Persistent Data Storage

31

The objective of this chapter is to introduce text-based XML files and relational databases as two ways to persistently store data.

31.1 Preconditions

The following should be true prior to starting this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models.
- You understand the process of normalization and have applied normalization forms to data designs.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_31

Table 31.1 XML examples

<student>	<student>	<student>
Sally	<name>Sally</name>	<name>Sally</name>
1234567	<id>1234567</id>	<id>1122334</id>
Computer Science	<major>Computer Science</major>	<major />
</student>	</student>	</student>

31.2 XML Concepts

This section introduces XML as a way to store data persistently.

31.2.1 What is XML?

The acronym XML stands for *eXtensible Markup Language*. It is a tag-based language used to describe data elements and their relationships to each other. A tag-based language allows us to define a data element by embedding a tag-name inside angled brackets. For example, <student> denotes the start of data associated with a student while </student> denotes the end of data for a student. In between these two tags could be data values and/or other tags.

Table 31.1 shows three examples. The example on the left shows three data values inside a student tag. The example in the middle shows tags and data embedded inside a student tag. This example illustrates how XML represents relationships between data elements using a tree (or hierarchy) structure. The example on the right shows how a single tag may represent both a start and an end tag. Specifically, <major /> tag indicates this student does not have a major. Since tag names are programmer-defined, we have control over the data element names we decide to use. For example, we could use <firstName> instead of <name> and <studentID> instead of <id>. An important aspect of creating tag names is to make these names descriptive.

A very good resource to learn more about XML is at www.w3schools.com, a free web site resource initially developed by a Norwegian company that has tutorials for many of the popular web-based technologies [1].

31.2.2 XML Files

XML may be stored in a file using one of two formats: binary or text. A binary XML file is not readable by a text editor. Instead, special software must be written to read and update a binary XML file. Examples of binary XML files you've likely used are the document files created and maintained by the Open Office suite and the

Microsoft Office suite of software applications. This book does not discuss creating or updating binary XML files.

A text-based XML file is readable by a text editor. This allows us to view the tree structure of the XML tags. We can also modify the XML tree structure, the tag names, and the data associated with any of the tags. The rest of this section will discuss creating and updating text-based XML files.

Listing 31.1 shows a sample XML file for the digital library discussed in Chap. 30. This file contains three Book instances and one Movie instance.

Listing 31.1 Digital Library XML File Sample 1

```
<digital_library>
    <book>
        <isbn>0441172717</isbn>
        <title>Dune</title>
        <publisher>Chilton Books</publisher>
        <yearPublished>1965</yearPublished>
        <author>Frank Herbert</author>
    </book>
    <book>
        <isbn>0151008116</isbn>
        <title>Life of Pi</title>
        <publisher>Harcourt</publisher>
        <yearPublished>2001</yearPublished>
        <author>Yann Martel</author>
    </book>
    <book>
        <isbn>0137135599</isbn>
        <title>Blown to Bits</title>
        <publisher>Addison Wesley</publisher>
        <yearPublished>2008</yearPublished>
        <author>Hal Abelson</author>
        <author>Ken Ledeen</author>
        <author>Harry Lewis</author>
    </book>
    <movie>
        <title>The Shawshank Redemption</title>
        <yearPublished>1994</yearPublished>
        <director>Frank Darabont</director>
    </movie>
</digital_library>
```

31.2.3 Document Object Model (DOM)

A text-based XML file is parsed into a Document Object Model (DOM). The Document Object Model [2] is:

- A specification that defines the logical structure of HTML and XML documents.
- An application programming interface (API) for HTML and XML documents.

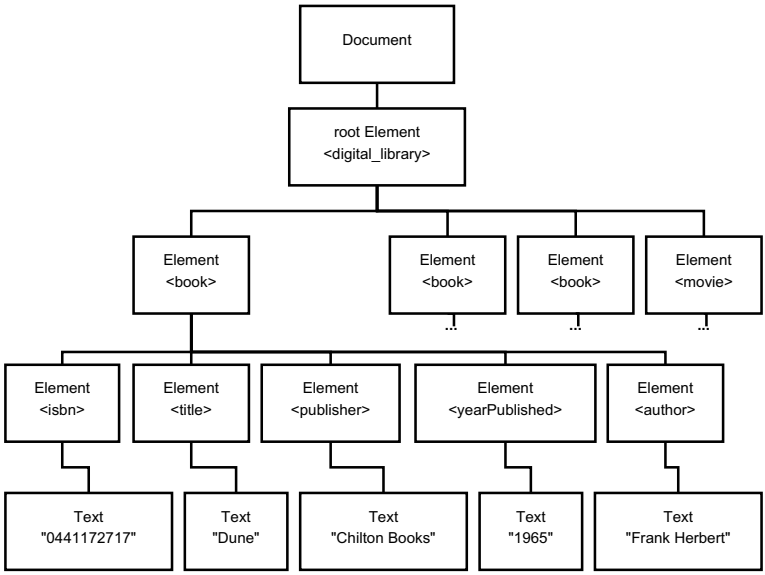

Fig. 31.1 DOM structure for XML Listing 31.1

- A World Wide Web Consortium (W3C) specification supported by all of the major web browsers.

Most of the popular programming languages, including Java and Python, provide an XML library that implements the W3C specification and associated API. The logical structure of a DOM is a tree of nodes representing the hierarchy of tags as expressed in a text-based XML file.

Figure 31.1 shows the DOM structure for the sample XML file in Listing 31.1. The hierarchy of nodes shown in this figure illustrates how an XML structure is translated into a DOM tree structure. At the top of the DOM is a Document object, which contains the entire hierarchy of nodes. Each of the XML tags is translated into an Element node, while the data values found in the XML file become Text nodes. The figure shows the entire tree structure for the first Book instance in the XML file. The other Book instances and the one Movie instance will have a similar structure.

The list below identifies the four most common types of objects found in a DOM.

Document: A container for the entire DOM tree structure and *conceptually* represents the root of the tree. A document will always contain one root Element, which represents the actual root of the tree.

Element: A start tag is translated into an Element node.

Node: A generic node type.

Text: The data associated with a start tag (i.e., Element node) is stored in a Text node.

The sample XML in Listing 31.1 uses indentation and whitespace characters to make it easy for people to read the XML text file. Listing 31.2 contains the same tags and data values as the first example but does not contain any newline or tab characters. Essentially, this text file contains one line of text by eliminating all of the whitespace characters which made the Sample 1 listing easier to read. (Important: for purposes of displaying the sample 2 XML file, newline characters were added to allow the content to be completely displayed inside the listing box.)

Listing 31.2 Digital Library XML File Sample 2

```
<digital_library><book><isbn>0441172717</isbn><title>Dune</title>
<publisher>Chilton Books</publisher><yearPublished>1965
</yearPublished><author>Frank Herbert</author></book><book><isbn>
0151008116</isbn><title>Life of Pi</title><publisher>Harcourt
</publisher><yearPublished>2001</yearPublished><author>Yann Martel
</author></book><book><isbn>0137135599</isbn><title>Blown to Bits
</title><publisher>Addison Wesley</publisher><yearPublished>2008
</yearPublished><author>Hal Abelson</author><author>Ken Ledeen
</author><author>Harry Lewis</author></book><movie><title>The
Shawshank Redemption</title><yearPublished>1994</yearPublished>
<director>Frank Darabont</director></movie></digital_library>
```

31.2.4 Java Examples

This section will introduce the use of Java to parse an XML file, to create a DOM, display the contents of a DOM, update a DOM, and to save a DOM back to an XML file.

31.2.4.1 Parse an XML File

The javax.xml.parsers package contains the classes and methods to parse an XML file to create a DOM. In Listing 31.3, the fileName parameter is the name of the XML file, including the file extension. When the DocumentBuilder parse method is successful, a Document object is returned. Any exception will cause null to be returned by the createDOM method.

Listing 31.3 Java createDom method

```
private Document createDOM(String filename)
{
    Document dom;
    DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance();
    try
    {
        DocumentBuilder db = dbf.newDocumentBuilder();
        dom = db.parse(filename);
    }
    catch (Exception ex)
```

```
    {
        System.out.println(ex);
        dom = null;
    }
    return dom;
}
```

31.2.4.2 Display a DOM

Since a DOM is a tree structure, we can display the entire DOM by moving through the tree structure using a recursive traversal algorithm. Our traversal will display information for the current node (wherever this is within the tree). It then recursively traverses the sub-tree for each child of the current node, moving through the child nodes from left-to-right.

Listing 31.4 shows the displayDOM and displayDOMhelper methods. The displayDOM method uses the Document object to obtain the root node for the DOM. It then calls the displayDOMhelper method, which implements the recursive traversal, displaying each node as it is reached. The indentLevel parameter for the recursive function keeps track of how much to indent each node as it's displayed to visually show the tree structure.

Listing 31.4 Java displayDom methods

```
private void displayDOM(Document dom)
{
    System.out.println("XML DOM node structure.\n" +
        "Each indentation is another level in the tree.");
    Element docElement = dom.getDocumentElement();
    displayDOMhelper(docElement, 0);
}

private void displayDOMhelper(Node element, int indentLevel)
{
    //display information for this element
    displayElement(element, indentLevel);

    //recursively display info for each child of this element
    if (element.hasChildNodes())
        displayDOMhelper(element.getFirstChild(), indentLevel + 1);

    //recursively display info for each sibling of this element
    Node nextSibling = element.getNextSibling();
    if (nextSibling != null)
        displayDOMhelper(nextSibling, indentLevel);
}
```

Listing 31.5 shows the display of the first Book instance from the first sample XML file shown in Listing 31.1. One of the interesting aspects associated with this DOM is it contains *extra* text nodes containing no data value. These are shown in Listing 31.5 as "type[Text],name[#text],value=[]". These nodes exist because of the whitespace characters (e.g., newline and tab characters) in the sample XML file

shown in Listing 31.1. The XML parser in Java treats white space characters as data, generating a Text node each time it finds whitespace characters.

Listing 31.5 Display of XML File Sample 1 using Java

```
Enter a file name (with extension): DigitalLibrary_1.xml
XML DOM node structure.
Each indentation is another level in the tree.
Displaying values for node type, name and value inside [].
type[Element],name[digital_library],value[]
  type[Text],name[#text],value[]
  type[Element],name[book],value[]
    type[Text],name[#text],value[]
    type[Element],name[isbn],value[]
      type[Text],name[#text],value[0441172717]
    type[Text],name[#text],value[]
    type[Element],name[title],value[]
      type[Text],name[#text],value[Dune]
    type[Text],name[#text],value[]
    type[Element],name[publisher],value[]
      type[Text],name[#text],value[Chilton Books]
    type[Text],name[#text],value[]
    type[Element],name[yearPublished],value[]
      type[Text],name[#text],value[1965]
    type[Text],name[#text],value[]
    type[Element],name[author],value[]
      type[Text],name[#text],value[Frank Herbert]
    type[Text],name[#text],value[]
  type[Text],name[#text],value[]
```

Listing 31.6 shows the display of the first Book instance from the second sample XML file shown in Listing 31.2. Since this XML file contains no whitespace characters, the display of the DOM shows no *extra* Text nodes.

Listing 31.6 Display of XML File Sample 2 using Java

```
Enter a file name (with extension): DigitalLibrary_2.xml
XML DOM node structure.
Each indentation is another level in the tree.
Displaying values for node type, name and value inside [].
type[Element],name[digital_library],value[]
  type[Element],name[book],value[]
    type[Element],name[isbn],value[]
      type[Text],name[#text],value[0441172717]
    type[Element],name[title],value[]
      type[Text],name[#text],value[Dune]
    type[Element],name[publisher],value[]
      type[Text],name[#text],value[Chilton Books]
    type[Element],name[yearPublished],value[]
      type[Text],name[#text],value[1965]
    type[Element],name[author],value[]
      type[Text],name[#text],value[Frank Herbert]
```

The entire Java program to display an XML file is found in displayDOM.java.

31.2.4.3 Update a DOM

A DOM tree structure can be updated by adding nodes at specific locations. The sample code in Listing 31.7 shows two methods. The changeDOM method calls getAuthorName to request the user to enter a new author name, calls addAuthorToFirstBook to add the entered name as a new author for the first book instance, and then saves the updated DOM back to an XML text file. We'll discuss saving the DOM to an XML file in the next section.

The addAuthorToFirstBook method gets a list of all book nodes in the DOM, saves the location of the first book node, creates a new Text node containing the user-supplied author name, creates a new Element node for an author tag, makes the new Text node a child of the new Element node, and then makes the new Element node a child of the first book node. Once this function has completed, the DOM structure has been modified by adding two additional nodes to the tree structure.

Listing 31.7 Java changeDom methods

```
private void changeDOM()
{
    String authorName = getAuthorName();
    addAuthorToFirstBook(authorName);
    saveDOMtoXML();
}

private void addAuthorToFirstBook(String authorName)
{
    //Get location of the first book node.
    NodeList nodes = dom.getElementsByTagName("book");
    if (nodes != null && nodes.getLength() > 0)
    {
        //The XML file has at least one book instance.
        Element firstBookNode = (Element)nodes.item(0);
        //Create text node to store the new author name.
        Text newText = dom.createTextNode(authorName);
        //Create element node for the new author.
        Element newElement = dom.createElement("author");
        //Add text node as child of the author element
        newElement.appendChild(newText);
        //Add author element to first book instance
        firstBookNode.appendChild(newElement);
    }
}
```

31.2.4.4 Save DOM to an XML File

Once a DOM structure has been updated, the updated DOM must be written to an XML file to persistently store the updated DOM. The code in Listing 31.8 shows the method to save a DOM to an XML file. This method creates an *output* file name, creates a Transformer object, creates a DOMSource object, creates a StreamResult object, and then uses the Transformer object to transform the DOMSource object (i.e., the DOM) into a StreamResult object (i.e., an output text file).

Listing 31.8 Java saveDOMtoXML method

```
private void saveDOMtoXML()
{
    try
    {
        String outputFileName =
            xmlFileName.substring(0,xmlFileName.length()-4) +
            "_OUTPUT" + xmlFileName.substring(xmlFileName.length()-4);
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer();
        DOMSource source = new DOMSource(dom);
        StreamResult result = new StreamResult(
            new File(outputFileName));
        transformer.transform(source, result);
        System.out.println("Updated DOM written to " +
            outputFileName);
    }
    catch (Exception ex)
    {
        System.out.println(ex);
    }
}
```

31.2.5 Python Examples

This section will introduce the use of Python to parse an XML file, to create a DOM, display the contents of a DOM, update a DOM, and to save a DOM back to an XML file.

31.2.5.1 Parse an XML File

The xml.dom.minidom module contains the functions to parse an XML file to create a DOM. In Listing 31.9, the xmlFileName parameter is the name of the XML file, including the file extension. When the xml.dom.minicom parse function is successful, a Document object is returned. Any exception will cause None to be returned by the createDOM method.

Listing 31.9 Python createDom function

```
def createDOM(xmlFileName):
    try:
        dom = xml.dom.minidom.parse(xmlFileName)
    except Exception as ex:
        print(ex)
        dom = None
    return dom
```

31.2.5.2 Display a DOM

Since a DOM is a tree structure, we can display the entire DOM by moving through the tree structure using a recursive traversal algorithm. Our traversal will display information for the current node (wherever this is within the tree). It then recursively traverses the sub-tree for each child of the current node, moving through the child nodes from left-to-right.

Listing 31.10 shows the displayDOM and displayDOMhelper functions. The displayDOM function used the Document object to obtain the root node for the DOM. It then calls the displayDOMhelper function, which implements the recursive traversal, displaying each node as it is reached. The indentLevel parameter for the recursive function keeps track of how much to indent each node as it's displayed to visually show the tree structure.

Listing 31.10 Python displayDom functions

```
def displayDOM(dom):
    print("XML DOM node structure.", \
        "Each indentation another level in tree.")
    docElement = dom.documentElement
    displayDOMhelper(docElement, 0)

def displayDOMhelper(element, indentLevel):
    #display information for this element
    displayElement(element, indentLevel);

    #display information for each child of this element
    if element.hasChildNodes():
        displayDOMhelper(element.firstChild, indentLevel + 1)

    #display information for each sibling of this element
    nextSibling = element.nextSibling
    if (nextSibling != None):
        displayDOMhelper(nextSibling, indentLevel)
```

Listing 31.11 shows the display of the first Book instance from the first sample XML file shown in Listing 31.1. One of the interesting aspects associated with this DOM is it contains *extra* text nodes containing no data value. These are shown in Listing 31.11 as "type<Text>,name<#text>,value<>". These nodes exist because of the whitespace characters (e.g., newline and tab characters) in the sample XML file shown in Listing 31.1. The XML parser in Python treats whitespace characters as data, generating a Text node each time it finds whitespace characters.

Listing 31.11 Display of XML File Sample 1 using Python

```
Enter a file name (with extension): DigitalLibrary_1.xml
XML DOM node structure. Each indentation another level in tree.
type<Element>,name<digital_library>,value<>
  type<Text>,name<#text>,value<>
  type<Element>,name<book>,value<>
    type<Text>,name<#text>,value<>
    type<Element>,name<isbn>,value<>
```

```
      type<Text>,name<#text>,value<0441172717>
    type<Text>,name<#text>,value<>
    type<Element>,name<title>,value<>
      type<Text>,name<#text>,value<Dune>
    type<Text>,name<#text>,value<>
    type<Element>,name<publisher>,value<>
      type<Text>,name<#text>,value<Chilton Books>
    type<Text>,name<#text>,value<>
    type<Element>,name<yearPublished>,value<>
      type<Text>,name<#text>,value<1965>
    type<Text>,name<#text>,value<>
    type<Element>,name<author>,value<>
      type<Text>,name<#text>,value<Frank Herbert>
    type<Text>,name<#text>,value<>
  type<Text>,name<#text>,value<>
```

Listing 31.12 shows the display of the first Book instance from the second sample XML file shown in Listing 31.2. Since this XML file contains no whitespace characters, the display of the DOM shows no *extra* Text nodes.

Listing 31.12 Display of XML File Sample 2 using Python

```
Enter a file name (with extension): DigitalLibrary_1.xml
XML DOM node structure. Each indentation another level in tree.
type<Element>,name<digital_library>,value<>
  type<Element>,name<book>,value<>
    type<Element>,name<isbn>,value<>
      type<Text>,name<#text>,value<0441172717>
    type<Element>,name<title>,value<>
      type<Text>,name<#text>,value<Dune>
    type<Element>,name<publisher>,value<>
      type<Text>,name<#text>,value<Chilton Books>
    type<Element>,name<yearPublished>,value<>
      type<Text>,name<#text>,value<1965>
    type<Element>,name<author>,value<>
      type<Text>,name<#text>,value<Frank Herbert>
```

The entire Python program to display an XML file is found in displayDOM.py.

31.2.5.3 Update a DOM

A DOM tree structure can be updated by adding nodes at specific locations. The sample code in Listing 31.13 shows two functions. The changeDOM function calls getAuthorName to request the user to enter a new author name, calls addAuthorToFirstBook to add the entered name as a new author for the first book instance, and then saves the updated DOM back to an XML text file. We'll discuss saving the DOM to an XML file in the next section.

The addAuthorToFirstBook function gets a list of all book nodes in the DOM, saves the location of the first book node, creates a new Text node containing the user-supplied author name, creates a new Element node for an author tag, makes the

new Text node a child of the new Element node, and then makes the new Element node a child of the first book node. Once this function has completed, the DOM structure has been modified by adding two additional nodes to the tree structure.

Listing 31.13 Python changeDom functions

```
def changeDOM():
    authorName = getAuthorName()
    addAuthorToFirstBook(authorName)
    saveDOMtoXML()

def addAuthorToFirstBook(authorName):
    global dom
    #Get location of the first book node.
    nodes = dom.getElementsByTagName("book")
    if nodes != None and nodes.length > 0:
        #The XML file has at least one book instance.
        firstBookNode = nodes.item(0)
        #Create text node to store the new author name.
        newText = dom.createTextNode(authorName)
        #Create element node for the new author.
        newElement = dom.createElement("author")
        #Add text node as child of the author element
        newElement.appendChild(newText)
        #Add author element to first book instance
        firstBookNode.appendChild(newElement)
```

31.2.5.4 Save DOM to an XML File

Once a DOM structure has been updated, the updated DOM must be written to an XML file to persistently store the updated DOM. The code in Listing 31.14 shows the function that will save a DOM to an XML file. This function creates an *output* file name, creates an output file handle, and then uses the writexml method of the Document class to write the contents of the DOM to the output file handle (i.e., the output text file).

Listing 31.14 Python saveDOMtoXML function

```
def saveDOMtoXML():
    global xmlFileName
    global dom
    outputFileName = xmlFileName[0:len(xmlFileName)-4] +
            "_OUTPUT" + xmlFileName[len(xmlFileName)-4:]
    fileHandle = open(outputFileName,"wt")
    dom.writexml(fileHandle)
    fileHandle.close()
    print("Updated DOM written to", outputFileName)
```

Table 31.2 Rdb table example

StudentID	First	Middle	Last	Street1	City	State	Zip	Phone
12233345	Sue	B.	Doe	123 Doe St	Rome	NY	13440	3155559999
11933367	Joe		Doe	951 K St	Utica	NY	13502	3155551111

31.3 Relational Database Concepts

This section introduces Relational Database (Rdb) as a way to store data persistently.

31.3.1 What is a Relational Database?

A relational database (Rdb) is one of the most widely used persistent data storage formats currently in use. Conceptually, a Rdb is represented as a collection of tables, where each table is a physical manifestation of a logical data entity found on a LDM. A table is simply a two-dimensional matrix of rows and columns. Each row represents one record or data instance while each column represents a logical data attribute or a data field. For example, a table storing information about a student might have columns for student id, name (first, middle, last), address (street, city, state, zip), and phone number. Each of these columns represents a specific piece of information about a student, while the entire set of columns represents a collection of logically related information about a student. This collection of information would exist within one row of a student table and represents a student instance or student record. Table 31.2 shows two student instances (i..e., records). Each instance has nine distinct data values (i.e., fields, attributes).

The concept of a relational table as a bunch of rows and columns is simple to understand. We frequently see information presented to us in this manner. For example, voting results for an election, nutrition information for packaged food, and statistics for a sporting event. Another benefit of a relational database is that it is based on a few simple mathematical principles. Specifically, relational databases are based on relational algebra, which is a combination of predicate logic and set theory.

31.3.1.1 Predicate Logic

Predicate logic uses a few mathematical symbols to express quantifiers, logical operators, and variables. The quantifiers, logical operators, and variables are then used to define predicates.[1] The most common quantifiers are ∃ (there exists) and ∀ (for all). The most common logical operators are ∧ (conjunction i.e, and), ∨ (disjunction,

[1] A simple definition of a predicate is a statement that may be true or false depending on the logic expressed and the value of the variables used.

i.e., or), $\Rightarrow$ (implication), and $\neg$ (negation i.e., not). Variables are usually denoted using lowercase letters. A few sample predicates are listed below.

- $\exists p \mid \text{knows}(p, \text{Java})$
  There exists a programmer that knows Java.
- $\forall p, \text{programmer}(p) \Rightarrow \text{knows}(p, language)$
  For all persons, if person is a programmer, then this person knows a programming language.
- $\forall p, \text{programmer}(p) \wedge (\text{knows}(p, \text{Java}) \vee \text{knows}(p, \text{Python})) \Rightarrow \text{benefit}(p, \text{learn (book)})$
  For all persons, if person is a programmer that knows Java or Python, then this person will benefit from reading this book.

31.3.1.2 Set Theory

Set theory uses a few mathematical symbols to express sets and set operators. A set is a collection of unique objects (or values). The contents of a set is often denoted using a list delimited by braces, for example, {1, 2, 3}. An empty set is denoted with the symbol $\emptyset$. The common set operators are $\cup$ (union), $\cap$ (intersection), $-$ (difference), $\times$ (Cartesian product), and power(S) (power set). A few examples are listed below.

- $\{1, 2, 3\} \cup \{1, 2, 4\}$ is $\{1, 2, 3, 4\}$
  If A and B are sets then A $\cup$ B is the set of all elements that are in A or B.
- $\{1, 2, 3\} \cap \{1, 2, 4\}$ is $\{1, 2\}$
  If A and B are sets then A $\cap$ B is the set of all elements that are in both A and B.
- $\{1, 2, 3\} - \{1, 2, 4\}$ is $\{3\}$
  If A and B are sets then A $-$ B is the set of all elements in A that are not in B.
- $\{1, 2, 3\} \times \{1, 2, 4\}$ is $\{(1, 1), (1, 2), (1, 4), (2, 1), (2, 2), (2, 4), (3, 1), (3, 2), (3, 4)\}$
  If A and B are sets then A $\times$ B is the set of all ordered pairs (a, b) where a is an element of A and b is an element of B.
- power($\{1, 2, 3\}$) is $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$
  If A is a set then power(A) is the set of all subsets of A.

Since we are using set theory to help explain how relational databases work, we use the term *relation* to mean a *set*.

31.3.2 Relational Database Model (aka: Physical Data Model)

Designing a relational database starts with creating an ERD (entity relationship diagram) and LDM (logical data model) to document the logical relationships between the data elements. This section talks about translating a LDM into a relational database model (i.e., a physical data model).

The entities and attributes in a LDM will be translated into tables and columns (i.e., fields) in a relational database model. The primary key attributes for each entity

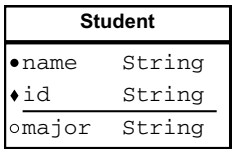

Fig. 31.2 Relational database model: one table

will become the primary key fields in the table. The fields defined in each table will include a data type. Relationships in a LDM describe the cardinality of the relationship, which would be one of the following:

1:1 A one-to-one relationship between two data entities. Each instance of entity A is related to one instance of entity B and vice versa. For example, a relationship between person and student is 1:1. A person may be a student and a student is a person.

1:M A one-to-many relationship between two data entities. Each instance of entity A is related to many instances of entity B but each instance of entity B is related to one instance of entity A. For example, a relationship between student and term gpa is 1:M. A student may have many term-gpas but a term-gpa is for one student.

M:N A many-to-many relationship between two data entities. Each instance of entity A is related to many instances of entity B and each instance of entity B is related to many instances of entity A. For example, a relationship between student and course is M:N. A student may take many courses and a course may be taken by many students.

A simple relational database model is shown in Fig. 31.2. This shows a database with a single table called Student with three fields. All three fields contain character string data and the id field is the primary key. The small letter "oh" in front of the major field indicates this field is optional; a major value may not be provided (i.e., this field may be empty or what relational databases call a null value). The other two fields must contain a non-null value.

A more interesting example is to translate the logical data model for the digital library described in Chap. 30 into a relational database model. Figure 31.3 shows the Version 3 logical data model for the digital library. Figure 31.4 is the corresponding relational database model. Note the following about this translation from a LDM to a PDM:

- Each logical data entity has been translated to a relational table.
- Each logical data attribute has been translated into a field. Each field includes its data type.
- Each primary key attribute has been translated into a primary key field. For example, see isbn in Book and title and yearPublished in Movie.

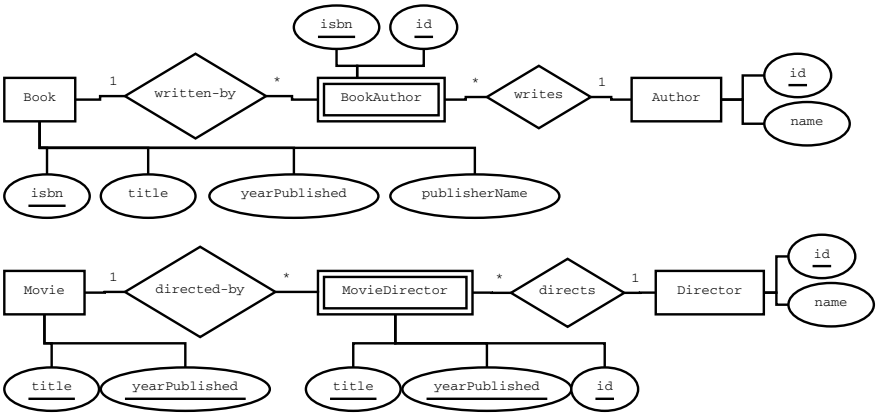

Fig. 31.3 Digital library LDM Version 3 (from Chap. 30)

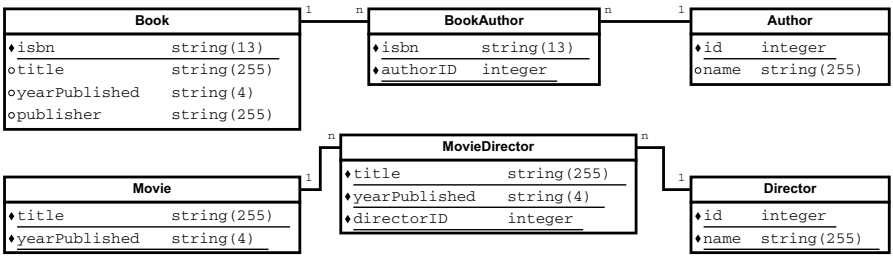

Fig. 31.4 Digital library relational database model Version 3

31.3.2.1 Translating LDM to PDM

There are two issues when translating a logical data model into a relational database model. The logical data model in Fig. 31.5 will be used to illustrate these two issues, while the relational database model in Fig. 31.6 shows the results of resolving the two issues described below.

1. How does a weak entity in a logical data model get translated into a relational database model?
   The LDM in Fig. 31.5 has a weak entity called TermGPA with a weak key of the term. When this weak entity gets translated into a table in the relational database model, the primary key from the related strong entity must be included in the table. Figure 31.6 shows table TermGPA with three fields while the corresponding entity only has two attributes. The studentID will have the same value as the corresponding id field from the related Student table. This is combined with the term field to form the primary key for the TermGPA. That is, combining the values from studentID and term will uniquely identify each row in the TermGPA table. The studentID field in the TermGPA table is also called a *foreign key* field. This term applies to any field in a table where this field is a primary key in another

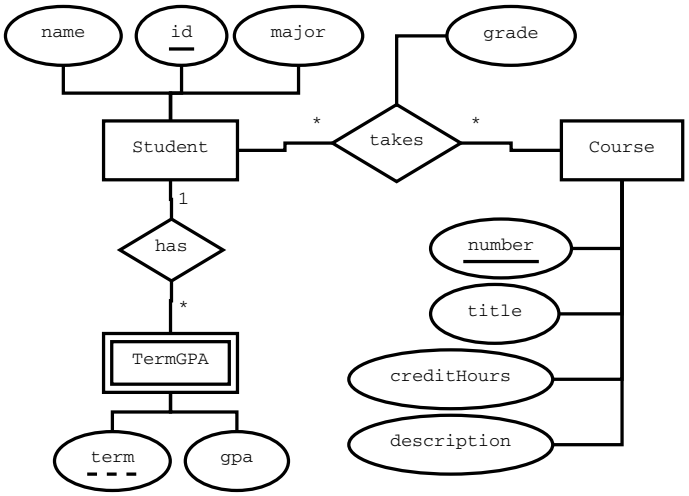

Fig. 31.5 Student-GPA-Course LDM

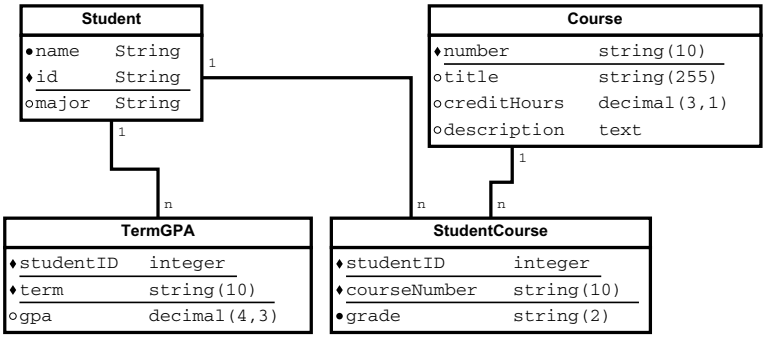

Fig. 31.6 Student-GPA-Course PDM

table. Why do we care about foreign keys? Remember the normalization process described in Chap. 30? One of the stated goals of the normalization process is to eliminate redundant data in our logical data models. It looks like the concept of a foreign key, as applied to relational database models, is now adding redundant data! While technically this is correct, relational databases have ways to automatically avoid issues with redundant data. Specifically, foreign key constraints within a relational database product (e.g., MySQL, SQL Server, Oracle) will automatically enforce update and delete scenarios. For example, if I changed an id value in a Student instance, this change would automatically be applied to all studentID foreign keys in the TermGPA table that match the original value. Similarly, if I decide to delete a Student instance from the table, all instances in the TermGPA table which match the id value being deleted are automatically deleted.

2. How does a many-to-many relationship in a logical data model get translated into a relational database model?
   The LDM in Fig. 31.5 has a many-to-many relationship: A Student takes many Courses; A Course is taken by many Students. Most of the time, translating this type of relationship in a relational database model results in replacing the many-to-many relationship with a table, which produces two one-to-many relationships in the relational database model. Figure 31.6 shows table StudentCourse, which does not exist as an entity in Fig. 31.5. Note the following regarding this translation of a logical many-to-many relationship to a relational database model.

   - The Student and Course entities are translated into tables.
   - The "takes" relationship is translated into a table called StudentCourse. It is fairly common practice to name this table by combining the entity names from the LDM.
   - The many-to-many logical relationship is now two distinct one-to-many physical relationships: a Student has many StudentCourse instances and a Course has many StudentCourse instances.
   - The StudentCourse table has as its primary key the primary key fields from the two tables it is related to. Thus, the primary key fields in the StudentCourse table are also foreign key fields.
   - The StudentCourse table contains a grade field which corresponds to the LDM showing grade as an attribute of the many-to-many "takes" relationship.

   The result of this translation is the following: (1) A Student can take a Course multiple times by taking the same course in different terms. Each time the Student shall earn a distinct grade; and (2) A Course offered in a term may be taken by multiple Students. Each Student shall earn a distinct grade.

31.3.3 Structured Query Language (SQL)

The structured query language (SQL—pronounced sequel) is a declarative programming language[2] supported by all relational database products.[3] Learning SQL allows you to do two things: create and modify relational database structures; and create, read, update, and delete data in a relational database. An excellent tutorial on SQL is found on the www.w3schools.com web site [3].

[2] A declarative programming language contains statements to indicate what needs to be done. However, the language does not allow the programmer to express how to achieve the result. In the case of SQL, the database software, often called the database engine, determines how to accomplish what needs to be done.
[3] SQL is both an ANSI and ISO standard.

31.3.3.1 Data Definition Language (DDL)

The data definition language (DDL) is the part of SQL used to create and modify relational database structures. DDL capabilities include the following list, which will be described below. Other DDL statements exist to create and modify other aspects of a relational database. These additional DDL features are not discussed in this chapter.

- Create and modify a database.
- Create and modify a table in a database.
- Create and modify an index in a database.

The script shown in Listing 31.15 illustrates some of the common DDL statements. This script is written for MySQL version 5.7 and would create the Digital Library relational database shown in Fig. 31.4. In The MySQL 5.7 Reference Manual [4], see *Chap.* 11 *Data Types* and *Chap.* 13 *SQL Statement Syntax* for details on these DDL statements.

Listing 31.15 DDL Digital Library

```
drop database if exists DigitalLibrary;
create database DigitalLibrary;
use DigitalLibrary;

create table Book
    (isbn          CHAR(13) PRIMARY KEY,
    title          VARCHAR(255),
    publisher      VARCHAR(255),
    yearPublished  CHAR(4));

create table Author
    (id      INTEGER AUTO_INCREMENT PRIMARY KEY,
    name     VARCHAR(255));

create table BookAuthor
    (isbn        CHAR(13),
    authorID     INTEGER,
    PRIMARY KEY (isbn, authorID),
    FOREIGN KEY (isbn)
        REFERENCES Book(isbn)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (authorID)
        REFERENCES Author(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE);

create table Movie
    (title         VARCHAR(255),
    yearPublished  CHAR(4),
    PRIMARY KEY (title, yearPublished));
```

```
create table Director
    (id       INTEGER AUTO_INCREMENT PRIMARY KEY,
    name      VARCHAR(255));

create table MovieDirector
    (title        VARCHAR(255),
    yearPublished    CHAR(4),
    directorID       INTEGER,
    PRIMARY KEY (title, yearPublished, directorID),
    FOREIGN KEY (title, yearPublished)
        REFERENCES Movie(title, yearPublished)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (directorID)
        REFERENCES Director(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE);

/* A few sample indexes */
create index BookIndex_1
    ON Book(title);
create index BookIndex_2
    ON Book(publisher);
create index BookIndex_3
    ON Book(yearPublished);
```

Note the yearPublished fields are defined as CHAR(4) instead of INTEGER. While a year value is an integer, it is unlikely we would need to do arithmetic on a yearPublished value. Thus, we use a string data type for this field.

A few comments regarding the SQL script in Listing 31.15.

- The first two statements delete and then create a relational database named DigitalLibrary.
- The third statement tells the database server that the remaining SQL statements in the script will be using the DigitalLibrary database.
- Most relational databases have both a CHAR and VARCHAR data type. A CHAR data type is a fixed length string. When the data value is less than the fixed length, spaces are added to reach the defined length of the field. In contrast, a VARCHAR data type will not pad the value with spaces.
- The BookAuthor and MovieDirector CREATE TABLE statements show how a foreign key is defined. The constraints ON DELETE and ON UPDATE indicate what should happen when a primary key value is deleted or modified in the referenced table.
- Each PRIMARY KEY automatically generates a database index, allowing good performance when accessing the data using the primary key.

- The sample CREATE INDEX statements show how to create additional indexes for a table. Additional indexes are created for a table when its data is accessed using something other than the primary key fields.

31.3.3.2 Data Manipulation Language (DML)

The data manipulation language (DML) portion of SQL is used to implement CRUD transactions that operate on the data stored in the database. CRUD stands for:

- Create: use SQL INSERT statements to add data to a database.
- Read: use SQL SELECT statements to query a database.
- Update: use SQL UPDATE statements to modify data already stored in a database.
- Delete: use SQL DELETE statements to remove data from a database.

The scripts in this section illustrate the insert, select, update, and delete DML statements. These scripts are written for MySQL version 5.7 and use the Digital Library relational database shown in Fig. 31.4. In the MySQL 5.7 Reference Manual [4], see *Chap.* 13 *SQL Statement Syntax* for details on these DML statements. For readability purposes only, the SQL keywords are in all uppercase letters. With the exception of data values, SQL statements may be written in all uppercase, all lowercase, or mixed case; data values are typically case-sensitive.

Listing 31.16 shows insert statements to add data to the database tables. Assuming the DDL statements in Listing 31.15 are executed just before using these insert statements, the id values inserted into the Author table are 1, 2, 3, and 4. If we were to run the insert statements a second time, the id values would be 5 through 8. When you want to reset the AUTO_INCREMENT so the first insert gets an id value of 1, the best approach is to use DROP TABLE followed by CREATE TABLE. In the case of the Digital Library tables, if you want to reset the Author table's AUTO_INCREMENT, you would need to drop both the Author and BookAuthor tables, since the BookAuthor table has a FOREIGN KEY CONSTRAINT on the Author id field.

When large amounts of data need to be added to a database, using a feature that takes data from a file and loads it into a database table is a more efficient method. In the case of MySQL, you may use the *mysqlimport* client program[4] or the *LOAD DATA* SQL statement.[5]

Listing 31.16 DML Example 1: Create Data using INSERT

```
/* Insert a book with one author */
INSERT INTO Book (isbn, title, publisher, yearPublished)
VALUES ("0441172717","Dune","Chilton Books","1965");

INSERT INTO Author (name) VALUES ("Frank Herbert");
```

[4] See Sect. 4.5 MySQL Client Programs in [4].
[5] See Sect. 13.2 Data Manipulation Statements in [4].

```
INSERT INTO BookAuthor (isbn, authorID) VALUES ("0441172717",1);

/* Insert a book with same author */
INSERT INTO Book (isbn, title, publisher, yearPublished)
VALUES ("0441172695","Dune Messiah","Putnam Publishing","1969");

INSERT INTO BookAuthor (isbn, authorID) VALUES ("0441172695",1);

/* Insert a book with many authors */
INSERT INTO Book (isbn, title, publisher, yearPublished)
VALUES ("0137135599","Blown to Bits","Addison Wesley","2008");

INSERT INTO Author (name) VALUES ("Hal Abelson");
INSERT INTO Author (name) VALUES ("Ken Ledeen");
INSERT INTO Author (name) VALUES ("Harry Lewis");

INSERT INTO BookAuthor (isbn, authorID) VALUES ("0137135599",2);
INSERT INTO BookAuthor (isbn, authorID) VALUES ("0137135599",3);
INSERT INTO BookAuthor (isbn, authorID) VALUES ("0137135599",4);

/* Insert a movie */
INSERT INTO Movie (title, yearPublished)
VALUES ("The Shawshank Redemption","1994");

INSERT INTO Director (name) VALUES ("Frank Darabont");

INSERT INTO MovieDirector (title, yearPublished, directorID)
VALUES ("The Shawshank Redemption","1994",1);
```

Based on the INSERT statements in Listing 31.16, the *SELECT* statement in Listing 31.17 returns the data shown in Table 31.3. The asterisk ("*") in the SELECT statement requests data for all columns in the table.

Listing 31.17 DML Example 2: Read Data using SELECT Example 1

```
/* Get all of the books in the database */
SELECT * FROM Book;
```

Based on the INSERT statements in Listing 31.16, the *SELECT* statement in Listing 31.18 returns the data shown in Table 31.4. The WHERE clause returns only those Author's whose name contains an uppercase "H". The wildcard character percent sign ("%") matches zero or more of any character. Using this as part of a LIKE operator returns any Author name that contains an uppercase "H".

Table 31.3 Rdb SELECT Example 1

isbn	title	yearPublished	publisher
"0441172717"	"Dune"	"Chilton Books"	"1965"
"0441172695"	"Dune Messiah"	"Putnam Publishing"	"1969"
"0137135599"	"Blown to Bits"	"Addison Wesley"	"2008"

Table 31.4 Rdb SELECT Example 2

Id	Name
1	"Frank Herbert"
2	"Hal Abelson"
4	"Harry Lewis"

Table 31.5 Rdb SELECT Example 3

title	yearPublished
"Dune"	"1965"
"Dune Messiah"	"1969"
"Blown to Bits"	"2008"
"Blown to Bits"	"2008"

Listing 31.18 DML Example 3: Read Data using SELECT Example 2

```
/* Get all of the authors that have "H" in their name */
SELECT id, name FROM Author
WHERE name LIKE "%H%";
```

Based on the INSERT statements in Listing 31.16, the *SELECT* statement in Listing 31.19 returns the data shown in Table 31.5. The WHERE clause returns title and yearPublished data only for those books with an Author name containing an uppercase "H". Specifically, the first clause (i.e., WHERE name LIKE "%H%") returns Author row instances where the name contains at least one uppercase "H". The second clause (i.e., AND Author.id = BookAuthor.authorID) returns BookAuthor row instances where the author id values match. Finally, the third clause (i.e., AND Book.isbn = BookAuthor.isbn) returns Book row instances where the isbn values match.

Listing 31.19 DML Example 4: Read Data using SELECT Example 3

```
/* Get all books whose author has "H" in their name */
SELECT title, yearPublished
FROM Book, BookAuthor, Author
WHERE name LIKE "%H%"
  AND Author.id = BookAuthor.authorID
  AND Book.isbn = BookAuthor.isbn;
```

Based on the INSERT statements in Listing 31.16, the *UPDATE* statement in Listing 31.20 changes the name of Author "Hal Abelson" to "Harold Abelson".

Listing 31.20 DML Example 5: Update Data using UPDATE

```
/* Change first name of author Abelson */
UPDATE Author SET name = "Harold Abelson" WHERE id = 2;
```

Based on the INSERT statements in Listing 31.16, the *DELETE* statement in Listing 31.21 removes the row in the Author table containing the name "Harry Lewis". Since a FOREIGN KEY constraint has been defined between Author.id and BookAuthor.authorID, this delete statement will also remove any rows in the

BookAuthor table containing Harry Lewis's author.id value. One important note: without a WHERE clause, a DELETE statement will remove all rows from a table.

Listing 31.21 DML Example 6: Delete Data using DELETE

```
/* Delete author Harry Lewis */
DELETE FROM Author WHERE id = 4;
```

31.3.4 Embedded SQL

SQL statements may be given to a database server for execution in two fundamental ways.

Using client programs: These client programs may be provided by the database server or by a third party. In either case, SQL statements are entered via a command-line or graphical user interface. The MySQL [4] database provides a number of client programs, some allowing DML statements and database configuration commands to be entered (e.g., mysqladmin) and some allowing DDL and DML statements to be entered (e.g., mysql).

Embedding SQL in code: Software developers can embed SQL statements into their code. The details of how this is done are dependent on the specific programming language and the type of database server being used.

Embedding SQL into code is conceptually similar to doing file input/output. The details associated with embedding SQL into Java or Python code are described in the next two sections.

31.3.5 Java Examples

This section will introduce the use of Java to connect to and use a relational database. We'll be using a MySQL database server in these examples.

31.3.5.1 Connect to a Rdb

In order to use a relational database in your code, first, you need to connect to a database server and a database on that server. For Java, we are using the MySQL Connector/J described in *Chap.* 23 *Connectors and API* in [4]. Once the Connector/J is installed, you need to add the jar file (mysql-connector-java-*version*.jar) to the CLASSPATH environment variable.

Listing 31.22 shows the Java code to connect to a MySQL database server on the local host and to the DigitalLibrary database on this local server. The user and password values were created specifically for this example. We give the DriverManager the name of the MySQL driver to use and then we get a connection to the database

on the local machine. The dbConnection variable in this code listing is an instance variable in the class containing this code.

Listing 31.22 Java Example: Connect to MySQL Database

```
try
{
    //Register driver name with the DriverManager.
    //Class.forName("com.mysql.jdbc.Driver").newInstance(); //old
    Class.forName("com.mysql.cj.jdbc.Driver").newInstance();//new
    //Connect to a database
    dbConnection = DriverManager.getConnection(
            "jdbc:mysql://localhost/DigitalLibrary?" +
            "user=dave&password=dave");
    userInput = new Scanner(System.in);
}
catch (SQLException ex)
{
    dbConnection = null;
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
catch (Exception ex)
{
    dbConnection = null;
    System.out.println(ex);
}
```

31.3.5.2 Query a Rdb

The example in Listings 31.23, 31.24 and 31.25 display Book instances that contain a user-supplied search string. Listing 31.23 shows two different SELECT statements. The first SELECT statement looks for Book instances where the publisher's name contains the user-supplied string value, while the second statement looks for Book instances where the title contains the user-supplied string value. The LIKE BINARY phrase ensures Book instances matching the case of the user-supplied string value are returned, while the question mark ("?") is a placeholder representing where the user-supplied data will be put within the SELECT statement.

Listing 31.23 Java Example: Query a MySQL Database Part 1

```
private final String SELECT_BOOKS_PUB =
     "SELECT isbn, title, publisher, yearPublished FROM Book " +
     "WHERE publisher LIKE BINARY ?;";
private final String SELECT_BOOKS_TITLE =
     "SELECT isbn, title, publisher, yearPublished FROM Book " +
     "WHERE title LIKE BINARY ?;";
```

Listing 31.24 shows the logic associated with the user indicating whether they want to list books matching a publisher search string or a title search string. The userData value represents the search value entered by the user.

Listing 31.24 Java Example: Query a MySQL Database Part 2

```
if (menuChoice.equals(MENU_PUB))
{
  System.out.println("The following books contain<" + userData +
      "> in publisher field:");
  displayBooks(SELECT_BOOKS_PUB, userData);
}
else if (menuChoice.equals(MENU_TITLE))
{
  System.out.println("The following books contain<" + userData +
      "> in title field:");
  displayBooks(SELECT_BOOKS_TITLE, userData);
}
```

Listing 31.25 shows the displayBooks method. A PreparedStatement object is created using the SELECT statement represented by the first formal parameter. The PreparedStatement object is then used to set the value of the placeholder (i.e., question mark) found in the SELECT statement. The statement is then executed, which produces a ResultSet. A ResultSet object contains all of the Book instances returned based on the just executed SELECT statement. The do-while will iterate as long as there are more Book instances in the ResultSet. The method ends by closing the ResultSet and PreparedStatement.

Listing 31.25 Java Example: Query a MySQL Database Part 3

```
private void displayBooks(String selectStmt, String userData)
{
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try
    {
        stmt = dbConnection.prepareStatement(selectStmt);
        stmt.setString(1, PERCENT + userData + PERCENT);
        rs = stmt.executeQuery();
        if (rs.next())
            do
            {
                System.out.println("  isbn:" + rs.getString(1) +
                        "\ttitle:" + rs.getString(2) +
                        "\tpublisher:" + rs.getString(3) +
                        "\tyear published:" + rs.getString(4));
            } while (rs.next());
        else
          System.out.println("  No books satisfy search criteria.");
    }
    catch (SQLException ex)
    {
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
```

```
    }
    closeResultSet(rs);
    closeStatement(stmt);
}
```

31.3.5.3 Update a Rdb

The example in Listings 31.26 and 31.27 adds a new author to the first book instance in the database. The insertAuthor method adds the user-supplied name to the Author table. Similar to the SELECT statement example, a placeholder (i.e., question mark) is used to provide the user-supplied author name to the insert statement. The result from executing the insert statement should be an integer 1, indicating that 1 row (aka data instance) was added to the Author table. The insertAuthor method returns the Author id value representing the author's name just added to the table.

Listing 31.26 Java Example: Insert into a MySQL Database Part 1

```
private int insertAuthor(String authorName)
{
    final String INSERT_AUTHOR =
            "INSERT INTO Author (name) VALUES (?)";
    PreparedStatement stmt = null;
    int authorID = 0;
    try
    {
        stmt = dbConnection.prepareStatement(INSERT_AUTHOR);
        stmt.setString(1, authorName);
        int count = stmt.executeUpdate();
        if (count == 1)
        {
            authorID = getMaxAuthorID();
            System.out.println("Added author<" + authorName +
                    "> with id<" +
                    Integer.toString(authorID) + ">.");
        }
        else
          System.out.println("Logic error occurred while inserting" +
                    " new author");
    }
    catch (SQLException ex)
    {
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }
    closeStatement(stmt);
    return authorID;
}
```

The insertAuthorForBook method in Listing 31.27 uses the Author id value returned by the insertAuthor method along with the isbn value for the first book in the database. These two data values are used to add a row to the BookAuthor

table. Note the use of two placeholders in this INSERT statement. Again, the insert should result in one row being added to the BookAuthor table.

Listing 31.27 Java Example: Insert into a MySQL Database Part 2

```
private void insertAuthorForBook(int authorID, String isbn)
{
    final String INSERT_AUTHOR =
            "INSERT INTO BookAuthor (isbn, authorID) VALUES (?, ?)";
    PreparedStatement stmt = null;
    try
    {
        stmt = dbConnection.prepareStatement(INSERT_AUTHOR);
        stmt.setString(1, isbn);
        stmt.setInt(2, authorID);
        int count = stmt.executeUpdate();
        if (count == 1)
            System.out.println("Added author<" + authorID +
                    "> to ISBN<" + isbn + ">.");
        else
            System.out.println("Logic error occurred while inserting" +
                    " new author");
    }
    catch (SQLException ex)
    {
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }
    closeStatement(stmt);
}
```

31.3.6 Python Examples

This section will introduce the use of Python to connect to and use a relational database. We'll be using a MySQL database server in these examples.

31.3.6.1 Connect to a Rdb

In order to use a relational database in your code, first, you need to connect to a database server and a database on that server. For Python, we are using the MySQL Connector/Python described in *Chap.* 23 *Connectors and API* in [4]. The Python connector code is put into the Lib\site-packages folder found within a folder whose name is associated with the Python version you are using (e.g., \Python33).

Listing 31.28 shows the Python code to connect to a MySQL database server on the local host and to the DigitalLibrary database on this local server. The user and password values were created specifically for this example.

Listing 31.28 Python Example: Connect to MySQL Database

```
import mysql.connector
from mysql.connector import Error

def createMySQLconnection():
    global dbConnection
    dbConnection = None
    try:
        dbConnection = mysql.connector.connect(host='localhost',
            database='DigitalLibrary',
            user='dave',
            password='dave',
            raise_on_warnings=True)
        if dbConnection.is_connected():
            print("Connected to DigitalLibrary database")
    except Error as err:
        print(err)
```

31.3.6.2 Query a Rdb

The example in Listings 31.29, 31.30, and 31.31display Book instances that contain a user-supplied search string. Listing 31.29 shows two different SELECT statements. The first SELECT statement looks for Book instances where the publisher's name contains the user-supplied string value, while the second statement looks for Book instances where the title contains the user-supplied string value. The LIKE BINARY phrase ensures Book instances matching the case of the user-supplied string value are returned, while the "%s" is a placeholder representing where the user-supplied data will be put within the SELECT statement.

Listing 31.29 Python Example: Query a MySQL Database Part 1

```
SELECT_BOOKS_PUB = "SELECT isbn, title, publisher, " +
        "yearPublished FROM Book WHERE publisher LIKE BINARY %s"
SELECT_BOOKS_TITLE = "SELECT isbn, title, publisher, " +
        "yearPublished FROM Book WHERE title LIKE BINARY %s"
```

Listing 31.30 shows the logic associated with the user indicating whether they want to list books matching a publisher search string or a title search string. The userData value represents the search value entered by the user.

Listing 31.30 Python Example: Query a MySQL Database Part 2

```
if menuChoice == MENU_PUB:
    print("The following books contain <", userData,
            "> in publisher field:", sep="")
    displaySomeBooks(SELECT_BOOKS_PUB, userData)
```

```
elif menuChoice == MENU_TITLE:
    print("The following books contain <", userData,
          "> in title field:", sep="")
    displaySomeBooks(SELECT_BOOKS_TITLE, userData)
```

Listing 31.31 shows the displaySomeBooks function. A cursor object is created and then executed. The cursor's execute method is given the SELECT statement provided via the first parameter and the value of the placeholder (i.e., "%s") found in the SELECT statement. The placeholder values must be given to the execute method in a tuple. Executing the SELECT statement produces an iterator used to move through the tuples returned from the query. The function ends by closing the cursor.

Listing 31.31 Python Example: Query a MySQL Database Part 3

```
def displaySomeBooks(selectStmt, userData):
    global dbConnection
    try:
      bookCursor = dbConnection.cursor()
      bookCursor.execute(selectStmt,
           (PERCENT + userData + PERCENT,))
      for (isbn, title, publisher, yearPublished) in bookCursor:
         print('  isbn:', isbn, '\ttitle:', title, '\tpublisher:',
             publisher, '\tyearPublished:', yearPublished)
      bookCursor.close()
    except Error as err:
      print(err)
```

31.3.6.3 Update a Rdb

The example in Listings 31.32 and 31.33 adds a new author to the first book instance in the database. The insertAuthor function adds the user-supplied name to the Author table. Similar to the SELECT statement example, a placeholder (i.e., "%s") is used to provide the user-supplied author name to the insert statement. The result from executing the insert statement should be a row count of 1, indicating that 1 row (aka data instance) was added to the Author table. The insertAuthor method returns the Author id value representing the author's name just added to the table.

Listing 31.32 Python Example: Insert into a MySQL Database Part 1

```
def insertAuthor(authorName):
  global dbConnection
  INSERT_AUTHOR = 'INSERT INTO Author (name) VALUES (%s)'
    authorID = None
    try:
      authorCursor = dbConnection.cursor()
      authorCursor.execute(INSERT_AUTHOR, (authorName,))
      if authorCursor.rowcount == 1:
        authorID = authorCursor.lastrowid
        print("Added author <", authorName, "> with id <", authorID,
              ">.", sep="")
      else:
```

```
        print("Logic error occurred while inserting new author")
      authorCursor.close()
    except Error as err:
      print(err)
    return authorID
```

The insertAuthorForBook function in Listing 31.33 uses the Author id value returned by the insertAuthor function along with the isbn value for the first book in the database. These two data values are used to add a row to the BookAuthor table. Note the use of two placeholders in this INSERT statement. Again, the insert should result in one row being added to the BookAuthor table.

Listing 31.33 Python Example: Insert into a MySQL Database Part 2

```
def insertAuthorForBook(authorID, isbn):
    global dbConnection
    INSERT_AUTHOR = 'INSERT INTO BookAuthor (isbn, authorID) ' +
            'VALUES (%s, %s)'
    try:
      authorCursor = dbConnection.cursor()
      authorCursor.execute(INSERT_AUTHOR, (isbn, authorID))
      if authorCursor.rowcount == 1:
        print("Added author <", authorID, "> to ISBN <",
            isbn, ">.", sep="")
      else:
        print("Logic error occurred while inserting new author")
      authorCursor.close()
    except Error as err:
        print(err)
```

31.4 Post-conditions

The following should have been learned when completing this chapter.

- You understand the following regarding eXtensible Markup Language (XML):
  - XML files represent data using a tree structure of tags. An XML tag will contain a data value and/or other tags, is given a name delimited with < and > (e..g, <lastname>), and typically has a matching end tag (e.g., </lastname>).
  - A DOM (document object model) represents an XML file as a tree structure of nodes. You navigate to any node in a DOM by starting at the root of the tree.
  - Both Java and Python provide support for using XML files and the DOM. First, an XML file is parsed to produce a DOM. The DOM contains Element nodes (one per tag) and Text nodes (one per data value).
- You understand the following regarding relational databases (Rdb):

- A Rdb is a collection of one or more tables used to store data. Each table has rows representing data instances (e.g., a student) and columns representing data values (e.g., student has name "Ahmad" and an id of 123456). Relational databases are based on predicate logic and set theory.
- SQL (structured query language) provides capabilities to create and manipulate databases. DDL (data definition language) statements are used to create a database, tables, indexes, and other types of database objects. DML (data manipulation language) statements are used to create, read, update, and delete data in tables.
- Both Java and Python provide support for using Rdb. First, you connect to a relational database server and a specific database. You then execute SQL statements to manipulate the data in the database.

Exercises

Discussion Questions

1. Are certain types of data relationships (i.e., one-to-one, one-to-many, many-to-many) easier to represent using XML? using Rdb?
2. When comparing XML and Rdb, what are the advantages of using XML to persistently store your data?
3. When comparing XML and Rdb, what are the advantages of using Rdb to persistently store your data?

References

1. Refsnes data: XML tutorial. Available via https://www.w3schools.com/xml/default.asp. Cited 17 Jul 2019
2. Refsnes data: XML DOM tutorial. Available via https://www.w3schools.com/xml/dom_intro.asp. Cited 17 Jul 2019
3. Refsnes data: SQL tutorial. Available via https://www.w3schools.com/sql/default.asp. Cited 19 Jul 2019
4. Oracle corporation: MySQL 5.7 reference manual. Available via https://dev.mysql.com/doc/refman/5.7/en/. Cited 19 Jul 2019