling :: author [fn =''jane'']][ln =''poe'']]`.

Given an XML database D, and a twig query Q, a match of Q in D is identified by a mapping from nodes in Q to nodes in D, such that: (i) Q's node labels (i.e., element tags, attribute-names and values) are preserved under the mapping; and (ii) Q's edge labels (i.e., XPath axis steps) are satisfied by the corresponding pair of nodes in D under the mapping. For example, the twig query of Fig. 2a matches a root book element that has a descendant author element that (i) has a child `ln` element with value `poe`; and (ii) has a following sibling `author` element that has a child `fn`



XML Indexing. Figure 1. (a) An XML database fragment. (b) XML tree (Id numbering).



XML Indexing. Figure 2. (a) Twig query. (b) Twig query fragments.

element with value `jane`. Thus, the `book` element in Fig. 1b is a match of the twig query in Fig. 2a.

An *XML Index* I is a data structure that is used to efficiently look up all matches of a *fragment* of the twig query *Q*, where some of the twig query fragment nodes may have been mapped to specific nodes in the XML database. Some fragments of the twig query of Fig. 2a are shown in Fig. 2b and include the edges `book/descendant::author` and `author/following-sibling::author`, and the path `book/descendant::author[ln ='‘poe’’]`. The matches returned by XML Index lookups on different fragments of a twig query *Q* can be "stitched together" using query processing algorithms to compute matches to *Q*.

The following sections describe various techniques that have been proposed in the literature to index node, edge, path and twig fragments of twig queries.

Node Indexes

When the fragment of a twig query *Q* that needs to be looked up in the index is a single node labeled by an element tag, an attribute name, or a value, a classical *inverted index* is adequate. This index is constructed by associating each element tag (or attribute name, value) with the list of all the node Ids in the XML database with that element tag (attribute name, value, respectively). For example, given the data of Fig. 1b, the list associated with the element tag author would be [7,27,47], and the list associated with the value `jane` would be [9,49].

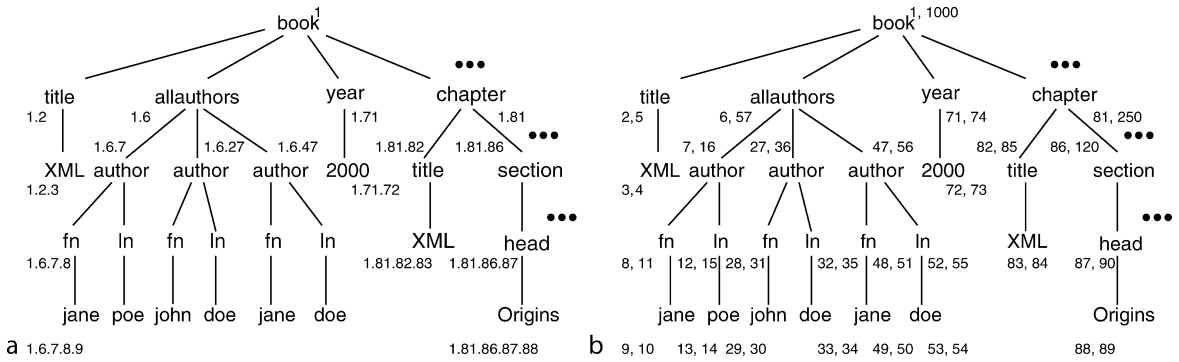
Positional Numberings, Edge Indexes

Now consider the case when the fragment of a twig query *Q* that needs to be looked up in the index is an edge labeled by an XPath axis step. For example, find all `author` nodes that are `descendants` of the book node with Id = 1. As another example, find all `author` nodes that are `following-siblings` of the `author` node with Id = 7.

Using Node Ids A simple solution is to use an inverted index that associates each ($node_1$ label, $node_2$ label, XPath axis) triple to the list of all pairs of XML database node Ids that satisfy the specified node labels and axis relationship. For example, given the data of Fig. 1b, the list associated with the triple (`book, author, descendant`) would be [(1,7),(1,27),(1,47)], and the list associated with the triple (`author, author, following-sibling`) would be [(7,27),(7,47),(27,47)].

In general, this inverted index could be much larger than the number of nodes in the original XML database, especially for XPath axes such as `descendant, following-sibling` and `following`. To overcome this limitation, more sophisticated approaches are required. A popular approach has been to (i) use a positional numbering system to identify nodes in an XML database; and (ii) demonstrate that each XPath axis step corresponds to a predicate on the positional numbers of the corresponding nodes. Two such approaches, using Dewey numbering and using Interval numbering, are described next.

Using Dewey Numbering An elegant solution for edge indexing is to associate each XML node *n* with its `DeweyId`, proposed by [14], and obtained as follows: (i) associate each node with a numeric `Id` ensuring that sibling nodes are given increasing numbers in a left-to-right order; and (ii) the `DeweyId` of a node is obtained by concatenating the `Ids` of all nodes along



XML Indexing. Figure 3. (a) Dewey numbering. (b) Interval numbering.

the path from the root node of n's XML document to n itself (The similarity with the Dewey Decimal System of library classification is the reason for its name). Figure 3a shows the DeweyIds of some nodes in the XML tree, using the numeric Ids associated with those nodes in Fig. 1b. For example, the fn node with Id = 8 has DeweyId = 1.6.7.8.

DeweyIds can be used to easily find matches to various XPath axis steps. In particular, node n_2 is a descendant of node n_1 if and only if (i) n_1.DeweyId is a prefix of n_2.DeweyId. For example, in Fig. 3a, the jane node with DeweyId = 1.6.7.8.9 is a descendant of the author node with DeweyId = 1.6.7. Similarly, node n_2 is a child of node n_1 if and only if (i) n_2 is a descendant of n_1; and (ii) n_2's DeweyId extends n_1's DeweyId by one Id. By maintaining DeweyIds of nodes in classical trie data structures, various XPath axis lookups can be done efficiently.

The main limitation of DeweyIds is that their size depends on the depth of the XML tree, and can get quite large. This limitation is overcome by using Interval numbering, described next.

Using Interval Numbering The position of an XML node n is represented as a 3-tuple: (LeftPos, RightPos, PLeftPos), where (i) numbers are generated in an increasing order by visiting each tree node twice in a left-to-right, depth-first traversal; n.LeftPos is the number generated before visiting any node in n's subtree and n.RightPos is the number generated after visiting every node in n's subtree; and (ii) n.PLeftPos is the LeftPos of n's parent node (0 if n is the root node of DocId). Figure 3b depicts the LeftPos and RightPos numbers of each node in the XML document.

It can be seen that each XPath axis step between a pair of XML database nodes can be tested using a conjunction of equality and inequality predicates on the components of the 3-tuple. In particular, node n_2 is a descendant of node n_1 if and only if: (i) n_1.LeftPos < n_2.LeftPos; and (ii) n_1.RightPos > n_2.RightPos. An element n_2 is a child of an element n_1 if and only if n_1.LeftPos = n_2.PLeftPos. An element n_2 is a following-sibling of an element n_1 if and only if: (i) n_1.RightPos < n_2.LeftPos; and (ii) n_1.PLeftPos = n_2.PLeftPos. For example, in Fig. 3b, the jane node with interval number (9,10,8) is a descendant of the author node with interval number (7,16,6).

Thus, the set of 3-tuples corresponding to the interval numbering of nodes of an XML database can be indexed using a 3-dimensional spatial index such as an R-tree, and the different XPath axis steps correspond to different regions within the 3-dimensional space. Variations of this approach have been considered in, e.g., [10,2,8].

Note that for both Dewey numbering and Interval numbering, one would need to leave gaps between numbers to allow for insertions of new nodes in the XML database [5].

Path Indexes

When the fragment of a twig query Q that needs to be looked up in the index is a subpath of a root-to-leaf path in Q, where some (possibly none) of the nodes have been mapped to specific nodes in the XML database, XML path indexes are very useful. The works in the literature have primarily focused on the case where

each edge in the subpath of Q is labeled by child, i.e., all matches are subpaths of root-to-leaf paths in an XML document. For example, a path index can be used to efficiently look up all matches to the path fragment `author[ln='poe']` of the twig query depicted in Fig. 2a.

A framework by Chen et al. [3] is described next, which covers most existing XML path index structures, and solves the BoundIndex problem.

Problem BoundIndex: Given an XML database D, a subpath query P with k node labels and each edge labeled by `child`, and a specific database node id n, return all k-tuples $(n_1,...,n_k)$ that identify matches of P in D, rooted at node n.

The framework of [3] requires each node in an XML document to be associated with a unique numeric identifier; this could be, e.g., the `Id` of the node in Fig. 1b. To create a path index, [3] conceptually separates a path in an XML document into two parts: (i) a *schema path*, which consists solely of schema components, i.e., element tags and attribute names; and (ii) a *leaf value* as a string if the path reaches a leaf. Schema paths can be dictionary-encoded using special characters (whose lengths depend on the dictionary size) as designators for the schema components. Most of the works in the literature have focused on indexing XML schema paths (see, e.g., [7,12,4]). Notable exceptions that also consider indexing data values at the leaves of paths include [6,17,3].

In order to solve the BoundIndex problem, one needs to explicitly represent paths that are arbitrary subpaths of the root-to-leaf paths, and associate each such path with the node at which the subpath is rooted. Such a relational representation of *all* the paths in an XML database is (`HeadId, SchemaPath, LeafValue, IdList`), where `HeadId` is the id of the start of the path, and `IdList` is the list of all node identifiers along the schema path, except for the `HeadId`. As an example, a fragment of the 4-ary relational representation of the data tree of Fig. 1b is given in Table 1; element tags have been encoded using boldface characters as designators, based on the first character of the tag, except for `allauthors` which uses **U** as its designator.

Given the 4-ary relational representation of XML database D, each index in the family of indexes: (i) stores a subset of all possible `SchemaPaths` in D; (ii) stores a sublist of IdList; and (iii) indexes a subset of the columns `HeadId, SchemaPath`, and `LeafValue`.

XML Indexing. Table 1. The 4-ary relation for path indexes

HeadId	SchemaPath	LeafValue	IdList
1	**B**	null	[]
1	**BT**	null	[2]
1	**BT**	XML	[2]
1	**BU**	null	[6]
1	**BUA**	null	[6,7]
1	**BUAF**	null	[6,7,8]
1	**BUAF**	jane	[6,7,8]
1	**BUAL**	null	[6,7,12]
1	**BUAL**	poe	[6,7,12]
	...		
6	**U**	null	[]
6	**UA**	null	[7]
6	**UAF**	null	[7,8]
6	**UAF**	jane	[7,8]
6	**UAL**	null	[7,12]
6	**UAL**	poe	[7,12]
	...		

Given a query, the index structure probes the indexed columns in (iii) and returns the sublist of `IdList` stored in the index entries. Many existing indexes fit in this framework, as summarized in Table 2. For example, the DataGuide [7] returns the last Id of the `IdList` for every root-to-leaf prefix path. Similarly, IndexFabric [6] returns the `Id` of either the root or the leaf element (first or last `Id` in `IdList`), given a root-to-leaf path and the value of the leaf element. Finally, the `DATAPATHS` index is a regular B^+-tree index on the concatenation of `HeadId, LeafValue` and the reverse of `SchemaPath` (or the concatenation `LeafValue·HeadId·ReverseSchemaPath`), where the `SchemaPath` column stores all subpaths of root-to-leaf paths, and the complete `IdList` is returned; the `DATAPATHS` index can solve the BoundIndex problem in one index lookup.

Twig Indexes

ViST [16] and PRIX [13] are techniques that encode XML documents as sequences, and perform subsequence matching to look up all matches to twig queries.

XML Indexing. Table 2. Members of family of path indexes

Index	Subset of `SchemaPath`	Sublist of `IdList`	Indexed Columns
Value [11]	paths of length 1	only last Id	`SchemaPath, LeafValue`
Forward link [11]	paths of length 1	only last Id	`HeadId, SchemaPath`
DataGuide [7]	root-to-leaf path prefixes	only last Id	`SchemaPath`
Index Fabric [6]	root-to-leaf paths	only first or last Id	`reverse SchemaPath, LeafValue`
ROOTPATHS [3]	root-to-leaf path prefixes	full IdList	`LeafValue, reverse SchemaPath,`
DATAPATHS [3]	all paths	full IdList	`LeafValue, HeadId, reverse SchemaPath`

Key Applications

XML Indexing is important for efficient XML query processing, both in relational implementations of XML databases and in native XML databases.

Future Directions

It is important to investigate XML Path Indexes for the case of path queries with edge labels other than `child`, especially when different edges on a query path have different edge labels. Another extension worth investigating is to identify classes of twig queries that admit efficient XML Twig Indexes.

Data Sets

University of Washington XML Repository: http://www.cs.washington.edu/research/xmldatasets/.

Cross-references

- ▶ XML Document
- ▶ XML Tree Pattern, XML Twig Query
- ▶ XPath/XQuery
- ▶ XQuery Processors

Recommended Reading

1. Bertino E. and Kim W. Indexing techniques for queries on nested objects. IEEE Trans. Knowledge and Data Eng., 1 (2):196–214, 1989.
2. Bruno N., Koudas N., and Srivastava D. Holistic twig joins: optimal XML pattern matching. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 310–321.
3. Chen Z., Gehrke J., Korn F., Koudas N., Shanmugasundaram J., and Srivastava D. Index structures for matching XML twigs using relational query processors. Data Knowl. Eng., 60 (2):283–302, 2007.
4. Chung C.-W., Min J.-K., and Shim K. APEX: an adaptive path index for XML data. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 121–132.
5. Cohen E., Kaplan H., and Milo T. Labeling dynamic XML trees. In Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 2002, pp. 271–281.
6. Cooper B.F., Sample N., Franklin M.J., Hjaltason G.R., and Shadmon M. A fast index for semistructured data. In Proc. 27th Int. Conf. on Very Large Data Bases, 2001, pp. 341–350.
7. Goldman R. and Widom J. DataGuides: enabling query formulation and optimization in semistructured databases. In Proc. 23th Int. Conf. on Very Large Data Bases, 1997, pp. 436–445.
8. Grust T. Accelerating XPath location steps. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 109–120.
9. Kemper A. and Moerkotte G. Access support in object bases. ACM SIGMOD Rec., 19(2):364–374, 1990.
10. Kha D.D., Yoshikawa M., and Uemura S. An XML indexing structure with relative region coordinate. In Proc. 17th Int. Conf. on Data Engineering, 2001, pp. 313–320.
11. McHugh J. and Widom J. Query optimization for XML. In Proc. 25th Int. Conf. on Very Large Data Bases, 1999, pp. 315–326.
12. Milo T. and Suciu D. Index structures for path expressions. In Proc. 15th Int. Conf. on Data Engineering, 1999, pp. 277–295.
13. Rao P. and Moon B. PRIX: Indexing and querying XML using Pruffer sequences. In Proc. 20th Int. Conf. on Data Engineering, 2004, p. 288.

14. Tatarinov I., Viglas S., Beyer K., Shanmugasundaram J., Shekita E., and Zhang C. Storing and querying ordered XML using a relational database system. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 204–215.
15. Valduriez P. Join indices. ACM Trans. Database Syst., 12 (2):218–246, 1987.
16. Wang H., Park S., Fan W., and Yu P. ViST: a dynamic index method for querying XML data by tree structures. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 110–121.
17. Yoshikawa M., Amagasa T., Shimura T., and Uemura S. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Trans. Internet Tech., 1(1):110–141, 2001.

XML Information Integration

Alon Halevy
Google Inc., Mountain View, CA, USA

Synonyms

XML data integration; XML enterprise information integration

Definition

Information integration systems offer uniform access to a set of autonomous and heterogeneous data sources. Sources can range from database systems and legacy systems to forms on the Web, web services and flat files. The data in the sources need not be completely structured as in relational databases. The number of sources in a information integration application can range from a handful to thousands. XML information integration systems are ones that based on an XML data model and query language.

Key Points

XML played a significant role in the development of information integration, from the research and from the commercialization perspectives. The emergence of XML fueled the desire for information integration, because it offered a common syntactic format for sharing data. Once this common syntax was in place, organizations could start thinking about sharing data with each other (or even, within the organization) and integrating data from multiple sources. Of course, XML did very little to address the semantic integration issues – sources could still share XML files whose tags were completely meaningless outside the application. However, the fact that XML is semi-structured was an advantage when modeling heterogeneous data that may have different schematic structure.

From the technical perspective, several information integration systems were developed with an XML data model at their core. Developing these systems lead to many advances in XML data management, including the development of query and update languages, languages for schema mapping, algorithms for query containment and answering queries using views, and efficient processing of XML data streaming from external sources. Every aspect of information integration systems had to be re-examined with XML in mind. Interestingly, since the push to commercialize information integration came around the same time as the emergence of XML, many of the innovations mentioned above had to be developed by start-ups on a very short fuse.

Key Applications

Some of the key applications of information integration are:

- Enterprise data management, querying across several enterprise data repositories
- Accessing multiple data sources on the web (and in particular, the deep web)
- Large scientific projects where multiple scientists are independently producing data sets
- Coordination accross mulitple government agencies

Cross-references

▶ Adaptive Query Processing
▶ Information Integration
▶ Model Management

Recommended Reading

1. Abiteboul S., Benjelloun O., and Milo T. The active XML project: an overview. VLDB J., 17(5):1019–1040, 2008.
2. Deshpande A., Ives Z., and Raman V. Adaptive Query Processing. Foundations and Trends in Databases. Now Publishers, 2007. www.nowpublishers.com/dbs.
3. Haas L. Beauty and the Beast: The theory and practice of information integration. In Proc. 11th Int. Conf. on Database Theory, 2007, pp. 28–43.
4. Halevy A.Y., Ashish N., Bitton D., Carey M.J., Draper D., Pollock J., Rosenthal A., and Sikka V. Enterprise information integration: successes, challenges and controversies. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2005, pp. 778–787.

XML Information Retrieval

► INitiative for the Evaluation of XML retrieval (INEX)

XML Integrity Constraints

Marcelo Arenas
Pontifical Catholic University of Chile, Santiago, Chile

Synonyms

XML data dependencies

Definition

An XML integrity constraint specifies a semantic restriction over the data stored in an XML document. A number of integrity constraint languages have been proposed for XML, which can be used to enforce different types of semantic restrictions. These proposals, together with some languages for specifying restrictions on the element structure of XML documents (e.g., DTD and XML Schema), are currently used to specify the schema of XML documents.

Historical Background

The problem of defining and manipulating integrity constraints is one of the oldest problems in databases. Soon after the introduction of the relational model by Codd in the '70s, researchers developed several languages for specifying integrity constraints, and studied many fundamental problems for these languages.

In the relational model, a database is viewed as a collection of relations or tables. For instance, a relational database storing information about courses in a university is shown in Fig. 1. This database consists of a time-varying part, the data about courses, and a part considered to be time independent, the *schema* of the relations, which is given by the name of the relations (*CourseInfo* and *CourseTerm*) and the names of the attributes of the relations.

Usually, the information contained in a database satisfies some *semantic* restrictions. For example, in the relation *CourseInfo* shown in Fig. 1, it is expected that only one title is associated to each course number. By providing the schema of a relation, a syntactic constraint is specified (the structure of the relation), but a semantic constraint as the one mentioned above cannot be specified. To overcome this problem, it is necessary to specify separately a set of semantic restrictions. These restrictions are called *integrity constraints*, and they are expressed by using suitable languages.

Although a number of integrity constraints languages were developed for relational databases, functional and inclusion dependencies are the ones used most often. A functional dependency is an expression of the form $X \rightarrow Y$, where X and Y are sets of attributes. A relation satisfies $X \rightarrow Y$ if for every pair of tuples t_1, t_2 in it, if t_1 and t_2 have the same values on X, then they have the same values on Y. An inclusion dependency is an expression of the form $R[X] \subseteq S[Y]$, where R and S are relation names, and X, Y are sets of attributes of the same cardinality. A relational database satisfies $R[X] \subseteq S[Y]$ if for every tuple t_1 in R, there exists a tuple t_2 in S such that the values of t_1 on X are the same as the values of t_2 on Y. For example, relation *CourseTerm* shown in Fig. 1 satisfies functional dependency $\{Number, Section\} \rightarrow Room$, since every section of a course is given in only one room, and relations *CourseInfo* and *CourseTerm* satisfy inclusion dependency $CourseTerm[Number] \subseteq CourseInfo[Number]$, since every course number mentioned in *CourseTerm* is also mentioned in *CourseInfo* (and thus every course given in some term has a name).

Integrity constraint languages have also been developed for more recent data models, such as nested relational and object-oriented databases. Functional and inclusion dependencies are also present in all these models. In fact, two subclasses of functional and inclusion dependencies, namely, keys and foreign keys, are most commonly found in practice. For the case of relational databases, a key dependency is a functional dependency of the form $X \rightarrow U$, where U is the set of attributes of a relation, and a foreign key dependency is formed by an inclusion dependency $R[X] \subseteq S[Y]$ and a key dependency $Y \rightarrow U$, where U is the set of attributes of S. For example, $Number \rightarrow Number, Title$ is a key dependency for relation *CourseInfo*, while $CourseTerm[Number] \subseteq CourseInfo[Number]$ and $Number \rightarrow Number, Title$ is a foreign key dependency. Keys and foreign keys are fundamental to conceptual database design, and are supported by

CourseInfo	Number	Title
	CSC258	Compilers
	CSC434	Data management systems

CourseTerm	Number	Section	Room
	CSC258	1	LP266
	CSC258	2	GB258
	CSC434	1	GB248

XML Integrity Constraints. Figure 1. Example of a relational database.

the SQL standard. They provide a mechanism by which objects can be uniquely identified and referenced, and they have proved useful in update anomaly prevention, query optimization and index design.

Foundations

A number of integrity constraint specifications have been proposed for XML. The hierarchical nature of XML data calls for not only *absolute constraints* that hold on an entire document, such as dependencies found in relational databases, but also *relative constraints* that only hold on sub-documents, beyond what it is encountered in traditional databases. Here, both absolute and relative functional dependencies, keys, inclusion dependencies and foreign keys are considered for XML.

In general, integrity constraints for XML are defined as restrictions on the nodes and data values reachable by following paths in some tree representation of XML documents. For example, Fig. 2 shows an XML document storing information about courses at the University of Toronto (UofT), and Fig. 3 shows a tree representation of this document. Given an XML document D, element types $\tau_1,...,\tau_n$ in D and a symbol ℓ such that ℓ is either an element type in D or an attribute in D or the reserved symbol `text()`, the string $\tau_1,...,\tau_n.\ell$ is a path in D if there exist nodes $u_1,...,u_n$ in D such that (i) each u_i is if type τ_i, (ii) each u_{i+1} is a child of u_i, (iii) if ℓ is an element type, then there exists a node u of type ℓ that is a child of u_n, (iv) if ℓ is an attribute, then ℓ is defined for u_n, and (v) if ℓ = `text()`, then u_n has a child of type `PCDATA`. For example, `student.taking`, `student.taking.cno`, `UofT.course.title` and `UofT.course.title.text()` are all paths in the XML document shown in Figs. 2 and 3.

Given a node u in an XML document D and a path p in D, $reach(u, p)$ is defined to be the set of all nodes and values reachable by following p from u in D. Furthermore, if P is a regular expression over an alphabet consisting of the reserved symbol `text()` and the element types and attributes in an XML document D, then a node v is reachable from a node u in D by following P, if there exists a string p in the regular language defined by P such that $v \in reach(u, p)$. The set of all such nodes is denoted by $reach(u, P)$. For example, for the XML tree shown in Fig. 3, $reach(u_1,$ `UofT.student`$) = \{u_2\}$, $reach(u_1,$ `UofT.student.name`$) =$ {`J. Smith`}, $reach(u_1,$ `UofT.course.title.text()`$) =$ {`Compilers`} and

$reach(u_1,$ `UofT.(student.taking+course).cno`$) =$ {`CSC258,CSC309`}.

Keys and Functional Dependencies for XML

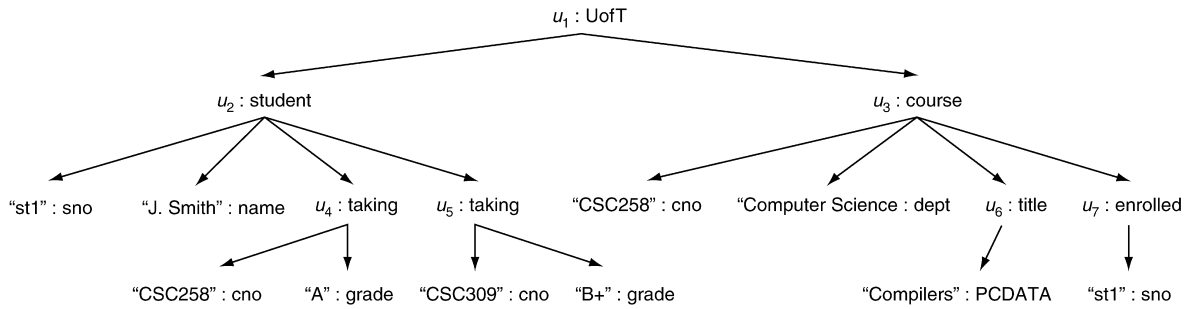
Absolute keys for XML were first considered by Fan and Simeón [7]. Let D be an XML document and τ an element type in D. Then $ext(\tau)$ is defined to be the set of all nodes of D of type τ. For example, if D is the XML tree shown in Fig. 3, then $ext($`taking`$) = \{u_4, u_5\}$. Moreover, given a node υ and a list of attributes $X = [a_1,...,a_k]$ that are defined for υ, $\upsilon[X]$ is defined as $[\upsilon.a_1,...,\upsilon.a_k]$, where $\upsilon.a_i$ is the value of attribute a_i for node υ. For example, u_2[`sno`, `name`] is [`st1`, `J. Smith`] in the XML tree shown in Fig. 3.

Absolute keys for XML are defined as follows [6,7]. An absolute key is an expression of the form $\tau[X] \rightarrow \tau$, where τ is an element type and X is a nonempty set of attributes defined for τ. An XML document D satisfies this constraint, denoted by $D \vDash \tau[X] \rightarrow \tau$, if for every $v_1, v_2 \in ext(\tau)$, if $v_1[X] = v_2[X]$, then $v_1 = v_2$. Thus, $\tau[X] \rightarrow \tau$ says that the X-attribute values of a τ-element uniquely identify the element in $ext(\tau)$. Notice that two notions of equality are used to define keys: value equality is assumed when comparing attributes, and node identity is used when comparing elements. The same symbol ' = ' is used for both, as it will never lead to ambiguity.

The following are typical keys for the XML document shown in Fig. 2: `student[sno]` $\rightarrow$ `student` and `course[cno]` $\rightarrow$ `course`. The first constraint says that student number (`sno`) is an identifier for students, and the second constraint says that course number

```
<UofT>
  <student sno = "st1" name = "J. Smith">
     <taking cno = "CSC258" grade = "A"/>
     <taking cno = "CSC309" grade = "B+"/>
  </student>
  <course cno = "CSC258" dept = "Computer Science">
     <title> Compilers </title>
     <enrolled sno = "st1"/>
  </course>
</UofT>
```

XML Integrity Constraints. Figure 2. Example of an XML document.



XML Integrity Constraints. Figure 3. Tree representation of the XML document shown in Fig. 2.

(cno) is an identifier for courses. It should be noticed that if courses in different departments can have the same course number, then key course[cno] → course has to be replaced by course[cno, dept] → course.

Since XML documents are hierarchically structured, users may be interested in the entire document as well as in its sub-documents. The latter give rise to relative keys, that only hold on certain sub-documents. The extension of Fan and Simeón's language to the relative case was done by Arenas et al. in [8]. This language is introduced below. Notation $v_1 \prec v_2$ is used when v_1 and v_2 are two nodes in an XML document and v_2 is a descendant of v_1. For example, $u_1 \prec u_4$ and $u_3 \prec u_7$ in the XML tree shown in Fig. 3.

A relative key is an expression of the form $\tau(\tau_1[X] \rightarrow \tau_1)$, where τ, τ_1 are element types and X is a nonempty set of attributes that is defined for τ_1. This constraint says that relative to each node v of type τ, the set of attributes X is a key for all the τ_1-nodes that are descendants of v. That is, an XML document D satisfies this constraint, denoted by $D \models \tau(\tau_1[X] \rightarrow \tau_1)$, if for every $v \in ext(\tau)$, and for every v_1, $v_2 \in ext(\tau_1)$ such that $v \prec v_1$ and $v \prec v_2$, if $v_1[X] = v_2[X]$, then $v_1 = v_2$. For example, the following is a typical relative key for the XML document shown in Fig. 2: course (enrolled[sno] → enrolled). This constraint says that relative to the elements of type course, sno is an identifier for the elements of type enrolled. Thus, this constraint states that every student can enroll at most once in a given course. It should be noticed that the previous constraint cannot be replaced by the absolute key enrolled[sno] → enrolled, since this states that every student can enroll in at most one course. It should also be noticed that absolute keys are a special case of relative keys when τ is taken to be the type of the root of the XML documents. For example, for the case of the XML document shown in Fig. 2, absolute key student [sno] → student is equivalent to relative key UofT (student[sno] → student).

A more powerful language for expressing XML keys was introduced by Buneman et al. [4,3]. This language allows the definition of absolute and relative keys. More precisely, an absolute key is an expression of the form $(P, \{Q_1,...,Q_n\})$, where P, $Q_1,...,Q_n$ are regular expressions. An XML document D satisfies this key if for every pair of nodes u, $v \in reach(root, P)$, where *root* is the node identifier of the root of D, if $reach(u, Q_i) \cap reach(v, Q_i) \neq \emptyset$, for every $i \in [1, n]$, then u and v are the same node. For example, a key dependency can be used to express that name is an identifier

for students in the University of Toronto database: (UofT.student, {name}). Notice that if a nested structure is used in this database to distinguish first names from last names:

```
<UofT>
  <student sno="st1">
    <name>
      <first> John </first>
      <last> Smith </last>
    </name>
  ...
</UofT>
```

then to characterize name as an identifier for students, two paths are included in the right-hand side of the key dependency: (UofT.student, {name.first.text(), name.last.text()}).

In [4,3], Buneman et al. defined a relative key as a pair of the form (P', K), where P' is a regular expression and K is an absolute key of the form $(P, \{Q_1,...,Q_n\})$. An XML document satisfies this key if every node reached from the root by following a path in P' satisfies K, that is, for every $u \in reach(root, P')$ and for every $v_1, v_2 \in reach(u, P)$, if $reach(v_1, Q_i) \cap reach(v_2, Q_i) \neq \emptyset$, for every $i \in [1, n]$, then v_1 and v_2 are the same node. For example, a relative key constraint can be used to express that a student cannot take the same course twice in the XML database shown in Fig. 2: (UofT.student, (taking, {cno})). This key dependency is relative since two distinct students can take the same course. It should be noticed that every key $\tau(\tau_1[a_1,...,a_k] \rightarrow \tau_1)$ in Arenas et al.'s language [8] can be represented as $(\Sigma^*.\tau, (\Sigma^*.\tau_1, \{a_1,...,a_k\}))$ in Buneman et al.'s language [4,3], where Σ is the alphabet consisting of all the element types in an XML document.

In [2], Arenas and Libkin introduced a functional dependency language for XML. In this language, a functional dependency over an XML document D is an expression of the form $X \rightarrow p$, where $X \cup \{p\}$ is a set of paths in D. To define the notion of satisfaction for functional dependencies, Arenas and Libkin [2] used a relational representation of XML documents. Given an XML document D, let $paths(D)$ be the set of all paths in D. Then a tree tuple over D is a mapping t that assigns to each path $p \in paths(D)$ either a node identifier or a data value or the null value $\perp$, in a way that is consistent with the tree representation of D. Formally, if $p \in paths(D)$, the last symbol of p is an element type τ and $t(p) \neq \perp$, then (i) $t(p) = u$, where u is a node identifier in D of type τ; (ii) if p' is a prefix of p, then $t(p')$ is a node identifier that lies on the path from the root to u in D; (iii) if a is an attribute defined for u in D, then $t(p.a) = u.a$; and (iv) if p.text() is a path in D, then there is a child s of u of type PCDATA such that $t(p.\texttt{text()}) = s$. A tree tuple is maximal if it cannot be extended to another one by changing some nulls to either node identifiers or data values. The set of maximal tree tuples in D is denoted by $tuples(D)$. Then functional dependency $\{p_1,...,p_m\} \rightarrow p$ is true in D if for every pair $t_1, t_2 \in tuples(D)$, whenever $t_1(p_i) = t_2(p_i) \neq \perp$ for all $i \leq n$, then $t_1(p) = t_2(p)$ holds.

For example, let D be the XML document shown in Figs. 2 and 3. In this database, there is at most one name associated with each student number, which can be represented by means of functional dependency UofT.student.sno $\rightarrow$ UofT.student.name. XML document D satisfies this constraint, which can be proved formally by constructing $tuples(D)$. Table 1 shows the two tree tuples contained in this set. These tuples satisfy UofT.student.sno $\rightarrow$ UofT.student.name since they have the same values in UofT.student.sno and also in UofT.student.name.

Arenas and Libkin's language [2] can also be used to express relative functional dependencies. For example, in the XML document shown in Fig. 2, every student has at most one grade in each course (which is the final grade for the course). This relative functional dependency can be expressed as {UofT.student, UofT.student.taking.cno} $\rightarrow$ UofT.student.taking.grade, whose satisfaction can be checked by considering again the tree tuples shown in Table 1.

Inclusion Dependencies and Foreign Keys for XML

One of the first XML integrity constraint languages was proposed by Abiteboul and Vianu [1]. They considered inclusion dependencies of the form $P \subseteq Q$, where P and Q are regular expressions. An XML document D rooted at u satisfies this constraint if $reach(u, P) \subseteq reach(u,Q)$. For example, in the database shown in Fig. 2, the following constraint expresses that the set of courses taken by each student is a subset of the set of courses given by the university: UofT.student.taking.cno $\subseteq$ UofT.course.cno. An inclusion constraint $P \subseteq Q$ where P are Q are paths, like in the previous example, is called a path constraint [1]. In [5], Buneman et al. introduced a more powerful path constraint language. Given an XML document D and paths p_1, p_2, p_3 in D, in this language a constraint is an

expression of either the forward form $\forall x \forall y(x \in reach(root, p_1) \wedge y \in reach(x, p_2) \rightarrow y \in reach(x, p_3))$, where *root* represents the root of the XML document, or the backward form $\forall x \forall y(x \in reach(root, p_1) \wedge y \in reach(x, p_2) \rightarrow x \in reach(y, p_3))$. This language can be used to express relative inclusion dependencies. For example, if the document shown in Fig. 2 is extended to store information about students and courses in many universities, ⟨UofT⟩ is replaced by ⟨university⟩:

```
<db>
  <university name="UofT"> ...
  </university>
  <university name="UCLA"> ...
  </university>
</db>
```

Then the following dependency in Buneman et al.'s language [5] can be used to state that for each university, the set of courses taken by each one of its students is a subset of the set of courses given by that university:

$$\forall x \forall y(x \in reach(root, \text{db.university}) \wedge y \in reach(x, \text{student.taking.cno}) \rightarrow y \in reach(x, \text{course.cno})).$$

This constraint is relative to each university and, thus, it cannot be expressed by using Abiteboul and Vianu's path constraints [1], as this language can only express constraints on the entire document. By using Abiteboul and Vianu's approach, it can only be said that if a student is taking a course, then this course is given in some university: db.university.student.taking.cno ⊆ db.university.course.cno.

A language for expressing absolute foreign keys for XML was proposed by Fan and Libkin in [6]. In this language, an absolute foreign key is an expression of the form $\tau_1[X] \subseteq_{FK} \tau_2[Y]$, where τ_1, τ_2 are element types, and X, Y are nonempty lists of attributes defined for τ_1 and τ_2, respectively, and $|X| = |Y|$. This constraint is satisfied by an XML document D, denoted by $D \vDash \tau_1[X] \subseteq_{FK} \tau_2[Y]$, if $D \vDash \tau_2[Y] \rightarrow \tau_2$, and in addition for every $v_1 \in ext(\tau_1)$, there exists $v_2 \in ext(\tau_2)$ such that $v_1[X] = v_2[Y]$. The extension of Fan and Libkin's proposal to the relative case was done by Arenas et al. in [8]. In this proposal, a relative foreign key is an expression of the form $\tau(\tau_1[X] \subseteq_{FK} \tau_2[Y])$, where τ, τ_1, τ_2 are element types, X and Y are nonempty lists of attributes defined for τ_1 and τ_2, respectively, and $|X| = |Y|$. It indicates that for each node v of type τ, X is a foreign key of descendants of v of type τ_1 that references a key Y of τ_2-descendants of v. That is, an XML document D satisfies this constraint, denoted by $D \vDash \tau(\tau_1[X] \subseteq_{FK} \tau_2[Y])$, if $D \vDash \tau(\tau_2[Y] \rightarrow \tau_2)$ and for every $v \in ext(\tau)$ and every $v_1 \in ext(\tau_1)$ such that $v \prec v_1$, there exists $v_2 \in ext(\tau_2)$ such that $v \prec v_2$ and $v_1[X] = v_2[Y]$.

Key Applications

XML integrity constraints are essential to schema design, query optimization, efficient storage, index

XML Integrity Constraints. Table 1. Maximal tree tuples in the XML tree shown in Fig. 3

Path	t_1	t_2
UofT	u_1	u_1
UofT.student	u_2	u_2
UofT.student.sno	st1	st1
UofT.student.name	J. Smith	J. Smith
UofT.student.taking	u_4	u_5
UofT.student.taking.cno	CSC258	CSC309
UofT.student.taking.grade	A	B+
UofT.course	u_3	u_3
UofT.course.cno	CSC258	CSC258
UofT.course.dept	Computer Science	Computer Science
UofT.course.title	u_6	u_6
UofT.course.title.text()	Compilers	Compilers
UofT.course.enrolled	u_7	u_7
UofT.course.enrolled.sno	st1	st1

design, data integration, and in data transformations between XML and relational databases.

Cross-references

- ▶ Database Dependencies
- ▶ Functional Dependency
- ▶ Key

Recommended Reading

1. Abiteboul S. and Vianu V. Regular path queries with constraints. In Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1997, pp. 122–133.
2. Arenas M. and Libkin L. A normal form for XML documents. In Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2002, pp. 85–96.
3. Buneman P., Davidson S., Fan W., Hara C., and Tan W.C. Reasoning about keys for XML. In Proc. 8th Int. Workshop on Database Programming Languages, 2001, pp. 133–148.
4. Buneman P., Davidson S., Fan W., Hara C., and Tan W.C. Keys for XML. In Proc. 10th Int. World Wide Web Conference, 2001, pp. 201–210.
5. Buneman P., Fan W., and Weinstein S. Path constraints in semistructured and structured databases. In Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1998, pp. 129–138.
6. Fan W. and Libkin L. On XML integrity constraints in the presence of DTDs. J. ACM, 49(3):368–406, 2002.
7. Fan W. and Siméon J. Integrity constraints for XML. In Proc. 19th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2000, pp. 23–34.
8. Marcelo Arenas, Wenfei Fan and Leonid Libkin. On the Complexity of Verifying Consistency of XML Specifications. SIAM Journal on Computing, 38(3):841–880, 2008.

XML Message Brokering

- ▶ XML Publish/Subscribe

XML Metadata Interchange

Michael Weiss
Carleton University, Ottawa, ON, Canada

Synonyms

XMI

Definition

XMI (XML Metadata Interchange) is an XML-based integration framework for the exchange of models, and, more generally, any kind of XML data. XMI is used in the integration of tools, repositories, applications, and data warehouses. The framework defines rules for generating XML schemas from a metamodel based on the Metaobject Facility (MOF). XMI is most frequently used as an interchange format for UML, although it can be used with any MOF-compliant language.

Key Points

The motivation for introducing XMI was the need to provide a standard way through which UML tools could exchange UML models. XMI produced by one tool can generally be imported by another tool, which allows exchange of models among tools by different vendors, or the exchange of models with other types of tools upstream or downstream the tool chain. As stated above, XMI is not limited to mapping UML to XML, but it provides rules to generate DTDs or XML schemas and XML documents from any MOF-compliant language. Thus, a model that conforms to some MOF-compliant metamodel can be translated to an XML document that conforms to a schema generated according to the rules of the XMI standard.

However, since XMI represents UML models in XML, XMI is more widely applicable. For example, XMI has been used for the analysis of models and for the integration of applications, both internal and with external parties. The key idea underlying XMI is to provide rules for generating schemas from a metamodel. These include the mapping of model attributes to XML elements and attributes, and model associations to containment relationships between XML elements or to links. Notably, XMI maps inheritance to composition in XML, in which elements and attributes from superclasses in the model are "copied down" into the XML document, as XML lacks inheritance.

Cross-references

- ▶ Extensible Markup Language
- ▶ Metamodel
- ▶ Meta Object Facility
- ▶ UML

Recommended Reading

1. Carlson D. Modeling XML Applications with UML. Addison-Wesley, Reading, MA, 2001.
2. Grose T., Doney G., and Brodsky S. Mastering XMI. Wiley, New York, 2002.
3. OMG, MOF 2.0/XMI Mapping, version 2.1.1, 2007, http://www.omg.org/spec/XMI/2.1.1

XML Parsing, SAX/DOM

Chengkai Li
University of Texas at Arlington, Arlington, TX, USA

Definition

XML parsing is the process of reading an XML document and providing an interface to the user application for accessing the document. An XML parser is a software apparatus that accomplishes such tasks. In addition, most XML parsers check the *well-formedness* of the XML document and many can also validate the document with respect to a DTD (Document Type Definition) or XML schema. Through the parsing interface, the user application can focus on the application logic itself, without dwelling on the tedious details of XML.

There are mainly two categories of XML programming interfaces, DOM (Document Object Model) and SAX (Simple API for XML). DOM is a tree-based interface that models an XML document as a tree of nodes, upon which the application can search for nodes, read their information, and update the contents of the nodes. SAX is an event-driven interface. The application registers with the parser various event handlers. As the parser reads an XML document, it generates events for the encountered nodes and triggers the corresponding event handlers. Recently, there have been newly proposed XML programming interfaces such as *pull-based parsing*, *e.g.*, StAX (Streaming API for XML), and *data binding*, *e.g.*, JAXB (Java Architecture for XML Binding).

Historical Background

DOM (Document Object Model) was initially used for modeling HTML (HyperText Markup Language) by various Web browsers. As inconsistencies existed among the individual DOM models adopted by different browsers, inter-operability problems arose in developing browser-neutral HTML codes. W3C (World Wide Web Consortium) standardized and released DOM Level 1 specification on October 1, 1998, with support for both HTML and XML. DOM Level 2 was released in November 2000 and added namespace support. The latest specification DOM Level 3 was released in April 2004.

SAX (Simple API for XML) was developed in late 1997 through the collaboration of several implementers of early XML parsers and many other members of the XML-DEV mailing list. The goal was to create a parser-independent interface so that XML applications can be developed without being tied to the proprietary API (Application Programming Interface) of any specific parser. SAX1 was finalized and released on May 11, 1998. The latest release SAX2, finalized in May 2000, includes namespace support.

Foundations

XML parsing is the process of reading an XML document and providing an interface to the user application for accessing the document. An XML parser is a software apparatus that accomplishes such tasks. In addition, most XML parsers check the *well-formedness* of the XML document and many can also validate the document with respect to a DTD (Document Type Definition) [2] or XSD (XML Schema (W3C)) [11]. Through the parsing interface, the user application can focus on the application logic itself, without dwelling on the tedious details of XML, such as Unicode support, namespaces, character references, well-formedness, and so on.

XML Programming Interfaces

Most XML parsers can be classified into two broad categories, based on the types of API that they provide to the user applications for processing XML documents.

Document Object Model (DOM): DOM is a tree-based interface that models an XML document as a tree of various nodes such as elements, attributes, texts, comments, entities, and so on. A DOM parser maps an XML document into such a tree rooted at a `Document` node, upon which the application can search for nodes, read their information, and update the contents of the nodes.

Simple API for XML (SAX): SAX is an event-driven interface. The application receives document information from the parser through a `ContentHandler` object. It implements various event handlers in the interface methods in `ContentHandler`, and registers the `ContentHandler` object with the SAX parser. The parser reads an XML document from the beginning to the end. When it encounters a node in the document, it generates an event that triggers the corresponding event handler for that node. The handler thus applies the application logic to process the node specifically.

The SAX and DOM interfaces are quite different and have their respective advantages and disadvantages. In general, DOM is convenient for random

access to arbitrary places in an XML document, can not only read but also modify the document, although it may take a significant amount of memory space. To the contrary, SAX is appropriate for accessing local information, is much more memory efficient, but can only read XML.

- DOM is not memory efficient since it has to read the whole document and keep the entire document tree in memory. The DOM tree can easily take as much as ten times the size of the original XML document [4]. Therefore it is impossible to use DOM to process very large XML documents, such as the ones that are bigger than the memory. In contrast, SAX is memory efficient since the application can discard the useless portions of the document and only keep the small portion that is of interests to the application. A SAX parser can achieve constant memory usage thus easily handle very large documents.
- SAX is appropriate for streaming applications since the application can start processing from the beginning, while with DOM interface the application has to wait till the entire document tree is built before it can do anything.
- DOM is convenient for complex and random accesses that require global information of the XML document, whereas SAX is more suited for processing local information coming from nodes that are close to each other. The document tree provided by DOM contains the entire information of the document, therefore it allows the application to perform operations involving any part of the document. In comparison, SAX provides the document information to the application as a series of events. Therefore it is difficult for the application to handle global operations across the document. For such complex operations, the application would have to build its own data structure to store the document information. The data structure may become as complex as the DOM tree.
- Since DOM maintains information of the entire document, its API allows the application to modify the document or create a new document, while SAX can only read a document.

DOM and SAX are the two standard APIs for processing XML documents. Most major XML parsers support them. There are also alternative tree-based XML models that were designed to improve upon DOM, including JDOM and DOM4J. In addition to DOM and SAX, other types of APIs for processing XML documents have emerged recently and are supported by various parsers. Two examples are *pull-based parsing* and *Java data binding*.

Pull-Based Parsing: Evolving from XMLPULL, StAX (Streaming API for XML) also works in a streaming fashion, similar to SAX. Different from SAX where the parser pushes document information to the application, StAX enables the application to pull information from the parser. This API is more natural and convenient to the programmer since the application takes full control in processing the XML document.

Java Data Binding: A data-binding API provides the mapping between an XML document and Java classes. It can construct Java objects from the document (*marshalling*) or build a document from the objects (*unmarshalling*). Accessing and manipulating XML documents thus become natural and intuitive, since such operations are performed through the methods of the objects. The application can thus focus on the semantics of the data themselves instead of the details of XML. JAXB (Java Architecture for XML Binding) is a Java specification based on this idea.

Validating Parsers

In addition to accessing XML documents, another critical functionality of XML parsers is to validate the correctness of the documents. Given that XML is a popular data model for data representation and exchange over the Internet, the correctness of XML documents is important for applications to work properly. It is difficult to let the application itself handle incorrect documents that it does not expect. Fortunately, most major XML parsers have the ability to validate the correctness of XML documents.

The correctness of an XML document can be defined at several levels. At the bottom, the document should follow the syntax rules of XML. Such a document is called a *well-formed* document. For example, every non-empty element in a well-formed document should have a pair of starting tag and ending tag. Furthermore, the document should conform to certain semantic rules defined for the application domain, such as a "`state`" node containing one and only one "`capital`" node. Such semantic rules can be defined by XML schema specifications such as DTD or XSD (XML Schema (W3C)). An XML document that complies with a

schema is called a *valid* document. Finally, the application may enforce its own specific semantic rules.

In principle all the parsers are required to perform mandated checks of well-formedness, although there are parser implementations that do not. A parser that checks for the validity of XML documents with respect to XML schema in addition to their well-formedness is a *validating parser*. Schema validation is support by both DOM and SAX API. Most major XML parsers are validating parsers, although some may turn off schema validation by default.

XML Parsing Performance

There are relatively few studies in the literature on performance of XML parsing. However, as the first step in every application that takes XML documents to process, parsing can easily become the bottleneck of the application performance.

DOM is memory intensive since it has to hold the entire document tree in memory, making it incapable in handling very large documents. Therefore, efforts have been made to improve DOM parser performance by exploiting *lazy* XML parsing [7]. The key idea is to avoid loading unnecessary portions of the XML document into the DOM tree. It consists of two stages. The pre-parsing stage builds a virtual DOM tree and the progressive parsing stage expands the virtual tree with concrete contents when they are needed by the application. Farfán et al. [3] further extended the idea to reduce the cost of the pre-parsing stage by partitioning an XML document, thus only reading a partition into the DOM tree when it is needed.

Nicola et al. [6] investigated several real-world XML applications where the performance of SAX parsers is a key obstacle to the success of the projects. They further verified that schema validation can add significant overheads, which can sometimes even be several times more expensive than parsing itself.

Validation often incurs significant processing costs. One reason for such low efficiency is the division of parsing and validation steps. In conventional parsers these two steps are separate this is because validation often requires the entire document to be in the memory thus having to wait till the parsing is finished. Therefore, even for a SAX parser, the advantage of memory efficiency is lost. To cope with this challenge, there have been studies on integrating parsing and validation into a *schema-specific parser* [1,9,12]. For example, [1] constructs a push-down automaton to combine parsing and validation. Van Engelen et al. [10] uses deterministic finite state automata (DFA) to integrate them and the DFA is built upon the schema according to mapping rules. Kostoulas et al. [5] further applies compilation techniques to optimize such integrated parsers.

Takase et al. [8] explores a different way to improve parser performance. It memorizes parsed XML documents as byte sequences and reuses previous parsing results when the byte sequence of a new XML document partially matches the memorized sequences.

Key Applications

Every XML application has to parse an XML document before it can access the information in the document and perform further processing. Therefore, XML parsing is a critical component in XML applications.

URL to Code

List of XML parsing interfaces

DOM, http://www.w3.org/DOM/
JDOM, http://jdom.org/
DOM4J, http://dom4j.org/
SAX, http://www.saxproject.org/
StAX, http://jcp.org/en/jsr/detail?id=173
XMLPULL, http://www.xmlpull.org/
JAXB, http://www.jcp.org/en/jsr/detail?id=222

List of XML parsers

Ælfred, http://saxon.sourceforge.net/aelfred.html
Crimson, http://xml.apache.org/crimson/
Expat, http://expat.sourceforge.net/
JAXP, https://jaxp.dev.java.net/
Libxml2, http://xmlsoft.org/index.html
MSXML, http://msdn.microsoft.com/en-us/library/ms763742.aspx
StAX Reference Implementation (RI), http://stax.codehaus.org/
Sun's Stax implementation, https://sjsxp.dev.java.net/
XDOM, http://www.philo.de/xml/
Xerces, http://xerces.apache.org/

Cross-references

▶ XML
▶ XML Attribute
▶ XML Document
▶ XML Element
▶ XML Programming
▶ XML Schema

Recommended Reading

1. Chiu K., Govindaraju M., and Bramley R. Investigating the limits of SOAP performance for scientific computing. In Proc. 11th IEEE Int. Symp. on High Performance Distributed Computing, 2002, pp. 246–254.
2. Document Type Declaration, http://www.w3.org/TR/REC-xml/#dt-doctype
3. Farfán F., Hristidis V., and Rangaswami R. Beyond lazy XML parsing. In Proc. 18th Int. Conf. Database and Expert Syst. Appl., 2007, pp. 75–86.
4. Harold E.R. Processing XML with Java(TM): a Guide to SAX, DOM, JDOM, JAXP, and TrAX. Addison-Wesley, MA, USA, 2002.
5. Kostoulas M., Matsa M., Mendelsohn N., Perkins E., Heifets A., and Mercaldi M. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In Proc. 15th Int. World Wide Web Conference, 2006, pp. 93–102.
6. Nicola M. and John J. XML parsing: a threat to database performance. In Proc. Int. Conf. on Information and knowledge Management, 2003, pp. 175–178.
7. Noga M., Schott S., and Löwe W. Lazy XML processing. In Proc. 2nd ACM Symp. on Document Engineering, 2002, pp. 88–94.
8. Takase T., Miyashita H., Suzumura T., and Tatsubori M. An adaptive, fast, and safe XML parser based on byte sequences memorization. In Proc. 14th Int. World Wide Web Conference, 2005, pp. 692–701.
9. Thompson H. and Tobin R. Using finite state automata to implement W3C XML schema content model validation and restriction checking. In Proc. XML Europe, 2003, pp. 246–254.
10. Van Engelen R. Constructing finite state automata for high performance XML web services. In Proc. Int. Symp. on Web Services, 2004, pp. 975–981.
11. XML Schema (W3C), http://www.w3.org/XML/Schema
12. Zhang W. and Van Engelen R. A table-driven streaming XML parsing methodology for high-performance web services. In Proc. IEEE Int. Conf. on Web Services, 2006, pp. 197–204.

XML Persistence

► XML Storage

XML Process Definition Language

Nathaniel Palmer
Workflow Management Coalition, Hingham, MA, USA

Synonyms

XPDL

Definition

The primary standards body for workflow management and business process interoperability.

Key Points

The XML Process Definition Language (XPDL) is a format standardized by the Workflow Management Coalition to interchange business process definitions between different modeling tools, BPM suites, workflow engines and other software applications. XPDL defines a XML schema for specifying the declarative part of workflow.

XPDL is designed to exchange the process design, both the graphics and the semantics of a workflow business process. XPDL contains elements to hold the X and Y position of the activity nodes as well as the coordinates of points along the lines that link those nodes. XPDL provides the serialization for BPMN. This distinguishes XPDL from BPEL, which is also a process definition format, but does not contain elements to represent the graphical aspects of a process diagram and BPEL focuses exclusively on the executable aspects of the process.

Cross-references

► Business Process Execution Language
► Business Process Modeling Notation
► Workflow Model
► Workflow Schema

XML Programming

Peter M. Fischer
ETH Zurich, Zurich, Switzerland

Synonyms

XML application development

Definition

XML programming [2] covers methods and approaches to process, transform and modify XML data, often within the scope of a larger application which uses imperative programming languages. Similar to database programming, an important issue in XML programming is the impendence mismatch between the existing programming models, which are mostly based on an object-oriented data model and use an

imperative style, and XML programming approaches, which are based on an XML data model, and apply various programming styles. A plethora of XML programming approaches exists, driven by different usage patterns of XML in applications. The XML programming approaches can be classified into three areas: (i) XML APIs to existing languages, (ii) XML extensions of existing programming languages, and (iii) Native XML processing languages. The varying sets of XML programming requirements and XML programming approaches make it impossible to declare a clearly preferable approach. Careful analysis by the application designer is needed to determine which technique is best suited for a particular setting.

Historical Background

The need for XML programming arose soon after XML had been established as a simplified derivative of SGML, capable of representing any kind of semi-structured data. Historically, three areas were influential to shape the directions in XML programming:

- Low-level APIs oriented towards document parsing: XML being regarded as a structured document format, a popular way to program XML is based on letting a document parser transform the XML into some internal structure of the target programming language, based on concepts of the areas of compiler construction. This approach had already been used for HTML with great success, making DOM the default API for client-side web programming.
- Document Transformation languages are a second influence also based on the document nature of XML. Such languages specify rules to apply to specific fragments of a document and generate new document parts out of the matched fragments and modification/creation statements. A well-known transformation language out of the SGML world is DSSSL, which is often used in the context of DocBook for scientific document processing.
- Database programming: a third influence stems from treating XML as a generic data representation format that, in turn, can be maintained and queried in a similar way as relational data. In database programming, two different data models (relational/OO) and two different programming styles (declarative/imperative) need to be reconciled as well. Four main directions were developed:

1. Call-level interfaces: a literal string of the database language is given as a parameter to a function of the imperative host language. This function is used to interface with the DBMS and returns a result that can be turned into data types of the host language. Typical examples of these are ODBC or JDBC.
2. Embedded SQL: the query expressions of the relational language are embedded in the host application code, allowing for better type checking and hiding the details of the actual interfacing to the DBMS.
3. Automatic mapping layers: instead of modeling both the application with (usually) object-oriented methodologies (UML) and the database with relational methodologies (E/R) and writing expressions in both the database and the host language, only the application is modeled and the code for the application is written. The necessary database schema and the query expressions to retrieve and insert data into the DBMS are automatically generated from the application. The database aspect is hidden and development is simplified. The drawbacks are the lack of flexibility and often lower performance than with a separate database design.
4. Procedural extensions to SQL: an inverse approach to hiding the database is to integrate the application into the database, thus benefitting from the stability and scalability of the DBMS environment while exploiting performance benefits by being "close" to the data. For this approach, SQL has been extended with imperative constructs to enhance its expressive power (Turing-completeness) and make the development style more suitable to general applications. Well-known examples of such SQL extensions are PL/SQL (Oracle) or Transact-SQL (Sybase, Microsoft).

Foundations

Specific Requirements of XML Programming

While there are certain similarities to related areas like database programming (in particular, the impedance mismatch), XML programming has its own special set of requirements and challenges, which can be traced back to two specific areas: (i) the advantages and deficiencies of XML as data and programming model,

and the resulting differences to other, established programming methodologies, and (ii) the widely varying and non-uniform use of XML in terms of usage and operations.

Conceptual Aspects of XML

XML has a clear set of advantages that set it apart from other approaches used to represent data, and led to its rapid acceptance.

A determining factor of the success of XML has been the independence from particular vendors and platforms. This advantage has been further strengthened by a large number of high-quality tools for XML technologies that are often available under permissive licenses, making the integration of XML technology into existing or new applications easy. As a result, knowledge about XML and related technologies is available freely.

The XML syntax is both human readable and machine readable, avoiding problems that occur if a format is only specialized for one way of interpretation, such as unstructured text or complex binary encodings.

XML is not just a document format, but includes methods and technologies for metadata description, RPC, workflow management, declarative querying, security, document processing and many more. These technologies and methods cover a large part of the required features for application development and ensure high interoperability not just among the classes of XML technology, but also among the applications building on them.

From a data and application design point of view, XML provides a number of benefits that make it a good choice over competing approaches: in contrast to relational or object-oriented approaches, data and its interpretation are decoupled. This decoupling allows writing code that works on schema-less data, but does not prohibit adding schema when needed. Working with schema-less data is particularly helpful to shorten time-to-market times when building new applications, as the time-consuming and tedious schema design phase can be shortened. Similarly, it is helping with long-lived data (common in many business settings) where the code is already outdated, but the data will still be needed for new applications. The benefit of not being forced to have a schema is being further enhanced by the fact that the semi-structured model of XML allows representing a large spectrum of data "shapes," reaching from unstructured data like annotated text to highly structured data like a relational table. The XML syntax and data models are not limited to represent data, but are also used to represent metadata and code, allowing uniform management and modifications.

XML, however, does have limitations that reduce its usefulness for certain applications: many standardization efforts, as well as many development efforts, follow a bottom-up approach by defining small entities with low complexity. While this approach ensures that the individual standards and components are easy to understand and use, a combination of standards or components does not always cover all required aspects, thus leaving room for interpretation and incompatibility. A more serious conceptual problem is the limitation of the data model towards tree structures, making it difficult to express arbitrary graph structures or N:M relations. This issue is being aggravated because there is not a commonly accepted way to specify references in XML data. Another important deficiency is the lack of standard design methodology. UML and Entity-Relationship-Modeling have greatly helped to establish the concepts of object-oriented and relational technologies, respectively. By using them, modeling and developing applications are greatly simplified.

While not being a deficiency of XML per se, the mismatch between XML concepts and programming-language/database concepts complicates the use of XML and makes programming for it more difficult: since the data model of XML is neither object-based nor relational, translation needs to be done from and to XML, keeping it outside the usual type system and program analysis of the existing environments. The ability of XML to both work without any schema and represent many shapes of data makes direct mapping of arbitrary XML instances to a strict object-based structure (or to relational schema) often impossible, forcing the use of generic and less useable APIs. The object-oriented approach of hiding the data and binding the methods tightly to this data are juxtaposed to the nature of XML where data are explicitly exposed and accessible to many different methods.

Differences in XML Usage

The second main issue in programming is the wide variety of ways in which XML is used. This usage includes the type of content that an XML instance represents, the operations that are performed on the data and

some non-functional properties that have an impact on the processing such as size, structure, persistence etc.

XML is used to represent a large number of different *types of content*, each leading to different functional requirements, favoring specific approaches for XML programming.

The first, and currently most popular, class of using XML is as a *document storage format*. Typical examples are XHTML, office documents (OOXML, ODF), and graphics formats (SVG). These files are usually used to be presented in a human readable format by an application, transformed into another document format, modified and stored again. A second class is *database content*, either in forms of document collections or representing complex data in XML format. Here, the focus is on maintaining a large data set of XML and being able to successfully retrieve matching data. A third class is about program *code*, *metadata* and *configuration data*, where XML is used to determine the behavior of a data processing system. A fourth class is to use it as *a communication format*, in cases such as SOAP, or REST, putting the focus of transferring data or state in a loosely coupled, yet efficient way from one system to the other. A fifth and upcoming class is to use XML for *log files*, *event streams* or *scientific data streams*, requiring the analysis of the data by correlating data or detecting event patterns.

On these types of content, different *operations* are performed, which in turn are better supported by certain XML programming approaches: Next to the complete retrieval of the XML content, limited or full-scale *query operations* such as filtering/selection, projection or the joins are common operations. A typical operation for many scenarios is also the *creation of new XML data*. *Updating existing XML data* is common in database settings, document management, code and metadata. More specialized operations are *full text search*, relevant for document collections and databases and *trigger processing* and *event generation*, which are commonly used either in databases or data streams.

In addition to the types of content and the operations, a number of non-functional properties have an impact on which XML programming approach to use in particular scenarios. Having large *volumes* of XML requires an approach that supports the relevant technologies for scaling well, but for small volumes such an approach might be too heavyweight. Similarly, dealing with *persistent data* requires different, more elaborate approaches if only *temporary data* are used. Using very *structured data* allows different optimization in storage and processing of XML than using *unstructured data*, since implementation techniques out of either relational databases or text processing are more appropriate. Again, different XML programming approaches are better suited for one or the other. Similarly, some approaches can deal better with data that is *read only*, *append only* or freely *updateable*.

Additional Classification Criteria

Next to the criteria that come from the basic properties of XML and the variety of the XML usage scenarios, there are additional classification criteria that – in the widest sense – are concerned with architecture, "user experience" of the approaches, typing, compatibility and performance.

The *integration of XML processing into the architecture* of an application is an important aspect of an XML programming approach. The traditional approach is to use XML just as the input and/or serialization format and leave the architecture of the application unaffected. A second, more disruptive but also more powerful approach, is to use an XML type inside the type system of the host programming language. This type is usually also augmented by more or less powerful expressions that work on it. The third, and most intrusive approach, is to use an XML data model and a native XML expression language throughout the whole application. These three approaches show an increasing amount of *disruptiveness*, forcing developers and architects to give up on the existing knowledge on application development. On the other hand, the three approaches also show an increasing ability to utilize the advantages of XML, such as schema-less processing while reducing the impedance mismatch between the XML world and the application. The first approach allows a programmer to have high *productivity* quickly, since existing programming knowledge can be re-used, but on the long run the complications of dealing with the impedance mismatch in this approach make the second and the third approach a more compelling option. *Compliance to W3C standards* is important when interoperating with other XML-based applications, but certain standards, such as XML Schema, can add a significant amount of complexity to an approach. Closely related are the aspects of the *XML data model* (e.g., Infoset, XDM, proprietary), *type support* for the XML data

(general node types versus full schema types) and the support for *static type analysis*, because an extended XML type support adds complexity to an API, but allows for more strict correctness checking and better optimizations. Important aspects of the acceptance of an XML programming approach are *performance* and *optimizability*. Low-level approaches that are not deeply integrated into the architecture tend to have the best performance on the short run, as developers can choose their access pattern on the data and optimize a program written in a language they are familiar with. On the long run, declarative solutions with a uniform data and expression model and strong typing support hold the most promise, as they can shift the burden of optimizing from the programmer to optimizers built into the system.

Approaches to XML Programming

A large number of approaches to develop XML-based applications exist, they are influenced by document processing and database programming methods. In this section, the approaches are clustered along the amount of change to the architecture and programming style they require compared to object-oriented/imperative programming: when making a decision on a specific approach to use, the main trade-off is how much of the XML advantages should/need to be used compared to the disruption caused by moving away from the well-known application development environments.

The three main classes of approaches are XML interfaces to existing languages, XML extensions to existing languages and XML-oriented/native XML programming languages.

The first class, *interfaces to existing languages*, provides methods to maintain all of the program logic and complexity inside the established imperative/object-oriented languages. XML is treated like any other source of outside data by limiting its impact to an adaptation layer/API and representing XML as instances of the native, non-extended type system, e.g., as tree of node objects. By doing so, the impact on existing programming models is kept low, but at the cost of either having very generic APIs with low productivity or limited flexibility. The impedance mismatch in the data model and the expressions as well as the purely imperative programming style limits the possibilities for optimization.

This first class can further be broken down into two subclasses, *generic XML-oriented APIs* and *schema-driven code generation*. Generic XML-oriented APIs do not require any knowledge of the particular structure of an XML instance, thus representing XML as generic, low-level objects in the host language such as trees of nodes, event sequences or node item sequences. The majority of these APIs provide low level, parser-oriented programming interfaces such as DOM, SAX and StaX. They support a limited set of querying operations such as selection or projection, the creation of new XML and updates to existing XML; any more high-level functionality needs to be implemented in the imperative language. This allows for a large amount of generality and possibly high performance, since access and processing can be tailored to the needs of a specific application. This advantage, however, comes at a high price: the generic and low-level nature of the APIs cause low developer productivity. There also exist call-level interfaces to XML-oriented programming languages or database system such as XQJ, which allow shifting/moving of some XML-oriented operations outside the imperative program. This frees the developer from low-level work, but requires learning and understanding of a separate expression language with a different data model and different semantics.

Schema-driven code generation increases the abstraction level and developer productivity by automatically creating high-level host programming language objects representing specific, typed XML, e.g., by turning an XML item representing "person" data into a person object. This object can then be accessed and manipulated by the methods the object exposes, e.g., a method to get the name of a person. By doing so, the level of abstraction is increased, developers do not need to learn many details about XML and can stay within their well-understood object-oriented/imperative world, all leading to higher productivity. Schema knowledge (e.g., DTD, XML schema or an ad-hoc format) is needed to perform this automatic code generation, restricting this approach to scenarios where the XML instances are highly structured and this structure information is known in advance. This limits the flexibility of the code generation approach, as it reduces XML to a representation format of host language objects. Well-known examples of such code-generation approaches are XML Beans or JAXB.

The second class of XML programming approaches, *XML extensions to existing programming languages*, still maintains all the logic and complex application code in the imperative/OO programming

language, but extends the host language to represent XML as a first-class data type of this language. This extension of the type system is often accompanied by expressions that work on that new type, which can range from accessor methods to (limited) declarative querying possibilities. The XML type system support is often aligned with the properties of the host language type system, thus reducing the mismatch, but also limiting the compatibility with W3C standards (e.g., XML Schema not being implemented). The tighter integration with the host programming language increases programmer productivity, without restricting the flexibility as much as schema-driven code generation.

This second class can again be broken down into subclasses: XML as a native programming language type, XML as a native ORDBMS database type and query capabilities inside the programming language.

Adopting *XML as a native programming language data type* is an approach that has been taken especially by web-oriented programming languages such as Javascript/Ecmascript and PHP, because using XML interfaces was not considered sufficient any more. XML is a first-rate data type that can be constructed inside the program or read from a file. Instances of this XML data type can be accessed by functions that incorporate a subset of path navigation with a syntax that resembles the access to a field or a class member, e.g., `x.balance` to access the balance child of the XML variable x in XML extension of Javascript, E4X. While the mismatch to the host language is being reduced, a mismatch to W3C XML standards is often created. The imperative nature of the languages often limits the optimizability.

Using *XML as native DB data type* tries to achieve similar effects in the relational/object-relation database scenario, since there also exist impedance mismatches between the relational space and the XML space. Adding XML as a column type provides one possible solution to store XML in relational database system. On the language side, SQL/XML [3] blends SQL (as a relational query language) with XQuery as an XML query language by providing methods to map from one data model to the other, and embed XQuery expressions into SQL. This approach takes advantage of the DBMS infrastructure including triggers, transactional support, scalability, clustering, and reliability. Since both languages are declarative and there is mapping between the data models, global optimization over both XML and relational expression is possible. SQL/XML is supported by the major database vendors, making it well-known and providing good tool support and documentation. The drawbacks include the high overhead of loading all the XML data into the database, which is not useful for temporary XML or small volumes of data, and the complexity and cost of running a database server. The combination of two different query languages with different syntax, semantics and data models hinders the productivity of developers.

Adding *query capabilities inside existing imperative language* takes imperative languages a step closer to the database and increases the productivity, flexibility and optimizability of an existing programming language. The most prominent example of including such capabilities is the LinQ extension of the Microsoft .NET framework, which is based on the COmega research prototype [4]. On top of XML data type extensions and basic accessors eventwell (as described in the previous cases) and similar extension for relation data, LinQ provides query capabilities over all supported data models. These query capabilities come in the form of explicit query operators similar to relational algebra including collection-oriented selection and projection, joins, grouping etc., that work on all data types. On top of these operations, LinQ provides declarative queries similar in style to the SQL Select-From-Where. LinQ provides a good integration of the different data models and with the rest of the language, including the libraries, and yields a high productivity for developers familiar with .NET. Significant drawbacks are the lack of support for typed XML, the limited scope of static analysis, and the dominance of imperative constructs in the language, making database-style optimizations like lazy evaluation, streaming and indexing hard to do automatically.

The third class, *XML-oriented programming languages*, handles all application logic in an environment that is based purely on an XML data model and expressions working on this data model. Doing so represents a significant disruption from conventional languages, including giving up existing tools and design approaches. Compliance to W3C standard is high as is productivity related to XML processing. Many of these languages are, however, not designed to be general-purpose programming languages, lacking libraries and support for areas like GUI programming or numerical computations.

This third class can again be split into three subclasses: domain-specific languages, expression languages with an XML type system, and XML scripting.

Domain-specific languages utilize XML types and expression for a specific task that is often related to one particular use of XML. Clearly, such a language only works well within its intended design domain, but the close match to XML technologies and the restriction to relevant concepts facilitate high productivity. A very prominent example is BPEL, a workflow description language with XML syntax, which is used to "orchestrate" Web Services. It is transparent/agnostic to the actual XML data model, query language and expression language by just focusing on the control flow expressions needed in workflow environments. It provides a high abstraction level and many useful concepts needed in workflow environment. The separation of control flow and the actual expressions restricts the optimizability even in its intended domain.

Languages with an XML Type System are designed to handle operations on XML data in the most useable and efficient way, but not as general programming languages. The best known languages are XPath 2.0, XSLT 2.0 and XQuery. All of these languages work with a consistent data model (XDM) and provide high compliance to other W3C standards. The operations covered-include querying XML instances, construction of new XML instances (XQuery+XSLT), updating existing XML instances (XQuery) and full text search (XQuery). XSLT carries an XML syntax and uses a recursive pattern approach of document transformation language. It works best for small volumes of temporary XML data that are of unknown structure. XQuery uses a declarative style based in iterations and support for static type analysis. It works well for structured data and provides the facilities for good optimizability. Implementations exist for persistent and temporary data, large and small volumes of data in database systems, and as stand-alone expression engines. *XML Scripting* languages represent a recent development where a declarative XML type language (often XQuery) is extended with imperative constructs. The goal is to gain usability and a certain level of expressiveness by allowing (limited) side effects while still maintaining optimizability. Such a language could be used for at all tiers of XML software stacking, proving the potential for a truly XML-only application development. Major challenges in the long run will be developer acceptance and the building of good compilers/optimizers to actually achieve the potential performance benefits. Compared to imperative languages with query capabilities, the chances for optimizability are better, since the starting point is already well optimizable language. The most prominent example of such an XML scripting language is [1].

Key Applications

XML programming is relevant for almost all usage scenarios of XML that go beyond simple storage and retrieval. The large variety of use cases for XML is reflected in the different requirements for XML programming. A very common case is web application programming, where the browser, as the client layer, and the database server layer both contain and manipulate XML data. Web services, as a way of loosely coupling applications via XML messages, have become popular for information and application integration. An area in between these two use cases are mashups, were data and services from different sources are combined to form new applications.

Future Directions

A promising future direction is the integration of continuous/streaming XML data in the "regular" programming environments. Many data stream sources already produce their data as XML, and this data needs to be combined with other streaming or static data. Semantic querying, which is now handled using RDF/OWL, can be brought towards more standard XML programming models, thus enabling more effective ways for data integration. The interaction between XML integrity constraints and programming will become more important, as reasoning over programs and data (including verification) will improve the quality of applications. Automatic maintenance of a well-defined state (similar to relational integrity constraints in RDBMSs) will simplify programming XML applications in data-intensive environments. Native XML languages have the potential to change the architecture for many data-intensive applications which currently are built in a multi-tier fashion. Using the same data model and expressions allows-collapsing layers/tiers when needed, setting the main direction of partitioning not along the tiers, but along services and their respective data.

Cross-references

► Active XML
► AJAX
► Composed Services and WS-BPEL

- ▶ Entity Relationship Model
- ▶ Java Database Connectivity
- ▶ Open Database Connectivity
- ▶ SOAP
- ▶ Unified Modeling Language
- ▶ W3C
- ▶ Web Services
- ▶ XML
- ▶ XML Information Integration
- ▶ XML Integrity Constraints
- ▶ XML Parsing, SAX/DOM
- ▶ XML Process Definition Language
- ▶ XML Schema
- ▶ XML Stream Processing
- ▶ XPath/XQuery
- ▶ XQuery Full-Text
- ▶ XQuery Processors
- ▶ XSL/XSLT

Recommended Reading

1. Chamberlin D., Carey M.J., Fernandez M., Florescu D., Ghelli G., Kossmann D., Robie J., and Simeon J. XQueryP: an XML application development language. In Proc. XML 2006 Conference. 2006.
2. Florescu D. and Kossmann D. Programming for XML. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2006, p. 801.
3. Funderburk J.E., Malaika S., and Reinwald B. XML programming with SQL/XML and XQuery. IBM Syst. J., 41(4):642–665, 2002.
4. Meijer E., Schulte W., and Bierman G. Unifying tables, objects and documents. In Proc. Workshop on Declarative Programming in the Context of Languages, 2003.

XML Publish/Subscribe

Yanlei Diao[1], Michael J. Franklin[2]
[1]University of Massachusetts Amherst, MA, USA
[2]University of California-Berkeley, Berkeley, CA, USA

Synonyms

Selective XML dissemination; XML filtering; XML message brokering

Definition

As stated in the entry "Publish/Subscribe over Streams," publish/subscribe (pub/sub) is a many-to-many communication model that directs the flow of messages from senders to receivers based on receivers' data interests. In this model, publishers (i.e., senders) generate messages without knowing their receivers; subscribers (who are potential receivers) express their data interests, and are subsequently notified of the messages from a variety of publishers that match their interests.

XML publish/subscribe is a publish/subscribe model in which messages are encoded in XML and subscriptions are written in an XML query language such as a subset of XQuery 1.0. (In the context of XML pub/sub, "messages" and "documents" are often used exchangeably.)

In XML-based pub/sub systems, the message brokers that serve as central exchange points between publishers and subscribers are called XML message brokers.

Historical Background

As described in "Publish/Subscribe over Streams," XML pub/sub has emerged as a solution for loose coupling of disparate systems at both the communication and content levels. At the communication level, pub/sub enables loose coupling of senders and receivers based on the receivers' data interests. With respect to content, XML can be used to encode data in a generic format that senders and receivers agree upon due to its flexible, extensible, and self-describing nature; this way, senders and receivers can exchange data without knowing the data representation in individual systems.

Foundations

XML pub/sub raises many technical challenges due to the requirements of large-scale publish/subscribe (as described in the entry "Publish/Subscribe over Streams"), the volume of XML data, and the complexity of XML processing. Among all, two challenges are highlighted below:

- *XML stream processing.* In XML-based pub/sub systems, XML data continuously arrives from external sources, and user subscriptions, stored as continuous queries in a message broker, are evaluated every time when a new data item is received. Such XML query processing is referred to as *stream-based.* In cases where incoming messages are large, stream-based processing also needs to start before the messages are completely received in order to reduce the delay in producing results.

- *Handling simultaneous XML queries.* Compared to XML stream systems, a distinguishing aspect of XML pub/sub systems lies in the size of their query populations. All the queries stored in an XML message broker are simultaneously active and need to be matched efficiently with each incoming XML message. While multi-query processing has been studied for relational databases and relational streams, the complexity of XML processing, including structure matching, predicate evaluation, and transformation, requires new techniques for efficient multi-query processing in this new context.

Foundation of XML Stream Processing for Publish/Subscribe

Event-based parsing. Since XML messages can be used to encode data of immense sizes (e.g., the equivalent of a database's worth of data), efficient query processing requires fine-grained processing upon arrival of small constituent pieces of XML data. Such fine-grained XML query processing can be implemented via an event-based API. A well known example is the *SAX* interface that reports low-level parsing events incrementally to the calling application. Figure 1 shows an example of how a SAX interface breaks down the structure of the sample XML document into a linear sequence of events. "Start document" and "end document" events mark the beginning and the end of the parse of a document. A "start element" event carries information such as the name of the element and its attributes. A "characters" event reports a text string residing between two XML tags. An "end element" event corresponds to an earlier "start element" event and marks the close of that element. To use the SAX interface, the application receiving the events must implement handlers to respond to different events. In particular, stream-based XML processors can use these handlers to implement event-driven processing.

An automata-based approach. A popular approach to event-driven XML query processing is to adopt some form of *finite automaton* to represent path expressions [1,13]. This approach is based on the observation that a path expression (a small, common subset of XQuery) written using the axes ("/," "//") and node tests (element name or "*") can be transformed into a regular expression. Thus, there exists a finite automaton that accepts the language described by such a path expression [11]. Such an automaton can be created by mapping the location steps of the path expression to the automaton states. Figure 2 shows an example automaton created for a simple path expression, where the two concentric circles represent the accepting state. When arriving, XML messages are parsed with an event-based parser, the events raised during parsing are used to drive the execution of the automaton. In particular, "start element" events drive the automaton through its various transitions, and "end element" events cause the execution to backtrack to the previous states. A path expression is said to match a message if during parsing, the accepting state for that path is reached.

XML Filtering

In XML filtering systems, user queries are written using path expressions that can specify constraints over both *structure* and *content* of XML messages. These queries are applied to individual messages (hence, stateless processing). Query answers are "yes" or "no" – computing only Boolean results in XML filtering avoids complex issues of XML query processing related to multiple matches such as ordering and duplicates, hence enabling simplified, high-performance query processing.

XFilter [1], the earliest XML filtering system, considers matching of the structure of path expressions and explores *indexing* for efficient filtering. It builds a dynamic index over the queries and uses the parsing events of a document to probe the query index. This approach quickly results in a smaller set of queries that can be potentially matched by a message, hence avoiding processing queries for which the message is irrelevant. Built over the states of query automata, the dynamic index identifies the states that the execution of these automata is attempting to match at a particular moment. The content of the index constantly changes as parsing events drive the execution of the automata.

YFilter [4] significantly improves over XFilter in two aspects. By creating a separate automaton per query, XFilter can perform redundant work when significant commonalities exist among queries. Based on this insight, YFilter supports *sharing* in processing by using a combined automaton to represent all path expressions; this automaton naturally supports shared representation of all common prefixes among path expressions. Furthermore, the combined automaton is implemented as a *Nondeterministic Finite Automaton* (NFA) with two practical advantages: i) a relatively small number of states required to represent even

```
<?xml version="1.0" ?>
<report>
  <section id="intro" difficulty="easy">
      <title>Pub/Sub</title>
      <section difficulty="easy">
          <figure source="g1.jpg">
              <title>XML Processing</title>
          </figure>
      </section>
      <figure source="g2.jpg">
          <title>Scalability</title>
      </figure>
  </section>
</report>
```

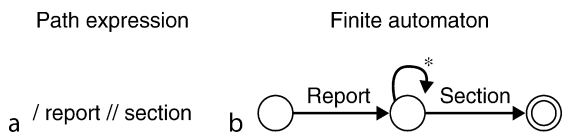
```
<Start Document
<Start Element:     report
<Start Element:     section
<Start Element:     title
  characters:       Pub/Sub
>End Element:       title
<Start Element:     section
<Start Element:     figure
<Start Element:     title
  Characters:       XML processing
>End Element:       title
>End Element:       figure
>End Element:       section
  ...
>End Element:       section
>End Element:       report
>End Document
```

XML Publish/Subscribe. Figure 1. An example XML document and results of SAX parsing.

large numbers of path expressions and complex queries (e.g., with multiple wildcards "*" and descendent axes "//"), and ii) incremental maintenance of the automaton upon query updates. Results of YFilter show that its shared path matching approach can offer order-of-magnitude performance improvements over XFilter while requiring only a small maintenance cost.

Structure matching that XFilter considers is one part of the XML filtering problem; another significant part is the evaluation of predicates that are applied to path expressions (e.g., addressing attributes of elements, text data of elements, or even other path expressions) for additional filtering. Since shared structure matching has been shown to be crucial for performance, YFilter supports predicate evaluation using *post-processing* of path matches after shared structuring matching, and further leverages relational processing in such post-processing.

Figure 3 shows two example queries and their representation in YFilter. Q1 contains a root element "/nitf" with two nested paths applied to it. YFilter decomposes the query into two linear paths "/nitf/head/pubdata[@edition.area="SF"]," and "/nitf//tobject.subject[@tobject.subject.type ="Stock"]." The structural part of these paths is represented using the NFA with the common prefix "/nitf" shared between the paths. The accepting states of these paths are state 4 and state 6, where the network of operators (represented as boxes) for the remainder of Q1 starts. At the bottom of the network, there is a selection (σ) operator above each accepting state to handle the value-based predicate in the corresponding path. To handle the correlation between



XML Publish/Subscribe. Figure 2. A path expression and its corresponding finite automaton.

the two paths (e.g., the requirement that it should be the same "nitf" element that makes these two paths evaluate to true), YFilter applies a join ($\triangleright\triangleleft$) operator after the two selections. Q2 is similar to Q1 and hence shares a significant portion of its representation with Q1.

Index-Filter [2] builds indexes over both queries and streaming data. The index over data speeds up the processing of large documents while its construction overhead may penalize the processing of small ones. Results of a comparison between Index-Filter and YFilter show that Index-Filter works better when the number of queries is small or the XML document is large, whereas YFilter's approach is more effective for large numbers of queries and short documents.

XMLTK [8] converts YFilter's NFA to a *Deterministic Finite Automaton* (DFA) to further improve the filtering speed. A straightforward conversion could theoretically result in severe scalability problems due to an explosion in the number of states. This work, however, shows that such explosion can be avoided in many cases by using lazy construction of the DFA and placing certain restrictions on the types of documents and queries supported (when suitable for the application). *XPush* [9] further explores a pushdown

automaton for shared processing of both structure and value-based constraints. Such an automaton can provide high efficiency when wildcard ("*") and descendant ("//") operators are rare in queries and periodic reconstruction of the automaton can be used.

FiST [14] views path expressions with predicates as twig patterns and considers ordered twig pattern matching. For such ordered patterns, it transforms the patterns as well as XML documents intro sequences using Prufer's method. This approach allows holistic matching of ordered twig patterns, as opposed to matching individual paths and then merging their matches during post-processing in YFilter (which works for both ordered and unordered patterns), resulting in significant performance improvements over YFilter.

XML Filtering and Transformation

XML filtering solutions presented above have not addressed the transformation of XML messages for customized result delivery, which is an important feature in XML-based data exchange and dissemination. For XML transformation, queries are written using a richer subset of XQuery, e.g., the *For-Where-Return* expressions.

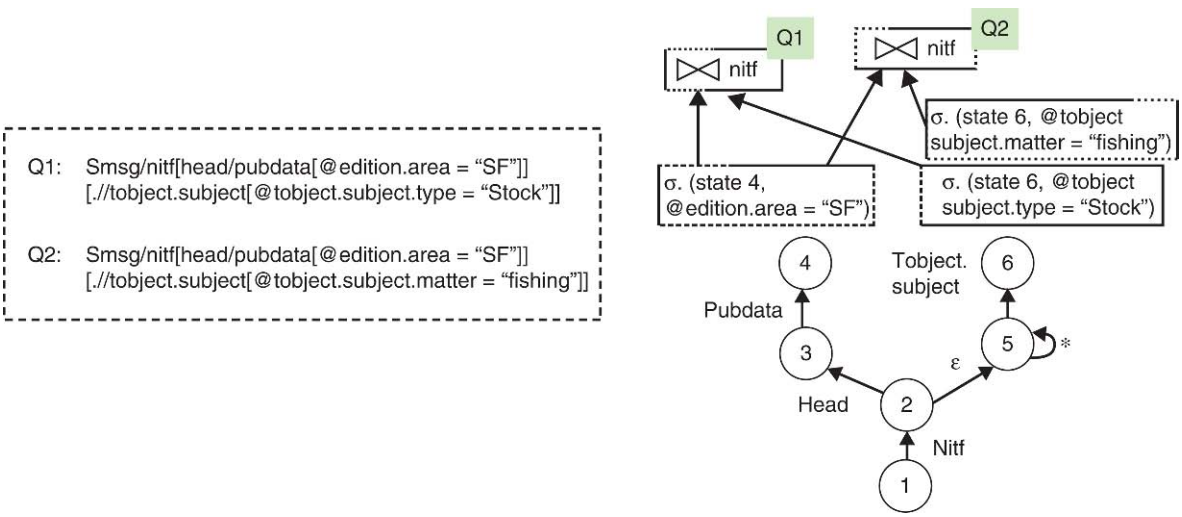
To support efficient transformation for many simultaneous queries, YFilter [5] further extends its NFA-based framework and develops alternatives for building transformation functionality on top of shared path matching. It explores the tradeoff between shared path matching and post-processing for result customization, by varying the extent to which they push paths from the For-Where-Return expressions into the shared path matching engine. To further reduce the remarkable cost of post-processing of individual queries, it employs provably correct optimizations based on query and DTD (if available) inspection to eliminate unnecessary operations and choose more efficient operator implementations for post-processing. Moreover, it provides techniques for also sharing post-processing across multiple queries, similar to those in continuous query processing over relational streams.

Stateful XML Publish/Subscribe

In [10], efficient processing of large numbers of continuous inter-document queries over XML Streams (hence, stateful processing) is addressed. The key idea that it exploits is to dissert query specifications into tree patterns evaluated within individual documents and value-based joins preformed across documents. While employing existing path evaluation techniques (e.g., YFilter) for tree pattern evaluation, it proposes a scalable join processor that leverages relational joins and view materialization to share join processing among queries.

XML Routing

As described in "publish/subscriber over streams" distributed pub/sub systems need to efficiently route messages from their publishing sites to the brokers hosting relevant queries for complete query processing. While the concept of content-based routing and many architectural solutions can be applied in XML-based



XML Publish/Subscribe. Figure 3. Example queries and their representation in YFilter.

pub/sub systems, routing of XML messages raises additional challenges due to the increased complexity of XML query processing.

Aggregating user subscriptions into compact routing specifications is a core problem in XML routing.

Chan et al. [3] aggregate tree pattern subscriptions into a smaller set of generalized tree patterns such that (i) a given space constraint on the total size of the subscriptions is met, and (ii) the loss in precision (due to aggregation) during document filtering is minimized (i.e., a constrained optimization problem). The solution employs tree-pattern containment and minimization algorithms and makes effective use of document-distribution statistics to compute a precise set of aggregate tree patterns within the allotted space budget.

ONYX [6] leverages YFilter technology for efficient routing of XML messages. While subscriptions can be written using For-Where-Return expressions, the routing specification for each output link at a broker consists of a *disjunctive normal form* (DNF) of absolute linear path expressions, which generalizes the subscriptions reachable from that link while avoiding expensive path operations. These routing specifications can be efficiently evaluated using YFilter, even with some work shared with complete query processing at the same broker. To boost the effectiveness of routing, ONYX also partitions the XQuery-based subscriptions among brokers based on exclusiveness of data interests.

Gong et al. [7] introduce Bloom filters into XML routing. The proposed approach takes a path query as a string and maps all query strings into a Bloom filter using hash functions. The routing table is comprised of multiple Bloom filters. Each incoming XML message is parsed into a set of candidate paths that are mapped using the same hash functions to compare with the routing table. This approach can filter XML messages efficiently with relatively small numbers of false positives. Its benefits in efficiency and routing table maintenance are significant when the number of queries is large.

Key Applications

Personalized News Delivery. News providers are adopting XML-based formats (e.g., *News Industry Text Format* [12]) to publish news articles online. Given articles marked up with XML tags, a pub/sub-based news delivery service allows users to express a wide variety of interests as well as to specify which portions of the relevant articles (e.g., title and abstract only) should be returned. *Really Simple Syndication* (RSS) provides similar yet simpler services based on URL- and/or keyword-based preferences.

Application Integration. XML publish/subscribe has been widely used to integrate disparate, independently-developed applications into new services. Messages exchanged between applications (e.g., purchase orders and invoices) are encoded in a generic XML format. Senders publish messages in this format. Receivers subscribe with specifications on the relevant messages and the transformation of relevant messages into an internal data format for further processing.

Mobile services. In mobile applications, clients run a multitude of operating systems and can be located anywhere. Information exchange between information providers and a huge, dynamic collection of heterogeneous clients has to rely on open, XML-based technologies and can be further facilitated by pub/sub technology including filtering and transformation for adaptation to wireless devices.

Data Sets

XML data repository at University of Washington, http://www.cs.washington.edu/research/xmldatasets/ and Niagara experimental data, http://www.cs.wisc.edu/niagara/data.html.

URL to Code

YFilter is an XML filtering engine that processes simultaneous queries (written in a subset of XPath 1.0) against streaming XML messages in a shared fashion. For each XML message, it returns a result for every matched query (http://yfilter.cs.umass.edu/).

ToXgene is a template-based generator for large, consistent collections of synthetic XML documents (http://www.cs.toronto.edu/tox/toxgene/).

XMark is an XQuery benchmark suite to analyze the capabilities of an XML database (http://www.xml-benchmark.org/).

Cross-references

- ▶ Continuous Queries
- ▶ Publish/Subscribe Over Streams
- ▶ XML
- ▶ XML Document
- ▶ XML Parsing
- ▶ XML Schema
- ▶ XML Stream Processing
- ▶ XPath/XQuery

Recommended Reading

1. Altinel M. and Franklin M.J. Efficient filtering of XML documents for selective dissemination of information. In Proc. 26th Int. Conf. on Very Large Data Bases, 2000, pp. 53–64.
2. Bruno N., Gravano L., Doudas N., and Srivastava D. Navigation- vs. Index-based XML Multi-query processing. In Proc. 19th Int. Conf. on Data Engineering, 2003, pp. 139–150.
3. Chan C.Y., Fan W., Felber P., Garofalakis M.N., and Rastogi R. Tree pattern aggregation for scalable XML data dissemination. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 826–837.
4. Diao Y., Altinel M., Zhang H., Franklin M.J., and Fischer P.M. Path sharing and predicate evaluation for high-performance XML filtering. ACM Trans. Database Syst., 28, (4)467–516, 2003.
5. Diao Y. and Franklin M.J. Query processing for high-volume XML message brokering. In Proc. 29th Int. Conf. on Very Large Data Bases, 2003, pp. 261–272.
6. Diao Y., Rizvi S., and Franklin M.J. Towards an Internet-Scale XML dissemination service. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004, pp. 612–623.
7. Gong X., Qian W., Yan Y., and Zhou A. Bloom filter-based XML packets filtering for millions of path queries. In Proc. 21st Int. Conf. on Data Engineering, 2005, pp. 890–901.
8. Green T.J., Gupta A., Miklau G., Onizuka M., Suciu D. Processing XML streams with deterministic automata and stream indexes. ACM Trans. Databases, 29(4):752–788, 2004.
9. Gupta A.K. and Suciu D. Streaming processing of XPath queries with predicates. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 419–430.
10. Hong M., Demers A.J., Gehrke J., Koch C., Riedewald M., and White W.M. Massively multi-query join processing in publish/subscribe systems. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2007, pp. 761–772.
11. Hopcroft J.E. and Ullman J.D. Introduction to Automata Theory, Languages and Computation. Addition-Wesley, Boston, MA, 1979.
12. Internal Press Telecommunications Council. News Industry Text Format. Available online at: http://www.nitf.org/, 2004.
13. Ives Z.G., Halevy and A.Y., Weld D.S. An XML query engine for network-bound data. VLDB J., 11(4): 380–402, 2002.
14. Kwon J., Rao P., Moon B., and Lee S. FiST: scalable XML document filtering by sequencing twig patterns. In Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 217–228.

XML Publishing

Zachary Ives
University of Pennsylvania, Philadelphia, PA, USA

Synonyms

XML export

Definition

XML Publishing typically refers to the creation of XML output (either in the form of a character stream or file) from a relational DBMS. XML Publishing typically must handle three issues: converting an XML query or view definition into a corresponding SQL query; encoding hierarchy in the SQL data; and generating tags around the encoded hierarchical data. Since in some cases the relational data may have originated from XML, the topics of *XML Storage* and XML Publishing are closely related and often addressed simultaneously.

Historical Background

The topic of XML Publishing arose very soon after database researchers suggested a connection between XML and semi-structured data [5], a topic that had previously been studied in the database literature [1,2,8]. Initially the assumption was that XML databases would probably need to resemble those for semi-structured data in order to get good performance. Florescu and Kossmann [11] showed that storing XML in a relation database could be more efficient than storing it in a specialized semi-structured DBMS. Concurrently, Deutsch et al. were exploring hybrid relational/semi-structured storage in STORED [7]. Soon after, the commercial DBMS vendors became interested in adding XML capabilities to their products. The most influential developments in that area came from IBM Almaden's XPERANTO research project [4,14], which formed the core of Shanmugasundaram's thesis work [9]. Today, most commercial DBMSs use a combination of all of the aforementioned techniques: storing XML data in relations, storing hybrid relational/semi-structured data, and storing XML in a hierarchical format resembling semi-structured data.

Foundations

As every student of a database class knows, a good relational database schema is in first normal form (1NF): separate concepts and multi-valued attributes are split into separate tables, and taken together the tables can be visualized as a graph-structured Entity-Relationship Diagram. In contrast, XML data are fundamentall *not* in 1NF: an XML document has a single root node and *hierarchically* encodes data.

Thus, there are two main challenges in XML Publishing: first, taking queries or templates describing XML output and mapping them into operations over 1NF tables; and second, efficiently computing and adding XML tags to the data being queried. Typically, the latter problem is addressed with two separate

modules, and hence the XML Publishing problem consists of three steps:

1. *Converting queries*, which typically are posed in a language other than SQL, into an internal representation from which SQL can be constructed.
2. *Composition and optimization* of the resulting queries, such that redundant work is minimized.
3. *Adding tags*, which is often done in a postprocessing step outside the DBMS.

Each of these topics is addressed in the remainder of this section. The discussion primarily focuses upon the XPERANTO and SilkRoute systems, which established many of the basic algorithms used in XML Publishing.

Converting Queries

The first issue in XML Publishing is that of taking the specification of the XML output, and converting it into some form from which SQL can be constructed. A number of different forms have been proposed for specifying the XML output:

Proprietary XML template languages, such as IBM DB2's Document Access Definition (DAD) or SQL Server's XML Data Reduced (XDR), which essentially provide a scheme for annotating XML templates with SQL queries that produce content.

SQL extensions, such as the SQL/XML standard [12], which extend SQL with new functions to add tags around relational attributes and intermediate table results.

Relational-XML query languages, such as the RXL language in early versions of SilkRoute [10], which provide a language that creates hierarchical XML content using relational tables. RXL resembles a combination of Datalog and an XML query language (specifically, XML-QL [6]), and it can be used to define XML views that can be queried using a standard XML query language.

XML query languages with built-in XML views of relations, an approach first proposed in XPERANTO [3], where each relational table is mapped into a virtual XML document and a standard XML language (like XPath or XQuery) can be used to query the relations and even to define XML views.

Over time, the research community has come to the consensus that the last approach offers the best set of trade-offs, as it offers a single compositional language for all XML queries over the data in the RDBMS. The commercial market has settled on a combination of this same approach, for XML-centric users, plus SQL extensions for relation-centric users. Here the discussion assumes an XML-centric perspective, with XQuery as the language, since it is representative.

XML Publishing systems generally only attempt to tackle a subset of the full XQuery specification, focusing on the portions that are particularly amenable to execution in a relational DBMS. There are many commonalities between basic XQueries and SQL: selections, projections, and joins can be similarly expressed in both of these languages, views can be defined, and queries can be posed over data in a way that is agnostic as to whether the data comes from a view or a raw source. The two main challenges in conversion lie in the input to the XQuery – where XPaths must be matched against relations or XML views defined over relations – and in creating the hierarchical nested XQuery output.

XPath matching over built-in views of relations is, of course, trivial, as each relation attribute maps to an XML element or attribute in the XPath. Conversion gets significantly more complex when the XPaths are over XML views constructed over the original base relations: this poses the problem of XML query composition, where the DBMS should not have to compute each composed view separately, but rather should statically *unfold* them.

Composition and Optimization

As a means of composing queries, SilkRoute takes each XQuery definition and creates an internal *view forest* representation that describes the structure of the XML document; it converts an XQuery into an canonical representation called XQueryCore, and its algorithms compose XQueryCore operations over an input document. XPERANTO, based in large part on IBM's DB2, performs very similar operations, but starts by creating an internal representation of each query block called in a model XQGM, then executes a series of rewrite rules to merge the blocks by merging steps that create hierarchy in a view's output with steps that traverse that hierarchy in a subsequent query's input.

The resulting SQL is often highly complex and repetitive, with many SQL blocks being unioned together: each level of the XML hierarchy may require a separate SQL block, and each view composition step in the optimization process may create multiple SQL query blocks (an XPath over a view may match over multiple relational attributes from the base data).

Thus, optimizing the resulting SQL becomes of high importance. XML publishing systems typically convert from XQuery to SQL, not to actual executable query plans, so they can only do a limited amount of optimization on their own. Here they use a significant number of heuristics [10,14] to choose the best form for the sets of SQL queries, and rely on the RDBMS's cost-based optimizer to more efficiently optimize the queries. (Recently, commercial vendors have invested significant effort in improving their cost-based query optimizers for executing the types of queries output by XML Publishing systems.)

Adding Tags

The final step in XML Publishing is that of adding tags to the data from the relational query. There have been two main approaches to this task: *in-engine*, relying on SQL functions to add the tags; and *outside the DBMS*, relying on an external middleware module to add the tags. Each is briefly described.

In-engine Tagging In [14], the authors proposed a method for adding XML tags using the SQL functions XMLELEMENT, XMLATTRIBUTE, and XMLAGG: each of these takes a relational attribute or tuple and converts it into a partial XML result, encoded in a CLOB datatype. Naturally, the first two create an element or attribute tag, respectively, around a data value. The third function, XMLAGG, functions as an SQL aggregate function: it takes an SQL row set (possibly including XMLELEMENT or XMLATTRIBUTE values) and outputs a tuple with a CLOB containing the XML serialization of the rowset's content. The drawback to the in-engine tagging approach is that CLOBs are not always handled efficiently, as they are typically stored separately from the tuples with which they are associated. Hence, when this method was incorporated into the SQL/XML standard, a new XML datatype was proposed that could be implemented in a manner more efficient than the CLOB.

Tagging Middleware A more common approach is to simply a tuple representation of the XML document tree/forest within the SQL engine, and to convert this tuple stream into XML content externally, using a separate tagging module [9,14]. There are two common techniques for encoding hierarchy in tuple streams: *outer join* and *outer union*. Suppose there are relationaltables in Fig. 1a, encoding an element *a* and its child element *b*, and one wants to encode their output as the following XML:

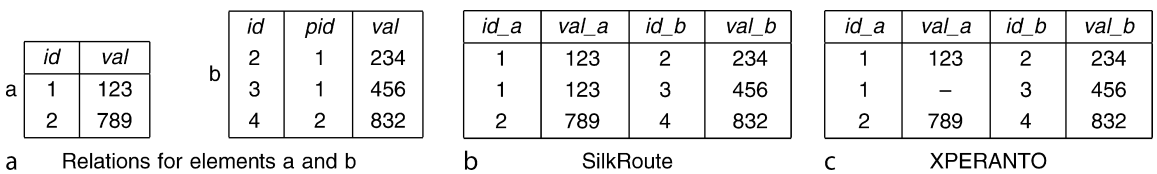
```
<a id="1" val="123"><b id="2"
val="234"/><b id="3" val="456"/></a>
<a id="2" val="789"><b id="4"
val="832"/></a>
```

Outer Join. In SilkRoute, tuples are generated using outerjoins between parent and child relations. The tuple stream will be sorted in a way that allowsthe tagger to hold the last tuple, not all data (see Fig. 1b). The "parent" portion of the tree (the *a* element) will appear in both tuples, and the external XML tagger will compare consecutive tuples in the tuple stream to determine the leftmost position where a value changed – from which it can determine what portion of the XML hierarchy has changed.

Outer union. In contrast, XPERANTO uses a so-called *sorted outer union* representation (Fig. 1c), which substitutes null values for repeated copies of values (but not keys). Its approach relies on two characteristics of IBM's DB2 engine that are shared by some but not all other systems: (i) null values can be very efficiently encoded and processed, meaning that the query is more efficient; (ii) null values *sort high*, i.e., DB2's considers null values to have a sort value greater than any non-null value. The tagger simply needs to look for the first non-null attribute to determine what portion of a the XML to emit.

Key Applications

XML publishing has become a key capability in the relational database arena, as increasingly data



XML Publishing. Figure 1. Tuple representations of XML data.

interchange mechanisms and Web services are being built using XML data from an RDBMS. Today the "Big Three" commercial DBMS vendors all use some techniques for XML Publishing, and it is likely that smaller DBMSs, including those in open source, will gradually incorporate them as well.

Cross-references

- ▶ Approximate XML Querying
- ▶ XML Storage
- ▶ XML Views

Recommended Reading

1. Abiteboul S., Quass D., McHugh J., Widom J., and Winer J.L. The Lorel query language for semistructured data. In Int. J. Digit. Libr., 1(1):68–88, 1997.
2. Buneman P., Davidson S.B., Fernandez M.F., and Suciu D. Adding structure to unstructured data. In Proc. 13th Int. Conf. on Data Engineering, 1997, pp. 336–350.
3. Carey M.J., Florescu D., Ives Z.G., Lu Y., Shanmugasundaram J., Shekita E., and Subramanian S. XPERANTO: publishing object-relational data as XML. In Proc. 3rd Int. Workshop on the World Wide Web and Databases, 2000, pp. 105–110.
4. Carey M., Kiernan J., Shanmugasundaram J., Shekita E., and Subramanian S. XPERANTO: a middleware for publishing object-relational data as XML documents. In Proc. 26th Int. Conf. on Very Large Data Bases, 2000, pp. 646–648.
5. Deutsch A., Fernández M.F., Florescu D., Levy A.Y., and Suciu D. XML-QL. In Proc. The Query Languages Workshop, 1998.
6. Deutsch A., Fernandez M.F., Florescu D., Levy A., and Suciu D. A query language for XML. Comp. Networks, 31(11–16):1155–1169, 1999.
7. Deutsch A., Fernandez M.F., and Suciu D. Storing semistructured data with STORED. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1999, pp. 431–442.
8. Fernandez M.F., Florescu D., Kang J., Levy A.Y., and Suciu D. Catching the boat with strudel: experiences with a web-site management system. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1998, pp. 414–425.
9. Fernandez M.F., Kadiyska Y., Suciu D., Morishima A., and Tan W.C. SilkRoute: A framework for publishing relational data in XML. ACM Trans. Database Syst., 27 (4):438–493, 2002.
10. Fernandez M., Tan W.C., and Suciu D. SilkRoute: trading between relations and XML. Comp. Networks, 33 (1–6):723–745, 2000.
11. Florescu D. and Kossmann D. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Tech. Rep. 3684, INRIA, 1999.
12. ISO/IEC 9075-14:2003 Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML).
13. Shanmugasundaram J. Bridging Relational Technology and XML. Ph.D. thesis, University of Wisconsin-Madison, 2001.
14. Shanmugasundaram J., Shekita E.J., Barr R., Carey M.J., Lindsay B.G., Pirahesh H., and Reinwald B. Efficiently publishing relational data as XML documents. VLDB J., 10(2–3):133–154, 2001.

XML Retrieval

Mounia Lalmas[1], Andrew Trotman[2]
[1]Queen Mary, University of London, London, UK
[2]University of Otago, Dunedin, New Zealand

Synonyms

Structured document retrieval; Structured text retrieval; Focused retrieval; Content-oriented XML retrieval

Definition

Text documents often contain a mixture of structured and unstructured content. One way to format this mixed content is according to the adopted W3C standard for information repositories and exchanges, the *eXtensible Mark-up Language* (XML). In contrast to HTML, which is mainly layout-oriented, XML follows the fundamental concept of separating the *logical structure* of a document from its layout. This logical document structure can be exploited to allow a more focused sub-document retrieval.

XML retrieval breaks away from the traditional retrieval unit of a document as a single large (text) block and aims to implement *focused retrieval* strategies aiming at returning document components, i.e., XML elements, instead of whole documents in response to a user query. This focused retrieval strategy is believed to be of particular benefit for information repositories containing long documents, or documents covering a wide variety of topics (e.g., books, user manuals, legal documents), where the user's effort to locate relevant content within a document can be reduced by directing them to the most relevant parts of the document.

Historical Background

Managing the enormous amount of information available on the web, in digital libraries, in intranets, and so on, requires efficient and effective indexing and retrieval methods. Although this information

is available in different forms (text, image, speech, audio, video etc), it remains widely prevalent in text form. Textual information can be broadly classified into two categories, structured and unstructured.

Unstructured information has no fixed pre-defined format, and is typically expressed in natural language. For instance, much of the information available on the web is unstructured. Although this information is mostly formatted in HTML, thus imposing some structure on the text, the structure is only for presentation purposes and carries essentially no semantic meaning. Correct nesting of the HTML structure (that is, to form an unambiguous document logical structure) is not imposed. Accessing unstructured information is through flexible but mostly simplistic means, such as a simple keyword matching or bag of words techniques.

Structured information is usually represented using XML, a mark-up language similar to HTML except that it imposes a rigorous structure on the document. Moreover, unlike HTML, XML tags are used to specify semantic information about the stored content and not the presentation. A document correctly marked-up in XML has a fixed document structure in which semantically separate document parts are explicitly identified – and this can be exploited to provide powerful and flexible access to textual information.

XML has been accepted by the computing community as a standard for document mark-up and an increasing number of documents are being made available in this format. As a consequence numerous techniques are being applied to access XML documents. The use of XML has generated a wealth of issues that are being addressed by both the database and information retrieval communities [3]. This entry is concerned with content-oriented XML retrieval [2,5] as investigated by the information retrieval community.

Retrieval approaches for structured text (marked-up in XML-like languages such as SGML) were first proposed in the late 1980s. In the late 1990s, the interest in structured text retrieval grew due to the introduction of XML in 1998. Research on XML information retrieval was first coordinated in 2002 with the founding of the Initiative for the Evaluation of XML Retrieval (INEX). INEX provides a forum for the evaluation of information retrieval approaches specifically developed for XML retrieval.

Foundations

Within INEX, the aim of an XML retrieval system is "to exploit the logical structure of XML documents to determine the best document components, i.e., best XML elements, to return as answers to queries" [7]. Query languages have been developed in order to allow users to specify the nature of these best components. Indexing strategies have been developed to obtain a representation not only of the content of XML documents, but their structure. Ranking strategies have been developed to determine the best elements for a given query.

Query Languages

In XML retrieval, the logical document structure is additionally used to determine which document components are most meaningful to return as query answers. With appropriate query languages, this structure can be specified by the user. For example, "I want a paragraph discussing penguins near to a picture labeled Otago Peninsula." Here, "penguins" and "Otago Peninsula" specify *content* (textual) constraints, whereas "paragraph" and "picture" specify *structural* constraints on the retrieval units.

Query languages for XML retrieval can be classified into content-only and content-and-structure query languages. Content-only queries have historically been used as the standard form of input in information retrieval. They are suitable for XML search scenarios where the user does not know (or is not concerned with) the logical structure of a document. Although only the content aspect of an information need can be specified, XML retrieval systems must still determine the best granularity of elements to return to the user.

Content-and-structure queries provide a means for users to specify conditions referring both to the *content* and the *structure* of the sought elements. These conditions may refer to the content of specific elements (e.g., the returned element must contain a section about a particular topic), or may specify the type of the requested answer elements (e.g., sections should be retrieved). There are three main categories of content-and-structure query languages [1]:

1. Tag-based queries allow users to annotate words in the query with a single tag name that specifies the type of results to be returned. For example the query "section:penguins" requests section elements on "penguins."
2. Path-based queries are based upon the syntax of XPath. They encapsulate the document structure in the query. An example the NEXI language is: "//document[about(.,Otago Peninsula)]//section [about(.//title, penguins)]." This query asks for sections that have a title about "penguins," and that are contained in a document about "Otago Peninsula."
3. Clause-based queries use nested clauses to express information needs, in a similar way to SQL. The most prominent clause-based language for XML retrieval is XQuery. A second example is XQuery Full-Text, which extends XQuery with text search predicates such as proximity searching and relevance ranking.

The complexity and the expressiveness of content-and-structure query languages increases from tag-based to clause-based queries. This increase in expressiveness and complexity often means that content-and-structure queries are viewed as too difficult for end users (because, they must, for example, be intimate with the document structure). Nonetheless they can be very useful for expert users in specialized scenarios, and also have been used as an intermediate between a graphical query language (such as Bricks [12]) and an XML search engine.

Indexing Strategies

Classical indexing methods in information retrieval make use of term statistics to capture the importance of a term in a document; and consequently for discriminating between relevant and non-relevant content. Indexing methods for XML retrieval require similar term statistics, but for each element. In XML retrieval there are no a priori fixed retrieval units. The whole document, one of its sections, or a single paragraph within a section, all constitute potential answers to a single query. The simplest approach to allow the retrieval of elements at any level of granularity is to index each element separately (as a separate document in the traditional sense). In this case, term statistics for each element are calculated from the text of the element and all its descendants.

An alternative is to derive the term statistics through the aggregation of term statistics of the element own text, and those of each of its children. A second alternative is to only index leaf elements and to score non-leaf elements through propagation of the score of their children elements. Both alternatives can include additional parameters incorporating, for instance, element relationships or special behavior for some element types.

It is not uncommon to discard elements smaller than some given threshold. A single italicized word, for example, may not be a meaningful retrieval unit. A related strategy, selective indexing, involves building separate indexes for those element types previously seen to carry relevant information (sections, subsections, etc, but not italics, bold, etc.). With selective indexing the results from each index must be merged to provide a single ranked result list across all element types.

It is not yet clear which indexing strategy is the best. The best approach appears to depend on the collection, the types of elements (i.e., the DTD) and their relationships. In addition, the choice of the indexing strategy currently has an effect on the ranking strategy. More details about indexing strategies can be found in the entry on Indexing Units.

Ranking Strategies

XML documents are made of XML elements, which define the logical document structure. Thus sophisticated ranking strategies can be developed to exploit the various additional (structural) evidence not seen in unstructured (flat) text documents.

Element Scoring Many of the retrieval models developed for flat document retrieval have been adapted for XML retrieval. These models have been used to estimate the relevance of an element based on the evidence associated with the element only. This is done by a scoring function based, for instance, on the vector space, BM25, the language model, and so on. They are typically adapted to incorporate XML-specific features. As an illustration, a scoring function based on language models [6] is described next:

Given a query $q = t_1,\ldots,t_n$ made of n terms, an element e and its corresponding element language

model θe, the element *e* is ranked using the following scoring function:

$$P(e|q) \propto P(e) \times P(q|\theta_e)$$

where *P(e)* is the prior probability of relevance for element *e* and *P(q|θe)* is the probability of the query *q* being "generated" by the element language model and is calculated as follows:

$$P(t_1,...,t_n|\theta_e) = \prod_{i=1}^{n} \lambda p(t_i|e) + (1-\lambda)p(t_i|C)$$

Here $P(t_i|e)$ is the maximum likelihood estimate of term t_i in element *e*, $P(t_i|C)$ is the probability of query term in the collection, and λ is the smoothing parameter. $P(t_i|e)$ is the element model based on element term frequency, whereas $P(t_i|C)$ is the collection model based on inverse element frequency. An important XML-specific feature is element length, since this can vary radically – for example, from a title to a paragraph to a document section. Element length can be captured by setting, the prior probability $P(e)$, as follows:

$$p(e) = \frac{length(e)}{\sum_{C} length(e)}$$

$length(e)$ is the length of element *e*. Including length in the ranking calculation has been shown to lead to more effective retrieval than not doing so.

Contextualization The above strategy only scores an element based on the content of the element itself. Considering additional evidence has shown to be beneficial for XML retrieval. In particular for long documents, using evidence from the element itself as well as its context (for example the parent element) has shown to increase retrieval performance. This strategy is referred to as contextualization. Combining the element score and a separate document score has also been shown to improve performance.

Propagation When only leaf elements are indexed, a propagation mechanism is used to calculate the relevance score of the non-leaf elements. The propagation combines the retrieval scores of the leaf elements (often using a weighted sum) and any additional element characteristics (such as the distance between the element and the leaves). A non-trivial issue is the estimation of the weights for the weighted sum.

Merging It has also been common to obtain several ranked lists of results, and to merge them to form a single list. For example, with the selective indexing strategy [10], a separate index is created for each a priori selected type of element (such as article, abstract, section, paragraph, and so on). A given query is then submitted to each index, each index produces a separate list of elements, normalization is performed (to take into account the variation in size of the elements) and the results are merged. Another approach to merging produces several ranked lists from a single index and for all elements in the collection (a single index is used as opposed to separate indices for each element). Different ranking models are used to produce each ranked list. This can be compared to document fusion investigated in the 1990s.

Processing Structural Constraints Early work in XML retrieval required structural constraints in content-and-structure queries to be strictly matched but specifying an information need including structural constraints is difficult; XML document collections have a wide variety of tag names. INEX now views structural constraints as hints as to where to look (what sort of elements might be relevant). Simple techniques for processing structural constraints include the construction of a dictionary of tag synonyms and structure boosting. In the latter, the retrieval score of an element is generated ignoring the structural constraint but is then boosted if the element matches the structural constraint. More details can be found in the entry on Processing Structural Constraints.

Processing Overlaps It is one task to provide a score expressing how relevant an element is to a query but a different task to decide which of a set of several overlapping relevant elements is the best answer. If an element has been estimated relevant to a query, it is likely that its parent element is also estimated relevant to the query as these two elements share common text. But, returning chains of elements to the user should be avoided to ensure that the user does not receive the same text several times (one for each element in the

chain). Deciding which element to return depends on the application and the user model. For instance, in INEX, the best element is one that is highly relevant, but also specific to the topic of request (i.e., does not discuss unrelated topics).

Removing overlap has mostly been done as a post-ranking task. A first approach, and the most commonly adopted one, is to remove elements directly from the result list. This is done by selecting the highest ranked element in a chain and removing any ancestors and descendents in lower ranks. Other techniques looked at the distribution of retrieved elements within each document to decide which ones to return. For example, the root element would be returned if all retrieved elements were uniformly distributed in the document. This technique was shown to outperform the simpler techniques. Efficiency remains an issue as the removal of overlaps is done at query time. More details can be found in the entry on Processing Overlaps.

Key Applications

XML retrieval approaches (from query languages to ranking strategies) are relevant to any applications concerned with the access to repositories of documents annotated in XML, or similar mark-up languages such as SGML or ASN.1. Existing repositories include electronic dictionaries and encyclopedia such as the Wikipedia [4], electronic journals such as the journals of the IEEE [7], plays such as the collected works of William Shakespeare [8], and bibliographic databases such as PubMed. (www.ncbi.nlm.nih.gov/pubmed/) XML retrieval is becoming increasingly important in all areas of information retrieval, the application to full-text book searching is obvious and such commercial systems already exist [11].

Experimental Results

Since 2002 work on XML retrieval has been evaluated in the context of INEX. Many of the proposed approaches have been presented at the yearly INEX workshops, held in Dagsthul, Germany. Each year, the INEX workshop pre-proceedings (which are not peer-reviewed) contain preliminary papers describing the details of participant's approaches. Since 2003 the final INEX workshop proceedings have been peer-reviewed, and since 2004 they have been published by Springer as part of the Lecture Notes in Computer Science series. Links to the pre- and final proceedings can be found on the INEX web site (http://www.inex.otago.ac.nz/).

Data Sets

Since 2002 INEX has collected data sets that can be used for conducting XML retrieval experiments [9]. Each data set consists of a document collection, a set of topics, and the corresponding relevance assessments. The topics and associated relevance assessments are available on the INEX web site (http://www.inex.otago.ac.nz/). It should be noted that the relevance assessments on the latest INEX data set are released first to INEX participants.

Cross-references

- ▶ Aggregation-Based Structured Text Retrieval
- ▶ Content-and-Structure Query
- ▶ Evaluation Metrics for Structured Text Retrieval
- ▶ Indexing Units
- ▶ Information Retrieval Models
- ▶ INitiative for the Evaluation of XML Retrieval
- ▶ Integrated DB&IR Semi-Structured Text Retrieval
- ▶ Logical Structure
- ▶ Narrowed Extended XPath I
- ▶ Presenting Structured Text Retrieval Results
- ▶ Processing Overlaps
- ▶ Processing Structural Constraints
- ▶ Propagation-Based Structured Text Retrieval
- ▶ Relationships in Structured Text Retrieval
- ▶ Structure Weight
- ▶ Structured Document Retrieval
- ▶ Structured Text Retrieval Models
- ▶ Term Statistics for Structured Text Retrieval
- ▶ XML
- ▶ XPath/XQuery
- ▶ XQuery Full-Text

Recommended Reading

1. Amer-Yahia S. and Lalmas M. XML search: languages, INEX and scoring. ACM SIGMOD Rec., 35(4):16–23, 2006.
2. Baeza-Yates R., Fuhr N., and Maarek Y.S. (eds.). Special issue on XML retrieval, ACM Trans. Inf. Syst., 24(4), 2006.
3. Blanken H.M., Grabs T., Schek H.-J., Schenkel R., and Weikum G. (eds.). Intelligent Search on XML Data, Applications, Languages, Models, Implementations, and Benchmarks, Springer, Berlin, 2003.

4. Denoyer L. and Gallinari P. The Wikipedia XML corpus, comparative evaluation of XML information retrieval systems. In Proc. 5th Int. Workshop of the Initiative for the Evaluation of XML Retrieval, 2007, pp. 12–19.
5. Fuhr N. and Lalmas M. (eds.). Special issue on INEX, Inf. Retrieval, 8(4), 2005.
6. Kamps J., de Rijke M., and Sigurbjörnsson B. The importance of length normalization for XML retrieval. Inf. Retrieval, 8(4):631–654, 2005.
7. Kazai G., Gövert N., Lalmas M., and Fuhr N. The INEX Evaluation Initiative. In Intelligent search on XML data, applications, languages, models, implementations, and benchmarks, H.M. Blanken, T. Grabs, H. Schek, R. Schenkel, G. Weikum (eds.). Springer, 2003, pp. 279–293.
8. Kazai G., Lalmas M., and Reid J. Construction of a test collection for the focused retrieval of structured documents, In Proc. 25th European Conf. on IR Research, 2003, pp. 88–103.
9. Lalmas M. and Tombros A. INEX 2002–2006: understanding XML retrieval evaluation. In Proc. 1st Int. DELOS Conference, 2007, pp. 187–196.
10. Mass Y. and Mandelbrod M. Component ranking and automatic query refinement for XML retrieval. In Proc. 3rd Int. Workshop of the Initiative for the Evaluation of XML Retrieval, 2004, pp. 73–84.
11. Pharo N. and Trotman A. The use case track at INEX 2006. SIGIR Forum, 41(1): 64–66, 2007.
12. van Zwol R., Baas J., van Oostendorp H., and Wiering F. Bricks: the building blocks to tackle query formulation in structured document retrieval. In Proc. 28th European Conf. on IR Research, 2006, pp. 314–325.

XML Schema

Michael Rys
Microsoft Corporation, Sammamish, WA, USA

Synonyms

XML schema; W3C XML schema

Definition

XML Schema is a schema language that allows to constrain XML documents and provides type information about parts of the XML document. It is defined in a World Wide Web Consortium recommendation [3–5].

Key Points

XML Schema is a schema language that allows the constraint of documents and provides type information about parts of the XML document. It is defined in a World Wide Web Consortium recommendation [3–5]. The current version is 1.0.

Unlike document type descriptions (DTDs) that have been defined in the XML recommendation [1], XML Schema is using an XML based vocabulary to describe the schema constraints.

A schema describes elements and attributes and their content models and types. It provides for complex types that describe the content model of elements and simple types that describe the type of attributes and leaf element nodes. Types can be related to each other in so called derivation hierarchies, either by restricting or extending a super type. Elements with different names can be grouped together into substitution groups if their types are in a derivation relationship.

The element, attribute and type declarations are called schema components. A schema is normally associated with a target namespace to which these schema components belong. Figure 1 depicts an example schema (based on examples in [10]) that defines a schema for the target namespace http://www.example.com/PO1 containing the following schema components: two global element declarations, three global complex type declarations and 1 global simple type declaration:

Besides constraining XML documents, XML Schemas are also being used to define vocabularies and semantic models, often in the context of information exchange scenarios to define and constrain the data contracts for data exchanged between clients and services. The schema components can in addition be used for providing additional type information in XQuery [7,8], XPath [2,8] and XSLT [9].

Note that version 1.1 of XML Schema is currently under development at the W3C [6,10].

Cross-references

- ▶ XML
- ▶ XML Attribute
- ▶ XML Document
- ▶ XML Element
- ▶ XPath/XQuery
- ▶ XQuery/XQuery
- ▶ XSLT/XSLT

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:po="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1"
        elementFormDefault="unqualified"
        attributeFormDefault="unqualified">

  <xs:element name="purchaseOrder" type="po:PurchaseOrderType"/>
  <xs:element name="comment"       type="xs:string"/>

  <xs:complexType name="PurchaseOrderType">
    <xs:sequence>
      <xs:element name="shipTo" type="po:USAddress"/>
      <xs:element name="billTo" type="po:USAddress"/>
      <xs:element ref="po:comment" minOccurs="0"/>
      <xs:element name="items"  type="po:Items"/>
    </xs:sequence>
    <xs:attribute name="orderDate" type="xs:date"/>
  </xs:complexType>

  <xs:complexType name="USAddress">
    <xs:sequence>
      <xs:element name="name"   type="xs:string"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city"   type="xs:string"/>
      <xs:element name="state"  type="xs:string"/>
      <xs:element name="zip"    type="xs:decimal"/>
    </xs:sequence>
    <xs:attribute name="country" type="xs:NMTOKEN"
                  fixed="US"/>
  </xs:complexType>

  <xs:complexType name="Items">
    <xs:sequence>
      <xs:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="productName" type="xs:string"/>
            <xs:element name="quantity">
              <xs:simpleType>
                <xs:restriction base="xs:positiveInteger">
                  <xs:maxExclusive value="100"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name="USPrice"  type="xs:decimal"/>
            <xs:element ref="po:comment"   minOccurs="0"/>
            <xs:element name="shipDate" type="xs:date" minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="partNum" type="po:SKU" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="SKU">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{3}-[A-Z]{2}"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

XML Schema. Figure 1. Example XML Schema.

Recommended Reading

1. XML 1.0 Recommendation, latest edition. Available at: http://www.w3.org/TR/xml
2. XML Path Language (XPath) 2.0, latest edition. Available at: http://www.w3.org/TR/xpath20/
3. XML Schema Part 0: Primer, latest edition. Available at: http://www.w3.org/TR/xmlschema-0/
4. XML Schema Part 1: Structures, latest edition. Available at: http://www.w3.org/TR/xmlschema-1/
5. XML Schema Part 2: Datatypes, latest edition. Available at: http://www.w3.org/TR/xmlschema-2/
6. XML Schema 1.1 Part 2: Datatypes, latest edition. Available at: http://www.w3.org/TR/xmlschema11–2/
7. XQuery 1.0: An XML Query Language, latest edition. Available at: http://www.w3.org/TR/xquery/
8. XQuery 1.0 and XPath 2.0 Data Model (XDM), latest edition. Available at: http://www.w3.org/TR/xpath-datamodel/
9. XSL Transformations (XSLT) Version 2.0, latest edition. Available at: http://www.w3.org/TR/xslt20/
10. W3C XML Schema Definition Language (XSDL) 1.1 Part 1: Structures, latest edition. Available at: http://www.w3.org/TR/xmlschema11–1/

XML Schemas

► XML Types

XML Selectivity Estimation

Maya Ramanath[1], Juliana Freire[2], Neoklis Polyzotis[3]
[1]Max-Planck Institute for Informatics, Saarbrücken, Germany
[2]School of Computing, University of Utah, UT, USA
[3]University of California Santa Cruz, Santa Cruz, CA, USA

Synonyms

XML cardinality estimation

Definition

Selectivity estimation in database systems refers to the task of estimating the number of results that will be output for a given query. Selectivity estimates are crucial in query optimization, since they enable optimizers to select efficient query plans. They are also employed in interactive data exploration as timely feedback about the expected outcome of user queries, and can even serve as approximate answers for count queries.

Selectivity estimators apply an *estimation procedure* on a *synopsis* of the data. Due to the stringent time and space constraints of query optimization, of which selectivity estimation is only one of the steps, selectivity estimators are faced with two, often conflicting, requirements: they have to accurately and efficiently estimate the cardinality of queries while keeping the synopsis size to a minimum.

While there is a large body of literature on selectivity estimation in the context of relational databases, the inherent complexity of the XML data model creates new challenges. Synopsis structures for XML must capture statistical characteristics of document structure in addition to those of data values. Moreover, the flexibility allowed by the use of regular expressions to define XML elements typically results in data that has *highly-skewed structure*. Finally, queries over XML documents require path expressions over possibly long paths, which essentially translates to a number of joins that is much larger than what is found in typical database applications. This increases the scope for inaccurate estimates, since it is known that errors propagate rapidly when estimating multi-way join queries.

Historical Background

Selectivity estimation for XML queries has its roots in early works on *semi-structured data* [7]. These include both lossy techniques for structural summarization [9] and lossless techniques for deriving structural indices [4,8]. The development of XML and associated query languages (XPath and XQuery) has created new challenges to the problem of selectivity estimation. In particular, XML query languages involve a richer set of structure- and content-related operators and hence require more sophisticated techniques for synopsis generation and estimation. A similar observation can be made for the XML data model itself, which is more involved compared to the early models for semi-structured data. This diversity has given rise to a host of different XML selectivity estimation techniques that address different aspects of the general problem. These techniques can be characterized with respect to a number of different features, including: information captured in the synopsis structure (e.g., tree- vs. graph-based view of XML data, structure only vs. structure and content); the class of supported queries (e.g., simple path expressions, XPath); the use of schema information; the ability to provide accuracy guarantees; synopsis generation strategy (e.g., streaming vs.

non-streaming); support for maintaining the synopsis in the presence of updates to the data.

Foundations

In general, XML selectivity estimation techniques consist of two components: a synopsis structure that captures the statistical features of the data, and an estimation framework that computes selectivity estimates based on the constructed synopsis. The techniques presented in the recent literature cover a wide spectrum of design choices in terms of these two components. For instance, Bloom histograms [17] and Markov tables [1] base the synopsis on an enumeration of *simple paths*, while XSketch [11], XSeed [19], and StatiX [3] build synopses structures that capture the *branching path structure* of the data set. Other points of variation include: the supported query model, e.g., structure-only queries [1] vs. queries with *value predicates* [3,11,10]), or linear queries [1] vs. twig queries [2,12]; the use of schema information [3,18]; probabilistic guarantees on estimation error [17]; explicit support for data updates [5,13,17].

Table 1 provides an overview of the current literature on XML selectivity estimation techniques. A detailed description of each technique is beyond the scope of this entry. Instead, the following classification of techniques is discussed: synopses based on path enumeration, graph-based synopses, and updatable synopses.

Path Enumeration

A straightforward approach to handle *simple path expression* queries is to enumerate all paths in the data and store the count for each path. Of course, the number of paths in the data can be very large and thus summarization techniques are required to concisely store this information. Markov tables [1] record information on paths of up to specified length m only, and further reduce the number of paths by deleting paths with low frequencies. The selectivity of path expressions for paths with length less than or equal to m is derived by a lookup of this table. For longer path expressions, selectivity is computed by combining the counts of shorter paths. In contrast, the bloom histogram method, proposed by Wang et al. [17], groups together simple paths with "similar" frequencies into buckets of a histogram. Each bucket of paths is then summarized using a bloom filter to represent the contents (paths) of the bucket and a representative count is associated with it. Selectivities are estimated by first identifying the bucket containing the query path and retrieving its count. Estimation errors are bounded and depend on the number of histogram buckets and the size of the bloom filter.

While the above techniques use path enumeration as a basic operation to estimate the selectivity of simple path expressions, the same concept is used in [2] to answer more complex "twig" queries, i.e., path queries with branches. The idea is to record, for each path, a small fixed-length signature of the element ids that constitute its roots. These signatures can be "intersected" in order to estimate the number of common roots for several simple paths, or equivalently, the number of matches for the twig query that is formed by joining the simple paths at their root. The set of simple paths and their counts as well as their signatures are organized into a summary data structure called the *correlated subtree* (CST). Given a twig query, it is first broken down in a set of constituent simple paths based on the CST, and the corresponding signatures and counts are combined in order to estimate the overall selectivity.

Graph-Synopsis-Based Techniques

At an abstract level, a graph synopsis summarizes the basic graph structure of an XML document. More formally, given a data graph $G = (V_G, E_G)$, a graph synopsis $\mathcal{S}(G) = (V_S, E_S)$ is a directed node-labeled graph, where (i) each node $v \in V_S$ corresponds to a subset of element (or attribute) nodes in V_G (termed the *extent* of) that have the *same label*, and (ii) an edge in $(u,v) \in E_G$ is represented in E_S as an edge between the nodes whose extents contain the two endpoints u and v. For each node u, the graph synopsis records the common tag of its elements and a count field for the size of its extent.

In order to capture different properties of the underlying path structure and value content, a graph-synopsis is augmented with appropriate, localized distribution information. As an example, the XSketch-summary mechanism [11], which can estimate the selectivity of simple path expressions with branching predicates, augments the general graph-synopsis model with: (i) localized per-edge stability information, indicating whether the synopsis edge is *backward- and/or forward-stable*, and, (ii) localized per-node value distribution summaries. In short, an edge (u,v) in the synopsis is said to be *forward-stable* if all the elements of u have at least one child in v; similarly, (u,v) is backward stable if all the elements in v have at least one parent in u. Accordingly,

XML Selectivity Estimation. Table 1. Summary of work on XML selectivity estimation

Proposal	Input	Summary structure	Structure predicates	Value predicates	Updates	Error guarantees
Chen et al. [2]	Data	Correlated subpath tree	Tree pattern	Substring	No	No
Aboulnaga et al. [1]	Data	Path tree and Markov tables	Simple paths	No	No	No
XPathLearner [5]	Query feedback	Markov tables	Simple Paths	Equality Predicates	Yes	No
CXHist [6]	Query feedback	Markov tables	Simple Paths	Substrings	Yes	No
Wu et al. [18]	Data, Schema, Predicates	Position Histogram	Tree pattern	General predicates	No	No
XSketches [11]	Data	Graph	Tree Pattern	Numerical range	No	Yes
XSeed [19]	Data	Graph	Tree pattern	No	No	Yes
XCluster [10]	Data	Graph	Tree pattern	Numerical range, Term Containment and Substring	No	No
StatiX [3]	Data and Schema	Schema graph and Histograms	Tree Pattern	Numerical Range	No	No
Sartiani [15]	Data and Schema	Tagged region graph	Tree Pattern	General Predicates	No	No
Bloom Histograms [17]	Data	Bloom Filter	Simple Paths	No	Yes	Yes
IMAX [13]	Data and Schema	Schema graph and Histogram	Tree pattern	Numerical Range	Yes	No
Wang et al. [16]	Data	Histograms on data samples	Simple Paths	No	No	No
SketchTree [14]	Data Stream	Randomized Sketch	Tree pattern with child axis	General predicates	Yes	Yes

for each node u that represents elements with values, the synopsis records a summary $H(u)$ which captures the corresponding value distribution and thus enables selectivity estimates for value-based predicates.

Recent studies [10,12,19] have proposed a variant of graph-synopses that employ a clustering-based model in order to capture the path and value distribution of the underlying XML data. Under this model, each synopsis node is viewed as a "cluster" and the enclosed elements are assumed to be represented by a corresponding "centroid" that aggregates their characteristics. The TreeSketch [12] model, for instance, defines the centroid of a node u as a vector of average child counts $(c_1,c_2,...,c_n)$, where c_j is the average child count from elements in u to every other node v_j in the synopsis. Thus, the assumption is that each element in u has exactly c_j children to node v_j. Furthermore, the clustering error, that is, the difference between the actual child counts in u and the centroid, provides a measure of the error of approximation.

While the above techniques derive the graph structure from the data itself, StatiX [3] exploits the graph structure provided by the XML Schema to build synopses. In a way, the schema can be regarded as a synopsis of the data since it describes the general structure of the data. StatiX makes use of the mapping between an element in the data and a type in the schema (typically assigned during the validation phase) in order to build its synopsis structure. In addition, the statistics gathering phase consists of assigning ordinal numbers to each type found in the data. The schema graph, which forms a basic synopsis, is augmented with histograms

which capture parent-child distributions (built using ordinal numbers) and value distributions (built using values occurring in the data). By appropriately tuning the number of types through a set of *schema transformations* and by decreasing or increasing the number of histogram buckets, coarse to fine-grained synopsis structures can be built. Selectivities of branching path expressions with value predicates can be estimated using either a simple lookup (possible for simple path expressions where the return value for the query corresponds to a single type) or histogram-based cardinality estimation techniques to combine value histograms with parent-child distributions (needed for branching path expressions with value predicates).

Updatable Synopses

Many selectivity estimation techniques for XML assume that the underlying data are *static* and require an *offline scan* over the data in order to gather statistics and build synopsis structures. Although this assumption is valid for applications that primarily use XML as a document exchange format, it does not hold for scenarios where XML is used as a native storage format. For these, it is important that estimators support *synopsis maintenance* in addition to synopsis construction.

The Bloom Histogram technique [17] outlined in Table 1 supports updates by maintaining a "full" summary (maybe on disk) and rebuilding the bloom histogram periodically. That is, each update is first parsed into paths, and a table of paths is updated with the new cardinalities. The bloom histogram which is used for selectivity estimation will be rebuilt from this path table when a sufficient number of updates have been received and the bloom histogram estimates are no longer reliable.

IMAX [13], which is built on top of StatiX [3], updates the synopsis structure directly as and when the update is received. The key concept in this technique is to first identify the correct location of the update (corresponding to the ordinal numbers of the types in the update) and then to estimate and update the correct buckets of the required histograms. It is possible to estimate the amount of error in the histograms and to schedule a re-computation of the required set of histograms from the data if the error becomes too large.

While the approaches outlined above respond directly to changes in the data, XPathLearner [5] and CXHist [6] observe the query processor and use a *query feedback loop* to maintain the synopsis structure. In short, these techniques obtain the true selectivity of each query as it is processed, and use this information in order to update the synopsis accordingly. As a result, the synopsis becomes more refined with respect to frequent queries and can thus provide more accurate selectivity estimates for a significant part of the workload.

Key Applications

The main utility of cardinality estimation is to serve as input into a query optimizer which determines the best execution plan for a query. These estimates can also serve as approximate answers to "COUNT" queries. In addition, they are used in interactive data exploration in order to provide users with early feedback regarding the number of results to the submitted query.

Cross-references

- ▶ Query Optimization
- ▶ Query Processing
- ▶ XML
- ▶ XML Tree Pattern, XML Twig Query
- ▶ XPath/XQuery

Recommended Reading

1. Aboulnaga A., Alameldeen A.R., and Naughton J. Estimating the selectivity of XML path expressions for internet scale applications. In Proc. 27th Int. Conf. on Very Large Data Bases, 2001, pp. 591–600.
2. Chen Z., Jagadish H.V., Korn F., Koudas N., Muthukrishnan S., Ng R.T., and Srivastava D. Counting twig matches in a tree. In Proc. 17th Int. Conf. on Data Engineering, 2001, pp. 453–462.
3. Freire J., Haritsa J., Ramanath M., Roy P., and Siméon J. StatiX: Making XML Count. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 181–191.
4. Goldman R. and Widom J. Dataguides: enabling query formulation and optimization in semistructured databases. In Proc. 23th Int. Conf. on Very Large Data Bases, 1997, pp. 436–445.
5. Lim L., Wang M., Padmanabhan S., Vitter J., and Parr R. XPathLearner: An on-line self-tuning markov histogram for XML path selectivity estimation. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 442–453.
6. Lim L., Wang M., and Vitter J. CXHist: an on-line classification-based histogram for XML string selectivity estimation. In Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 1187–1198.
7. McHugh J., Abiteboul S., Goldman R., Quass D., and Widom J. A database management system for semistructured data. ACM SIGMOD Rec., 26(3):54–66, September 1997.
8. Milo T. and Suciu D. Index structures for path expressions. In Proc. 7th Int. Conf. on Database Theory, 1999, pp. 277–295.
9. Nestorov S., Ullman J., Wiener J., and Chawathe S. Representative objects: concise representations of semistructured,

hierarchical data. In Proc. 13th Int. Conf. on Data Engineering, 1997, pp. 79–90.
10. Polyzotis N. and Garofalakis M. XCluster Synopses for structured XML content. In Proc. 22nd Int. Conf. on Data Engineering, 2006, p. 63.
11. Polyzotis N. and Garofalakis M. XSketch synopses for XML data graphs. ACM Trans. on Database Syst., 31(3):1014–1063, 2006.
12. Polyzotis N., Garofalakis M., and Ioannidis Y. Approximate XML query answers. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 263–274.
13. Ramanath M., Zhang L., Freire J., and Haritsa J. IMAX: incremental maintenance of schema-based XML statistics. In Proc. 21st Int. Conf. on Data Engineering, 2005, pp. 273–284.
14. Rao P. and Moon B. Sketchtree: approximate tree pattern counts over streaming labeled trees. In Proc. 22nd Int. Conf. on Data Engineering, 2006, p. 80.
15. Sartiani C. A framework for estimating XML query cardinality. In Proc. 6th Int. Workshop on the World Wide Web and Databases, 2003, pp. 43–48.
16. Wang W., Jiang H., Lu H., and Yu J.X. Containment join size estimation: models and methods. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 145–156.
17. Wang W., Jiang H., Lu H., and Yu J.X. Bloom histogram: path selectivity estimation for XML data with updates. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004, pp. 240–251.
18. 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.
19. Zhang N., Özsu M.T., Aboulnaga A., and Ilyas I.F. XSEED: accurate and fast cardinality estimation for XPath queries. In Proc. 22nd Int. Conf. on Data Engineering, 2006, p. 61.

XML Storage

Denilson Barbosa[1], Philip Bohannon[2], Juliana Freire[3], Carl-Christian Kanne[4], Ioana Manolescu[5], Vasilis Vassalos[6], Masatoshi Yoshikawa[7]
[1]University of Alberta, Edmonton, AB, Canada
[2]Yahoo! Research, Santa Clara, CA, USA
[3]University of Utah, Salt Lake City, UT, USA
[4]University of Mannheim, Mannheim, Germany
[5]INRIA Saclay–Île-de-France, Orsay, France
[6]Athens University of Economics and Business, Athens, Greece
[7]University of Kyoto, Kyoto, Japan

Synonyms

XML persistence; XML database

Definition

A wide variety of technologies may be employed to physically persist XML documents for later retrieval or update, from relational database management systems to hierarchical systems to native file systems. Once the target technology is chosen, there is still a large number of *storage mapping* strategies that define how parts of the document or document collection will be represented in the back-end technology. Additionally, there are issues of optimization of the technology and strategy used for the mapping. XML Storage covers all the above aspects of persisting XML document collections.

Historical Background

Even though the need for XML storage naturally arose after the emergence of XML, similar techniques had been developed earlier, since the mid-1990's, to store semi-structured data. For example, the LORE system included a storage manager specifically designed for semi-structured objects, while the STORED system allowed the definition of mappings from semi-structured data to relations. Even earlier, storage techniques and storage systems had been developed for object-oriented data. These techniques focused on storing individual objects, including their private and public data and their methods. Important tasks included performing garbage collection, managing object migration and maintaining class extents. Object clustering techniques were developed that used the class hierarchy and the *composition* hierarchy (i.e., which object is a component of which other object) to help determine object location. These techniques, and the implemented object storage systems, such as the O2 storage system, influenced the development of subsequent semi-structured and XML storage systems.

Moreover, the above solutions or ad-hoc approaches had also been used for the storage of large SGML (Standard Generalized Markup Language, a superset and precursor to XML) documents.

Foundations

Given the wide use of XML, most applications need or will need to process and manipulate XML documents, and many applications will need to store and retrieve data from large documents, large collections of documents, or both. As an exchange format, XML can be simply serialized and stored in a file, but serialized document storage often is very inefficient for query processing and updates.

As a result, a large-scale XML storage infrastructure is critical to modern application performance.

Figure 1a shows a simple graphical outline of an XML DTD for movies and television shows.

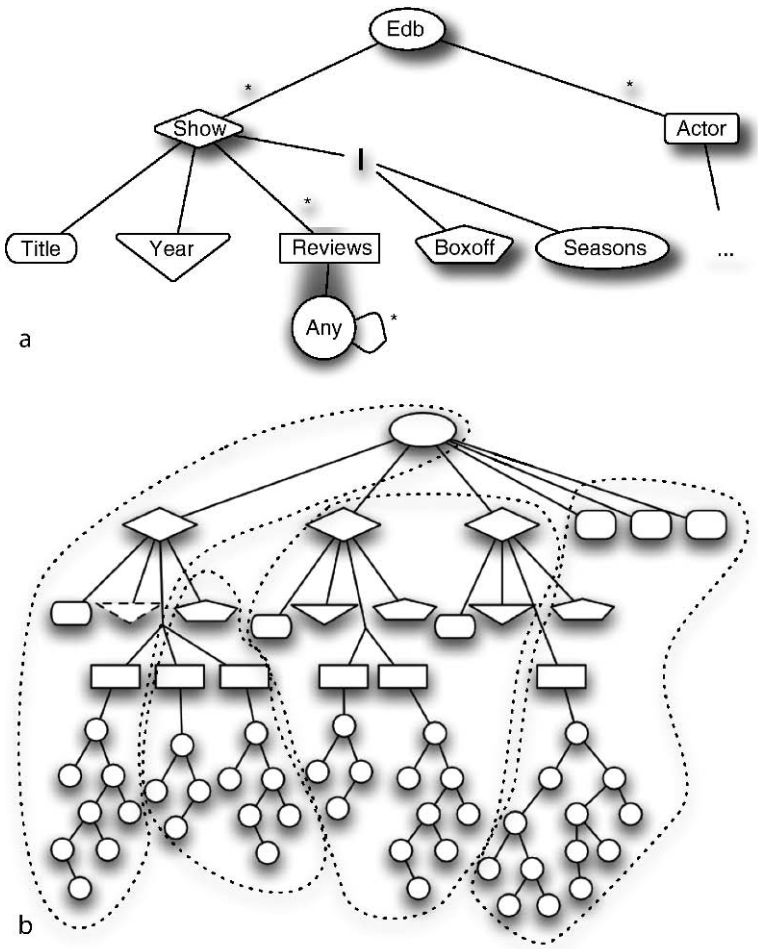
As this example shows, XML data may exhibit great variety in their structure. At one extreme, relational-style data like `title`, `year` and `boxoff` children of `show` in Fig. 1a may be represented in XML. At the opposite extreme are highly irregular structures such as might be found under the `reviews` tag. Figure 2 shows a similar graphical representation of a real-life DTD for scientific articles. Since every HTML structure or formatting element is also an XML element or attribute, the corresponding XML tree is very deep and wide, and no two sections are likely to have the same structure.

XML processing workloads are also diverse. Queries and updates may affect or return few or many nodes. They may also need to "visit" large portions of an XML tree and return nodes that are far apart, such as all the box-office receipts for movies, or may only return or affect nodes that are "close" together, such as all the information pertaining to a single review.

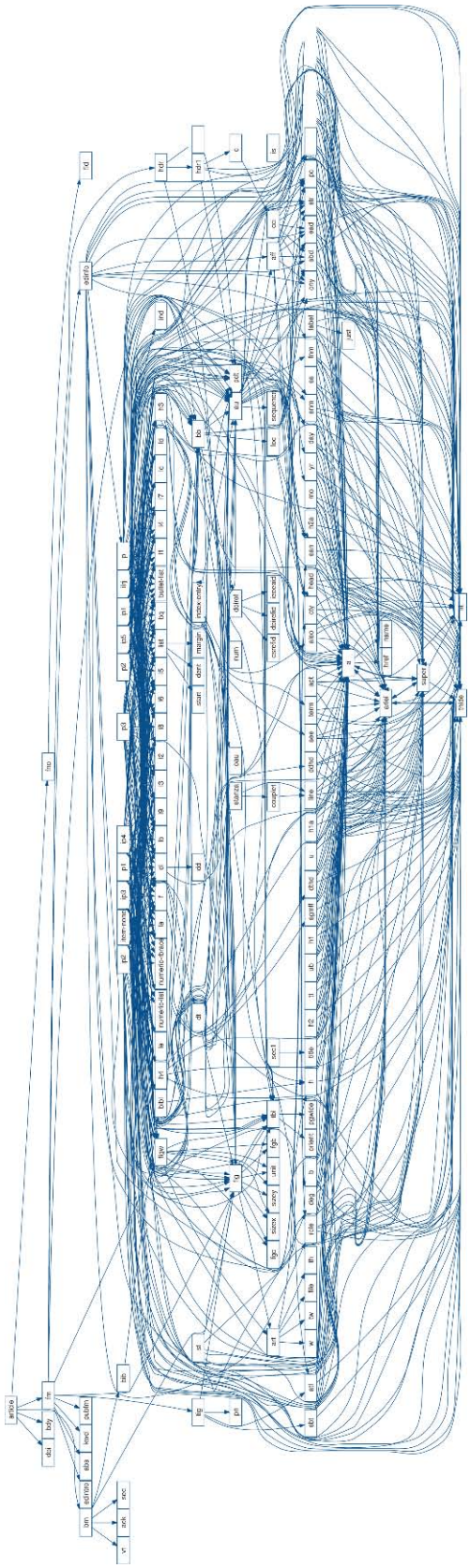
A few different ways of persisting XML document collections are used, and each addresses differently the challenges posed by the varied XML documents and workloads.

Instance-Driven Storage

In *instance-driven storage*, the storage of XML content is driven by the tree structure of the individual document, ignoring the types assigned to the nodes by a schema (if one exists). In some cases, e.g., when documents have irregular structure or an application mostly generates navigations to individual elements, instance-driven storage can greatly simplify the task of storing XML content. One instance-driven technique is to



XML Storage. Figure 1. Movie DTD and example native storage strategy.



XML Storage. Figure 2. Graphical outline of a complex article DTD (strongly simplified).

store nodes and edges in one or more relational tables. A second approach is to implement an XML data model natively.

Tabular Storage of Trees A relational schema for encoding any XML instance may include relations `child`, modeling the parent-child relationship, and `tag, attr, id, text`, associating to each element node respectively a tag, an attribute, an identity and a text value, as well as sets that contain the root of the document and the set of all its elements. Notice that such a schema does not allow full reconstruction of an original XML document, as it does not retain information on element order, whitespace, comments, entity references etc. The encoding of element order, which is a critical feature of XML, is discussed later in this article.

A relational schema for encoding XML may also need to capture *built-in integrity constraints* of XML documents, such as the fact that every child has a single parent, every element has exactly one tag, etc.

Tabular storage of trees as described enables the use of relational storage engines as the target storage technology for XML document collections. While capable of storing arbitrary documents, with this approach a large number of joins may be required to answer queries, especially when reconstructing subtrees. This is the basic storage mapping supported by Microsoft's SQL Server as of 2007.

Native XML Storage Native XML storage software implements data structures and algorithms specifically designed to store XML documents on secondary memory. These data structures support one or more of the XML data models. Salient functional requirements implied by standard data models include the preservation of child order, a stable node identity, and support for type information (depending on the data model supported). An additional functional requirement in XML data stores is the ability to reconstruct the exact textual representation of an XML document, including details such as encoding, whitespace, attribute order, namespace prefixes, and entity references.

A native XML storage implementation generally maps tree nodes and edges to storage blocks in a manner that preserves *tree locality*, i.e., that stores parents and children in the same block. The strategy is to map XML tree structures onto records managed by a storage manager for variable-size records. One possible approach is to map the complete document to a single Binary Large Object and use the record manager's large object management to deal with documents larger than a page. This is one of the approaches for XML storage supported by the commercial DBMS Oracle as of 2007. This approach incurs significant costs both for update and for query processing.

A more sophisticated strategy is to divide the document into partitions smaller than a disk block and map each partition to a single record in the underlying store. Large text nodes and large child node lists are handled by chunking them and/or introducing auxiliary nodes. This organization supports efficient local navigation and tree reconstruction without, for example, loading the entire tree into memory. Such an approach is used in the commercial DBMS IBM DB2 as of 2007 (starting with version 9). Native stores can support efficiently updates, concurrency control mechanisms and traditional recovery schemes to preserve durability (see ACID Properties).

Figure 1b shows a hypothetical instance of the schema of Fig. 1a. The types of nodes are indicated by shape. One potential assignment of nodes to physical storage records is shown as groupings inside dashed lines. Note that `show` elements are often physically stored with their `review` children, and reviews are frequently stored with the next or previous review in document order.

Physical-level heuristics that can be implemented to improve performance include compressed representation of node pointers inside a block, and string dictionaries allowing integers to replace strings appearing repeatedly, such as tag names and namespace URIs.

Schema-Driven Storage

When information about the structure of XML documents is given, e.g., in a DTD or an XML Schema (see XML Schema), techniques have been developed for XML storage that exploit this information. In general, nodes of the same type according to the schema are mapped in the same way, for example to a relational table. Schema information is primarily exploited for tabular storage of XML document collections, and in particular in conjunction with the use of a relational storage engine as the underlying technology, as described in the next paragraph. In *hybrid* XML storage different data models, and potentially even different systems, store different document parts.

Relational Storage for XML Documents

Techniques have been developed that enable the effective use of a relational database management system to store XML. Figure 3a illustrates the main tasks that must be performed for storing XML in relational databases. First, the schema of the XML document is mapped into a suitable relational schema that can preserve the information in the original XML documents (Storage Design). The resulting relational schema needs to be optimized at the physical level, e.g., with the selection of appropriate file structures and the creation of indices, taking into account the distinctive characteristics of XML queries and updates in general and of the application workload in particular. XML documents are then shredded and loaded into the flat tables (Data Loading). At runtime, XML queries are translated into relational queries, e.g., in SQL, submitted to the underlying relational system and the results are translated back into XML (Query Translation). Schema-driven relational storage mappings for XML documents are supported by the Oracle DBMS.

An XML-to-relational mapping scheme consists of *view definitions* that express what data from the XML document should appear in each relational table and constraints over the relational schema. The views generally map elements with the same type or tag name to a table and define a *storage mapping*. For example, in Fig. 3b, two views, `V1` and `V2` are used to populate the `Actors` and `Shows` tables respectively. A particular set of storage views and constraints along with physical storage and indexing options together comprise a *storage design*. The process of parsing an XML document and populating a set of relational views according to a storage design is referred to as *shredding*.

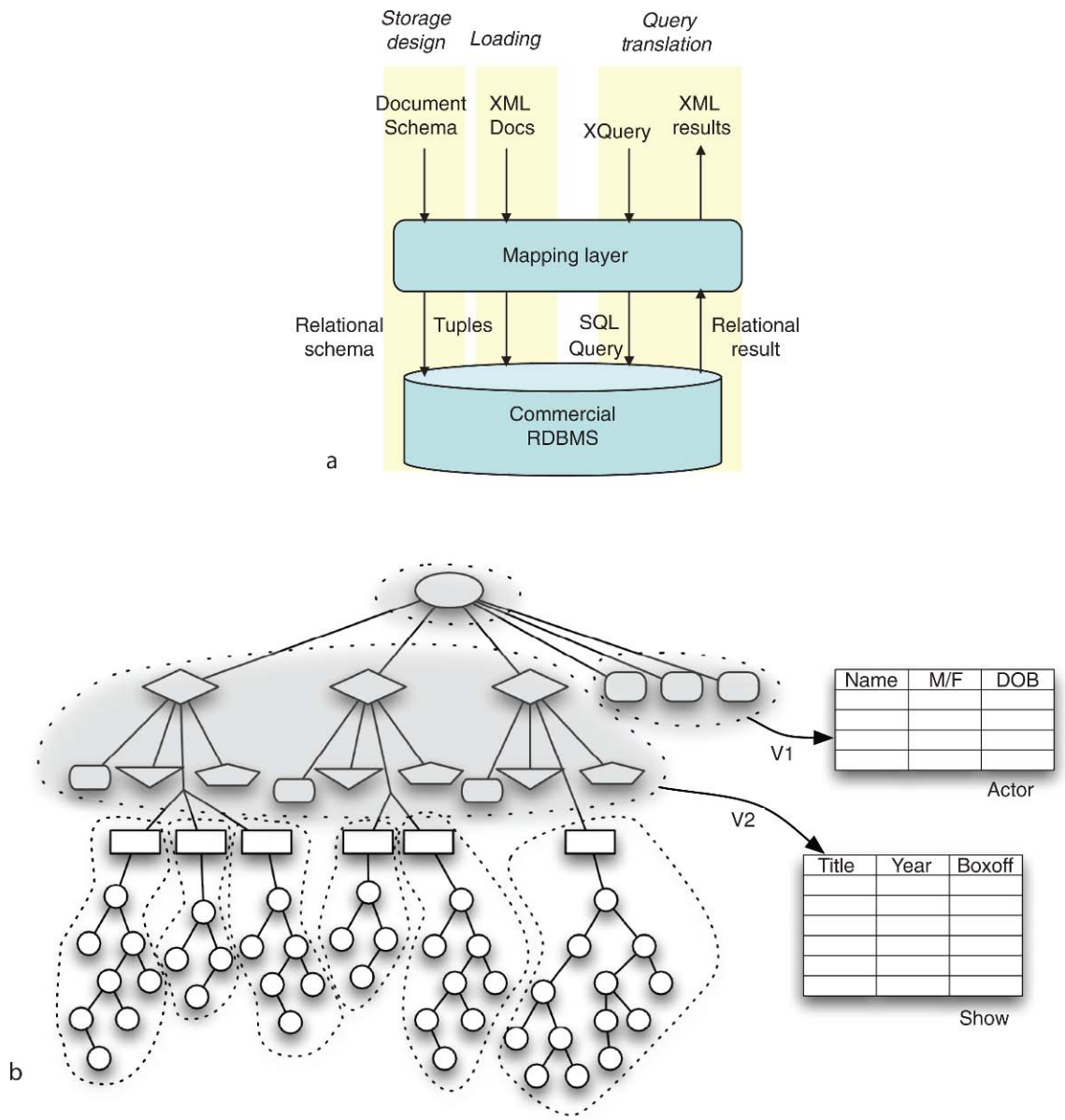
Due to the mismatch between the tree-structure of XML documents and the flat structure of relational tables, there are many possible storage designs. For example, in Fig. 3b, if an element, such as `show`, is guaranteed to have only a single child of a particular type, such as `seasons`, then the child type may optionally be *inlined*, i.e., stored in the same table as the parent.

On the other hand, due to the nature of XML queries and updates, certain indexing and file organization options have been shown to be generally useful. In particular, the use of B-tree indexes (as opposed to hash-based indexes) is usually beneficial, as the translation of XML queries into relational languages often involves range conditions. There is evidence that the best file organization for the relations resulting from XML shredding is index-organized tables, with the index on the attribute(s) encoding the order of XML elements. With such file organization, index scanning allows the retrieval of the XML elements in document order, as required by XPath semantics, with a minimum number of random disk accesses. The use of a path index that stores complete root-to-node paths for all XML elements also provides benefits.

Cost-Based Approaches A key quality of a storage mapping is efficiency – whether queries and updates in the workload can be executed quickly. Cost-based mapping strategies can derive mappings that are more efficient than mappings generated using fixed strategies. In order to apply such strategies, statistics on the values and structure of an XML document collection need to be gathered. A set of transformations and annotations can be applied to the XML schema to derive different schemas that result in different relational storage mappings, for example by merging or splitting the types of different XML elements, and hence mapping them into the same or different relational tables. Then, an efficient mapping is selected by comparing the estimated cost of executing a given application workload on the relational schema produced by each mapping. The optimizer of the relational database used as storage engine can be used for the cost estimation. Due to the size of the search space for mappings generated by the schema transformations, efficient heuristics are needed to reduce the cost without missing the most efficient mappings. Physical database design options, such as vertical partitioning of relations and the creation of indices, can be considered in addition to logical database design options, to include potentially more efficient mappings in the search space.

The basic principles and techniques of cost-based approaches for XML storage are shared with relational cost-based schema design.

Correctness and Losslessness An important issue in designing mappings is correctness, notably, whether a given mapping preserves *enough* information. A mapping scheme is *lossless* if it allows the reconstruction of the original documents, and it is *validating* if all legal relational database instances correspond to a valid XML document. While losslessness is enough for applications involving only queries over the documents, if documents must conform to an XML schema



XML Storage. Figure 3. Relational storage workflow and example.

and the application involves both queries and updates to the documents, schema mappings that are validating are necessary. Many of the mapping strategies proposed in the literature are (or can be extended to be) lossless. While none of them are validating, they can be extended with the addition of constraints to only allow updates that maintain the validity of the XML document. In particular, even though losslessness and validation are undecidable for a large class of mapping schemes, it is possible to guarantee information preservation by designing mapping procedures which guarantee these properties by construction.

Order Encoding Schemes Different techniques have been proposed to preserve the structure and order of XML elements that are mapped into a relational schema. In particular, different labeling schemes have been proposed to capture the positional information of each XML element via the assignment of node labels. An important goal of such schemes is to be able

to express structural properties among nodes, e.g., the child, descendant, following sibling and other relationships, as conditions on the labels. Most schemes are either *prefix-based* or *range-based* and can be used with both schema-driven and instance-based relational storage of XML.

In prefix-based schemes, a node's label includes as a prefix the label of its parent. Dewey-based order encodings are the best known prefix-based schemes. The Dewey Decimal Classification was originally developed for general knowledge classification. The basic *Dewey-based encoding* assigns to each node in an XML tree an identifier that records the position of a node among its siblings, prefixed by the identifier of its parent node. In Fig. 1b, the Dewey-based encoding would assign the identifier 1.1.2 to the dashed-line `year` element. In range-based order encodings, such as *interval* or *pre/post encoding*, a unique {*start*,*end*} interval identifies each node in the document tree. This interval can be generated in multiple ways. The most common method is to create a unique identifier, *start*, for each node in a preorder traversal of the document tree, and a unique identifier, *end*, in a postorder traversal. Additionally, in order to distinguish children from descendants, a level number needs to be recorded with each node.

An important consideration for any order-encoding scheme is to be able to handle updates in the XML documents, and many improvements have been made to the above basic encodings to reduce the overhead associated with updates.

Hybrid XML Storage

Some XML documents have both very structured and very unstructured parts. This has lead to the idea of *hybrid* XML storage, where different data models, and even systems using different storage technologies, store different document parts. For example, in Fig. 3b, `review` elements and their subtrees can be stored very differently from `show` elements, for example by serializing each review according to the dashed lines in the figure or storing them in a native XML storage system.

Prototype systems such as MARS and XAM have been proposed that support a hybrid storage model at the system level, i.e., provide physical data independence. In these systems, different access methods corresponding to the different storage mappings are formally described using views and constraints, and query processing involves the use of query rewriting using views. Moreover, an appropriate tool or language is necessary to specify hybrid storage designs effectively and declaratively.

An additional consideration in favor of hybrid XML storage is that storing some information redundantly using different techniques can improve the performance of querying and data retrieval significantly by combining their benefits. For example, schema-directed relational storage mappings often give better performance for identifying the elements that satisfy an XPath query, while native storage allows the direct retrieval of large elements. In environments where updates are infrequent or update cost less important than query performance, such as various web-based query systems, such redundant storage approaches can be beneficial.

Key Applications

XML Storage techniques are used to efficiently store XML documents, XML messages, accumulated XML streams and any other form of XML-encoded content. XML Storage is a key component of an XML database management system. It can also provide significant benefits for the storage of semi-structured information with mostly tree structure, including scientific data.

Cross-references

- ▶ Dataguide
- ▶ Deweys
- ▶ Intervals
- ▶ Storage Management
- ▶ Top-k XML Query Processing
- ▶ XML Document
- ▶ XML Indexing
- ▶ XML Schema
- ▶ XPath/XQuery

Recommended Reading

1. Arion A., Benzaken V., Manolescu I., and Papakonstantinou Y. Structured materialized views for XML queries. In Proc. 33rd Int. Conf. on Very Large Data Bases, 2007, pp. 87–98.
2. Barbosa D., Freire J., and Mendelzon A.O. Designing information-preserving mapping schemes for XML. In Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 109–120.
3. Beyer K., Cochrane R.J., Josifovski V., Kleewein J., Lapis G., Lohman G., Lyle B., Özcan F., Pirahesh H., Seemann N., Truong T., der Linden B.V., Vickery B., and Zhang C. System RX: one part relational, one part XML. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2005, pp. 347–358.
4. Chaudhuri S., Chen Z., Shim K., and Wu Y. Storing XML (with XSD) in SQL databases: interplay of logical and physical designs. IEEE Trans. Knowl. Data Eng., 17(12):1595–1609, 2005.

5. Deutsch A., Fernandez M., and Suciu D. Storing semi-structured data with STORED. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1999, pp. 431–442.
6. Fiebig T., Helmer S., Kanne C.C., Moerkotte G., Neumann J., Schiele R., and Westmann T. Anatomy of a native XML base management system. VLDB J., 11(4):292–314, 2003.
7. Georgiadis H. and Vassalos V. XPath on steroids: exploiting relational engines for XPath performance. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2007, pp. 317–328.
8. Härder T., Haustein M., Mathis C., and Wagner M. Node labeling schemes for dynamic XML documents reconsidered. Data Knowl. Eng., 60(1):126–149, 2007.
9. McHugh J., Abiteboul S., Goldman R., Quass D., and Widom J. Lore: a database management system for semistructured data. ACM SIGMOD Rec., 26:54–66, 1997.
10. Shanmugasundaram J., Tufte K., He G., Zhang C., DeWitt D., and Naughton J. Relational databases for querying XML documents: limitations and opportunities. In Proc. 25th Int. Conf. on Very Large Data Bases, 1999, pp. 302–314.
11. Vélez F., Bernard G., and Darnis V. The O2 object manager: an overview. In Building an Object-Oriented Database System, The Story of O2. F. Bancilhon, C. Delobel, and P.C. Kanellakis (eds.), Morgan Kaufmann, San Francisco, CA, USA, 1992, pp. 343–368.

XML Stream Processing

Christoph Koch
Cornell University, Ithaca, NY, USA

Definition

XML Stream Processing refers to a family of data stream processing problems that deal with XML data. These include XML stream filtering, transformation, and query answering problems.

A main distinguishing criterion for XML stream processing techniques is whether to filter or transform streams. In the former scenario, XML streams are usually thought of as consisting of a sequence of rather small XML documents (e.g., news items), and the (Boolean) queries decide for each item to either select or drop it. In the latter scenario, the input stream is transformed into a possibly quite different output stream, often using an expressive transformation language.

Historical Background

With the spread of the XML data exchange format in the late 1990s, the research community has become interested in processing streams of XML data. The selective dissemination of information that is not strictly tuple-based, such as electronically disseminated news, was one of the first, and remains one of the foremost applications of XML streams. Subscriptions to items of interest in XML Publish-Subscribe systems are usually expressed in weak XML query languages such as tree patterns and fragments of the XPath query language. This problem has been considered with the additional difficulty that algorithms have to scale to large numbers of queries to be matched efficiently in parallel.

In addition to XML publish-subscribe in the narrow sense, a substantial amount of research has addressed the problems of processing more expressive query and stream transformation languages on XML streams. This includes automata as well as well as data-transformation query language-based approaches (e.g., XQuery).

Foundations

Controlling Memory Consumption. Efficient stream processing is only feasible for data processing problems that can be solved by strictly linear-time one-pass processing of the data using very little main memory. For the XML stream filtering problem, and restricted query languages such as XML tree patterns, linear-time evaluation techniques are not hard to develop (see e.g., [1] for an extensive survey). However, developing one-pass filtering techniques that require little main memory is nontrivial.

Techniques from communication complexity have been used to study memory lower bounds of streaming XPath evaluation algorithms. It has been observed in [5] that there can be no streaming algorithm with memory consumption sublinear in the depth of the data tree, even for simple tree pattern queries and very small XPath fragments. This is a tight bound. Boolean queries in query languages of considerable expressiveness can be processed on XML streams using memory linear in the depth of the tree and independent of any other aspects of its size. By classical reductions between logics and automata on trees, this includes queries in first-order and even monadic second-order logic and as a consequence tree pattern queries and a large class of XPath queries [1]. Since the nesting depth of XML documents tends to be shallow, this is a positive result.

Selecting Nodes or Subtrees. The above observation – that memory consumption does not depend on the size of the XML document but only on the depth of the parse tree – only holds for the problem of testing, for each document, whether it matches the tree pattern or

not. If output is to be produced by selecting documents or subtrees, then there are simple queries which require most of the document to be buffered. For an example, consider the following two XML documents.

$$\langle A\rangle\langle B/\rangle \ldots \langle B/\rangle\langle C/\rangle\langle/A\rangle \quad \langle A\rangle\langle B/\rangle \ldots \langle B/\rangle\langle D/\rangle\langle/A\rangle$$

and the XPath query /*[C]. Any implementation of this query that is to output selected documents must select the entire left document but not the right. Hence such an implementation will have to buffer the prefix of either document up to the *C* resp. *D* node. This may amount to buffering almost all the document.

The problem of efficiently selecting nodes using XPath on XML streams with close to minimum space requirements was studied in several works, and results are usually space bounds depending linearly on the depth of the data tree, a function of certain properties of the query, and the number of candidate output nodes from the data tree, which in some cases can be nearly all the nodes in the parse tree of the XML document.

Automata-Based Stream Processing. A large part of the research into efficient XML stream processing is based on compiling queries or subscriptions into automata that then run on the data stream. This is not surprising since automata provide a natural one-pass processing model. For most forms of automata one can analyze the runtime memory usage easily. Note, however, that finite word automata are not sufficiently expressive to keep track of the position in the nested XML stream.

Translating XPath queries into pushdown automata has been studied in several works. Pushdown automata yield document depth-bounded space usage. The pushdown automaton nature is somewhat concealed in some of this work; the processing model can be thought of as a finite word automaton for a query path expression which runs on the path from the root node of the XML tree to the current data tree node. There is also a pushdown automaton, independent of the path expression, that acts as a controller for the word automaton, managing the current path using a stack and conceptually rerunning the word automaton every time a new node in the stream is encountered.

The blow-up required to compute *deterministic* finite word automata for query path matching is exponential in size of the query. The sources of this exponentiality were explored in [4]. An alternative is to maintain a nondeterministic finite word automaton for path matching as done in YFilter [3]. This technique is described in detail in.

Transducer Networks. In the following, a technique for constructing automata that have the power to effect stream transformation while being deterministic, consuming little main memory, and being of size polynomial in the size of the query is described. The exponential size of deterministic automata is avoided by not compiling automata for managing and recognizing the subexpressions of an XPath query into a single "flat" automaton. These automata are instead kept apart, as a *transducer network* [6].

A transducer network consists of a set of synchronously running transducers (here, deterministic pushdown transducers) where each transducer runs, possibly in parallel with some other transducers, either on the input XML stream, or on the output of another transducer (in which case the input is the original stream where some nodes may have been annotated using labels). Two transducers may also be "joined," producing output whose annotations are pairs consisting of the annotations produced by the two input transducers.

Next, this is formalized, and some of the transducers that form part of a transducer network are exhibited.

XPath queries are first rewritten into nested filters with paths of length one; for instance, query child::*A*/descendant::*B* is first rewritten into child[lab() = $A \wedge$ descendant[lab() = *B*]]. To emphasize the goal of checking whether the query can be successfully matched, rather than computing nodes matched by a path, axis filters are written as $\exists$child[ϕ] and $\exists$descendant[ϕ]. The rewritten queries will now be translated into transducer networks inductively.

The axes used have to be forward axes (child, next sibling, descendant, and following). A large class of XPath queries with other axes (e.g., ancestor) can be rewritten to use just forward axes [2].

A deterministic pushdown transducer T is a tuple $(\Sigma, \Gamma, \Omega, Q, q_0, F, \delta)$ with input alphabet Σ, stack alphabet Γ, output alphabet Ω, set of states Q, start state q_0, set of final states F, and transition function $\delta : Q \times \Sigma \times (\varepsilon \cup \Gamma) \to Q \times \Gamma^* \times \Omega$. For no $q \in Q$, $s \in \Sigma$, $\gamma \in \Gamma$, both $\delta(q, s, \varepsilon)$ and $\delta(q, s, \gamma)$ are defined. Here ε denotes the empty word. All transducers will have $Q = F$; that is, all states are final states, so all valid

runs will be accepting. If the transducer T is in state q and has uv on the stack, and if $\delta(q, s, v) = (q', w, s')$, then T makes a transition to state q' and stack uw (u,v, $w \in \Gamma^*$) on input s, and produces output o, denoted $(q,uv) \xrightarrow{s/o} (q',uw)$. A run on input $s_1...s_n$ is a sequence of transitions $(q_0,\varepsilon) \xrightarrow{s_1/o_1} \cdots \xrightarrow{s_n/o_n} (q,u)$ that produces output $o_1...o_n$.

A transducer $T[\exists\text{descendant}[\phi]]$ running on the output stream of transducer $T[\phi]$ is a deterministic pushdown transducer with $\Sigma = \Omega = \{\langle\ \rangle, t, f\}$, $\Gamma = \{t,f\}$, $Q = F = \{q_f, q_t\}$, $q_0 = q_f$, and transition function

$$\delta : \begin{cases} q_x, \langle\rangle, \epsilon) \mapsto (q_f, x, \langle\rangle) \\ (q_x, y \in \{t,f\}, z) \mapsto (q_{x \vee y \vee z}, \epsilon, x) \end{cases}$$

On seeing an opening tag of a node, this transducer memorizes on the stack whether ϕ was matched in the subtrees of the previously seen siblings of that node. On returning (i.e., seeing a closing tag), the transducer labels the node (by its proxy the closing tag) with t or f (true or false) depending on whether ϕ was matched in the node's subtree, which is encoded in the state. Example 1. On input $\langle\ \rangle\ \langle\ \rangle\ \langle\ \rangle\ \langle\ \rangle\ f\ t\ t\langle\ \rangle\ \langle\ \rangle\ \langle\ \rangle\ t\ f\ f\ t$, $T[\exists\text{descendant}[\cdot]]$ has the run

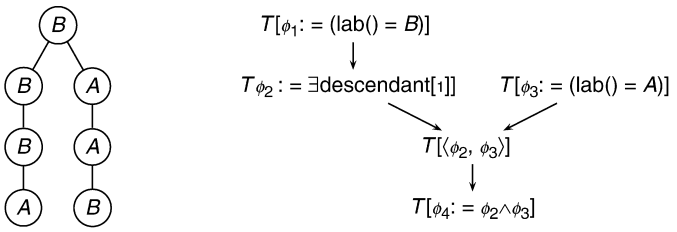
$$(q_f, \epsilon) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, f) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, ff) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, fff) \xrightarrow{\langle\rangle/\langle\rangle}$$
$$(q_f, ffff) \xrightarrow{f/f} (q_f, fff)) \xrightarrow{t/f} (q_t, ff) \xrightarrow{t/t} (q_t, f) \xrightarrow{\langle\rangle/\langle\rangle}$$
$$(q_f, ft) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, ftf) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, ftff) \xrightarrow{t/f} (q_t, ftf) \xrightarrow{f/t}$$
$$(q_t, ft) \xrightarrow{f/t} (q_t, f) \xrightarrow{t/t} (q_t, \epsilon)$$

and produces output $\langle\ \rangle\ \langle\ \rangle\ \langle\ \rangle\ \langle\ \rangle\ fft\ \langle\ \rangle\ \langle\ \rangle\ \langle\ \rangle\ fttt$ (see Fig. 1).

A transducer $T[\exists\text{child}[\phi]]$ can be defined similarly.

The transducers for testing labels and computing conjunctions of filters do not need a stack. The transducer $T[\text{lab}() = A]$ has the opening and closing tags of the XML document as input alphabet $\Sigma, \Omega = \{\langle\ \rangle, t, f\}$, $Q = F = \{q_0\}$, and $\delta = \{(q_0,\langle\cdot\rangle,\varepsilon) \mapsto (q_0,\varepsilon,\langle\ \rangle), (q_0,\langle/A\rangle, \varepsilon) \mapsto (q_0,\varepsilon,t), (q_0,\langle/B\rangle,\varepsilon) \mapsto (q_0,\varepsilon,f)\}$ (where B stands for all node labels other than A). The transducer $T[\phi \wedge \psi]$ has $\Sigma = \{\langle\ \rangle\} \cup \{t,f\}^2$, $\Omega = \{\langle\ \rangle, t, f\}$, $Q = F = \{q_0\}$ and $\delta = \{(q_0,\langle\ \rangle, \varepsilon) \mapsto (q_0,\varepsilon,\langle\ \rangle), (q_0,(x,y),\varepsilon) \mapsto (q_0,\varepsilon, x \wedge y)\}$.

The overall execution of a transducer network is exemplified in Fig. 1, where the filter that matches the XPath expression self::*A*/descendant::*B*, rewritten into $(\exists\text{descendant}[\text{lab}() = B]) \wedge \text{lab}() = A$ is evaluated using a transducer network. The transducers for the different subexpressions run synchronously; each symbol (opening or closing tag) from the input stream is first transformed by $T[\phi_1]$ and $T[\phi_3]$; the output of $T[\phi_1]$ is piped into $T[\phi_2]$ and the output of both $T[\phi_2]$ and $T[\phi_3]$, as a pair of symbols, is piped into $T[\phi_4]$. Only then is the next symbol of the input stream processed, which is handled in the same way, and so on. In the example of Fig. 1, the final transducer labels exactly those nodes t on which the filter is true. Checking whether the filter can be matched on the root node, which is not the case in this example, can be done using an additional pushdown automaton which is not exhibited here but is simple to define.



	Time ⟶													
Input stream	⟨*B*⟩	⟨*B*⟩	⟨*B*⟩	⟨*A*⟩	⟨/*A*⟩	⟨/*B*⟩	⟨/*B*⟩	⟨*A*⟩	⟨*A*⟩	⟨*A*⟩	⟨/*B*⟩	⟨/*A*⟩	⟨/*A*⟩	⟨/*B*⟩
transducer	Synchronous output													
$T[\phi_1$: = (lab() =*B*)]	⟨⟩	⟨⟩	⟨⟩	⟨⟩	*f*	*t*	*t*	⟨⟩	⟨⟩	⟨⟩	*t*	*f*	*f*	*t*
$T[\phi_2$: = ∃descendant [ϕ_1]]	⟨⟩	⟨⟩	⟨⟩	⟨⟩	*f*	*f*	*t*	⟨⟩	⟨⟩	⟨⟩	*f*	*t*	*t*	*t*
$T[\phi_3$: = (lab() = *A*)]	⟨⟩	⟨⟩	⟨⟩	⟨⟩	*t*	*f*	*f*	⟨⟩	⟨⟩	⟨⟩	*f*	*t*	*t*	*f*
$T[\langle\phi_2, \phi_3\rangle]$	⟨⟩	⟨⟩	⟨⟩	⟨⟩	(*f*,*t*)	(*f*,*f*)	(*t*,*f*)	⟨⟩	⟨⟩	⟨⟩	(*f*,*f*)	(*t*,*t*)	(*t*,*t*)	(*t*,*f*)
$T[\phi_4 : = \phi_2 \wedge \phi_3]$	⟨⟩	⟨⟩	⟨⟩	⟨⟩	*t*	*f*	*f*	⟨⟩	⟨⟩	⟨⟩	*f*	*t*	*t*	*f*

XML Stream Processing. Figure 1. Document tree (top left), transducer network (top right), and run of the transducer network (bottom).

Key Applications

See XML Publish-Subscribe.

Cross-references

- ▶ Data Stream
- ▶ XML Publish/Subscribe

Recommended Reading

1. Benedikt M. and Koch C. Xpath Unleashed. ACM Comput. Surv., 41(3), 2009.
2. Bry F., Olteanu D., Meuss H., and Furche T. Symmetry in XPath. Tech. Rep. PMS-FB-2001-16, LMU München, 2001, short version.
3. Diao et al. Y. YFilter: efficient and scalable filtering of XML documents. In Proc. 18th Int. Conf. on Data Engineering, 2002, pp. 341–342.
4. Green T.J., Miklau G., Onizuka M., and Suciu D. Processing XML streams with deterministic automata. In Proc. 9th Int. Conf. on Database Theory, 2003, pp. 173–189.
5. Grohe M., Koch C., and Schweikardt N. Tight lower bounds for query processing on streaming and external memory data. Theor. Comput. Sci., 380(1–2):199–217, 2007.
6. Olteanu D. SPEX: streamed and progressive evaluation of XPath. IEEE Trans. Knowl. Data Eng., 19(7):934–949, 2007.

XML Tree Pattern, XML Twig Query

Laks V. S. Lakshmanan
University of British Columbia, Vancouver, BC, Canada

Synonyms

Tree pattern queries; TPQ; TP; Twigs

Definition

A *tree pattern query* (also known as *twig query*) is a pair $Q = (T, F)$, where T is node-labeled and edge-labeled tree with a distinguished node $x \in T$ and F is a boolean combination of constraints on nodes. Node labels are variables such as *$x, $y*. Edge labels are one of "pc," "ad," indicating parent-child or ancestor-descendant. Node constraints are of the form *$x.tag* = TagName or *$x.data* relOp *val*, where *$x.data* denotes the data content of node *$x*, and *relOp* is one of $=, <, >, \leq, \geq, \neq$.

Informally, a tree pattern query specifies a pattern tree, with a set of constraints. Some of the constraints specify what the node labels (tags) should be. Some of them specify how pairs of nodes are related to one another – as a parent-child or as an ancestor-descendant. Finally, constraints on the data content of nodes enforce what data values are expected to be present at the nodes. Taken together, a tree pattern is similar in concept to a selection condition in relational algebra. There, the condition specifies what it takes a tuple to be part of a selection query result. A tree pattern similarly specifies what it takes an XML fragment (i.e., a subtree of the input data tree) to be part of a selection query result. This will be formalized below.

Historical Background

One of the earliest papers to use the term *twig query* was Chen et al. [1]. In this paper, the authors were interested in estimating the number of matches of a twig query against an XML data tree, using a summary data structure. However, the notion of a twig in this paper was confined to tree patterns without ancestor-descendant relationships. Furthermore, data values were required to be strings over an alphabet (i.e., PCDATA) and a match was defined based on identity of node labels in the twig query and the labels of corresponding nodes in the data tree.

Amer-Yahia et al. [2] introduced the notion of tree pattern query where the authors allowed both parent-child and ancestor-descendant relationships between pattern nodes. They motivated such queries in the context of querying LDAP-style network directories and as a core operation in query languages such as XML-QL and Quilt (the prevalent predecessors of XQuery). They focused on minimization of tree pattern queries. Jagadish et al. [3] was the first paper to define tree pattern queries in the current general form. They did so in the context of TAX – a tree algebra for XML data manipulation. Actually, tree patterns as defined in [3] allow for a richer class of node predicates than defined above. An XML data management system called TIMBER that was developed using the foundations of TAX algebra is described in Jagadish et al. [4]. Subsequently, an extensive body of work has flourished on various aspects of tree pattern (or twig) queries, ranging from their minimization [2,5,6], their efficient evaluation via the so-called structural and holistic joins [7,8] and their extensions leveraging indices [9,10]. Twig queries have served as a basis for defining the semantics of group-by for XML data [11]. Subsequently, efficient algorithms for computing group-by queries over XML data were reported in [12] while [13] addresses efficient computation of XML cube. All of

these works use tree patterns in an essential way. Given the importance of approximate match when searching an XML document, papers dealing with approximate match have also taken tree patterns as a basis. These include tree pattern relaxation [14] and FleXPath [15]. Last but not the least, given the importance of twig queries, much work has been done on estimating selectivity of twig queries [16–18].

Foundations

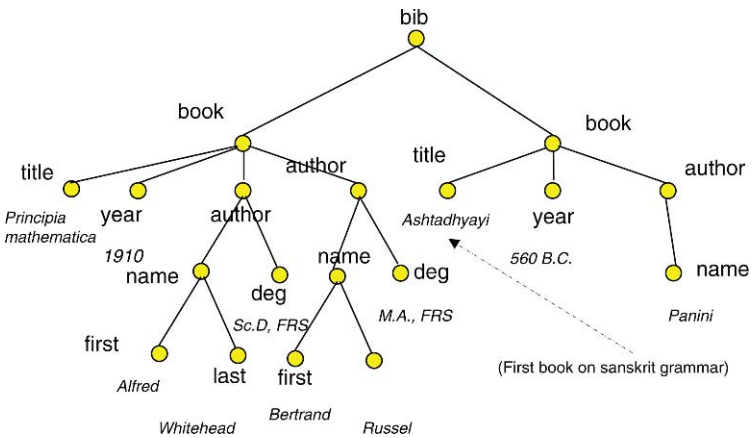
Let $D = (N, A)$ be an XML data tree, i.e., an ordered node-labeled tree whose nodes are labeled by tags, which are element names drawn from an alphabet $\prod$ and whose leaf nodes are additionally labeled by strings from Σ^*. The latter correspond to PCDATA and represent the data content of leaf nodes. A twig query specifies a pattern tree. The semantics of such a query is based on the notion of a match. Let $Q = (T, F)$ be a tree pattern, where $T = (V, E)$ is a node-labeled and edge-labeled tree and let D be an XML data tree. Then a match is defined based on the notion of an embedding. Given Q and D as above, an *embedding* is a function h: V → N, mapping query or pattern nodes to data nodes such that the following conditions are satisfied:

- Whenever $(x, y) \in E$ is an edge labeled "pc" (resp., "ad"), $h(y)$ is a child (resp., descendant) of $h(x)$ in D.
- The boolean combination of conditions of the form and *$x.tag* = TagName *and $x.data relOp val* in F is satisfied:
  - An atom *$x.tag* = TagName is satisfied provided *$x* is the label of node v in Q and the tag of the data node $h(v)$ is TagName.
  - An atom *$x.data relOp val is satisfied provided $x* is the label of node v of Q and the node label of the data node $h(v)$ stands in relationship *relOp* to the constant *val*.
  - Satisfaction w.r.t. boolean combinations of atoms is defined in the standard way.

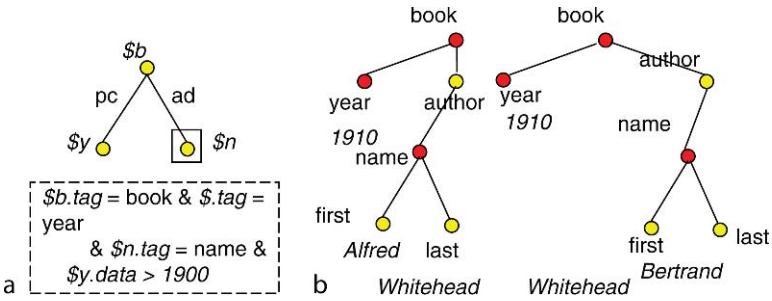
Figure 1 shows a sample data tree. Figure 2 shows a tree pattern query and illustrates matches. In this figure, the distinguished node (node labeled *$n*) is identified by a surrounding box. Informally, this pattern says find the immediate or transitive name sub-elements of book elements such that the book was published after 1900, as indicated by an immediate year sub-element of the book element. Note the edge labels "pc" and "ad" in Fig. 2(a) that specify the intended relationship between elements. In Fig. 2(b), the matching data nodes are indicated in red. The result of a selection query with this tree pattern against the input of Fig. 1(a) consists of the two name sub-elements corresponding to matches of the distinguished node of the tree pattern.

Key Applications

As mentioned earlier, a substantial amount of work on XML query evaluation and optimization has flourished using twig/tree patterns as a basis. One of the reasons for this is that tree pattern queries can be seen as an abstraction of a "core" subset of XPath [19]. In addition to the query evaluation issues mentioned above, tree pattern queries have also attracted attention from the point of view of query static analysis, study of structural properties, and query answering using views. In static analysis, Hidders [20], Lakshmanan



XML Tree Pattern, XML Twig Query. Figure 1. Example XML data tree.



XML Tree Pattern, XML Twig Query. Figure 2. (a) Example tree pattern query and (b) Matches.

et al. [21], and Benedikt et al. [22] study satisfiability of fragments of XPath queries, which can be modeled as tree patterns with possible extensions. Benedikt et al. [23] study structural properties of several XPath fragments corresponding to different extensions of the basic class of tree patterns as defined in this chapter. Xu and Ozsoyoglu [24], Deutsch and Tannen [25], and Lakshmanan et al. [26] study different formulations of the query answering using views problem for XML.

It is important to investigate richer extensions of tree patterns, capturing a greater subset of the features found in XPath. By far, the most important such feature that is lacking in tree patterns is element order. Various papers have examined XPath fragments together with sibling axis. An alternative approach is to consider tree patterns with two kinds of internal nodes – ordered and unordered. An unordered node is just like a node in a standard tree pattern. An ordered internal node is one for which the order of its children in the pattern should obey a specified order. e.g., suppose x is an ordered internal node and it has children y, z, w in that order, in the pattern. Then this imposes a constraint that a match of node z should follow the corresponding match of y. Similarly, a match of w should follow the corresponding match of z. Another example extension that is worthwhile investigating is a notion of relaxation of a tree pattern with order and keyword search that is appropriate for searching XML documents.

Cross-references

- ▶ Top-k XML Query Processing
- ▶ XML Retrieval
- ▶ XML Schema
- ▶ XML Selectivity Estimation
- ▶ XML Views
- ▶ XPath/XQuery
- ▶ XSL/XSLT

Recommended Reading

1. Al-Khalifa S., Jagadish H.V., Koudas N., Patel J.M., Srivastava D., and Wu Y. Structural joins: a primitive for efficient XML query pattern matching. In Proc. 18th Int. Conf. on Data Engineering, 2002, pp. 141–152.
2. Amer-Yahia S., Cho S.R., Lakshmanan L.V.S., and Srivastava D. Minimization of tree pattern queries. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001, pp. 497–508.
3. Amer-Yahia S., Cho S.R., and Srivastava D. Tree pattern relaxation. In Advances in Database Technology, Proc. 8th Int. Conf. on Extending Database Technology, 2002, pp. 496–513.
4. Amer-Yahia S., Lakshmanan L.V.S., and Pandit S. FleXPath: flexible structure and full-text querying for XML. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 83–94.
5. Benedikt M., Fan W., and Geerts F. XPath satisfiability in the presence of DTDs. In Proc. 24th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2005, pp. 25–36.
6. Benedikt M., Fan W., and Kuper G.M. Structural properties of XPath fragments. In Proc. 9th Int. Conf. on Database Theory, 2003, pp. 79–95.
7. Bruno N., Koudas N., and Srivastava D. Holistic twig joins: optimal XML pattern matching. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 310–321.
8. Chen Z., Jagadish H.V., Korn F., Koudas N., Muthukrishnan S., Ng R., and Srivastava D. Counting twig matches in a tree. In Proc. 17th Int. Conf. on Data Engineering, 2001, pp. 595–604.
9. Chien S.Y., Vagena Z., Zhang D., Tsotras V.J., and Zaniolo C. Efficient structural joins on indexed XML documents. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 263–274.
10. Deutsch A. and Tannen V. Reformulation of XML queries and constraints. In Proc. 9th Int. Conf. on Database Theory, 2003, pp. 225–241.
11. Flesca S., Furfaro F., and Masciari E. On the minimization of Xpath queries. In Proc. 29th Int. Conf. on Very Large Data Bases, 2003, pp. 153–164.
12. Freire J., Haritsa J., Ramanath M., Roy P., and Simeon J. StatiX: making XML count. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 181–191.
13. Gokhale C., Gupta N., Kumar P., Lakshmanan L.V.S., Ng R., and Prakash B.A. Complex group-by queries for XML. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 646–655.

14. Hidders J. Satisfiability of XPath expressions. In Proc. 9th Int. Workshop on Database Programming Languages, 2003, pp. 21–36.
15. 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. 149–169.
16. Jagadish H.V., Al-Khalifa S., Chapman A., Lakshmanan L.V.S., Nierman A., Paparizos S., Patel J.M., Srivastava D., Wiwatwattana N., Wu Y., and Yu C.TIMBER: a native XML database. VLDB J., 11(4):274–291, 2002.
17. Jiang H., Lu H., Wang W., and Ooi B.C. XR-tree: indexing XML data for efficient structural joins. In Proc. 19th Int. Conf. on Data Engineering, 2003, pp. 253–263.
18. Lakshmanan L.V.S., Ramesh G., Wang H., and (Jessica) Zhao Z. On testing satisfiability of tree pattern queries. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004, pp. 120–131.
19. Lakshmanan L.V.S., Wang H., and (Jessica) Zhao Z. Answering tree pattern queries using views. In Proc. 32nd Int. Conf. on Very Large Data Bases, 2006, pp. 571–582.
20. Miklau G. and Suciu D. Containment and equivalence for a fragment of XPath. In Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2002, pp. 65–76.
21. Paparizos S., Al-Khalifa S., Jagadish H.V., Lakshmanan L.V.S., Nierman A., Srivastava D., and Wu Y. Grouping in XML. In EDBT 2002 Workshop on XML-Based Data Management LNCS, vol. 2490, Springer, Berlin, 2002, pp. 128–147.
22. Polyzotis N. and Garofalakis M. XSketch synopses for XML data graphs. ACM Trans. Database Syst., September 2006.
23. Polyzotis N., Garofalakis M., and Ioannidis Y. Selectivity estimation for XML twigs. In Proc. 20th Int. Conf. on Data Engineering, 2004, pp. 264–275.
24. Wiwatwattana N., Jagadish H.V., Lakshmanan L.V.S., and Srivastava D. X^3: a cube operator for XML OLAP. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 916–925.
25. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath.
26. Xu W. and Meral Özsoyoglu Z. Rewriting XPath queries using materialized views. In Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 121–132.

XML Tuple Algebra

Ioana Manolescu[1], Yannis Papakonstantinou[2], Vasilis Vassalos[3]
[1]INRIA Saclay–Île-de-France, Orsay, France
[2]University of California-San Diego, La Jolla, CA, USA
[3]Athens University of Economics and Business, Athens, Greece

Synonyms

Relational algebra for XML; XML algebra

Definition

An XML tuple-based algebra operates on a domain that consists of sets of tuples whose attribute values are *items*, i.e., atomic values or XML elements (and hence, possibly, XML trees). Operators receive one or more sets of tuples and produce a set, list or bag of tuples of items. It is common that the algebra has special operators for converting XML inputs into instances of the domain and vice versa. XML tuple-based algebras, as is also the case with relational algebras, have been extensively used in query processing and optimization [1–10].

Historical Background

The use of tuple-based algebras for the efficient set-at-a-time processing of XQuery queries follows a typical pattern in database query processing. Relational algebras are the most typical vehicle for query optimization. Tuple-oriented algebras for object-oriented queries had also been formulated and have a close resemblance to the described XML algebras.

Note that XQuery itself as well as predecessors of it (such as Quilt) are also algebras, which include a list comprehension operator ("FOR"). Such algebras and their properties and optimization have been studied by the functional programming community. They have not been the typical optimization vehicle for database systems.

Foundations

The emergence of XML and XQuery motivated many academic and industrial XML query processing works. Many of the works originating from the database community based XQuery processing on tuple-based algebras since in the past, tuple-based algebras delivered great benefits to relational query processing and also provided a solid base for the processing of nested OQL data and OQL queries. Tuple-based algebras for XQuery carry over to XQuery key query processing benefits such as performing joins using set-at-a-time operations.

A generic example algebra, characteristic of many algebras that have been proposed as intermediate representations for XQuery processing, is described next. It is based on a *Unified Data Model* that extends the XPath/XQuery data model with sets, bags, and lists of tuples; notice that the extensions are only visible to algebra operators.

Given an XML algebra, important query processing challenges include efficient implementation of the operators as well as cost models for them, algebraic properties of operator sequences and algebraic rewriting techniques, and cost-based optimization of algebra expressions.

Unified Data Model

The *Unified Data Model* (UDM) is an extension of the XPath/XQuery data model with the following types:

- *Tuples*, with structure $[\$a_1 = val_1,...,\$a_k = val_k]$, where each $\$a_i = item_i$ is a *variable-value* pair. Variable names such as $\$a_1$,$\$a_2$,etc. follow the syntactic restrictions associated with XQuery variable names; Variable names are unique within a tuple. A *value* may be (i) thespecial constant $\perp$(*null*) (The XQuery Data Model also has a concept of *nil*. For clarity of exposition, the two concepts should be considered as distinct, in order to avoid discrepancies that may stem from the overloading.), (ii) an *item*, i.e., a node (generally standing for the root of an XML tree)or atomic value, or (iii) a (nested) set, list, or bag of tuples. Given a tuple $[\$a_1 = val_1,...,\$a_k = val_k]$ the list of names $[\$a_1,...,\$a_k]$ is called the *schema* of the tuple. We restrict our model to *homogeneous* collections (sets, bags or lists) of tuples. That is, the values taken by a givenvariable $\$a$ in all tuples are of the same kind: either they are all items (some of which may be null), or they are all collections of tuples. Moreover, in the lattercase, all the collections have the same schema.

  Note that the nested sets/bags/lists are typically exploited for buildingefficient evaluation plans.

  If the value of variable $t.\$a$ is a set, list or bag of tuples, let $[\$b_1, \$b_2,...,\$b_m]$ be the schema of a tuple in $t.\$a$. In this case, for clarity, we may denote a $\$b_i$ variable, $1 \leq i \leq m$, by $\$a.\$b_i$ (concatenating the variable names, from the outermost to the innermost, and separating them by dots).
- *Lists, bags and sets of tuples*, denoted as $\langle t_1,...,t_n\rangle$, $\{\{t_1,...,t_n\}\}$, and $\{t_1,...,t_n\}$ and referred to collectively as *collections*. In all three cases the tuples $t_1,...,\ t_n$ must have the same schema, i.e., collections are homogeneous. Sets have no duplicate tuples, i.e., no two tuples in a set are *id-equal* as defined below.

Two tuples are id-equal, denoted as $t_1 =_{id} t_2$, if they have the same schema and the values of the corresponding variables either (*i*) are both null, or (*ii*) are both equal atomic values (i.e., they compare equal via $=_v$), or are both nodes with the same id (i.e., they compare equal via $=_{id}$), or (*iii*) are both sets of tuples and each tuple of a set is id-equal to a tuple of the other set. For the case (*iii*), similar definitions apply if the variable values are bags or lists of tuples, by taking into account the multiplicity of each tuple in bags and the order of the tuples for lists.

Notation Given a tuple $t = [\ldots\$x = v\ldots]$ it is said that $\$x$ maps to v in the context of t. The value that the variable $\$x$ maps to in the tuple t is $t.\$x$. The notation $t' = t + (\$var = v)$ indicates that the tuple t' contains all the variable-value pairs of t and, in addition, the variable-value pair $\$var = v$. The tuple $t' = t + t'$ contains all the variable-value pairs of both tuples t and t'. Finally, (*id*) denotes the node with identifier *id*.

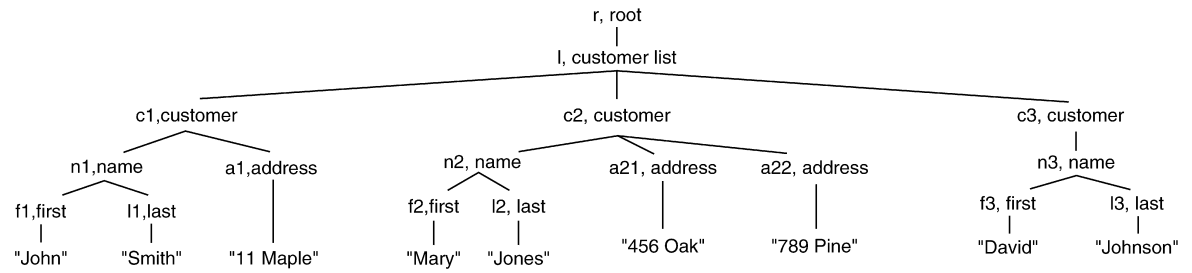
Sample document and query A sample XML document is shown in tree form in Fig. 1. For readability, the figure also displays the tag or string content of each node. The following query will be used to illustrate the XML tuple algebra operators:

```
for $C in $I//customer
return <customer>
  {for $N in $C/namē
       $F in $N/first
       $L in $N/last
  return {$F, $L, $C/address } }
<customer>                              (Q1)
```

Unified Algebra

Tuple-based XQuery algebras typically consist of operators that: (i) perform navigation into the data and deliver collections of bindings (tuples) for the variables; (ii) construct XPath/XQuery Data Model values from the tuples; (iii) create nested collections; (iv) combine collections applying operations known from the relational algebra, such as joins It is common to have redundant operators for the purpose of delivering performance gains in particular environments; structural joins are a typical example of such redundant operators.

An XQuery q, with a set of free variables $\overline{V}$, is translated into a corresponding algebraic expression f_q, which is a function whose input is a collection with schema $\overline{V}$. The algebraic expressions outputs a collection in the *Unified Data Model*. The trivial operators used to convert the collection into an XML-formatted document are not discussed.



XML Tuple Algebra. Figure 1. Tree representation of sample XML document.

Selection The selection operator σ is defined in the usual way based on a logical formula (predicate) that can be evaluated over all its input tuples. The selection operator outputs those tuples for which the predicate evaluated to true.

Projection The projection operator π is defined by specifying a list of column names to be retained in the output. The operator π does not eliminate duplicates; duplicate-eliminating projections are denoted by π^0.

Navigation The navigation operator follows the established tree pattern paradigm. XQuery tuple-based algebras involve tree patterns where the root of the navigation may be a variable binding, in order to capture nested XQueries where a navigation in the inner query may start from a variable of the outer query. Furthermore, XQuery tuple-based algebras have extended tree patterns with an option for "optional" subtrees: if no match is found for optional subtrees, then those subtrees are ignored and a $\perp$ (null) value is returned for variables that appear in them. This feature is similar to outerjoins, and has been used in some algebras to consolidate the navigation of multiple nested queries (and hence of multiple tree patterns) into a single one. In what follows, the tree patterns are limited to child and descendant navigation. The literature describes extensions to all XPath axes.

An *unordered tree pattern* is a labeled tree, whose nodes are labeled with (i) an element/attribute or the wildcard $*$ and (ii) optionally, a variable $\$var$. The edges are either *child* edges, denoted by a single line, or *descendant* edges, denoted by a double line. Furthermore, edges can be *required*, shown as continuous lines, or *optional*, depicted with dashed lines. The variables appearing in the tree pattern form the *schema* of the tree pattern.

The semantics of the navigation operator is based on the *mapping* of a tree pattern to a value. Given a variable-value pair $\$R = r_i$ and a tree pattern T with schema V, a mapping of T to r_i is a mapping of the pattern nodes to nodes in the XML tree represented by r_i such that the structural relationships among pattern nodes are satisfied. A *mapping tuple* (also called *binding*) has schema V and pairs each variable in V to the node to which it is mapped corresponding to such a mapping. The function *map* (T, r_i) returns the set of bindings corresponding to all mappings.

A mapping may be partial: nodes connected to the pattern by optional edges may not be mapped, in which case this node is mapped to a $\perp$.

To formally define embeddings for tree patterns T that have optional edges, the auxiliary *pad* and *sp* functions are introduced.

Given a set of variables $\overline{V}$, the function $pad_{\overline{V}}(t)$ extends the tuple t to have a variable-value pair $\$V = \perp$ for every variable $\$V$ of $\overline{V}$ that is not included in t. i.e., *pad* pads t with nulls so that it has the desired schema $\overline{V}$. The function is overloaded to apply on a set of tuples S, so that $pad_{\overline{V}}(S)$ extends each tuple of S. Given two tuples t and t', it is said that t is *more specific* than t' if for every attribute/variable $\$V$, either $t.\$V = t'.\$V$ or $t'.\$V = \perp$. For example, $[\$A = n_a, \$B = \perp, \$C = \perp]$ is less specific than $[\$A = n_a, \$B = n_b, \$C = \perp]$. Given a set S of tuples named $sp(S)$ the set that consists of all tuples S that are not less specific than any other tuple of S. For example, $sp(\{[\$A = n_a, \$B = \perp, \$C = \perp], [\$A = n_a, \$B = n_b, \$C = \perp]\}) = \{[\$A = n_a, \$B = n_b, \$C = \perp]\}$.

Then given a tree pattern T with set of variables $\overline{V}$ and optional edges, the set of tree patterns $T^1,...,T^n$ are created that have no optional edges and are obtained by non-deterministically replacing each optional edge with a required edge, or removing the edge and the

subtree that is adjacent to it. The embeddings of the pattern T were than defined as:

$$sp(pad_{\overline{V}}(map(T^1, I) \cup \ldots \cup map(T^n, I)))$$

mapping an unordered tree pattern to a value produces unordered tuples. In an *ordered tree pattern* the variables are ordered by the preorder traversal sequence of the tree pattern, and this ordering translates into an order of the attribute values of each result tuple. Tuples in the mapping result are then ordered lexicographically according to the order of their attribute values.

The navigation operator nav_T inputs and outputs a list of tuples with schema V. The parameter T is a tree pattern, whose root must be labeled with a variable $\$R$ that also appears in V. The input to the operator is a list of the form $\langle t_1,...,t_n \rangle$, where each t_i, $i = 1,..., n$ is a tuple of the form $[...,\$R = r_i,...]$. The output of the operator is the list of tuples $t_i + t'_i$, for all i, where t'_i is defined as $t'_i \in map(T, r_i)$.

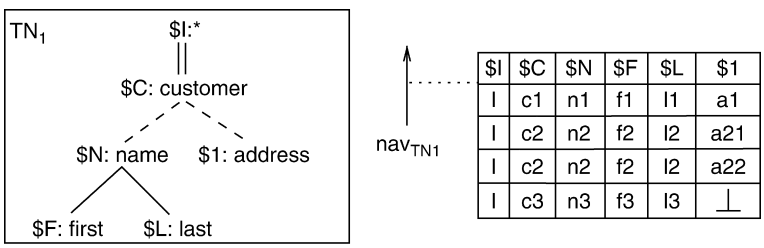
Figure 2 shows the pattern TN_1 corresponding to the navigation part in query Q1, and the result of nav_{TN_1} on the sample document. The order of the tuples is justified as follows: The depth-first pre-order of the variables in the tree pattern is ($\$C$, $\$N$, $\$F$, $\$L$, $1). Consequently, the first tuple precedes the second because $c_1 << c_2$ and the second tuple precedes the third tuple because $a_{21} << a_{22}$.

Construction Tuple-based XQuery algebras include operators that construct XML from the bindings of variables. This is captured by the $crList_L$ operator, which inputs a collection of tuples and outputs an XML tree/forest. The parameter L is a list of *construction tree patterns*, called a *construction pattern list*. For example, consider the query:

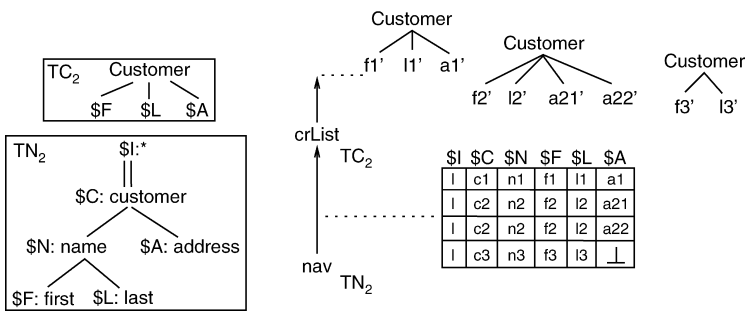
```
for $C in $I//customer, $N in $C/name,
    $F in $N/first, $L in $N/last,
    $A in $C/address
return <customer> { $F, $L, $A } </customer>
```
(Q2)

Figure 3 depicts the navigation pattern TN_2, the construction pattern list TC_2, which consists of a single construction pattern, and a simple algebraic expression, corresponding to Q2.

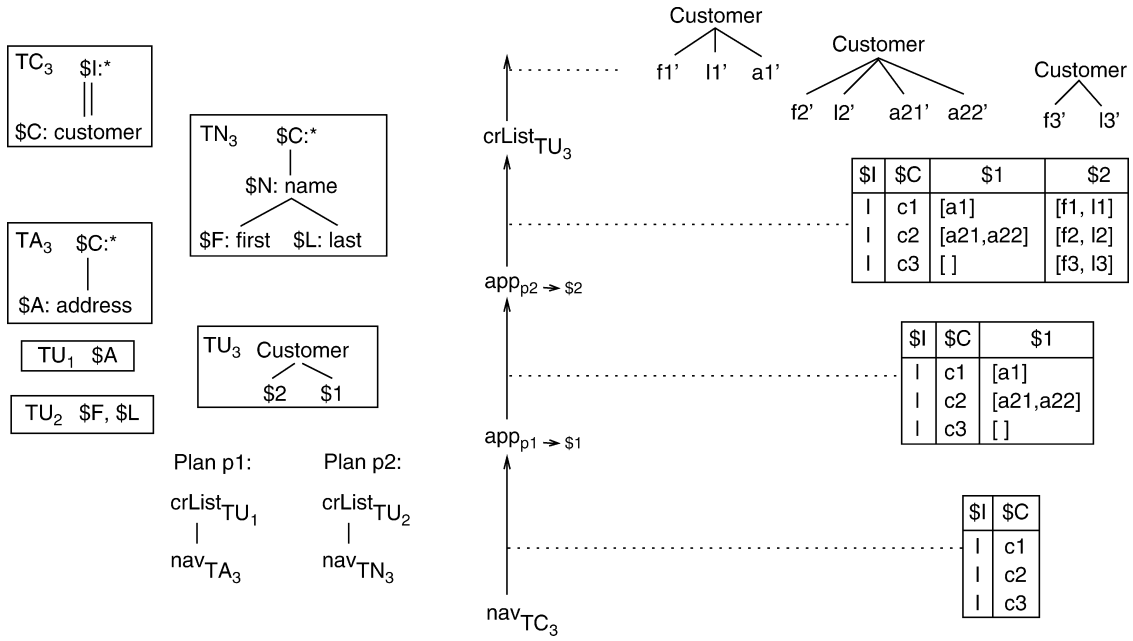
Nested Plans The combination of navigation and construction operators captures the navigation and construction of unnested FLWR XQuery expressions. The following operators can be used to handle nested queries.



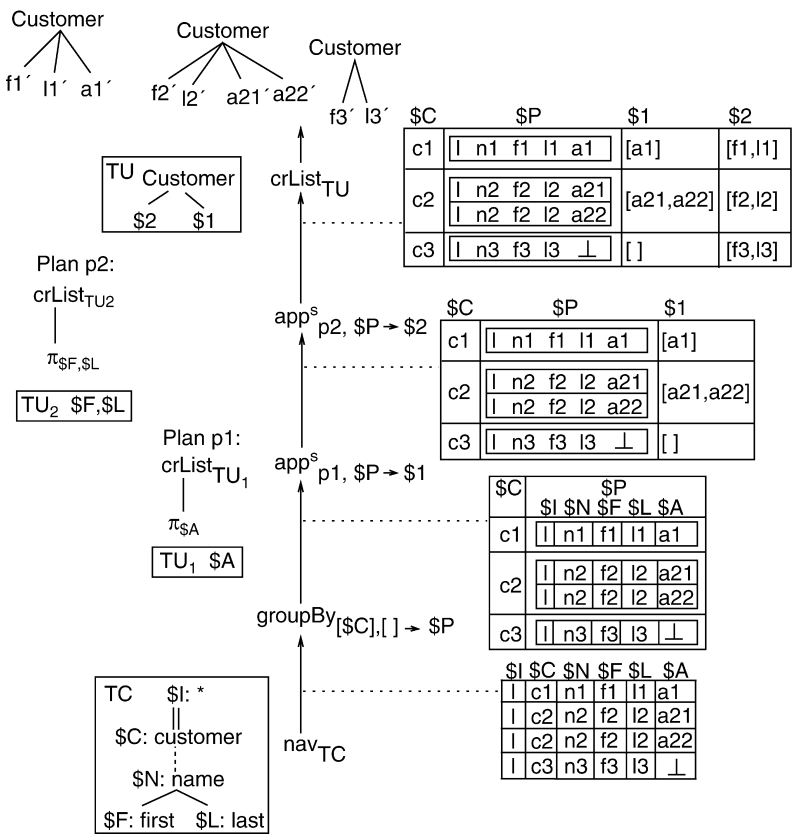
XML Tuple Algebra. Figure 2. Tree pattern for XQuery Q1.



XML Tuple Algebra. Figure 3. Navigation and construction for Q2.



XML Tuple Algebra. Figure 4. Algebraic plan for Q1.



XML Tuple Algebra. Figure 5. Alternative algebraic plan for Q1.

Apply The $app_{p\mapsto\$R}$ (as in "*app*ly plan") unary operator takes as parameter an algebra expression p, which delivers an XML "forest", and a result variable $\$R$, which should not appear in the schema V of the input tuples. Intuitively, for every tuple t in the input collection I, $p(\{t\})$ is evaluated and the result is assigned to $\$R$.

$$app_{p\mapsto R}(I) = \{t + (R = r) | t \in I, R = p(\{t\})\}$$

For example, query Q1 produces a `customer` output element for every `customer` element in the input. The output element includes the first name and last name pairs (if any) of the customer and for each name all the address children (if any) of the input element. Figure 4 depicts an algebraic plan for Q1, using the *app* operator. TC_3, TA_3 and TN_3 are navigation patterns, while TU_1, TU_2 and TU_3 are construction pattern lists, consisting of one, two, and one, respectively, construction patterns. The partial plans p_1 and p_2 each apply some navigation starting from $\$C$ and construct XML output in \$1 and, respectively, \$2. The first app operator reflects the XQuery nesting, while the second app corresponds to the inner return clause. The final crList operator produces `customer` elements.

The *app* operator is defined on one tuple at a time. However, many architectures (and algebras) consider also a set-at-a-time execution. For instance, instead of evaluating $app_{p1\mapsto\$1}$ on one tuple at a time (which means twice for customer $c2$ since she has two addresses), one could group its input by customer ID, and apply p_1 on one group of tuples at a time. In such cases, the following pair of operators is useful.

Group-By The $groupBy_{\overline{G_{id}},\overline{G_v}\mapsto\$R}$ has three parameters: the list of group-by-id variables $\overline{G_{id}}$, the list of group-by-value variables $\overline{G_v}$, and the result variable $\$R$. The operator partitions its input I into sets of tuples $P_{\overline{G}}(I)$ such that all tuples of a partition have id-equal values for the variables of $\overline{G_{id}}$ and equal values for the variables of $\overline{G_v}$. The output consists of one tuple for each partition. Each output tuple has the variables of $\overline{G_{id}}$ and $\overline{G_v}$ and an additional variable $\$R$, whose value is the partition (Some algebras do not repeat the variables of $\overline{G_{id}}$ and $\overline{G_v}$ in the partition, for efficiency reasons.). Note that input tuples that have a $\perp$ in any of the of the $\overline{G_{id}}$ or $\overline{G_v}$ variables are not considered.

Apply-on-Set The $app^s_{p,\$v\mapsto\$R}$ operator assumes that the variable $\$V$ of its input I is bound to a collection of tuples. The operator applies the plan p on $t.\$V$ for every tuple t from the input, and assigns the result to the new variable $\$R$.

For example, Fig. 5 shows another possible plan for Q1. This time a single navigation pattern TC, is used which includes optional edges. The navigation result may contain several tuples for each customer with multiple addresses. Thus, the navigation result is grouped by $\$C$ before applying p_1 on the sets.

The benefits of implementing nested plans by using grouping and operators that potentially deliver null tuples (outerjoin in particular) had first been observed in the context of OQL.

Comparing Fig. 5 with Fig. 4 shows that the same query may be expressed by different expressions in the unified algebra. Providing multiple operators (or combinations thereof) which lead to the same computation is typical in algebras.

Key Applications

An XML tuple algebra is an important intermediate representation for the investigation and the implementation of efficient XML processing techniques. An XML tuple algebra or similar extensions to relational algebra are used as of 2008 by XQuery processing systems such as BEA Aqualogic Data Services Platform and the open source MonetDB/XQuery system.

Cross-references

- ▶ Top-k XML Query Processing
- ▶ XML Document
- ▶ XML Element
- ▶ XML Storage
- ▶ XML Tree Pattern, XML Twig Query
- ▶ XPath/XQuery

Recommended Reading

1. Arion A., Benzaken V., Manolescu I., Papakonstantinou Y., and Vijay R. Algebra-based identification of tree patterns in XQuery. In Proc. 7th Int. Conf. Flexible Query Answering Systems, 2006, pp. 13–25.
2. Beeri C. and Tzaban Y. SAL: an algebra for semistructured data and XML. In Proc. ACM SIGMOD Workshop on The Web and Databases, 1999, pp. 37–42.
3. Cluet S. and Moerkotte G. Nested Queries in Object Bases. Technical report, 1995.
4. Deutsch A., Papakonstantinou Y., and Xu Y. The NEXT logical framework for XQuery. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004, pp. 168–179.

5. Michiels P., Mihaila G.A., and Siméon J. Put a tree pattern in your algebra. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 246–255.
6. Papakonstantinou Y., Borkar V.R., Orgiyan M., Stathatos K., Suta L.,Vassalos V., and Velikhov P. XML queries and algebra in the Enosys integration platform. Data Knowl. Eng., 44 (3):299–322, 2003.
7. Re C., Siméon J., and Fernández M. A complete and efficient algebraic compiler for XQuery. In Proc. 22nd Int. Conf. on Data Engineering, 2006, p. 14.
8. The XQuery Language. www.w3.org/TR/xquery, 2004.
9. XQuery 1.0 and XPath 2.0 Data Model. www.w3.org/TR/xpath-datamodel.
10. XQuery 1.0 Formal Semantics. www.w3.org/TR/2005/WD-xquery-semantics.

XML Typechecking

Véronique Benzaken[1], Giuseppe Castagna[2]
Haruo Hosoya[3], Benjamin C. Pierce[4],
Stijn Vansummeren[5]
[1]University Paris 11, Orsay Cedex, France
[2]C.N.R.S. and University Paris 7, Paris, France
[3]The University of Tokyo, Tokyo, Japan
[4]University of Pennsylvania, Philadelphia, PA, USA
[5]Hasselt University and Transnational University of Limburg, Diepenbeek, Belgium

Definition

In general, *typechecking* refers to the problem where, given a program P, an input type σ, and an output type τ, one must decide whether P is *type-safe*, that is, whether it produces only outputs of type τ when run on inputs of type σ. In the XML context, typechecking problems mainly arise in two forms:

- *XML-to-XML transformations,* where P transforms XML documents conforming to a given type into XML documents conforming to another given type.
- *XML publishing,* where P transforms relational databases into XML views of these databases and it is necessary to check that all generated views conform to a specified type.

A *type* for XML documents is typically a regular tree language, usually expressed as a schema written in a schema language such as DTD, XML Schema, or Relax NG (see *XML Types*). In the XML publishing case, the input type σ is a relational database schema, possibly with *integrity constraints.*

Typechecking problems may or may not be decidable, depending on (i) the class of programs considered, (ii) the class of input types (relational schemas, DTDs, XML Schemas, Relax NG schema, or perhaps other subclasses of the regular tree languages), and (iii) the class of output types. In cases where it is decidable, typechecking can be done *exactly.* In cases where it is undecidable, one must revert to *approximate* or *incomplete* typecheckers that may return false negatives – i.e., may reject a program even if it is type-safe. Even when exact typechecking is possible, approximate typechecking may be preferable as this is often computationally cheaper than exact typechecking.

In the programming languages literature, typechecking often not only entails verifying that all outputs are of type τ, but also requires detecting when the program may abort with a run-time error on inputs of type σ [3]. The above definition encompasses such cases: view run-time errors as a special result value error and then typecheck a program against an output type that does not contain the value error.

Historical Background

Although typechecking is a fundamental and well-studied problem in the theory of programming languages [3], the types necessary for XML typechecking (based on regular tree languages) differ significantly from the conventional data types usually considered (i.e., lists, records, classes, and so on). Indeed, although it is possible to encode XML types into conventional datatypes, this encoding lacks flexibility in the sense that programs tend to need artificial changes when types evolve [22]. For this reason, Hosoya et al. [22] proposed regular tree languages as the "right" notion of types for XML and presented an approximate typechecker in this context. The typechecker was implemented in the XML-to-XML transformation language XDuce [21] whose approach was later extended to general purpose programming by CDuce (functional programming) and Xtatic (object-oriented programming). XDuce's approach also lies at the basis of XQuery's typechecking algorithm [6].

The contemporary study of exact typechecking for XML-to-XML transformations started with an investigation of relatively simple transformation languages [29,32,33]. Ironically enough, the fundamentals of exact typechecking for more advanced transformation languages were already investigated a long time before XML appeared [7,8]. These fundamentals were

revived in the XML era by Milo, Suciu, and Vianu in their seminal work on *k*-pebble tree transducers [30], which was later extended to other transformation languages [26,39,40]. The computational complexity of exact typechecking was investigated in [14,27,28]. Exact typechecking algorithms for XML publishing scenarios were given by Alon et al. [1].

Foundations

Exact Typechecking

XML-to-XML Transformations Recall that in this setting, P is a program that should transform XML documents of a type σ into documents of a type τ. When the languages in which the transformation and the types are expressed are sufficiently restricted in power, exact typechecking is possible. There are two major approaches to the construction of an exact typechecking algorithm: *forward inference* and *backward inference*. Forward inference solves the typechecking problem directly by first computing the image O of the input type σ under the transformation P, i.e., $O := \{P(t) | t \in \sigma\}$, and then checking $O \subseteq \tau$ [28,30,32,33,37]. This approach does not work if O goes beyond *context-free tree languages* as checking $O \subseteq \tau$ then becomes undecidable. Sadly, this is already the case when P is written in very simple transformation languages, such as the *top-down tree transducers* (this fact is known as folklore; see, e.g., [14].) Also, computing O itself becomes undecidable for more advanced transformation languages.

Backward inference, on the other hand, first computes the pre-image I of the output type τ under P, i.e., $I := \{t \mid P(t) \in \tau\}$, and then checks $\sigma \subseteq I$. Backward inference often works even when the transformation language is too expressive for forward inference. The technique has successfully been applied to a range of formal models of real-world transformation languages like XSLT, from the top-down and bottom-up tree transducers [7], to macro tree transducers [8,14,27], macro forest transducers [34], *k*-pebble tree transducers [30], tree transducers based on alternating tree automata [39], tree transducers dealing with atomic data values [37], and high-level tree transducers [40].

As mentioned in the definition, static detection of run-time errors can be phrased as a particular form of typechecking by introducing a special output value error. Exact typechecking in this form has been investigated for XQuery programs written in the nonrecursive for-let-where return fragment of XQuery without automatic coercions but with the various XPath axes; node constructors; value and node comparisons; and node label and content inspections, in the setting where the input type σ is given by a recursion-free regular tree language. The crux of decidability here is a small-model property: if $P(t) =$ error for some t of type σ then there exists another input t' of type σ whose size depends only on P and σ such that $P(t') =$ error. It then suffices to enumerate all inputs $t' \in \sigma$ up to the maximum size and check $P(t') =$ error. There are only a finite number of such t' (up to isomorphism, and P cannot distinguish between isomorphic inputs), from which decidability follows [41]. This small model property continues to hold when we extend the above XQuery fragment with arbitrary primitives satisfying some general niceness properties [41].

XML Publishing In this setting, the input to the program P is a relational database D. Suppose that P computes its output XML tree by posing simple select-project-join queries to D, nesting the results to these queries, and constructing new XML elements. Exact typechecking for programs of this form, when the input type σ is a relational database schema with key and foreign key constraints, and the output type τ is a "star-free" DTD, is decidable [1]. (See [1] for a precise definition and examples of the concept "star-free.") As was the case for detecting runtime errors in XQuery programs, the crux of decidability here is again a small model property. Typechecking remains decidable for output DTDs τ that are not star-free, but then the queries in P must not use projection. Typechecking unfortunately becomes undecidable when the output types τ are given by XML Schemas or Relax NG Schemas [1]. Typechecking also becomes undecidable when P uses queries more expressive than select-project-join queries.

Approximate Typechecking

The expressive power that realistic applications require of practical transformation languages is often too high to allow for exact typechecking. In such cases, one must revert to approximate or incomplete typecheckers that guarantee that all successfully checked programs are type-safe, but that may also reject some type-safe programs. Existing techniques

can be grouped into two categories: type systems and flow analyses.

Type Systems Many conventional programming languages (such as C and Java) specify what programs to accept by a *type system* [35]. Typically, such a system consists of a set of *typing rules* that determine the type of each subexpression of a program. Often, in order to help the typechecker, the programmer is required to supply *type annotations* on variable declarations and in other specified places.

The pioneer work applying this approach to the XML setting was the XDuce (transduce) language [21], whose type system is based on regular tree languages. One significant point in this work is its definition of a natural notion of *subtyping* as the inclusion relation between regular tree languages and its demonstration of the usefulness of allowing a value of one type to be viewed as another type with a syntactically completely different structure [22]. In addition, although the decision problem for subtyping is known to be EXPTIME-complete, the "top-down algorithm" used in the XDuce implementation is empirically shown to be efficient in most cases that actually arise in typechecking [13,22,36]. Such a type system also needs machinery to reduce the amount of type annotations that otherwise tends to be a burden to the user, in particular when the language supports a non-trivial mechanism to manipulate XML documents such as regular expression patterns [20] or filter expressions [17]. A series of works address this problem by proposing automatic type inference schemes that have certain precision properties in a sense similar to the exact typechecking in the previous section [17,20,42]. These ideas have further been extended for XML attributes [19] and para-metric polymorphism [18,43].

CDuce (pronounced "seduce") extends XDuce in various ways [2]. From a language point of view, CDuce embraces XDuce's approach of a functional language based on regular expression patterns [20] and extends it with finer-grained pattern matching, complete two-way compatibility with programs and libraries in the OCaml programming language, Unicode, queries, XML Schema validation, and, above all, higher-order and overloaded functions. XML types are enriched with general purpose data types, intersection and negation types, and functional types. Finally, the CDuce type inference algorithm for patterns is implemented by a new kind of tree automaton and proved to be optimal [10]. Among these extensions, the addition of higher-order functions is significant. Theoretically, this extension is not trivial, first because functions do not fit well in the framework of finite tree automata, but more deeply because this entails a definitional cycle: the definition of typechecking uses subtyping, whose definition then uses the semantics of types (NB: subtyping is defined as inclusion between the sets denoted by given two types), whose definition in turn uses well-typedness of values; the last part depends on typechecking in the presence of higher-order functions since typechecking of a function abstraction $\lambda x.e$ requires analysis of its internal expression e. Some solutions are known for breaking this circularity [14,43]. Also, an approach to combine one of these treatments with high-order functions has been proposed [43].

Several research groups explore ways of mixing a XDuce-like type system with an existing popular language. Xtatic [15] carries out this program for the C# language, developing techniques to blend regular expression types with an object-oriented type system. XJ [16] makes a closely related effort for Java. OCamlDuce [11] mixes with OCaml, proposing a method to intermingle a standard ML type inference algorithm with XDuce-like typechecking. XHaskell [25] is also another instance for Haskell; their approach is, however, to embed XML types into Haskell typing structures (such as tuples and disjoint sums) in the style of data-binding, yet support XDuce-like subtyping in its full flexibility by deploying a coercion technique [25].

The formal semantics of XQuery defined by the W3C contains a type system based on a set of inductive typing rules [6]. Their first draft was heavily based on XDuce's type system [9]. Later, they switched to a different one that reflects the object-oriented hierarchical typing structure adopted by XML Schema.

As byproducts of the above pieces of work, several optimization and compilation techniques that exploit typing information have been proposed [10,24].

Flow-Analysis Flow analysis is a static analysis technique that has long been studied in the programming language community. A series of investigations has been conducted for adapting flow analysis to approximate XML typechecking, concurrently to XDuce-related work, [3,23,31]. In this approach, the user needs to write no type annotations for intermediate values like in XDuce, but instead the static anlyzer completely infers them, thus providing a more

user-friendly system. One potential drawback is that the specification is rather informal and therefore, when the analyzer raises an error, the reason can sometimes be unclear; empirically, however, such false negatives are rare.

Flow analysis is applied first to Bigwig language system, an extension of Java with an XML-manipulating facility called "templates" [3]. Though this first attempt handles only XHTML types, they naturally generalize it to arbitrary XML types, calling the resulting system XAct [40]. Their techniques are further extended and applied to static analysis of XSLT [31].

Key Applications

XML typechecking is a key component of XQuery, the standard XML query language. As outlined above, XML typechecking in XQuery is based on a set of inductive typing rules that reflects the object-oriented hierarchical typing structure adopted by XML Schema. Different approaches to XML typechecking may be found in research prototypes like CDuce, OcamlDuce, XDuce, XAct, XHaskell, and Xtatic for which references are given below.

Url to Code

CDuce: http://www.cduce.org
OCamlDuce: http://www.cduce.org/ocaml.html
XAct: http://www.brics.dk/Xact/
Xduce: http://xduce.sourceforge.net
XHaskell: http://taichi.ddns.comp.nus.edu.sg/taichi-wiki/XhaskellHomePage
Xtatic: http://www.cis.upenn.edu/~bcpierce/xtatic

A gentle introduction to exact typechecking for both XML-to-XML transformations and XML Publishing can be found in [37]. A non-technical presentation of Regular Expression Types and Patterns and their use in query languages can be found in the joint DPBL and XSym 2005 invited talk [4]. For a more complete presentation of Regular Expression Types and Patterns and the associated type-checking and subtyping algorithms we recommend the reader to refer to the seminal JFP article by Hosoya, Pierce, and Vouillon [22]. The joint ICALP and PPDP 2005 keynote talk [5] constitutes a relatively simple survey of the problem of type-checking higher-order functions and an overview on how to derive subtyping algorithms semantically: full technical details can be found in an extended version published in the JACM [13].

Cross-references

- ▶ Database Dependencies
- ▶ XML
- ▶ XML Types
- ▶ XPath/XQuery

Recommended Reading

1. Alon N., Milo T., Neven F., Suciu D., and Vianu V. Typechecking xml views of relational databases. ACM Trans. Comput. Log., 4(3):315–354, 2003.
2. Benzaken V., Castagna G., and Frisch A. CDuce: an XML-centric general-purpose language. In Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming, 2003, pp. 51–63.
3. Brabrand C., Møller A., and Schwartzbach M.I. The <bigwig> project. ACM Trans. Internet Tech., 2(2):79–114, 2002.
4. Castagna G. Patterns and types for querying XML. In Proc. of DBPL 2005, Tenth International Symposium on Database Programming Languages, 2005, pp. 1–26.
5. Castagna G. and Frisch A. A gentle introduction to semantic subtyping. In Proc. 7th Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, 2005, pp. 198–208.
6. Draper D., Fankhauser P., Ashok Malhotra M.F., Rose K., Rys M., Siméon J., and Wadler P. XQuery 1.0 and XPath 2.0 Formal Semantics, 2007. http://www.w3.org/Tr/query-semantics/.
7. Engelfriet J. Top-down tree transducers with regular look-ahead. Math. Syst. Theory, 10:289–303, 1977.
8. Engelfriet J. and Vogler H. Macro tree transducers. J. Comput. Syst. Sci., 31(1):710–146, 1985.
9. Fernández M.F., Siméon J., and Wadler P. A semi-monad for semi-structured data. In Proc. 8th Int. Conf. on Database Theory, 2001, pp. 263–300.
10. Frisch A. Regular tree language recognition with static information. In Proc. 3rd IFIP Int. Conf. on Theoretical Computer Science, 2004, pp. 661–674.
11. Frisch A. OCaml+CDuce. In Proc. 11th ACM SIGPLAN Int. Conf. on Functional Programming, 2006, pp. 192–200.
12. Frisch A., Castagna G., and Benzaken V. Semantic subtyping. In Proc. 17th IEE Conf. on Logic in Computer Science, 2002, pp. 137–146.
13. Frisch A., Castagna G., and Benzaken V. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. J. ACM, 55(4):1–64, 2008.
14. Frisch A. and Hosoya H. Towards practical typechecking for macro tree transducers. In Proc. 11th Int. Workshop on Database Programming Languages, 2007, pp. 246–261.
15. Gapeyev V., Levin M.Y., Pierce B.C., and Schmitt A. The Xtatic experience. In Proc. Workshop on Programming Language Technologies for XML (PLAN-X). January 2005. University of Pennsylvania Technical Report MS-CIS-04-24, 2004.
16. Harren M., Raghavachari M., Shmueli O., Burke M.G., Bordawekar R., Pechtchanski I., and Sarkar V. XJ: facilitating XML processing in Java. In Proc. 14th Int. World Wide Web Conference, 2005, pp. 278–287.
17. Hosoya H. Regular expression filters for XML. J. Funct. Program., 16(6):711–750, 2006.

18. Hosoya H., Frisch A., and Castagna G. Parametric polymorphism for XML. In Proc. 32nd ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, 2005, pp. 50–62.
19. Hosoya H. and Murata M. Boolean operations and inclusion test for attribute-element constraints. Theor. Comput. Sci., 360 (1–3):327–351, 2006.
20. Hosoya H. and Pierce B.C. Regular expression pattern matching for XML. J. Funct. Program., 13(6):961–1004, 2002.
21. Hosoya H. and Pierce B.C. XDuce: a typed XML processing language. ACM Trans. Internet Tech., 3(2):117–148,2003.
22. Hosoya H., Vouillon J., and Pierce B.C. Regular expression types for XML. ACM Trans. Program. Lang. Syst., 27(1):46–90, 2004.
23. Kirkegaard C. and Møller A. Xact – XML transformations in Java. In Proc. Programming Language Technologies for XML, 2006, p. 87.
24. Levin M.Y. and Pierce B.C. Type-based optimization for regular patterns. In Proc. 10th Int. Workshop on Database Programming Languages, 2005, pp. 184–198.
25. Lu K.Z.M. and Sulzmann M. XHaskell: regular expression types for Haskell. Manuscript, 2004.
26. Maneth S., Perst T., Berlea A., and Seidl H. XML type checking with macro tree transducers. In Proc. 24th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2005, pp. 283–294.
27. Maneth S., Perst T., and Seidl H. Exact XML type checking in polynomial time. In Proc. 11th Int. Conf. on Database Theory, 2007, pp. 254–268.
28. Martens W. and Neven F. Frontiers of tractability for typechecking simple xml transformations. J. Comput. Syst. Sci., 73(3):362–390, 2007.
29. Milo T. and Suciu D. Type inference for queries on semistructured data. In Proc. 18th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1999, pp. 215–226.
30. Milo T., Suciu D., and Vianu V. Typechecking for XML transformers. J. Comput. Syst. Sci., 66(1):66–97, 2003.
31. Møller A., Olesen M.O., and Schwartzbach M.I. Static validation of XSL transformations. ACM Trans. Programming Languages and Syst., 29(4): Article 21, 2007.
32. Murata M. Transformation of documents and schemas by patterns and contextual conditions. In Proc. 3rd Int. Workshop on Principles of Document Processing, 1996, pp. 153–169.
33. Papakonstantinou Y. and Vianu V. DTD inference for views of XML data. In Proc. 19th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2000, pp. 35–46.
34. Perst T. and Seidl H. Macro forest transducers. Inf. Process. Lett., 89(3):141–149, 2004.
35. Pierce B.C. Types and Programming Languages. MIT, 2002.
36. Suda T. and Hosoya H. Non-backtracking top-down algorithm for checking tree automata containment. In Proc. 10th Int. Conf. Implementation and Application of Automata, 2005, pp. 83–92.
37. Suciu D. The XML typechecking problem. ACM SIGMOD Rec., 31(1):89–96, 2002.
38. Sulzmann M. and Lu K.Z.M. A type-safe embedding of XDuce into ML. Electr. Notes Theor. Comput. Sci., 148(2):239–264, 2006.
39. Tozawa A. Towards static type checking for XSLT. In Proc. 1st ACM Symp. on Document Engineering, 2001, pp. 18–27.
40. Tozawa A. XML type checking using high-level tree transducer. In Proc. 8th Int. Symp. Functional and Logic Programming, 2006, pp. 81–96.
41. Vansummeren S. On deciding well-definedness for query languages on trees. J. ACM, 54(4):19, 2007.
42. Vouillon J. Polymorphism and XDuce-style patterns. In Proc. Programming Languages Technologies for XML, 2006, pp. 49–60.
43. Vouillon J. Polymorphic regular tree types and patterns. In Proc. 33rd ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, 2006, pp. 103–114.

XML Types

Frank Neven
Hasselt University and Transnational University of Limburg, Diepenbeek, Belgium

Synonyms

XML schemas

Definition

To constrain the structure of allowed XML documents, for instance with respect to a specific application, a target schema can be defined in some schema language. A schema consists of a sequence of type definitions specifying a (possibly infinite) class of XML documents. A type can be assigned to every element in a document valid w.r.t. a schema. As the same holds for the root element, the document itself can also be viewed to be of a specific type. The schema languages DTDs, XML Schema, and Relax NG, are, on an abstract level, different instantiations of the abstract model of unranked regular tree languages.

Historical Background

Brüggemann-Klein et al. [3] were the first to revive the theory of regular unranked tree automata [14] for the modelling of XML schema languages. Murata et al. [10] provided the formal taxonomy as presented here. Martens et al. [8] characterized the expressiveness of the different models and provided type-free abstractions.

Foundations

Intuition

Consider the XML document in Fig. 1 that contains information about store orders and stock contents.

```
<store>
  <order>
    <customer>
       <name>John Mitchell</name>
       <email> j.mitchell@yahoo.com </email>
    </customer>
    <item> <id> I18F </id>
          <price> 100 </price>
    </item>
    <item>... </item>... <item>... </item>
  </order>
  <order>... </order>... <order>... </order>
  <stock>
    <item>
      <id> IG8 </id> <qty> 10 </qty>
      <supplier> <name> Al Jones </name>
                 <email> a.j@gmail.com </email>
                 <email> a.j@dot.com </email>
      </supplier>
     </item>
    <item>
       <id> J38H </id> <qty> 30 </qty>
       <item>
          <id> J38H1 </id> <qty> 10 </qty>
          <supplier> ... </supplier>
       </item>
       <item>
          <id> J38H2 </id> <qty> 1 </qty>
          <supplier> ... </supplier>
       </item>
       <item> ... </item> ... <item> ... </item>
    </item>
    ...
    <item> ... </item>
  </stock>
</store>
```

XML Types. Figure 1. Example XML document.

Orders hold customer information and list the items ordered, with each item stating its id and price. The stock contents consists of the list of items in stock, with each item stating its id, the quantity in stock, and – depending on whether the item is atomic or composed from other items – some supplier information or the items of which they are composed, respectively. It is important to emphasize that order items do not include supplier information, nor do they mention other items. Moreover, stock items do not mention prices. DTDs are incapable of distinguishing between order items and stock items because the content model of an element can only depend on the element's name in a DTD, and not on the context in which it is used. For example, although the DTD in Fig. 2 describes all intended XML documents, it also allows supplier information to occur in order items and price information to occur in stock items.

The W3C specification essentially defines an XSD as a collection of *type definitions*, which, when abstracted away from the concrete XML representation of XSDs, are rules like

$$store \rightarrow \texttt{order}[order]^*, \texttt{stock}[stock] \qquad (\star)$$

that map type names to regular expressions over pairs $a[t]$ of element names a and type names t. Intuitively, this particular type definition specifies an XML fragment to be of type *store* if it is of the form where $n \geq 0$; $f_1,...,f_n$ are XML fragments of type *order*; and g is an XML fragment of type *stock*. Each type name that occurs on the right hand side of a type definition in an XSD must also be defined in the XSD, and each type name may be defined only once. Using types, an XSD can specify that an item is an order item when it occurs under an order element and is otherwise a stock item. For example, Fig. 2 shows an XSD describing the intended set of store document. Note in particular the use of the types $item_1$ and $item_2$ to distinguish between order items and stock items.

It is important to remark that the "Element Declaration Consistent" constraint of the W3C specification requires multiple occurrences of the same element name in a single type definition to occur with the same type. Hence, type definition ($\star$) is legal, but

$$persons \rightarrow (\texttt{person}[male] + \texttt{person}[female])^+$$

is not, as `person` occurs both with type *male* and type *female*. Of course, element names in *different* type definitions can occur with different types (which is exactly what yields the ability to let the content model of an element depend on its context). On a structural level, ignoring attributes and the concrete syntax, the structural expressiveness of Relax NG corresponds to XSDs without the EDC constraint.

A Formalization of Relax NG

An XML fragment $f = f_1...f_n$ is a sequence of labeled trees where every tree consists of a finite number of nodes, and every node v is assigned an element name denoted by $lab(v)$. There is always a virtual root which

```
<!ELEMENT store (order*, stock)>
<!ELEMENT order (customer, item+)>
<!ELEMENT customer (name, email*)>
<!ELEMENT item (id,price+ (qty,(supplier
                                  +item+)))>
<!ELEMENT stock (item+)>
<!ELEMENT supplier (name, email*)>
```

```
root    → store [store]
store   → order [order]*, stock [stock]
order   → customer [person], item [item1]+
person  → name [emp], email [emp]+
item1   → id [emp], qty [emp], price [emp]
stock   → item [item2]+
item2   → id [emp], qty[emp],
              (supplier [person] + item [item2]+)
emp     → ε
```

XML Types. Figure 2. A DTD and an XSD describing the document in Fig. 1.

acts as the common parent of the roots of the different f_i. For a set EName and Types of element and type names, respectively, the set of elements is defined as $\{a[t] | a \in \text{EName}, t \in \text{Types}\}$. The set of regular expressions is given by the following syntax:

$$r ::= \epsilon \mid \alpha \mid r, r \mid r + r \mid r^* \mid r^+ \mid r?$$

where ε denotes the empty string and α is an element. Their semantics is the usual one and is therefore omitted.

An *XSchema* is a tuple $S = (\text{EName}, \text{Types}, \rho, t_0)$ where EName and Types are finite sets of elements and types, respectively, ρ is a mapping from Types to regular expressions, and, $t_0 \in$ Types is the start type.

A *typing* τ of f is a mapping assigning a type $\tau(v) \in$ Types to every node v in f (including the virtual root). For a node v with children $v_1,\ldots,v_m$, define child-string (τ, v) as the string $\text{lab}(v_1)[\tau(v_1)]\ldots\text{lab}(v_1)[\tau(v_1)]$. An XML fragment f then *conforms to* or is *valid w.r.t.S* if there is a typing τ of f such that for every node v, child-string (τ, v) matches the regular expression $\rho(\tau(v))$, and $\tau(\text{root}) = t_0$. The mapping τ is then called a *valid typing*.

Despite the clean formalization, the above definition does not entail a validation algorithm. One possibility is to compute for each node v in f a set of possible types $\Delta(v) \subseteq$ Types such that for each type $t \in \Delta(v)$, the XML subfragment rooted at v is valid w.r.t. the schema with start type t. The XML fragment is then valid w.r.t. S itself when the start type t_0 belongs to $\Delta(\text{root})$. The sets $\Delta(v)$ can be computed in a bottom-up fashion. Indeed, $t \in \Delta(v)$ iff (i) v is a leaf node and $\rho(t)$ contains the empty string; or, (ii) v is a non-leaf node with children $v_1,\ldots,v_n$ and there are $t_1 \in \Delta(v_1),\ldots,t_n \in \Delta(v_n)$ such that $\text{lab}(v_1)[t_1]\ldots\text{lab}(v_n)[t_n] \in \rho(t)$. A valid typing can then be computed from the sets Δ by an additional top-down pass through the tree. Although this kind of bottom-up validation is a bit at odds with the general concept of top-down or streaming XML processing, the algorithm can be adapted to this end (cf. for instance, [10,13]). For general XSchema's, a valid typing is not necessarily unique and cannot always be computed in a single pass [8].

XSchemas as defined above correspond precisely to the class of unranked regular hedge languages [3] and can be seen as an abstraction of Relax NG. Note that the present formalization is overly simplistic w.r.t. attributes as Relax NG treats them in a way uniform to elements using attribute–element constraints [6].

Relationship with Tree and Hedge Automata

Although an XML fragment can consist of a sequence of labeled trees, in the literature it is accustomed to restrict this sequence to simply one tree. XSchemas as defined above then define precisely the unranked regular tree languages [3,11]. Although several automata formalism capturing this class have been defined [3,4,5], each with their own advantages [9], XSchemas correspond most closely to the model of Brüggemann-Klein et al. [3], which is defined as follows. A tree automaton $A = (Q, \text{EName}, \delta, q_0)$, where Q is the set of states (or, types), $q_0 \in Q$ is the start state (start type), and δ maps pairs $(q, a) \in Q \times \text{EName}$ to regular expressions over Q. An input tree f is accepted by the automaton if there exists a mapping τ from the nodes of f to Q, called a run (or, a typing), such that the root is labeled with the start state, and for every non-root node v with children $v_1,\ldots,v_n$, the string $\text{lab}(v_1)\ldots\text{lab}(v_n)$ matches $\delta(\tau(v), lab(v))$. The translation between XSchemas and tree automata is folklore and can for instance be found in [3].

Deterministic Regular Expressions

The unique particle attribution constraint (UPA) requires regular expressions to be deterministic in the following sense: the form of the regular expression should allow each symbol of the input string to match

uniquely against a position in the expression when processing the input string in one pass from left to right. That is, without looking ahead in the string. For instance, the expression $r = (a+b)^* a$ is not deterministic as the first symbol in the string *aaa* can already be matched to two different *a*'s in *r*. The equivalent expression $b^* a(b^* a)^*$, on the other hand, is deterministic. Unfortunately, not every regular expression can be rewritten into an equivalent deterministic one [2]. Moreover, it is not a very robust subclass, as it is not closed under union, concatenation, or Kleene-star, prohibiting an elegant constructive definition [2]. Deterministic regular expressions are characterized as one-unambiguous regular expressions by Brüggemann-Klein and Wood [2]. Deciding whether a regular expression is one-unambiguous can be done in quadratic time [1]. Furthermore, it can be decided in EXPTIME whether there is a deterministic regular expression equivalent to a given regular expression [2]. If so, the algorithm can return an expression of a size which is double exponential. It is unclear whether this can be improved.

A Formalization of DTDs and XSDs

Let $S = (\text{EName}, \text{Types}, \rho, t_0)$ be an XSchema. Then, S is *local* when EName = Types and regular expressions in ρ are defined over the alphabet $\{a[a] | a \in \text{EName}\}$. This simply means that the name of the element also functions as its type. Furthermore, S is *single-type* when there are no elements $a[t_1]$ and $a[t_2]$ in a $\rho(t)$ with $t_1 \neq t_2$. A DTD is then a local XSchema where regular expressions are restricted to be deterministic. Finally, an XSD is then a single-type XSchema where regular expressions are restricted to be deterministic.

Expressiveness and Complexity

XSchemas form a very robust class, for instance, equivalent to the monadic second-order logic (MSO) definable classes of unranked trees [12], and are closed under the Boolean operations. XSDs on the other hand are not closed under union or complement [8,10]. They define precisely the subclasses of XSchemas closed under ancestor-guarded subtree exchange, and are much closer to DTDs than to XSchemas as becomes apparent from the following equivalent type-free alternative characterization. A *pattern-based XSDP* is a set of rules $\{r_1 \to s_1, \ldots, r_m \to s_m\}$ where all r_i are horizontal regular expressions and all s_i are deterministic vertical regular expressions. An XML fragment f is *valid* with respect to P if, for every node v of f, there is a rule $r \to s \in P$ such that the string formed by the labels of the nodes on the path from the root to v match r and the string formed by the children of v match s (cf. [7,8] for more details). The single-type restriction further ensures that XSDs can be uniquely typed in a one-pass top-down fashion. To be precise, one-pass typing in a top-down fashion means that the first time a node is visited, a type should be assigned (so only based on what has been seen up to now), and that a child can be visited only when its parent has already visited. Type inclusion and equivalence is EXPTIME-complete for XSchemas and is in PTIME for DTDs and XSDs. In fact, w.r.t. the latter, the problem reduces to the corresponding problem for the class of employed regular expressions [8].

Cross-references

- ▶ XML Schema
- ▶ XML Typechecking

Recommended Reading

1. Brüggemann-Klein A. Regular expressions into finite automata. Theor. Comput. Sci., 120(2):197–213, 1993.
2. Brüggemann-Klein A. and Wood D. One unambiguous regular languages. Inform. Comput., 140(2):229–253, 1998.
3. Brüggemann-Klein A., Murata M., and Wood D. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.
4. Carme J., Niehren J., and Tommasi M. Querying unranked trees with stepwise tree automata. In Proc. 15th Int. Conf. Rewriting Techniques and Applications, 2004, pp. 105–118.
5. Cristau J., Löding C., and Thomas W. Deterministic automata on unranked trees. In Proc. 15th Int. Symp. Fundamentals of Computation Theory, 2005, pp. 68–79.
6. Hosoya H. and Murata M. Boolean operations and inclusion test for attribute-element constraints. Theor. Comput. Sci., 360(1–3):327–351, 2006.
7. Martens W., Neven F., and Schwentick T. Simple off the shelf abstractions for XML schema. ACM SIGMOD Rec., 36(4):15–22, 2007.
8. Martens W., Neven F., Schwentick T., and Bex G.J. Expressiveness and complexity of XML schema. ACM Trans. Database Syst., 31(3):770–813, 2006.
9. Martens W. and Niehren J. Minimizing tree automata for unranked trees. In Proc. 10th Int. Workshop on Database Programming Languages, 2005, pp. 232–246.
10. Murata M., Lee D., Mani M., and Kawaguchi K. Taxonomy of XML schema languages using formal language theory. ACM Trans. Internet Tech., 5(4):660–704, 2005.
11. Neven F. Automata theory for XML researchers. ACM SIGMOD Rec., 31(3):39–46, 2002.

12. Neven F. and Schwentick T. Query automata on finite trees. Theor. Comput. Sci., 275:633–674, 2002.
13. Segoufin L. and Vianu V. Validating streaming XML documents. In Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2002, pp. 53–64.
14. Thatcher J.W. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. J. Comput. Syst. Sci., 1(4):317–322, 1967.

XML Updates

Giorgio Ghelli
University of Pisa, Pisa, Italy

Definition

The term *XML Updates* refers to the act of modifying XML data while preserving its *identity*, through the operators provided by an XML manipulation language. Identity preservation is crucial to this definition: the production of XML data from XML data *without* preserving the original data identity is called *XML transformation*. The general notion of *identity* has many concrete incarnations. The XQuery/XPath data model (see [14]) associates a *Node Identity* to each node of the XML syntax tree. In a language based on this data model, *updates* differ from *transformations* because the former modify the data but preserve node identities. Another hallmark of updates is that an expression that refers to the data being updated may have a different value after the update is evaluated, while the evaluation of XML transformations does not change the value of any other expression.

XML updates may be embedded in any XML manipulation language, but this entry will be focused on XML updates in the context of languages of the XQuery family.

Historical Background

The first languages proposed to manipulate XML data did not support XML updates, because XML transformations can often be used as a substitute for XML updates, but the former are simpler to optimize and have cleaner semantics. However, many applications need to update persistent XML data, and eventually the problem was tackled. The first widely-known proposal in the scientific literature was presented in [11], and was clearly influenced by previous work on SQL updates and on primitives for tree updates.

Foundations

The design of an XML update mechanism has three important aspects:

- Definition of the update operators
- Revalidation of modified data after the updates
- Embedding of the operators in the language

Operators

There is a wide agreement on the basic update operators. Almost every proposal includes operators to delete a subtree, insert a subtree in a specific position, rename a node, and replace the value of a node. The deletion of a node T may be defined as an operation that just detaches T from its parent (*detach* semantics) or as an operation that invalidates every node of T (*erasure* semantics). The first choice needs to be supported by a garbage-collector, while the second may require the management of pointers to invalidated nodes. The insertion of a tree T below a node n may break the tree-structure of XML data, if the tree T has already a parent, and would create a cycle if n were a node of T. For this reason, the insertion operator typically copies T before inserting it below n. Node renaming must be defined with some care because it may break invariants connected to name spaces. Some proposals also include a *move T1 into T2* operation to move a subtree $T1$ to a different location without altering its identity. This operation cannot be encoded by inserting $T1$ into $T2$, and then deleting the original $T1$, because the insertion performs a copy, hence this encoding does not preserve the identity of $T1$.

Revalidation

Dynamic revalidation is, currently, the technique of choice, in order to ensure that modified data still respect type invariants. The complexity of revalidation depends on the language used to express structural invariants. For example, if a DTD is used, when a valid subtree rooted at an element with name q is moved from a position to another, it is only necessary to verify whether an element with name q may be found in that position. If XML Schema is used, then the content model of the moved subtree may depend on its position, hence the subtree has to be revalidated. In any case, a full revalidation of the updated structure is usually not needed, and many optimizations are generally possible. In some cases, static code analysis may avoid dynamic revalidation altogether. Work on this aspect is only at its beginning stages.

The Operators and the Full Language

The presence of update operators in an XML language may heavily affect the possibility of optimization. Any important optimization aims at reducing the number of times an expression is evaluated. Such a reduction is typically possible if the value of that expression at a certain time is guaranteed to be equal to its value when it was last evaluated. Updates make this property very difficult to prove. This problem has been faced with two main approaches: separation of queries and updates, and delayed update application.

The separation approach is exemplified by [6,10,11]: these languages distinguish between *expressions* and *statements* and put strong limitations on the places where statements may appear. The proposals in [2,3] perform a "partial" separation of queries and updates. They use the same syntax for non-updating expressions and updating expressions, which means, for example, that standard FLWOR expressions are used to iterate both classes of expressions. However, the places where updating expressions may appear are severely limited: for example, in a FLWOR expression, they can only appear in the *return* clause. The languages XQuery! and LiXQuery$^+$ [8,9], instead, have just one syntactic category (*expressions*) and no syntactic limitation on where the updates may appear.

The delayed-application approach is based on the definition of a *snapshot scope*; all the update expressions, or statements, in the scope are not executed immediately, but only when the scope is closed. This ensures that, inside the scope, the value of an expression is not affected by updates. For example, the proposals of [3,10,11] define a whole-query snapshot scope, while in XQueryP [2] every update is applied immediately. Finally, in XQuery! and LiXQuery$^+$ [8,9], the programmer has a full control of the snaphot scope.

Apart from optimization, the delayed approach is also proposed for consistency reasons. The partial execution of a set of correlated updates would create consistency problems. These problems can be avoided by collecting these updates in a scope, and by imposing that all the updates in each snapshot scope are evaluated atomically.

Key Applications

Updates are unavoidable in any language that manipulates XML persistent data, as the one proposed in [11], and are very useful in any language for scripting purposes, as XQueryP [2]. Application scenarios are detailed in [4] and in [5].

Future Directions

The most important open problems, in this field, are optimization and static analysis. Some work on the optimization of queries with updates has been done (see [1,7], for example), but the field is huge. Static analysis is also an area where lot of work has to be done. Here, a crucial issue is the design of algorithms to substitute dynamic post-update revalidation with some form of static type-checking.

Experimental Results

Some of the proposals have a public implementation that has been use to experiment with the semantics and the performance of the language; for example, XL has a demo reachable from [13], and XQuery! has a demo reachable from [15]. The W3C maintains a list of XQuery implementations in [12].

Cross-references

- ▶ XML
- ▶ XML Algebra
- ▶ XML Query Processing
- ▶ XML Type Checking
- ▶ XPath/XQuery

Recommended Reading

1. Benedikt M., Bonifati A., Flesca S., and Vyas A. Adding updates to XQuery: Semantics, optimization, and static analysis. In Proc. 2nd Int. Workshop on XQuery Implementation, Experience and Perspectives, 2005.
2. Carey M., Chamberlin D., Fernandez M., Florescu D., Ghelli G., Kossmann D., Robie J., and Siméon J. XQueryP: an XML application development language. In Proc. of XML, 2006.
3. Chamberlin D., Florescu D., and Robie J. XQuery update facility. W3C Working Draft, July 2006.
4. Chamberlin D. and Robie J. XQuery update facility requirements. W3C Working Draft, June 2005. http://www.w3.org/TR/xquery-update-requirements/.
5. Engovatov D., Florescu D., and Ghelli G. XQuery scripting extension 1.0 requirements. W3C Working Draft, June 2007. http://www.w3.org/TR/xquery-sx-10-requirements.
6. Florescu D., Grünhagen A., and Kossmann D. XL: an XML programming language for Web service specification and composition. In Proc. 11th Int. World Wide Web Conference, 2002, pp. 65–76.
7. Ghelli G., Onose N., Rose K., and Siméon J. XML query optimization in the presence of side effects. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2008, pp. 339–352.

8. Ghelli G., Ré C., and Siméon J. XQuery!: an XML query language with side effects. In Proc. 2nd Int. Workshop on Database Technologies for Handling XML Information on the Web, 2006, pp. 178–191.
9. Hidders J., Paredaens J., Vercammen R., and Demeyer S. On the expressive power of XQuery-based update languages. In Database and XML Technologies, 5th Int. XML Database Symp., 2006, pp. 92–106.
10. Sur G.M., Hammer J., and Siméon J. An XQuery-based language for processing updates in XML. In Proc. Programming Language Technologies for XML(PLAN-X), 2004.
11. Tatarinov I., Ives Z., Halevy A., and Weld D. Updating XML. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001, pp. 413–424.
12. W3C. W3C XQuery site. http://www.w3.org/XML/Query.
13. XL team. XL site. http://xl.inf.ethz.ch.
14. XQuery 1.0 and XPath 2.0 data model (XDM). W3C Recommendation, January 2007.
15. XQuery! team. XQuery! site. http://xquerybang.cs.washington.edu.

XML Views

Murali Mani
Worcester Polytechnic, Institute, Worcester, MA, USA

Synonyms

XML publishing

Definition

Database applications provide an XML view of their data so that the data is available to other applications, especially web applications. Database systems provide support for the client applications to use (query and/or manipulate) the data. The operations specified by the client applications are composed with the view definitions by the database system, thus performing these actions. The internal data model used by the database application, as well as how the operations are performed are transparent to the client applications; they see only an XML view of the entire system. XML views help the database systems to maintain their legacy data, as well as utilize the optimization features present in legacy systems (especially SQL engines), and at the same time make the data accessible to a wide range of web applications.

Historical Background

Views (external schema) are a feature [12] present universally in almost all database systems. Views provide data independence as well as the ability to control access of portions of data to different classes of users. With XML [2] becoming the standard for information exchange over the web since 1998, database applications have used XML views to publish their data and to make the data accessible to web applications. Nowadays, XML views are supported by most major database engines like Microsoft SQL server, Oracle, and IBM DB2.

Foundations

The views that database systems support can either be virtual or materialized [12]. When the view is virtual, only the view definition is stored in the system. Whenever a client application accesses the data by issuing a query over the view, the database system composes the user query with the view definition and this combined query is executed over the underlying data. The advantage of virtual views are that the data is never out-of-date as the data is stored, accessed from and manipulated in only one place. Materialized views on the other hand store the data for the view along with the view definition. Therefore the same data is now in more than one location, and thus the different copies of the data need to be kept consistent by maintaining the materialized views whenever the underlying data changes. Incremental and efficient maintenance of materialized views have been studied [6,9,11] and are supported by most commercial SQL engines. The advantage of materialized views is that the user query can potentially be answered faster as the user query can be directly answered from the materialized view, and does not require composing it with the view definition. In this article, both virtual and materialized XML views that are defined primarily over relational data are considered, and state-of-the-art techniques, and open problems for these are described.

Mapping Between the XML View and the Underlying Data

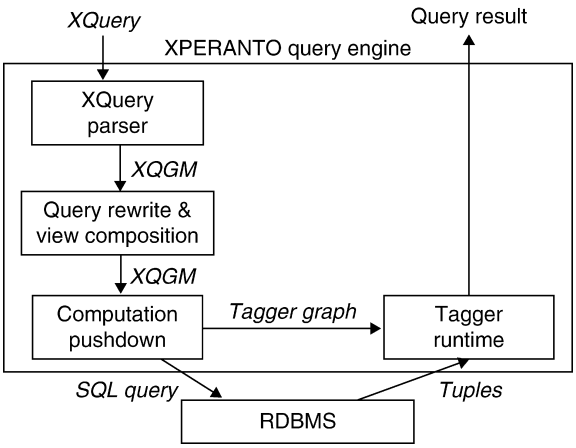
The mapping between the XML view and the underlying data are used for publishing the data, as well as for performing the user requested actions (such as answering queries) when the view is virtual. Different ways of specifying mappings between the XML view and the underlying data are possible. The *canonical mapping* [13] is a very simple one where there is a 1-1 mapping between the relational tuples and the XML elements in the view. An XML element is constructed for every row in every table in the relational database. Thus the entire

data in the relational database is captured in this canonical XML view. Slightly more complex mappings that still capture the entire relational data in the XML view are studied in [8], where the key-foreign key constraints are used to nest XML elements within each other. The translation of queries (especially navigation queries as in XPath) in the above mapping schemes is quite straight forward.

However, often times, a database application needs to publish their data as an XML view that conforms to a standard schema. In such cases, a more complex mapping scheme is needed as all of the underlying data may not be exposed in the view; also the underlying data might need to be restructured to conform to the schema. In [10], the authors study how the user can specify relationships between the underlying schema and the view schema diagrammatically. Based on some assumptions, the system then analyzes the user specified relationships and translates them into meaningful mappings that can be understood by the system (such as SQL queries). Also the user is consulted when there is potential ambiguity. In [5,13], the mapping is specified using XQuery language [15]. This is similar to the scenario that is well-understood by SQL engines (where view definitions are specified in SQL). The database administrator can specify the XML view using an XQuery expression that conforms to the required schema.

User Queries over XML Views

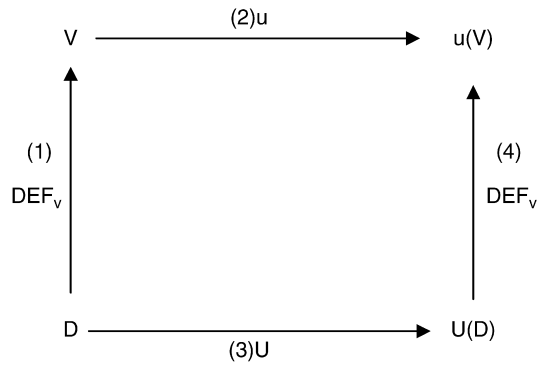
The systems that support XML views need to provide the capability for users to query the data. In [5,13], the authors study how the user queries specified using XQuery can be answered efficiently in the scenario where the XML view is also specified using XQuery. The architecture for such a system as described in [13] is shown in Fig. 1. One of the main assumptions made by these systems is that the SQL engine is best equipped to handle computations efficiently (those computations that an SQL engine can handle); therefore one needs to push down as much computation as possible within the SQL engine. The view composer shown in Fig. 1 composes the user query with the view query. The computation pushdown module then rearranges the combined query plan so that the SQL portion is at the bottom of the query plan, and it includes everything that can be done by the SQL engine. In such a case, the middle-ware only needs to do tagging and this is done in a single pass over the data returned by the SQL engine.



XML Views. Figure 1. Typical architecture for processing user queries over XML views.

User Updates over XML Views

While a lot of work has focussed on how to answer user queries over XML views, very little work has focussed on handling user updates over virtual XML views. As the view is virtual, the view update needs to be performed by updating the base data in such a way that the effect expected by the user is achieved. A common semantics used for view updates is the side-effect free semantics shown in Fig. 2, as described in [3,7]. In the figure, D represents the database instance, DEF_v represents the view definition, V is the view instance, u is the user specified view update. u(V) represents the effect that the user wants to achieve on the view, the view update problem therefore is to find an update U over the base data such that the user desired effect is



XML Views. Figure 2. Illustrating side-effect free semantics.

achieved on the view. It is possible that such an update U, does not exist in which case the view update cannot be performed. In some cases, there could be a unique U, and in other cases, it is possible that multiple such updates over the base data, exist in which case the ambiguity needs to be resolved using heuristics, by the user, or by rejecting the view update.

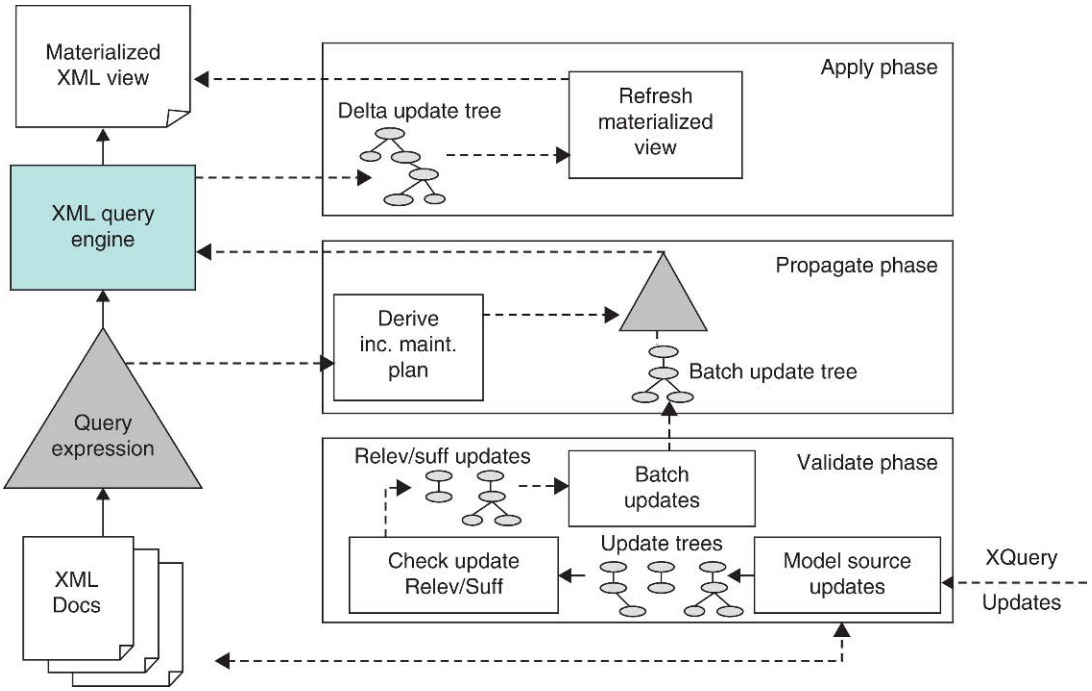
Updating SQL views itself is considered a hard problem, and the existing solutions handle only a subset of the view definitions. When a user specifies updates over view definitions that use "non-permissible" operators (such as aggregation operators), the system rejects these updates. Most solutions, including commercial ones, use a schema level analysis to perform/reject the view update, where they utilize the base schema, the view definition and the user specified update statement. Some solutions also examine the base data, in which case fewer view updates need to be rejected.

The main approaches that study updates over XML views of relational databases include [1,14]. In [1], the authors translate the XML view into a set of SQL views; now the solutions for SQL views can be utilized. In [14], the authors identify that the XML view update problem is harder than the SQL view update problem – SQL view update problem boils down to the case where the XML view schema has only one node. For a general XML view, given an update (such as delete) to be performed on an XML view element, the authors partition the XML view schema nodes into three categories – for one of the categories, the authors utilize the results from the SQL view update research, for the other two categories, the authors propose new approaches to check for side-effects. As follow up work to [14], the authors have studied how data level analysis can be used for the XML view update problem as well.

Maintenance of Materialized XML Views

In order to keep materialized views consistent with the underlying base data, one approach is to recompute the view every time there is a base update. However, this approach is not efficient as the base updates are typically very small compared to the entire base data. It would be efficient if *incremental view maintenance* could instead be performed, where only the update to the view is computed. Incremental view maintenance consists of typically two steps – in the *propagate* phase, the delta change to the view is computed using a *incremental maintenance plan*, and in the *apply* phase, the view is refreshed using this delta change to the view.

Incremental maintenance of XML views is more complex than maintenance of SQL views, because of



XML Views. Figure 3. Architecture for maintaining materialized XML views.

the more complex features of XML including nested structure (a form of aggregation) and order, and of XML query languages such as XQuery. In [4], the authors study how to incrementally maintain XML views defined over underlying data sources that are also XML (note that the solutions apply to the case where the underlying data sources are relational as well). The architecture of their approach is shown in Fig. 3. As the underlying base is XML, the base updates can come in many different granularities – this requires a *validate phase* which combines the update with the base data to make it a complete update. This is then fed to the incremental maintenance plan in the *propagate phase*, which computes the delta change to the view. In the *apply phase*, the view is refreshed using the delta change to the view.

Key Applications

XML views are useful to any database application that wishes to publish their data on the web.

Future Directions

XML views are already being used widely by database applications, and their usage is expected to increase further in future. There are still several issues that need to be studied to make such systems more efficient. For processing queries, query composition will result in several unnecessary joins. Therefore the query optimizer must be able to remove unnecessary joins, this requires the query optimizer to be able to infer key constraints in the query plan. For processing view updates, a combined schema and data analysis promises to be an efficient approach and needs to be investigated. Incremental view maintenance further requires new efficient approaches for handling operations (such as aggregation) present in XML query languages.

Cross-references

- ▶ Top-k XML Query Processing
- ▶ XML Information Integration
- ▶ XML Publishing
- ▶ XML Updates
- ▶ XPath/XQuery

Recommended Reading

1. Braganholo V.P., Davidson S.B., and Heuser C.A. From XML view updates to relational view updates: Old solutions to a new problem. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004, pp. 276–287.
2. Bray T., Paoli J., Sperberg-McQueen C.M., Maler E., and Yergeau F. Extensible Markup Language (XML) 1.0, W3C Recommendation. Available at: http://www.w3.org/XML
3. Dayal U. and Bernstein P.A. On the correct translation of update operations on relational views. ACM Trans. Database Syst., 7(3):381–416, September 1982.
4. El-Sayed M., Rundensteiner E.A., and Mani M. Incremental maintenance of materialized XQuery views. In Proc. 22nd Int. Conf. on Data Engineering, 2006, p. 129.
5. Fernandez M., Kadiyska Y., Suciu D., Morishima A., and Tan W.-C. SilkRoute: A framework for publishing relational data in XML. ACM Trans. Database Syst., 27(4):438–493, December 2002.
6. Griffin T. and Libkin L. Incremental maintenance of views with duplicates. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1995, pp. 328–339.
7. Keller A.M. Algorithms for translating view updates to database updates for views involving selections, projections and joins. In Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1985, pp. 154–163.
8. Lee D., Mani M., Chiu F., and Chu W.W. NeT & CoT: Translating relational schemas to XML schemas using semantic constraints. In Proc. Int. Conf. on Information and Knowledge Management, 2002, pp. 282–290.
9. Palpanas T., Sidle R., Cochrane R., and Pirahesh H. Incremental maintenance for non-distributive aggregate functions. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 802–813.
10. Popa L., Velegrakis Y., Miller R.J., Hernandez M.A., and Fagin R. Translating Web data. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 598–60.
11. Quass D. Maintenance expressions for views with aggregates. In Proc. workshop on Materialized Views: Techniques and Applications, 1996, pp. 110–118.
12. Ramakrishnan R. and Gehrke J. Database Management Systems. McGraw Hill, 2002.
13. Shanmugasundaram J., Kiernan J., Shekita E., Fan C., and Funderburk J. Querying XML views of relational data. In Proc. 27th Int. Conf. on Very Large Data Bases, 2001, pp. 261–27.
14. Wang L., Rundensteiner E.A., and Mani M. Updating XML views published over relational databases: Towards the existence of a correct update mapping. Doc. Knowl. Eng. J., 58 (3):263–298, 2006.
15. W3C XQuery Working Group. Available at: http://www.w3.org/XML/Query/

XPath/XQuery

Jan Hidders, Jan Paredaens
University of Antwerp, Antwerpen, Belgium

Synonyms

W3C XML path language; W3C XML query language

Definition

XPath (XML path language) and XQuery (XML query language) are query languages defined by the W3C (World Wide Web Consortium) for querying XML documents.

XPath is a language based on path expressions that allows the selection of parts of a given XML document. In addition it also allows some minor computations resulting in values such as strings, numbers or booleans. The semantics of the language is based on a representation of the information content of an XML document as an ordered tree. An XPath expression consist usually of a series of steps that each navigate through this tree in a certain direction and select the nodes in that direction that satisfy certain properties.

XQuery is a declarative, statically typed query language for querying collections of XML documents such as the World Wide Web, a file system or a database. It is based on the same interpretation of XML documents as XPath, and includes XPath as a sublanguage, but adds the possibility to query multiple documents in a collection of XML documents and combine the results into completely new XML fragments.

Historical Background

The development of XPath and XQuery as W3C standards is briefly described in the following.

XPath 1.0 XPath started as an initiative when the W3C XSL working group and the W3C XML Linking working group (which was working on XLink and XPointer) realized that they both needed a language for matching patterns in XML documents. They decided to develop a common language and jointly published a first working draft for XPath 1.0 in July 1999, which resulted in the recommendation [8] in November 1999.

XQuery 1.0 and XPath 2.0 The history of XQuery starts in December 1998 with the organization by W3C of the workshop *QL '98* on XML query languages. This workshop received a lot of attention from the XML, database, and full-text search communities, and as a result the W3C started the XML Query working group [7]. Initially, its goals were only to develop a query language, and a working draft of the requirements was published in January 2000 and a first working draft for XQuery in February 2001. More than 3 years later, in August 2004 the charter of the group was extended with the goal of codeveloping XPath 2.0 with the XSL working group. Finally, in January 2007 the recommendations for both XQuery 1.0 [10] and XPath 2.0 [9] were published.

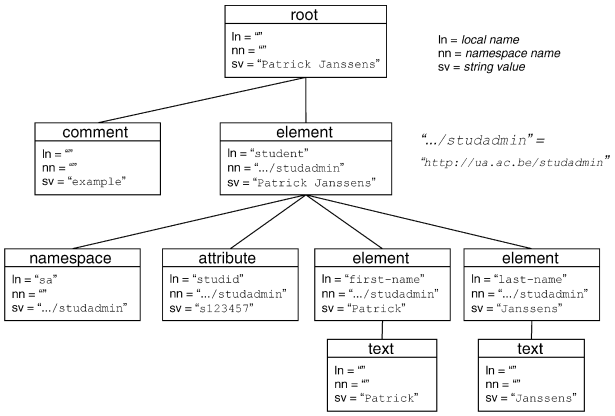
Historical Roots of XQuery 1.0 Most features of XQuery can be traced back to its immediate predecessor Quilt [2]. This language combined ideas from other predecessors such as XPath 1.0 and XQL from which the concept of path expressions was taken. From XML-QL came the idea of using variable bindings in the construction of new values. Older influences where SQL whose SELECT-FROM-WHERE expressions formed the inspiration for the FLWOR expressions, and OQL which showed the benefit of a functional and fully orthogonal query language. Finally there were also influences from other query languages for semi-structured data such as Lorel and YATL.

Foundations

The organization of this section is as follows. First, the XPath 1.0 data model is presented, then the XPath 1.0 language, which is followed by a description of XQuery 1.0 and finally XPath 2.0 is briefly discussed.

The XPath 1.0 Data Model The information content of an XML document is represented by an ordered tree that can contain seven types of nodes: *root* nodes, *element* nodes, *attribute* nodes, *text* nodes, *comment* nodes, *processing instruction* nodes and *namespace* nodes. Three properties can be associated with these nodes: a *local name*, a *namespace name* and a *string-value*. The local name is defined for element, attribute, processing instruction and namespace nodes. For the latter two it represents the target for the processing instruction and the prefix for the namespace, respectively. The namespace name represents the full namespace URI of a node and is defined for the element and attribute nodes. The string-value is defined for all types of nodes but for the root and element node it is derived from the string-values of the nodes under it. For attribute nodes it is the normalized attribute value, for text nodes it is the text content of the node, for processing instruction nodes it is the processing instruction data, for comment nodes it is the text of the comment and for namespace nodes it is the absolute URI for the namespace. An example of an XML document and its corresponding data model is given in Fig. 1. Over the nodes in this tree a *document*

```
<?xml version="1.0"?>
<!-- example -->
<sa:student
     xmlns:sa="http://ua.ac.be/studadmin"
     sa:studid="s1234567" >
   <sa:first-name>Patrick</sa:first-name>
   <sa:last-name>Janssens</sa:last-name>
</sa:student>
```



XPath/XQuery. Figure 1. An XML document and its XPath 1.0 data model.

order is defined that orders the nodes as they are encountered in a pre-order walk of the tree.

The XPath 1.0 Language The most important type of expression in XPath is the *location path* which is a path expression that selects a set of nodes from the tree describing the document. A location path can be relative, in which case it starts navigating from a certain context node, or absolute, in which case it starts from the root node of the document.

In its simplest form a relative location path consist of a number of steps separated by `/` such as for example `class/student/@id` where `class` and `student` are steps that navigate to children element nodes with those names, and `@id` navigates to attribute nodes with name `id`. It selects all nodes that can be reached from the context node by these steps, i.e., the `id` attribute nodes that are directly below `student` element nodes that are in turn children of `class` element nodes that are children of the context node. The path expression becomes an absolute location path, i.e., it starts navigation from the root node, if it is started with `/` as in `/class/student/@id`. The wildcards `*` and `@*` can be used to navigate to element nodes and attribute nodes, respectively, with any name.

The steps can also be separated by `//` which means that the path expression navigates to the current node and all its descendants before it applies the next step. So `class//first-name` navigates to all `first-name` elements that are directly or indirectly below a `class` element directly below the context node. A special step is the self step which is denoted as `.` and remains in the same place, so `class/.` returns the `class` element node and all the nodes below it. Another special step is the parent step which is denoted as `..` and navigates to the parent of the current context node. A location path can also start with `//` which means that it is an absolute location path whose first step navigates to the root node and all the nodes below it. For example, `//student` will select all `student` element nodes in the document. Path expressions can also be combined with the union operator, denoted as `|`, which takes the union of the results of two path expressions. For example, `//(student | teacher)/(last-name | first-name)` returns the first and last names of all students and teachers.

With each step zero or more predicates can be specified that must hold in order for nodes to be selected. Such predicates are indicated in square brackets such as for example in `//student[@id=''s123456'']/grade` which select the `grade` elements below `student` elements with a certain `id` attribute value. Conditions that can be specified include (i) comparisons between the string-values of the results of path expressions and other expressions, (ii) existential predicates that check whether a certain path expression returns a nonempty result and (iii) positional predicates that check the position of the current node in the result of the step for the context node. Comparisons between path expressions that return more than one node have an existential semantics, i.e., the comparison as assumed to hold if at least one of the returned nodes satisfies the equation. For example, `//course[enrolled/student/last-name=''Janssen'']` returns the courses in which at least one student with the last name "Janssen" enrolled. Available comparison operators include `=`, `!=` (not equals), `<` and `<=`. An example of an existential

predicate is `//course[enrolled/student]` that selects courses in which at least one student enrolled. Finally, an example of a positional predicate is given in `//course/teacher[1]/name` that selects for each course the name of the first teacher of that course. If there are multiple predicates then a positional predicate takes the preceding predicates into account. For example, `//course[subject=''databases''][1]` selects the first of the courses that have databases as their subject. The comparisons and existential conditions can be combined with `and` and `or` operations, and the `not()` function.

For path expressions that have to navigate over more types of nodes and require other navigation steps a more general syntax is available. In this syntax a single step is specified as `axis::node-test` followed by zero or more predicates. Here `axis` specifies the direction of navigation and `node-test` a test on the name or the type of the node that has to be satisfied. There are 13 possible navigation axes: `child`, `attribute`, `descendant`, `descendant-or-self`, `parent`, `ancestor`, `ancestor-or-self`, `following`, `preceding`, `following-sibling`, `preceding-sibling` `self` and `namespace`. The `following` axis navigates to all nodes that are larger in document order but not descendants, and the `following-sibling` axis navigates to all siblings that are larger in document order. Possible node-tests are names, the `*` wild-card and tests for specific node types such as `comment()`, `text()`, `processing-instruction()` or `node()` which matches all nodes. Finally two special functions `position()` and `last()` can be used in predicates and denote the position of the current node in the result of the step and the size of that result, respectively. To illustrate, the location path `/class//student[1]` can also be written in this syntax as `/child::class/descendant-or-self::node()/child::student[position()=1]`.

Next to the operators for navigation XPath also has functions for value manipulation of node sets, strings, booleans and numbers. This includes the arithmetic operators `+`, `*`, `-`, `div` and `mod`, aggregation operators such as `count()` and `sum()`, string manipulation such as string concatenation, space normalization, substring manipulation, etc., type conversion operators such as `string()`, `number()` and `boolean()`, and finally functions that retrieve properties of nodes such as `name()`, `string()` (also overloaded for type conversion) and `namespace-uri()`.

XQuery 1.0 The data model for XML documents is for XQuery 1.0 largely the same as for XPath 1.0. The main changes are that the root of a document is represented by a *document node*, and that typed values are associated with nodes. The types of these typed values include the built-in types from XML Schema and are associates with all values computed in XQuery. However, for the sake of brevity these types are mostly ignored here.

Another important change is that the fundamental data structure is no longer sets of nodes but ordered sequences of nodes and atomic values as defined by XML Schema. All results of expressions are such sequences and single values such as `1` and `''mary''` are always interpreted as singleton sequences. Note that these sequences are flat, i.e., they cannot contain nested sequences. The concatenation of sequences s_1 and s_2 is denoted as s_1, s_2. So the expression `(''a,'' ''b'')` denotes in fact the concatenation of the singleton sequence `''a''` and the singleton sequence `''b.''` Therefore `(1, (2, 3))` is equivalent to `(1, 2, 3)`.

The path expressions as defined in XPath 1.0 are almost all included and have mostly the same semantics. An important difference is that they do not return a set of nodes but a sequence of nodes sorted in document order. Several extensions to path expressions are introduced. Next to the `union` operator that is equivalent with the `|` operator in XPath 1.0, there is also the `intersect` and `difference` operators that correspond to the set intersection and set difference. These also return their results sorted in document order. The aggregation functions such as `count()` and `sum()` are naturally generalized as `fn:count()` and `fn:sum()` for sequences, and new ones such as `fn:min()`, `fn:max()` and `fn:avg()` are added. Note that built-in functions in XQuery are prefixed with a namespace, usually `fn`. Next to the old value comparisons new types of comparisons are introduced, such as `is` and `is not` that compare the node identity of two nodes, and `<<` that checks precedence in document order. For navigating over references, the `fn:id()` function is introduced that given an identifier retrieves the element node with that identifier, and the function `fn:idref()` that given an identifier retrieves the element nodes that refer to this identifier.

The access to collections of XML documents is provided by the functions `fn:doc()` and `fn:collection()` that both expect as argument a string containing a URI. These functions retrieve the requested

XML fragments associated with this URI and, if this was not already done before, construct their data models and return a document node or a sequence of nodes that are the roots of the fragments. An example of their use would be `fn:doc(''courses.xml'')/course[@code=''DB201'']/enrolled/student` that retrieves the students enrolled in the course `DB201` from the file `courses.xml`.

The core expressions of XQuery are the FLWOR expressions which are illustrated by the following example:

```
for $s in fn:doc(students.xml)//student,
   $e  in  fn:doc(''enrollments.xml'')//
    enrollment
let $cn := fn:doc(''courses.xml'')//course
[@crs-code=e/@crs-code]/name
where $s/@stud-id = $e/@stud-id
order by $cn
return <enroll> {$s/name ,$cn} </enroll>
```

A FLWOR expression starts with one or more `for` and `let` clauses that each bind one or more variables (that always start with `$`). The `for` clause binds variables such that they iterate over the elements of the result sequence of an expression, and the `let` clause binds the variable to the entire sequence. This is followed by an optional `where` clause with a selection condition, an optional `order by` clause that specifies a list of sorting criteria and a `return` clause that contains an expression that constructs the result. In the example the `for` clause binds the variable `$s` such that it iterates over the student elements in the file `students.xml` in the order that they appear there. Then, for each of those, it binds the variables `$e` such that it iterates over all the enrollments in the file `enrollments.xml` in the order that they appear there. For every binding of the variables it evaluates the `let` clause where it binds `$cn` with the name of the course in the enrollment `$e`. Then it selects those combinations for which the condition in the `where` clause is true, i.e., if the student `$s` belongs to the enrollment `$e`. The resulting bindings are sorted by the `order by` clause on the course name in `$cn`. Finally, the `return` clause creates for each binding in the result of the preceding clause an `enroll` element that contains the `name` element of student `$s` and the `name` element in `$cn`. Note that if there had been no `order by` clause then the result would have been sorted in the order that the students are listed in `students.xml`.

An additional optional feature for `for` clauses is the `at` clause that binds an extra variable to the position of the current binding. For example, the expression `for $x at $p in (''a,'' ''b,'' ''c'') return ($p, $x)` returns the sequence `(1,''a,'' 2,''b,'' 3,''c'')`.

New nodes can be constructed in two ways. The first is the direct constructor which is demonstrated in the first FLWOR example and consists of literal XML with embedded XQuery expression between curly braces. For the literal XML the corresponding nodes are created and deep copies of the results of the expressions are inserted into that. This can be used to compute the content and the attribute values of the new node such as in `<airport code=''{$x/code}''>{$x/full-name}</airport>`. An alternative that also allows computation of the element name are the computed element constructors of the form `element{`e_1`}{`e_2`}` that construct a new element node with the name as computed by e_1 and the content, i.e., all nodes directly below it, deep copies of those computed by e_2. For all types of nodes, except namespace nodes, are such computed constructors available.

XQuery also adds conditional expressions such as an `if (`e_1`) then` e_2 `else` e_3, and type switches of the form `typeswitch(`e`) case` t_1 `return` e_1 `case` t_2 `return` e_2 ... `default return` e_d that returns the result of the first e_i such that the result of e matches type t_i or the result of e_d if none of the types match. It also has logical quantifiers such as `some` $\$v$ `in` e_1 `satisfies` e_2 and `every` $\$v$ `in` e_1 `satisfies` e_2.

In order to allow query optimization techniques for unordered data formats there is a function `fn:unordered()` that allows the user to indicate that the ordering of a result is of no importance. In addition, there is a global parameter called the `ordering mode` such that when declared as `unordered` it means that, informally stated, the path expressions may produce unordered results and FLWOR expressions without an `order by` clause may change the iteration order. This ordering mode can be reset locally for an expression e with the statements `unordered{`e`}` and `ordered{`e`}`.

A very powerful feature is the possibility to start a query with a list of possibly recursive function definitions. For example, `declare function countElem ($s){fn:count($s/self::element()) + fn:sum(for $e in $s/* return countElem($e))}` defines a function that counts the number of element

nodes in a fragment. This feature makes XQuery Turing complete and gives it the expressive power of a full-blown programming language. Finally, there is a wide range of predefined functions for the manipulation and conversion of atomic values as defined in XML Schema, and functions for sequences such as `fn:distinct-values()` that removes duplicate values, `fn:reverse()` that reverses a sequence and `fn:deep-equal()` that tests if two sequences are deep equal.

XPath 2.0 This version of XPath is based on the same data model as XQuery 1.0 and is semantically and syntactically a subset of XQuery 1.0. The main omissions are (i) user-defined functions, (ii) all clauses in FLWOR expressions except the `for` clause without `at` and the `return` clause, (iii) the node constructors and (iv) the `typeswitch` expression.

Key Applications

The XPath language is used in several W3C standards such as DOM (Level 3), XSL, XLink, XPointer, XML Schema and XForms, and also in ISO standards such as Schematron. Most programming languages that offer some form of support for XML manipulation also support XPath for identifying parts of XML fragments. This includes C/C++, Java, Perl, PHP, Python, Ruby, Schema and the .Net framework.

The usage of the XQuery language can be roughly categorized into three different but not completely disjoint areas, for which different types of implementations are available. The first is that of *standalone XML processing*, where the XML data that is to be queried and transformed consists of documents stored on a file system or the World Wide Web, and these data are processed by a standalone XQuery engine. The second area is that of *database XML processing* where the XML documents are stored in an XML-enabled DBMS that has an integrated XQuery engine. The final and third area is that of *XML-based data integration* where data from different XML and non-XML sources are integrated into an XML view that can be queried and transformed with XQuery.

For each of the mentioned areas the XQuery engine faces different challenges. For example, for database XML processing it must optimally use the data structures provided by the DBMS, which might have an XML-specific storage engine, a relational storage engine or a mixture of both. On the other hand, for data integration it may be more important to determine how to optimally combine data from streaming and non-streaming sources and how to recognize and delegate query processing tasks to the data sources that have those capabilities themselves. As a consequence, different XQuery engines are often specialized in one of the mentioned application areas.

Future Directions

Since XPath 2.0 and XQuery 1.0 have become W3C recommendations, the involved working groups have continued to work on several extensions of these languages:

The XQuery Update Facility This extension adds update operations to XQuery. It allows expressions such as

```
for $s in /inventory/clothes/shirt[@size =
"XXL"]
return do replace value of $s/@price with
$s/@price - 10
```

that combine the XQuery syntax with update operations such that multiple elements can be updated at once. A last call working draft for the XQuery Update facility was published by W3C in August 2007.

XQuery 1.1 and XPath 2.1 In March 2007 the XML Query working group published a first version of the XML Query 1.1 requirements, and they plan to produce with the XSL working group the requirements for XPath 2.1. The proposed extensions for XQuery include grouping on values, grouping on position, calling external functions that for example, invoke web services, adding explicit node references that can be used as content and can be dereferenced, and finally higher order functions.

XQuery 1.0 and XPath 2.0 Full-Text This extension adds constructs for doing full text searches on selections of documents, text-search scoring variables that can be used in FLWOR expressions, and full-text matching options that can be defined in the query prolog. A last call working draft was published in May 2007.

XQuery Scripting Extensions This adds imperative features such that the resulting language can be more readily used for tasks that would otherwise typically be accomplished using an imperative language with XML

capabilities. Proposed extensions include constructs for controlling the order of computation in FLWOR expressions, operations that cause side effects such as variable assignments, and operations that observe these side effects. A first working draft describing the requirements for this extension was published by the XML Query working group in March 2007.

Cross-references

- ▶ XML Information Integration
- ▶ XML Programming
- ▶ XML Storage
- ▶ XML Tuple Algebra
- ▶ XML Updates
- ▶ XML Views
- ▶ XQuery Full-Text
- ▶ XQuery Processors
- ▶ XSL/XSLT

Recommended Reading

1. Brundage M. XQuery: The XML Query Language. Pearson Higher Education, Addison-Wesley, Reading, MA, USA, 2004.
2. Chamberlin D.D., Robie J., and Florescu D. Quilt: An XML query language for heterogeneous data sources. In Proc. 3rd Int. Workshop on the World Wide Web and Databases, 2000, pp. 53–62.
3. Hidders J., Paredaens J., Vercammen R., and Demeyer S. A light but formal introduction to XQuery. In Database and XML Technologies, 2nd Int. XML Database Symp., 2004, pp. 5–20.
4. Katz H., Chamberlin D., Kay M., Wadler P., and Draper D. XQuery from the experts: A guide to the W3C XML query language. Addison-Wesley Longman, Boston, MA, USA, 2003.
5. Melton J. and Buxton S. Querying XML: XQuery, XPath, and SQL/XML in context. Morgan Kaufmann, San Francisco, CA, USA, 2006.
6. Walmsley P. XQuery. O'Reilly Media, 2007.
7. W3C. W3C XML query (XQuery). http://www.w3org/XML/Query/.
8. W3C. XML path language (XPath), version 1.0, W3C recommendation 16 November 1999. http://www.w3.org/TR/xpath/, November 1999.
9. W3C. XML path language (XPath) 2.0, W3C recommendation 23 January 2007. http://www.w3.org/TR/xpath20/ January 2007.
10. W3C. XQuery 1.0: An XML query language, W3C recommendation 23 January 2007. http://www.w3.org/TR/xquery/ January 2007.

XPDL

- ▶ Workflow Management Coalition
- ▶ XML Process Definition Language

XQFT

- ▶ XQuery Full-Text

XQuery 1.0 and XPath 2.0 Full-Text

- ▶ XQuery Full-Text

XQuery Compiler

- ▶ XQuery Processors

XQuery Full-Text

Chavdar Botev[1], Jayavel Shanmugasundaram[2]
[1]Yahoo Research!, Cornell University, Ithaca, NY, USA
[2]Yahoo Research!, Santa Clara, USA

Synonyms

XQuery 1.0 and XPath 2.0 Full-Text; XQFT

Definition

XQuery Full-Text [11] is a full-text search extension to the XQuery 1.0 [9] and XPath 2.0 [8] XML query languages. XQuery 1.0, XPath 2.0, and XQuery Full-Text are query languages developed by the World Wide Web Consortium (W3C).

Historical Background

The XQuery [9] and XPath languages [8] have evolved as powerful languages for querying XML documents. While these languages provide sophisticated structured query capabilities, they only provide rudimentary capabilities for querying the text (unstructured) parts of XML documents. In particular, the main full-text search predicate in these languages is the `fn:contains`

`($context, $keywords)` function (*http://www.w3.org/TR/xpath-functions/#func-contains*), which intuitively returns the Boolean value `true` if the items in the `$context` parameter contain the strings in the `$keywords` parameter. The `fn:contains` function is sufficient for simple sub-string matching but does not provide more complex search capabilities. For example, it cannot support queries like "*Find titles of books (//book/title) which include "xquery" and "full-text" within five words of each other, ignoring capitalization of letters.*" Furthermore, XQuery does not support the concept of relevance scoring, which is very important in the area of full-text search.

To address these short-comings, W3C has formulated the XQuery 1.0 and XPath 2.0 Full-Text 1.0 Requirements [12]. These requirements specify a number of features that must, should or may be supported by full-text search extensions to the XQuery and XPath languages. These include the level of integration with XQuery 1.0 and XPath 2.0, support for relevance scoring, and extensibility as the most important features of such extensions.

W3C has also described a number of use cases for full-text search within XML documents that occur frequently in practice. These use cases can be found in the document XQuery 1.0 and XPath 2.0 Full-Text 1.0 Use Cases [13]. The use cases contain a wide range of scenarios for full-text search that vary from simple word and phrase queries to complex queries that involve word proximity predicates, word ordering predicates, use of thesauri, stop words, stemming, etc.

The XQuery Full-Text language has been designed to meet both the XQuery 1.0 and XPath 2.0 Full-Text 1.0 Requirements and the XQuery 1.0 and XPath 2.0 Full-Text 1.0 Use Cases.

XQuery Full-Text is also related to previous work on full-text search languages for semi-structured data: ELIXIR [5], JuruXML [4], TEXQuery [3], TiX [1], XIRQL [6], and XXL [7].

Foundations

XQuery Full-Text provides a full range of query primitives (also known as *selections*) that facilitate the search within the textual content of XML documents. The textual content is represented as a series of *tokens* which are the basic units to be searched. Intuitively, a token is a character, n-gram, or sequence of characters. An ordered sequence of tokens that should occur in the document together is referred to as a *phrase*. The process of converting the textual content of XML documents to a sequence of tokens is known as *tokenization*.

XQuery Full-Text proposes four major features for support of full-text search the XQuery and XPath query languages:

1. Tight integration with the existing XQuery and XPath syntax and semantics
2. Query primitives for support of complex full-text search in XML documents
3. A formal model for representing the semantics of full-text search
4. Support for relevance scoring.

Each of these features is discussed below.

XQuery Full-Text Integration

One of the main design paradigms of XQuery Full-Text is the tight integration with the XQuery and XPath query languages. The integration is both on the syntax and data-model levels.

The syntax level integration is achieved through the introduction of a new XQuery expression, the *FTContainsExpr*. The *FTContainsExpr* acts as a regular XQuery expression and thus, it is fully composable with the rest of the XQuery and XPath expressions. The basic syntax of the *FTContainsExpr* is:

```
Expr ''ftcontains'' FTSelection
```

The XQuery expression *Expr* on the left-hand side is called the *search context*. It describes the nodes from the XML documents that need to be matched against the full-text search query described by the *FTSelection* on the right-hand side. Expr can be any XQuery/XPath expression that returns a sequence of nodes. The syntax of the *FTSelection* will be described later. The entire *FTContainsExpr* returns `true` if some node in *Expr* matches the full-text search query *FTSelection*. Note, that since *FTContainsExpr* returns results within the XQuery data model, *FTContainsExpr* can be arbitrary nested within XQuery expressions. Consider the following simple example.

```
//book[./title ftcontains ''information'' ftand ''retrieval'']//author
```

The above example returns the authors of books whose title contains the tokens "information" and "retrieval." The expression `./title` defines the search context for the *FTContainsExpr* in predicate expression

within the brackets [] which is the `title` child element node of the current `book` element node. The *FTSelection* "`information'' ftand ''retrieval''` specifies that book titles must contain the tokens "information" and "retrieval" to be considered a match.

Integration can be achieved not only for XQuery Full-Text expressions within XQuery/XPath expressions but vice versa. Consider the following example.

```
for $color in (''red,'' ''yellow'')
for $car in (''ferrari,''
''lamborghini,'' ''maserati'')
return //offer[. ftcontains
{$color, $car} phrase]
```

The above query returns `offer` element nodes which contain any of a number of combinations of colors and cars as a phrase. The *FTContainsExpr* will match phrases like "red lamborghini" or "yellow ferrari" but it will not match "red porsche" or "yellow rusty ferrari."

XQuery Full-Text Query Primitives

This section describes the basic XQuery Full-Text query primitives and how they can be combined to construct complex full-text search queries.

The XQuery Full-Text query primitives include token and phrase search, token ordering, token proximity, token scope, match cardinality, and Boolean combinations of the previous. Further, XQuery Full-Text allows control over the natural language used in the queried documents, the letter case in matched tokens, and the use of diacritics, stemming, thesauri, stop words, and regular-expression wildcards though the use of *match options*.

The XQuery Full-Text query primitives are highly composable within each other. This allows for the construction of complex full-text search queries. The remainder of this section will briefly describe some of the available query primitives and show how they can be composed.

The most basic query primitive is to match a sequence of tokens also known as *FTWords*. For example, the above car offer query can also be written as:

```
//offer[. ftcontains {''red ferrari,''
''yellow ferrari,''
''red lamborghini,'' ''yellow lambor-
ghini,'' ''red maserati,''
''yellow maserati''} any]
```

The above *FTContainsExpr* matches `offer` nodes which contain any of the listed phrases. The `any` option specifies that it is sufficient to match a single phrase within an `offer` node. It is also possible to use the option `all` which specifies that all phrases should be matched within a single node. As it was shown earlier, the `phrase` options specifies that all nested phrases should be combined into a single phrase. Other possible options are `any word` or `all word` which specify that the nested phrases need to be first broken into separate tokens before applying the respective disjunctive or conjunctive match semantics. For example, the expression

```
//offer[. ftcontains {''red ferrari,''
''yellow lamborghini''} all word]
```

is equivalent to

```
//offer[. ftcontains {''red,'' ''fer-
rari,'' ''yellow,'' ''lamborghini''}
all]
```

Two other basic query primitives are the ability to restrict the proximity of the matched tokens. There are two flavors of proximity predicates. The *FTWindow* primitive specifies that the matched tokens have to be within a window of a specified size. Consider the example

```
//offer[. ftcontains {''red ferrari,''
''yellow lamborghini''} all
```

window at least 6 words]

The above expression will return `offer` nodes which contain, say, "red ferrari and brand new yellow lamborghini" because both phrases occur within a window of seven words, i.e., the window matches the *FTWindow* size restriction of at least six words. The expression will not match nodes which contain "red ferrari and yellow lamborghini" because the phrases occur within a window of five words.

The other flavor of proximity primitive is *FTDistance*. It can be used to restrict the number of intervening tokens between the matched tokens or phrases. For example, consider the expression

```
//offer[. ftcontains {''new,'' ''brand,''
''ferrari''} all
```

distance at most 1 word]

The above expression will return `offer` nodes which contain the specified query tokens with at most one intervening token between consecutive

occurrences of the query tokens. For example, the expression will return nodes which contain "brand new red ferrari" because there are no intervening tokens between "brand" and "new" and one intervening token between "new" and "red." On the other hand, the expression will not return nodes which contain "new car of the ferrari brand" because there are three intervening tokens between "new" and "ferrari."

XQuery Full-Text allows also the specification of the order of the matched tokens using the *FTOrder* primitive.

```
//offer[. ftcontains {''new,'' ''brand,''
''ferrari''} all
```

distance at most 1 word ordered]

The above query expands the previous *FTDistance* example by specifying that the query tokens can occur in the nodes only in the specified order.

More complex query expressions can be built using *FTAnd*, *FTOr*, *FTUnaryNot*, and *FTMildNot*. They allow for the combination of other *FTSelections* into conjunctions, disjunctions, and negations. Here is a complex example using *FTAnd* and some of the *FTSelections* introduced earlier.

```
//offer[. ftcontains
(({''red,'' ''ferrari''} all
window at most 3 words)
ftand
({''yellow,'' ''lamborghini''} all win-
dow 3 words))
window at most 20 words]
```

The above expression specifies that the resulting `offer` nodes must contain "red" and "ferrari" within a window of 3 words, "yellow" and "lamborghini" within a window of 3 words, and all of them within a window of 20 words.

```
//offer[. ftcontains ''red ferrari''
ftnot ''yellow lamborghini'']
```

This example of *FTNot* will return `offer` nodes which contain the phrase "red ferrari" but not the phrase "yellow lamborghini." Sometimes, this kind of negation can be too strict. For example, consider a query that looks for articles about the country Mexico but not the state New Mexico.

```
//article[. ftcontains ''Mexico'' ftnot
''New Mexico'']
```

The above query will not return articles which talk about both the country Mexico and the state "New Mexico." The query can be rewritten using *FTMildNot*.

```
//article[. ftcontains ''Mexico'' not in
''New Mexico'']
```

Intuitively, the above query specifies that the user wants articles where the token "Mexico" has occurrences not part of the phrase New Mexico (although there are still be occurrences of the phrase New Mexico).

The power of XQuery Full-Text queries can be further increased with the use of match options that control the way tokens are matched within the document. For example, if the user wants to find offers containing "FERRARI" (all capital letters), she can use the "uppercase" match option:

```
//offer[. ftcontains
''ferrari'' uppercase]
```

This query will match only tokens "FERRARI" regardless of how the token is specified in the query.

Match options can be used to encompass several *FTSelections*. For example, consider the query

```
//article[. ftcontains (''car'' ftand
''aircraft'')
window at most 50 words
with thesaurus at ''http://acme.org/
thesauri/Synonyms"]
```

The above query searches for articles that contain the tokens "car" and "aircraft" or their synonyms using the thesaurus identified by the URI "http://acme.org/thesauri/Synonyms." The matched tokens have to be within a window of at most 50 words.

As mentioned in the beginning of this section, there are other *FTSelections* and match options provided by XQuery Full-Text. The description of all of these features goes beyond the scope of the article. The reader is referred to the complete specification of the language available at [11].

This section will finalize the brief overview of the language primitives in XQuery Full-Text with the description of the feature *extension points*. Extension points be used to provide additional implementation-defined full-text search functionality. An example of the use of such an extension can be found below.

```
declare namespace acmeimpl = ''http://
acme.org/AcmeXQFTImplementation;''
//book/author[name ftcontains (#
acmeimpl:use-entity-index #) {''IBM''}]
```

The above example shows the use of the extension `use-entity-index` provided by the ACME implementation of XQuery Full-Text. It directs the implementation to use an entity index for the token "IBM" so it can also match other references to the same entity, such as "International Business Machines" or even "Big Blue." If the above query is evaluated using another implementation the `use-entity-index` extension will be ignored.

XQuery Full-Text Formal Model

Amer-Yahia et al. [3] have shown that the XQuery 1.0 and XPath 2.0 Data Model (XDM) [10] is inadequate to support the composability of the complex full-text query primitives described in the previous section. Intuitively, XDM represents data only about entire XML nodes but for sub-node entities like the positions of the tokens within a node. This precludes, for example, the evaluation of nested proximity *FTSelectionsFTWindow* and/or *FTDistance* such as the ones used in the *FTAnd* example.

Therefore, XQuery Full-Text needs a more precise data model that can support the complex full-text search operations and their composability. The language specification [11] introduces the *AllMatches* model. The result of every *FTSelection* applied on a node from the search context can be represented as an *AllMatches* object within this model.

The *AllMatches* object has a hierarchical structure. Each *AllMatches* object consists of zero, one, or more *Match* objects. Intuitively, a *Match* object describes one set of positions of tokens in the node that match the corresponding *FTSelection*. The *AllMatches* contains all such possible *Match* objects. The positions of tokens in a match are described using *TokenInfo* objects which contain data about the relative position of the token in the node and to which query token it corresponds. This model is sufficient to describe the semantics of all *FTSelections*.

The semantics of every *FTSelection* can be represented as a function that takes as input one or more *AllMatches* objects from its nested *FTSelections* and produces an output *AllMatches* object. The remainder of this section will illustrate the process. The illustration will use some simplifications not to burden the exposition with excessive details. Nevertheless, the example below will illustrate some of the basic techniques in the workings of the model.

Consider the simple query

```
//offer[. ftcontains {''red,'' ''ferrari''} all window at most 3 words]
```

Let's assume that the token "red" occurs in the current node from the search context in relative positions 5 and 10 and the token "ferrari" occurs in relative positions 7 and 25. The *AllMatches* objects for each of these tokens contain two *Match* objects – one for each position of occurrence:

```
''red'': AllMatches { Match
{TokenInfo{token:''red,'' pos:5}},
Match{TokenInfo{token:''red,''
pos:10}}}
''ferrari'': AllMatches { Match{TokenInfo{token:''ferrari,'' pos:7}},
Match{TokenInfo{token:''red,''
pos:25}}}
```

To obtain the *AllMatches* for the *FTWords* `{''red,'' ''ferrari''} all`, each pair of positions for "red" and "ferrari" has to be combined into a single *Match* object.

```
(''red,'' ''ferrari'') all:
AllMatches { Match
{TokenInfo{token: ''red,'' pos:5}},
TokenInfo{token:''ferrari,'' pos:7}},
Match{TokenInfo{token:''red,''
pos:10},
TokenInfo{token:''ferrari,'' pos:7}},
Match{TokenInfo{token:''red,''
pos:5}},
TokenInfo{token:''red,'' pos:25}},
Match{TokenInfo{token:''red,''
pos:10}},
TokenInfo{token:''red,'' pos:25}}}
```

Finally, to obtain the *AllMatches* for the outer most *FTWindow*, those *Matches* which violate the window size condition have to be filtered out. Only the first *Match* satisfies it: the window sizes are 3, 4, 21, and 16 respectively. Therefore, the final *AllMatches* is:

```
(''red,'' ''ferrari'') all window at most
3 words:
AllMatches { Match{TokenInfo{token:
''red,'' pos:5}},
TokenInfo{token:''ferrari,'' pos:7}}}
```

The resulting *AllMatches* object is non-empty and therefore, the current node from the search context

satisfies the *FTSelection* and the *FTContainsExpr* should return `true`.

The semantic of every *FTSelection* can be defined in terms of similar operations on *AllMatches* objects. The description of all such operations goes beyond the scope of this article. The reader is referred to the language specification [11] for further details.

Relevance Scoring in XQuery Full-Text

One of the core features of full-text search is the ability to rank the results in the order of their decreasing relevance. Usually, the relevance of a result is represented using a number called a *score* of the result. Higher scores represent more relevant results.

To support scores and relevance ranking, XQuery Full-Text extends the XQuery language with the support for *score variables*. Score variables give access to the underlying scores of the results of the evaluation of an XQuery Full-Text expression. Score variables can be introduced as part of a for- or let-clause in a FLWOR expression. Here is a simple example

```
for $o score $s in //offer[. ftcontains
{''red,'' ''ferrari''}
all window at most 3 words]
order by $s descending
return <offer id=''
{$o/@id}'' score=''{$s}'' />
```

The above is a fairly typical query iterates over all offers that contain "red" and "ferrari" within a window of three words using the `$o` variable, binds the score of each offer to the `$s` variable, and uses those variables to return the id of offers and their scores in order of decreasing relevance.

Score variables can also be bound in let clauses. This allows for the use of a scoring condition that is different than the one uses for filtering the objects. Consider the following example.

```
for $res at $p in (
for $o in //offer[. ftcontains {''red,''
''ferrari''}all ]
let score $s := $o ftcontains {''excel-
lent,'' ''condition''}
all window at most 10 words
order by $s descending
return $o)
where $p <= 10
return $res
```

The above query obtains the top 10 offers for red Ferrari's whose relevance is estimated using the full-text search condition `''excellent condition'' all window at most 10 words`.

It should be noted that XQuery Full-Text does not specify how scoring should be implemented. Each XQuery Full-Text implementation can use a scoring method of their choice as long the generated scores are in the range [0,1] and higher scores denote greater relevance.

Key Applications

Support for full-text search in XML documents.

Cross-references

- ▶ Full-Text Search
- ▶ Information Retrieval
- ▶ XML
- ▶ XPath/XQuery

Recommended Reading

1. Al-Khalifa S., Yu C., and Jagadish H. Querying structured text in an XML database. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 4–15.
2. Amer-Yahia S., Botev C., Doerre J., and Shanmugasundaram J. XQuery full-text extensions explained. IBM Syst. J., 45 (2):335–351, 2006.
3. Amer-Yahia S., Botev C., and Shanmugasundaram J. TE XQuery: A full-text search extension to XQuery. In Proc. 12th Int. World Wide Web Conference, 2004, pp. 583–594.
4. Carmel D., Maarek Y., Mandelbrod M., Mass Y., and Soffer A. Searching XML documents via XML fragments. In Proc. 26th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 2003, pp. 151–158.
5. Chinenyanga T. and Kushmerick N. Expressive and efficient ranked querying of XML data. In Proc. 4th Int. Workshop on the World Wide Web and Databases, 2001, pp. 1–6.
6. Fuhr N. and Grossjohann K. XIRQL: An extension of XQL for information retrieval. In Proc. ACM SIGIR Workshop on XML and Information Retrieval, 2000, pp. 172–180.
7. Theobald A. and Weikum G. The index-based XXL search engine for querying XML data with relevance ranking. In Advances 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

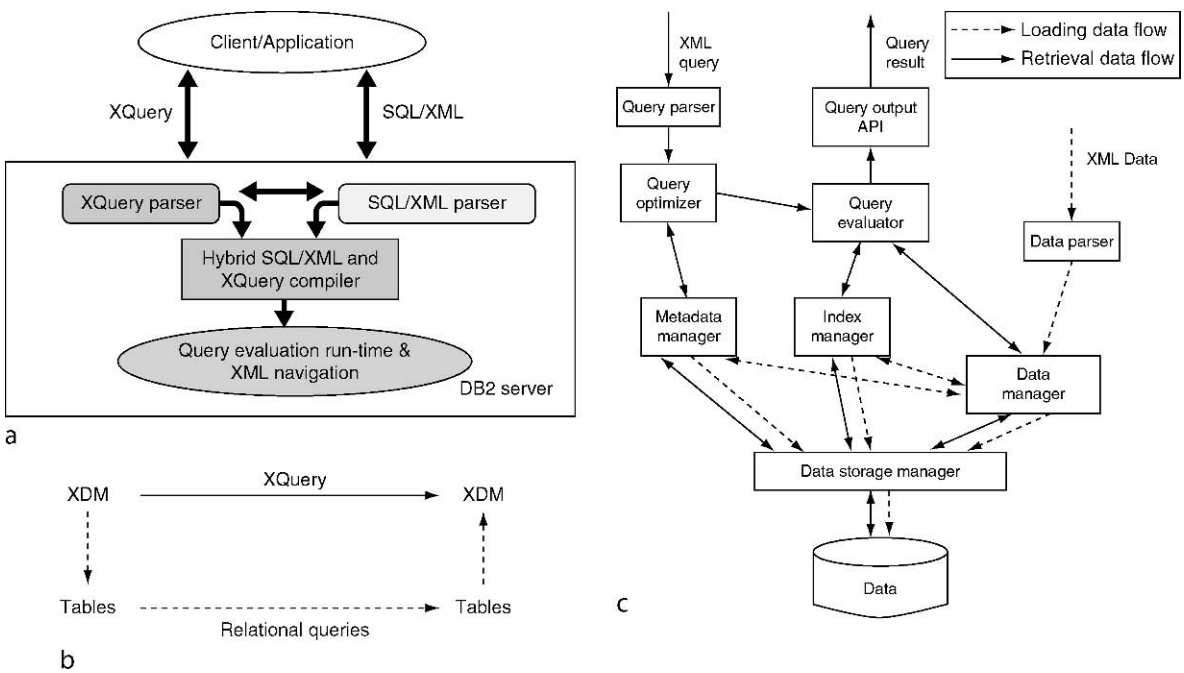
The *Pathfinder* XQuery compiler has been developed under the main hypothesis that the well-understood relational database kernels also make for efficient XQuery processors. Such a relational account of XQuery processing can indeed yield scalable XQuery implementations – provided that the system exploits 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 generated plan to derive a series of operator annotations (keys, multi-valued dependencies, *etc.*) that drive plan simplification and reshaping. A number of



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].

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

- ▶ Top-k XML Query Processing
- ▶ XML Benchmarks
- ▶ XML Indexing
- ▶ 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

relational queries on the XML representation of a relational database db. The database contains two relations R(a: int,b: int) and S(b: int, c: int). Each relation is represented by a single element containing a sub-element of element type t for each tuple:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<db>
<R>
    <t a="1" b="5" />
    <t a="2" b="5" />
    <t a="3" b="6" />
    <t a="4" b="7" />
</R>
<S>
    <t b="5" c="1" />
    <t b="6" c="3" />
    <t b="6" c="4" />
</S>
</db>
```

The first transformation is defined by the following two rules:

Rule 1:

```
<xsl:template match="/">
    <T>
        <xsl:apply-templates select="db/*/t" />
    </T>
</xsl:template>
```

Rule 2:

```
<xsl:template match="t">
    <xsl:copy-of select="." />
</xsl:template>
```

Both rules specify the class of nodes to which they can be applied by an absolute (starting from the document root ′/′) or a relative XPath expression. The first rule applies to the document root ′/′, whereas the second rule matches all document elements of type t (independently of their absolute position in the document).

The XSLT processing model is based on the recursive application of template rules to a *current node list*, which is initialized by the list containing only the document root. The transformation consists of binding each node to exactly one transformation rule (The existence of at least one matching rule is guaranteed by a default rule for each kind of document node.) and replacing it by the corresponding rule template. Templates contain static XML data and dynamic XSLT instructions which recursively generate new transformation requests (XSLT instructions, are distinguished from static fragments by using the XSLT namespace http://www.w3.org/1999/XSL/Transform which is generally (but not necessarily) bound to the prefix <<xsl:>>). The whole transformation process stops when all transformation requests have been executed.

The rule template of rule 1 generates an element of type T, which contains a new transformation request type xsl:apply-template for all tuples in the database returned by the XPath expression db/*/t. This expression is evaluated against the *context node* of the rule (the document root) and returns a new *current node list* containing all elements of type t in R and S (wildcard * matches all element names). Rule 2 matches all elements (tuples) of type t and creates a copy by instruction xsl:copy-of select=".". The final result is a relation T containing the *union* of R and S.

The previous example illustrates the usage of XPath for the dynamic selection of template rules and for recursively generating new current node lists for transformation. Rule patterns can simply match nodes by their element types, but they can also use more complex XPath patterns defining value-based and structural matching constraints. For example, in order to copy only tuples of R, one might replace rule 2 by the following two rules:

Rule 3:

```
<xsl:template match="R/t">
    <xsl:copy-of select="." />
</xsl:template>
```

Rule 4:

```
<xsl:template match="S/t" />
```

Rule 3 matches and copies sub-elements t of some element R, whereas rule 4 matches tuples of S without creating any node in the output tree. Observe that the same result can be achieved by keeping rule 2 and replacing rule 1 by the following rule:

Rule 5:

```
<xsl:template match="/">
    <T>
        <xsl:apply-templates select="db/R/t" />
    </T>
</xsl:template>
```

The following three rules simulate relational selection $\sigma_{b=5}(R)$ by copying only of tuples in R with attribute value b=5:

Rule 6:

```
<xsl:template match="R/t[@b=5]">
```

```
    <xsl:copy-of select="." />
  </xsl:template>
Rule 7:
  <xsl:template match="R/t" />
Rule 8:
  <xsl:template match="S/t" />
```

Rule 6 matches only tuples of `R` with attribute value `b` equal to 5, and creates a copy of this subset in the output tree. Rule 7 applies to *all* tuples of `R` without any other restriction. The resulting binding conflict is solved by a set of laws based on import precedence, priority attribute values and certain syntactical criteria, which generally choose the rule with the most specific matching criteria. For example, rule 7 applies to all nodes that can be transformed by rule 6, whereas the opposite is not true. The inclusion problem for XPath patterns has been formally shown to be PSPACE-complete [9], and the recommendation document defines an incomplete set of syntactical criteria that can easily be implemented and allowed to solve a large set of conflicts in practice. If the conflict resolution algorithm for template rules leaves more than one matching rule, the XSLT processor must generate a recoverable dynamic error where the optional recovery action is to select, from the matching template rules that are left, the one that occurs last in declaration order.

XSLT rules can be assigned to different computation *modes* by using an optional *mode* attribute. For example, the following three rules compute the natural join of R and S on attribute `b` by a simple nested loop. Rule 9 applies to all tuples of R in *default* mode and represents the outer loop of the join: XPath expression `/db/S/t[@b=current()/@b]` selects for each *current* tuple in R, all tuples in S with the same `b` attribute value. The inner loop is represented by rule 10 which is triggered by rule 9 and joins parameter `$tuple` with some tuple in S. The template of this rule creates a new tuple by copying all attributes element `$tuple` and all attributes of the current node (tuple). Attribute `mode` is used for distinguishing between the outer loop (mode `default`) and the inner loop (mode `join`). In order to "neutralize" the `default` transformation of S, rule 11 applies to S in default mode and generates nothing.

```
Rule 9:
  <xsl:template match="R/t">
    <xsl:apply-templates select="/db/S/t[@b=current()/@b]" mode="join">
      <xsl:with-param name="tuple" select="." />
    </xsl:apply-templates>
  </xsl:template>
Rule 10:
  <xsl:template match="S/t" mode="join">
  <xsl:param name="tuple"> <t /> </xsl:param>
  <t>
    <xsl:copy-of select="$tuple/@*" />
    <xsl:copy-of select="@*" />
  </t>
  </xsl:template>
Rule 11:
  <xsl:template match="S/t" />
```

XSL-FO document typesetting

XSL-FO is a vocabulary of XML element types for defining the main typographic objects (chapters, pages, paragraphs, figures, etc.) and their properties (character set, indentation, justification, etc.) used for the paper-oriented typesetting of documents. The main structure of an XSL-FO document is defined by a document model (fo:layout-master), one or several page sequences (fo:page-sequence) composed of one or several flows (fo-flow) of blocks (fo-block). Each of these elements can be parametrized by a set of attribute values for configuring the layout of pages (page-height, margins), tables (column width and height, padding, etc.), lists (item label and distance) and blocks (font-family and size, text alignment etc.). The richness and complexity of the XSL-FO typesetting model is illustrated by the size of the recommendation document (400 pages), and the reader is invited to consult the W3C's XSL-FO tutorial for a detailed introduction.

Theoretical Foundations of XSLT

The most important theoretical aspects of XSL concern XSLT and its interaction with XPath. XSLT and XPath have been theoretically evaluated and compared to other XML query languages by using different theoretical frameworks. For example, the formal semantics of XSLT has been defined by a rewriting process starting from an empty ordered output document, a set of rewriting rules (the XSLT program) and an ordered labeled input tree with "pebbles" (the current node list) [6]. Based on this representation it was possible to show that, under certain constraints (no join on data values), given two XML schemas (regular tree languages) τ_1 and τ_2 and an XSLT stylesheet (k-pebble transducer) T, it is possible to type-check T with respect to its input

type τ_2: $\forall\ t \in \tau_1$:$T(t) \subseteq \tau_2$ is decidable. Similarly, [2] proposed a formal model for a subset of XSLT using tree-walking tree-transducers with registers and look-ahead, which can compute all unary monadic second-order (MSO) structural patterns. This expressiveness result provides an important theoretical foundation for XSLT, since MSO captures different robust formalisms like first order logics with set quantification, regular tree languages, query automata and finite-valued attribute grammars.

The computation model of XSLT can also be compared with that of *structural recursion* for semi-structured data [1]. Structural recursion is proposed by functional programming languages like CAML for the dynamic selection of function definitions by matching function parameter values with pattern signatures. For example, the following transformation function `f` removes all elements of type `E` and renames all elements of type `A` to `B`:

f(v)=v
f({})={}
f({'A' : t})={ 'B' : f(t)}
f({'E' : t})=f(t)

This function can be defined in XSLT as follows:

```
<xsl:template match='*'>
    <xsl:copy><xsl:apply-templates select='*'>
    </xsl:copy>
</xsl:template>
<xsl:template match='E' />
<xsl:template match='A'>
    <B><xsl:apply-templates select='.'></B>
</xsl:template>
```

There are two main differences between XSLT and structural recursion. First, XSLT is defined on trees, whereas structural recursion can be applied on arbitrary graphs. Second, all structural recursion programs are guaranteed to terminate, whereas it is easily possible to write *infinite* XSL transformations as it is shown in the following rule, which recursively generates for any element a new element containing the result of its own transformation:

```
<xsl:template match='*'>
    <a><xsl:apply-templates select='.'></a>
</xsl:template>
```

The result of this transformation rule is an infinite tree <a><a><a>....

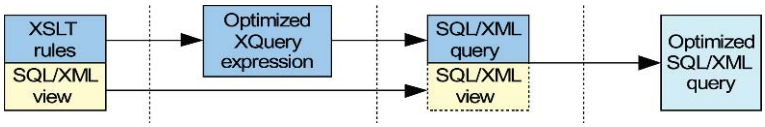
XSLT and XQuery

XSLT 2.0 is the result of the collaboration between the W3C working groups on XSL and XQuery. Both languages, XQuery 1.0 and XSLT 2.0 obtained recommendation status in January 2007 and share the same data model, type system and function library. The most notable change concerning the XSL recommendation is the evolution of XPath 1.0 to XPath 2.0 and the corresponding changes in the underlying data model. XPath 2.0 is the core language of XQuery 1.0 and integrates the type system of XML Schema with a rich library of built-in types, functions and operators. Every XSLT value is now a sequence and XPath 1.0 node-sets are replaced by node sequences. Variables can be bound to arbitrary sequences of values and trees, which simplifies certain programming issues observed in XSLT 1.0. Other new important features concern grouping of nodes, text matching functions, user defined functions and the possibility to generate multiple result trees by one transformation.

XSLT 2.0 and XQuery 1.0 are two declarative languages for querying and transforming XML document trees with similar expressive power. Both languages are Turing complete and [3] shows how to translate XSLT 2.0 programs into XQuery 1.0. However, the objectives of both languages are specific and their implementations are optimized for a particular usage. XSLT has been conceived for transforming individual documents into some fixed output format whereas XQuery is a query language that allows the retrieved relevant information retrieval of from large document collections. The dynamic binding of rule templates makes XSLT a flexible and modular language that facilitates code reuse and the development of large applications. In particular, it is possible to *import* XSLT template rule collections defined for external document fragments that are integrated into a unique XML document. On the other hand, modern XQuery engines are able to efficiently query large collections of documents using advanced query optimization techniques based on static type analysis.

Optimizing XSLT

XML is used in many applications as a publishing and exchange format for structured data. In this kind of application, XSLT serves for publishing XML query results according to the different usages and clients. Each data exchange might need one or several XSLT transformations on the data producer and the data



XSL/XSLT. Figure 1. XSLT optimization in XML-enabled RDBMS.

consumer side, and optimizing the transformation process becomes an important issue.

[7] proposes an XSLT optimization approach based on algorithms for optimizing the template rule selection process and for rewriting algebraic translations of XSLT stylesheets. XSLT optimization is revealed to be a hard problem because of two undecidability results concerning rule conflict detection at compile time, and finding optimal evaluation plans under any reasonable cost function. [5] study the problem of XSLT processing and optimization in an XML-enabled relational database system (RDBMS). The whole process is illustrated in Fig. 1.

The optimization process starts from a collection of XSLT transformation rules, which are applied to a XML data generated by a SQL/XML query on top of a relational database. The basic idea is to translate the XSLT rules into a SQL/XML query on a SQL/XML view, which can then be optimized by a standard relational query optimizer. The translation step generates an intermediate XQuery expression, which is optimized by exploiting structural schema information obtained from the underlying RDBMS.

Key Applications

The possibility to define transformations from any XML document structure A to any other structure B makes XSLT a powerful and generic tool in many XML applications. The main key applications are information publishing, data exchange and model transformation.

Information Publishing

XSL was initially designed for the publishing of XML documents, where document transformation consists of transforming any document into some specific format for screen display and printing. The final document format can be conformed to XML like XHTML, SMIL, DocBook or WML, but it also can be non XML-compliant like HTML (which is not XML), PDF or RTF. In the first case, XSLT is sufficient for producing the output. In the second case, the transformation process is followed by a *formatting* step that is implemented by a specific software mapping XML data to a non-XML format. This formatting step can be a simple mapping from XHTML to HTML by changing some small syntactic differences, and a more complex generation of high-quality electronic documents in a binary format like RTF and PDF. For the second case, XSL proposes XSL-FO, a standard XML vocabulary for typesetting XML documents.

Data Exchange and Service Integration

Service-oriented architectures (SOA) based on the W3C SOAP/WSDL web service recommendations represent a modern solution for XML-based data and application integration. These kind of infrastructures strongly depend on efficient XML data transformation tools for wrapping data exchanged between heterogeneous web services. By its declarative and modular nature, XSLT is a powerful language for defining data and service wrappers, and is proposed as such by standard business process modeling languages (BPEL) and infrastructures (JBI).

Model Exchange and Transformation

On a more abstract level, XSLT has been proposed for the transformation of XMI (XML Metadata Interchange) documents. XMI is a standard interchange mechanism used model driven in component based development and deployment infrastructures (MDA). The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels). In this kind of environment, XSLT can be used for transforming definitions a some model A, for example UML, into a different model B, for example XML Schema.

Url to Code

XSLT code examples can be found at the following URL:

http://www-poleia.lip6.fr/~amann/XSLT/index.xml

Cross-references

- ▶ Tree Grammars and Languages
- ▶ Web Services
- ▶ XML
- ▶ XPath/XQuery

Recommended Reading

1. Abiteboul S., Buneman P., and Suciu D. Data on the Web: from relations to semistructured data and XML. Morgan Kaufmann, Los Altos, CA, 1999.
2. Bex G.J., Maneth S., and Neven F. A formal model for an expressive fragment of XSLT. In Proc. 1st Int. Conf. Computational Logic, 2000, pp. 1137–1151.
3. Fokoue A., Rose K.H., Siméon J., and Villard L. Compiling XSLT 2.0 into XQuery 1.0. In Proc. 14th Int. World Wide Web Conference, 2005, pp. 682–691.
4. Kay M. XSLT Programmer's Reference, 2nd edition, WROX Press Ltd., 2002.
5. Liu Z.H. and Novoselsky A. Efficient XSLT processing in relational database system. In Proc. 32nd Int. Conf. on Very Large Data Bases, 2006, pp. 1106–1116.
6. Milo T., Suciu D., and Vianu V. Typechecking for XML Transformers. In Proc. 19th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2000, pp. 11–22.
7. Moerkotte G. Incorporating XSL processing into database engines. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 107–118.
8. Muench S. Building Oracle XML Applications, O'Reilly, 2000.
9. Neven F. and Schwentick T. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. Logic. Methods Comput. Sci., 2(3), 2006.
10. W3C XSL-FO Tutorial, http://www.w3schools.com/xslfo/default.asp.
11. W3C. XSL Transformations (XSLT) Version 1.0, W3C Recommendation, J. Clark (ed.). http://www.w3.org/TR/xslt, 1999.
12. W3C. XML Path Language (XPath) Version 1.0, W3C Recommendation, J. Clark and S. DeRose (eds.). http://www.w3.org/TR/xpath, 1999.
13. W3C. Extensible Stylesheet Language (XSL) Version 1.0, W3C Recommendation, S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, S. Zilles, (eds.). hhttp://www.w3.org/TR/2001/REC-xsl-20011015/, 2001.
14. W3C. XML Path Language (XPath) 2.0, W3C Recommendation, A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, J. Siméon (eds.). http://www.w3.org/TR/xpath20, 2007.
15. W3C. XSL Transformations (XSLT) Version 2.0, W3C Recommendation, M. Kay, (ed.). http://www.w3.org/TR/xslt20, 2007.

XSL-FO

- ▶ XSL/XSLT