The Definitive Guide to Berkeley DB XML

Danny Brian

Apress®

The Definitive Guide to Berkeley DB XML

Copyright © 2006 by Danny Brian

ISBN-13: 978-1-59059-666-1

ISBN-10: 1-59059-666-8

Library of Congress Cataloging-in-Publication data is available upon request.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 7

XQuery with BDB XML

XQuery is a unique query language, providing all the XML referencing power of XPath with a complete miniature scripting language. It allows modular coding and importing, mathematical operations, function definitions, results post-processing, and even reshaping and outputting of new XML. Applied to BDB XML, a single XQuery expression can query many containers (or documents) simultaneously, performing set operations on multiple data sources. With the BDB XML indexes powering the query processor, XQuery makes possible some impressive searches with huge collections of XML data.

Simply put, XQuery 1.0 is to XML what SQL is to relational database (RDB) tables: it is used to get information out of XML-formatted data. The language represents something of a coming of age for XML, with support in all major database engines (IBM, Oracle, Microsoft) and a mature W3C recommendation on its way to becoming a standard. XQuery is built on existing XML technologies, including XPath and XML Schema, making it immediately familiar to most XML users, with additional features making it sophisticated enough for complex query processing.

Appendix A, "XML Essentials," looks at XPath 1.0 and explains the fundamentals of paths, predicates, and functions. XPath 2.0 and XQuery share these basics and are supersets of the functionality thus far described. This chapter will explain XQuery basics, presuming XML and XPath knowledge on the part of the reader. XQuery is a topic on which entire books could be (and have been) written, and its details are out of the scope of this one. Nonetheless, this chapter will provide a thorough look at its core functionality. More information is available from the resources listed at the end of the chapter, and Appendix C, "XQuery Reference," contains a reference for the language's operators and functions.

Trying XQuery

All good development software enables quick experimentation, as with the BDB XML `dbxml` shell utility. The tool provides an `-s` option that allows a file containing a script to be provided as argument. This script needs to provide the same commands and syntax as the shell in interactive mode, meaning a query will take this form:

```
query 'expression'
```

The `print` command is then necessary to see the query's result.

The XQuery distribution used by and included with BDB XML 2.2.*x* also provides a command-line tool—`eval`—found in the directory `xquery-1.x/examples/eval/` of the distribution. This tool permits execution of XQuery queries stored in files on disk, making it an excellent way to debug and experiment with XQuery. It takes an XQuery file as argument, along with a number of optional parameters. BDB XML 2.3 will see the XQuery implementation moved to a new project, XQilla, in which this tool is called `xqilla`. Many of the command-line options are the same; the options for both are shown in Table 7-1, with options added by `xqilla` identified with an asterisk (*).

Table 7-1. *Options for the* `eval/xqilla` *XQuery Utility*

Option	Description
`-q`	Quiet mode, suppresses output
`-n`	Runs the query a provided number of times
`-i`	Loads the provided XML file and binds it as the context item for the query
`-b`	Sets the base Uniform Resource Identifier (URI) for the context
`-o`	Writes the result to the specified file
`-d`	Enables remote debugging on the specified `host:port`
`-p`	Parses in XPath 2 mode, as opposed to the default XQuery*
`-P`	Parses in XPath 1.0 compatibility mode, as opposed to the default XQuery*
`-t`	Outputs an XML representation of the syntax tree*

All the XQuery examples in this section can be loaded as shown into an XQuery file or executed using the `dbxml` shell (provided as a query argument). Of course, where `collection()` is used within queries to refer to a BDB XML container, only the `dbxml` shell will yield the expected results.

Two Integrated Device Electronics (IDEs) are worth mentioning in the context of BDB XML and XQuery. First, Stylus Studio is a visual editor for XML, XSL, XQuery, and XML Schema that includes integration for several databases, including BDB XML. This allows a BDB XML collection to be queried using XQuery written and debugged within the editor.

Another offering, <oXygen/>, provides a similar editor environment with support for XML diffs and merges, schemas, XSLT 1.0 and 2.0, and so on. It permits execution against BDB XML containers, environment configuration from within the IDE, as well as monitoring of debugging messages to view query plans.

Depending on your preference, there are many options to learning and using XQuery with BDB XML. After queries (and your XQuery knowledge) are refined, you'll likely either copy them into your own BDB XML application or save them as XQuery files that your application (or another query) can then load and process.

Sample Data

Throughout this chapter, the XQuery examples shown apply to a file or a collection of files similar to the `<person/>` data used in Appendix A, and the Wordnet data used in Chapter 2, "The Power of an Embedded XML Database," Chapter 4, "Getting Started," and Chapter 6, "Indexes." An example of our person data for a container `people.dbxml` is shown in Listing 7-1.

Listing 7-1. *Sample XML Data for the Container* `people.dbxml`

```
<person id="6641">
    <name>
        <last>Brown</last>
        <first>Jim</first>
        <middle>Austin</middle>
        <nick>Big</nick>
    </name>
    <age>24</age>
    <phone>
        <office>612-555-0091</office>
        <home/>
    </phone>
    <street> Attn: Jim Brown
                Pleax Systems, Inc.
                18520 25th Ave
    </street>
    <city>Minneapolis</city>
    <state>MN</state>
    <sex>male</sex>
    <hobby>boats</hobby>
    <hobby>carpentry</hobby>
</person>
```

Listing 7-2 contains an example of the Wordnet synset data; they are basically dictionary entries with numeric pointers to other entries for relationships such as “ this is a kind of X” and “Y is a part of this”. Further knowledge on this format is not necessary for understanding the examples in this chapter.

Listing 7-2. *Sample XML Data for the* `synsets.dbxml` *Container*

```
<Synset fileVersion="1.0" pos="n">
    <Id>14861</Id>
    <WnOffset version="2.1" pos="n">02772480</WnOffset>
    <LexFileNum>06</LexFileNum>
    <SsType>n</SsType>
    <Word lexId="0">baseball</Word>
    <Pointers>
        <Hypernym>14746</Hypernym>
        <Hypernym>14866</Hypernym>
    </Pointers>
    <Gloss>a ball used in playing baseball</Gloss>
</Synset>
```

We’ll be basing the example queries in this chapter on these two samples, so feel free to refer to them often.

XPath

The first thing to know about XQuery is that virtually all XPath 1.0 expressions are valid XQuery expressions, depending slightly on the implementation. The same path syntax and usage (step selections, predicates, operators, functions) are available within XQuery.

Thus, the following XPath expression will execute without trouble, assuming that the processor knows what the context node is. (Context nodes here are what the query processor sees as the “current” node within the document, making queries relative to that node.)

```
/person[@id="6641"]/name/first
```

Other programs have different ways to bind the context node. Within the `dbxml` shell, you can make the query “absolute” by using the `doc()` function:

```
doc('file:./person.xml')/person[@id='6641']/name/first
```

■ **Tip** You can also use the `contextQuery` command within the `dbxml` shell to set the context to a query's results.

Within XQuery, the same path selection syntax is used when addressing particular nodes and constructing predicates for their selection. The major difference between XPath 1.0 and 2.0 is the addition of new functions, including `doc()` and `collection()`, a complete set of data types based on XML Schema, sequence expressions, and some of the functional clauses explained later. As an example, the following XPath 2.0 expression evaluated against our `<person/>` XML will output each child element of `<name/>` using the `for` and `return` clauses:

```
for $name in (doc("file:./person.xml")/person[@id="6641"]/name/*)
return $name
```

This is equivalent to the following, which is more straightforward:

```
doc("file:./person.xml")/person[@id="6641"]/name/*
```

The use of the `for` clause to iterate the results of the path selection makes it easy to expand the query to include conditionals, additional iteration, and more advanced functionality, as you'll see.

Expressions

Every piece of an XQuery expression evaluates to a value, including the path expressions just discussed. Thus, the following is valid XQuery:

```
2+2
```

```
4
```

To see this, type `q '2+2'` right into the `dbxml` shell, followed by `print`. Individual function calls are legal as well.

```
upper-case("test string")
```

```
TEST STRING
```

Nearly all expressions can be nested in XQuery, as with this predicate:

```
doc("file:./person.xml")/person[@id="6641"]/name/*[2+2]
```

This makes for simplified debugging and testing of larger examples because they can often be broken down and evaluated in smaller parts. There are many types of expressions in XQuery, from this simple arithmetic to complex sequence comparisons and logic computations.

■ **Note** As with other XML technologies such as XPath and XSLT, XQuery is a "zero-side-effect" language, although this may change with a future version. At the present time, XQuery does not provide a means of changing XML or otherwise updating its data sources unless user-created functions are called within XQuery to accomplish the same. XQuery is a declarative language.

Knowing the value of a given expression is helpful for understanding how it affects its context within XQuery because expressions as simple as comparisons can yield different results depending on the values and their data types. BDB XML does not use XML Schema type information associated with given XML within queries; nonetheless, types defined for data and casted values within queries can yield varying results when used in comparisons and other processing. For example, an XQuery comparison such as equality (`eq`) will yield a `true` result when equal values of the same data type are compared or when equal untyped values are compared, and will yield `false` with the same conditions when the values are not equal. An error results when two values of incompatible types are compared, however. A section later in this chapter discusses data types at more length, but note that evaluating small expressions within the `dbxml` shell will quickly give you an idea of what you can expect to result from such expressions.

Expressions in XQuery can and usually do span multiple lines, as we've seen. They might themselves contain XML. This expression evaluates to `true`:

```
string(<test>Hello</test>) eq "Hello"
```

Seeing XML inside XQuery is a bit "trippy" for newcomers used to the compact syntax of XPath. As you can imagine, it plays a bit part in allowing XQuery to reshape results by itself returning XML, as seen in Listing 7-3.

Listing 7-3. *Reshaping Results*

```
for $name in (doc("file:./person.xml")/person[@id="6641"]/name/*)
return <name>{$name/string()}</name>
```

```
<name>Brown</name>
<name>Jim</name>
<name>Austin</name>
<name>Big</name>
```

Of course, this is BDB XML, and working within a single XML document is a constraint we don't have. So to accomplish something similar, we'll use `collection()` in our expression in Listing 7-4.

Listing 7-4. *Querying Document Collections*

```
for $name in (collection("people.dbxml")/person[@id="6641"]/name/*)
return <name>{$name/string()}</name>
```

We'll delve deeper into XQuery's main clauses later.

Sequences

Those familiar with XPath 1.0 will take note of the variable assignment in the previous section as storing a node set. In XQuery, what was a node set is now a *sequence*, which is much more powerful. Sequences within XQuery are constructed with parentheses:

```
(1, 2, "B", <test/>)
```

Formally speaking, a sequence is an ordered sequence of one or more items, where items are usually (but not necessarily) values. A sequence can contain values of any data type including nodes, and expressions and functions may accept and return sequences themselves. Sequences are never nested; a sequence within a sequence is seen by XQuery as a flattened list. This permits otherwise cumbersome operations to be made quite simple; for example, a general comparison of sequences:

```
(1, 4, 8) < (1, 4, 7)
```

This general less-than (`<`) comparison determines whether any values in the left sequence are less than any value in the right sequence, evaluating to `true` in this case.

In truth, every XQuery expression evaluates to a sequence, even though that sequence often contains a single item. Sequences are especially useful when it comes time to perform set operations. You'll see some examples later in this chapter.

A Complete Example

Listing 7-5 contains a complete XQuery example that demonstrates many of the language's core features, which I will dissect it into its component parts.

Listing 7-5. *A Complete XQuery Example*

```
declare namespace people = "urn:something";
declare variable $name as xs:string external;
declare function people:age-ok($age) {
    if (21 < $age and $age < 100)
        then true()
        else false()
};
(: Here is a comment. :)
<people>
    {
    for $person in collection("people.dbxml")/person
    where people:age-ok($person/age/number()) and $person/name/*/string() = $name
    order by $person/name/last
    return
    <person>
        <name>{$person/name/*}</name>
        <age>{$person/age/string()}</age>
    </person>
    }
</people>
```

This example queries for every `<person/>` in our `people.dbxml` container with an age value between 21 and 100 that has at least one name equal to the value of `$name`, has an external variable supplied by the processor (set in the `dbxml` shell using the `setVariable` command), and returns the results as an XML document with document node `<people/>`. Of course, such a query could be written many ways, but here we want to look at the various parts of such an example. The result of this query follows:

```
<people>
    <person>
        <name>
            <last>Brown</last>
            <first>Jim</first>
            <middle>Austin</middle>
            <nick>Big</nick>
        </name>
        <age>24</age>
    </person>
</people>
```

Everything prior to the comment (surrounded by a parenthesis and colon) in this example is the XQuery *prolog*. The prolog is optional, and is used to declare global and external variables, declare namespaces, import external modules, and declare modules. The rest of the example comprises the query *body*, which is evaluated to produce the query result.

The expressions within the body will ring some bells for SQL users, given the presence of the `where` clause, establishing query criteria for what functions like an SQL `SELECT` statement. Of course, the entire path selection on `people.dbxml` could have been written using a predicate expression:

```
...
for $person in collection("people.dbxml")/person[people:age-ok(age/number() and name/* eq
    $name]
...
```

Breaking it up as in Listing 7-5 makes the expression more readable and maintainable because additional logic could be added to—for example—make parts of our query criteria conditional. Suppose that we expected `$name` to be given a value by our application prior to evaluating the query, but want to accommodate cases in which no such parameter exists. Because XQuery expressions might be nested, even a `where` clause can be made conditional. As you'd expect, the `where` expression evaluates to `true` and `false` for the criteria it precedes. This makes putting an `if` conditional after `where` a simple matter:

```
...
for $person in collection("people.dbxml")/person
    where if (empty($name)) then true() else $person/name/*/string() = $name
...
```

Of course, we want this query to select records that both match `$name` and meet the age range requirement:

```
...
for $person in collection("people.dbxml")/person
    where if (empty($name)) then true() else $person/name/*/string() = $name
    and people:age-ok($person/age/number())
...
```

And that's not even doing sorting! We'll get to that shortly. Don't worry if this real world example is confusing; we'll back up momentarily and break things apart a bit more in the following sections. But first let's consider a more complex query example—this time using the `<Synset/>` files stored in `synsets.dbxml`. Recall that each record has a word or words, and also a short definition for the record stored in `<Gloss/>`. Imagine that we wanted to get the `<Id/>` of every record that has a pointer of any kind to any other record named "baseball"; perhaps we're hoping that the result set will be a good smattering of all things baseball. (SQL programmers will recognize this as a join operation.) We'd need the `<Id/>` of each baseball record first and then use it in the predicate of the pointer query. We do this in Listing 7-6 using a straight path expression.

Listing 7-6. *Performing a "Join" with a Pure Path Expression*

```
collection("synsets.dbxml")/Synset[Pointers/Hypernym =
    collection("synsets.dbxml")/Synset[Word="baseball"]/Id/string()]/Id
```

But that will only take us so far, particularly when we start performing set operations on these results. Listing 7-7 shows the more intelligible XQuery version.

Listing 7-7. *A "Join" with XQuery*

```
for $baseball in collection("synsets.dbxml")/Synset[Word="baseball"]/Id,
    $synset in collection("synsets.dbxml")/Synset
where $synset/Pointers/Hypernym = $baseball/Id
return $synset/Id
```

There are other ways this query could be written, but not all will take adequate advantage of BDB XML's indexes. In Listing 7-7, the query processor is required to retrieve all matches to both path expressions in our `for` clause before performing the comparisons. We'd be better off—depending on our indexing strategy—to store our "baseball IDs" in a sequence using XQuery's `let` clause before using it in a `where` clause to perform our select, as evidenced in Listing 7-8.

Listing 7-8. *Storing and Reusing a Sequence*

```
let $baseballs := collection("synsets.dbxml")/Synset[Word="baseball"]/Id
for $synset in collection("synsets.dbxml")/Synset
    where $synset/Pointers/Hypernym = $baseballs
return $synset/Id
```

Later in this chapter we'll discuss tips for building queries to use indexing strategies and for building queries to best use indexing strategies. We'll now look more closely at these clauses themselves.

FLWOR Expressions

FLWOR (pronounced "flower") stands for the five main clauses in XQuery: `for`, `let`, `where`, `order by`, and `return`. These clauses make many expressions simpler, but their main purpose is to construct sequences that require processing beyond path expressions, such as joins, as well as to reorder items.

A FLWOR expression consists of one or more `for` clauses, one or more `let` clauses (`for` and `let` may occur in any order), optional `where` and `order by` clauses (in that order), and a `return` clause. Each of them is described in the following subsections.

for

The `for` clause brings in to scope one or more variables. In this way it is not unlike `let`, but acts iteratively, with the variable value changing for each item in a sequence. In this way it is analogous to the start of a code "block" in traditional programming languages and the keyword `for` is appropriate for this reason. Its syntax follows this pattern:

```
for $var in expression at $position, $var2 in expression2, ...
```

We already demonstrated the introduction of multiple variables in previous examples; the keyword `at` is used to add a position variable. This variable is then updated with each iteration, storing the respective variable's value location in the sequence. Variables brought into scope by `for` remain in scope the life of the query's evaluation; they only pass out of scope with the query itself, at which point, well, there really isn't any scope.

You can have many `for` clauses in a query, which is useful when a lot of iteration is needed to perform joins.

let

The `let` clause functions like `for` to introduce variables into scope, assigning to them values, but does so without iteration. The common use is to store values that are used repeatedly, such as the "baseball IDs" expression shown earlier. XQuery variables cannot be updated repeatedly within a single scope. The syntax is the following:

```
let $var := expression, $var2 := expression2, ...
```

Keep in mind that XQuery variables can store any XQuery data type, making them useful for storing results from expression, as well as user-supplied sequences with items of any data types. Variables introduced by `let` are available to all other expressions within the query; there is no "block scope" as with traditional programming languages.

■ **Tip** XQuery is smart about the identity of nodes. When an item in a sequence is a node, variables retain the absolute identity of that node. In other words, two variables both assigned a node of the same name and string value will not equate upon comparison because they are not the same node. Two variables each assigned the exact same node (in the same external document, at the same document location, and so on) equate as expected: `true`.

where

The `where` clause supplies a condition expression for the query. This expression is like any other in XQuery, except that rather than evaluating to a sequence, it is converted to a boolean value capable of use as a conditional. This makes possible conditional conditionals, in which `where` may be following with an `if` clause. Again, as a language with nested expressions, it's easy to try individual parts of larger expressions in the `dbxml` shell to be certain of their result value.

The syntax for `where` is simply this:

```
where condition
```

Multiple conditions may be supplied by simply stringing multiple expressions together with `and`, and placing them after `where`. The functions `true()` and `false()` can be used where a condition expression needs to explicitly name the boolean. The `where` clause is nearly identical in function to a path predicate, acting as a filter for results.

order by

It wouldn't be a good query language without a versatile means of reordering results. The `order by` clause takes the query results (after being filtered with the `where` clause) and applies sorting using a supplied value and order. The syntax for `order by` is as follows:

```
order by key modifier, key2 modifier2, ...
```

The sort key is a variable or expression to be used for the sorting, and the modifier specifies a sort direction, `ascending` or `descending`. Several other modifiers are available (but not described here) to determine string sort orders and where empty values should occur in the sort. The keys with their modifiers are used in the order listed, of course.

return

This last clause of FLWOR builds the actual result returned from a query, after results are filtered and sorted. Keep in mind that `return` occurs only as the final clause in a query; as with the other FLWOR clauses, it is not allowed within expressions or even functions. The syntax is simply this:

```
return expression
```

You already saw examples of `return` that included XML to reshape results. Here again, remember that expressions themselves might contain XML, and the one provided to `return` is no different. The `return` expression can contain its own path selection, function calls, and comparisons.

■ **Tip** XQuery processors maintain what is commonly referred to as a *tuple space* while processing expressions. This is essentially a matrix containing values and sequences each time a variable is introduced with `for` and `let`, filtered with the `where` clause, and sorted with `order by`. Understanding the tuple space is not critical to using XQuery, but it does provide insight into how XQuery processors compute sequence operations through the life of a query expression using FLWOR. Refer to the XQuery specification or other online XQuery resources for more information.

Data Types

We've talked a little about XQuery's data types being derived from XML Schema. We'll fill this out a bit more in this section, but please refer to XQuery and XML Schema documentation for a full explanation of types in XQuery. You can skip this discussion and still use XQuery with BDB XML to near full capacity, and then return to it when you begin having trouble with your data types.

■ **Note** XML documents you add to BDB XML containers that have associated XML Schemas are validated only when the document is inserted into a container—and only then if validation is explicitly enabled. BDB XML does not retain this information and does not enforce the types thereafter. Within your XQuerys, you can make the schema's type declarations available using the `import schema` expression from a query's prolog.

Data types in XQuery belong to a type hierarchy. This inheritance tree determines in large part how operations see different values and what the results from operations will be. Every value in XQuery is a sequence of zero or more `items`, and `item` is the primitive XQuery data type (because sequence is not a data type). Because all values are a sequence, a single `item` is equivalent to a sequence containing that one `item` (singleton). The rest of XQuery's data types derive from `item`, but `item` is not a creatable data type (abstract). Figure 7-1 contains a tree of the common XQuery data types in their relationship to `item`. Note that all types are either types of nodes or atomic types because all descend from either `node` or `xdt:anyAtomicType`, which are also abstract.

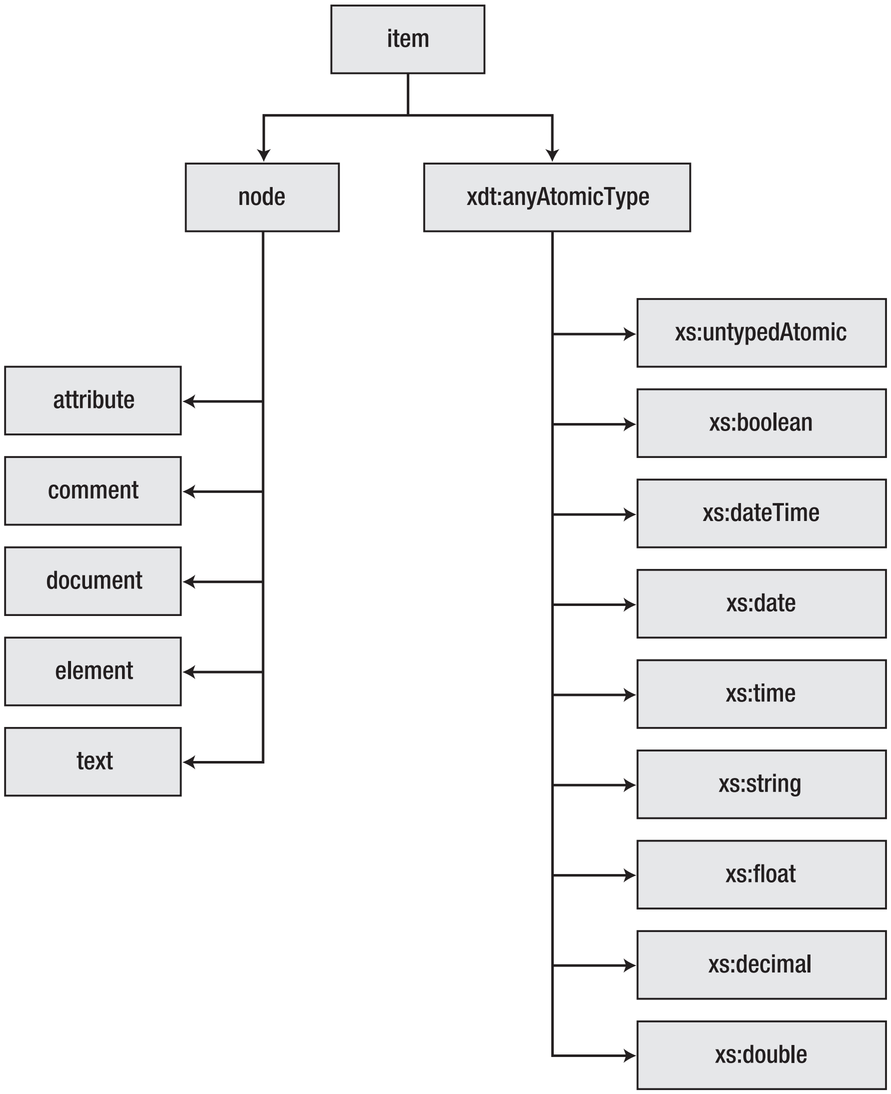

Figure 7-1. *Abbreviated XQuery data type tree*

XQuery data types can be cast into other data types using the function of the same name as the type. Thus, a `document` value can be cast as a string using `string()` and so on. Within a path expression (and when called with no arguments), these functions match the data type they represent (as with `text()` at the end of a path expression), or convert the path's value to the given data type (as `string()` at the end of a path expression). When necessary, the XQuery keyword `castable as` can be used to test an expression for "castability," given an expression before and a type after:

```
"person" castable as xs:integer
```

```
false
```

And the keyword `cast` as used to perform an explicit cast, also given an expression before and a type after:

```
<b>person</b> cast as xs:string
```

```
person
```

This, of course, has the same effect as the following:

```
string(<b>person</b>)
```

Nodes

Values of type `node` represent an XML node and have properties that include `node-name`, `attributes`, `parent`, and `children`. Because of this type inheritance, you can test any value to see if it's derived from `node` with the `node()` function. Nodes in the XQuery data model retain their identity in that they know from which document and at which location they exist, and are ordered according to those locations.

Nodes are selected from existing XML documents or by constructing them within a query. All nodes belong to a tree, each tree with exactly one root node. A tree might be a complete document (in which case the root node is of node kind `document`) or a fragment (in which case the root node is not a `document`).

Nodes operate not unlike nodes in the DOM model, in which a node's children are elements, comments, and text; its parent a document or element node (unless it's the root node).

Atomic Values

Values of types that descend from `xdt:anyAtomicType` are so called because, unlike nodes, they have no structure or inherent relationship to other values. Values of these types are the customary data types in traditional programming languages: strings, numbers, booleans, and so on. In all, XQuery has 50 different atomic types; most are specialized and see only occasional usage.

Worth noting is the `xdt:untypedAtomic` type, for values with no data type, including those from untyped XML. Values pulled from XML documents—unless a `node` or cast as a `string` —are likely to be untyped. (This is especially true with BDB XML, in which everything that isn't cast or indexed as a specific value is `untypedAtomic`.) This type behaves like `string` with few exceptions, resulting in behavior you probably expect (especially if you're coming from XPath 1.0) without problems.

Boolean values are expressed with the `true()` and `false()` functions, as has been demonstrated.

Navigation

Appendix A describes XPath 1.0 path expressions at length; they remain compatible in XQuery, and I won't expound much on what is there. In XQuery, as with XPath 1.0, a path consists of steps similar to a file directory expression. Each step depends on the previous step for its context, and defines an axis or direction. The `@` is a shortcut for the `attribute` axis, `..` for `parent`, `.` for `self`, and empty for the most-common `child` axis.

The only substantial differences with path expressions in XQuery (aside from the XQuery expressions allowed in predicates `[]`) are as follows:

- Unlike XPath 1.0, namespaces and variables in XQuery are often defined within the query, as opposed to outside of it.
- XQuery expressions can declare user functions and call them inside path expressions.
- XQuery provides many navigation functions, including `collection()` and `doc()`, which occur regularly in path expressions.
- XQuery adds several ways to establish context for a path; for example XPath 1.0 had `position()` and `last()`, XQuery adds `current-date()` and `base-uri()` to the list.

See Appendix C for a list of navigation functions and context functions in XQuery.

■ **Tip** Because XQuery allows grouping of expressions using parentheses, predicates can be applied to larger queries to apply predicates to a nested expression:

```
(collection("synsets.dbxml")/Synset[Word="baseball"])[1]
```

Comparisons

XQuery has a wide range of operators and functions, many providing similar functionality for different data types. For example, XQuery provides XPath 1.0's comparison operators (`<`, `>`, `=`, and so on), but calls them *general comparison operators*. The earlier section on sequences showed how these operators behave with multi-item sequences, in which any item in the sequences on each side of the operator can satisfy the test. This can have the unexpected result of the following expression evaluating to `true`:

```
(2, 5) = (2, 3)
```

Because the left sequence has an item (`2`) equal to an item in the right sequence (`2`), the expression is true.

As with XQuery's many data types, several other comparisons are necessary to achieve the desired result. *Value comparisons* use the letter notations (`lt`, `gt`, `eq`, and so on) to perform straight value tests, as with the following that evaluates to `false`:

```
5 lt 3
```

These comparisons work for strings and other types as you'd expect, but do not work for sequences of more than one item (nonsingleton sequences).

■ **Note** String comparisons in XQuery are implementation-specific and use an XQuery "collation" that can be modified. This is typically the Unicode code point collation, as is the case with BDB XML.

Several other comparisons are available in XQuery, including the `is` operator to compare nodes. This example evaluates to `true`:

```
let $team := doc("file:./person.xml")
return $team is doc("file:./person.xml")
```

Recall the discussion about nodes retaining their identity? The following expression is `false`:

```
<person/> is <person/>
```

This is because XQuery constructs the node on each side of the operator, making them different nodes by definition. The `is` comparison operator returns `true` only when the same node, in the same file, at the same location, is being compared to itself. For this reason, it can be useful for locating a point at which two sequences "intersect," as demonstrated in Listing 7-9.

Listing 7-9. *Navigating to a Node via Two Expressions*

```
for $person in collection("people.dbxml")//person[name/first = "Jim"]
for $known in collection("people.dbxml")//person[name/last = "Brown"]
where $person is $known
return $person
```

■ **Tip** Although not an operator, XQuery's `deep-equal()` function allows for the comparison of entire sequences as well as XML trees.

User Functions

User-defined functions are declared in a query's prolog or imported as part of a separate module. They use the following syntax:

```
declare function funcname ( parameters ) { expression };
```

Functions can also be defined to cast the function's result and to declare a function as "external" and supplied by the implementation. The parameter definition contains both typed and untyped variables that are made available to the scope of the function. They are typed using the `as` keyword, and the list is separated with a comma. Functions can include a namespace (BDB XML requires one). They cannot override built-in functions, and they cannot declare optional parameters or accept a variable number of parameters (overloaded). A simple example of a user-defined function is given in Listing 7-10.

Listing 7-10. *A Simple User-Defined Function*

```
declare namespace my = "http://brians.org/temperature";
declare function my:celsius-to-fahrenheit ($celsius as xs:decimal) as xs:decimal {
    ($celsius + 32) * (9 div 5)
};
my:celsius-to-fahrenheit(15)
```

```
84.6
```

User-defined functions can be anything allowed in a standard query body (including FLWOR expressions) as long as the declarations are proper. The evaluated value of a function given parameters is the result value for a function call, making it a convenient way to organize, test, and debug code.

Modules

User-defined functions would be more useful if they could be imported as modules, as is the case. In truth, the query expressions we have examined thus far comprise a *main module* as the XQuery engine sees things; additional modules that are loaded are *library modules*. They get loaded via the `import module` expression; most imports follow this form:

```
import module namespace at location;
```

The module itself has a module declaration of the following form:

```
module namespace namespace = uri;
```

As an example, if I had saved the example from the previous section into a file `temperature.xqm`, I would have the code contained in Listing 7-11.

Listing 7-11. *A Simple Library Module*

```
module namespace temp = "http://brians.org/temperature";
declare function temp:celsius-to-fahrenheit ($celsius as xs:decimal) {
    ($celsius + 32) * (9 div 5)
};
```

I could import it into the main module and call it as shown in Listing 7-12.

Listing 7-12. *Importing a Library Module and Calling One of Its Functions*

```
import module namespace temp = "http://brians.org/temperature" at "temperature.xqm";
temp:celsius-to-fahrenheit(10)
```

```
75.6
```

Some XQuery Tricks

The word *tricks* implies techniques that are obscure, which these are not. However, for a newcomer to XQuery (especially one familiar with SQL, XPath, or XSLT), the ease and power of many operations with XQuery is refreshing. No, it isn't *always* the case that an XQuery implementation is simpler than its SQL counterpart, but this is XML we're working with. The following are some useful examples for getting results with XQuery.

Iteration vs. Filtering

XQuery makes easy many queries that are not possible with path expressions, even though the operation seems a simple one. Consider the case in which we want to select from our `people.dbxml` record all persons with phone numbers in the 612 area code. We're tempted to use this query:

```
collection("people.dbxml")/person[starts-with(phone, "612")]
```

However, this query will fail unless the path expression before our predicate yields only one result, which it clearly does not. This is because the `starts-with()` function (as well as `contains()` and `matches()`)accepts only singleton arguments, not sequences of more than one item. Note that this is true of many operators and functions in XQuery. In such a case, FLWOR provides the solution (see Listing 7-13).

Listing 7-13. *Supplying a* `where` *Clause to Filter Results*

```
for $person in collection("people.dbxml")/person
where starts-with($person/phone, "612")
return $person
```

Not so fast. Of course, `starts-with()` again gets a first argument of a multi-item sequence because our `people.dbxml` records tend to have multiple phone numbers. One more iteration could do the trick, as evidenced in Listing 7-14.

Listing 7-14. *Two Iterations*

```
for $person in collection("people.dbxml")/person
for $phone in $person/phone
where starts-with($phone, "612")
return $person
```

We have thus created a "join" operation. Of course, this type of iteration is not always desirable because this returns the `<person/>` document each time it finds a match, which in this case is twice when both phone numbers start with "612".

The key to this query lays in the XQuery `every` and `some` condition keywords, which both begin a quantifier expression that evaluates to `true`. They are often placed after the `FLWOR` `where` clause to qualify filtered results beyond a single expression. The `some` quantifier takes this form:

```
some $variable in expression1 satisfies expression2
```

The variable (or variables) in the statement functions as introduced for the scope of both expressions, the first providing its value or iteration values (as with `for` … `in`); the second expression determining whether the expression will evaluate to `true` or `false`. The `some` operator causes the conditional to be true of any iterations of the variable that satisfy the condition expression; `every` evaluates to `true` only if all iterations satisfy the expression. Listing 7-15 gives us the proper query.

Listing 7-15. *Using a* `some` … `in` … `satisfies` *Conditional*

```
for $person in collection("people.dbxml")/person
where some $phone in $person/phone satisfies (starts-with($phone, "612"))
return $person
```

Regular Expressions

In the previous section, the `starts-with()` function was used to match an area code in a string. Although this function is retained in XQuery for compatibility with XPath 1.0, XQuery's `matches()` function is the desired replacement and offers the power of regular expression matching to the language. The equivalent example using `matches()` retains the variable as the first argument, with the second the "612" string matched to the start of the string with the ^ anchor, as seen in Listing 7-16.

■**Note** BDB XML does not currently optimize `matches()`, meaning you should only use it on small queries that do not need the benefit of indexes. The `contains()` function is optimized to use substring indexes, however.

Listing 7-16. *Using Regular Expressions to Match an Area Code*

```
for $person in collection("people.dbxml")/person
where some $phone in $person/phone satisfies (matches($phone, "^612"))
return $person
```

Of course, more-sophisticated queries using regular expressions could also include a determination of whether a phone number actually has an area code (versus "612" being the prefix of a seven-digit number):

```
matches("612-3321", "^612-\\d{3}-\\d{4}$")
```

```
false
```

Regular expression can permit several different delineating characters besides the hyphen.

```
matches("612.423.1124", "^612[-\\.]\d{3}[-\\.]\\d{4}$")
```

```
true
```

■ **Tip** For many queries to run in the `dbxml` shell, characters need escaping for the query processor to see them. This includes the backslashes in the pattern matching quotes, which need to be written as double backslashes (`\\`) because of the layers of interpolation before the expression is interpreted.

And variables can be inserted into a query; remember that the variable's content is here used as part of the regular expression. An example is given in Listing 7-17.

Listing 7-17. *Using a Dynamic Regular Expression to Match an Area Code and Phone Number*

```
let $areacode := "612"
let $match := concat("^", $areacode, "[-\\.]\\d{3}[-\\.]\\d{4}$")
return matches("612.423.1124", $match)
```

```
true
```

Querying for Metadata

BDB XML exposes document metadata attributes via the `dbxml:metadata()` function, making it easy to include metadata queries in your expressions. This function takes the metadata attribute name as argument, allowing comparisons as shown in Listing 7-18, retrieving by document name.

Listing 7-18. *Query Using a Document's Name as BDB XML Metadata*

```
for $document in collection("people.dbxml")/*
where $document[dbxml:metadata("dbxml:name") = "person1"]
return $document
```

Because BDB XML enables metadata attributes of any supported type, these comparisons can include anything from `dateTime` timestamps to price `decimal` data, allowing some complex range time and price queries, respectively.

Querying Multiple Data Sources

Comparing RDBs to XML data sources can be difficult, especially when dealing with BDB XML containers or multiple containers. Operations such as joins can happen within a single XML file, in which we could compare certain elements, groups of elements, and even XML fragments to RDB tables. But BDB XML containers are not properly analogous to tables; if a parallel is to be drawn, it would be to an RDB in its entirety. Thus, the equivalent of querying and processing data from multiple containers and documents could be performing set operations across many RDBs at once. Luckily, we won't be doing that.

Querying multiple containers or documents in a single XQuery expression is as simple as placing the input functions in our path expressions wherever we want them. Listing 7-19 shows an example of this.

Listing 7-19. *Querying Multiple Data Sources as a Join*

```
for $x in collection("people.dbxml")/person/name/first
for $y in collection("synsets.dbxml")/Synset/Word
where contains($y, $x)
return (string($x), "=>", string($y))
```

XQuery has a unary (`|`) or set union operator for computing the union between sets. Among its uses within expressions is to allow the same path expression to be tested against different containers.

```
(collection("people.dbxml") | collection("synsets.dbxml"))/person[name/first = "Jim"]
```

Because everything in XQuery is an expression, containers as well as stand-alone documents both on disk and on the network can be involved in queries using both techniques.

Recursion

Recursion is the capability of a function to call itself. With hierarchical data, recursion is a common means to repeatedly call a method to drill down with each invocation. Those familiar with XSLT are typically (and sometimes grossly) versed in the art of recursion because XSLT does not allow variables to be assigned values more than once. This prevents a variable from being used to cumulatively update a value toward the end of a total. Recursion is equally useful in XQuery because variables are assigned values only once.

Recall the `synset.dbxml` container? It holds XML documents representing synsets—similar to dictionary entries—from the Wordnet database. This data is interesting primarily because each record contains "pointers" a la foreign keys to other records, effectively telling us what things are "kinds" of other things. (A "banana" is a "fruit" is a "plant," and so on.) A recursive function is ideal for stepping up this kind of hierarchy. We're going to do this using the "banana" record, in fact, which is shown in Listing 7-20.

Listing 7-20. *The synset XML file for "banana". Stand.*

```
<Synset fileVersion="1.0" pos="n">
    <Id>41886</Id>
    <WnOffset version="2.1" pos="n">07647890</WnOffset>
    <LexFileNum>13</LexFileNum>
    <SsType>n</SsType>
    <Word lexId="0">banana</Word>
    <Pointers>
        <Hypernym>41581</Hypernym>
        <Meronym type="component">65855</Meronym>
        <Meronym type="component">65852</Meronym>
    </Pointers>
    <Gloss>elongated crescent-shaped yellow fruit with soft sweet flesh</Gloss>
</Synset>
```

More of this data can be ignored for our purposes; we'll focus on the `/Synset/Pointers/Hypernym` value, which is a "kind of" pointer to the record `/Synset[Id = 41581]`. That document will in turn have a `<Hypernym/>` element, pointing to another document, and so on until the top of this lexicon is reached. We want a function that will take the document itself as argument and give us a nicely printed list of this entry's "hypernym tree." We'll call the function `hypernyms` (it is defined in Listing 7-21).

Listing 7-21. *Recursively Processing Pointers Between Records*

```
declare namespace my = "http://brians.org/synsets";
declare function my:hypernyms ($synset) {
    let $hyp := $synset/Pointers/Hypernym[1]/string()
    return
        if (empty($hyp))
        then ($synset/Word)[1]/string()
        else
            let $next := my:hypernyms(collection("synsets.dbxml")/Synset[Id = $hyp])
            return concat($next, " => ", $synset/Word[1])
};
my:hypernyms((collection("synsets.dbxml")/Synset[Word="banana"])[1])
```

```
entity => physical entity => substance => solid => food => produce => edible fruit => banana
```

Notice the liberal use of the numeric predicates (`[1]`) throughout this example; they are to avoid errors during comparisons and functions in which a singleton is required. If we filled this example out a bit more, we'd iterate the values in those expressions to end up with more interesting output. But for this example, using only the first value in each sequence sufficed.

This next (academic?) example uses a recursive function to convert a decimal value into a string representation of its binary equivalent. Listing 7-22 shows an example of this.

Listing 7-22. *Converting Decimal to Binary Using a Recursive Function*

```
declare namespace my = "http://brians.org/temperature";
declare function my:binary ($dec as xs:decimal) {
    if ($dec eq 0 or $dec eq 1)
    then $dec
    else
        let $m := xs:integer($dec div 2)
        let $j := $dec mod 2
        let $D := my:binary($m)
        return concat(string($D), string($j))
};
```

```
my:binary(46)
```

```
101110
```

Well, you never know.

Reshaping Results

You've already seen how XQuery lets you output XML. You can do this with inline XML elements or you can use XQuery's constructors to build individual node types in XQuery variables. In Listing 7-23, we'll modify the example from Listing 7-21 to use the node constructors `element` and `attribute` to build XML containing the actual hierarchy represented in the record pointers.

Listing 7-23. *Building XML to Mirror a Conceptual Hierarchy*

```
declare namespace my = "http://brians.org/synsets";
declare function my:steps ($synset) as element() {
    let $hyp := $synset/Pointers/Hypernym[1]/string()
    return
        if (empty($hyp))
        then element step { attribute name {($synset/Word)[1]} }
        else
            let $next := my:steps(collection("synsets.dbxml")/Synset[Id = $hyp])
            return element step {
                attribute name { $synset/Word[1] },
                $next
            }
};
my:steps((collection("synsets.dbxml")/Synset[Word="banana"])[1])
```

```
<step name="banana">
    <step name="edible fruit">
        <step name="produce">
            <step name="food">
                <step name="solid">
                    <step name="substance">
                        <step name="physical entity">
                            <step name="entity"/>
                        </step>
                    </step>
                </step>
            </step>
        </step>
    </step>
</step>
```

We have results, but they aren't quite the results we're after. In fact, they're the inverse of what we want, where the `<step/>` element would have an attribute `name` with value `entity` as the root element of our fragment. We need to turn this XML inside out.

We don't have too many options for reversing things because we have to crawl "up" the hierarchy, and there's no good way to append to existing nodes. This could be done most easily by having two recursive functions: one to build our step list as a sequence (similar to what Listing 7-21 generated), and another that takes that sequence and outputs the XML to represent it. In Listing 7-24, our `hypernyms` function will return a sequence of Id strings. Note that we're reversing our sequence (using the `reverse()` function) to build the XML in inverted order and using the `remove()` function to remove the first item in the sequence with each function iteration. This time, the XML includes the IDs in preparation for the example in the next section.

Listing 7-24. *Building XML to Reflect a Conceptual Hierarchy: Take Two*

```
declare namespace my = "http://brians.org/synsets";
declare function my:hypernyms ($synset) {
    let $hyp := $synset/Pointers/Hypernym[1]/string()
    return
        if (empty($hyp))
        then $synset/Id/string()
        else
            let $next := my:hypernyms(collection("synsets.dbxml")/Synset[Id = $hyp])
            return ($synset/Id/string(), $next)
};
declare function my:tree ($idlist) {
    if (empty($idlist))
    then ()
    else
        element step {
            attribute id {$idlist[1]},
            attribute name {collection("synsets.dbxml")/Synset[Id = $idlist[1]]/Word[1]},
            my:tree(remove($idlist, 1))
        }
};
let $list := my:hypernyms((collection("synsets.dbxml")/Synset[Word="flan"])[1])
return my:tree(reverse($list))
```

```
<step id="1" name="entity">
    <step id="2" name="physical entity">
        <step id="23" name="substance">
            <step id="80020" name="solid">
                <step id="40564" name="food">
                    <step id="41580" name="produce">
                        <step id="41581" name="edible fruit">
                            <step id="41886" name="banana"/>
                        </step>
                    </step>
                </step>
            </step>
        </step>
    </step>
</step>
```

Now we have an XML hierarchy that reflects the actual hierarchy being represented.

Utilizing Hierarchy

If you're like me, there will be cases in which you'll want to use XQuery not only to pull information out of existing XML but also to use its computing power for its own sake. Where you can define the XML yourself—as we did in the previous section—the effect can be not unlike relational key tables, in which a single table is used only to store keys for mappings between tables and use them to perform complex joins.

Some readers, knowing that the `synsets.dbxml` data I've been working with is a flat hierarchy, will wonder why this hierarchy doesn't match up to an XML hierarchy. After all, isn't storing this information in XML files with hierarchy "pointers" to other files basically equivalent to using an RDB for the same task? The answer is, absolutely, and we've just seen how we could instead reflect a real conceptual hierarchy in our data hierarchy. A container full of such documents would provide for useful lookups, effectively functioning as a pure index data source.

The `dbxml` shell lets you supply a query such as this to the `putDocument` command, enabling an entire container to be populated with XQuery-generated documents. Having created a container `steps.dbxml` and added equality indexes for `id` and `name` attributes, we can issue the `putDocument` with the query and the `q` parameter. BDB XML autogenerates document names with this usage, which is fine for our purposes. I abbreviated the function definitions from Listing 7-25.

Listing 7-25. *Populating a Container with XQuery via the* `dbxml` *Shell*

```
dbxml> open steps.dbxml
dbxml> preload synsets.dbxml
dbxml> putDocument "" '
...
for $id in (1 to 1000)
let $strid := $id cast as xs:string
let $list := my:hypernyms(collection("synsets.dbxml")/Synset[Id = $strid])
return my:tree(reverse($list))
' q
```

```
Document added, name = dbxml_2, content = <step id="1" name="entity"/>
Document added, name = dbxml_3, content = <step id="1" name="entity"><step id="2"
name="physical entity"/></step>
Document added, name = dbxml_4, content = <step id="1" name="entity"><step id="3"
name="abstract entity"/></step>
...
```

Granted, this is a fairly large database container with a lot of duplicate information. (In all, the container holds 117,598 documents—some small; others large.) But the effect is that we can now query our hierarchy as if it were contained in a single XML file, with our path expressions matching the conceptual hierarchy itself. Listing 7-26 queries for all presidents (ID value 56161) in the database.

Listing 7-26. *Querying* `steps.dbxml` *for Every President of the United States*

```
collection("steps.dbxml")//*[@id = "56161"]//*
```

```
<step id="58176" name="Adams"/>
<step id="58177" name="Adams"/>
<step id="58266" name="Arthur"/>
<step id="58509" name="Buchanan"/>
```

```
<step id="58539" name="Bush"/>
<step id="58541" name="Bush"/>
<step id="58589" name="Carter"/>
<step id="58677" name="Cleveland"/>
<step id="58680" name="Clinton"/>
...
```

We would use these IDs to retrieve the full records from `synsets.dbxml`.

■Note The `steps.dbxml` examples in this section don't reflect the fact that records in Wordnet can have multiple hypernyms ("kind of" pointers), or that words can exist in many synsets. Nor are they taking into account the other pointer types in Wordnet. The examples here are simplified to demonstrate functionality.

It's easy to imagine the hierarchy being used for a game of "20 Questions", in which an application first executes the query in Listing 7-27 to get a random word. Random number generation in any functional language is a fairly inelegant ordeal; rather than demonstrate a poor example, our function will rely on an externally generated random decimal between 0 and 1.

Listing 7-27. *A "Random" Record Selector*

```
declare namespace my = "http://brians.org/synsets";
declare variable $rand as xs:decimal external;
declare function my:random-synset () {
    let $count := 250000   (: the number of records for our set :)
    let $synset := (collection("steps.dbxml")//*[@id="9"]//*)[($count * $rand) cast as
        xs:integer]
    return ($synset/@id/string(), $synset/@name/string())
};
my:random-synset()
```

```
56056
policeman
```

When the user supplies a guess as a question of the form ("Are you a[n] *X*?"), the application parses the word, looks up its ID, and calls the `guess()` function in Listing 7-28, supplying the answer and the guess. We could imagine the user asking, "Are you a person?" and "Are you a cook?" (shown in the code at the end of the listing).

Listing 7-28. *A Question Function for "20 Questions"*

```
declare namespace my = "http://brians.org/synsets";
declare function my:guess($answerId as xs:decimal, $guessId as xs:decimal) {
    if (collection("steps.dbxml")//*[@id = $guessId]//*[@id = $answerId])
    then true()
    else false()
};
my:guess(56056, 19),   (: policeman, person :)
my:guess(56056, 53188)   (: policeman, cook :)
```

```
true, false
```

The example in Listing 7-29 queries for every record that descends from “food” (ID value 24) and has a name ending with the letters “an”.

Listing 7-29. *Searching for All Foods that End with the Letters “an”*

```
for $step in collection("steps.dbxml")//*[@id = "24"]//*
    where matches($step/@name, "an$")
    return $step/@name/string()
```

```
bran
jelly bean
marzipan
flan
veal parmesan
broad bean
vanilla bean
Parmesan
moo goo gai pan
White Russian
manhattan
```

And finally a query for any words that describe both a person (ID value “19”) and some kind of man-made thing (ID value “23”) is shown in Listing 7-30.

Listing 7-30. *Searching for All Words that Describe Both People and Things*

```
for $person in collection("steps.dbxml")//*[@id = "19"]//*/@name
for $artifact in collection("steps.dbxml")//*[@id = "23"]//*/@name
    where $person = $artifact
    return $person/string()
```

```
precursor
ace
bishop
conductor
batter
bomber
builder
cookie
cracker
joker
suit
stud
...
```

Ranges

Range queries are useful for many data types, including timestamps (finding all values within a certain time span) and decimal values such as prices (finding all products within a certain price range). XQuery’s regular expressions even allow you to test and match characters in given Unicode ranges.

Zip codes and area codes are often used to compute approximate proximity; for example, the distance of a store to a customer's location, with the information typically served by a website. Less often, exact latitude and longitude (and altitude) are used, but are becoming more common as geographic data becomes more readily available. Several XML dialects have emerged to express geographic coordinates and other data, but none are quite as striking as Google Earth's use of the KML format. Thousands of these files are available online to establish *placemarks* within the Google Earth application, and loading them into a BDB XML container gives us a chance to use them in searches. A typical (but shortened) placemark file is shown in Listing 7-31.

Listing 7-31. *Typical KML Placemark File*

```
<kml xmlns="http://earth.google.com/kml/2.0">
    <Placemark id="property_1">
        <name>Four Seasons Cairo at Nile Plaza</name>
        <description><![CDATA[<p>1089 Corniche El Nile<br/>Cairo, Egypt </p>
       <p><strong>Score: <span style="color:red;">86.09</span></strong></p>
       <p>30-story hotel near Garden City on the east bank of the river. </p>
              ...
        &#169; 2006 <em>Travel + Leisure</em></p>]]></description>
        <Point>
            <altitudeMode>relativeToGround</altitudeMode>
            <coordinates>31.229338,30.03595,0</coordinates>
        </Point>
    </Placemark>
</kml>
```

Notice that the `<coordinates/>` element combined the latitude, longitude, and altitude in a single string value, separated by commas. (The KML format does allow for individual longitude and latitude elements, but not all have them.) This will make the individual values difficult to query and impossible to index properly. If we were insistent on maintaining the original files, one option would be to store each individual value as a metadata attribute for the document; unfortunately, many placemarks can be stored in a single document, so this is not an option. Instead, writing new documents (to the same or a new container) will let us break these values up; we do this using the `tokenize()` function, which splits a string into a sequence of strings, provided a string on which to split. We have already added decimal equality indexes for both latitude and longitude. An example follows in Listing 7-32.

Listing 7-32. *Populating a Container with Reshaped XML*

```
dbxml> openContainer coord.dbxml
dbxml> preload kml-files.dbxml
dbxml> putDocument '' '
declare namespace google = "http://earth.google.com/kml/2.0";
for $place in collection("kml-files.dbxml")//google:Placemark
let $coord := tokenize($place/google:Point[1]/google:coordinates, ",")
return <place>
           <name>{$place/google:name/string()}</name>
           <longitude>{$coord[1]}</longitude>
           <latitude>{$coord[2]}</latitude>
         </place>' q
```

Range queries are also now fairly straightforward, given some coordinates of our own. Making the search a function that accepts a longitude value, a latitude value, and a range lets us reuse it.

Ranges here are in degrees, in which a degree is about 69.2 miles (1/360th of the earth's circumference). Listing 7-33 shows an example of this.

Listing 7-33. *Function for Range Queries*

```
declare namespace my = "http://brians.org/range";
declare function my:in-range ($myLon as xs:decimal, $myLat as xs:decimal, $range as
    xs:decimal) {
    for $place in collection("coord.dbxml")/place
        where ($place/latitude > ($myLat - $range) and $place/latitude < ($myLat +
            $range))
           and ($place/longitude > ($myLon - $range) and $place/longitude < ($myLon +
               $range))
        return $place/name/text()
};
my:in-range (-111.651862515931, 40.00652821419428, 2)
```

```
Stein Eriksen Lodge
```

Note that our range function returns text nodes instead of strings. This is intentional because nodes are required for the set operations, retaining their context identity.

■**Note** If the reader is looking for an interesting XQuery challenge, I suggest writing a function to find the record with the shortest coordinate distance from a given coordinate, without a provided range.

Unions, Intersections, and Differences

Set operations are easy with XQuery, a fact owed in large part to sequences as the underlying data list format. All the examples in this section operate only on sequences of node values.

A union—returning all nodes from two sets with duplicates removed—can be computed simply by using the `union` operator, which works on node sequences. The example query in Listing 7-34 demonstrates a union computation. In this case, the union is performed on the sequences of places "close" to two locations, generating a list of both without duplicates. This example is using the `in-range()` function from Listing 7-33.

Listing 7-34. *Union Computation On a Sequence of Nodes*

```
declare namespace my = "http://brians.org/range";
declare function my:in-range ($myLon as xs:decimal, $myLat as xs:decimal, $range as
    xs:decimal) {
    for $place in collection("coord.dbxml")/place
        where ($place/latitude > ($myLat - $range) and $place/latitude < ($myLat +
            $range))
           and ($place/longitude > ($myLon - $range) and $place/longitude < ($myLon +
               $range))
        return $place/name/text()

};

let $placesCloseToHome := my:in-range (-111.651862515931, 40.00652821419428, 12)
let $placesCloseToJim := my:in-range (-93.49764084020113, 45.01312134030998, 12)
```

```
return $placesCloseToHome union $placesCloseToJim
```

```
Little Nell
St. Regis Resort, Aspen
Grand Hotel, Minneapolis
...
```

An intersection—returning all nodes from two sets that exist in both sequences—is similarly straightforward. This example omits the declarations (as do the rest of the examples in this section) that are the same as Listing 7-34. Here, we use the intersect operator between the sequences. This could be used to tell us, for example, which luxury hotels are within a certain range from two locations.

```
...
let $placesCloseToHome := my:in-range (-111.651862515931, 40.00652821419428, 12)
let $placesCloseToJim := my:in-range (-93.51860234945821, 45.0018515180072, 12)
return $placesCloseToHome intersect $placesCloseToJim
```

```
The Broadmoor
```

The set difference identifies all nodes in one set, but not in the other. This, too, is a simple operation, using XQuery's difference operator, `except`, which returns all nodes from the first set that are not in the second set.

```
...
return $placesCloseToHome except $placesCloseToJim
```

Finally, the symmetric difference is the list of nodes that are only in one of the sequences, and not both. This is computed with a union between the difference of each set.

```
...
return ($placesCloseToHome except $placesCloseToJim)
       union ($placesCloseToJim except $placesCloseToHome)
```

It's important to remember that they set operations for node values only. XQuery does provide the `distinct-values()` function for computing the union of untyped value sequences, but does not have built-in functions for arbitrary value intersection or difference. These operations require iteration, which I'll leave as an exercise to the reader.

Indexes and Queries

When determining an indexing strategy, as described in Chapter 6, it's important to consider the types of queries you intend to execute. The opposite is true as well, in that queries within BDB XML must be written with an awareness of the existing indexing strategy. This section considers some of the previous examples with respect to BDB XML indexing, including some potential pitfalls to be avoided.

Query Plans

We won't delve much more into query plans than we already did in the previous chapter, but do not be hesitant to examine query plans for any given query to better understand how it utilizes indexes and areas in which room for improvement can be found. The `dbxml` shell's `queryPlan` command functions just like the `query` command, but instead outputs the processor's "action plan" in the form of a syntax tree for your query. Pay particular attention to the Plan elements; each starts with a key character, followed by the index description for the index that will be used to satisfy that step in the query operation. A description for each query plan element name is shown in Table 7-2 and a legend for the key characters is shown in Table 7-3. They are valid for BDB XML version 2.2 and are subject to change.

Table 7-2. *Query Plan Element Descriptions*

Element	Description
`<RQPlan/>`	Raw Query Plan; identifies query steps prior to any optimizations. The content will describe the plan before variable lookups and before pertinent indexes are determined.
`<POQPlan/>`	Partially Optimized Query Plan; includes incomplete optimizations.
`<OQPlan/>`	Optimized Query Plan; describes a query step after optimization. It names the index used to look up the query step's result.

Table 7-3. *Query Plan Key Legend*

Key	Description
`P`	Presence lookup; the named index is used to determine the presence of a node, as opposed to node value for equality or substring tests.
`V`	Value lookup; the index is used to look up the value of a node to satisfy an equality or substring test.
`R`	Range lookup; the index is used to satisfy a range query.
`D`	Document name lookup; the index is used to get a document's name as metadata.
`U`	Universal set; all documents in the container are used to satisfy the query step.
`E`	Empty set; no documents in the container are used to satisfy the query step.
`n`	Intersection; sets are being intersected to satisfy the query step.
`u`	Union; a set union operation is being performed to satisfy the query step.

Primarily the `<OQPlan/>` elements can help you understand the query results and their lookup speed. Within the `dbxml` shell, turning up verbosity (with the `setVerbose` command) will yield additional information upon query execution, including query processing times.

It's worth noting that node and edge indexes can be used to satisfy presence tests, making presence indexes the "lowest common denominator" as far as indexes are concerned. In other words, you don't need presence indexes where you have value indexes. This shortcut has a downside in that some queries will result in query plans with a `P` (presence) step where you expect to see a `V` (value) lookup. This happens when BDB XML cannot determine the correct type for a lookup from the query and tests for presence instead of value. In such cases, the query should use some explicit casting; it isn't always enough to use the data type functions (`string()`, `number()`, and so on) to convert values. In such cases, don't be afraid to declare additional variables, coping the value of another using the `cast as` expression and using the new variable in your expression.

Node Names and Wildcards

BDB XML indexes nodes using their name, not their path within an XML document. The query optimizer needs to know a node's name to effectively determine the index to use to satisfy the query. This means that any query that does not know the name of nodes for which it must test cannot take full advantage of indexes. Consider our `people.dbxml` container, with XML containing a `<phone/>` element with children `<home/>` and `<office/>`:

```
<person id="6645">
    ...
    <phone>
        <office>612-555-0133</office>
        <home>612-555-9901</home>
    </phone>
</person>
```

We may want to query for any phone number that matches 612-555-9901 and use a query something like this:

```
collection("people.dbxml")/person[phone/*/string() = "612-555-9901"]
```

Unfortunately, this query does not give the query processor enough information to utilize indexes that exist for the `<office/>` and `<home/>` nodes. Going about this with iteration (see Listing 7-35) doesn't solve the problem because the `<office/>` and `<home/>` indexes are still not being used for the lookup, as a glance at the query plan will show.

Listing 7-35. *A Wildcard Misses the Mark*

```
for $person in collection("people.dbxml")/person
    for $phone in $person/phone/*/string()
    where $phone = "612-555-9901"
    return $person
```

For this query to work with speed, we'd need to include both node names in the query:

```
collection("people.dbxml")/person[phone/office eq "612-555-9901" or phone/home eq "612-
555-9901"]
```

Or we need to use a more "XQuery style," shown in Listing 7-36.

Listing 7-36. *Iterating for Individual Element Tests*

```
for $person in collection("people.dbxml")/person
    for $office in $person/phone/office
    for $home in $person/phone/home
    where $office = "612-555-9901" or $home = "612-555-9901"
    return $person
```

It's true that XQuery is probably overkill for this example, but if instead we wanted to get the phone number itself from a record before a cross-lookup that uses it ... you get the idea.

Note that in cases in which you control or create the XML itself, indexing limitations can be effectively bypassed. Imagine that our `people.dbxml` files follow this format:

```
<person id="6645">
    ...
    <phone loc="office">612-555-0133</phone>
    <phone loc="home">612-555-9901</phone>
</person>
```

And we have an index for the `<phone/>` elements. Because we plan to perform lookups using its value, it makes more sense to organize our data in this way. Grabbing the `@loc` attribute value is a simple matter after we have the matching node, and if we did want to use the `@loc` in our query, it would just mean another index and an appended query. If we make the phone number external, we have the example in Listing 7-37.

Listing 7-37. *Better Queries When You Have Control Over the XML*

```
declare variable $phone xs:string external;
for $person in collection("people.dbxml")/person
    where $person/phone eq $phone
    return concat($person/name/first, "'s ", $person/phone[string() = $phone]/@loc,
            " phone is: ", $phone)
```

```
Julie's home phone is: 612-555-9901
```

You can see why understanding how BDB XML indexes documents is useful for writing effective queries and why understanding how queries use an indexing strategy is useful for creating that indexing strategy. But where possible, creating XML that makes it easy to key name-value pairs within your queries will result in easier queries *and* simpler indexing strategies.

Queries Against Results

BDB XML permits queries to be executed against the results from a previous query stored in an `XmlResults` API object. However, unless these queries use the `collection()` or `doc()` functions, they do not utilize BDB XML indexes. This typically isn't an issue because the results set is small to begin with, but keep in mind that if your results sets are potentially large, you won't have indexes to speed up queries when issuing subqueries against them. It's better to get as narrow a results set as you intend to use with the initial query than to depend on subsequent queries against results to get needed information.

Conclusion

BDB XML's query strength lies in XQuery, which combined with BDB XML's flexible indexing, enables some powerful processing of large document collections. This chapter has presented a cross-section of XQuery functionality; more information is available from resources dedicated to the subject, including the following web locations:

- W3C XQuery 1.0 recommendation: `http://www.w3.org/TR/xquery/`
- W3C XQuery tutorials: `http://www.w3schools.com/xquery/default.asp`
- XQuery 1.0 and XPath 2.0 data model: `http://www.w3.org/TR/xpath-datamodel/`