Chapter LXXII
Indices in XML Databases

Hadj Mahboubi
University of Lyon (ERIC Lyon 2), France

Jérôme Darmont
University of Lyon (ERIC Lyon 2), France

INTRODUCTION

Since XML (eXtensible Markup Language) (Bray, Paoli, Sperberg-McQueen, Maler & Yergeau, 2004) emerged as a standard for information representation and exchange, storing, indexing, and querying, XML documents have become major issues in database research. Query processing and optimization are very important in this context, and indices are data structures that help enhance performances substantially. Though XML indexing concepts are mainly inherited from relational databases, XML indices bear numerous specificities.

The aim of this chapter is to present an overview of state-of-the-art XML indices and to discuss the main issues, trade-offs, and future trends in XML indexing. Furthermore, since XML is gaining importance for representing business data for analytics (Beyer, Chamberlin, Colby, Özcan, Pirahesh & Xu, 2005), we also present an index we developed specifically for XML data warehouses.

BACKGROUND

Indexing and querying XML documents through path expressions expressed in XPath (Clark & DeRose, 1999) and XQuery (Boag, Chamberlin, Fernandez, Florescu, Robie & Siméon, 2006) have been the focus of many research studies. Two families of approaches aim at efficiently processing path join queries. They are based on structural summaries and numbering schemes, respectively.

Structural Summary-Based Indices

Structural index-based approaches help traverse XML documents' hierarchies by referencing structural information about these documents. These techniques extract structural information directly from data and create a structural summary that is a labeled, directed graph. Graph schemas can be used as indices for path queries. Dataguide (Goldman & Widom, 1997) and 1-index (Milo & Suciu, 1999) belong to this family of indices.

Dataguide's structure describes by one single label all the nodes whose labels (names) are identical. Its definition is based on targeted path sets (i.e., sets of nodes that are reached by traversing a given path).

1-index clusters nodes according to a bisimilarity relationship. Two nodes are said to be bisimilar if they share identical label paths in the XML data graph. Bisimilar nodes are grouped into one index node. A 1-index is smaller than the initial data graph and thereby facilitates query evaluation. To help select labels or evaluate path expressions, hash tables or B-trees are used to index graph labels.

Dataguide and 1-index code all paths from the root node. The size of such summary structures may be larger than the original XML document, which degrades query performance. A(k)-index (Kaushik, Shenoy, Bohannon & Gudes, 2002) is a variant of 1-index based on k-dissimilarity and builds an approximate index to reduce its graph's size. An A(k)-index can retrieve, without referring to the data graph, path expressions of length of at most k, where k controls index resolution and influences index size in a proportional manner. However, for large values of k, index size may still become very large. For small values of k, index size is substantially smaller, but A(k)-index cannot handle long path expressions.

To accommodate path expressions of various lengths without unnecessarily increasing index size, D(k)-index (Qun, Lim & Ong, 2003) assigns different values of k to index nodes. These values conform to a given set of frequently used path expressions (FUPs). Small or large values of k are assigned to index parts that are targeted by short or long path expressions, respectively. To help evaluate path expressions with branching, a variant called UD(k, l)-index (Wu, Wang, Xu Yu, Zhou & Zhou, 2003) also imposes downward similarity.

AD(k)-index (He & Yang, 2004) builds a coarser index than A(k)-index but suffers from over-refinement. M(k)-index, an improvement of D(k)-index, and solves the problem of large scan space within the index without affecting path coverage. However, there is a drawback in this design: M(k)-index requires adapting to a given list of FUPs.

U(*)-index (universal, generic index) (Boulos & Karakashian, 2006), like 1-index, exploits bisimilarity. However, U(*)-index exploits a special node-labeling scheme to prune the search space and accelerate XPath evaluations. Furthermore, U(*)-index does not need to be adapted to any particular list of FUP; it has a uniform resolution and is hence more generic.

APEX (Chung, Min & Shim, 2002) is an adaptive index that searches for a trade-off between size and effectiveness. Instead of indexing all paths from the root, APEX only indexes frequently used paths and preserves the structure of source data in a tree. However, since FUPs are stored in the index, path query processing is quite efficient. APEX is also workload-aware (i.e., it can be dynamically updated according to changes in query workload). A data mining method is used to extract FUPs from the workload for incremental update (Agrawal & Srikant, 1995).

The main weakness of these indices is that they can only answer single path expressions directly. To process so-called branching path expressions whose graphical representation contains branches and corresponds to a small tree (or twig), they must perform a costly join operation. To reduce