Editor: Siobhán Clarke • siobhan.clarke@cs.tcd.ie

XML Document Indexes: A Classification

Barbara Catania and **Anna Maddalena** • *University of Genoa, Italy*
Athena Vakali • *Aristotle University*

XML's increasing diffusion makes efficient XML query processing and indexing all the more critical. Given the semistructured nature of XML documents, however, general query processing techniques won't work. Researchers have proposed several specialized indexing methods that offer query processors efficient access to XML documents, although none are yet fully implemented in commercial products. The classification of XML indexing techniques in this article identifies current practices and trends, offering insight into how developers can improve query processing and select the best solution for particular contexts.

The fact that XML (www.w3.org/XML) lets developers simply and flexibly represent and exchange data over the Web has dramatically increased XML document proliferation. This, in turn, has increased the demand for ad hoc techniques for XML query processing. As in any data management application, XML-based systems can achieve effective and efficient query execution by providing

- a sufficiently expressive query language, such as XQuery (www.w3.org/XML/Query) and XPath (www.w3.org/TR/xpath) for XML;
- an efficient query processor; and
- efficient indexing techniques that let the query processor efficiently access (parts of) XML documents according to query conditions.

Because choosing the most efficient query execution plan relies on indexing techniques, such techniques play an important role in developing query processors. In the Web context, they're even more crucial, as XML documents are massively used and frequently queried. Given that XML documents are semistructured, however, general query processing techniques — such as those for relational or object-oriented data — won't work.

Researchers have proposed several XML-specific indexing approaches, but to our knowledge, a unified overview of XML indexing has yet to appear. Such an overview and classification would be useful both theoretically, during design and development, and practically, in helping XML application developers choose the appropriate product.

Here, we present a classification of XML indexing techniques based on two key factors. First, we identify the query type that can be optimized. Second, we identify the query processing strategy that benefits from a specific indexing technique. Our classification highlights each technique's main characteristics, advantages, and drawbacks. In addition to helping developers choose the best XML query management strategies, the classification can help users of those solutions understand why they get specific performance results.

Querying XML Documents

Each XML document is composed of a set of elements with a possible nested structure. Figure 1a shows an example XML document. The document's basic component is the *element*, a piece of text bounded by matching tags (such as `<Actor>` and `</Actor>`). Inside an element, there might be just text, other elements (such as `Role` inside

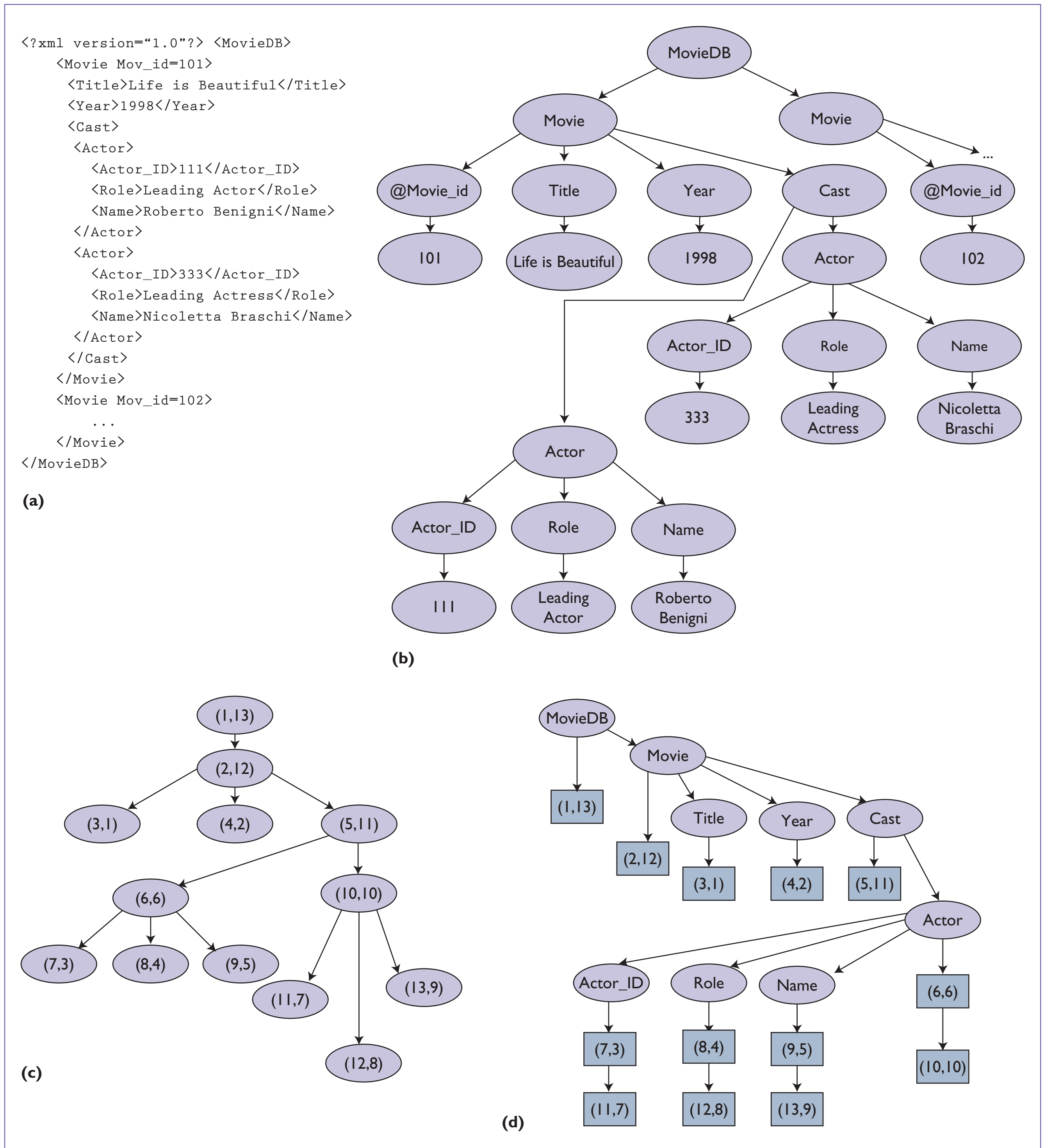

Figure 1. XML document representations. We present (a) a basic XML document; (b) the document's tree-based representation; (c) an ordering for the elements in the subtree corresponding to Movie 101; and (d) a summary index for the XML document corresponding to Movie 101.

`Actor`), or attributes (such as `Mov_id` for element `Movie`), or a mixture of such items. XML documents are generally represented as rooted, unordered, and labeled trees in which each node corresponds to an element, and each edge represents a parent-child relationship (see Figure 1b).

XML Queries

XPath and XQuery are the standard XML querying languages. An XML query specifies selection predicates for multiple elements or attributes that share some tree-based relationships (see Figure 2). In a query's tree-based representation, nodes represent an element tag, an attribute tag, or a value; edges represent hierarchical relationships between XML elements (ancestor–descendant, element–subelement, element–attribute, element–value, or attribute–value). Thus, both nodes and edges represent conditions that the retrieved XML documents must satisfy.

We can classify XML queries in three ways:

- **Tree structure.** As Figure 2 shows, XML queries can be classified into *simple path* or *branching path* expressions. In the first case, the tree corresponds to a chain-path. In the second case, it contains branches and corresponds to a small tree, called a twig.
- **Starting node.** *Total matching* queries are those that start from the root of the document representation, whereas *partial matching* queries start from some internal node. For example, the document in Figure 1a does not satisfy the total matching query `/cast/actor[@role='Leading actor']`. It does, however, satisfy the partial matching query `//cast/actor[@role='Leading actor']`.
- **Node types.** XML queries can contain nodes representing text associated with the father attribute or element node. We call such queries *content-based queries* because they check element or attribute content.

Query Processing

XML developers can currently choose from several XML query language implementations. However, most of them—especially the open source ones—rely on conventional top-down or bottom-up XML tree traversals to evaluate path expressions. As a result, they become highly inefficient for large document collections.

To reduce query processing overhead, we can use index structures to efficiently query large, unconstrained document collections. Each indexing technique is simply a data structure that provides efficient access to all the elements and attributes satisfying specific conditions. Query processors identify elements and attributes using specific location schemes. Such schemes assign a unique identifier to each element, depending on its position in the document (attributes always have the same location as their fathers). We can broadly classify location schemes into two main groups: position-based and path-based schemes.

Position-based schemes assign a numeric position to the XML datasets' (element) nodes according to their position in the document. Based on the chosen location scheme's properties, we can improve structural checks. Among the proposed location schemes, the Dietz's scheme assigns to each node a pair <*preorder, postorder*>, where *preorder* (*postorder*) is a number progressively assigned with a *preorder* (*postorder*) visit of the XML tree.[1] A node v is an ancestor of a node u if and only if $preorder(v) < preorder(u)$ and $postorder(v) > postorder(u)$ (see Figure 1c). This approach's main problem is that node positions change when new nodes are inserted or deleted. Introducing a gap between two consecutive labels can partially solve this problem.[2] This scheme fails, however, if the space required to hold inserted nodes exceeds the reserved space.

In *path-based schemes,* an element's position is represented by an element path from the document's root to the considered element. Also called *prefix labeling*, path-based schemes let each node inherit its parent's label as its own label's prefix so that inserting new nodes doesn't affect existing nodes' labels.[1,3] For example, a path-based position of `Roberto Benigni` in Figure 1a's document is 1.1.4.1.3.1. Although invariant with respect to updates, path-based schemes can lead to high storage overhead and might decrease join performance.

Once the elements and attributes are identified, the query processor can combine the results of various index-based selections to get the result of the overall path or branching query.

Indexing Technique Classification

We can broadly classify indexing techniques based on query types that can be executed with a single index lookup in *summary*, *structural join*, and *sequence-based* indexes.

Summary Indexes

As Figure 3b shows, summary indexes index attributes and elements based on XML documents' paths identifying them. Thus, simple path expressions involving only father–child relationships can be executed by a simple index lookup. On the other hand, with a few exceptions, general branching queries require an additional processing because often they must be decomposed into a set of inde-

pendently executed path queries, with the results then merged together. This approach can't efficiently handle path expressions containing ancestor–descendant relationships, including partial matching queries; it also fails to directly support selection conditions over internal tree nodes.

In its simplest form, a summary index associates XML document paths with the element set those paths reach. It can therefore be implemented as a tree, in which each node represents a tag name and is associated with all element positions a path reaches in the document from the root to that tag name (see Figure 1d). More complex data structures are required, however, to cope not only with path expressions, but also with more general query types, such as branching and content-based queries.

We can classify summary indexes as total or partial.

Total summary indexes. In these indexes, all XML dataset paths are represented inside the tree. Total summary indexes are convenient for processing path-based queries with selection conditions on the final node, because such queries can be executed with a single lookup. However, these indexes rarely offer direct support for branching queries or queries with selection conditions over internal nodes, and a further processing is required. Total summary indexes trace off all label paths starting from the root element, making them suitable to solve total matching queries, although they're less convenient for partial matching queries. Researchers have recently proposed specific data structures for overcoming some of these limitations.

The total summary approach's main disadvantage is its complexity. Indeed, in the general case, the approach's index-generation process is exponential to the database size. When XML documents in the input dataset don't conform to a DTD or XML schema — and thus have high structural variability — the index creation process can get quite expensive.

Partial summary indexes. Because not all of an XML dataset's paths are interesting during query execution, partial summary indexes contain only a subset representing the most common paths or path templates corresponding to frequent queries. Generally, partial summary indexes have the same drawbacks as total summary indexes with respect to branching and content-based queries.

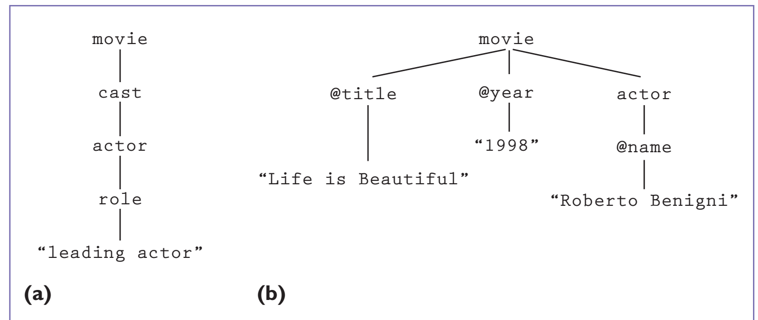

Figure 2. Tree-based representation of queries. (a) A simple content-based query: Q1 = `movie/cast/actor[@role = 'Leading actor']`. *(b) A branching content-based query: Q2 =* `movie[@title = 'Life is Beautiful' AND @year = '1998']/actor[@name = 'Roberto Benigni']`.

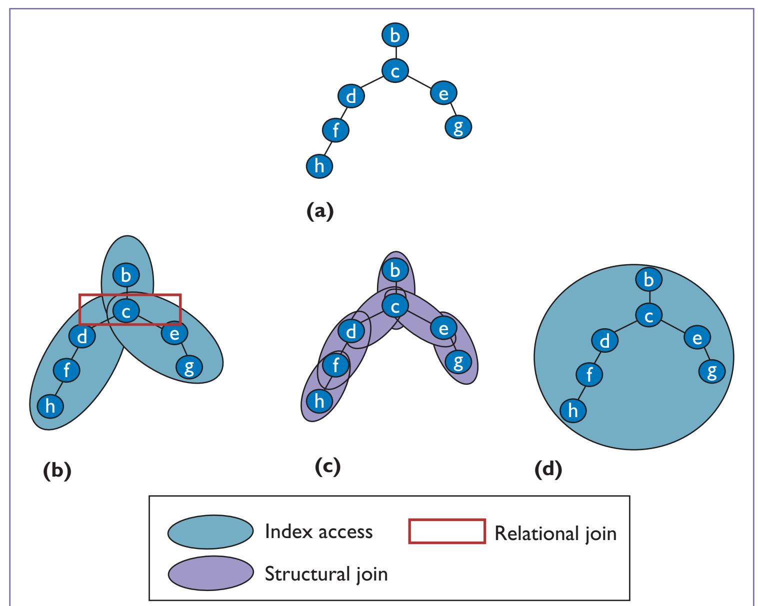

Figure 3. XML query processing approaches. (a) A branching query. (b) A schematic representation of path-based processing, which decomposes query (a) into a set of independently executed path queries. (c) A schematic join-based processing, which decomposes query (a) into a set of independently executed parent–child or ancestor–descendant queries. (d) A schematic representation of sequence-based processing, which executes query (a) in a single step.

Index examples. Table 1 (next page) summarizes the characteristics of several total and partial summary indexes:

- DataGuide was the first proposed summary index; in it, each path is associated with the set of elements it reaches.[4]
- Forward and Backward Index is a minimum index covering all branching queries.[5]

Table 1. Features of summary indexing techniques.

Name	Type	Path	Branching	Total match	Partial match	Content-based	Data structure
DataGuide	Total	x	Additional processing	x	Inefficient	Additional processing	Graph-based structure
Forward and Backward Index	Total	x	x	x	Inefficient	Additional processing	Disk-based clustered tree
Template Index	Partial	x	Additional processing	x	Template dependent	Additional processing	DataGuide
Adaptive Path Index for XML Data	Partial	x	Additional processing	x	x	Additional processing	Hash-table + graph-based structure

- Template Index supports the representation of paths obtained by instantiating some specific path template.[6] The index stores the equivalence class of paths with respect to the chosen template, according to a bisimilarity relation.
- Adaptive Path Index for XML Data considers the most frequently used paths and efficiently solves partial matching queries involving ancestor–descendant relationships.[7]

Structural Join Indexes

Structural join indexes index attributes and elements with a particular name or those whose content satisfies a given condition. In general, developers use these indexes in the context of *join-based processing*, in which the query processor first determines elements that match query tree nodes. It then joins the obtained sets together with a *structural join* algorithm, using specific location schemes to improve processing (see Figure 3c). Structural join is a key issue in optimizing XML queries, and researchers have proposed various techniques, ranging from variations of the relational merge–join algorithms[8] to techniques for reducing the computation of useless intermediate results — possibly by relying on total summary index use.[9] Developers can use join-based processing to solve both path and branching queries, as well as total and partial matching queries, without changes in processing or performance.

We classify structural join indexes in three main groups: simple, full-text, and structure-based indexes.

Simple indexes. These indexes return the set of elements and attributes that satisfy a certain condition that can be locally checked against them. The condition might involve the content associated with an element or attribute (*value-based index*) or their tag name (*name index*). Developers typically build simple indexes using relational index technologies, such as B-tree.

Full-text indexes. These indexes return the set of elements that satisfy a certain condition over their textual content. Full-text indexes typically rely on inverted indexes, which associate each word with its position in its host document. Developers can implement a full-text index as a B-tree. The location scheme they use can change the number of full-text queries that the application can execute. Path-based location schemes, for example, don't support proximity queries (queries that ask for groups of words at a certain distance from each other).

Structure-based indexes. These indexes return elements based on their structural relationships (ancestor–descendant or parent–child). Applications can, for example, skip ancestor and descendant elements that don't participate in a join. Some structure-based indexes guarantee good performance during updates, even when they rely on position-based schemes.

Index examples. Table 2 summarizes the characteristics of several structural join indexes:

- XML Indexing and Storage System supports simple name indexes.[2]
- XML Region Tree reduces the number of elements retrieved, depending on their required relationships.[10] Researchers have also proposed a new structural-join algorithm using XR-tree.
- The Boxes methods use a position-based labeling scheme and ad hoc data structures to achieve good query and update costs.[11]
- Lazy Join executes updates in batch and models XML documents as a tree of XML document segments, then indexes them.[12]

Sequence-Based Indexes

In *sequence-based indexes*, XML documents and branching queries are represented as sequences, and subsequence matching provides the query

Table 2. Features of structural join indexing techniques.

Name	Type	Path	Branching	Total match	Partial match	Content-based	Data structure
XML Indexing and Storage System	Simple	x	x	x	x	x	B+-tree
XML Region Tree	Structure-based	x	x	x	x	x	B+-tree
The Boxes methods	Structure-based	x	x	x	x	x	B+-tree variant, weight-balanced B-trees
Lazy Join	Structure-based	x	x	x	x	x	B+-tree

Table 3. Features of sequence-based indexing techniques.

Name	Type	Path	Branching	Total match	Partial match	Content-based	Data structure
Virtual Suffix Tree	Total	x	x	x	x	Additional processing	B+-tree
Prüfer sequences	Total	x	x	x	x	Additional processing	B+-tree
Wang/Meng method	Total	x	x	x	x	Additional processing	Tree-like data structure

answers. Thus, unlike other approaches, sequence-based XML indexing uses the tree structure as the basic query unit (see Figure 3d), thus avoiding the need to disassemble a structured query into multiple subqueries. As with summary indexes, however, sequence-based indexes don't directly support queries with selection conditions over internal nodes.

Sequence-based indexes can generate false hits—that is, a subsequence match won't always correspond to a subtree match between the query and the documents. In such cases, adding a refinement step can eliminate the false hits. Researchers have also defined specific sequencing methods to avoid this problem.

Index examples. Table 3 summarizes the characteristics of three sequence-based indexes:

- Virtual Suffix Tree codifies XML documents and queries as sequences of pairs, each representing a node and the path (including node content) to reach it, according to a tree preorder visit.[13] The worst-case storage is more than linear in the total number of elements.
- Prüfer sequences for indexing XML codify XML documents and queries as sequences of labels, corresponding to Prüfer sequences.[14] The worst-case storage is linear in the total number of elements.
- Haixun Wang and Xiaofeng Meng's method uses classes of sequencing methods to preserve query equivalence between a structural match and a subsequence match.[15] They also propose algorithms to index the resulting sequences.

Commercial and Prototype Indexing Systems

Developers can efficiently manage XML documents using one of two main types of database management systems:

- *XML-enabled DBMSs*. These (usually relational) DBMSs provide interfaces for transforming data from XML to the internal data model and vice versa. Representation can be structured (if the underlying data model represents XML documents) or unstructured (or *native*, when XML documents are represented as texts using specific XML types). Examples of this DBMS type are Oracle, Microsoft SQL Server, and IBM DB2.
- *XML-native DBMSs*. These DBMSs store and retrieve XML documents according to a proprietary data model. The only interface to XML data is based on XML technologies (such as SAX, DOM, XPath, XQuery, and so on). Examples here include dbXML, eXist, Xindice, Ipedo, Timber, and Tamino.

Table 4 (next page) elaborates on various example technologies and their features.

Because XML-enabled DBMSs rely on (rela-

Table 4. Indexing techniques in XML management systems.

Name	Summary index	Value index	Name index	Full-text index
XML-Enabled Systems				
Oracle (http://technet.oracle.com/products/)	Based on Oracle text index	x	x	x
Microsoft SQL Server (www.microsoft.com/sql)	Dataguide	Ordpath numbering	Ordpath numbering	x
IBM DB2 (www-306.ibm.com/software/data/highlights/techpapers.html)		x	x	x
XML-Native Systems				
dbXML (www.dbxml.com/docs/programmer.html)		x	x	x
eXist (http://exist-db.org/webdb.pdf)		x	x	x
XIndice (http://xml.apache.org/xindice)		x		
Ipedo (www.ipedo.com)		x		x
Timber (www.eecs.umich.edu/db/timber)		x	x	x
Tamino (http://softwareag.com/tamino)		x		

tional) database technology, they provide efficiency, scalability, concurrency control, and reliability. However, under the structured representation, given that XML document content is factorized in several structures based on the underlying data model, query processing performance could decrease. In contrast, unstructured representations in XML-Native DBMSs use proprietary storage techniques, providing high flexibility and efficient space utilization. Current XML-native systems are far from supporting traditional DBMSs functionalities, however, and mainly provide storage and querying services.

Neither XML-native nor XML-enabled approaches implement advanced indexing techniques, which researchers have only recently defined. Both approaches thus support only basic solutions. In particular, XML-enabled DBMSs typically use simple structural indexes (B-trees) for structured representation. Similarly, most XML-native DBMSs index only element and attribute content and tag names. In both cases, the technologies typically adopt full-text (inverted) indexes for indexing textual content or paths. Among XML-enabled systems, Microsoft SQL Server 2005 implements structural join and summary indexes based on Ordpath, a recently defined position-based numbering scheme.[3]

For XML-enabled systems, the gap between theoretical and commercial proposals is probably due to the fact that developers can easily implement simple and full-text structural join indexes using B-trees and text indexes — data structures that the underlying architecture already supports. In contrast, XML-native DBMS technology is probably not mature enough to implement sophisticated indexing techniques.

Given the above classification, we can generally state that when no information is known about the input dataset and the target queries, sequence-based and structural join indexes seem to be the best choice for XML query processors. When the structure of XML documents and queries is defined a priori, however, summary indexes are a convenient option.

Despite the multiplicity of XML indexing techniques proposed in the literature, commercial systems are far from implementing these approaches. Additional work must be done from both theoretical and practical viewpoints to integrate the proposed techniques in commercial systems and further improve their performance.

References

1. I. Tatarinov et al., “Storing and Querying Ordered XML Using a Relational Database System,” *Proc. Int’l Conf. Management of Data* (ACM Sigmod), ACM Press, 2002, pp. 204–215.
2. Q. Li and B. Moon, “Indexing and Querying XML Data for Regular Path Expressions,” *Proc. Int’l Conf. Very Large Databases* (VLDB 01), Morgan Kaufmann, 2001, pp. 361–370.

3. P.E. O'Neil et al., "Ordpaths: Insert-Friendly XML Node Labels," *Proc. Int'l Conf. Management of Data* (ACM Sigmod), ACM Press, 2004, pp. 903–908.
4. R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," *Proc. Int'l Conf. Very Large Databases* (VLDB 01), Morgan Kaufmann, 1997, pp. 436–445.
5. W. Wang et al., "Efficient Processing of XML Path Queries Using the Disk-Based F&B Index," to appear, *Proc. Int'l Conf. Very Large Databases* (VLDB), Morgan Kaufmann, 2005.
6. T. Milo and D. Suciu, "Index Structures for Path Expressions," *Proc. Int'l Conf. Database Theory* (ICDT 99), LNCS 1540, Springer-Verlag, 1999, pp. 277–295.
7. C.W. Chung et al., "APEX: An Adaptive Path Index for XML Data," *Proc. Int'l Conf. Management of Data* (ACM Sigmod), ACM Press, 2002, pp. 121–132.
8. S. Al-Khalifa et al., "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," *Proc. Int'l Conf. Data Eng.* (ICDE 02), IEEE CS Press, 2002, pp. 141–152.
9. T. Chen, J. Lu, and T.W. Ling, "On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques," *Proc. Int'l Conf. Management of Data* (ACM Sigmod), ACM Press, 2005, pp. 455–466.
10. H. Jiang et al., "XR-Tree: Indexing XML Data for Efficient Structural Joins," *Proc. Int'l Conf. Data Eng.* (ICDE 02), IEEE CS Press, 2002, pp. 253–263.
11. A. Silberstein et al., "BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data," *Proc. Int'l Conf. Data Eng.* (ICDE), IEEE CS Press, 2005, pp. 285–296.
12. B. Catania et al., "Lazy XML Updates: Laziness as a Virtue of Update and Structural Join Efficiency," *Proc. Int'l Conf. Management of Data* (ACM Sigmod), ACM Press, 2005, pp. 515–526.
13. H. Wang et al., "ViST: A Dynamic Index Method for Querying XML Data by Tree Structures," *Proc. Int'l Conf. Management of Data* (ACM Sigmod), ACM Press, 2003, pp. 110–121.
14. P.R. Raw and B. Moon, "PRIX: Indexing and Querying XML Using Prüfer Sequences," *Proc. Int'l Conf. Data Eng.* (ICDE), IEEE CS Press, 2004, pp. 288–300.
15. H. Wang and X. Meng, "On the Sequencing of Tree Structures for XML Indexing," *Proc. Int'l Conf. Data Eng.* (ICDE), IEEE CS Press, 2005, pp. 372–383.

Barbara Catania is an associate professor in the Department of Computer and Information Sciences at the University of Genoa, Italy. Her research interests include deductive and constraint databases, spatial databases, XML and Web databases, pattern management and data mining, indexing techniques, and access-control models. Catania has an MSc in information sciences from the University of Genoa and a PhD in computer science from the University of Milan. She is a member of the ACM. Contact her at catania@disi.unige.it.

Anna Maddalena is a PhD student in the Department of Computer and Information Sciences at the University of Genoa. Her research interests include pattern management and data mining, data warehousing, and XML query processing. Maddalena has an MS in computer science from the University of Genoa. Contact her at maddalena@disi.unige.it.

Athena Vakali is an assistant professor in the Department of Informatics at the Aristotle University of Thessaloniki, Greece. Her research interests include Web, XML, and multimedia data management, and Web caching and content delivery. Vakali has an MSc from Purdue University and a PhD from Aristotle University, both in computer science. She is a member of the ACM and the IEEE. Contact her at avakali@csd.auth.gr.

Write for Spotlight

Spotlight focuses on emerging technologies, or new aspects of existing technologies, that will provide the software platforms for Internet applications.

Spotlight articles describe technologies from the perspective of a developer of advanced Web-based applications. Articles should be 2,000 to 3,000 words. Guidelines are at www.computer.org/internet/dept.htm.

To check on a submission's relevance, please contact department editor Siobhán Clarke at siobhan.clarke@cs.tcd.ie.

Tried any new gadgets lately?

Any products your peers should know about? Write a review for *IEEE Pervasive Computing*, and tell us why you were impressed. Our New Products department features reviews of the latest components, devices, tools, and other ubiquitous computing gadgets on the market.

Send your reviews and recommendations to **pvcproducts@computer.org** today!

IEEE pervasive COMPUTING
MOBILE AND UBIQUITOUS SYSTEMS