CHAPTER 5

Environments, Containers, and Documents

The three core components of a BDB XML database are environments, containers, and documents. This chapter examines each from the `dbxml` shell as well as the Python API's `XmlManager` class. Other languages are covered in later chapters, but each API translates fairly easily from the examples provided here. This chapter presents the concepts with an overview of their capability, rather than a complete explanation of their operation. Consider it a look at various pieces of functionality, touching on the physical concepts and many of the programming classes used when working with BDB XML.

Environments

An environment is Berkeley DB's way of managing the database memory cache, locking, and features such as transactions and logging. At its simplest, an environment is simply the location (the directory) at which your database files are stored. Note that environments are not specific to BDB XML but are also used by the underlying DB system. Thus, the information presented here is common to both BDB XML installations, as well as non-XML Berkeley DB applications.

Tip Because BDB XML uses the same underlying DB format as regular Berkeley DB databases, an environment can hold both XML and non-XML containers.

When databases are stored on disk, their location is typically an environment. A single environment can contain zero to many databases, and many environments can exist on a single file system. The accessing program or API sets the configuration for each environment. This is an important difference between networked and embedded databases. Whereas a networked database maintains configuration and data such as access permissions at the database server (and usually using a database for that information), embedded databases require that the program itself maintain most of this data. (DB environments can contain a configuration file, discussed later.) Because the application is directly opening the database files, no layer exists to enforce configuration or access rules. The result is less overhead (with one less layer), but also less at-the-ready functionality.

Tip It's possible to create environments and containers in memory, which never get written to disk (other than being paged there by virtual memory management). In fact, this is the default for environments from the `dbxml` shell.

Creating and Opening Environments

A DB environment can be created implicitly, as with most of the examples thus far. However, only by creating an environment explicitly can you use transactional and other advanced features. The `-c` option to the `dbxml` command-line tool will create a database environment in the directory specified by `-h`:

```
$ dbxml -c -h myenv/
dbxml> quit
$ ls myenv/
```

```
__db.001    __db.002
```

Notice that the directory is populated with database files (we won't go into detail on the function of these files). As demonstrated in Chapter 4, "Getting Started," you can also use the `-t` option to enable transactions for a given shell session.

Creating an environment using the BDB XML API enables more options. It is done using the `DBEnv` (or `DbEnv`) class, which gets imported with the `dbxml` package.

```
from bsddb3.db import *
from dbxml import *

environment = DBEnv()
environment.open("myenv",
    DB_CREATE|DB_INIT_LOCK|DB_INIT_LOG|DB_INIT_MPOOL|
    DB_INIT_TXN|DB_RECOVER, 0)
environment.close(0)
```

After the `environment` object is constructed, its open method is called, with the first parameter the environment's directory path, the second a series of flags (these are bitwise `or`'d together), and the last a Unix file mode, ignored on Windows and with a default of readable and writable by owner and group, specified by `0`. The environment is then closed, which can be done explicitly as shown or by letting the `XmlManager` class do this automatically.

An incomplete list of flags with their meanings is shown in Table 5-1.

Table 5-1. *Abbreviated List of Environment Open Flags*

Flag	Description
`DB_CREATE`	Creates the environment if it doesn't already exist
`DB_INIT_LOCK`	Initializes the locking subsystem, used with concurrent reads and writes
`DB_INIT_LOG`	Initializes the logging subsystem, used for database recovery
`DB_INIT_MPOOL`	Initializes the memory pool subsystem, providing a cache required for multithreaded applications
`DB_INIT_TXN`	Initializes the transaction subsystem, permitting recovery in case of an error condition within a transaction
`DB_RECOVER`	Initializes recovery, ensuring that the database files agree with the database logs

Similar flags are passed to many of BDB XML's methods and constructors to modify their behavior. They are covered fully in the API chapters (Chapters 8 through 12) as well as the API reference in Appendix B, "BDB XML API Reference."

An opened environment object can then be passed to the `XmlManager` constructor:

```
manager = XmlManager(environment, 0)
```

The second argument (`0`) indicates no flags for the constructed `XmlManager` object. Possible flags include `DBXML_ALLOW_EXTERNAL_ACCESS` to permit XQuery queries to access data sources external to a container (network or disk files), and `DBXML_ALLOW_AUTO_OPEN` to automatically open and close unopened containers when an XQuery requires it. As with the environment `open()` method, these flags are bitwise `or`'d when used as a constructor argument.

■ **Note** Some language APIs implement slightly different usages with the flags. For example, the Java API uses configuration objects in place of flags for all options.

Additional Environment Configuration

In addition to those already mentioned, the `DBEnv` class provides methods for configuring and manipulating environments. These methods include `dbremove()` for deleting databases and `dbrename()` for moving them.

The `DBEnv` method `set_flags()` provides a long list of flags for manipulation of the environment's behavior, many specific to debugging and fine performance tuning. These flags can also be stored in a configuration file called `DB_CONFIG` within the environment directory. In this case, a single configuration line begins with `set_flags`, followed by a single flag parameter. For example, to cause an environment to automatically remove log files that are no longer needed (not necessarily the best practice), add this line to the `DB_CONFIG` file:

```
set_flags DB_LOG_AUTOREMOVE
```

Flags set in the configuration file will silently overrule application configuration. For this reason, it's a good idea to set flags in the configuration file when you want them enforced for the environment because setting flags with the `set_flags()` method typically affects only that environment handle or object instance. Table 5-2 shows an abbreviated list of settable flags.

Table 5-2. *Abbreviated List of Environment Set Flags*

Flag	Description
`DB_CDB_ALLDB`	Forces DB applications to perform environment-wide locking, rather than per-database locking
`DB_DSYNC_LOG`	Flushes writes to the log files before returning from log write calls
`DB_LOG_AUTOREMOVE`	Removes log files that are no longer in use

Other `set_*` methods enable you to change an environment's cache size, set error callbacks and error message prefixes, and configure locking timeouts. Lastly, `DBEnv` has some informational methods to get the home directory for the environment, retrieve the flags used to open the environment, and print environment statistics. (Complete lists of these methods and their uses are found in Appendix B.)

BDB XML includes several command-line utilities (in `install/bin` for Unix builds) that take an environment path as their argument. The `-h` option is standard for specifying the environment (or home directory) path to all of these. For example, the `db_archive` utility outputs the filenames of any logs that are no longer in use by the environment:

```
$ db_archive -h myenv/
```

The resulting list of files can then be moved to backup media or deleted without interfering with the operation of the databases in that environment.

■ **Caution** Permanently deleting log files not in use (as opposed to moving them to a restorable location) will usually make catastrophic recovery (in which the database itself is lost instead of just corrupted) impossible.

For a full reference of command-line utilities and their uses, see Chapter 13, "Managing Databases."

Containers

A BDB XML database is a container, which is a single file on disk that contains all documents within that database, as well as any indexes or metadata. Containers are created within a database environment (as files are created within directories). In previous chapters, in which containers were created in the absence of an explicit environment, the `XmlManager` object itself created a rudimentary environment for the storage of that container.

Containers do not themselves have or maintain any configuration data that does not correspond to the documents they contain. That is, any database settings that concern rules such as file locking or performance settings such as cache behavior are *not* part of containers, so you shouldn't look for them here. All such options are set at the environment to which a container belongs.

One more relationship between environments and containers warrants mention: stand-alone containers are portable—which means that you can copy the file itself to another location and still use it and all its indexes, assuming that you didn't copy it in midwrite—but contain no historical or transactional data. Thus, if you have not configured an environment with logging and transactions, recovery in case of database corruption will not be possible. A database's portability is a nice benefit of an embedded database, but keep in mind that the environment supplies everything that is not definable as "live data."

Creating and Opening Containers

Containers are easily created by using the `dbxml` shell, as previously demonstrated:

```
dbxml> createContainer synsets.dbxml
```

```
Creating document storage container with nodes indexed
```

This example creates the container file, `synsets.dbxml`, and opens it for operation. The API is almost identical:

```
manager = XmlManager(environment, 0)
container = manager.createContainer("synsets.dbxml")
```

The `createContainer()` method has multiple optional parameters, including a transaction object and a series of container flags. The following example creates a new container and sets a validation flag, causing XML documents subsequently loaded to be validated if they refer to a DTD or XML Schema:

```
container = manager.createContainer("synsets.dbxml", DBXML_ALLOW_VALIDATION)
```

Other common flags to create containers include `DB_RDONLY` to open a container in read-only mode (in which attempted writes will fail) and `DBXML_TRANSACTIONAL` to enable transaction support for the container.

Existing containers can be opened within the `dbxml` shell using the `openContainer` command:

```
dbxml> openContainer synsets.dbxml
```

The Python API's `openContainer()` method has the same format as `closeContainer()`:

```
container = manager.openContainer("synsets.dbxml")
```

The flags for `openContainer()` are the same as those for `createContainer()`. In fact, the methods are the same, with `createContainer()` always enforcing the `DB_CREATE` and `DB_EXCL` flags, which create the database if it doesn't exist and throw an error if it does, respectively. Thus, containers can be created using `openContainer()` and these flags. A complete list is found in the API chapters. A third `XmlManager` method, `existsContainer()`, enables you to test for the existence of a container with a single argument: the name of the container.

Container Types

BDB XML supports two types of containers, each entailing a slightly different storage technique. The container type can be set only at the time a container is created because it affects how documents are stored in the container and how its documents are indexed.

Containers of type `Wholedoc` store XML documents exactly as they are given to the storage methods, retaining document white space. By contrast, `Node` containers process the document prior to storage and then store documents as individual nodes, with a single leaf node and all its attributes and attribute values. Thus `Node` containers are generally faster to query, but `Wholedoc` containers retrieve entire documents (as opposed to just nodes or values) more quickly because they don't have to reconstruct the document as with containers of type `Node`.

■ **Note** `Wholedoc` containers are necessary when an application requires byte equivalence for its documents, for example, to retain checksums.

A good rule is to always use the `Node` type (the default) unless you expect to often retrieve entire XML documents or if your documents are small enough that the query advantage of `Node` containers is negligible. The `Wholedoc` type is intended to store and retrieve small documents; storing documents that approach or are greater than a megabyte using `Wholedoc` is discouraged, but in practice this will depend on your own application's needs, indexing strategy, and so on.

A container type can be set within the `dbxml` shell when creating a container using the argument after the container name:

```
dbxml> createContainer synsets.dbxml d
```

```
Creating document storage container
```

The d in this example forces the creation of a Wholedoc container; an n would force node storage and is the default. The same is done programmatically via the API with the createContainer() method with an argument after the flags:

```
container = manager.createContainer("synsets.dbxml", DBXML_ALLOW_VALIDATION,
XmlContainer.WholedocContainer)
```

Using the setDefaultContainerType() method of XmlManager, you can change the default container type and then omit it from the createContainer() call:

```
manager.setDefaultContainerType(XmlContainer.WholedocContainer)
container = manager.createContainer("synsets.dbxml")
```

An additional container flag warrants mention in the context of container types. Normally when using containers of type Node (again, the default and generally recommended), BDB XML indexes documents at the document level. This means that index lookups return a list of documents instead of the individual nodes. You can change this behavior using the DBXML_INDEX_NODES flag at container creation time (but not after). The result will be that index lookups return nodes instead of documents. This can be useful when dealing with large documents and needing to get node values to match a query. DBXML_INDEX_NODES is discussed in more detail in Chapter 6, "Indexes."

Some Container Operations

Most of the common BDB XML operations are performed with or on containers. Both createContainer() and openContainer() return objects of class XmlContainer. This class in turn provides methods for many operations on the container, including adding, updating, and deleting documents; adding and removing indexes; and retrieving documents after a database query. Most of these functions are also available in the dbxml shell.

Adding Documents to a Container

The XmlContainer class supplies the method putDocument() to simply add documents to the container. It is versatile in that documents can be strings, XmlDocuments, or input streams. This example adds a document with the name doc12 to the container by using a string:

```
container = manager.openContainer("test.dbxml")
container.putDocument('doc12', '<document id="12">test</document>',
manager.createUpdateContext())
```

Keep in mind that adding many documents before indexes have been created for a container will mean expensive indexing later. Before populating your container, be sure to read the next chapter and create indexes for your database.

Listing All Documents in a Container

To verify a container's contents, it can sometimes be useful to retrieve all documents in that container. Using the Python API, you can use the XmlManager.getAllDocuments() method. This will give a glimpse at working with query results, although we aren't supplying an actual query. This call returns an XmlResults object, which is BDB XML's interface for efficiently iterating results of a query.

```
container = manager.openContainer("synsets.dbxml")
results = container.getAllDocuments(0)
for value in results:
    document = value.asDocument()
    print document.getName()
```

This will output the name for each document in the container. The `value` here is of class `XmlValue`, which has a method `asDocument()` to retrieve the value as a document, returning an object of class `XmlDocument`. Alternatively, a value could be retrieved as a string or node if our results were from a query.

Performing Queries and Listing Results

The most common operation on containers is, of course, queries. Queries can be "prepared" as with most SQL implementations, or executed on the fly. Queries are executed by using the `XmlManager` object instead of a container object because they can include multiple (open) containers. This is why the `collection()` query prefix is used.

This example issues a query and outputs the number of results:

```
container = manager.openContainer("synsets.dbxml")
results = manager.query("collection('synsets.dbxml')/Synset/Word",
manager.createQueryContext())
print results.size()
```

Note the second argument to the `query()` method. Queries require a context, which provides data such as namespace mappings and variable bindings to the query processor. In this case, we have supplied a default query context. You can then iterate the results thus:

```
for value in results:
    print value.asString()
```

Alternatively, `XmlResults` objects have `next()`, `previous()`, and `peek()` methods to more efficiently browse query results one value at a time.

■ **Tip** XQuery allows for the notion of a default container, allowing a `collection()` without argument. This is set using the `XmlQueryContext.setDefaultCollection` function and is the most recently opened container when using the `dbxml` shell.

■ **Tip** The `addAlias` API method (or corresponding shell command) can be used to create aliases for your collections. This is useful when you have unwieldy paths to a collection and want to simplify your query expressions.

Documents

A single BDB XML document consists of a name, the content, and any metadata attributes that you associate with the document. Document names are the unique identifier for a record within a container and are indexed by default for all new containers.

BDB XML works with documents using the `XmlDocument` class. Under the hood, BDB XML uses the Xerces DOM to store and manipulate documents. This makes it possible to integrate with Xerces if you have need for a DOM interface to your documents.

■ **Tip** Many BDB XML applications do not necessarily regard the database as the authoritative location for the documents it contains. This is largely a matter of preference and architecture, but because the documents in the database have often been imported or created separately from the database, the documents are retained elsewhere and possibly accessed there by the applications. In such implementations, the database serves the primary function of indexing and querying—indeed, the purpose of a database. Keeping an external collection of XML documents has other benefits, including the ability to rebuild the database at will (although proper logging should make this an uncommon operation), the ability to allow regular changes to the document collection but batch write operations to the database, and so on.

Adding Documents

As was already demonstrated, documents can be added to containers using the `XmlContainer.putDocument()` method. When a document object is supplied as argument (as opposed to an XML string), the document creation looks like this:

```
document = manager.createDocument()
document.setName('doc13')
document.setContent("<document id='13'>test</document>")
container.putDocument(document, manager.createUpdateContext())
```

The document objects themselves also provide the means to be attached to data streams (using `setContentAsXmlInputStream()`) or to use a Xerces DOM object (with `setContentAsDOM()`). For example, to set a document's content using a file (and using an input stream, rather than just loading the file's contents manually):

```
inputFile = manager.createLocalFileInputStream("files/doc12.xml")
document.setContentAsXmlInputStream(input)
```

The API also provides streaming from memory, standard input, and URLs. The Java API provides an additional input stream to feed data from the application directly. Refer to the language-specific chapters for more information.

Retrieving a Document

As has been demonstrated, the `XmlResults` object returned from `XmlManager.query()` can iterate results supplied as `XmlDocuments`:

```
container = manager.openContainer("synsets.dbxml")
results = manager.query("collection('synsets.dbxml')/Synset/Word",
manager.createQueryContext())
for value in results:
    document = value.asDocument()
    print document.getName()
    print document.getContent()
```

You can also retrieve documents directly using their document name:

```
container = manager.openContainer("synsets.dbxml")
document = container.getDocument("doc12")
print document.getContent()
```

Database queries can access an individual document directly using the `doc()` query (as opposed to `collection()`), as with the following:

```
doc("synsets.dbxml/doc12")/Synset/Word
```

Or query for a document within a container by its name:

```
collection('synsets.dbxml')/*[dbxml:metadata('dbxml:name')='doc12']
```

Moreover, queries can be performed on individual documents or sets of documents returned from previous queries using the `XmlValue` and `XmlResults` classes. `XmlValue` encapsulates a primitive node's value (roughly equivalent to a node superclass in DOM implementations) and provides DOM-like methods for navigating a node's attributes and children. Queries are covered fully in Chapter 7, "XQuery with BDB XML," and working with documents after queries and with `XmlValue` is covered in later chapters.

Replacing Documents

Replacing a document within a container is simply a matter of setting its content with the `setContent()` method and then updating it in the container with the `updateDocument()` method:

```
container = manager.openContainer("synsets.dbxml")
document = container.getDocument("doc12")
document.setContent("<document id='12'>test again</document>")
container.updateDocument(document, manager.createUpdateContext())
```

Of course, the document's content can be set in other ways, as was already demonstrated with adding documents. Note that there is no performance advantage to using `updateDocument()` to replace a document as with this example; you might as well remove it and add a new document. Therefore, `updateDocument` will most often be used when parts of a document—content or metadata—are being updated and the rest retained.

Modifying Documents Programmatically

BDB XML provides its own class, `XmlModify`, for basic document manipulation. This is convenient when you want to modify documents in a container "in place" or reuse a series of modification steps across many documents, but not replace an entire document in the database. This operation uses several parameters, including an `XmlQueryExpression` (an object to store an XQuery string identifying the portion of the document to be modified), an object type identifier (to indicate the type of information being inserted; node, text, and so on), a name, and content. Then, depending on the `XmlModify` method called, a different change is affected to the document.

For example, the following will add a node to the previously replaced document:

```
container = manager.openContainer("synsets.dbxml")
modify = manager.createModify()
queryContext = manager.createQueryContext()
updateContext = manager.createUpdateContext()

query = manager.prepare("/document", queryContext)
name = "newchild"
content = "new content"
modify.addAppendStep(query, XmlModify::Element, name, content)
document = container.getDocument("doc12")
docValue = XmlValue(document)
modify.execute(docValue, queryContext, updateContext)
```

Note that multiple steps (including appending, removing, and renaming) could be added to the `modify` object. The `execute()` method can take an `XmlResults` object as argument, enabling you to pass an entire document result set. In this case, the modification steps would be performed against every document resulting from a given query.

Deleting Documents

Deleting a document from a container is straightforward using the `XmlContainer deleteDocument()` method, which takes either the document object or document name as argument:

```
container = manager.openContainer("synsets.dbxml")
updateContext = manager.createUpdateContext()
document = container.getDocument("doc12")
container.deleteDocument(document, updateContext)
```

The document name can also be used:

```
container.deleteDocument("doc12", updateContext)
```

Transactions

Note that most of the functions being demonstrated (and all of those that change containers) also accept an `XmlTransaction` object. They are *always* taken as the first argument to the method. Deleting a document within a transaction is as follows:

```
container = manager.openContainer("synsets.dbxml")
updateContext = manager.createUpdateContext()
transaction = manager.createTransaction()
container.deleteDocument(transaction, document, updateContext)
transaction.commit()
```

Of course, transactions are possible only on a container that has transactions enabled, as described in the "Creating and Opening Containers" section.

■ **Note** When a container is opened transactionally, all modifications are transacted by BDB XML, even if you don't supply a transaction object.

Validation

If a container has the `DBXML_ALLOW_VALIDATION` flag set (see the section on creating containers), BDB XML will validate documents that contain a DTD or schema reference when documents are placed in a container. Simply place the declaration or association in the XML document, enable the validation container flag, and use the `XmlContainer.putDocument()` method with the document file or string.

Note that BDB XML does not continually enforce DTD or schema constraints on a document. That is, documents can be modified using `XmlModify` in such a way as to violate an associated DTD or schema without an error being reported.

Metadata

BDB XML enables arbitrary metadata to be associated with all documents in a container. The filenames themselves are a kind of metadata, created automatically when documents are inserted. Metadata is typically any information about a document that is not contained within that document. Metadata can be advantageously used with XML documents, because some data—particularly data that changes often, such as timestamps and authors—is cumbersome to update within an XML document, and often is better stored elsewhere. Chapter 4 demonstrated the use of metadata within the `dbxml` shell.

To add metadata to a document record, you use the `XmlDocument.setMetaData()` method. It takes as arguments a URI (optional to define a namespace for the field), an attribute name, and an attribute value (as an `XmlValue`) object. Because this is performed on document objects, it must be done before a document is placed in a container, or the document must be updated (if it was retrieved from the container) after metadata has been set.

This API example sets metadata for an existing document:

```
document = container.getDocument("doc12")
value = XmlValue("doc12")
document.setMetaData("http://www.brians.org/2005/", "author", value)
container.updateDocument(document, manager.createUpdateContext())
```

Metadata can then be indexed and queried as with data in XML documents and demonstrated in later chapters.

Note that `XmlValue` objects can be of many data types—including strings, dates, and decimals—all based on the XQuery specification. Creating a metadata field as, for example, a date can be useful when indexing and querying those fields, allowing queries to recognize ordering of dates. This example shows how to create an `XmlValue` with a data type of `DATE_TIME`, which XQuery recognizes and can compare in queries:

```
document = container.getDocument("doc12")
value = XmlValue(XmlValue.DATE_TIME, "2006-01-02T21:24:25")
document.setMetaData("http://www.brians.org/2005/ ", "timestamp", value)
```

Note that the URL can be arbitrary, but is recommended. When issuing queries for metadata, the query context must map the namespace for the query to work:

```
queryContext = manager.createQueryContext()
queryContext.setNamespace("brians", "http://www.brians.org/2005/ ")
query = manager.prepare("/*dbxml:metadata('brians:timestamp')", queryContext)
```

Metadata is a powerful and efficient way to keep track of information about documents without storing that information within the documents. Indexing, querying, and working with metadata in the various language APIs are covered in later chapters.

Conclusion

Although later chapters detail the BDB XML API, much of this chapter dealt with the classes and objects used because they map generally to the physical pieces of the database. Figure 5-1 shows the relationship of the database components discussed.

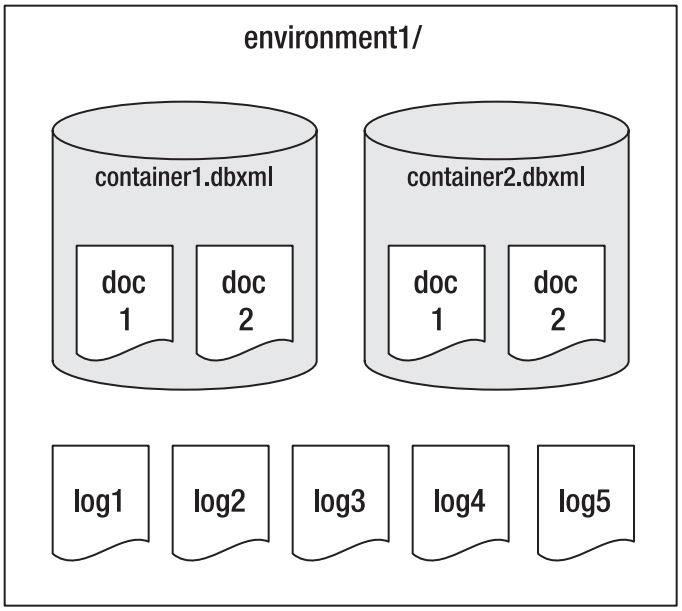

Figure 5-1. *Environments, containers, and documents*

Keep in mind the following key points:

- Environments are the outermost piece of a Berkeley database, XML or otherwise (as with non-XML Berkeley DB databases). They are both analogous to and implemented as file directories; in Figure 5-1, `environment1/` is a file directory. Although mostly optional, environments are helpful for configuring aspects of the database, and are critical to logging and recovery of data. Environments have their API expression in the `DbEnv` class, and do not contain functionality or options specific to BDB XML.
- Containers are the databases themselves. They exist as files on the file system that are directly opened, read, and manipulated by BDB XML's libraries. In Figure 5-1, the two listed containers `container1.dbxml` and `container2.dbxml` are actual files within the directory `environment1/`. Containers are managed via the API using the `XmlContainer` class and are created, opened, and otherwise manipulated with the `XmlManager` class. Containers are included in a database query, rather than executing queries on the container or container objects themselves. Thus, an XQuery to access `container1.dbxml` in Figure 5-1 would begin with `collection("container1.dbxml")/` after the environment and container had been opened. Finally, a default collection can be set and subsequently excluded from the `collection()` function.
- Documents are well-formed XML documents stored in containers; each can have arbitrary metadata associated with it. Documents can be created programmatically, expressed as a string, or loaded from a file or network stream, prior to placing them in a container. Each of the containers in Figure 5-1 contains two documents, `doc1` and `doc2`, which are XML documents. Documents can be manipulated with regard to their container using the `XmlContainer` class. Documents have their API implementation in the `XmlDocument` class—in which name, content, and metadata can be gotten and set—before being written to a container using `XmlContainer.putDocument` or `XmlContainer.updateDocument`.

- `XmlManager` is the primary class for working with BDB XML databases and is used to create (as a factory object) from scratch the most common objects for containers, documents, transactions, results, and update and query contexts. It is used to prepare and execute queries on open containers, providing context for those queries. It encapsulates a Berkeley DB environment and the open containers in that environment.
- Many API operations—including retrieving query results and changing documents in the database—entail the use of "helper" BDB XML classes, including `XmlResults` and `XmlModify`. They do not have a physical correspondence to any part of a BDB XML database.

With an understanding of the basic complements of BDB XML, we next explore more exciting parts of the database: indexes and the queries that use them.