Native XML Databases in the Real World

Ronald **Bourret**

Abstract

Based on a broad survey of native XML database companies, this presentation describes how native XML databases are being used in the real world, including descriptions of why native XML databases succeeded and relational and other technologies failed.

Table of Contents

1. Introduction

When native XML databases appeared on the heels of the XML 1.0 recommendation, most people weren't sure what to make of them. Were they a replacement for relational databases or a return to hierarchical databases? No, said the vendors. They were designed to manage large numbers of XML documents. But since many XML documents could be mapped to relational tables, the only obvious targets were document-centric XML, which lacked structure, and catalogs, which had deep hierarchies. Beyond that, it wasn't clear what they could be used for.

2. Got XML?

So what are the use cases for native XML databases? As John Merrells, one of the developers of Sleepycat Software's Berkeley DB XML, waggishly put it, there is only one use case, and that is simply, “Got XML?” In other words, if you have more than a handful of XML documents that you need to store, you should store them in a native XML database. The reasons are the same as for storing data in any other database -- ease of management, enhanced query performance, concurrent access, transactional safety, security, and so on.

Other vendors are more specific: native XML databases are used in a wide number of fields -- from genetics to health care to insurance -- and technologies -- from data integration to messaging to Web sites. This article describes the most common use cases and is based on discussions with roughly half of the native XML database vendors, as well as a handful of customers. It does not attempt to determine the size of the current native XML database market -- a task left to the market analysts -- nor does it attempt to describe the capabilities of individual native XML databases, which vary significantly.

While most real-world uses of native XML databases do not fit cleanly into any single category, it is possible to characterize them in terms of a limited number of use cases. The most popular of these are storing and querying document-centric XML, integrating data, and storing and querying semi-structured data. Native XML databases are used in these cases because the data involved does not easily fit the relational data model, while it does fit the XML data model.

Other common use cases for native XML databases include managing long-running transactions, handling rapidly evolving schemas, working with very large documents, querying hierarchical data, and running Web sites.

3. A very quick review of native XML databases

There are two different ways to "store" XML documents in a database. The first is to map the document's schema to a database schema and transfer data according to that mapping. The second is to use a fixed set of structures that can store any XML document. To understand the difference, consider how a sales order document might be stored in a relational database. The first method uses a set of tables designed specifically for storing sales orders: Orders, Items, Customers, Parts, and so on. The second method uses a set of tables designed to hold arbitrary XML documents: Elements, Attributes, Text, and so on.

Databases that support the first method are called "XML-enabled databases". (Such functionality is also provided by third-party middleware, which is said to "XML-enable" a database.) Databases that support the second method are called "native XML databases". (While our example shows how to build a native XML database on top of a relational database, virtually all native XML databases are built from scratch.) A more theoretically correct way to say this is that XML-enabled databases have their own data model -- relational, hierarchical, object-oriented -- and map instances of the XML data model to instances of their data model. Native XML databases use the XML data model directly.

XML-enabled databases are useful when publishing existing data as XML or importing data from an XML document into an existing database. However, they are not a good way to store complete XML documents. The reason is that they store data and hierarchy but discard everything else: document identity, sibling order, comments, processing instructions, and so on. In addition, because they require design-time mapping of schemas, they cannot store documents

whose schema is not known at design time. Native XML databases, on the other hand, store complete documents and can store any document, regardless of schema.

For a complete discussion of native XML databases, see Introduction to Native XML Databases [http://www.xml.com/pub/a/2001/10/31/nativexmldb.html] or XML and Databases [http://www.rpbourret.com/xml/XMLAndDatabases.htm#nativedb]. In this paper, we will only mention two more terms. In a native XML database, a "document" is the fundamental unit of storage, equivalent to a row in a relational database. While this might be a document in the traditional sense, such as a chapter in a book, it might simply be a set of related data, such as the sequence of a gene, a list of known mutations to the gene, and a list of papers about that gene. The latter sense is important for things like semi-structured data. A "collection" is a set of related documents and plays a role similar to that of a table in a relational database or a directory in a file system.

And now, on to the use cases.

4. Storing and querying document-centric XML

As might be guessed, the most common use case for native XML databases is storing and querying document-centric XML. Of the vendors surveyed, all but one listed it as a use case, and many noted it was their most common use case.

4.1. Documents in the real world

Let's start with some examples of how native XML databases are used with document-centric XML in the real world:

- Elsevier Science, a publisher of scientific, technical, and medical information, uses Mark Logic's Content Interaction Server to manage more than two terabytes of data: five million full-text journal articles, 60 million citations and abstracts, thousands of complete books, and five thousand informational pamphlets. The system is used to search and transform documents. Raining Data's TigerLogic XML Data Management Server is used in similar fashion by large scientific publishing companies.

- A variety of content management applications are built on top of IXIASOFT's TEXTML Server. For example, third-party editorial systems such as AILINK, Eurocortex, KnowledgeView, Protec, Rosebud, Modulo, and PCI use it to manage production, archive content, or both, and are “installed at most major publishing companies in the US and in Europe.” General Dynamics uses TEXTML Server in AFCV, a browser used to view technical manuals for various aircraft, including AWACS surveillance airplanes and the F-16.

- The Las Vegas Sun uses Snapbridge's Cross Media Server (a content management system built on its FDX XML Server native XML database) to manage content for its day-to-day operations as well as its Web site. The system holds more than 750,000 items (XML documents, images, and PDF documents), comprising tens of gigabytes of data.

- An unnamed bank uses Xyleme Zone Server as the basis for an equity research portal, which is composed of multiple sub-portals. It is used by roughly 10,000 employees inside the bank and more than 30,000 customers outside the bank, who perform both contextual and full-text searches. Analysts add thousands of documents daily. As these are loaded, a separate engine queries them and notifies users of any changes of interest. Latency between loading and user notification is “a couple of minutes.”

- The Tasmanian Government uses TeraText DBS to power a Web site that allows users to track Tasmanian legislation. A single piece of legislation is stored as a series of time-stamped fragments; this allows users to track changes to the legislation over time. In addition, links in the documents refer to fragments also stored in the database, allowing them to be implemented as queries on the database.

- Autodesk stores software manuals in Idiom's GEP, a content management system built on top of X-Hive/DB. Content is stored as reusable components, from which manuals are built. Manuals are published in a variety of

formats (HTML, PDF, CHM, print) and over 30 languages. X-Hive/DB's versioning mechanism is used to track changes, which reduces translation time. The system also relies heavily on XQuery and XLink.

- Amirsys uses Ipedo XML Store to manage descriptions of radiology diagnostic cases, image data, and data used to drive a document editor. The editor is designed for writing books about radiology and contains features such as queries on existing descriptions and the ability to insert images. Books are published using Apache FOP. In addition, managers can query across documents, such as to track the progress of authors or check that elements are used in the same way by all authors.

- Le Monde uses Xyleme Zone Server to manage an archive holding more than 800,000 documents and using 6 gigabytes of storage. The archive is used by employees, partners, and customers.

- The US Navy uses the GEMt content management system, which is built on Tamino, to store roughly 100,000 “volumes of on-board binders” for systems used on aircraft carriers.

- The XML Transactional DOM from Ontonet is used to manage data about an archive of medieval manuscripts: a thousand-page book cataloging the archive, scholarly markup about the archive, and METS metadata about each manuscript in the archive.

Native XML databases are also used to store a variety of other types of documents, such as contracts, case law, drug information sheets, insurance claims, e-forms, product support procedures, classified ads, and intelligence documents.

4.2. Inside the applications

Applications use document-centric documents in a variety of ways, but most use falls into four broad categories: managing documents, finding documents, retrieving information, and reusing content.

4.2.1. Managing documents

Many applications need to store and retrieve documents. For example, a content management system might add a new document to its data store or a Web server might retrieve a document for display. At the level of a native XML database, managing documents is quite simple. Applications either submit documents to be stored or request documents to be retrieved; the latter functionality uses a document ID, which is usually assigned by the user. (More complex functionality, such as the versioning, check-in/check-out, and workflow facilities found in content management systems, is usually built on top of the database.)

4.2.2. Finding documents

A wide class of applications needs to find whole documents. For example, a Web portal might allow users to search for all documents about a particular company and a content management system might allow users to find all documents relating to a certain part.

The least complex way to search for documents is with full-text searches. In native XML databases, these are "XML-aware". That is, they distinguish between content (which is searched) and markup (which is not).

More complex are "structured queries", which can query markup, text, or both. (XPath and XQuery are examples of structured query languages; native XML databases support a number of proprietary languages as well.) For example, consider the following queries:

- Find all books that Maria Lopez wrote:

```
for $b in collection("books")
```

```
where $b//Author="Maria Lopez"
return $b
```

- Find all articles written after June 1, 2004 with the words "presidential election" in the title:

```
for $a in collection("articles")
where $a//Date > 2004-06-01 and
      fn:contains($a//Title, "presidential election")
return $a
```

- Find all procedures with more than seven steps:

```
for $p in collection("procedures")
let $s := $p//Step
where fn:count($s) > 7
return $p
```

Although these queries are relatively simple, none can be satisfied by a full-text search: the first two queries restrict the search to certain sections of the document and the third doesn't even query the text. It is also interesting to note that these queries do not require all documents to use the same schema -- they only require documents to contain certain common elements that have roughly the same meaning.

4.2.3. Retrieving information

Although documents contain useful data, they haven't traditionally been used as a source of data. XML and XML query languages make that possible. For example, consider the following queries:

- From a procedure for synthesizing a compound, list the required chemicals:

```
for $p in collection("procedures")
return
   <Chemicals procedure="{$p/Title}">
      {$p//Chemical}
   </Chemicals>
```

- Find maintenance procedures for a specific spare part of a specific airplane with a specific effectivity:

```
let $today = fn:current-date()
for $proc in collection("maintenance_docs")//Procedure
let $p = $proc//Part
where $p = "AX723" and $p//AppliesTo = "Model 1023i"
      and fn:date-greater-than($today, $p//EffectivityStart)
      and fn:date-less-than($today, $p//EffectivityEnd)
return $proc
```

- Create a table of contents or index from a book document.

These queries are fundamentally different from those that return whole documents to be read or modified. Instead, they answer questions, create reports, or construct entirely new documents.

4.2.4. Reusing content

Reuse represents an important way for companies to extend the value of their investment in content. For example:

- Knowledge-based companies, such as newspapers and scientific publishers, commonly repackage and resell content. In addition, their own writers can reuse content, such as for background in a newspaper article or the basis of a quickly evolving online story.

- Companies that manufacture complex systems, such as airplanes and ships, must create and maintain large amounts of documentation. Since such systems are often configured for a specific client, each must have its own documentation. By modularizing documentation, custom manuals can be built from a library of topics. This reduces the chance of inconsistent documentation as well as surrounding costs, such as for editing and translation.

Other examples include building contracts from libraries of boilerplate text, publishing blogs and news releases as RSS feeds, and making documents available through an internal Web portal for arbitrary reuse.

4.3. Why you need a native XML database

Assuming they were built at all, many of the applications described earlier were originally built from full-text search engines, relational databases, and flat files. For example, they stored metadata in the relational database and documents in the file system, or they stored documents as CLOBs and copied them to the file system for use with a full-text search engine.

These systems suffered from two main problems. The first was scalability. According to one vendor, such systems “usually degrade very quickly past a few thousand documents, while the applications typically involve millions of documents.” The second problem was the lack of structured queries, since the full-text search engines were not XML aware and queries over metadata were limited to a few fields. Other problems included synchronization between the database and non-database components, the need to write custom code to process results, lack of node-based updates (a problem for large documents), brittleness in the face of evolving schemas, and the usual laundry list of reasons for using a database in the first place: concurrency, security, transactional safety, and so on.

(This is not to say that such systems don't work, especially under controlled conditions. For example, the American Geophysical Union uses one to manage roughly 95,000 scientific papers. Full papers are stored in a search engine repository, which provides full-text searches to Web site users. Metadata is extracted and stored in a relational database, where it is used to generate abstracts, bibliographic entries, and live citations. In the system's favor are a low growth rate (about 25 papers per day), very few updates, XML-awareness in the search engine, and a slowly evolving schema. The latter is particularly important, as schema evolution is the biggest problem in the system, requiring changes to the database schema, the XML-to-database mapping, and the extraction code.)

On the other hand, native XML databases have a number of features that are useful for working with document-centric XML. The most important are the XML data model, which is flexible enough to model documents, XML-aware full-text searches, and structured query languages like XQuery. These allow documents to be stored and queried in a single location, rather than multiple locations held together by glue code. Other useful features include node-level updates (which reduce the cost of updating large documents), links, versioning, and more flexibility in handling schema evolution than is found in relational databases. And while not all native XML databases can scale into the gigabyte or terabyte range, some clearly can.

4.4. A peek into the future

We will finish with a look at some recent developments in the field of content management:

- Some content management systems are being built on native XML databases rather than the file system or relational databases. For example, GEMt is based on Tamino; Docato, WorldServer GEP, and UltraXML are based on X-Hive/DB; Ingeniux CMS and Ektron CMS300 use TEXTML Server; TeraText DMS is based on TeraText DBS; and Syncato is based on Berkeley DB XML. While it is too early to say whether major systems like Documentum -- which currently uses a relational database -- will switch to native XML storage, it seems that the only barrier, at least for XML, is migration.

- JSR 170 [http://www.jcp.org/en/jsr/detail?id=170], which specifies a standard Java API for content repositories, has replaced SQL with XPath as its required query language. SQL support is now optional.

- Native XML databases are being used to extend systems that treat XML documents as opaque objects. IXIASOFT offers a plug-in for Microsoft's Content Management Server, which stores XML documents as CLOBs and allows queries only on metadata stored separately. The plug-in uses TEXTML Server to index documents and allows users to perform XML-aware full-text queries. These return pointers to the original documents. Similarly, QuiLogic offers a filter for the Windows Indexing Service. This uses SQL/XML-IMDB to query XML documents and return the contents of particular elements or attributes to the Indexing Service for indexing.

5. Data integration

The second major use of native XML databases is data integration. XML is well-suited to data integration because of its flexible data model and machine-neutral text format. In addition, XQuery is a good data integration language because of its ease of use, support for transformations, and ability to join data from different documents (data sources). Finally, there are a large number of tools for converting data from various formats to XML.

5.1. Data integration in the real world

Let's start by looking at some of the ways in which native XML databases are used to integrate data:

- *Business data*. A very common data integration problem is, in the words of Michael Champion, formerly of Software AG, to “get a coherent view of the mess in the back office.” For example, Vodafone uses Tamino in a system that lets customers and internal users view billing information over the Web; the system integrates data from SAP, Lotus Notes, and Microsoft Exchange. Snapbridge FDX XML Server and X-Hive/DB are used in similar fashion, integrating data from such diverse platforms as relational databases, flat files, weblogs, Web Services, LDAP, and HRM systems to give a single view of entities like customers or products.

- *Order analysis*. Hewlett Packard uses Ipedo to integrate internal financial data, such as order, shipment, and revenue data; expense data; and financial projections and goals. Data is gathered from a variety of sources, including relational databases, spreadsheets, flat files, PDF documents, and Word documents. It is queried and displayed through an internal Web portal.

  AZM -- a Swiss chain of convenience stores -- uses TEXTML Server to archive and search purchase orders and invoices from suppliers and customers. The primary data sources are ERP (Enterprise Requirements Planning) systems that use relational databases, but it turned out to be faster to search the data as XML than to perform the joins in the relational databases.

- *News production*. RTL, a German broadcaster, uses MaxiMedia Mpower for newsroom production. Mpower uses Tamino to integrate stories from a variety of news feeds using a subset of NewsML. The system currently stores more than 300,000 articles, comprising several gigabytes of data. Similarly, IXIASOFT's TEXTML Server is used to integrate data from multiple news feeds, each with its own format, into a single news production system.

- *Financial data*. Commerzbank uses Tamino to integrate data about customized-financial-derivative trades from 18 different systems. Data is first converted to XML and stored in Tamino. It is then transformed to a common schema for automatic evaluation and reporting by several back-office systems.

- *Flight information*. The central information system at Schiphol Airport in Amsterdam uses Tamino to integrate data from more than 38 systems in real time. The data describes flights -- arrival and departure times and gates, airplane types, baggage carousels, and so on -- and is used by more than 50 parties involved in flight handling -- from catering to baggage handling to gate assignment -- as well as being displayed to the public. Data is stored as XML, using an envelope schema derived from ebXML and predefined message types, such as for flight updates, airplane movements, and timetable changes.

- *Health care*. The TigerLogic XML Life Sciences Data Server (based on TigerLogic XDMS) can integrate medical and health care records from more than 20 data sources. Such integration problems are common in health care, where data usually resides on many legacy systems.

- *Manufacturing*. A production control system uses QuiLogic's SQL/XML-IMDB to integrate data from different machines and sensors, which is then queried through a Web services interface. Each machine and sensor publishes data in a different ASCII or binary format, which are converted to XML documents with a common schema.

- *Instrument data*. Renault uses X-Hive/DB to integrate data from sensors on its Formula 1 race cars, along with weather and track conditions and audio and video signals. The data is then made available to mechanics, engineers, and designers via a Web portal, who use it to perform "what-if" analyses.

- *Customer referrals*. TIBCO uses Xpriori's NeoCore XMS to integrate customer referral information from spreadsheets and various documents according to a common schema. The system is used to coordinate customer referrals worldwide.

- *Customer support*. A pharmaceutical clearinghouse uses Snapbridge's Cross Media Server to store existing RSS feeds from a number of relational databases. Customer support representatives query these feeds (primarily through full-text searches), then use an embedded URI to get back to the original data. While the solution is not elegant -- a schema more targeted to the data than RSS would have been better -- it had no impact on the backend databases and was very easy to write.

- *Law enforcement*. Raining Data's TigerLogic XDMS is used to search data gathered from government and law enforcement agencies, both domestically and internationally.

5.2. Inside the applications

Data integration applications must solve a number of problems, from data access to security to change management. In this section, we will look at the architectures used to solve two of these problems: queries and mapping schemas.

5.2.1. Query architectures

There are two query architectures for integrating data with a native XML database: local and distributed. In a local query architecture, data is imported into the database as XML and queried locally. In a distributed query architecture, data resides in remote sources and the query engine distributes queries across those data sources. The engine then compiles results and returns them to the application.

The main advantage of local queries is that they are faster, since no remote calls are made. They are also simpler to optimize, and the engine is simpler to implement, as all queries are local. Their main disadvantage is that data may be stale. A secondary problem is access control, as the local store must enforce controls previously handled by each source. Distributed queries have the opposite advantages and disadvantages: data is live, but queries are slower and harder to optimize and the engine is more complex.

Which architecture to use depends on a number of factors:

- *Support for distributed queries*. Only about one sixth of native XML databases, all of them commercial, support distributed queries.

- *Support for your data sources*. Most distributed query engines support a limited number of data sources, usually relational, hierarchical, or object-oriented databases.

- *Run-time availability of data sources*. Not all data sources are available at run time. For example, they might be disconnected for security reasons (a document database in an intelligence agency), connections to them might be unreliable (a Web site or a system in another company), or they might not allow external queries (a heavily-loaded production system).

- *Number of data sources*. Some vendors report that distributed queries perform well with a small number of data sources, but slow noticeably as the number of sources increases. While this is partially due to the number of sources, it may also be because of incorrect optimization choices made by the engine.

- *Update strategy*. A few native XML databases support distributed updates. While some vendors report that customers find it easier to write updates against a single view in the native XML database, others say that distributed updates have performance problems. They also note that updates done through a native XML database circumvent integrity checks and notifications performed by the applications normally used to update the data.

If a data source cannot be included in a distributed query, its data must be imported and queried locally. About a third of the commercial native XML databases, including most of the popular ones, can import data. Support ranges from a few sources, such as relational databases and Microsoft Word, to hundreds of sources. Some databases also have APIs so you can write your own import modules. And a few databases can refresh data in response to triggers or timeouts. If your database cannot import data, you must do so yourself, such as with the help of a third-party converter.

5.2.2. Handling differences in schemas

The biggest problem in integrating data is handling differences in schemas. With structural differences, the same concept is represented differently, such as a name using one or multiple fields. With semantic differences, slightly different concepts are represented; these can be straightforward (a price is in US dollars or Euros) or subtle (a price includes a discount). Handling schema differences is mostly just hard work, although some differences cannot be completely resolved. Since the actual resolutions depend on the differences, this section looks at where differences can be resolved.

If data is grouped by schema, such as in relational databases or (sometimes) in native XML database collections, three architectures are common:

- *Address differences in the query*. For example, the following query uses different functions to retrieve species names from collections of Bioinformatic Sequence Markup Language (BSML) [http://www.bsml.org/] and MicroArray and Gene Expression Markup Language (MAGE-ML) [http://www.mged.org/Workgroups/MAGE/mage-ml.html] documents:

```
declare function local:bsml_species() as element*
{
   for $o in collection("bsml")//Organism
   let $g = fn:string($o@genus)
   let $s = fn:string($o@species)
   return <Species>{$g} {$s}</Species>
}

declare function local:mage-ml_species() as element*
{
```

```
   for $s in collection("mage-ml")//Species
   let $v = $s/OntologyEntry[@category="NCBI:Taxonomy"]@value
   return <Species>{fn:string($v)}</Species>
}

<SpeciesList>
{
   let $s1 := local:bsml_species()
   let $s2 := local:mage-ml_species()
   for $s in fn:distinct-values(union($s1, $s2))
   order by $s
   return $s
}
</SpeciesList>
```

- *Convert all documents to the same schema*. This allows applications to ignore differences between schemas. Data can be converted at load time, such as when data is stored and queried locally, or at run time, such as when distributed queries are used. In the latter case, conversions are built into the XML views over the data. Note that the common schema only needs to include those fields to be queried. While this limits potential queries, it is a good way simplify development of limited systems.

- *Build common indexes over documents*. A few native XML databases allow users to specify how to build index values. These are used to resolve queries and return pointers to the original documents. For example, suppose one collection of academic papers uses multiple Author elements while another uses a single Authors element. A single index might use the Author elements directly and parse the Authors elements. This allows applications to query papers by author.

If data is not grouped by schema, application logic is more complex, since is it no longer possible to write location-based queries. If documents are limited to a set of schemas, it may be possible to base queries on document structure. For example, our species name query could check if the root element is Bsml or MAGE-ML before handing a document off to a function that extracts the species name.

If documents can have any schema, such as when a law office subpoenas all documents pertaining to a particular company and automatically converts them to XML, the best that can usually be done is to explore the documents with a browser to determine if there are common structures or fields. If so, it may be possible to use these in production queries or convert documents so that such queries are possible. As a last resort, users can always perform XML-aware full-text searches.

It may also be possible to ignore differences. For example, suppose an application displays information about a customer from multiple sources. Since a human is reading the information, they can resolve many differences, such as whether a name is stored in one or two fields.

5.3. Why you need a native XML database

Vendors report that most of their data integration customers were not able to solve their problems without a native XML database. The problem was that other solutions, such as federated relational databases and data integration toolkits, either could not model the types of data involved (documents, semi-structured data, hierarchical data, and so on), could not handle data whose schema was unknown at design time, and/or could not handle data whose schema changed frequently.

Native XML databases solve the first two problems with the XML data model, which is considerably more flexible than the relational model and can handle schemaless data. While native XML databases do not provide a complete solution for schema evolution, they can at least store documents with rapidly evolving schemas, as is discussed later.

Another advantage of native XML databases is that many support XQuery which, as was mentioned earlier, is a good data integration language.

5.4. A peek into the future

An interesting use of native XML databases in data integration is as a repository for metadata and semantic information. For example, the XML Business Information Portfolio from Software AG uses Tamino to store metadata and semantic information, as well as how to retrieve data from backend sources. While non-trivial to set up, this allows applications to execute queries against a common schema.

CompuCredit used an early version of this repository to integrate customer data from more than 100 systems and databases, each of which is exposed as a Web Service. In response to a query, the repository constructs an XML document from backend data and ships it across an Enterprise Service Bus to the customer service representative, who receives a single view of the customer's data.

6. Working with semi-structured data

Managing semi-structured data is the third major use case for native XML databases. Semi-structured data has some structure, but isn't as rigidly structured as relational data. While there is no formal definition for semi-structured data, some common characteristics are:

- Data can contain fields not known at design time. For example, the data comes from a source over which the database designer has no control.
- Data is self-describing. That is, metadata is associated with individual data values (as with element and attribute names in XML) rather than a group of values of the same type (as with column names in a relational database). Self-descriptions are used to interpret fields not known at design time.
- The same kind of data may be represented in multiple ways. For example, an address might be represented by one field or by multiple fields, even within a single set of data.
- Data may be sparse. That is, among fields known at design time, many fields will not have values.

6.1. Semi-structured data in the real world

Semi-structured data occurs in many fields. For example, here are some of the types of semi-structured data that are being stored in native XML databases today:

- *Data integration*. Integration data is semi-structured because the same concept is often represented differently in different data sources and changes to remote data sources can result in fields unknown to the integrator. Data integration was discussed earlier.
- *Schema evolution*. Rapidly evolving schemas result in semi-structured data because they introduce new fields and may change the way in which data is represented. These problems occur most commonly when data crosses organizational boundaries. Schema evolution is discussed separately.
- *Biological data*. Biological data -- especially molecular and genetic data -- is semi-structured because the field itself is evolving rapidly. As a result, the schemas used in these fields generally allow user-defined data. For example, much of the data in MAGE-ML is stored as hierarchies of user-defined property-value pairs. Similarly, BSML allows users to add arbitrary metadata in the form of property-value pairs.
- *Metadata*. Metadata is often semi-structured because users define their own types of metadata. For example, the Metadata Encoding and Transmission Standard (METS) [http://www.loc.gov/standards/mets/], which is used to

provide metadata for objects in digital libraries, defines only basic metadata, such as the name of the person who created the METS document, and allows users to define the rest. For example, a user might use Dublin Core [http://dublincore.org/] to provide information about the title, author, and publisher of a book whose digital image is in a library, and NISO MIX [http://www.loc.gov/standards/mix/] to provide technical data about how the image was created. On the other hand, while the Encoded Archival Description (EAD) [http://www.loc.gov/ead/] schema does not allow user-defined metadata, it is extremely flexible and will likely result in documents that sparsely populate the available fields.

- *Financial data*. Financial data is semi-structured because new financial instruments are constantly being invented and because it is often the result of integrating data from many proprietary systems. An additional source of change in the XML world is the rapid development of standards like the Financial Information eXchange Markup Language (FIXML) [http://www.fixprotocol.org/cgi-bin/Welcome.cgi?menu=1] and Financial products Markup Language (FpML) [http://www.fpml.org/]. (Of interest, the FIXML specification explicitly discusses how to customize FIXML.)

- *Health data*. Health data is semi-structured because it is sparsely populated, it is often the result of integrating data from many proprietary systems, and user-defined data is common. For example, HL7 [http://www.hl7.org/] has hundreds of elements --(it is unlikely that the description of any patient or organization will use all of them -- and makes frequent use of the xsd:any element.

- *Business documents*. Business documents are semi-structured because the real world is a highly variable place. Most documents contain a core set of fields -- name, address, date, and so on -- as well as user-defined fields. For example, while insurance claims have a number of fixed fields (name, policy number, date, and so on), the bulk of the information is free-form (accident description, police reports, photographs, and so on).

- *Catalogs*. Catalogs are hierarchies of product descriptions. While some catalogs are rigidly structured -- that is, a single set of fields can be used to describe each node in the catalog -- other catalogs are semi-structured. One reason is that different parts, such as a piston, a tire, and a carburetor, are described by different fields. Another reason is that some catalogs integrate data from different vendors, each of whom uses their own schema.

- *Entertainment data*. Entertainment data is semi-structured because the services being described (films, restaurants, hotels, and so on) vary tremendously. As a result, data is sparsely populated and schemas change frequently. Entertainment data also comes from a variety of sources (movie theatres, newspaper reviews, hotel chains, and so on), which may result in integration problems.

- *Customer profiles*. Customer profiles are semi-structured for two reasons. They are sparsely populated because few customers have data for all fields (frequent flier numbers, food preferences, preferred travel time, and so on). They evolve rapidly because the ways in which people are described (contact information, exercise preferences, medical conditions, and so on) change constantly.

- *Laboratory data*. Laboratory data is semi-structured because it is sparsely populated -- different measurements apply to different substances -- and because there is ample room for user-defined data. For example, in the pharmaceutical approval process, a single application might handle all of the documentation for applying for drug approval, yet different drugs are likely to require different sets of data.

6.2. Inside the applications

Applications that work with semi-structured data that has a known schema are not significantly different from applications that work with other kinds of data. For example, they use queries defined at design time to retrieve and update data. The main difference is that they often must handle data represented in different ways in different parts of the data set. While this may be unpleasant, as long as the number of variations is limited, it is usually possible.

Applications that work with semi-structured data containing fields not known at design time are fundamentally different. As a general rule, such applications pass unknown fields to humans for processing. For example, suppose a catalog

has a basic structure defined by a central authority and uses vendor-specific XML to describe individual items. A catalog browser might be hard-coded to navigate the known structure and use XML-aware full-text searches or // searches to search the unknown structure. Product data might be displayed as raw XML or converted to XHTML with a stylesheet that displays data based on nesting level.

Similar applications are found in molecular biology, genetics, health care, and library science. In each case, the data describes something -- a molecule, a gene, a patient, an archive -- and many of the fields are known. The application uses these fields, such as to allow the user to drill into the data, and then displays the unknown fields. The person reading the data can interpret it and take further action, such as reading a scientific paper, making a diagnosis, or adding comments.

Another common solution is for the application to evolve with the data. For example, incoming documents can be examined with a generic browser to decide what kinds of queries are possible. In some cases, it might be possible to write specific queries, such as //address to search for addresses; in other cases, the only choice might be full-text searches. While this kind of development is likely to be repugnant to programmers accustomed to working with well-defined schemas, it is a huge improvement for users whose previous choice was to wade through reams of paper or search files in a variety of formats using a variety of tools.

6.3. Why you need a native XML database

XML is a good way to represent semi-structured data: it does not require a schema; it is self-describing (albeit minimally so); and it represents sparse data efficiently. Thus, native XML databases are a good way to store semi-structured data. They support the XML data model, they can index all fields (even those unknown at design time), they support XML query languages and XML-aware full-text searches, and some support node-based updates.

Relational databases, on the other hand, do not handle semi-structured data well. The main problem is that they require rigidly defined schemas. Thus, fields not known at design time must be stored abstractly, such as with property-value pairs, which are difficult to query. They are also difficult to change as the schema evolves. A secondary problem is that they do not handle sparse data efficiently: the choices are a single table with lots of NULLs, which wastes space, or many sparsely populated tables, which are expensive to join.

According to vendors, many customers couldn't handle their semi-structured data until they used a native XML database. Other customers used a variety of tools, such as grep, full-text search engines, and proprietary applications, or stored some data in a relational database and complete documents as flat files, CLOBs, or even Word documents. As a general rule, these solutions worked in the initial stages, but had limited query capabilities, didn't scale well, and were difficult to maintain as schemas evolved.

(A notable exception occurred in the field of biology. The AceDB [http://www.acedb.org/] database was initially written to store data about the worm C. elegans. It has since evolved into a generic, object-oriented database with its own schema, query languages, and data browsers. Other databases, such as UniProt/Swiss-Prot [http://www.ebi.ac.uk/swissprot/] and GenBank [http://www.ncbi.nlm.nih.gov/Genbank/index.html] are (apparently) available in relational and flat-file formats, but are generally queried through proprietary tools such as SRS [http://www.lionbioscience.com/solutions/e20472/e20475/index_eng.html] and Entrez [http://www.ncbi.nlm.nih.gov/gquery/gquery.fcgi].)

6.4. A peek into the future

Semi-structured data is still straddling the boundary between academia and industry, so the near term is most likely to consist of gaining experience -- managing data, writing applications, handling evolution, and so on -- than creating definitive tools.

7. Schema evolution

As someone accustomed to the relatively rigid schemas of the relational world, I react to stories of rapid schema evolution with a mixture of horror and a sense that perhaps the people involved aren't as, well, *responsible* as they should be. In spite of this, almost every vendor and customer I spoke to listed schema evolution as one reason to use a native XML database. Worse yet, most had good reasons for doing so.

7.1. Schema evolution in the real world

Schema evolution is a normal thing. In the relational world it moves slowly, for both technical and political reasons. On the technical side, relational databases do not handle schema changes easily: existing data must be updated to match the new schema and altering tables may require unloading and reloading data. On the political side, database administrators (DBAs) tend to approach change cautiously because they don't want to break existing applications or destabilize database tuning.

(A number of vendors also noted that native XML databases allow developers to do an end run around DBAs, resulting in faster development times. One reason for this might be that native XML databases are often used to cache data on the middle tier, meaning DBAs are not aware of them and have not yet brought them under their control. Another reason might be that native XML databases do not have as many tuning options as relational databases, meaning that DBAs have less reason to exert control.)

In the XML world, change moves faster. This is sometimes due to the newness of XML. For example, FIXML has had four versions in six years and FpML has had four versions in just three years. XML has also exposed users to more sources of change. For example, the schemas used to move data across organizational boundaries are often controlled by other departments or trading partners. And XML is being used in rapidly evolving fields, such as finance and biology, as well as fields with long life spans, such as mortgage and insurance contracts, both of which force users to handle many versions of a schema.

7.2. Inside the applications

Handling schema evolution is rarely easy. The easiest solution is to update data to conform to the new schema and update applications accordingly. Unfortunately, this is not always possible. For example, updating existing data might be too expensive or might be prohibited (such as with contracts), new fields might not have reasonable defaults, or multiple applications might use the data and cannot all be updated.

When data cannot be updated, applications must handle both backwards and forwards compatibility. Since documents conforming to multiple versions of a schema are commonly stored together in native XML databases, applications must determine which version of a schema is being used, such as by checking a version attribute or checking whether a particular field exists.

Handling backwards compatibility usually just means a lot of hard work, such as providing default values for fields added in a new schema or processing each version of a field differently. However, some problems have no definitive solution, such as how to compute the average of a field not found in all documents.

Handling forwards compatibility means protecting applications against an unknown future. A liberal strategy is to ignore all unrecognized fields. Unfortunately, this is risky, as new fields may change the semantics of existing fields. A more conservative strategy is to only process documents with a known version number. This allows applications to continue working until it can be determined whether a schema change breaks existing code.

A strategy that avoids many forwards and backwards compatibility problems is to query only those fields that are unlikely to change. This works particularly well when humans are involved. For example, a customer service representative might search for contracts involving a particular customer or a researcher might search for documents describing a

particular organism. In both cases, searches are done on stable fields (customer or species name) and the reader can resolve any differences in schemas.

7.3. Why you need a native XML database

The main advantage of native XML databases with respect to schema evolution is the ability to store documents conforming to several different versions of a schema. This has several advantages over relational databases, which require data to conform to a single schema:

- Schemas can be changed without having to migrate data, as is the case for relational databases. For large data sets or rapidly evolving schemas, migration can be prohibitively expensive. Furthermore, it is not always possible to migrate data. For example, changing a contract invalidates it.

- The database can handle schema changes for which there is no data migration path, such as when a new field is required and has no reasonable default. In a native XML database, new documents can be stored in the same collection as old documents. In a relational database, new data must be stored in a different table from the old data, since the old data cannot be migrated. As a result, queries over unchanged fields continue to work in the native XML database but fail in the relational database because they do not include the new table.

- Data can be stored, even if it conforms to an unknown version of a schema. This means that no data is lost, even if it cannot be used immediately. Depending on the forward compatibility strategy, it might even be possible to process the new data.

A secondary advantage of native XML databases is support for XQuery. The conditional expressions and user-defined functions in this language are very useful in querying documents conforming to multiple versions of a schema.

This is not to say that native XML databases solve all schema evolution problems. Far from it -- schema evolution remains a painful problem that requires both foresight and hard work. However, the consensus among vendors and customers is that the flexibility of native XML databases makes solutions possible where they weren't before.

7.4. A peek into the future

The only happy news about schema evolution is that more people are becoming aware of the problem, as evidenced by the number of emails and conference presentations on the subject. Personally, I hold little hope for any silver bullets, as the problem pre-dates XML and has not been solved yet.

8. Long-running transactions

Long-running transactions are real-world transactions such as processing insurance claims, approving mortgages, or fulfilling orders. They generally require a mixture of human and machine processing and take anywhere from hours to weeks. They differ from traditional transactions in that they do not lock resources for the duration of the transaction and they use compensating transactions, such as refunds, instead of rollbacks.

How data flows through a long-running transaction depends on the application. It might be stored in a database and modified by a succession of applications, or it might be passed from application to application in one or more XML documents, as in a Service Oriented Architecture (SOA).

Native XML databases can be used in long-running transactions in a number of capacities:

- *Data stores*. Much of the data in long-running transactions is document-centric (contracts, appraisals, accident descriptions) or integrated from a variety of sources (credit agencies, appraisers, backend databases). As we have seen, native XML databases are useful both for storing document-centric XML and integrating data. Whether the

native XML database is the database of record depends on the application. In many cases, they serve as mid-tier data caches and data is off-loaded to backend databases.

- *Message queues*. Unlike traditional message queues, native XML databases can perform content-based routing and transform messages into different formats. While native XML databases are slower than traditional message queues due to parsing and reassembling messages, as well as querying and transforming them, vendors did not report any performance problems. However, this may be because native XML databases have not yet been used in sufficiently demanding environments.

- *Metadata archives*. In addition to storing application data, native XML databases are also used to store information used by applications. For example, Raining Data's TigerLogic XML Data Management Server is used in metadata-driven SOAs to store metadata about Web Services, access policies, and aggregated views of UDDI and home-grown service registries.

- *Data warehouses*. When native XML databases are used as data stores or message queues, they can also serve as data warehouses that can be mined for information about data or messages.

It is interesting to note that several Enterprise Service Buses (ESBs), which are used to implement SOAs, include native XML databases: Sonic ESB includes Sonic XML Server, Software AG's Enterprise Service Integrator includes Tamino, and OpenLink's Virtuoso includes a BPEL engine. These systems use native XML databases for all of the reasons described above.

9. Handling large documents

Large documents are difficult to query due to the time it takes to parse them. Native XML databases solve this problem by parsing and indexing documents when they are inserted. This allows documents to be queried without further parsing and may even allow queries to be resolved only by searching indexes.

Large documents are also difficult to process with XSLT and DOM, as these require the entire document to be in memory. Since sufficiently large documents exceed available memory, some native XML databases solve this problem by implementing XSLT and DOM directly on top of the database. These implementations populate in-memory nodes as necessary and swap nodes back to disk as needed, allowing DOM and XSLT to be used with documents of almost arbitrary size. In addition, changes made to DOM trees are reflected back to the database, either immediately or in response to a special call.

The main use of such DOM implementations is in browsers and editors for document-centric documents, such as catalogs and technical manuals. While most of these are custom applications built on top of native XML databases, Infonyte has built a customizable browser (the Infonyte Reader) on top of its native XML database (Infonyte DB), which features both query and XSLT engines.

10. Hierarchical data

Hierarchical data is a use case that overlaps all other use cases, since virtually all XML is hierarchical. Hierarchical data is either heterogenous, like sales orders, in which parents and children have different types, or homogenous, like catalogs or bills of material, in which parents and children have the same type. In a relational database, heterogenous hierarchies are stored in multiple tables, which must be joined during queries, and homogenous hierarchies are stored in a single table, for which there are a variety of query strategies, including nested sets [http://www.intelligententerprise.com/001020/celko.jhtml?_requestid=235427] and recursive queries.

While there is little public data available for the relative performance of native XML databases and relational databases in querying hierarchical data, it is interesting to note that Xyleme Zone Server outperforms [http://www.xyleme.com/index.v3page?sc_v=r&sc_eid=28302&p=47749] Oracle 9i by a factor of 19.5 when using Oracle's test data and Oracle's object-relational XML storage. Similarly, another native XML database vendor asserted

that even “three to four levels [in a heterogenous hierarchy] presents a problem for relational [databases], once the [number] of documents is in the hundreds of thousands.” (Of interest, vendors report that most of their customer's hierarchies are five to ten levels deep, although up to thirty levels are not uncommon.)

Unfortunately, similar data is not available for relative query performance in homogenous hierarchies. However, there is persistent confusion [http://www.dbazine.com/tropashko4.shtml] among SQL programmers about how to store and query hierarchical data. While this may be alleviated with the introduction of recursive queries in relational databases (available for several releases in Oracle, one release in DB2, and the next release in SQL Server), perhaps the most important thing that native XML databases bring to the table with respect to hierarchical data is a set of tools -- notably query languages -- that are explicitly designed for working with hierarchies.

11. Other uses

This article has described the most common use cases for native XML databases. Some other use cases are:

- *Local and shared data management*. QuiLogic's SQL/XML-IMDB is primarily used as a local data manager. That is, rather than using structures like lists and queues, SQL/XML-IMDB is used to store data of arbitrary complexity. This allows such structures to be handled in a declarative manner using XQuery and SQL. Furthermore, because SQL/XML-IMDB supports the use of shared memory, it can be used to share data among processes. For example, one such use allows a laboratory data collector written in C++ to talk to a front end written in Python.

- *Complete Web site*. Native XML databases can be used to build Web sites: data is stored in the database as XML, queried and updated with XQuery, and transformed into XHTML with XQuery or XSLT. This is an experimental use case for Sedna, and has been implemented by a number of people. For example, the University of Virginia's Rotunda [http://rotunda.upress.virginia.edu] site is built on Mark Logic's Content Interaction Server.

- *Performance*. Vendors report that customers do not choose native XML databases solely for performance reasons -- that is, when a relational or object-oriented database would do -- but that performance was frequently a secondary criteria. For example, applications that use document-centric XML or semi-structured data, and which originally used the file system or a relational database, were migrated to a native XML database for both feature and performance reasons.

- *Mid-tier data cache*. While this use case overlaps other use cases, it is worth mentioning separately. Native XML databases are often used to cache data on the middle tier, such as in data integration, e-commerce, Web sites, and long-running transactions. This is done both for performance and to manage data in a common format (XML).

12. A final peek into the future

Our final peek into the future looks at relational databases. In the strongest endorsement of native XML databases to date, the major relational databases are adding native XML storage. This is used to implement a first-class XML data type and data stored as this type can be queried with XPath or XQuery. It can also be mixed with relational data.

The implementation strategies used by relational databases are as varied as those found in commercial native XML databases: Oracle indexes documents and stores them as CLOBs; Sybase also indexes documents (it is not known how they store them); SQL Server stores pre-parsed documents as BLOBs, as well as in node-level storage built on relational tables (the query engine decides which to use at run time); and DB2 uses node-level storage built from the ground up.

In addition, Oracle is working on XML Data Synthesis (XDS) [http://www.oracle.com/technology/tech/xml/xds/index.html], an XQuery-based data integration engine.

13. Conclusion

This article has looked at how native XML databases are used in the real world -- most commonly for managing documents, integrating data, and managing semi-structured data. What is important about these uses is that most represent cases where people have tried to use relational or other types of databases and have either failed or written less sophisticated applications than they would like. Native XML databases have succeeded because of their query languages (most notably XQuery, but also XML-aware full-text queries), the flexibility of the XML data model, and their ability to handle schemaless data.

So is a native XML database in your future? That question is best answered by quoting Arun Gaikwad. In an article [http://www-106.ibm.com/developerworks/web/library/wa-xindice.html] about Xindice, a native XML database from Apache, he wrote: “A [native XML database] is something which you may think is unnecessary but once you start using it, you wonder how you would survive without it.”

14. Resources

Use cases for native XML databases:

- Use Cases for Native XML Servers [http://www.idealliance.org/papers/dx_xml03/papers/05-05-01/05-05-01.html] by Brian Quinn (Software AG). Discusses document management and data integration with native XML databases.
- Native XML database overview [http://sm.vanx.org/pipermail/vanx-list/2003-September/000050.html] by Scott Carroll (Bluestream Database Software). An e-mail summary of what native XML databases are and how they can be used.

Selected case studies:

- Time Out Customer Case Study (PDF) [http://www.xyleme.com/index.v3page?sc_v=r&sc_eid=28298&p=47749] by Xyleme. How Time Out (a publisher of entertainment guides) uses Xyleme Zone Server to handle semi-structured data.
- NeoCore XMS Powers Gene Expression Analysis Site (PDF) [http://www.xpriori.com/library/Xpriori_and_CCB_cs012003.pdf] by Xpriori. How the Center for Computational Pharmacology uses NeoCore XMS in a Web site for gene expression analysis.
- Formula 1 Racing [http://www.x-hive.com/customers/renault.html] by X-Hive. An interesting use of X-Hive/DB for near real-time data analysis.
- XML Plays Big Integration Role [http://www.informationweek.com/story/showArticle.jhtml?articleID=20900153] by Charles Babcock. Summary of how Software AG integrated customer data for CompuCredit. Not mentioned is Tamino's role as a metadata repository, which provides a single view of the data.
- Powered By eXist [http://demo.exist-db.org/exist/apps/applications.xml] by exist-db.org. Short list of applications that use eXist, an Open Source native XML database.
- Connected To The Law: Tasmanian Legislation Using EnAct (PDF) [http://www.teratext.com/documents/TasmainianWhitePaper.pdf] by Tim Arnold-Moore, Jane Clemes, and Matthes Tadd. A detailed description of EnAct, a “legislation drafting, management and delivery system” built on top of TeraText DBS. TeraText DBS is referred to as SIMS (an earlier name) in the study.
- Case Study: Cedars-Sinai Medical Center [http://www.marklogic.com/prod_cs_cedars.html] by Mark Logic. Three applications that use the Mark Logic Content Interaction Server to track medical data and regulatory correspondence.

- XML-centric workflow offers benefits to scholarly publishers [http://www.idealliance.org/proceedings/xml04/abstracts/paper71.html] by A. Schwarzman, H. Hur, S. Pai, and C. Glass. A description of the system used by the American Geophysical Union, which uses a relational database and an XML-aware full-text search engine.

Native XML databases:

- Introduction to Native XML Databases [http://www.xml.com/pub/a/2001/10/31/nativexmldb.html] by Kimbro Staken. An introduction to native XML databases.
- XML and Databases? Follow Your Nose [http://www.xml.com/pub/a/2001/10/24/follow-yr-nose.html] by Leigh Dodds. How to decide between an XML-enabled database and a native XML database.
- XML Database Products: Native XML Databases [http://www.rpbourret.com/xml/ProdsNative.htm] by Ronald Bourret. Short descriptions of most native XML databases.

Acknowledgements

I would like to thank representatives of the following companies and organizations for contributing their time and thoughts to this article: American Geophysical Union [http://www.agu.org/], Bluestream Database Software [http://www.bluestream.com/], Cincom [http://www.cincom.com], data ex machina [http://www.data-ex-machina.com/], IBM [http://www.ibm.com], Ipedo [http://www.ipedo.com/], ISPRAS modis [http://www.modis.ispras.ru/index.htm], IXIASOFT [http://www.ixiasoft.com/], M/Gateway Developments [http://www.mgateway.tzo.com/default.htm], Mark Logic [http://www.marklogic.com/index.html], Ontonet [http://www.ontonet.com/], OpenLink Software [http://www.openlinksw.com/], QuiLogic [http://quilogic.cc/], RainingData [http://www.rainingdata.com/], Snapbridge Software [http://www.snapbridge.com/], Software AG [http://www.tamino.com], X-Hive [http://www.x-hive.com/], Xpriori [http://www.xpriori.com/], and Xyleme [http://www.xyleme.com/]. Thanks also to developers and users who chose to remain anonymous.

Biography

Ronald **Bourret**
Researcher
rpbourret.com [http://www.rpbourret.com]
Felton
California
United States of America

Ronald Bourret is a freelance programmer, technical writer, and researcher. He has worked in the software industry since 1982 and his experience includes five years as a contractor at Microsoft, where he co-wrote the ODBC and OLE-DB manuals, and two years at the Technical University of Darmstadt in Germany, where he did research on XML and databases.

His XML work includes XML-DBMS, a set of Java packages for transferring data between XML documents and relational databases, several widely read papers on XML and databases, the XML Namespaces FAQ, and an XML schema language (DDML).

He has lectured on XML and databases at both commercial and academic conferences and has contributed articles to both XML.com and xmlhack.