PART

I

XML in the DB2 Hybrid Storage Engine

CHAPTER
1

What Is XML?

The Extensible Markup Language (XML) was born circa 1996. A "concept evolution" of previous markup languages, XML was created from a need to go beyond the simple markup of display properties to one that provided a data model for the business challenges introduced with technologies such as the World Wide Web (WWW), services-oriented architecture (SOA), and so on.

Consider the following snippet of code from a Hypertext Markup Language (HTML) document:

```
<body>
<h1>Books</h1>
<p><ul>
<li><i>A Pocket Guide to 200,000 Miles in a Year</i>
<p><b>George Baklarz</b> ID=47 <b>Paul Zikopoulos</b> ID=58
<p>35</p>
</li>
</ul>
<metadata>excessive traveling angry spouse</metadata>
</body>
```

You can see the markup surrounding the information in this example does nothing other than tell an application (for instance, a Web browser) that can process this code how to display this data. HTML does nothing to describe the data, facilitate its interchange, and so on.

XML is a *metadata* language; it's designed to describe the data within the tags and the structural relationship between them. Think of it as a model by which you can dynamically and easily define your own markup language. Metadata is data that describes data, so you can think of XML as the metadata for your markup language. You can write your own data language based on XML, which provides an efficient mechanism to define, share, store, and even validate your data—that's a heck of a lot more than telling an application to display some text in boldface.

For example, suppose a language you create based on XML, called AUTHORXML, was used to describe the data in the previous example as such:

```
<book>
<authors>
<author id="47">George Baklarz</author>
<author id="58">Paul Zikopoulos</author>
</authors>
<title>A Pocket Guide to 200,000 Miles in a Year</title>
<price>35</price>
<keywords>
<keyword>excessive traveling</keyword>
<keyword>angry spouse</keyword>
</keywords>
</book>
```

You can see that this same information in XML has become data; not just formatted text. Imagine an application interchange program that can parse this document and understand the names of the book authors and their titles. This sounds like such a simple capability, but it has radically changed the IT landscape—all because of XML.

XML provides a facility that allows you to exchange data among applications and systems without requiring changes to the application itself. And since this data-sharing ability is built on open standards, it means that you can reach across lines of business and value nets with minimal impediments.

Of course, the way the data looks to the end user is important, and you can use the related Extensible Stylesheet Language (XSL) technologies (translators, stylesheets, and so on) to shape the look of your data. Quite simply, while HTML stopped at the "glass" (in other words, at the desktop), XML leaps beyond this paradigm and into application enablement, data sharing, and more.

XML provides a paradigm that lets you define tags that describe the structure of your hierarchical data. Programmers like it because it's easy to use and flexible, and when you use XML to host data, it becomes easy to validate via another related standard named XML Schema Definition (hereafter referred to as *XML Schema*—more on this in a bit), evolve, and share. You could summarize XML as *a data model comprising nodes of several types linked through ordered parent/child relationships to form a hierarchy,* or you could just call it *a hierarchical data model*.

Beyond the application of semantic awareness to the data within a tag, XML offers (as its name implies) *extensibility*. Flexibility is the key to XML—don't forget that fact when you're reading the remaining chapters in this part of the book. Using XML, you can easily evolve your data model to accommodate new data on the fly, in a minimal amount of time (try that with a relational schema). For example, many customers today have multiple phone numbers. Adding extra phone numbers to a customer document is simple in XML. In a relational database model, it could require a new table with foreign key relationships to maintain third normal form (3NF).

XML is an open standard. Published standards tell you how to create these documents and the facilities that accompany them. This provides a technology that is assuredly easy to adopt, and you'll be able to find and share skill sets and applications built on it.

XML technology is well known to developers, but not so well known to database administrators (DBAs). We encourage DBAs to spend time investigating XML technology because a lot of data is being stored this way, and as data storage professionals, sooner or later, some of this data will wind up under your control (or you should be pushing for it to be).

The purpose of this chapter isn't to make you an XML expert, but rather to help you understand the terminology that surrounds XML, which will be helpful in understanding the XML technology in DB2 9.

Components of an XML Document

XML documents include various components and related technologies (not all of which are covered in this chapter):

- **Declarations** For example, `<?xml version='1.0' encoding='UTF-8'?>`
- **Start and end tags** For example, `<book>…</book>`

- **Attributes** For example, `id="47"`
- **Data** For example, `A Pocket Guide to 200,000 Miles in a Year`
- **Elements (nodes)** For example, `<author id="58">Paul Zikopoulos</author>`
- **Comments** For example, `<!-- This is a comment -->`

All XML documents start with an XML declaration that specifies the encoding scheme used so that an XML parser can read it, transpose it, and store it in Unicode. While an XML document can be encoded in any language, all XML parsers transform the XML data into Unicode. Other elements can be used in this declaration as well. For example, the `standalone` option can be used to declare that the XML document depends on an external file.

The term *node* is often associated with pieces of an XML document, and unfortunately, it's such an overloaded term that its use can get pretty confusing in the IT world. With respect to XML, you can use element and text nodes. At the bottom of the parsed XML representation in Figure 1-1 (in the next section) are leaf nodes that are considered text nodes (only elements have text nodes; attributes do not). Figure 1-1 shows both element nodes (`<book>`, `<title>`, and so on) and text nodes (`A Pocket Guide to 200,000 Miles in a Year`) that reside in an XML document.

You may have noticed that you could choose to use an attribute or an element to represent some of your data. For example, consider the following line from the XML code shown earlier:

```
<author id="47">George Baklarz</author>
```

This XML fragment could have been defined like so:

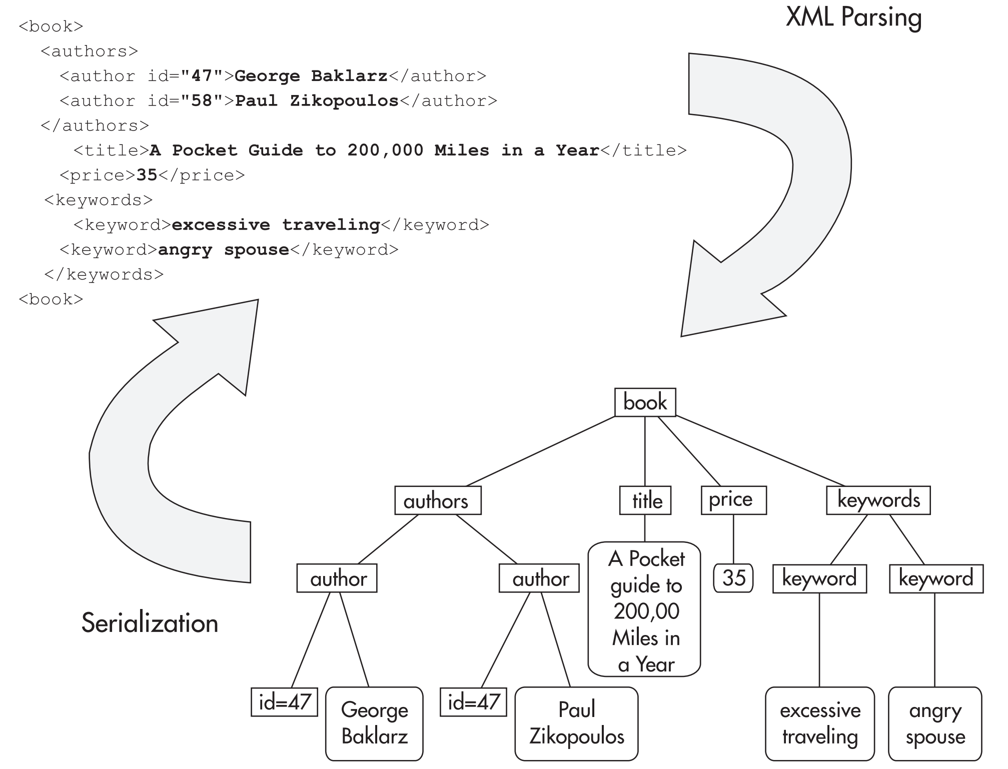
```
<author>
 <name>George Baklarz</name>
      <id>47</id>
</author>
```

Debate surrounds the decision of which approach (element or attribute) is the best way to represent this data, but that's outside the scope of this chapter.

Parsing and Serialization

When an XML document is used by an application, the application has to *parse* the data to turn the stream of text into a data structure so that it can be navigated by the application. The parsing of data is a relatively expensive operation that can use a lot of resources (and time) to accomplish. A number of parsers are available in the XML world. DB2 9 uses the XERCES open-source parser to perform parsing (with some minor modifications).

When you want to get your data back from its parsed format, it needs to be *serialized*. The parsing and serialization of an XML document is shown in Figure 1-1 (the boldface text is the actual data).



```
<book>
  <authors>
    <author id="47">George Baklarz</author>
    <author id="58">Paul Zikopoulos</author>
  </authors>
     <title>A Pocket Guide to 200,000 Miles in a Year</title>
    <price>35</price>
   <keywords>
     <keyword>excessive traveling</keyword>
    <keyword>angry spouse</keyword>
   </keywords>
<book>
```

Figure 1-1 *Parsing and serializing your data*

Figure 1-1 points out an important point about XML—only one representation of the data model is textual, while others are not. In the figure, you can see a textual and hierarchical representation of the sample XML document. Other representations include event streams, binary XML, and more. As you'll learn in reading this part of the book, the great thing about DB2 9 is that XML data is stored on disk in a parsed document object model (DOM)–like format that provides extremely fast query performance.

Well-Formed and Valid XML

An XML document must be *well-formed* for DB2 9 to store it in a pureXML column. In fact, all XML documents must be well-formed to be parsed successfully. If they are not well-formed, the parser you use will throw an exception of some sort.

A document is considered well-formed if it meets the following criteria:

- It has exactly one root element. For example, the `<book>` and `</book>` tags in this chapter's sample XML document. Each opening tag is matched by a closing tag. For example, if your XML document has a `<price>` element and no `</price>` element, it would not be well-formed. Although HTML uses the same well-formed principal, it is not strongly typed (enforced). For example, in HTML, you can follow a `<p>` tag with another `<p>` tag and the processor will imply a `</p>` tag between them (as shown in the HTML sample code at the start of this chapter)—this isn't the case in XML.

- All elements are properly nested. For example, this line is well-formed:

  ```
  <title>A Guide to 200,000 Miles in a Year</title><price>35</price>
  ```

  This line is not (note the order of the tags):

  ```
  <title>A Guide to 200,000 Miles in a Year<price>35</title></price>
  ```

- Attribute values are in quotes. For example, `<author id="58">`, not `<author id=58>`.
- It doesn't include reserved tags. For example, `<a>3<5</a>` would cause a parsing error and should be represented as `<a>3&lt;5</a>` (as `&lt;` is the appropriate XML symbol to indicate the less than sign).

An XML document is *well-formed* if it complies with all the rules in this list (the acid test is if it can be parsed by an XML parser without error). An XML document is *valid* (or typed) if it is well-formed *and* can be validated by an XML Schema or DTD (Document Type Definition) document. The XML parsers in DB2 9 can optionally perform validation against XML Schema Definition documents. If a document is well-formed but can't pass validation by an XML Schema, it is referred to as *untyped.*

So now you know that XML documents can be well-formed but invalid, but there is no way for a document to be valid if it isn't well-formed.

Quite simply, validity in terms of XML means the data structure complies with an XML Schema document—more on that in a bit. You can have an XML document and include whatever tags you want in it (and whatever data in those tags) and it will be considered well-formed so long as it complies with the terms in the preceding list. XML Schema indicates validation regarding what tags are mandatory and what type of data can go into them and so on.

XML Schema Definition Documents and DTDs

Two main types of technologies are used to validate XML documents: Document Type Definitions (DTDs) and XML Schema Definition (XSDs) documents. You can use these documents to define the structure, content, and data types for your XML documents. They define the rules as to how your XML document's hierarchical structure must look, what elements must be included, which elements are optional, and more. With DTDs and XSDs, the interchange of XML becomes a reality. When you pass either of these documents with your XML data, the receiving application knows exactly how to parse and therefore consume the data.

XSDs are a superior (and generally considered a replacement technology) for XML validation as opposed to DTDs. XSDs are better suited for XML validations for many reasons. One major difference between DTDs and XSDs is that XSDs are written in XML and DTDs are not.

In addition, XSDs are a superior method for validating XML documents when compared to DTDs, because they allow you to define data types for the elements and associated business rules for your XML documents. For example, you can specify that a particular element can only contain a specific data type, such as `INTEGER`, `BOOLEAN`, `FLOAT`, `DOUBLE`, `DECIMAL`, `STRING`, `DATE`, `DATETIME`, `BYTE`, and so on. You can add business rules to these data types. For example, you can restrict a simple `INTEGER` type between 5 and 10, or union two simple types of `INTEGER STRING`, and so on. What's more, you can create your own user-defined types, complex element types, and derived data types. Using XSDs, you not only define the data that resides within an element, but you have control over their occurrence and value range definitions, not to mention length and patterns for the data within these elements. In contrast, a DTD simply specifies what elements are allowed in an XML document.

For example, consider the following fragment:

```
<customer>
<companyname>Buy and Sell Depot</companyname>
<owner>Chris Neilson</owner>
<phone>905-898-4134</phone>
</customer>
```

If you used a DTD to validate this XML document, you could indeed ensure that the only child elements allowed within the `<customer>` parent element are `<companyname>`, `<owner>`, and `<phone>`, as you could with an XSD. However, with a DTD, you couldn't ensure that the data within any of these tags matched a specific data type or context length, or some other business rules such as default values.

With a DTD, for example, you could end up with this:

```
<customer>
<companyname>Buy and Sell Depot</companyname>
<owner>905-898-4134</owner>
<phone>Chris Neilson</phone>
</customer>
```

In this fragment, you can see that the `<phone>` element doesn't include a phone number, as you would expect. You could avoid such a data logic error by using XML Schema validation.

Another advantage of XSDs over DTDs is that they support XML namespaces (discussed in the next section). Namespaces allow you to reference different XSDs within the same document that have the same element names. XSDs are extremely flexible as well. A single XSD can be composed of multiple schema documents through import and inclusion. (For more about XSDs, see the "Wrap Up" section at the end of this chapter.)

DB2 9 can store both XML Schemas and DTDs, but can only validate your XML data using an XML Schema. If you want to validate an XML document against a DTD, you need to use the XML Extender. (We don't recommend using DTD documents to validate your

XML because it is an old technology and XML Schema is replacing older XML applications that use DTD to perform validation for all the reasons discussed earlier.)

You can store DTDs in DB2 9 even though you cannot validate against them for the purpose of entity resolution, since the XML Schema standard doesn't yet support symbols. You use symbols in XML to represent custom reserved keywords. For example, `&db2;` could be used to represent DB2 Enterprise 9. Then, whenever a document with this symbol is stored in DB2 9, the `&db2;` symbol would be changed to DB2 9 Enterprise in the on-disk format of the XML. XML contains a number of built-in symbols as well, such as the `&lt;` symbol used to represent a <, which you do not need to define.

Figure 1-2 shows how an XML Schema can contain multiple XSDs and namespaces.

XML Schema is shown here:

```
<xsd:schema  targetNamespace="http://www.mycompany/products"
                       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:simpleType name="PriceType">
         <xsd:restriction base="xsd:decimal">
              <xsd:minInclusive value="0"/>
              <xsd:maxInclusive value="100000"/>
              <xsd:totalDigits value="9"/>
              <xsd:fractionDigits value="3"/>
         </xsd:restriction>
     </xsd:simpleType>
     <xsd:complexType name="StockPriceType">
          <xsd:sequence>
               <xsd:element name="Ask" type="PriceType"/>
               <xsd:element name="Bid" type="PriceType"/>
               <xsd:element name="P50DayAvg" type="PriceType"/>
         </xsd:sequence>
     </xsd:complexType>
     <xsd:element name="StockPrice" type="StockPriceType"/>
</xsd:schema >
```

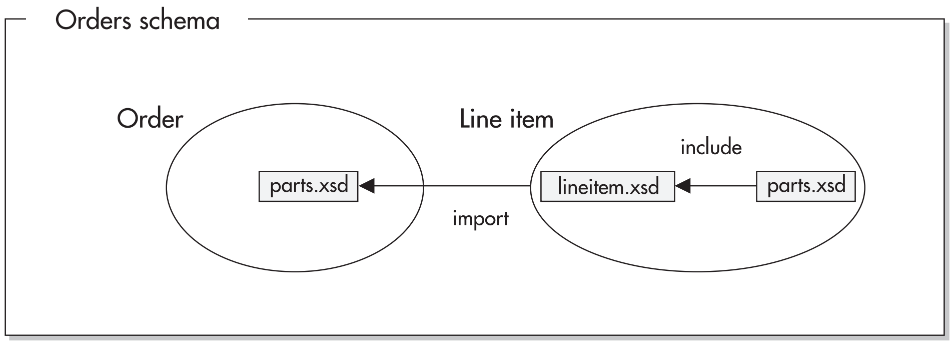


Figure 1-2 *The flexibility of XML Schema for validation*

The first thing you'll note is that an XML Schema is written in XML—this is in contrast to a DTD. Also note the reference to the namespace in the header portion of this XML document: `xmlns:xsd="http://www.w3.org/2001/XMLSchema`.

When you define an element within XML Schema, keep the following in mind:

- It cannot contain spaces—for example, `<book author>`.
- It must start with an alpha-based character or an underscore. Although you can't define an element that starts with a numeric or punctuation character, you can use these characters within an element name after the first character.
- It cannot contain reserved keywords, such as a colon (`:`).
- Like most things in XML, it is case-sensitive.

In the preceding XML Schema document, you can see the definition for a simple data type called `PriceType` (`<xsd:simpleType name="PriceType">`) that's derived from a base decimal data type (`<xsd:restriction base="xsd:decimal">`). The user-defined `PriceType` data type also has some business logic encoded within it. For example, the only allowable values that you can place within an element based on this data type are between 0 and 100,000 (`<xsd:minInclusive value="0"/><xsd:maxInclusive value="100000"/>`). Furthermore, the precision of this data type is such that it can have up to nine total digits with three fractional digits (`<xsd:totalDigits value="9"/><xsd:fractionDigits value="3"/>`).

Once a data type is defined within an XML Schema, you can source other data types from it multiple times within the data model. For example, in the preceding XML Schema you can see that the user-defined type `PriceType` is used as the base data type for a complex type `StockPriceType` (`<xsd:complexType name="StockPriceType">`). You can also see that the `StockPriceType` complex data type has multiple components to it (`Ask`, `Bid`, `P50DayAvg`) that are each based on the `PriceType` data type.

Finally, this XML Schema defines an element for your XML document called `<StockPrice>` that's based on the complex `StockPriceType` data type. An XML document that could be validated by this XML Schema document could look like this:

```
<StockPrice>
    <Ask>102.54</Ask>
    <Bid>125.871</Bid>
    <P50DayAvg>101.304</P50DayAvg>
</StockPrice>
```

Namespaces: Your Guide to Element Naming Collisions

As you can imagine, different XML Schemas may use element names that are the same. If you have multiple XML Schemas referenced from within an XML Schema document, your application could have problems understanding the data within the element tags.

Consider this example:

```
<title>Database Administrator</title>
<title>Mr.</title>
<title>DB2 9 New Features</title>
```

In these snippets of XML, the same element name (`<title>`) is used with entirely different meanings, which could result in processing and application errors. To address these types of collisions, the XML standard describes the concept of a *namespace*. A namespace is a prefix that identifies the domain of the element and can be used to distinguish among duplicate element names in an XML Schema document.

You can leverage multiple namespaces within an XML document through a defined syntax specification in an element tag. You could avoid schema collisions in this XML example by referencing each `<title>` element to the correct namespace to which it belongs, as shown here:

```
<job:title>Database Administrator</job:title>
<person:title>Mr.</person:title>
<books:title>DB2 9 New Features</books:title>
```

It's obvious here that `<job:title>` is different from `<person:title>`, as they both reside in different namespaces.

Helping a Namespace: A Universal Resource Identifier

Namespaces need to be uniquely identified, and this is accomplished through Universal Resource Identifiers (URIs). An example of a URI is http://www.ibm.com/db2xml. Though URIs look like the Universal Resource Locators (URLs) used on the Web that we all know and love, they are simply identifiers; they may point to a Web page, but they don't have to (that is, you don't have to access the Web to work with XML documents). You can learn more about URIs (on the Web) at http://www.ietf.org/rfc/rfc2396.txt.

In the following example, you can see that a local nickname (`myuri`) is defined to point to a URI (`http://www.paulleonchrisgeorge.org`) in an XML document:

```
<myuri:book xmlns:myuri="http://www.paulleonchrisgeorge.org">
   <myuri:title>DB2 9 New Features</myuri:title>
</myuri:book>
```

The reserved attribute `xmlns` is used to define a namespace and (optionally) assigns it to a namespace prefix, as shown above.

When you define a namespace within your XML document, it applies to the current element and all sub-elements and attributes that it contains. However, you can override this default rule and combine tags from different schemas within a node, as shown here:

```
<myuri:book xmlns:myuri="http://www.paulleonchrisgeorge.org">
   <myuri:title>DB2 9 New Features</myuri:title>
```

```
     <prod:product xmlns:prod="http://www.allbooks.com/product">
<prod:name>Database</prod:name>
      </prod:product>
  </myuri:book>
```

In this example, the scope of the namespace derived from the http://.www.allbooks.com/ product URI is anything between the `<prod>` and `</prod>` tags.

Default Namespaces

A namespace declaration without prefix defines a *default* namespace. A default namespace, as its name would imply, is implicit for all elements and attributes within the scope of the element if an overriding namespace prefix isn't used, as detailed in the preceding examples. For example, to declare a default namespace for an XML document, you could use the following XML fragment:

```
<book xmlns="http://www.paulleonchrisgeorge.org">
     <title>DB2 9 New Features</title>
</book>
```

XPath Expressions

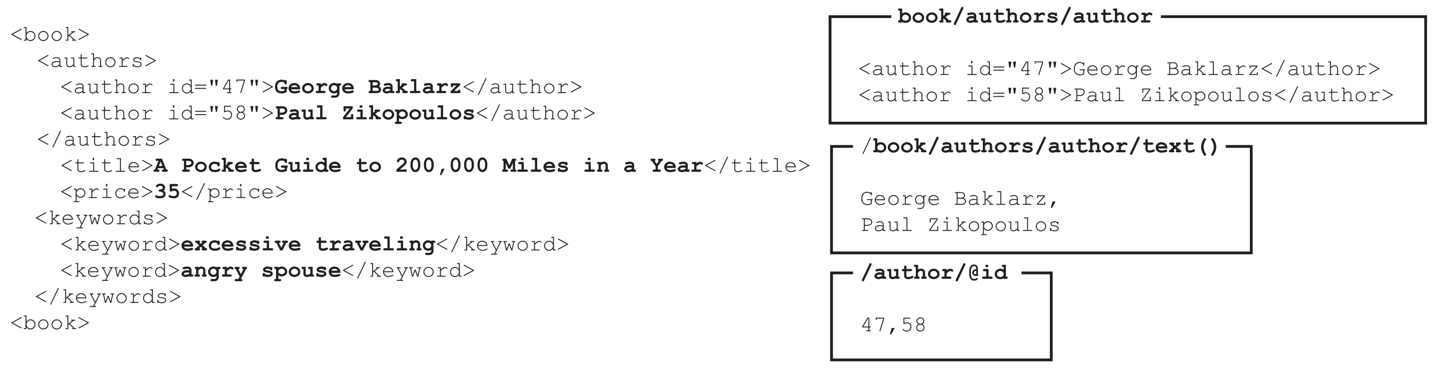
XPath is an important technology to understand when you're interfacing with XML data. XPath is an important technology in XML because it's used to locate information within a XML hierarchical data set. For example, when you use XQuery to retrieve XML data from DB2 9, you will use XPath (you'll learn more about XQuery in Chapter 13). It's beyond the scope of this chapter to dwell on XPath, but some of the concepts in this section will give you the gist of this navigational technology.

XPath has a number of hierarchical-based operators that you can use to navigate XML, as shown in the following examples:

- **Child** For example, `/authors/author`
- **Parent** For example, `/book/authors/author/../price`
- **Inclusive and descendants** For example, `//authors`
- **Attributes** For example, `/@id`
- **Self** For example, `/author/`
- **Combination of transversal functions** For example, `/author[@id = "47"]`

Examples of Navigating XML with an XPath Expression

Referring back to the XML document shown at the start of this chapter in Figure 1-1, Figure 1-3 shows the data retrieved from a sample of varying XPath expressions.



```
<book>
  <authors>
    <author id="47">George Baklarz</author>
    <author id="58">Paul Zikopoulos</author>
  </authors>
    <title>A Pocket Guide to 200,000 Miles in a Year</title>
    <price>35</price>
  <keywords>
    <keyword>excessive traveling</keyword>
    <keyword>angry spouse</keyword>
  </keywords>
<book>
```

```
book/authors/author

<author id="47">George Baklarz</author>
<author id="58">Paul Zikopoulos</author>
```

```
/book/authors/author/text()

George Baklarz,
Paul Zikopoulos
```

```
/author/@id

47,58
```

Figure 1-3 *A sample of data returned by different XPath expressions*

You can use wildcards within XPath statements to return multiple data elements within your XML document. Wildcards in XPath are denoted by the asterisk (`*`) character, while the `//` operator represents a "descendent-or-self" wildcard.

For example, the `/dept/employee/*/text()` XPath statement would return the following data from the XML document in Figure 1-3:

```
George Baklarz
Paul Zikopoulos
Pocket Guide to 200,000 Miles in a Year
35
excessive traveling
angry spouse
```

NOTE

Only the data within the data elements were returned in this example because the `text()` function was used.

XPath also supports predicates that can be used to restrict the result set that contains your XML data. XPath predicates are case-sensitive and enclosed within square brackets (`[]`). You can include multiple predicates in a single XPath declaration, and even use positional predicates. For example, the `/book/authors/author[2]/data(@id)/` XPath statement would return `58` as the data item. In this example, the XML document is navigated such that the second element's ID attribute (denoted by the `[2]`) found in the `/book/authors/author` path is returned to the application. XPath supports expressions like this one because order is a very important (and maintained) concept in XML: hierarchical order matters.

XPath also includes current context (`.`) and parental context (`..`) operators that can be used to simplify XML navigation. For example, the `/book/authors/author[./@id="58"]` XPath statement would return the data `<author>Paul Zikopoulos</author>`. In this example, the XML document is traversed down to the id attribute of the `<author>` element and its contents are returned to the application.

Defining How Your XML Looks

Recall that HTML is about display, while XML is about data. This, of course, is not the whole story: XML has a standard for defining the display of the XML data—XSL. XSL is an XML-based Stylesheet Language that you can use to define how the XML can look. You *pipe,* or transfer, the XML document, and one or more accompanying XSL stylesheets, to an XSL transformation (XSLT) engine. For example, using multiple stylesheets, you could display a single XML document on the Web in HTML, route it to a WAP display on a phone, then to a brail printer, and on and on.

This chapter won't get into XSLT, but it is sufficient to note that DB2 9 supports the calling of an XSLT engine with your XML data. DB2 Universal Database for Linux, UNIX, and Windows Version 8 (DB2 8) and DB2 9 support the functions `XSLTransformToClob()` and `XSLTransformToFile()`. They are available if the database is enabled for XML

Extender, even if you don't use the XML Extender otherwise. However, you should ask yourself if you really need to burn CPU cycles on the server to perform XSLT transformations. In many applications, this is done on the client or in the middle tier. XSLT is always CPU-intensive. Depending on what you want to do, XQuery without XSLT can sometimes be enough.

Wrap Up

A great resource for details on the XML concepts (and more) covered in this chapter is the W3 Schools Web site, at http://www.w3schools.com/. You'll find a vast array of tutorials and other resources to help you attain deep skills in XML.

Specifically, the following resources can be used to attain more specific skills that relate to XML:

- **XML in general** http://www.w3schools.com/xml/
- **Document Type Definition documents** http://www.w3schools.com/dtd/
- **XML Schema Definition documents** http://www.w3schools.com/schema/
- **XPath** http://www.w3schools.com/xpath/
- **XML Stylesheet Language and processing** http://www.w3schools.com/xsl/
- **XML namespaces** http://www.w3schools.com/xml/xml_namespaces.asp