Storing XML Documents in Databases

Albrecht Schmidt
Aalborg University, Denmark

Stefan Manegold
CWI, The Netherlands

Martin Kersten
Center for Mathematics and Computer Science, The Netherlands

INTRODUCTION

Ever since the Extensible Markup Language (XML) (W3C, 1998b) began to be used to exchange data between diverse sources, interest has grown in deploying data management technology to store and query XML documents. A number of approaches propose to adapt relational database technology to store and maintain XML documents (Deutsch, Fernandez & Suciu, 1999; Florescu & Kossmann, 1999; Klettke & Meyer, 2000; Shanmugasundaram et al., 1999; Tatarinov et al., 2002; O’Neil et al., 2004). The advantage is that the XML repository inherits all the power of mature relational technology like indexes and transaction management. For XML-enabled querying, a declarative query language (Chamberlin et al., 2001) is available.

Traditionally, database technology has been offering support for processing large amounts of data. Recent research has provided valuable insights into the nature of semistructured and XML data and has attempted to integrate them into existing paradigms. However, there are still challenges that have to be met to scale XML databases up to production levels as achieved by relational engines and, thus, to gain acceptance among practitioners. Naturally, XML warehouses inherit the power of relational warehouses (Roussopoulos, 1997), but they also face the same challenges; in particular, update and consistency problems of materialized, replicated, and aggregated views over source data need to be solved.

This article discusses techniques related to loading XML documents into a document warehouse. All techniques build on well-understood relational database technology and enable efficient management of large XML repositories. To get the most of relational database systems, we propose to do away with the pointer-chasing tree traversing operations, which many applications generate in the form of edit scripts and replace them with set-oriented operations. Edit scripts (Chawathe et al., 1996; Chawathe & Garcia-Molina, 1997) have been long known in text databases and are similar in behavior to Document Object Model (DOM) (W3C, 1998a) traversals, which are standard in the XML world; they tend to put relational technology at a disadvantage due to their excessive use of pointer-chasing algorithms. We investigate the use of these scripts and propose alternative strategies for cases when they perform poorly.

We implemented our ideas in the XML extension of the Monet Database System (Schmidt, Kersten & Windhouwer, 2001; Schmidt et al., 2000). A more detailed description of our experiments is found in Schmidt and Kersten (2002). As we benchmarked the system’s performance, it turns out that the use of edit-scripts is only sensible if they only update a rather small fraction of the database; once a certain threshold is exceeded, the replacement of a complete database segment is preferable. We discuss this threshold and try to quantify the trade-off for our example document database.

The application scenario which motivates our research consists of a set of XML data sources which are feature detectors that monitor multimedia data sources and analyze their content. The detectors feed protocols of analyses into a central data warehouse. The warehouse now provides the following services: (1) insertion of documents (a data source transmits a single protocol of an analysis to the warehouse), (2) insertion of versioned sets of documents (a set of check-out points transmits the result of a bulk analysis transcript to the warehouse), (3) deletion of documents and sets of documents (a document is deleted from the warehouse because it has become invalid or stale; duplicate analyses and erroneous insertion also happen frequently and need to be corrected), and (4) execution of edit-scripts that are transmitted from the sources and systematically correct errors in already inserted documents; for example, a posteriori normalization of feature values is frequently required.

While we regard (1) as a special case of (2), hence, do not treat it separately, there is an obvious trade-off between a combination of (2) and (3) and the use of edit-scripts (4). More precisely, the question is: When is it
cheaper to delete invalid data and reinsert a new consistent version than to use an edit script to “patch” the warehouse? This and other questions will be dealt with in detail later.

BACKGROUND

XML documents are commonly represented as syntax trees. This section recalls some of the usual terminology we need to work with XML documents. In the sequel, string and int denote sets of character strings, respectively integers; oid denotes a set of unique object identifiers. Figure 1 shows an XML fragment, which is taken from the area of content-based multimedia retrieval (Schmidt & Kersten, 2002). Figure 2 displays the corresponding schema tree (dotted arrows indicate XML attribute relationships, straight lines XML element relationships).

Before we discuss techniques on how to store a tree as a database instance, we introduce the notion of associations. They are used to cluster semantically related information in a single relation and constitute the basis for the Monet XML Model; the aim of the clustering process is to enable efficient scans over semantically related data, that is, data with the same element ancestry, which are the physical backbone of declarative associative query language like SQL. Different types of associations play different roles: associations of type oid×oid represent parent-child relationships. Both kinds of leaves, attribute values and character data, are modeled by associations of type oid×string, while associations of type oid×int are used to keep track of the original topology of a document. Paths describe the context of the element in the graph relative to the root node; we identify with path(o) the type of the association (x,o). The set of all paths in a document is called its Path Summary; it plays an important role in our query engine. The main rational for the path-centric storage of documents is to evaluate the ubiquitous XML path expressions efficiently; the high degree of semantic clustering distinguishes our approach from other mappings (see Florescu & Kossmann, 1999 for a discussion). Our approach is to store all associations of the same “type” in one binary relation. A relation that contains the tuple (.,o) is named R(path(o)). In Figure 2, the types or paths are the i. Clustering XML elements by their type implies that we do not have to cope with many of the irregularities induced by the semi-structured nature of XML, which are typically taken care of with NULLs or overflow tables (Deutsch et al., 1999). In the sequel, we describe the machinery we need to convert documents to Monet format and bulkload them efficiently. Also note that we are able to reconstruct the original document given this path-centric representation. A detailed discussion of the reconstruction can be found in Schmidt et al. (2001). We remark that we can also access the documents in an object-oriented manner, that is, object as node in the syntax tree, which is often more intuitive to the user and is adopted by standards like the DOM (W3C, 1998a). However, we do not optimize for this as we see later.

XML WAREHOUSES

Populating the XML Warehouse

There are two basic notions of interest that we are going to discuss in this section as indicated in the introduction: populating a database from scratch, that is, bulk load, and incremental insertion of new data into an already existing database. However, similar technology underlies both cases. Let us consider an example first. There are two standard ways of accessing XML documents: (1) A low-level event-based, called SAX (Meggison, 2001), scans an XML document for token like start tag, end tag, character data, and so forth and invokes user-supplied functions for each token that is encountered in the input. The advantage of the SAX parsers is they only require minimal resources to work;