XML Databases

Contents

Michael Kay
Software AG
March 2003

This paper examines the role of XML databases. It looks at the requirements for storing XML, the retrieval and query requirements, and the different approaches that are possible. In particular, it looks at the situations in which a native XML database such as Software AG's Tamino XML Server offers advantages over other methods of storing XML documents.

Introduction: Why is XML popular?

XML was developed initially as a way of storing documents on the Web, in a form that allows the information content of the document to be separated cleanly from the presentation details. Many people were finding that it was very difficult to manage the content of their websites in HTML, because this separation was missing. This made it difficult to repurpose the information for a different audience or for different display devices. Equally, the alternative approach of storing all the information content in a relational database and adding an HTML wrapper for displaying the information worked only for rather rigidly structured information such as price lists: It didn't give the flexibility that was needed to construct sites containing information such as holiday brochures, weather forecasts, or product catalogs.

So the killer application that made XML so successful was content management. It quickly became clear that XML also met another need, namely the need for flexible syntax to exchange data between applications. There have been plenty of other approaches to this problem, but what makes XML special is its simplicity, broad support, and low implementation cost. As more and more enterprises take advantage of the connectivity offered by the Internet to exchange information not only within their own organization but also to integrate their business processes with their suppliers and customers across enterprise boundaries, the advantages of a simple yet extensible, platform-neutral syntax have become widely recognized, and XML was there at the right time to meet this need.

As a result, XML is now widely used for information interchange. Take the UK Inland Revenue as an example: Taxpayers have a wide choice of off-the-shelf software to prepare their tax returns, but all these software packages output the data using the same XML schema for electronic filing, which means that the Inland Revenue can accept the data without worrying about which application was used to prepare the tax return.

The key features of XML that have led to its widespread adoption are:

- It is simple enough that parsers became available very quickly and were distributed in most cases as free or open-source software. This meant that the decision to use XML could be made in many organizations without a lengthy investment review.
- It largely gets rid of low-level character encoding problems by adopting Unicode as its single character repertoire, allowing worldwide deployment.
- It is sufficiently flexible to handle both narrative documents intended for a human readership (for example, Web pages) and arbitrary hierarchic data structures intended to be processed by applications, as well as combinations of the two.
- The syntax itself is human-readable, allowing simple documents to be created or read using nothing more than a standard text editor.
- There is no turf war: XML is supported by the entire IT industry, and the products offered by different vendors are highly interoperable.
- These factors mean that today, XML is widely deployed not only for the original application area of Web content management, but for a large variety of other applications. In many of these applications, the information needs to be stored somewhere, and this is where the requirement for XML databases comes from.

The need to store XML

As the previous section shows, XML was not initially developed as a model for storing information. But once an organization has decided to adopt XML to underpin a key business process, the need to store the information (and, of course, to retrieve it later) soon follows.

There are essentially three ways that this can be done:

- Store XML as files. In this approach, the XML document in its raw textual form is treated as any other text file, and is stored as such, either as a file in an operating system file store, or in some kind of general-purpose document management system, or perhaps as a "BLOB[1]" or "CLOB[2]" in a relational database. Some kind of external index is maintained to enable these files to be subsequently retrieved: Perhaps the only retrieval mechanism is to give each file a hierarchical name.
- Extract the data. In this approach, the XML document is parsed and the information it contains is extracted into some kind of database, typically a relational database. The original XML document is not retained. When the information is subsequently required, a new XML document is constructed by assembling all the relevant information items from the database.
- Use an XML database. With this approach, the XML document is stored in a database that understands the structure of the XML document and is able to perform queries and retrieve the data taking advantage of knowledge of its XML structure.

In this paper, we are concentrating on the third approach. The other two approaches cannot be completely dismissed, and in fact there are many situations where they might be appropriate. But they do have considerable disadvantages.

The main disadvantage of the file-based (or BLOB-based) approach is that the structured information inside the XML document cannot be used for retrieval purposes. Separate indexes need to be created to locate the required document based on one or more keys such as a date or employee number. If these keys are not known, it is very hard to locate the information. In addition, the only thing that can be retrieved is the original document.

It isn't possible, for example, to do an aggregation query, such as "Which resorts have an average February temperature above 25°C?" The documents need to be individually retrieved, and the information extracted from them, probably by hand. This means that the value of having the XML markup in the documents is not being exploited, and the information asset is not being used to its full potential.

Extracting the data into a relational database has other disadvantages. Firstly, it only works where the data fits comfortably into rows and columns. This doesn't work well for narrative documents such as news reports, and it doesn't work well for complex data structures such as medical records. Secondly, there is no record of the original document, which might well be needed for archiving purposes or for legal traceability requirements.

[1] BLOB = binary large object, see: http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci213817,00.html
[2] CLOB = character large object; a variable-length character string, can be up to several gigabytes long

This means that it might be necessary to keep the original document as well as the extracted data, which can only add to the information management problem. In addition, extracting the data from one document into thousands of small pieces, and then reassembling these to recreate the XML document when required, is a very inefficient process. Of course, where the primary purpose of the XML message is to send information to an application that already maintains its information in a relational database, this cost is necessary and appropriate. But where the primary aim is to record what was in the XML message so that it is available at a later time, taking the message apart and storing all the primitive facts separately is a quite unnecessary overhead.

Where XML is used for purely narrative documents, with very little structure, file-based storage might work. Where it is used simply as an interchange mechanism for straightforward tabular data, with a very rigid structure, relational storage might work. But the strength of XML is that it isn't limited to these two extremes.

In the real world, most information is semi-structured: consider a CV (résumé), an accident report, an invoice, a software bug report, an insurance claim, a description of a CD offered for sale on the Internet. Traditionally, in the IT world, we have handled the structured part of the information and the unstructured part separately, using different technologies and often using different business processes.

The Internet has changed that: Both parts now need to be handled as a whole, and a major reason for the success of XML is that it is the first mainstream technology that can handle the whole spectrum of information from the highly structured to the totally unstructured. Semi-structured data has become the norm, and the limitations of technologies that can handle only one end of the spectrum have become painfully apparent.

The only reason to store anything is so that the information can later be retrieved. So in the next section, we'll look at the query requirements for XML documents.

Query requirements

Most of the requirements for accessing XML information can be conveniently classified under one of three headings:

- Get me the documents
- Give me the facts
- Tell me about X

By **Get me the documents** we mean queries whose aim is to locate one or more documents. The documents that are returned are identical to documents that were stored in the database at some time in the past. The information used to retrieve the document may be something as simple as a unique reference, or it may be some combination of properties that the document must possess. For example: "Give me the most recent appraisal for employee E12345."

By **Give me the facts** we mean queries that extract factual information from documents. The required information may be all in one document, or it may be spread across many documents. For example: "When was the last time employee E12345 was recommended for promotion?" Or: "How many claims were made last year by policy-holders in Durham, and what was their average value?"

By **Tell me about X** we mean information retrieval queries of the kind that people submit to Web search engines. However much we take care to add markup to narrative documents, there will always be cases where the only way to find relevant documents is to search the text. For example, the best way to find an employee with connections in Peru might simply be to search for the word Peru, appearing anywhere in any part of the document. Searches that analyze the textual content and also take advantage of contextual information based on the XML markup can be especially useful.

A characteristic of this kind of query is that there is no right answer. It's up to the search engine to use as much intelligence as it can to find the documents that are most relevant to the user's enquiry.

Traditionally these three kinds of queries have been handled by different kinds of storage software. **Get me the documents** can be done using file-based storage with simple keyword indexing, so long as the attributes that will be used for retrieval are known in advance (WebDAV is a protocol that implements this idea, and is used by many content manage-

ment applications). **Give me the facts** is the traditional domain of relational databases: Facts are extracted from the source documents and stored separately, so that they can be searched and aggregated. **Tell me about X** is the domain of free text retrieval packages and Internet search engines.

Sometimes one of these patterns of enquiry dominates, in which case it makes sense to choose software that specializes in that kind of enquiry. But because XML is semi-structured, a general-purpose XML database needs to be good at dealing with all three kinds. The Worldwide Web Consortium (W3C), which was responsible for the development of the XML specification, is also defining a query language called XQuery for accessing XML databases. XQuery is designed to handle all three kinds of enquiry, though at present the development of free-text capabilities is lagging behind the syntax for more structured queries.

Software AG has played an active part in the development of W3C XQuery since its inception – as members of the working group, editors of the specification and developers of early implementations. A prototype implementation of W3C XQuery called QuiP can be downloaded from the Software AG website, and although the W3C specification is not yet finalized, features of the language have been incorporated into the current production release of Tamino XML Server (Version 4.1). In fact, the previously introduced Tamino X-Query language is an extension of the simpler W3C XPath language that is still available for use in the current Tamino XML Server release.

Database requirements

As well as supporting the kinds of query that people expect to perform on stored XML data, an XML database system must meet many other requirements. Many of these are not specific to XML. For example, a database must:

- Support the management of schemas to define the data structure, and the validation of input according to those schemas
- Provide mechanisms to add, modify or delete content
- Offer facilities for multi-user access, transaction-based isolation and deadlock detection, backup, recovery and replication
- Provide bulk data import and export capabilities
- Allow the physical storage of the database to be optimized, for example by allowing user control over the creation of indexes and the allocation of disk space and other resources
- Optimize queries to provide the most efficient possible execution, using the indexes and other access paths that have been made available

Because the development of an industrial-strength database system is a major engineering investment, many so-called XML databases are actually layered on top of databases that were originally designed for a different purpose: typically a relational database or an object database. Software AG has taken a somewhat different approach.

Tamino XML Server is designed from the ground up as a native XML database, meaning that it has the built-in capability to store XML documents as they are, without conversion to other storage formats. This ground-up development does not mean that Software AG reinvented the wheel in order to provide Tamino XML Server with high-performance transactional capabilities, etc., required for business-critical use. Tamino XML Server incorporates the knowledge and many years of experience that Software AG gained in developing such data-model-independent functions as transaction management and backup/recovery, and as such it maintains its competitive advantage against similar systems.

In fact, Software AG applies strong discipline to the management of reuse across its entire product line, so that developments to a component that are made to meet the requirements of a new product will also be fed back and benefit the users of older products.

This way, Software AG has not only reduced the cost of developing Tamino XML Server and accelerated its introduction to the market, but it has also taken advantage of the reliability of these components achieved through years of exposure in the field.

What is a native XML database?

Software AG invented the term "native XML database" to explain how Tamino XML Server differed from other ways of storing XML. Since then, the term has become widely used in the industry, though not always with exactly the meaning that Software AG had in mind when the phrase was coined.

[3] http://www.rpbourret.com/xml/XMLAndDatabases.htm#nativedb

The essence of the term is captured in a quote from Ronald Bourret[3], who has written extensively on XML databases. This is how he explains the concept:

> "Native XML databases are databases designed especially to store XML documents. Like other databases, they support features like transactions, security, multi-user access, programmatic APIs, query languages and so on. The only difference from other databases is that their internal model is based on XML and not something else, such as the relational model."

The focus in this definition is not on specific features, but on the fact that storage and retrieval of XML was the central objective of the product designers.

When you look at a checklist of product features, it can be difficult to distinguish a relational database that also supports XML data from an XML database that also supports relational data. But below the surface, the architectural differences that result from the difference in design perspectives are immense.

In particular, the heart of any database product is the query execution engine, which constructs, optimizes, and then executes a query execution plan to deliver the results of a user query. The design of the primitive operators that make up the query execution plan reflect the operational algebra that underpins the query language, whose formal semantics are in turn based on the invariants of a particular data model.

In a relational engine, these operators are the well-known relational primitives such as restriction, projection and join. An XML engine has a different set of operators, which are better suited to the recursive tree traversals required for efficient execution of path expressions. The different set of operators is needed because the basic data model for XML (that is, a recursive tree structure) is fundamentally different from the rows-and-columns data model of SQL.

Of course, XML queries can be mapped into operations in the relational algebra, just as SQL queries can be mapped to operations in a tree-based algebra. But the result is very unlikely to be optimal. (This is particularly true when mapping an XML query language to relational operators, because of the difficulty in handling recursive queries in SQL. This problem, which is sometimes referred to as the parts explosion problem, has been known since the 1970s, and reflects a fundamental limitation in the mathematical power of the first-order predicate calculus on which the relational model is based.)

So this gives us a more technical definition of what we mean by a Native XML Database: it is a database whose core query execution engine implements an algebra designed to perform operations on trees that represent XML documents. Why is this important?

The answer is performance and scalability.

Consider a simple XML Query:

input()[//section/title = 'US Pricing Information']

This query selects all documents in the input collection that have a section element (at any depth) whose title is "US Pricing Information."

In a large document collection, the only way any database can deliver acceptable performance on this query is if the data has been indexed in some way.

Tamino XML Server makes use of indexing structures that are specifically designed for XML (some of the ideas are even patented).

- One kind of index is a **structure index**: this allows rapid selection of documents that have a section element containing a title element.
- Another kind of index is a **value index**: this allows rapid selection of documents that have a particular value in the title element.
- Put **these two indexes together**, and the query can execute very efficiently, even on a database containing hundreds of gigabytes of data. This works because the query optimizer understands the meaning of path expressions, and can therefore readily detect which indexes are of use.

In contrast, a system whose core engine is relational has to translate this query into a set of operations in the relational algebra. This may be quite a complex task, depending on how the XML model has actually been mapped. The search for a section element at any depth may be particularly difficult. The final result is often a complex set of join conditions, which need to be evaluated using indexes that were designed for relational tables, not for tree-structured XML.

Of course, some relational vendors, if they take XML seriously, will add extra operators to their engine to reflect this, and may add extra indexing mechanisms as well. But the fact remains, there will always be a difference between an engine that was designed for XML from the ground up, and one that has had XML bolted on as an afterthought.

Designing an XML database

There is a wealth of experience, captured in books, methods, and tools, for designing relational databases. This experience has evolved over thirty years and the basic principles are now taught in every college course on the subject. XML databases are a much more recent phenomenon, and the database designer therefore has to work much more from first principles.

The traditional approach to database design follows the steps:

1. Collect information about the application domain, by interviewing users, collecting process descriptions and studying existing applications.

2. Build a model of the objects, attributes and relationships in the domain, using a notation such as UML.

3. Translate this into a relational schema by applying the principles of normalization.

4. Refine this design as necessary to ensure that the performance requirements of the application are met.

In principle, a similar approach can be followed for an XML database, with the exception of step three, where the model is translated into XML elements and attributes instead of relational tables and columns.

But there is one big difference. In many cases, the XML documents are designed primarily for information interchange, not for holding persistent data. The designer must therefore decide whether the XML database is to hold the documents in the form they arrived (essentially, a repository of messages) or whether to refactor the document contents for query purposes. The decision depends very much on the particular application requirements.

The question "What is a document?" is sometimes very easy to answer, and sometimes remarkably difficult. This decision is easy when it is built into the application requirements, for example "I want to store all the product descriptions." It is less easy where the notion of a document is not inherent in the requirements, or where the boundaries between documents are arbitrary, for example "I want to store logs of all the online transactions."

The concept of a document is one that users can easily relate to, for the simple reason that documents have been used in human communication for thousands of years. Therefore, choosing what goes into one document is something that deserves some care. It is always possible to assemble new documents as the result of a query, but as we saw earlier, the "Find me the documents" queries are probably the ones that are easiest for users to formulate and easiest for the system to execute efficiently. The document is also a convenient unit for management of information, for example updating and deletion of information will be easiest if it is done at the document level.

It's therefore generally a mistake to think of the whole database (or of a whole table) as being one document. Documents in an XML database should ideally relate to individual things or events that are familiar to people in the user community: A product, an inspection report, a job application. Sometimes it makes sense to assemble related information into a single document, for example, to collect together all the information relating to one medical episode, even though it may arrive in small pieces over a period of time. A good guide is: What is the packaging of information that is most often going to be requested in a single query?

Another factor that requires some thought is the modeling of relationships. Unlike the relational model, which essentially only offers one way to represent relationships (the primary key / foreign key combination), XML offers a bewildering variety of techniques. These range from the use of hierarchic nesting, to ID/IDREF pointers, and URLs and XPointer hyperlinks to reference one document to another. And of course relational-style foreign keys are also available as an option. This richness derives from the variety of mechanisms used in written documents, but some of these techniques are much more amenable to database queries than others.

The discipline of XML database design is still evolving. In this section we discussed a few ideas that reflect Software AG's growing experience of what works well and what doesn't.

The real message of this section, however, is that to succeed, you need more than good technology: You also need to understand how to apply it to your particular problems. You've guessed what the sales message is: You need a product that is backed by the experience and service delivery capabilities of a company that is not only a mature industry player, but has also committed itself 100 percent to the development of its XML capability.

Why Tamino XML Server?

We've discussed in this paper the different ways that XML can be stored, and the advantages that come from storing documents in a native XML database rather than saving it either as a text file or broken up into many separate facts in a relational database.

Once you have made the decision to use an XML database, the decision to use Tamino XML Server usually follows quite easily. Software AG is the most experienced vendor in the field, with a host of independent product endorsements and awards, a mature product that always compares well in competitive benchmarks and feature comparisons, and the capability to deliver the services that turn the technology into business success in Europe, North America, and around the world.

What are the actual technical factors that have made the product so well respected? There are basically two: the core engine, and the surrounding integration tools.

The core engine is:

- a thoroughly robust and scalable database engine, with all the expected capabilities in areas such as transaction management, resource management, security, and backup/recovery
- built from the ground up to handle XML as its core data model, XML queries as the core query language, and XML Schema as the core schema language, and with index structures, space management, and optimization algorithms all custom-designed for XML
- built for the Internet. Unlike most relational systems, which were originally designed in the client-server era and often optimized for in-house departmental applications with a few hundred users, Tamino was designed in the Internet age and its core architecture relies on the use of Internet protocols. This has a significant effect on the way user authentication, security, and session management operates, and means that Tamino fits very naturally into Web-based application frameworks such as Enterprise JavaBeans severs and Microsoft's .NET environment.

Around this core, Software AG has created a suite of tools and integration options that make it easy to connect Tamino's XML engine to a wide variety of other information sources, applications, and development environments. In fact, it is because this outer layer adds so much value to the Tamino offering that we refer to the product as an XML Server, not merely an XML database.

Included in this outer layer are:

- Tamino X-Node and the Data Map, which allow parts of the XML data model to be transparently mapped to external data sources, such as relational databases
- Tamino X-Tension, which makes it possible to write user-defined server extensions. These add custom logic inside the database, to implement application functions such as data validation, event notification, and linking to external applications via EntireX, Software AG's XML-based middleware for application integration
- Management tools integrated into Software AG's System Management Hub, which provides a central focus for Web-based remote management of all Tamino resources and other Software AG products
- APIs for Java and .NET, each based on the native data access conventions of the specific environment
- Tamino X-Plorer, an easy-to-use graphical front-end that allows the database contents to be browsed, queried, and updated
- Tamino WebDAV Server, a WebDAV interface, making Tamino directly accessible to Content Management applications that use this increasingly popular protocol
- XSLT transformation interface, making it easy to define XSLT transformations that render the XML results of a query as HTML for display to the user

- Schema designer, a graphical tool for designing the XML Schemas that define the contents of a Tamino database, as well as the definitions that control the mapping of the logical XML model to physical storage

In addition, Tamino offers an increasing range of third-party client tools that further increase the ability to integrate Tamino into a wide variety of application environments.

Summary

XML has become a huge success both in its original application area of Web content management, and also as the preferred vehicle for application-to-application data interchange. The success of XML means that there is an increasing requirement to store XML documents for subsequent retrieval.

Although it is possible to store XML in the form of text files, or to extract the data from structured XML and store it relationally, neither solution exploits the potential of XML fully, in particular its ability to hold semi-structured information. This has created a market need for XML databases. The fact that XML is radically different from other data models means that there are particular advantages in using a native XML database, designed from the ground up to support XML data and XML queries, rather than adapting a database that was originally designed for a quite different purpose.

Software AG's Tamino XML Server is the market leader in its field, and this paper shows some of the technical reasons behind this position.

Software AG
Corporate Headquarters
Uhlandstraße 12
64297 Darmstadt, Germany
Tel: +49 - 6151 - 92 - 0
Fax: +49 - 6151 - 92 - 1191
www.softwareag.com

INS/WP17E0403 051

software AG
THE XML COMPANY