DB2 Information Management Software

The IBM approach to unified XML/relational databases

Contents

What is native XML storage?

The data management industry has no definitive description of the term "native XML storage." The term is used commonly throughout the industry with varying degrees of interpretation. A review of various "native XML storage" offerings reveals some common elements that most definitions adhere to:

1. *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 include the XPath Data Model, the XML Information Set (Infoset) and the models implied by the Document Object Model (DOM) and the events in Simple API for XML (SAX) 1.0.*
2. *Uses an XML document as the fundamental unit of logical storage, just as a relational database has a row in a table as its fundamental unit of logical storage.*

Although these elements are useful, an important third element is necessary to complete the definition:

3. *Uses a physical storage model that is representative of the logical model or XML document.*

The XML document must be the fundamental basis for logical modeling, logical storage and physical storage to accurately represent and render the XML document. This approach leaves no layer or portion of the data engine exempt from understanding this XML model, from the data model through the engine down to disk and back to the client. The result is a data storage model with the flexibility to handle any XML statement in any column and uniform and exceptional performance across document and collection sizes.

What options are available today?

Stand-alone XML-only database products are currently available—however, these products are only XML databases and do not include support for relational data or data models other than XML. Because these products do not offer the capability or flexibility of unified offerings from the major relational vendors, they are not covered in detail in this document.

The unified support offered by major vendors—such as, Oracle, Microsoft, Sybase and IBM—can be loosely grouped into four categories, with each vendor offering support for one of more of the following:

- *Shred, or decompose, the XML into relational or object relational form*
- *Store the XML intact in character form in a character large object (CLOB)*
- *Store the XML in encoded binary form in a binary large object (BLOB)*
- *Store the XML in a truly native repository*

Shred

Shredding involves looking at the XML data, defining a corresponding relational schema (for example, looking at parent/child relationships in the XML data and representing each child as one or more tables in a referential integrity constraint with its parent) and defining a mapping from the XML data to the relational schema. That mapping might be manually defined, programmatically defined (most frequently using XML schema as input) or defined by some combination of automatic programming followed by manual editing for fine-tuning.

Shred provides the following advantages:

- *Very good fit into existing relational environments*
- *Optimal approach for tabular data that is not required to be retained as XML*
- *Easy data updates*
- *Fast SQL searches of data, largely because the data has ceased to be XML data and has become relational data*
- *Most reporting tools require a relational database as their source*

Major disadvantages of the shred approach include the following:

- *Mapping can be complex and fragile; mappings must be predefined for every XML document that is to be stored.*
- *After the data is decomposed, it ceases to be XML data, loses any digital signature and becomes difficult and expensive to reconstruct—often requiring many joins if it must be rebuilt.*
- *Parent/child relationships inherent in the structure of the XML might require generation of values to represent foreign key values for those relationships.*
- *Shred mapping typically applies only to a single schema of a document. Permitted changes are typically restricted to those allowable in the underlying relational schema, which can be quite limited. If that schema changes, the mapping might cease to be valid or it might require a complex and difficult change process.*
- *Queries are typically allowed in SQL or through an XPath or XQuery that is somehow mapped to SQL. The mapping of XPath or XQuery to SQL can be a complex task, making it difficult to understand, diagnose and explain.*

All major vendors support the shred approach in some form.

CLOB

The CLOB storage approach is perhaps the simplest for vendors to support. It stores the XML document in a CLOB column or a column of a similar type as a character string. Although indexing technology can boost search performance, maintaining these types of indexes can be expensive.

Advantages of CLOB storage include:

- *High fidelity–preserves the original document*
- *Uniform handling of any XML, including highly volatile schemas*
- *Simple, conceptual model for users*
- *No complex mapping*
- *No need for complex joins to reconstruct the document*
- *Easy XPath and XQuery searching*

Major performance disadvantages of CLOB storage include:

- *Without additional indexing technology, the XML must be parsed for all search or query activity, which is prohibitively expensive*
- *Retrieval of portions of a document is expensive*
- *Updates are expensive, requiring the entire document to be rewritten*
- *Costs increase as the document grows in size*

All major vendors support the CLOB approach in some form.

BLOB (pseudo native)

A BLOB-based storage approach is conceptually very similar to CLOB, but instead of storing the XML data as a preparsed string, BLOB stores it in a proprietary post-parse binary representation. This approach is sometimes called *pseudo native*, because the data representation remains in XML within the BLOB. However, XML data is still stored within a BLOB in relational models.

Because of its similarity to CLOB, the BLOB approach shares many CLOB advantages and resolves several disadvantages. The advantages and disadvantages of the BLOB approach depend on the binary encoding used within the BLOB—in general, BLOB performance can be quite good depending on the indexing technology used. However, in some cases performance problems can exist because the underlying storage for a document is virtualized as a single contiguous byte range (because of the BLOB approach). In particular:

- *Updating can require the entire document to be rewritten (for example, if the first node written in the stream grows by 1 byte)*
- *Updating can require the entire document to be locked*
- *Indexing is often based on byte offset into the BLOB, so any update requires updating every index entry*
- *Access to portions of the document might require the entire document to be read from disk (for example, reading the stream until the sought portion of the document is encountered or having to read backward if the query must navigate up the ancestor axis)*

True native

True native storage holds the post-parsed data on disk, enabling individual nodes of the data model to be stored on disk independently—that is, not as a stream—and then interconnected. True native storage provides the advantages of BLOB and CLOB, but resolves the remaining performance issues that are associated with BLOB because the document storage is not virtualized as a single contiguous byte range. The storage for the entire set of documents is virtualized as a contiguous byte range, but individual nodes can be relocated in this range with minimal impact on other nodes and indexing. Therefore, true native XML databases offer the following benefits:

- *Any update affects only the nodes that are being changed and their immediate ancestor, siblings and descendants*
- *Access to any portion of the document can be direct, without requiring a read of the whole document*
- *Indexing is based on collections rather than on byte offset into a document-level byte stream; therefore, index entries must be changed only for relocated nodes, not for all the nodes that follow in an in-order traversal*
- *Because the fundamental unit of storage is a node, concurrency control is possible at the subdocument level*

The IBM approach

The approach IBM has taken is to support both shredded and true native storage. Support for shredding is important because XML can be used to feed existing relational schemas.

Since documents can grow large and will be updatable in many cases, the advantages of non-BLOB storage for XML documents, which include storing at the node level of granularity instead of at the document level, are significant.

Storage

While interacting with IBM's native XML support, the abstraction shown is a column of type XML in a relational table. This column has no maximum length, unlike CLOB and BLOB implementations, and no mandatory constraining XML schema. Any well-formed XML statement can be inserted into that column. Therefore, the following statement is a valid table definition:

```
Create table Foo (C1 int, C2 xml)
```

A table is not limited to a single column of any given type, so the following statement is equally valid:

```
Create table Foo2 (C1 int, C2 xml, C3 xml, C4 xml))
```

Drilling past this abstraction into the physical storage layer, the primary storage unit in the IBM implementation is a node. A node exists on a page along with other nodes either from the same or different documents. Each node is linked not only to its parent, but also to its children. As a consequence, after the database engine is positioned on a node, the cost of navigating to its parent, siblings or children is quite efficient, operating at little more than pointer traversal speeds as long as the next referenced node is on the same page. Furthermore, nodes can grow or shrink in size, or they can be relocated to other pages without rewriting the entire document.

Indexing

In the IBM implementation, indexes are created on columns of type XML based on path expressions—a subset of XPath that does not contain predicates, among other things. When creating an index, it is possible to specify what paths to index and what type. Any nodes that match the path expression or the set of path expressions in XML that is stored in that column are indexed, and the index points directly to the node in storage that is linked to its parent and children for fast navigation.

For example, to index names of U.S. states in which tourist attractions are located, an index specification similar to the following statement might be used:

```
Create Index IX1 on Foo(C2) generate keys using '/attraction/
  state/text()' as char(2)
```

Naturally, indexes can be created on multiple path expressions on any given column of type XML, making the following statements also valid:

```
Create Index IX1 on Foo(C2) generate keys using '/attraction/
  state/text()' as char(2)

Create Index IX2 on Foo(C2) generate keys using '/attraction/
  zip/text()' as double
```

Furthermore, path expressions can include both wildcards and descendant-or-self axis traversal, so the following statements are also valid:

```
Create Index IX3 on Foo(C2) generate keys using '/*:id/text()' as
  double

Create Index IX4 on Foo(C2) generate keys using '/attraction/
  @name' as varchar(20)

Create Index IX4 on Foo(C2) generate keys using '//age/text()' as
  double
```

Query

The IBM implementation has no stand-alone XQuery or XPath processor. The basic XQuery and XPath primitives are built directly into the data engine. The query compiler itself is bilingual, having two interoperating query language parsers—one for SQL and the other for XQuery—to generate a new variation of the Query Graph Model[1] designed to process not only relational data, but also XML data. Because the resultant intermediate query representation is language-neutral, XQuery, SQL and combinations of XQuery and SQL compile at the same intermediate representation, go through the same rewrites and transformation, are optimized in a similar manner and generate similar executables. This process results in optimal and interoperating query plans regardless of the language used to specify them.

Because the two parsers interoperate, it is possible to mix SQL and XQuery in the same statement. For example, the following demonstrates a simple XQuery statement within SQL:

```
Select Xmlquery('for $a in $b return $/name' binding C2 as B)
  from Foo
```

The following demonstrates a simple XQuery statement within XQuery.

```
for $a in db2-fn:sqlquery('select C2 from Foo where C1=12')
  return $a/gronk
```

Deeper levels of nesting (SQL within XQuery in which SQL itself contains nested XQuery) is supported and can also be used in views and more complex statements.

[1] H. Pirahesh, J. M. Hellerstein, and W. Hasan. *Extensible/rule-based query rewrite optimization in starburst*. MOD, 1992.

Why the IBM approach is effective

The effectiveness of the IBM approach to data management is based on the richness and depth of the implementation and a deep understanding of XML and support for it as a data model from the client through the engine down to the disk and backup. No layer or portion of the data engine is exempt from this understanding, resulting in flexibility in handling any XML statement in any column and uniform and exceptional performance across document and collection sizes.

Flexibility

The IBM approach offers excellent storage flexibility relative to a shredded approach—there is no need to predefine XML schema, limit documents to a given schema or provide any mapping between XML and relational models. Any XML statement can be inserted into any column of type XML and a column can be restricted to a particular schema but—but the key is that this is not required.

BLOB/CLOB-based storage can offer similar flexibility, but in many cases does not. Some vendors with BLOB/CLOB-based approaches still require documents to match a single schema. While it can be argued that specification of a schema is desirable because it imposes a much stronger type-checking system, it is not necessary to require a schema in all cases.

Performance

Relative to a shredded solution, the principal advantage of the IBM approach to data management lies in the elimination of recomposition costs—the cost of the joins and other processing necessary to reconstitute the document from the tables into which it was decomposed. In the case of complex documents, these costs can be very significant.

When compared to CLOB-based approaches, the IBM approach eliminates the need for parsing the XML document at query time. Given XML parsing costs, any CLOB-based approach will be unusable if any form of "search into" the document—that is, parsing—is necessary. CLOB should be considered only when the usage models, now and forever, are expected to be whole document insertion, search by purely relational attributes and whole document retrieval. Any time "search into" is required, CLOB will be a failure.

Compared to BLOB-based approaches, the IBM approach provides more consistent behavior as the size of documents increases or when the amount of data to access is a small percentage of the total document size. This is because the entire BLOB must typically be retrieved to process the query. For example, when the BLOB in this approach contains a SAX-like event stream as the binary representation or the query involves a parent or ancestor axis that must be traversed to satisfy the query, the whole BLOB must be retrieved to process the query.

Summary

IBM provides a truly native unified XML/relational database, supporting the XML data model from the client through the database down to the disk and back again. By deeply implementing XML into a database engine that previously was purely relational, IBM offers superior flexibility and performance relative to other offerings.

IBM Corporation
IBM Software Group
Route 100
Somers, NY 10589
U.S.A.

Produced in the United States
March 2005

GC34-2496-00