XML Databases

George Papamarkos, Lucas Zamboulis, Alexandra Poulovassilis
School of Computer Science and Information Systems,
Birkbeck College, University of London

Contents

1 Introduction

In parallel with the rapid growth of the World Wide Web, arose the need for a common data format. Having proprietary data formats was sufficient for managing data within businesses, or even for communicating data across a small number of partners; but this model was not scalable. The advantages of interoperability, exemplified by the Web, gave rise to the need for a highly standardised common data format for data exchange between applications. The solution to this problem came with the advent of XML 1.0 [14] and XML 1.1 [19].

XML gave a significant boost to web-based and business-to-business (B2B) applications. Data were stored in relational databases and XML was used as a medium to transport data between partners. This model quickly became the de facto standard for deploying applications that managed large volumes of data and either wanted to be able to communicate with other businesses or to expose their data on the web. This model was supported by *XML-enabled databases* (XEDs), which adopted a simple strategy to store XML data: each XML document is decomposed and its data are stored within tables.

The now widespread adoption of XML brought on a new trend: storing data in XML format was now an area of new interest. The popularity of XML, a much less complex but still powerful enough subset of SGML [6], helped it gain ground in the application area of SGML itself, including the printing and publishing industries. Along with this shift in use, a new challenge started to become apparent. The XML data produced by these latter applications were no longer small to medium sized documents; however, XML-enabled databases were not capable of supporting documents of this size, as decomposing and recomposing huge documents meant a high number of join operations, leading to unacceptable performance, both in retrieving documents and in querying them.

This technology gap was covered by *native XML databases* (NXDs), that is database systems specifically developed for storing XML documents. The fundamental difference between XEDs and NXDs is that the latter adopt the XML data model for storing XML data. Much like hierarchical and object databases, they are able to preserve the hierarchy and ordering of nodes of XML documents in a much more efficient manner than XML-enabled databases, hence the tremendous performance improvement in handling large XML documents.

A third type of XML databases are *hybrid XML databases*. These are relational or object-oriented database systems that existed before the new paradigm of NXDs was invented. These database systems were subsequently extended to be able to handle XML documents natively; it is their ability to handle both relational/object-oriented data and XML data that has given them the characterisation as hybrid. These systems present a significant advantage in applications where interoperability between XML and other data is needed.

2 Flavours of XML Databases

Before specifying the different types of XML databases, it is necessary to note that the area of XML databases is relatively new and rapidly evolving. As a consequence, the boundaries between different flavours of XML databases are not clear. The fact that many times it is difficult to classify XML documents as data-centric or document-centric (see Section 3 is another factor contributing to this problem.

Having said this, the predominant opinion in the XML database community for distinguishing between XML-enabled and native XML databases is that the first uses schema-specific structures that must be mapped to the XML document at design time. Native XML storage uses generic structures that can contain any XML document. The following definition, proposed by the XML:DB initiative[1] reflects this opinion, and is the de facto definition for XML databases at this time.

There are three different types of XML databases:

1. Native XML Database (NXD): a database that:
   (a) Defines a (logical) model for an XML document — as opposed to the data in that document – and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Examples of such models are the XPath data model, the XML Infoset, and the models implied by the DOM and the events in SAX 1.0.
   (b) Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
   (c) Is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.
2. XML Enabled Database (XEDB): A database that has an added XML mapping layer provided either by the database vendor or a third party. This mapping layer manages the storage and retrieval of XML data. Data that is mapped into the database is mapped into application specific formats and the original XML meta-data and structure may be lost. Data retrieved as XML is NOT guaranteed to have originated in XML form. Data manipulation may occur via either XML specific technologies (e.g. XPath, XSLT, DOM or SAX) or other database technologies (e.g. SQL). The fundamental unit of storage in an XML Enabled Database is implementation dependent.
3. Hybrid XML Databases (HXD): A database that can be treated as either a Native XML Database or as an XML Enabled Database depending on the requirements of the application.

[1]http://xmldb-org.sourceforge.net/

3 Data-Centric vs. Document-Centric XML

Deciding whether to use an XML-enabled, native or hybrid XML database is in many cases difficult, and usually depends both on the specific application and the format of the XML documents. Before delving into the characteristics of each XML database type, one should have in mind the distinction between data-centric and document-centric documents. Even though it is dated, this metaphor usually helps to identify the type of database that is more suitable for a particular application.

3.1 Data-Centric Documents

Data-centric are documents produced as an import or export format, that is, data-centric XML documents are used for machine consumption. These documents are used for communicating data between companies or applications and the fact that XML is used as a common format is simply a matter of convenience, for reasons of interoperability. Examples of data-centric documents are sales orders, scientific data, and stock quotes.

Since data-centric documents are primarily processed by machines, they have fairly regular structure, fine-grained data and no mixed content[2]. Also, the order in which sibling element and text nodes is not significant for the semantics of the document[3].

For example, the following memo document is data-centric:

```
<Memo>
    <Meeting date="23/09/2005" time="10:30AM">Finance Committee</Meeting>
    <Purpose>Discuss 2006 Budget</Purpose>
    <Location>Room 923</Location>
</Memo>
```

The above XML document is clearly data-centric: it has rigid structure, fine-grained data and contains no mixed content. A not so obvious example of data-centric documents are web pages in an e-commerce site. Although such pages contain large amounts of text, such as information for a product, the content of the pages is highly structured and is common for all products. Such pages, even when designed as a combination of HTML and CSS stylesheets, could be easily designed as data-centric XML documents together with XSL stylesheets.

3.2 Document-Centric Documents

Document-centric are documents usually designed for human consumption, with examples ranging from books to hand-written XHTML documents. They are usually composed directly in XML, or some other format and then converted to XML.

Document-centric documents do not need to have regular structure, have coarse-grained data (that is the smallest independent data unit may as well be a document itself) and have mixed content. The examples given above make it clear that the ordering of elements in such documents is always significant.

For example, the following memo document is document-centric[4]:

[2]Mixed content refers to XML elements that can contain both element and text nodes as child nodes.

[3]Note, however, that it is possible for applications to process element children based on their ordering, causing them to crash if ordering is not preserved.

[4]Both 'memo' examples based on http://www.xmleverywhere.com/newsletters/20000525.htm

```
<Memo>
    Please can Finance Committee members come to
    <Location>Room 923</Location>on
    <MeetingDate>23/09/2005</MeetingDate> at
    <MeetingTime>10:30 AM</MeetingTime> to
    <Purpose>discuss the 2006 budget</Purpose>
</Memo>
```

3.3 Discussion

This section discussed the difference between data-centric and document-centric documents, a distinction which serves as a good entry point in the world of XML databases. It must be noted, however, that this distinction is a bit unrealistic in the sense that it is seldom easy to classify a document in just one of the two categories, since most documents have characteristics from both categories. Nevertheless, characterising the documents in an application as data-centric or document-centric will usually suffice to decide the type of database suitable for the application at hand.

4 XML-Enabled Databases

The need to support XML data storage, retrieval and update by utilising existing database systems, along with their well-established technologies, is the main reason that the solution of the XML-enabling layer over the RDBMS was adopted. This extra software layer contains extensions for transferring data between XML documents and the data structures supported by the underlying databases.

The storage methods employed by XML-enabled databases for XML data, along with the main features the systems provide for retrieving, converting and updating XML data are the main topics that are discussed in the sections that follow.

4.1 "Storing" XML in an XED

XML-enabled databases are primarily used in settings where XML is the format for exchanging data between the underlying database and an application or another database, for example when a Web service is providing data stored in a relational database as an XML document.

The main characteristic of the XML-enabled database storage methodology is that it uses a data model other than XML, most commonly the relational data model. Individual instances of this data model are mapped to one or more instances of the XML data model, for example a relational schema can be mapped to different XML schemas depending on the structure of the XML document required by the application that will consume it.

In XML-enabled databases, the XML documents are used as a data exchange format without the XML document to have any particular identity within the database. For example, suppose XML is used to transfer temperature data from a weather station to a database. After the data from a particular document is stored in the database, the document is discarded.

To the opposite direction, when an XML document is requested as a result of a query, it is constructed on-the-fly from the results that are retrieved after querying the underlying database, and once it is consumed by the client that has requested it, it is again discarded. There is no way to ask explicitly for a document by its name, nor does any guarantee exist that the original document that was stored in the database can be reconstructed. Because of this, it is not a good idea to shred a document into an XML-enabled database as a way of storing it. The basic use of XML-enabled systems is for publishing existing relational data (regardless of its source) as XML.

The process of extracting data from a database and constructing an XML document (or XML fragments) is known as *publishing* or *composition*. The reverse process (extracting data from an XML document/fragment and storing it in the database) is known as *shredding* or *decomposition*.

Regardless of whether we are shredding or publishing XML documents, there are two important things to note here. First, an XML-enabled database does not contain XML, i.e. the XML document is completely external to the database. Any XML document/fragment is constructed from data already in the database or is used as a source of new data to store in the database. Second, the database schema matches the XML schema, that is, a different XML schema is needed for each database schema.

XML-enabled databases generally include software for performing both publishing and shredding between relational data and XML documents. This extra piece of software can either be integrated into the database or be provided by a third-party vendor outside the database. This software, used by XML-enabled databases, cannot, generally, handle all possible XML documents. Instead, it can handle the subclass of documents that are needed to model the data found in the database.

Relational-to-XML schema mapping. When using an XML-enabled database, it is necessary to map the database schema to the XML schema (or vice versa). Such mappings are many-to-many. For example, a database schema for sales order information can be mapped to an XML schema for sales order documents, or it can be mapped to an XML schema for reports showing the total sales by region.

There are two important kinds of mappings:

(a) *table-based mapping*

(b) *object-relational mapping*

Both table-based mapping and object-relational mappings define bi-directional mappings, that is, the same mapping can be used to transfer data both to and from the database.

Table-based mapping. When using a table-based mapping, the XML document must have the same structure as a relational database. That is, the data is grouped into "rows" and rows are grouped into "tables". In the following example, the `SalesOrders` and the `Items` elements represent the corresponding relational tables containing a list of `SalesOrder` and `Item` elements, respectively, that in turn represent the rows of each table.

```
<Database>
    <SalesOrders>
        <SalesOrder>
            <Number>123</Number>
            <OrderDate>2003-07-28</OrderDate>
            <CustomerNumber>456</CustomerNumber>
        </SalesOrder>
    </SalesOrders>
    <Items>
        <Item>
            <Number>1</Number>
            <PartNumber>XY-47</PartNumber>
            <Quantity>14</Quantity>
            <Price>16.80</Price>
            <OrderNo>123</OrderNo>
        </Item>
        <Item>
            <Number>2</Number>
            <PartNumber>B-987</PartNumber>
            <Quantity>6</Quantity>
            <Price>2.34</Price>
            <OrderNo>123</OrderNo>
        </Item>
    </Items>
</Database>
```

Object-Relational Mapping. When using an object-relational mapping, an XML document is viewed as a set of serialised objects and is mapped to the database with an object-relational mapping. That is, objects are mapped to tables, properties are mapped to columns, and inter-object relationships are mapped to primary key / foreign key relationships. Below is the same document containing sales orders, as above, but encoded using the object-relational mapping.

```
<Database>
    <SalesOrder>
        <Number>123</Number>
        <OrderDate>2003-07-28</OrderDate>
        <CustomerNumber>456</CustomerNumber>
        <Item>
            <Number>1</Number>
            <PartNumber>XY-47</PartNumber>
            <Quantity>14</Quantity>
            <Price>16.80</Price>
        </Item>
        <Item>
            <Number>2</Number>
            <PartNumber>B-987</PartNumber>
            <Quantity>6</Quantity>
            <Price>2.34</Price>
        </Item>
    </SalesOrder>
</Database>
```

4.2 Retrieving and Modifying

4.2.1 Query Languages

Query languages in XML-enabled databases are used to extract data from the underlying database and transform it. The most widely used query languages for this purpose, *SQL/XML* and *XQuery*, are described in this Section.

SQL/XML. For XML-enabled relational databases, the most widely used query language is SQL/XML, which provides a set of extensions to SQL for creating XML documents and fragments from relational data and is part of the ISO SQL specification of 2003. The main features of SQL/XML are the provision of an XML data type, a set of scalar functions, `XMLELEMENT`, `XMLATTRIBUTES`, `XMLFOREST`, and `XMLCONCAT`, and an aggregate function, `XMLAGG` (see [5] for more details).

For example, the following call to the XMLELEMENT function:

```
XMLELEMENT(NAME Customer,
  XMLELEMENT(NAME Name, customers.name),
  XMLELEMENT(NAME ID, customers.id))
```

constructs the following Customer element for each row in the customers table:

```
<Customer>
  <Name>customer name</Name>
  <ID>customer id</ID>
</Customer>
```

XQuery. XQuery [22] is an XML query language proposed by the W3C. XQuery is designed to query XML documents, not relational databases. Thus, to implement it over a relational database so as to be used by XML-enabled databases, it is necessary to first map the relational database to one or more virtual documents. One way to do this is to map each table to a separate document using the table-based mapping.

For example, this XQuery statement is equivalent to the previous SQL/XML statement:

```
FOR $c IN document("customers")/row/Customer RETURN
<Customer>
    <Name>{$c/name}</Name>
    <ID>{$c/id}</ID>
</Customer>
```

4.2.2 Updating XML-enabled databases

The solutions related to updating XML data in XML-enabled databases vary from implementation to implementation, due to the idiosyncrasies of the storage model. The most common practice is either to use a custom extension of the SQL/XML prototype to support updates (e.g. Oracle XML DB [8]) or use a, custom again, product specific API to perform the modification operations over the stored XML data. In either case, though, the update action does not happen in-place, i.e. directly on the tables where the XML data is stored, but rather the document to be updated is extracted from the database, loaded in memory, updated, then returned to the XML-enabling software layer where it is shredded again and stored back to the database.

5 Native XML Databases

As stated in Section 2, an XML database is *native* if it (a) defines a logical model for an XML document and stores and retrieves documents according to that model and (b) has an XML document as its fundamental unit of (logical) storage, while (c) the storage model itself is not constrained. The rest of this section is structured as follows. First, Section 5.1 discusses logical models for XML data, Section 5.2 focuses on the fundamental unit of storage for NXDs and Section 5.3 illustrates the two basic categories of storage formats for NXDs. Finally, Section 5.4 discusses features provided by NXDs.

5.1 XML data models

A *data model* is an abstraction which incorporates only those properties thought to be relevant to the application at hand. As there are different ways in which one can use XML data, there are more than one XML data models. For example, all XML data models include elements, attributes, text, and document order, but some also include other node types, such as entity references and CDATA sections, while others do not.

Therefore, DTD [15] defines its own XML data model, while XML Schema [18] uses the XML InfoSet [20] data model; XPath [16, 21] and XQuery [22] define their own common data model for querying XML data. This plethora of data models makes it difficult for applications to combine different features, such as schema validation together with querying. The W3C is therefore in the process of merging these data models under the XML InfoSet, which is to be replaced by the Post-Schema-Validation Infoset (PSVI).

Since many NXDs were created prior to the XML InfoSet and the XPath 2.0 and XQuery data model, they were free to define their own data model. Since the data model of an XML query language defines the minimum amount of information that a native XML database must store, these NXDs need to be upgraded to support XQuery. Fortunately, the vast majority of NXDs currently supports at least XPath 1.0, and it is envisaged that all of them will support XQuery in the near future.

5.2 Fundamental unit of storage

A fundamental unit of storage is the smallest grouping of data that logically fits together in storage. From a relational database perspective, the fundamental unit of storage is a tuple. From a native XML database perspective, the fundamental unit of storage is a document. Since any document fragment headed by a single element is potentially an XML document, the choice of what constitutes an XML document is purely a design decision.

5.3 Storage Formats

5.3.1 Text-Based Native XML Databases

Text-based native XML databases store XML as text. This may be a file within the database itself, a file in the file system outside the database, or a proprietary text format. Note here that the first case implies that relational databases storing XML documents within CLOB (Character Large OBject) fields are considered native.

All text-based NXDs make use of indices, giving them a big performance advantage when retrieving entire documents or document fragments, as all it takes is a single index-lookup and a single read operation to retrieve the document. This is contrary to XML-enabled databases and some model-based NXDs which require a large number of joins in order to recreate an XML document that has been shredded and inserted in the database. This makes the comparison between text-based NXDs and relational databases analogous to the comparison between hierarchical and relational databases, in that text-based NXDs will outperform relational databases when returning the document or document fragments in the form in which the document is stored; returning data in any other form, such as inverting the document hierarchy, will lead to performance problems.

5.3.2 Model-Based Native XML Databases

Model-based NXDs have an internal object model and use this to store their XML documents. The way this model is stored varies: one way is to use a relational or object-oriented database; other databases use proprietary storage formats, optimised for their chosen data model.

Model-based NXDs built on other databases will have performance similar to those databases, since they rely on these systems to retrieve data; the data model used, however, can make a notable difference in performance.

Model-based NXDs using a proprietary storage format usually have physical pointers between nodes and therefore are expected to have similar performance to text-based NXDs. Clearly, the output format is significant here, as text-based NXDs will probably be faster in outputting in text format, whereas a model-based NXD using the DOM model will be faster in returning a document in DOM format.

5.4 Features

Document Collections Most NXDs organise documents within *collections* (possibly nested), which are similar to tables in a relational database or directories in a file system. This feature allows querying and manipulation of the documents within a collection as a set. An NXD supporting collections *may not* require a schema to be associated with a collection; although this provides a high degree of flexibility in application development, it raises the risk of low data integrity.

Query Languages At first, most NXDs supported either a proprietary query language or programmatic access through DOM. With the advent of XPath, most NXDs built support for it. However, XPath was not designed to be a query language (XPath 1.0 is defined as "a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer"), and therefore has several limitations when used as one, including lack of grouping, sorting, cross document joins, and support for data types.

XQuery [22] has just become a W3C Recommendation. Many NXDs have been building support for it based on the XQuery Working Drafts and it is envisaged that XQuery will become the de facto XML query language. Note that this will probably not include object-oriented NXDs; as an example, Ozone[5] supports an ODMG[6] interface. Note, however, that the existence of standalone or embeddable XQuery engines, such as Galax[7], means that the

[5]http://www.ozone-db.org
[6]http://www.odmg.org/
[7]http://www.galaxquery.org/

lack of XQuery support in an NXD does not prohibit its use when the application requires XQuery support.

It is worth noting here that almost all NXDs support, in some form or another, the creation of indices on the data stored in collections, to improve the query execution speed. The implementation of these indices varies widely between each NXD product, and is mainly based on the storage implementation. The three possible types of indices are:

1. Value indices index text and attribute values.

2. Path indices index the location of elements and attributes.

3. Full-text indices index the individual tokens in text and attribute values.

Most NXDs support both value and path indices, while some NXDs support full-text indices.

Updates and Deletes Up to now, there is no standard for updating XML data. Because of this, most NXDs provide only simple replace/delete operations on documents. Updates on node level are possible only by retrieving a certain document, modifying it using DOM/SAX, then returning it to the database. There are some NXDs that support proprietary update languages, while the hope for a standardised update language for XML rests within the XML:DB XUpdate[8] language and extensions to XQuery (see [7], which builds on the work of [13]); both have been implemented in a number of NXDs.

Transactions, Locking, and Concurrency All NXDs support transactions, but the level at which locking is implemented differs. Most often, locking occurs at the document level, and this could lead to problems in multi-user concurrency.

The problem with node-level locking is that, in order to lock a node n, the parent of n, p_n needs to be locked too, otherwise another transaction would be able to delete p_n and therefore delete n. This goes all the way to the root of the document, making it impossible to update other parts of the document that are not in the path from the root to n. A partial solution to this problem has been proposed by [3] where the locked node is annotated with a query that defines the path from the locked node to the target node, that is the node to be updated.

There are a few commercial NXDs that already support node-level locking and, in the future, most NXDs will probably offer node-level locking.

Application Programming Interfaces (APIs) All NXDs provide at least one API in a programming language such as Java/C++/C# etc. These provide an interface for connecting to the database, executing queries, retrieving results and performing other tasks. At the moment, there is only one vendor-neutral XML API, XML:DB API from XML:DB.org. It is programming language-neutral, supports XPath, is currently being extended to support XQuery, and is supported by most NXDs and some XEDs. A second XML API, relating to XQuery in the same way that JDBC relates to SQL, is XQuery API for Java (XQJ)[9].

Note also that most native XML databases also offer the ability to execute queries and return results over HTTP.

[8]http://xmldb-org.sourceforge.net/xupdate/
[9]http://www.jcp.org/en/jsr/detail?id=225

Round-Tripping A significant issue in XML databases is the round-tripping problem, that is, the ability of an XML database to store a document and then retrieve it without changing it. There are various levels of round-tripping: other databases are able to offer only basic round-tripping, that is for elements, attributes, text and hierarchy, and are therefore better suited for data-centric applications, while others extend their round-tripping support for processing instructions, comments and even physical structure (whitespace) — these are better suited for document-centric applications and applications that are required by law to keep exact copies of documents. As a general rule, model-based NXDs offer round-tripping support at the level of their model, while text-based NXDs can offer the total round-tripping support.

Normalisation & Referential Integrity Normalisation is one of the cornerstones of relational theory and has as a goal to avoid unnecessary redundancy in a database; as a consequence, this also eliminates the risk of inconsistent data. For a comprehensive discussion of normalisation and normal forms, see [12].

Unlike relational data, there are many cases where normalisation for XML data is a non-issue, as there is no data redundancy — and this is usually for document-centric XML documents. Like in relational databases, the decision whether to normalise, and if so up to which normal form, is a design decision. For a discussion on XML Normal Form (XNF), see [4, 1]. For a practical discussion on normalising XML data, see [10, 11].

Closely related to normalisation is referential integrity, which refers to the validity of pointers to related data; this is a necessary part of maintaining a consistent database state and complements normalisation. In a relational setting, referential integrity refers to checking that a primary key tuple referenced by a foreign key actually exists. In an NXD, referential integrity refers to checking that pointers in XML documents refer to valid documents or document fragments.

There are several referencing mechanisms in XML. XML Schema [18] supports the notion of a primary and foreign key through the use of `key` and `keyref`, while DTD [15] uses `ID` and `IDREF`; a third mechanism is XLink [17], while other proprietary mechanisms exist which are able to reference data from relational databases.

Support for referential integrity is offered by all NXDs, by performing a validation check against the schema whenever a document is inserted in the database. If the database does not support node-level updates, then this is sufficient; otherwise the database needs a mechanism for checking referential integrity violations due to updates.

6 Hybrid XML databases

The need to combine the features of both native and XML-enabled databases has led to the creation of a new category of databases call *hybrid XML databases*. A hybrid XML database provides XML data management in both native and XML-enabled fashion. Most modern RDBMSs today are hybrid and allow, depending on the applications requirements, shredding and composition of the XML document to and from relational tables as well as native storage using XML-specific datatypes, that are often supported by XML-specific indexing schemes.

6.1 State-of-the-art Hybrid Databases

In the following sections we give a brief description of the major RDBMS today in terms of their support of XML data. All three RDBMSs below provide hybrid XML management characteristics.

6.1.1 Oracle XML DB

Oracle Database provides XML support since version 9i Release 2. The latest 10g set of releases provides more efficient storage retrieval, querying, updating and generation of XML data. Oracle XML DB, as the XML extension of the Oracle RDBMS is called, provides three ways for storing an XML document within it:

(a) Parse and shred the XML document to an XML-data-driven relational schema

(b) As plain text encoded as `VARCHAR`

(c) As an XML-specific datatype called `XMLType`, either as a column of a relational table or a stand-alone XML document within the Oracle built-in XML repository.

The third is the most advanced way for managing XML data within Oracle, providing highly expressive query and modification capabilities of the XML data via the combined use of SQL/XML and full XQuery. The set of features is completed by the full support of the XML Schema storage and validation capabilities.

The existence of a separate XML repository, within the XML DB, is one of the greatest advantages of the Oracle approach, since its allows the logical organisation of the XML documents in collections and directories and their identification via a URL.

For indexing the **XMLType** columns Oracle XML DB provides a series of options:

- Create B*Tree indexes on the tables that provide storage of **XMLType** tables and columns.
- Use **CONTEXT** domain index type of Oracle Text Indexes to allow text indexing of the content of the **XMLType** tables and columns.
- Create *function-based indexes* that can create indexes on explicit XPath expressions for any XML storage type.
- Use **CTXXPATH** domain index type of Oracle Text Indexes. This index type provides an XML-specific text index with transactional semantics. This index type can speed up certain XPath-based searches on any XML storage type.

For more details on the actual way on creating each of the index schemes above, please see [9].

6.1.2 IBM DB2 XML Extender

XML Extender, as the name reveals, is the XML extension of the IBM DB2 RDBMS. XML Extender provides a wide range of options on storing, retrieving, querying and modifying XML data within DB2.

Storing XML Data The user of the DB2 is given three options on storing XML data:

(a) *Outside the DB2*: The simplest way to store a small amount of XML data. Lacks all the data management features and guarantees provided by the RDBMS.

(b) *Within DB2 but bypassing XML Extender*: As a `VARCHAR`, `VARGRAPHIC` or `CLOB` typed column of a relational table

(c) *Within DB2 using the XML Extender-provided functionality*: Using the XML Extender-provided datatypes: `XMLVARCHAR` for documents up to 3kbytes, `XMLCLOB` and `XMLDBCLOB` for documents up to 2Gbytes, and `XMLFILE` that stores only the filename of the XML document in DB2, keeping the document itself in the filesystem.

Querying and Modifying XML Data DB2 XML Extender provides user-defined functions (UDFs) for performing retrieval, query and update on the stored XML documents making also use of XPath to specify certain parts of the document.

A part or the whole of the XML document can be retrieved using the SELECT statements in conjunction with UDFs and a WHERE clause that filters the results out. When it comes to updating an XML document residing in a column of a table there are two ways: (a) Using the SQL UPDATE statement, that is used for replacing an XML document by a new one, (b) Using the `Update()` UDF, that provides more fine-grained update operations such as updating a specific part of the XML document. The `Update()` function utilises XPath for locating parts of the XML document. Finally, deletions of a whole XML document are performed by simply using the SQL DELETE statement allowing deletion of the whole XML column. For more details see [5].

In DB2 we can index the XML data stored with the DB2 by using the XML-specify solutions. The system of the `CREATE INDEX` statement is as follows:

```
CREATE INDEX name_of_index
    FOR TEXT ON table_name (column_name) FORMAT XML
    [DOCUMENTMODEL document_alias IN path_to_doc_on_the_filesystem]
```

6.1.3 Microsoft SQL Server 2005

Storing XML Data SQL Server 2005 introduces a new datatype for storing XML data as a column in a relational table. This new XML datatype can be set to conform to an XML schema or not, defining a typed or an untyped XML document, respectively. The developer is also able to decide whether the XML document will be based on a single or multiple schemas or even conform on a fragment of an XML schema. XML data is stored in the binary form of a BLOB object in a table column; the XML document's size is limited to 2Gb.

Querying and Modifying XML Data To query XML instances stored in a table column as an XML datatype, SQL Server parses the binary XML data in the column using four XML datatype methods that utilise XQuery expressions to identify the subject data.

- `query()` : Extracts part of an XML instance. The XQuery expression evaluates to a list of XML nodes. The subtree rooted at each of these nodes is returned in document order. The result type is untyped XML.
- `value()` : Extracts a scalar value from an XML instance. It returns the value of the node the XQuery expression evaluates to. This value is converted to an SQL type specified as the second argument of the `value()` method.
- `exist()` : Useful for existential checks on an XML instance.
- `nodes()` : Yields instances of a special XML data type, each of which has its context set to a different node that the XQuery expression evaluates to.

The `modify()` method is used for modifying the XML data stored in a column. It allows insertions and deletions of subtrees or replacing scalar values.

Indexing XML Data SQL Server 2005 has two types of indexes for XML data stored in table columns, the so-called primary and secondary XML index.

Primary indexes apply on an XML column in a table. A primary XML index is a B+-tree-based index that shreds the whole XML data in it, storing:

- Each tag name in the XML
- The path from the root of the document to the tag
- The value of the node
- The type of the node
- The corresponding primary key of the base table

To create a primary XML index, execute the `CREATE PRIMARY XML INDEX` statement:

```
CREATE PRIMARY XML INDEX index_name
    ON table_name (xml_column_name)
```

In addition to the primary XML index, each XML column can have up to three secondary XML indexes. These are specialised XML indexes that help with particular types of XML queries, and can only be created on columns that already have a primary XML index:

- The PATH secondary XML index helps with queries that use XML path expressions.
- The VALUE secondary XML index helps with queries that search for values anywhere in the XML document.
- The PROPERTY secondary XML index helps with queries that retrieve particular object properties from within an XML document.

To create a secondary XML index, we use the CREATE XML INDEX statement:

```
CREATE XML INDEX index_name
    ON table_name (xml_column_name)
    [USING XML INDEX xml_index_name
        [FOR {VALUE|PATH|PROPERTY}]
```

7 Discussion

It is evident from the description of the different types of XML databases in the previous sections that deciding the flavour of the XML database to be used in a certain application is not easy. This Section discusses the strengths and weaknesses of each type of XML database in an effort to simplify this task.

XML-enabled databases are a good solution when it comes to XML data exchange between applications and other databases. An important consequence of using XML as a data exchange format is that an XML-enabled database will only retain information captured by the the underlying data model. For example, in the case of an XML-enabled relational database, this means that only the data itself and the hierarchical relationships among nodes are retained. All other information, including entity references, CDATA sections, comments, processing instructions, and the DTD, are ignored. Even the order in which elements appear in their parent is lost, since relational databases have no concept of order among columns or rows.

In some applications, this is considered to be an advantage, since it keeps existing data and applications intact. Adding XML functionality to the database is simply a matter of adding and configuring the software that transfers data between XML documents and the database, and there is no need to change existing data or applications. In other applications, however, such as those required by law to keep exact copies of documents, round-tripping is important, and the fact that part of the information contained in XML documents is not preserved can be considered as unacceptable.

Moreover, this model is not the most appropriate for fully and efficiently managing large XML documents, as shredding/publishing XML data to/from relational tables requires a large number of join operations.

Native XML databases are focused on applications that need to store whole XML documents, in contrast with applications that simply store data that may or may not have been in an XML format. The two fundamental reasons to opt for a native XML database is performance and space. Depending on the storing strategy adopted, a native XML database may have much better performance in retrieving and/or recomposing documents, in structural queries and full-text searches. If performance is not an issue, the second issue to consider is wasted space: the irregular structure of document-centric documents will unavoidably lead to a large number of null values in the XML-enabled database.

Another reason to prefer a native XML database is support for XML specific capabilities, such as XML query languages. However, the current shift of relational database products into the hybrid domain makes this reason obsolete.

Hybrid XML databases are usually relational database products extended with native XML support. As such, hybrid XML databases are ideal for applications which at one point stored data in relational form, but now need to move to the XML world; performing such a data transformation within a single DBMS greatly simplifies the task. The second application scenario for hybrid databases is applications where there is need for storing some data in relational form and other in XML form and, at the same time, the application needs to combine the relational and XML data. The single DBMS solution again simplifies the task by removing the necessity of writing custom code to glue different DBMSs together.

Acknowledgements

Section 4 of this document is largely based on [5], while Section 3 and Section 5 are largely based on [2].

References

[1] M. Arenas and L. Libkin. A normal form for XML documents. *ACM Transactions on Database Systems*, 29(1):195–232, 2004.

[2] R. Bourret. XML and Databases. http://www.rpbourret.com/xml/XMLAndDatabases.htm, 1999-2005.

[3] S. Dekeyser, J. Hidders, and J. Paredaens. A transaction model for XML databases. *WWW Journal*, 7(1):29–57, March 2004.

[4] D.W. Embley and W.Y. Mok. Developing XML documents with guaranteed "good" properties. In *Proc. ER'01, LNCS 2224*, 2001.

[5] B. Steegmans et. al. *XML for DB2 Information Integration.* IBM Corporation, 2004.

[6] International Organization for Standardization, Geneva. *Information processing — Text and office systems — Standard Generalized Markup Language (SGML) ISO 8879*, 1986.

[7] Patrick Lehti. Design and Implementation of a Data Manipulation Processor for an XML Query Language. Diplomarbeit, August 2001.

[8] Oracle Corp. Oracle Database 10g Release 2, XML DB Technical Overview.

[9] Oracle Corp. Oracle XML DB Developer's Guide 10g Release 2 (10.2).

[10] W. Provost. Normalizing XML, Part 1. http://www.xml.com/pub/a/2002/11/13/normalizing.html, November 2002.

[11] W. Provost. Normalizing XML, Part 2. http://www.xml.com/pub/a/2002/12/04/normalizing.html, December 2002.

[12] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, 2005.

[13] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proc. SIGMOD Int. Conference on Management of Data*, pages 413–424, 2001.

[14] W3C. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210, February 1998.

[15] W3C. Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1. http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm#AEN56, June 1998.

[16] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/, November 1999.

[17] W3C. XML Linking Language (XLink) Version 1.0. http://www.w3.org/TR/xlink/, June 2001.

[18] W3C. XML Schema Specification. http://www.w3.org/XML/Schema, May 2001.

[19] W3C. Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/2004/REC-xml11-20040204/, February 2004.

[20] W3C. XML Information Set (Second Edition). http://www.w3.org/TR/xml-infoset/, February 2004.

[21] W3C. XML Path Language (XPath) Version 2.0. http://www.w3.org/TR/xpath20/, January 2007.

[22] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, January 2007.