SQL Server, XML, and the DBA

Some of the reasons you have bought this book may include:

- ❑ You need to return queries on SQL Server relational data in an XML format
- ❑ You want to store XML data in SQL Server
- ❑ You want to find out more about SQL Server's XML capabilities in general

But you may well be wondering why data needs to be transmitted in an XML format at all, when relational datasets offer a mechanism for presenting data to the end user or system that works well? You would be right to wonder.

Why change? Why invest all this time in recreating and optimizing the whole storage and retrieval mechanism, when a perfectly good system exists already? Well, as we shall see, the data format that the end application or system needs is changing (for all the right reasons).

Whatever the reason for your interest, this book will help by providing you with a thorough grounding in the different capabilities offered by SQL Server 2000 to provide XML to applications, end users, or other systems, and also to store XML data in your SQL Server instance.

This first chapter provides you with an overview of where the need for XML as a data format comes from, and its main uses from a DBA's viewpoint, so that you can best decide how to implement an XML solution that's right for you.

This chapter is structured as follows:

- ❑ *Data Based Computing*. A look at what's driving the integration of relational databases and the web, and how XML fits in.
- ❑ *What Aspects of XML are Relevant to Databases?* A brief overview of XML standards relevant to this book.

- *XML and Relational Structures*. Assuming that you have at least a basic appreciation of these structures, this section will highlight the different uses of each.
- *The Main Uses of XML Structures*. Consideration will be given to just why XML is becoming critical to today's IT environment.
- An *Architectural View* (or a different approach to the same problem). This sub-section will provide different ways to look at the 'bigger picture' and understand the bigger forces at work

Let's now move on to the first of these sections.

Data Based Computing

In order to understand the approach to using relational data and XML advocated by Microsoft and covered in this book, we need to take a short look at the history of **data based** (rather than **database**) computing.

Throughout the evolution of data based computing there have been (at least) four constants:

- Data needs to be stored somewhere in a structured, readily retrievable way
- Data must be presented somehow (electronically or on paper) in a semi-structured way
- Data must somehow be extracted, transmitted, and transformed between its persisted storage location and its presentation location
- All of the above must work optimally at each stage, without losing flexibility

These constants say nothing about the actual data or any relations it may have internally; it's assumed that they will be captured. Instead, they refer more to the environment data must exist and operate in to be considered "a working solution".

The Success of Relational Databases

Relational databases have been a huge success. At the present time, they offer support for:

- Many thousands of concurrent users
- Terabytes of data
- Storage of almost any data type you can mention
- Relational and analytical models of data
- Querying data in an ad hoc and flexible manner
- Security, both authentication and authorization
- Programming within the database

- Network connectivity to applications and other databases allowing efficient data retrieval
- Tools to develop, manage, and report on huge amounts of data

Relational databases do well on every one of these four constants, SQL Server among them. There's no compelling reason to change from the relational model as the primary store for structured data.

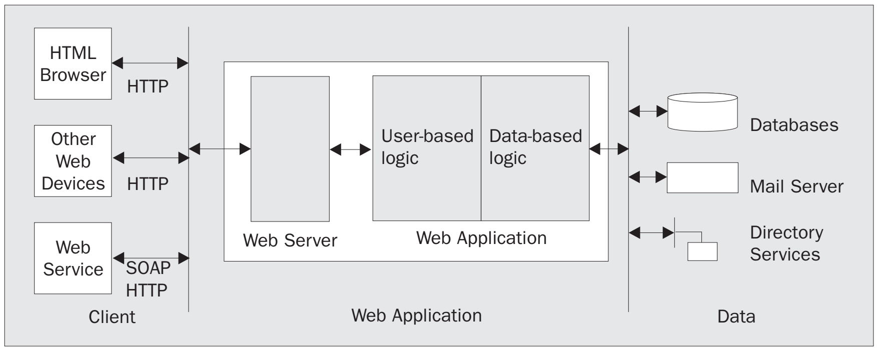
The Success of the Web

The Web application architecture is basically the latest in a series of computer architectures:

- The computer is the platform – data is shared within a computer, whether mainframe or PC.
- The operating system is the platform – data is shared between networked computers running the same operating system.
- The application or database is the platform – data is shared between databases from the same manufacturer, and between relational databases using data connectivity tools.
- The Web is the platform – data is shared between all sorts of data stores using standard protocols, languages, and discovery mechanisms implemented on all platforms, and supported by all applications.

Each level increases the amount of data that can be shared, expands the audience who can use that data, and simplifies the provision of the pieces that make up the system.

Data-driven web applications have been the making of the commercial web. The now "classic" web application consists of data stored in a database, processing done either within the database or on a web server, and presentation as semi-structured HTML documents over HTTP to a web browser:



The main change in this approach, compared to client-server, is that the browser does not work in the same way as the client GUI. It doesn't understand a `DataSet` object natively; it has to be converted to HTML in order to work. Something fundamental has happened here. In the past the `Recordset` was the means of communication; the database knew how to give a `Recordset`, the client application knew how to process it.

What is clear is that HTML and browsers work. They allow easy rendition of both documents and data, and provide a solid framework for interoperability of a myriad different applications, with different native backgrounds. It is also clear that the potential end user community is now global and not limited to clients that have the right software installed. Sorry to all those in IT security for throwing down this gauntlet, but this is the new challenge.

It is also clear that HTML isn't optimal. Hence the creation, and ongoing momentum of standards generated by the W3C, to enhance and enrich what functionality is available in a browser (XML, XHTML, XSLT, XPath, and so on).

XML – The Next Step

Most of the web applications to date create documents for humans to read. But the principles of web applications, central data store, closely-tied business logic, and loosely-coupled communication between authenticated clients and URL-based services, can be applied to client-server computing to:

- Make more pieces of the system generic and thus reduce "re-coding the wheel".
- Expand the audience of applications and users who can consume that data.

XML and other Internet technologies provide the future format for data moving over the Web. The pieces of this puzzle are:

- TCP/IP – the network extends everywhere – the medium for conversation.
- HTTP/SOAP – the protocol for sharing documents – the conversation.
- XML – the language of languages that can be shared. The data is 'readable', and can't be lost in proprietary formats – the language of the conversation.

Why are the relational database giants slugging it out in a war over their XML support? The reason is that the closer the interaction between the source of the data, the database, and the format in which the data is transferred, the better.

Until this point it has been application developers who have had to consider the best way of turning relational data into HTML or XML. In the Microsoft world, ADO does this very effectively; as does ADO.NET, which provides rich XML functionality for free, but such functionality is gradually reaching into the database. SQL Server has offered functionality to allow retrieval as XML from relational data as an add-on to Version 7, with fuller support in SQL Server 2000, and much more promised in Yukon (see Chapter 17). So where do you do the conversion? Indeed, do you store data as XML in the first place in order to enhance performance? Do you offer a dataset following an URL query of the data? It's a question you can no longer ignore. It depends upon the application, as we shall see.

This applies to traditional client-server systems too, where the need to make the application as flexible as possible means that the web is used to link rich clients and server applications. Whilst ADO, OLE DB, and ODBC formerly offered the cleanest mechanism for supplying data to a Windows forms application, this *may* no longer be the case. Again, it depends upon the application. We'll see more of this in Chapter 13 on .NET

System Interoperability and Data Interchange in an n-tier Environment

Vendors have realized the potential for the use of XML as a base structure to provide cross communication between different systems. First XML-RPC, then SOAP have been released, which provide communication directly by file transfer as recommended, over HTTP, or through SMTP. This opens up the standard presentation tier, using HTTP as a sound base for cross communication.

So just when you thought you had enough to be getting on with, there is another massive area to understand, which I am sure will dominate the evolution of IT architectures and environments for many years to come. That area is **web services**.

Web Services

Though not directly an evolution of the n-tier architecture, web services represent the next step in IT thinking. Web services are all about software as service; that is, rather than simply offer data to a web browser, why not offer (or expose) methods and business logic functionality? Web services utilize SOAP and WSDL to expose functionality to a consumer. As long as the consumer understands the WSDL, they can query the web service. In terms of n-tier architecture, web services can be incorporated into both the presentation tier and the business tier (or, as we shall see in Chapter 14, the data tier).

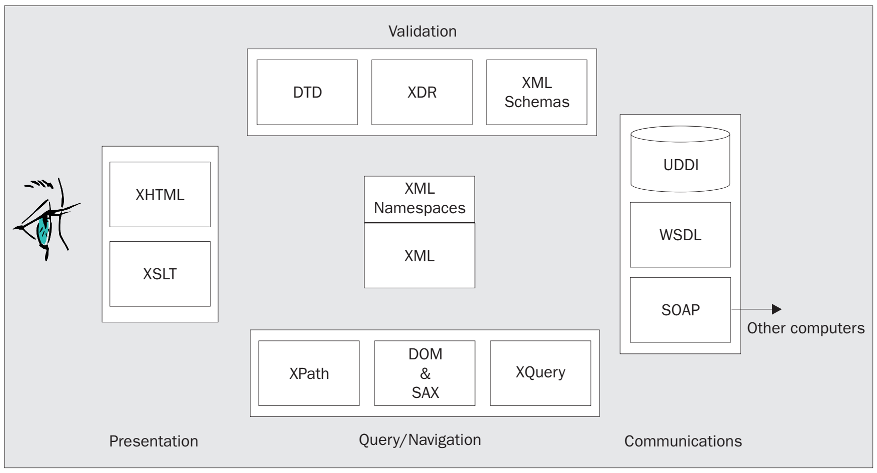
In terms of the constants we talked about, web services should be seen as another way to make the data available either for presentation, or for distributed interoperability.

In summary, the following worlds have been converging:

- Multi-user databases, with a central data store, central data-specific processing, and great concurrency.
- Internet networking protocols that allow for platform-independent access to data.
- Client-server models allowing a division of processing.
- Distributed data, allowing clients to query multiple databases and combine information from different sources.
- Interaction between platforms, necessitating platform independent data that can be worked with on any of the platforms involved, and shared between them.

What Aspects of XML are Relevant to Databases?

The following figure is a high level overview of the many XML standards and their purposes:



We'll follow this with a table to summarize the nature of these standards. It is not meant to be definitive (importantly it doesn't talk about versions): more a general overview of the XML standards discussed in this book:

XML Target Area	Name	Description and URL
Core	XML	Simply put, XML on its own is no more than a standard format for representing data. **Well-formed XML** conforms to the XML Recommendation. XML specifies how to represent elements and attributes, how elements nest, and what elements may contain. http://www.w3.org/XML/

Core	XML Namespaces	Namespaces are used to identify the uniqueness of an XML element. For example:
		
		`<xsl:stylesheet version="1.0"`
		`xmlns:xsl="http://www.w3.org/1999/XSL/Transform">`
		
		ties any XML element preceded by `xsl:` to the XSL standard.
		
		http://www.w3.org/TR/REC-xml-names/

XML Target Area	Name	Description and URL
Querying/ Navigation	DOM parser (Document Object Model)	A method of reading and manipulating an XML document which works by creating a virtual tree structure in memory. DOM APIs exist for most platforms. http://www.w3.org/DOM/
Querying/ Navigation	XPath	XPath provides a mechanism to interrogate an XML document. http://www.w3.org/TR/xpath
Querying/ Navigation	XQuery	XQuery expands upon the interrogative nature of XPath and introduces a complete query language structure, similar to SQL, but for XML documents. http://www.w3.org/TR/xquery/
Validation	DTD (Document Type Definition)	An early SGML-based method which allows the definition of an XML document structure. An XML document which conforms to the structure defined in a DTD is said to be **valid**. http://www.w3.org/TR/REC-xml#dt-doctype
Validation	XDR (XML-Data Reduced) Schema	An early approach by Microsoft to define a validation mechanism which can allow the definition of datatypes as well as XML document structure. Similar to the final World Wide Web Consortium(W3C) XML Schema Recommendation. http://www.w3.org/TR/1998/NOTE-XML-data/
Validation	XML Schema	A W3C Recommendation which allows the definition of datatypes as well as XML document structure. http://www.w3.org/XML/Schema

Table continued on following page

XML Target Area	Name	Description and URL
Presentation	XHTML (Extensible HTML)	An evolution of HTML which defines standard document types and module structures for more rigid implementation of a browser page. An XHTML document is an XML document. http://www.w3.org/TR/xhtml1/
Presentation	XSLT (Extensible Stylesheet Language for Transformations)	XSLT provides a transformation mechanism which is used to convert XML to other formats (including XML). In the context of HTML creation, an XSLT stylesheet provides the presentation information to be applied to the XML content. A transformation API is used to merge the XML and XSLT stylesheet. http://www.w3.org/TR/xslt
Presentation	XSL	XSL is the overall term used to encapsulate the XSLT and XSL:FO (a document presentation technology) vocabularies. http://www.w3.org/TR/xsl/
Communication**	XML-RPC (XML Remote Procedure Call)	XML-RPC is basically the use of XML to define the request/response mechanism for a method call and response over a network (LAN, WAN, or Internet). It is the precursor to SOAP. http://www.xmlrpc.com/
Communication	SOAP (Simple Object Access Protocol)	SOAP is an evolution of XML-RPC and provides a low level XML structure for method call/response. As a protocol, it is richer than XML-RPC and offers asynchronous calls, error handling, and data typing (using XML Schema in SOAP 1.2). SOAP is the bedrock of web services. http://www.w3.org/2002/ws/

XML Target Area	Name	Description and URL
Communication	WSDL (Web Service Description Language)	WSDL is an XML vocubulary that is used to describe web services. It describes the "rules of engagement" for a web service. A WSDL file can be read by a web service creation mechanism in order to understand what methods are available to be consumed. http://www.w3.org/TR/wsdl
Communication	UDDI (Universal, Description, Discovery and Integration)	UDDI is a repository-based registry service for looking up web services. http://www.uddi.com/

***All XML-based communication protocols are typically used over HTTP and offer easy interoperability between disparate systems.*

If many of these standards and Recommendations are new to you, you might like to refer to *Beginning XML, 2nd Edition* (Wrox Press, ISBN 1861005598) for good introductory information.

XML and Relational Structures

Let's now take a brief look at the main similarities and differences between the two different data structures, to get a feel for the areas they work in, and the areas where they are not optimal.

Relational Structures

Within SQL Server, a data item, for example a customer name, is stored as a field, within a record, within a table, and it conforms to a defined data type. It is indexed as appropriate, and because of its status in the database (that is, as part of the customer record), it can be extracted via any route using SQL, as long as a relationship exists. SQL Server databases are optimized for the storage and retrieval of such data items and, because they sit within a SQL Server database, they dwell in a secure, versioned, and administered environment.

XML Structures

Let's consider the same data item (customer name) within an XML document. It is stored as an element (or an attribute), it is a child of a parent element, and it can conform to a defined data type (using XML Schema). Because of its status in the XML tree, it can be extracted using XPath. The security of the XML document depends upon where it is stored: NT file security, third party application, web server, or SQL Server. Did I say SQL Server? I did.

Storing XML in Relational Structures

An XML document is just a string, right? Therefore, it makes common sense to store my XML in SQL Server as a string (or for bigger documents, a BLOB). That way, I can let my customer name data item enjoy all of the benefits of SQL Security. There are issues here though:

- ❑ How do I retrieve my customer name?
- ❑ How can I use SQL to access XML?
- ❑ How do I add a customer?
- ❑ How can I update or delete it?

Logically, it would seem that the best way is to extract the XML document and stream it through an XML parser; then query it with XPath. The performance hit of this is large and anyway, what if I wanted to cross query an XML document with a relational dataset? You can begin to see the complexity behind this, and also see why SQL Server 2000 needs to offer different approaches to solve the problem.

Storing Relational Data Using XML

Relational data is just data, right? Just put a few tags around it and call it XML? Right, but wrong. If your goal is to duplicate a relational data set as XML then, in all seriousness, why not just keep it as relational data? XML is hierarchical by nature (cynics may refer back to how mainframes used to store data), which means that it is not meant be structured relationally.

XML documents match how human logic works; they are designed to be human readable. Everything in this world has a natural logical tree structure to it; basic scientific classification is designed around hierarchies. However, XML documents don't yet offer the flexibility for querying that relational datasets do, so the solution is to use both formats appropriately.

The remainder of this chapter will offer some ideas to help you decide when each is appropriate; the most important thing to realize here being that it is by no means as simple as it first appears.

Mapping XML and Relational Data

Mapping one relational data structure to another is relatively straightforward, as is mapping one XML structure to another. Each environment has its own tools and mechanisms to allow for internal mapping, but cross mapping is an issue. The SQL Server 2000 XML tools available offer help with this cross mapping, but the following must be considered:

- ❑ Do not expect a one-to-one mapping or, if you do, examine why you're not sticking with relational datasets.
- ❑ Instances of the same element are allowed in XML. Consider as an example a database table where you define one customer name field; if it were appropriate to have many customer name fields, you would add another table with a foreign key. In XML you do not have to do this; the `<CustomerName>` element can be duplicated many times as shown below:

```
<Customer_Details>
   <CustomerName>Ran Hawthorn</CustomerName>
   <CustomerName>Ralph Brown</CustomerName>
   <CustomerName>Fred Simmons</CustomerName>
</Customer_Details>
```

- ❑ Once you have decided upon your mapping to XML, you must be aware that you have effectively 'set in stone' that the XML document structure is no longer relational, which has its own implications. This means that you lose all the benefits of storing data relationally, but your data is now more portable.
- ❑ XML is designed to be human readable; many physical database field names are not. To what degree do you support the goal of XML to be human readable? The level of importance will depend on how your XML is going to be used. If it is destined to be consumed by another system, then this will not be a major concern.
- ❑ When converting XML structures to relational structures, consideration must be given to the resultant physical database structure. If it is a new database, then that's not really a problem, but what if you have to transform the data into an existing database structure? Perhaps some intermediate tables are required.

All of this might lead you to wonder why we shouldn't just have the whole end-to-end as either XML or relational, but for the reasons we have discussed it isn't as easy as that.

The Main Uses of XML Structures

Up to this point in the chapter, we have focused on trying to understand the precursors to web based development, application interoperability, and data interchange, and have started to consider the concept of a different type of approach. We have also looked at the complexity of relational and XML structured data mapping. Let us now look at the uses of XML in a little more detail to understand the main concepts involved.

XML for Messaging – The Other Application Wants XML

With the advent of distributed computing, the question has always been how to communicate. TCP/IP at the network level is now mostly standard. There is general agreement on using HTTP/HTML for the presentation of document-structured data.

For distributed object computing, COM/MTS, DCOM, COM+, or .NET Remoting exist for the Microsoft environment, and CORBA and IIOP exist for the heterogeneous environment. The advent of object based computing has resulted in a set of common object based approaches to solving problems irrespective of native environment. This means that if a common message format exists then, in theory, it is fairly straightforward to cross communicate between different native environments. XML-RPC, and latterly SOAP and web services, offer this facility.

XML as Document – Markup Based Document Objects

This is not a book about document management, but some of you may well have heard talk of storing document structured information (that is, not data tables) relationally. Much of this has come about because document management systems offer similar data storage and retrieval mechanisms to traditional database management systems. Document management systems also allow the storage of document 'objects' in XML format. So, the lines are blurring between documents and data, and the beauty of it all is that, at last, a similar base format is being used.

XML for Presentation – The Browser Wants Markup

Browsers love markup, in fact it is what they do; they take a HTML file and render it to the screen. OK, there are other mechanisms to allow non-HTML functionality (ActiveX, applets, Flash, audio/video streaming, and so on) but markup is preferred. The main reason is that HTTP and HTML are non-vendor specific standards based mechanisms, which provide a common approach to working with or viewing data. HTML itself is a limited tag set offering fairly simple functionality. The standards bodies (W3C in particular) acknowledge this; initiatives such as Dynamic HTML, Cascading Style Sheets, and XHTML, address the issue in great detail.

HTML is perceived by many as a solution of the past but a problem of the future. If you look at a typical HTML file, you will see information content blindly mixed up with presentation formatting (not to mention the whole swathes of script that can sometimes be found interspersed). The major XML based initiative XSL addresses this very issue by separating content from presentation, allowing us to transform XML into HTML, or any other format of our choice, for display. The key is to realize that XML is here to stay in terms of browser markup, and that is a good thing.

An Architectural View

If you think about it, XML only makes sense when viewed architecturally (OK, that's my opinion but I believe it!). What really matters is the end-to-end performance of an application and, sorry to use a cliché, getting the right data in the right format at the right time. That means not viewing each part of the chain as a separate entity requiring a separate solution. If, for example, the end application doesn't use or remotely want, XML, then why on earth provide XML as an output format? I have actually witnessed this happening more than once. Conversely, if the end application works with a markup focus, why force the poor application developer to use only relational datasets?

To be honest, the debate about where to process what, and how, is as old as the hills. For example, depending upon whom you speak to, you will get a different answer every time about where business processing should occur. Should it be in a stored procedure, or COM component, or raw SQL in an ASP page? The arbitrator in me suggests that the best approach is the one that makes common sense given the budget, size, and experience of the project team. Understand the alternatives, but choose the most appropriate technology on a case-by-case basis.

The rest of this book gives you a detailed view on the SQL Server 2000 methods currently available for handling XML. Try to think of what follows as looking at software development as a design problem, where the problem could be solved with the use of SQL Server 2000 XML tools; *not* the other approach of "We've got these tools, where shall we use them?"

Different Approaches to the Problem

This subsection aims to provide you with some answers and give you some guidelines. It also provides you with an architectural view of what is going on, so that you can best decide on an approach that is right for you. It will do this by talking about four likely paths that organizations will take in trying to solve the problems we have discussed, and will address them in terms of SQL Server 2000.

Here are our four likely paths:

1. XML end-to-end
2. Relational, end-to (near the)-end
3. Ad-hoc, just do it and see where we get
4. Let's take a step back before we decide

We will focus on five main areas of each solution:

- **Performance** – what sort of end-to-end performance will we achieve, from a user's perspective?
- **Volumes** – what volumes of data are expected to flow both in and between tiers?
- **Security** – what security is applied to the data in terms of authorization and authentication?
- **Scalability** – how is the system able to handle expected volumes over time? Is it expandable?
- **Future proofing** – how easy will it be to manage any new design requirements that may follow?

XML End-to-End

If you were to implement this solution using SQL Server 2000, the following would provide a typical approach:

- The XML documents would be stored as BLOBs.

- In order to extract the information, you would use SQL to extract the BLOB and then parse it using an MSXML parser.
- You could then apply an XSLT stylesheet to the XML document to render it within a browser.
- The XML document could be queried using XPath (using the MSXML API).
- The XML document could also be manipulated using the MSXML Document Object Model (DOM).
- Finally, the XML Document could then be persisted as a BLOB again, using a SQL `UPDATE`.

The first major question to ask is: if the relational database already exists and is stable and reliable, does it makes any sense at all to initiate a complete upheaval in order to get the data stored as XML? If the solution were a brand new one, native XML databases could be an option but you would need to consider the viability of any vendor, and decide whether the content of the application really justifies full XML end-to-end.

If we now consider this approach against the five main areas:

- **Performance** – For small systems, where the developer knowledge sits firmly outside SQL Server 2000, this could provide an acceptable solution. For medium to large-scale operations the performance of this type of application is non-comparable to existing relational methods. Having said that, this approach could be appropriate for certain parts of the application. For example, if the XML document was a set of reference data that could sit on the web server and only get updated once a week.
- **Volumes** – The MSXML parser could handle large volumes in terms of manipulating the XML document using the DOM, before it is persisted back to SQL Server. It is important to consider, however, that whilst an XML document exists in the DOM it is effectively 'in memory' and not persisted securely. Most DBAs wouldn't see the logic in not using SQL Server transactions to conduct this type of operation.
- **Security** – From a DBA's perspective (especially if the DBA is responsible for the data), any data that is handled outside the DBMS is non-supported. The answer then, as to whether it makes sense in terms of security to handle critical data in this manner, is no.
- **Scalability** – SQL Server is very efficient at handling many BLOBs and simple SQL to update and retrieve them, so this approach is easily scalable. Looking at the end-to-end view, however, it can be seen that this is not the optimal approach.
- **Future proofing** – Imagine that you are a DBA and you manage 15 different applications (in terms of the database) and they are all structured this way; you have no control over the data except to consider it as a BLOB. It would make reporting a real headache and all the years of handling structured relational data, such that it is easily readable and updateable now and in the future, would appear to have been in vain.

In summary, this approach may be suitable for smaller applications, but for larger ones other approaches should be considered.

Relational End-to (near the)-end

I call this approach the 'Reluctant Codd Approach', where DBAs see any use of XML as an erosion of the great relational world. I know I hit a spot with that comment because I know many DBAs.

Typically, this solution would utilise an existing, stable SQL Server database that performs well, and any existing applications would be thoroughbred in terms of their use of ODBC, OLE DB, and ADO. The application developers would be proficient at using ADO `Recordsets` or `DataSets`, and also readily able to use the MSXML API. They would be looking to move to ADO.NET in the near future.

- The XML data would be stored relationally just as it always has been.
- Standard SQL stored procedures, or embedded SQL in COM components, would be used to extract the `Recordsets` or `DataSets`.
- The application developer would create the required XML by turning the `Recordset` into an XML document, using the MSXML API, or ADO/ADO.NET methods for persisting XML.
- An XSLT stylesheet could be applied to the XML document in order to render it within a browser.
- The XML document could be queried using XPath (using the MSXML API).
- There would be no manipulation of the XML document because all data manipulation happens using SQL through ADO, OLE DB, ODBC, or stored procedures.

Considering this approach against the five main areas:

- **Performance** – From a DBA's perspective, there is no performance hit. SQL Server is running optimally. The additional performance hit happens outside in the application where ADO recordsets are manipulated and MSXML API is used to create the XML documents.
- **Volumes** – One could validly argue that the optimal way to insert, update, and delete records is to use tried and tested mechanisms; so the use of ADO, OLE DB, ODBC, or stored procedures is no bad thing. For high volume reads, though, where the output is required as XML, the application would suffer terribly, especially in an environment where the XML is generated on the fly.
- **Security**
- – The security is handled by tried and trusted mechanisms and the data is secure, *but* all the XML documents external to the database are not covered. OK, you can say that the root data is fully secure within the database, but what about when the data extracted from the database is only half of the XML document? For example, the other half could be data that defines the XML document structure, or a key component of it, and it might not be backed up. Even worse, the XML document might be visible to someone who shouldn't see it.
- **Scalability** – Again, the classic SQL Server application approach is highly scalable; but as the volume of XML documents that the application is creating increases, so does the processing overhead at run time. The scalability of this approach will be questioned for this reason.

- **Future proofing** – This is a main area where thought needs to be given. We have established that XML is here to stay, and that its use is going to grow and grow. Thought needs to be given to the viability of forcing every application to create its XML documents outside the database. Given that XML documents are going to become more and more prolific, there needs to be a push to get them stored natively within a highly secure environment.

In summary, this approach is just a manifestation of classic n-tier application design. It does not really address the issues of the ever-growing demand for XML. Whether another solution is needed will depend on how critical XML is to your application.

Ad Hoc

Before you read any further I have to tell you that this is the nightmare scenario. Unfortunately, it is also the most likely path for many organisations.

Probably many of the terms used are new to the reader; this book covers them in great detail, so don't worry. Come back to this section once you have considered the rest of this book and you will appreciate even more the potential for confusion and performance problems.

Typically the solution utilises one or more existing SQL Server databases and, just like any organization I have ever worked in, there are more than a few inspired application developers and/or DBAs involved. By inspired, I mean that they readily try new technologies and approaches. This in itself is not bad, but it is one thing to try out new approaches, and another to use them wherever possible in a live environment. Good planning and design becomes an afterthought (not to mention actually documenting an application). It is fair to say that deadlines and bad management practices can also heavily influence this.

- Some applications (or parts of applications) would store XML documents as BLOBs; some would maintain the relational storage of the data, to be converted to XML at runtime. Other applications (or parts of applications) would make use of `OPENXML` in order to provide a relational view of an XML document. Many would focus their efforts on keeping the data relationally but using SQLXML to retrieve it as XML that matches a predefined schema (good schema creation is another story). Another application could use URL based queries directly. Finally, some applications would continue to use ADO XML functionality to extract XML from a relational source.
- Application developers would then use the MSXML API to parse the XML document before use. Perhaps some would retrieve the data as XML using XPath queries on SQL Server, thereby not needing to use MSXML at all. XSLT could be applied appropriately to any XML document, but where should the XSLT file be stored? After all, it is only an XML document, perhaps a BLOB, in SQL Server or cached on the web server.
- In terms of updating SQL Server, many applications would use the Updategram functionality. Others would use the SQLXML bulk load facility. The remainder would use the classic relational mechanisms.

Considering this approach against the five main areas, it is fair to say the following:

- **Performance, Volumes, Security, Scalability** – Because SQL Server 2000 tools have been used (in places) it would be fair to say that the retrieval of XML documents could perform well and be a scalable solution. Volumes and security could have been handled properly, but without an end-to-end view it is hard to say anything concretely.
- **Future proofing** – It should be fairly obvious that the many approaches, used in the many different possible ways, probably do not make any solution future proof. Just consider the data itself for example. Some would be held relationally, some as XML, some would be persisted in the database, and some outside. This would be really ugly.

In summary, this solution isn't a solution at all. It is a badly managed approach that results in loss of control, inability to plan any future application development, and no overall view of the environment. This approach is not recommended, so let's move on to look at one that is.

Take a Step Back

This solution uses the opposite approach to the previous solution, starting at the front end rather than the back end. There are four main types of XML document used here:

- **Document-focused** – The XML produced relates directly to a human readable flow of information; for example, a company report where the report paragraphs are stored in a database.
- **Data-focused** – The XML produced is an XML representation of basic tabular data; for example, sales figures by region for a report.
- **Document and data focused** – A combination of both of the above.
- **A system XML file that is used by the application** – An XSLT stylesheet, for example.

Storage

All the documents above are XML based, but the difference is in the way they naturally fit (or not) into a relational structure, and the issue is whether it makes sense to pull them apart to store them. For example, it probably would not make sense to store an XSLT file in anything other than a BLOB, as it would only be used as a whole. For performance reasons, it may not make sense to store it in SQL Server at all, and instead keep it cached on the web server.

As a general rule, the more data-centric an XML document is, the more it makes sense to store it relationally.

Data stored relationally could then be extracted using the SQLXML 3.0 Managed Classes, or the `FOR XML` (`AUTO` or `EXPLICIT`) functionality.

Conversely, the more document-centric an XML document, the more it makes sense to store it as XML.

If the document is hardly ever manipulated, or the majority is not touched when retrieving it, then storing it as a BLOB, or on a file system and using `OPENXML` to put the data in the database if you wanted to manipulate it, makes sense.

The next chapter explains exactly what each of the new technologies mentioned above are, and how they fit into SQL Server architecture.

Retrieval

This basically comes down to how it is best to persist the (document- or data-centric) document combined with the performance required for output.

If the document stored is data-centric (that is, held in SQL Server relationally), and the speed of the output requirement is not critical, then any of the SQLXML methods for retrieval would make sense, depending upon how formal you want the XML to be (`FOR XML RAW`, for example, is not 'beautiful' XML). If the requirement is for XML documents that contain different data, then use of the XPath query mechanisms should be considered.

For critical output, **pre-fetching** must be considered. Pre-fetching could mean that XML documents are persisted on the web server as part of a daily batch run, or SQLXML routines update a BLOB field on occasion with an XML document.

The more critical the retrieval performance required, the safer it is to pre-fetch as XML or store as XML.

Updates

This is perhaps the hardest area to decide upon, given the relative performance of pure SQL Server data processes against the XML routines.

It all depends on the amount of XML that is contained in your application. If your application is purely XML structures on the front end, and it is not critical (let's face it, most applications aren't that critical), then the use of Updategrams, `OPENXML`, or SQLXML Bulk Load should be considered as an alternative to using the MSXML API and ADO connections.

If performance isn't critical, and your application has a logical mix of XML on the front end and relational data on the back, then the SQL Server 2000 XML update mechanisms make sense.

Interoperability

This may seem glaringly obvious, but it is important to consider an application alongside any other applications it might interface with.

If the interoperating applications are mainly relational, then it is completely sensible to keep the interfaces relational.

Another important factor to consider is that it may make sense to design an application so that it has many possible end-to-end views in order to match the end systems. For example, an XML interface for one part, and relational for another. The key is to be aware of where you are using what, and why.

To summarize this solution, we will consider this approach against our five main areas:

- **Performance** – It should now be clear that end-to-end performance is what counts, and the type of XML document required has a large impact upon the best approach. It is no longer relevant to just benchmark against traditional retrieval methods when the use of XML is vital at the front end.
- **Volumes** – The volumes involved generally dictate how optimal the XML handling should be. For small applications, it could well make sense to just use ADO and the MSXML API, but for generic data loads, the SQLXML Bulk Load facility is appropriate.
- **Security** – Anything that occurs within an IIS and SQL Server environment can only be as secure as the administrators can make it. Common sense applies when considering what data to make available over the Internet. For example, I would not be happy exposing any highly secret corporate data via a URL query to all Internet users.
- **Scalability** – Because an application will be defined by the type of XML document it processes (data- or document-centric), scalability should be a fairly straightforward consideration. If, for example, the right decision is made in the short term about processing a data-centric XML document using Updategrams, then this decision could be expected to scale, as long as the XML document types remain constant.
- **Future proofing** – as a concept, future proofing is all about planning. By planning an implementation, as this subsection highlights, you will be aware of what is being used and why. That logical approach will then show itself throughout your SQL Server implementation, when you come to create another application.

Which Approach?

The aim of this architectural section was to show you that it is vital to consider your approach end-to-end. If you only see the database end, it is easy to assume that the application developer is doing the right thing at the other end, only to find out (or perhaps not find out!) that the performance of the application has been hit massively.

The first two approaches show what it is like to ignore completely the fact that SQL Server 2000 has tried to provide solutions for the handling of XML. Hopefully, by virtue of the fact that you are reading this book, you will appreciate that they are two extremes. The approach you really have to try and avoid is the third one. If this chapter has persuaded you to give the last one a try, then it has been successful.

Summary

In this chapter we have considered a great deal, with the goal of understanding at a high level the main factors that need to be considered by a DBA who is required to expose or store XML using SQL Server 2000:

- We considered the evolution of the n-tier architecture in order to try and establish where XML came from, and why it is important today, given its use for web presentation, application interoperability, and data interchange.
- We got a feel for the relevant XML standards and mechanisms appropriate to this book.
- We looked at XML structures and relational structures, to establish the main differences.
- XML structures and their main uses were shown to give a feel for their relevance in today's IT environment.
- Finally, and very importantly, we took an architectural view to try and gather an overall appreciation of the end-to-end use of XML.

In the next chapter we will take a look at how Microsoft has implemented functionality both within SQL Server itself, and via the SQLXML 3.0 add in, to allow us to work with XML.