5.5.3 XML Schema

Having settled on a particular design for an XML document, we need to be able to write down the design in some way.

We need the design written down so that we can check that an XML document follows the design that we have chosen and so that we can communicate our design to others so that we can share XML documents that have this particular format. In particular, we need to write the design down in such a way that a computer understands the design, so that a computer can check that an XML document obeys the design.

Yes, we need to learn another computer language.

The way that the XML design can be specified is by creating a schema for an XML document, which is a description of the structure of the document. A number of technologies exist for specifying XML schema, but we will focus on the Document Type Definition (DTD) language.

A DTD is a set of rules for an XML document. It contains element declarations that describe which elements are permitted within the XML document, in what order, and how they may be nested within each other. The DTD also contains attribute declarations that describe what attributes an element can have, whether attributes are optional or not, and what sort of values each attribute can have.

5.5.4 Case study: Point Nemo (continued)

Figure 5.16 shows the temperature data at Point Nemo in an XML format (this is a reproduction of part of Figure 5.13 for convenience).

The structure of this XML document is as follows. There is a single overall temperatures element that contains all other elements. There are several elements containing various sorts of metadata: a variable element containing a description of the variable that has been measured; a filename element and a filepath element containing information about the file from which these data were extracted; and three elements, subset, longitude, and latitude, that together describe the temporal and spatial limits of this data set. Finally, there are a number of case elements that contain the core temperature data; each case element contains a temperature measurement and the date of the measurement as attributes.

A DTD describing this structure is shown in Figure 5.17.

The DTD code consists of two types of declarations. There must be an
"itdl" — 2008/9/30 — 7:45 — page 111 — #131

Data Storage 111

<?xml version="1.0"?>
<temperatures>
    <variable>Mean TS from clear sky composite (kelvin)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_dsets/data/</filepath>
    <subset>93 points (TIME)</subset>
    <longitude>123.8W(-123.8)</longitude>
    <latitude>48.8S</latitude>
    <case date="16-JAN-1994" temperature="278.9" />
    <case date="16-FEB-1994" temperature="280" />
    <case date="16-MAR-1994" temperature="278.9" />
    <case date="16-APR-1994" temperature="278.9" />
    <case date="16-MAY-1994" temperature="277.8" />
    <case date="16-JUN-1994" temperature="276.1" />
    ...
</temperatures>

Figure 5.16: The first few lines of the surface temperature at Point Nemo in an XML format. This is a reproduction of part of Figure 5.13.

<!ELEMENT temperatures (variable,
    filename,
    filepath,
    subset,
    longitude,
    latitude,
    case*)>
<!ELEMENT variable (#PCDATA)>
<!ELEMENT filename (#PCDATA)>
<!ELEMENT filepath (#PCDATA)>
<!ELEMENT subset (#PCDATA)>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT case EMPTY>
<!ATTLIST case
date ID #REQUIRED
temperature CDATA #IMPLIED>

Figure 5.17: A DTD for the XML format used to store the surface temperature at Point Nemo (see Figure 5.13). The line numbers (in grey) are just for reference.
<![ELEMENT> declaration for each type of element that appears in the XML design and there must be an <!ATTLIST> declaration for every element in the design that has one or more attributes.

The main purpose of the <!ELEMENT> declarations is to specify what is allowed as the content of a particular type of element. The simplest example of an <!ELEMENT> declaration is for case elements (line 14) because they are empty (they have no content), as indicated by the keyword EMPTY. The components of this declaration are shown below.

```
keyword:  <!ELEMENT case EMPTY>
element name:  <!ELEMENT case EMPTY>
keyword:  <!ELEMENT case EMPTY>
```

The keywords ELEMENT and EMPTY will be the same for the declaration of any empty element. All that will change is the name of the element.

Most other elements are similarly straightforward because their contents are just text, as indicated by the #PCDATA keyword (lines 8 to 13). These examples demonstrate that the declaration of the content of the element is specified within parentheses. The components of the declaration for the longitude element are shown below.

```
keyword:  <!ELEMENT longitude (#PCDATA)>
element name:  <!ELEMENT longitude (#PCDATA)>
parentheses:  <!ELEMENT longitude (#PCDATA)>
element content:  <!ELEMENT longitude (#PCDATA)>
```

The temperatures element is more complex because it can contain other elements. The declaration given in Figure 5.17 (lines 1 to 7) specifies seven elements (separated by commas) that are allowed to be nested within a temperatures element. The order of these elements within the declaration is significant because this order is imposed on the elements in the XML document. The first six elements, variable to latitude, are compulsory because there are no modifiers after the element names; exactly one of each element must occur in the XML document. The case element, by contrast, has an asterisk, *, after it, which means that there can be zero or more case elements in the XML document.

The purpose of the <!ATTLIST> declarations in a DTD are to specify which attributes each element is allowed to have. In this example, only the case elements have attributes, so there is only one <!ATTLIST> declaration (lines 16 to 18). This declaration specifies three things for each attribute: the
name of the attribute, what sort of value the attribute can have, and whether
the attribute is compulsory or optional. The components of this declaration
are shown below.

<table>
<thead>
<tr>
<th>keyword</th>
<th>&lt;!ATTLIST case</th>
</tr>
</thead>
<tbody>
<tr>
<td>element name</td>
<td>&lt;!ATTLIST case</td>
</tr>
<tr>
<td>attribute name</td>
<td>date</td>
</tr>
<tr>
<td>attribute value</td>
<td>date ID #REQUIRED</td>
</tr>
<tr>
<td>compulsory attribute</td>
<td>date ID #REQUIRED</td>
</tr>
<tr>
<td>attribute name</td>
<td>temperature</td>
</tr>
<tr>
<td>attribute value</td>
<td>temperature CDATA #IMPLIED&gt;</td>
</tr>
<tr>
<td>optional attribute</td>
<td>temperature CDATA #IMPLIED&gt;</td>
</tr>
</tbody>
</table>

The date attribute for case elements is compulsory (#REQUIRED) and the
value must be unique (ID). The temperature attribute is optional (#IMPLIED)
and, if it occurs, the value can be any text (CDATA).

Section 6.2 describes the syntax and semantics of DTD files in more detail.

The rules given in a DTD are associated with an XML document by adding
a Document Type Declaration as the second line of the XML document.
This can have one of two forms:

**DTD inline:**

The DTD can be included within the XML document. In the Point
Nemo example, it would look like this:

```xml
<?xml version="1.0"?>
<!DOCTYPE temperatures [
  DTD code
]>
<temperatures>
  ...
```

**External DTD**

The DTD can be in an external file, say pointnemotemp.dtd, and the
XML document can refer to that file:

```xml
<?xml version="1.0"?>
<!DOCTYPE temperatures SYSTEM "pointnemotemp.dtd">
<temperatures>
  ...
```

The DRY principle suggests that an external DTD is the most sensible
approach because then the same DTD rules can be applied efficiently to many XML documents.

An XML document is said to be well-formed if it obeys the basic rules of XML syntax. If the XML document also obeys the rules given in a DTD, then the document is said to be valid. A valid XML document has the advantage that we can be sure that all of the necessary information for a data set has been included and has the correct structure, and that all data values have the correct sort of value.

The use of a DTD has some shortcomings, such as a lack of support for precisely specifying the data type of attribute values or the contents of elements. For example, it is not possible to specify that an attribute value must be an integer value between 0 and 100. There is also the difficulty that the DTD language is completely different from XML, so there is another technology to learn. XML Schema is an XML-based technology for specifying the design of XML documents that solves both of those problems, but it comes at the cost of much greater complexity. This complexity has lead to the development of further technologies that simplify the XML Schema syntax, such as Relax NG.

Standard schema

So far we have discussed designing our own XML schema to store data in an XML document. However, many standard XML schema already exist, so another option is simply to choose one of those instead and create an XML document that conforms to the appropriate standard.

These standards have typically arisen in a particular area of research or business. For example, the Statistical Data and Metadata eXchange (SDMX) format has been developed by several large financial institutions for sharing financial data, and the Data Documentation Initiative (DDI) is aimed at storing metadata for social science data sets.

One downside is that these standards can be quite complex and may require expert assistance and specialized software to work with the appropriate format, but the upside is integration with a larger community of researchers and compatibility with a wider variety of software tools.

5.5.5 Advantages and disadvantages

We will now consider XML not just as an end in itself, but as one of many possible storage formats. In what ways is the XML format better or worse
than other storage options, particularly the typical unstructured plain text format that we saw in Section 5.2.

A self-describing format

The core advantage of an XML document is that it is self-describing. The tags in an XML document provide information about where the data is stored within the document. This is an advantage because it means that humans can find information within the file easily. That is true of any plain text file, but it is especially true of XML files because the tags essentially provide a level of documentation for the human reader. For example, the XML element shown below not only makes it easy to determine that the value 48.8S constitutes a single data value within the file, but it also makes it clear that this value is a north-south geographic location.

```xml
<latitude>48.8S</latitude>
```

The fact that an XML document is self-describing is an even greater advantage from the perspective of the computer. An XML document provides enough information for software to determine how to read the file, without any further human intervention. Looking again at the line containing latitude information.

```xml
<latitude>48.8S</latitude>
```

There is enough information for the computer to be able to detect the value 48.8S as a single data value, and the computer can also record the latitude label so that if a human user requests the information on latitude, the computer knows what to provide.

One consequence of this feature that may not be immediately obvious is that it is much easier to modify the structure of data within an XML document compared to a plain text file. The location of information within an XML document is not so much dependent on where it occurs within the file, but where the tags occur within the file. As a trivial example, consider reversing the order of the following lines in the Point Nemo XML file.

```xml
<longitude>123.8W(-123.8)</longitude>
<latitude>48.8S</latitude>
```

If the information was stored in the reverse order, as shown below, the task of retrieving the information on latitude would be exactly the same. This can be a huge advantage if larger modifications need to be made to a data
set, such as adding an entire new variable.

```xml
<latitude>48.88</latitude>
<longitude>-123.8W(-123.8)</longitude>
```

### Representing complex data structures

The second main advantage of the XML format is that it can accommodate complex data structures. Consider the hierarchical data set in Figure 5.15. Because XML elements can be nested within each other, this sort of data set can be stored in a sensible fashion with families grouped together to make parent-child relations implicit and avoid repetition of the parent data. The plain text representation of these data are reproduced from page 74 below along with a possible XML representation.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>33</td>
<td>male</td>
</tr>
<tr>
<td>Julia</td>
<td>32</td>
<td>female</td>
</tr>
<tr>
<td>John</td>
<td>33</td>
<td>male</td>
</tr>
<tr>
<td>Julia</td>
<td>32</td>
<td>female</td>
</tr>
<tr>
<td>John</td>
<td>33</td>
<td>male</td>
</tr>
<tr>
<td>Julia</td>
<td>32</td>
<td>female</td>
</tr>
<tr>
<td>John</td>
<td>33</td>
<td>male</td>
</tr>
<tr>
<td>Julia</td>
<td>32</td>
<td>female</td>
</tr>
<tr>
<td>John</td>
<td>33</td>
<td>male</td>
</tr>
<tr>
<td>Julia</td>
<td>32</td>
<td>female</td>
</tr>
<tr>
<td>David</td>
<td>45</td>
<td>male</td>
</tr>
<tr>
<td>Debbie</td>
<td>42</td>
<td>female</td>
</tr>
<tr>
<td>David</td>
<td>45</td>
<td>male</td>
</tr>
<tr>
<td>Debbie</td>
<td>42</td>
<td>female</td>
</tr>
<tr>
<td>David</td>
<td>45</td>
<td>male</td>
</tr>
<tr>
<td>Debbie</td>
<td>42</td>
<td>female</td>
</tr>
</tbody>
</table>

```xml
<family>
  <parent gender="male" name="John" age="33" />
  <parent gender="female" name="Julia" age="32" />
  <child gender="male" name="Jack" age="6" />
  <child gender="female" name="Jill" age="4" />
  <child gender="male" name="John jnr" age="2" />
</family>

<family>
  <parent gender="male" name="David" age="45" />
  <parent gender="female" name="Debbie" age="42" />
  <child gender="male" name="Donald" age="16" />
  <child gender="female" name="Dianne" age="12" />
</family>
```

The XML format is superior in the sense that the information about each person is only recorded once. Another advantage is that it would be very easy to represent a wider range of situations using the XML format. For
example, if we wanted to allow for a family unit to have a third parent (e.g., a step-parent), that would be straightforward in XML, but it would be much more awkward in the fixed rows-and-columns plain text format.

Data integrity

Another important advantage of the XML format is that it provides some level of checking on the correctness of the data file (a check on the data integrity). First of all, there is the fact that any XML document must obey the rules of XML, which means that we can use a computer to check that an XML document at least has a sensible structure.

If an XML document also has a DTD, then we can perform much more rigid checks on the correctness of the document. If data values are stored as attribute values, it is possible for the DTD to provide checks that the data values themselves are valid. The XML Schema language provides even greater facilities for specifying limits and ranges on data values.

Verbosity

The major disadvantage of XML is that it generates large files. With it being a plain text format, it is not memory efficient to start with, then with all of the additional tags around the actual data, files can become extremely large. In many cases, the tags can take up more room than the actual data!

These issues can be particularly acute for research data sets, where the structure of the data may be quite straightforward. For example, geographic data sets containing many observations at fixed locations naturally form a 3-dimensional array of values, which can be represented very simply and efficiently in a plain text or binary format. In such cases, having highly repetitive XML tags around all values can be very inefficient indeed.

The verbosity of XML is also a problem for entering data into an XML format. It is just too laborious for a human to enter all of the tags by hand, so, in practice, it is only sensible to have a computer generate XML documents.

Costs of complexity

It should also be acknowledged that the additional sophistication of XML creates additional costs. Users have to be more educated and the software
has to be more complex, which means that fewer software packages are able to cope with data stored as XML.

In summary, the fact that computers can read XML easily and effectively, plus the fact that computers can produce XML rapidly (verbosity is less of an issue for a computer), means that XML is an excellent format for transferring information between different software programs. XML is a good language for computers to use to talk to each other, with the added bonus that humans can still easily eavesdrop on the conversation.

Recap

**XML is a language for describing a data set.**

XML consists of elements and attributes, with data values stored as the content of elements or as the values of attributes.

*The design of an XML document—the choice of elements and attributes—is important. One approach has an element for each different object that has been measured, with the actual measurements recorded as attributes of the appropriate element.*

*The DTD language can be used to formally describe the design of an XML document.*

*The major advantage of XML is that XML documents are self-describing, which means that each data value is unambiguously labeled within the file, so that a computer can find data values without requiring additional information about the file.*

### 5.6 Databases

When a data set becomes very large, or even just very complex in its structure, the ultimate storage solution is a **database**.

The term “database” can be used generally to describe any collection of information. In this section, the term “database” means a **relational database**, which is a collection of data that is organized in a particular way.

The actual physical storage mechanism for a database—whether binary formats or text formats are used, whether one file or many files are used—will not concern us. We will only be concerned with the high-level, conceptual organisation of the data and will rely on software to decide how best to
store the information in files.

The software that handles the representation of the data in computer memory, and allows us to work at a conceptual level, is called a database management system (DBMS), or in our case, more specifically a relational database management system (RDBMS).

The main benefits of databases for data storage derive from the fact that databases have a formal structure. We will spend much of this section describing and discussing how databases are designed, so that we can appreciate the benefits of storing data in a database and so that we know enough to be able to work with data that has been stored in a database.

5.6.1 The database data model

We are not concerned with the file formats that are used to store a database. Instead, we will deal with the conceptual components used to store data in a database.

A relational database consists of a set of tables, where a table is conceptually just like a plain text file or a spreadsheet: a set of values arranged in rows and columns. The difference is that there are usually several tables in a single database, and the tables in a database have a much more formal structure than a plain text file or a spreadsheet.

In order to demonstrate the concepts and terminology of databases, we will work with a simple example of storing information about books. The entire set of information is shown in Figure 5.18 but we will only consider specific subsets of book information at various stages throughout this section in order to demonstrate different ideas about databases.

Shown below is a simple example of a database table that contains infor-
information about some of the books in our data set. This table has three columns—the ISBN of the book, the title of the book, and the author of the book—and four rows, with each row representing one book.

<table>
<thead>
<tr>
<th>ISBN</th>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0618260307</td>
<td>The Hobbit</td>
<td>J. R. R. Tolkien</td>
</tr>
<tr>
<td>0908606664</td>
<td>Slinky Malinki</td>
<td>Lynley Dodd</td>
</tr>
<tr>
<td>0393310728</td>
<td>How to Lie with Statistics</td>
<td>Darrell Huff</td>
</tr>
<tr>
<td>0908783116</td>
<td>Mechanical Harry</td>
<td>Bob Kerr</td>
</tr>
</tbody>
</table>

Each table in a database has a unique name and each column within a table has a unique name within that table.

Each column in a database table also has a **data type** associated with it, so all values in a single column are the same sort of data. In the book database example, all values in all three columns are text or character values. The ISBN is stored as text, not as an integer, because it is a sequence of 10 digits (as opposed to a decimal value). For example, if we stored the ISBN as an integer, we would lose the leading 0.

Each table in a database has a **primary key**. The primary key must be unique for every row in a table. In the book table, the ISBN provides a perfect primary key because every book has a different ISBN.

It is possible to create a primary key by combining the values of two or more columns. This is called a **composite primary key**. A table can only have one primary key, but the primary key may be composed from more than one column. We will see some examples of composite primary keys later in this chapter.

A database containing information on books might also contain information on book publishers. Below we show another table **in the same database** containing information on publishers.

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mallinson Rendel</td>
<td>NZ</td>
</tr>
<tr>
<td>2</td>
<td>W. W. Norton</td>
<td>USA</td>
</tr>
<tr>
<td>3</td>
<td>Houghton Mifflin</td>
<td>USA</td>
</tr>
</tbody>
</table>

In this table, the values in the ID column are all integers. The other columns all contain text. The primary key in this table is the ID column.

Tables within the same database can be related to each other using **foreign keys**. These are columns in one table that specify a value from the primary key in another table. For example, we can relate each book in the
book_table to a publisher in the publisher_table by adding a foreign key to the book_table. This foreign key consists of a column, pub, containing the appropriate publisher ID. The book_table now looks like this:

<table>
<thead>
<tr>
<th>ISBN</th>
<th>title</th>
<th>author</th>
<th>pub</th>
</tr>
</thead>
<tbody>
<tr>
<td>0618260307</td>
<td>The Hobbit</td>
<td>J. R. R. Tolkien</td>
<td>3</td>
</tr>
<tr>
<td>0908606664</td>
<td>Slinky Malinki</td>
<td>Lynley Dodd</td>
<td>1</td>
</tr>
<tr>
<td>0393310728</td>
<td>How to Lie with Statistics</td>
<td>Darrell Huff</td>
<td>2</td>
</tr>
<tr>
<td>0908783116</td>
<td>Mechanical Harry</td>
<td>Bob Kerr</td>
<td>1</td>
</tr>
</tbody>
</table>

Notice that two of the books in the book_table have the same publisher, with a pub value of 1. This corresponds to the publisher with an ID value of 1 in the publisher_table, which is the publisher called Mallinson Rendel.

Also notice that a foreign key column in one table does not have to have the same name as the primary key column that it refers to in another table. The foreign key column in the book_table is called pub, but it refers to the primary key column in the publisher_table called ID.

### 5.6.2 Database notation

The examples of database tables in the previous section have shown the contents of each database table. In the next section, on Database Design, it will be more important to describe the design, or structure, of a database table—the table schema. For this purpose, the contents of each row are not important; instead we are most interested in how many tables there are and which columns are used to make up those tables.

We can describe a database design simply in terms of the names of tables, the names of columns, which columns are primary keys, and which columns are foreign keys.

The notation that we will use is a simple text description, with primary keys and foreign keys indicated in square brackets. The description of a foreign key includes the name of the table and the name of the column that the foreign key refers to. For example, these are the schema for the publisher_table and the book_table in the book database:

```
publisher_table ( ID [PK], name, country )

```
The diagram below shows one way that this design could be visualized. Each “box” in this diagram represents one table in the database, with the name of the table as the heading in the box. The other names in each box are the names of the columns within the table; if the name is bold, then that column is part of the primary key for the table and if the name is italic, then that column is a foreign key. Arrows are used to show the link between a foreign key in one table and the primary key in another table.

The \texttt{publisher\_table} has three columns and the column named \texttt{ID} is the primary key for the table.

The \texttt{book\_table} also has four columns. In this table, the primary key is the \texttt{ISBN} column and the \texttt{pub} column is a foreign key that refers to the \texttt{ID} column in the \texttt{publisher\_table}.

\section*{5.6.3 Database design}

Like we saw with XML documents in Section 5.5.2, databases allow us to store information in a variety of ways, which means that there are design decisions to be made. In this section, we will briefly discuss some of the issues relating to database design.

The design of a database comes down to three things: how many tables are required; what information goes in each table; and how the tables are linked to each other. The remainder of this section provides some rules and guidelines for determining a solution for each of these steps.

This section provides neither an exhaustive discussion nor a completely rigorous discussion of database design. The importance of this section is to provide a basic introduction to some useful ideas and ways to think about data. A basic understanding of these issues is also necessary for us to be able to work with data that has been stored in a database.
Entities and attributes

One way to approach database design is to think in terms of entities, their attributes, and the relationships between them.

An entity is most easily thought of as a person, place, or physical object (e.g., a book), an event, or a concept. An attribute is a piece of information about the entity. For example, the title, author, and ISBN are all attributes of a book entity.

In terms of a research data set, each variable in the data set corresponds to an attribute. The task of designing a database to store the data set comes down to assigning each variable to a particular entity.

Having decided upon a set of entities and their attributes, a database design consists of a separate table for each entity and a separate column within each table for each attribute of the corresponding entity.

Rather than storing a data set as one big table of information, this rule suggests that we should use several tables, with information about different entities in separate tables. In the book database example, there is information about at least two entities, books and publishers, so we have a separate table for each of these.

These ideas of entities and attributes are the same ideas that were discussed for XML design back in Section 5.5.2, just with different terminology.

Relationships

A relationship is an association between entities. For example, a publisher publishes books and a book is published by a publisher. Relationships are represented in a database by foreign key-primary key pairs, but the details depend on the cardinality of the relationship—whether the relationship is one-to-one, many-to-one, or many-to-many.

For example, a book is published by exactly one publisher, but a publisher publishes many books, so the relationship between books and publishers is many-to-one.

This sort of relationship can be represented by placing a foreign key in the table for books (the “many” side) that refers to the primary key in the table for publishers (the “one” side). This is the design that we have already seen, on page 121, where the book_table has a foreign key, pub, that refers to the primary key, ID, in the publisher_table.

One-to-one relationships can be handled similarly to many-to-one relation-
ships (it does not matter which table gets the foreign key), but many-to-many relationships are more complex.

In our book database example, we can identify another sort of entity: authors.

In order to accommodate information about authors in the database, there should be another table for author information. In the example below, the table only contains the author’s name, but other information, such as the author’s age and nationality could be added.

\[
\text{author_table} \quad (\text{ID [PK]}, \text{name})
\]

What is the relationship between books and authors? An author can write several books and a book can have more than one author, so this is an example of a many-to-many relationship.

A many-to-many relationship can only be represented by creating a new table in the database.

For example, we can create a table, called the \text{book_author_table}, that contains the relationship between authors and books. This table contains a foreign key that refers to the author table and a foreign key that refers to the book table. The representation of book entities, author entities, and the relationship between them now consists of three tables, as shown below.

\[
\begin{align*}
\text{author_table} & \quad (\text{ID [PK]}, \text{name}) \\
\text{book_table} & \quad (\text{ISBN [PK]}, \text{title}, \\
& \quad \text{pub [FK publisher_table.ID]}) \\
\text{book_author_table} & \quad (\text{ID [PK]}, \\
& \quad \text{book [FK book_table.ISBN]}, \\
& \quad \text{author [FK author_table.ID]})
\end{align*}
\]

The book database design, with author information included, is shown in the diagram below.
The contents of these tables for several books are shown below. The author table just lists the authors for whom we have information:

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Lynley Dodd</td>
</tr>
<tr>
<td>5</td>
<td>Eve Sutton</td>
</tr>
</tbody>
</table>

The book_table just lists the books that are in the database:

<table>
<thead>
<tr>
<th>ISBN</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>0908606664</td>
<td>Slinky Malinki</td>
</tr>
<tr>
<td>1908606206</td>
<td>Hairy Maclary from Donaldson’s Dairy</td>
</tr>
<tr>
<td>0908606273</td>
<td>My Cat Likes to Hide in Boxes</td>
</tr>
</tbody>
</table>

The book_author_table contains the association between books and authors:

<table>
<thead>
<tr>
<th>ID</th>
<th>book</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0908606664</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1908606206</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0908606273</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0908606273</td>
<td>5</td>
</tr>
</tbody>
</table>

Notice that author 2 (Lynley Dodd) has written more than one book and book 0908606273 (My Cat Likes to Hide in Boxes) has more than one author.
Designing for data integrity

Another reason for creating a table in a database is for the purpose of constraining the set of possible values for an attribute. For example, if the table of authors records the gender of the author, it can be useful to have a separate table that contains the possible values of gender. The column in the author table then becomes a foreign key referring to the gender table and, because a foreign key must match the value of the corresponding primary key, we have a check on the validity of the gender values in the author table.

The redesigned author table and gender table are described below.

\[
\text{author\_table} \quad \begin{array}{l}
\text{( ID \{PK\}, name, } \\
\text{ gender \{FK gender\_table.ID\} )}
\end{array}
\]

\[
\text{gender\_table} \quad \begin{array}{l}
\text{( ID \{PK\}, gender )}
\end{array}
\]

The \text{gender\_table} only contains the set of possible gender values, as shown below.

\[
\begin{array}{ll}
\text{ID} & \text{gender} \\
\hline
1 & \text{male} \\
2 & \text{female}
\end{array}
\]

The final book database design, consisting of five tables, is shown in the diagram below.
Database normalization

Another way to approach database design is to choose tables and columns within tables based on whether they satisfy a set of rules called normal forms.

This, more formal, process of database design is called normalization.

There are several normal forms, but we will only mention the first three because these will cover most simple situations.

The proper definition of normalization depends on more advanced relational database concepts that are beyond the scope of this book, so the descriptions below are just to give a feel for how the process works.

First normal form

First normal form requires that the columns in a table must be atomic, there should be no duplicative columns, and every table must have a primary key.

The first part of this rule says that a column in a database table must only contain a single value. As an example, consider the following possible design for a table for storing information about books. There is one column for the title of the book and another column for all authors of the book.

book_table ( title, authors )

Two rows of this table are shown below.

<table>
<thead>
<tr>
<th>title</th>
<th>authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slinky Malinki</td>
<td>Lynley Dodd</td>
</tr>
<tr>
<td>My Cat Likes to Hide in Boxes</td>
<td>Eve Sutton, Lynley Dodd</td>
</tr>
</tbody>
</table>

The first column of this table is acceptable because it just contains one piece of information: the title of the book. However, the second column is not atomic because it contains a list of authors for each book. The book on the second row has two authors recorded in the authors column. This violates first normal form.

The second part of the rule says that a table cannot have two columns containing the same information. For example, the following possible redesign of the book table provides a solution to the previous problem by having a separate column for each author of the book.

book_table ( title, author1, author2 )
Two rows from this table are shown below.

<table>
<thead>
<tr>
<th>title</th>
<th>author1</th>
<th>author2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slinky Malinki</td>
<td>Lynley Dodd</td>
<td>NULL</td>
</tr>
<tr>
<td>My Cat Likes to Hide in Boxes</td>
<td>Eve Sutton</td>
<td>Lynley Dodd</td>
</tr>
</tbody>
</table>

This solves the problem of atomic columns because each column only contains the name of one author. However, the table has two duplicative columns: \texttt{author1} and \texttt{author2}. These two columns both record the same information, author names, so this design also violates first normal form.

A possible redesign that satisfies the requirement that each column is atomic and not duplicative is shown below. We now have just one column for the book title and one column for the names of the authors.

\texttt{book\_table ( title, author )}

The contents of this table are shown below. Notice that the second book now occupies two rows because it has two authors.

<table>
<thead>
<tr>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slinky Malinki</td>
<td>Lynley Dodd</td>
</tr>
<tr>
<td>My Cat Likes to Hide in Boxes</td>
<td>Eve Sutton</td>
</tr>
<tr>
<td>My Cat Likes to Hide in Boxes</td>
<td>Lynley Dodd</td>
</tr>
</tbody>
</table>

The final part of the first normal form rule says that there must be a column in the table that has a unique value in every row (or it must be possible to combine several columns to obtain a unique value for every row). In other words, every table must have a primary key.

Can we find a primary key in the table above?

Neither the \texttt{title} column nor the \texttt{author} column by themselves are any use as a primary key because some values repeat in each of these columns.

We could combine the two columns together to create a composite primary key. However, it is also important to think about not just the data that are currently in a table, but also what possible values could be entered into the table in the future (or even just in theory). In this case, it is possible that a book could be published in both hard cover and paperback formats, both of which would have the same title and author, so while a composite primary key would work for the three rows shown below it is not necessarily a smart choice.
As described previously, for the case of information about books, a great candidate for a primary key is the book's ISBN because it is guaranteed to be unique for a particular book. If we add an ISBN column to the table, we can finally satisfy first normal form, though it still has to be a composite primary key involving the combination of ISBN and author.

book_table ( ISBN [PK],
            title,
            author [PK] )

The contents of this table are shown below.

<table>
<thead>
<tr>
<th>ISBN</th>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0908606664</td>
<td>Slinky Malinki</td>
<td>Lynley Dodd</td>
</tr>
<tr>
<td>0908606273</td>
<td>My Cat Likes to Hide in Boxes</td>
<td>Lynley Dodd</td>
</tr>
<tr>
<td>0908606273</td>
<td>My Cat Likes to Hide in Boxes</td>
<td>Eve Sutton</td>
</tr>
</tbody>
</table>

This is not an ideal solution for storing this information, but at least it satisfies first normal form. Consideration of second and third normal form will help to improve the design.

Second normal form

Second normal form requires that a table must be in first normal form and all columns in the table must relate to the entire primary key.

This rule formalizes the idea that there should be a table for each entity in the data set.

As a very basic example, consider the following table that contains information about authors and publishers. The primary key of this table is the author ID. In other words, each row of this table only concerns a single author.

author_table ( ID [PK], name, publisher)

Two rows from this table are shown below.

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Lynley Dodd</td>
<td>Mallinson Rendel</td>
</tr>
<tr>
<td>5</td>
<td>Eve Sutton</td>
<td>Mallinson Rendel</td>
</tr>
</tbody>
</table>

The name column of this table relates to the primary key (the ID); this is the name of the author. However, the publisher column does not relate to the primary key. This is the publisher of a book. In
other words, the information about publishers belongs in a table about publishers (or possibly a table about books), not in a table about authors.

As a more subtle example, consider the table that we ended up with at the end of first normal form.

\[
\text{book\_table} \ ( \ ISBN \ [PK], \\
\quad \quad \quad \text{title}, \quad \\
\quad \quad \quad \author \ [PK] )
\]

<table>
<thead>
<tr>
<th>ISBN</th>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0908606664</td>
<td>Slinky Malinki</td>
<td>Lynley Dodd</td>
</tr>
<tr>
<td>0908606273</td>
<td>My Cat Likes to Hide in Boxes</td>
<td>Lynley Dodd</td>
</tr>
<tr>
<td>0908606273</td>
<td>My Cat Likes to Hide in Boxes</td>
<td>Eve Sutton</td>
</tr>
</tbody>
</table>

The primary key for this table is a combination of ISBN and author (each row of the table carries information about one author of one book).

The title column relates to the ISBN: this is the title of the book. However, the title column does not relate to the author: this is not the title of the author!

The table needs to be split into two tables, one with the information about books and one with the information about authors. Shown below is the book-related information separated into its own table.

\[
\text{book\_table} \ ( \ ISBN \ [PK], \\
\quad \quad \quad \text{title} )
\]

<table>
<thead>
<tr>
<th>ISBN</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>0908606664</td>
<td>Slinky Malinki</td>
</tr>
<tr>
<td>0908606273</td>
<td>My Cat Likes to Hide in Boxes</td>
</tr>
</tbody>
</table>

It is important to remember that each of the new tables that we create to satisfy second normal form must also satisfy first normal form. In this case, it would be wise to add an ID column to act as the primary key for the table of authors, as shown below, because it is entirely possible that two distinct authors could share the same name.

\[
\text{author\_table} \ ( \ ID \ [PK], \\
\quad \quad \quad \author )
\]
As this example makes clear, having split a table into two or more pieces, it is very important to link the pieces together by adding one or more foreign keys, based on the relationships between the tables. In this case, the relationship is many-to-many, so the solution requires a third table to provide a link between books and authors.

```
book_author_table ( ID [PK],
                    author [FK author_table.ID] )
```

```
ID  book    author
---  -------- -------
 2  0908606664  2
 6  0908606273  2
 7  0908606273  5
```

**Third normal form**

Third normal form requires that a table must be in second normal form and all columns in the table must relate only to the primary key (not to each other).

This rule further emphasizes the idea that there should be a separate table for each entity in the data set. For example, consider the following table for storing information about books.

```
book_table ( ISBN [PK],
             title,
             publisher,
             country )
```

```
ISBN  title             publisher        country
------- ------------------ --------------- -------
0395193958  The Hobbit   Houghton Mifflin  USA
0836827848  Slinky Malinki Mallinson Rendel  NZ
0908783116  Mechanical Harry Mallinson Rendel  NZ
```

The primary key of this table is the ISBN, which uniquely identifies a book. The **title** column relates to the book; this is the title of the book. Each row of this table is about one book.

The **publisher** column also relates to the book; this is the publisher of the book. However, the **country** column does not relate directly
to the book; this is the country of the publisher. That obviously is information about the book—it is the country of the publisher of the book—but the relationship is indirect, through the publisher.

There is a simple heuristic that makes it easy to spot this sort of problem in a database table. Notice that the information in the publisher and country columns is identical for the books published by Mallinson Rendel. When two or more columns repeat the same information over and over, it is a sure sign that either second or third normal form is not being met.

In this case, the analysis of the table suggests that there should be a separate table for information about the publisher.

Applying the rules of normalization usually results in the creation of multiple tables in a database. The previous discussion of relationships should be consulted for making sure that any new tables are linked to at least one other table in the database using a foreign-key, primary-key pair.

**Denormalization**

The result of normalization is a well-organized database that should be easy to maintain. However, normalization may produce a database that is slow in terms of accessing the data (because the data from many tables has to be recombined).

**Denormalization** is the process of deliberately violating normal forms, typically in order to produce a database that can be accessed more rapidly.

### 5.6.4 Flashback: The DRY Principle

A well designed database, particularly one that satisfies third normal form, will have the feature that each piece of information is stored only once. Less repetition of data values means that a well-designed database will usually require less memory than storing an entire data set in a naïve single-table format. Less repetition also means that a well-designed database is easier to maintain and update, because if a change needs to be made, it only needs to be made in one location. Furthermore, there is less chance of errors creeping into the data set. If there are multiple copies of information, then it is possible for the copies to disagree, but with only one copy there can be no disagreements.

These ideas are an expression of the DRY principle from Section 2.7.
VARIABLE : Mean TS from clear sky composite (kelvin)
FILENAME : ISCCPMonthly_avg.nc
FILEPATH : /usr/local/fer_dsets/data/
SUBSET : 24 by 24 points (LONGITUDE-LATITUDE)
TIME : 16-JAN-1995 00:00
   113.8W 111.2W 108.8W 106.2W 103.8W 101.2W 98.8W ...
   27 28 29 30 31 32 33 ...
   36.2N / 51: 272.7 270.9 269.7 273.2 275.6 277.3 ...
   33.8N / 50: 279.5 279.5 275.0 277.3 279.5 281.6 ...
   31.2N / 49: 284.7 284.7 281.6 281.6 280.5 282.2 284.7 ...
   28.8N / 48: 289.3 286.8 286.8 283.7 284.2 286.8 287.8 ...
   26.2N / 47: 292.2 293.2 287.8 287.8 285.8 288.8 291.7 ...
   23.8N / 46: 294.1 295.0 296.5 286.8 286.8 285.2 289.8 ...
   ...

Figure 5.19: One of the plain text files from the original format of the Data Expo data set, which contains data for one variable for one month. The file contains information on latitude and longitude that is repeated in every other plain text file in the original format (for each variable and for each month; in total, over 500 times).

A well-designed database is the ultimate embodiment of the DRY principle for data storage.

5.6.5 Case Study: The Data Expo (continued)

The Data Expo data set consists of seven atmospheric variables recorded at 576 locations for 72 time points (every month for 6 years), plus elevation data for each location (see Section 5.2.8).

The data were originally stored as 505 plain text files, where each file contains the data for one variable for one month. Figure 5.19 shows the first few lines from one of the plain text files.

As we have discussed earlier in this chapter, this simple format makes the data very accessible. However, this is an example where a plain text format is quite inefficient, because many values are repeated. For example, the longitude and latitude information for each location in the data set is stored in every single file, which means that that information is repeated over 500 times! That not only takes up more storage space than is necessary, but it also violates the DRY principle, with all of the negative consequences that follow from that.
In this section, we will consider how the Data Expo data set could be stored as a relational database.

To start with, we will consider the problem from an entities and attributes perspective. What entities there are in the data set? In this case, the different entities that are being measured are relatively easy to identify. There are measurements on the atmosphere, and the measurements are taken at different locations and at different times. We have information about each time point (i.e., a date), we have information about each location (longitude and latitude and elevation), and we have several measurements on the atmosphere. This suggests that we should have three tables: one for atmospheric measures, one for locations, and one for time points.

It is also useful to look at the data set from a normalization perspective. For this purpose, we will start with all of the information in a single table (only 7 rows shown):

<table>
<thead>
<tr>
<th>date</th>
<th>lon</th>
<th>lat</th>
<th>elv</th>
<th>chi</th>
<th>cmid</th>
<th>clo</th>
<th>ozone</th>
<th>press</th>
<th>stemp</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995-01-16</td>
<td>-56.25</td>
<td>36.25</td>
<td>0.0</td>
<td>25.5</td>
<td>38.5</td>
<td>298.0</td>
<td>1000.0</td>
<td>289.8</td>
<td>288.8</td>
<td></td>
</tr>
<tr>
<td>1995-01-16</td>
<td>-56.25</td>
<td>33.75</td>
<td>0.0</td>
<td>23.5</td>
<td>36.5</td>
<td>290.0</td>
<td>1000.0</td>
<td>290.7</td>
<td>289.8</td>
<td></td>
</tr>
<tr>
<td>1995-01-16</td>
<td>-56.25</td>
<td>31.25</td>
<td>0.0</td>
<td>20.5</td>
<td>36.5</td>
<td>286.0</td>
<td>1000.0</td>
<td>291.7</td>
<td>290.7</td>
<td></td>
</tr>
<tr>
<td>1995-01-16</td>
<td>-56.25</td>
<td>28.75</td>
<td>0.0</td>
<td>12.5</td>
<td>37.5</td>
<td>280.0</td>
<td>1000.0</td>
<td>293.6</td>
<td>292.2</td>
<td></td>
</tr>
<tr>
<td>1995-01-16</td>
<td>-56.25</td>
<td>26.25</td>
<td>0.0</td>
<td>10.0</td>
<td>35.0</td>
<td>272.0</td>
<td>1000.0</td>
<td>296.0</td>
<td>294.1</td>
<td></td>
</tr>
<tr>
<td>1995-01-16</td>
<td>-56.25</td>
<td>23.75</td>
<td>0.0</td>
<td>12.5</td>
<td>32.0</td>
<td>270.0</td>
<td>1000.0</td>
<td>297.4</td>
<td>296.0</td>
<td></td>
</tr>
<tr>
<td>1995-01-16</td>
<td>-56.25</td>
<td>21.25</td>
<td>0.0</td>
<td>7.0</td>
<td>31.0</td>
<td>260.0</td>
<td>1000.0</td>
<td>297.8</td>
<td>296.5</td>
<td></td>
</tr>
</tbody>
</table>

In terms of first normal form, all columns are atomic and there are no duplicative columns, and we can, with a little effort, find a (composite) primary key: we need a combination of date, lon (longitude), and lat (latitude) to get a unique value for all rows.

Moving on to second normal form, the column elv (elevation) immediately fails. The elevation at a particular location clearly relates to the longitude and latitude of the location, but it has very little to do with the date. We need a new table to hold the longitude, latitude, and elevation data.

The new table design and the first three rows of data are shown below.

<table>
<thead>
<tr>
<th>lon</th>
<th>lat</th>
<th>elv</th>
</tr>
</thead>
<tbody>
<tr>
<td>-56.25</td>
<td>36.25</td>
<td>0.0</td>
</tr>
<tr>
<td>-56.25</td>
<td>33.75</td>
<td>0.0</td>
</tr>
<tr>
<td>-56.25</td>
<td>31.25</td>
<td>0.0</td>
</tr>
</tbody>
</table>
This “location” table is in third normal form. It has a primary key (a combination of longitude and latitude), and the elevation column relates directly to the entire primary key.

Going back to the original table, the remaining columns of atmospheric measurements are all related to the primary key; the data in these columns represents an observation at a particular location at a particular time point.

However, we now have two tables rather than just one, so we must make sure that the tables are linked to each other, and in order to achieve this, we need to determine the relationships between the tables.

We have two tables, one representing atmospheric measurements, at various locations and times, and one representing information about the locations. What is the relationship between these tables? Each location (each row of the location table) corresponds to several measurements, but each individual measurement (each row of the measurement table) corresponds to only one location, so the relationship is many-to-one.

This means that the table of measurements should have a foreign key that references the primary key in the location table. The design could be expressed like this:

```
location_table ( longitude [PK],
                latitude [PK],
                elevation )

measure_table ( date [PK],
                longitude [PK] [FK location_table.longitude],
                latitude [PK] [FK location_table.latitude],
                cloudhigh, cloudlow, cloudmid, ozone,
                pressure, surftemp, temperature )
```

```
<table>
<thead>
<tr>
<th>measure_table</th>
<th>location_table</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>longitude</td>
</tr>
<tr>
<td>longitude</td>
<td>latitude</td>
</tr>
<tr>
<td>cloudhigh</td>
<td></td>
</tr>
<tr>
<td>cloudlow</td>
<td></td>
</tr>
<tr>
<td>cloudmid</td>
<td></td>
</tr>
<tr>
<td>ozone</td>
<td></td>
</tr>
<tr>
<td>pressure</td>
<td></td>
</tr>
<tr>
<td>surftemp</td>
<td></td>
</tr>
<tr>
<td>temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>elevation</td>
</tr>
</tbody>
</table>
```
Both tables have composite primary keys. The `measure_table` also has a composite foreign key, to refer to the composite primary key in the `location_table`). Finally, the `longitude` and `latitude` columns have roles in both the primary key and the foreign key of the `measure_table`.

A possible adjustment to the database design is to consider a surrogate auto-increment key—a column that just corresponds to the row number in the table—as the primary key for the location table, because the natural primary key is quite large and cumbersome. This leads to a final design that can be expressed as below.

```
location_table ( ID [PK],
               longitude, latitude, elevation )
```

```
measure_table ( date [PK],
               location [PK] [FK location_table.ID],
               cloudhigh, cloudlow, cloudmid, ozone,
               pressure, surftemp, temperature )
```

Another adjustment would be to break out the `date` column into a separate table. This is partly motivated by the idea of data integrity; a separate table for dates would ensure that all dates in the `measure_table` are valid dates. Also, if the table for dates uses an auto-increment ID column, the `date` column in the `measure_table` can become just a simple integer, rather than a lengthy date value. Finally, the table of date information can have the year and month information split into separate columns, which can make it more useful to work with the date information.
The final Data Expo database design is shown below.

date_table ( ID [PK], date, month, year )

location_table ( ID [PK],
    longitude, latitude, elevation )

measure_table ( date [PK] [FK date_table.ID],
    location [PK] [FK location_table.ID],
    cloudhigh, cloudlow, cloudmid, ozone,
    pressure, surftemp, temperature )

As a final check, we should confirm that these tables all satisfy third normal form.

Each table has a primary key, all columns are atomic, and there are no duplicative columns, so first normal form is satisfied. All of the columns in each table correspond to the primary key of the table—in particular, each measurement in the measure_table corresponds to a particular combination of date and location—so second normal form is also satisfied. The tables mostly also satisfy third normal form because columns generally relate only to the primary key in the table. However, it could be argued that, in the date_table, the month and year columns relate to the date column as well as to the primary key of the table. This is a good demonstration of a possible justification for denormalization; we have split out these columns because we anticipate that they will be useful for asking questions of the database in the future. The ideas of normalization should be used as guides for achieving a sensible database design, but other considerations may also come into play.
5.6.6 Advantages and disadvantages

The previous sections have demonstrated that databases are a lot more complex than most of the other data storage options in this chapter. In this section, we will look at what we can gain by using a database to store a data set and what the costs are compared to other storage formats.

The relatively formal data model of relational databases, and the relatively complex processes that go into designing an appropriate database structure, are worthwhile because the resulting structure enforces constraints on the data in a database, which means that there are checks on the accuracy and consistency of data that is stored in a database. In other words, databases ensure better data integrity.

For example, the database structure ensures that all values in a single column of a table are of the same data type (e.g., they are all numbers). It is possible, when setting up a database, to enforce quite specific constraints on what values can appear in a particular column of a table. Section 8.3 provides some information on this topic the creation of datasets.

Another important structural feature of databases is the existence of foreign keys and primary keys. Database software will enforce the rule that a primary key must be unique for every row in a table and it will enforce the rule that the value of a foreign key must refer to an existing primary key value.

Databases tend to be used for large data sets because, for most DBMS, there is no limit on the size of a database. However, even when a data set is not enormous, there are advantages to using a database because the organisation of the data can improve accuracy and efficiency. In particular, databases allow the data to be organized in a variety of ways so that, for example, data with a hierarchical structure can be stored in an efficient and natural way.

Databases are also advantageous because most DBMS provide advanced features that are far beyond what is provided by the software that is used to work with data in other formats (e.g., text editors and spreadsheet programs). These features include the ability to allow multiple people to access and even modify the data at once and advanced security to control who has access to the data and who is able to modify the data.

The first cost to consider is monetary. The commercial database systems offered by Oracle and Microsoft can be very expensive, although open-source options exist (see Section 5.6.9) to relieve that particular burden. However, there is also the cost of acquiring or hiring the expertise necessary to create, maintain, and interact with data stored in a database.
Another disadvantage of using a database as a storage format is that the data can only be accessed using a specific piece of DBMS software.

Finally, all of the sophistication and flexibility that a database provides may just not be necessary for small data sets or for data sets that have a simple structure. For example, a binary format such as netCDF is very well suited to a geographical data set where observations are made on a regular grid of locations and at a fixed set of time points and it will outperform a more general-purpose database solution.

The investment required to create and maintain a database means that it will not always be an appropriate choice.

5.6.7 Flashback: Database design and XML design

In Section 5.5.2 we discussed some basic ideas for deciding how to represent a data set in an XML format.

The ideas of database design that we have discussed in Section 5.6.3—entities, attributes, relationships, and normalization—are very similar to the ideas from XML design, if a little more formal.

This similarity arises from the fact that we are trying to solve essentially the same problem in both cases and this can be reflected in a simple correspondence between database designs and XML designs for the same data set.

As a rough guideline, a database table can correspond to a set of XML elements of the same type. Each row of the table will correspond to a single XML element, with each column of values recorded as a separate attribute within the element. The caveats about when attributes cannot be used still apply (see page 108).

Simple one-to-one or many-to-one relationships can be represented in XML by nesting several elements (the many) within another element (the one). More complex relationships cannot be solved by nesting, but attributes corresponding to primary keys and foreign keys can be used to emulate relationships between entities via XML elements that are not nested.

5.6.8 Case study: The Data Expo (continued)

The Data Expo data set consists of several atmospheric measurements taken at many different locations and at several time points. A database design that we developed for storing these data consisted of three tables: one
for the location data, one for the time data, and one for the atmospheric measurements (see Section 5.6.5). The database schema is reproduced below for easy reference.

date_table ( ID [PK], date, month, year )

location_table ( ID [PK],
                longitude, latitude, elevation )

measure_table ( date [PK] [FK date_table.ID],
                location [PK] [FK location_table.ID],
                cloudhigh, cloudlow, cloudmid, ozone,
                pressure, surftemp, temperature )

We can translate this database design into an XML document design very simply, by creating a set of elements for each table, with attributes for each column of data. For example, the fact that there is a table for location information implies that we should have location elements, with an attribute for each column in the database table. The data for the first few locations is represented like this in a database table:

<table>
<thead>
<tr>
<th>ID</th>
<th>lon</th>
<th>lat</th>
<th>elv</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-113</td>
<td>36</td>
<td>1526.25</td>
</tr>
<tr>
<td>2</td>
<td>-111</td>
<td>36</td>
<td>1759.56</td>
</tr>
<tr>
<td>3</td>
<td>-108</td>
<td>36</td>
<td>1948.38</td>
</tr>
</tbody>
</table>

The same data could be represented in XML like this:

```xml
<location id="1" longitude="-113.75" latitude="36.25" elevation="1526.25" />
<location id="2" longitude="-111.25" latitude="36.25" elevation="1759.56" />
<location id="3" longitude="-108.75" latitude="36.25" elevation="1948.38" />
```

As an analogue of the primary keys in the database design, the DTD for this XML design could specify id as an ID attributes (see Section 6.2.2).

An XML element for the first row from the date_table might look like this (again with id as an ID attribute in the DTD):

```xml
<date id="1" date="1995-01-16"
      month="January" year="1995" />
```
Because there is a many-to-many relationship between locations and dates, it would not make sense to nest the corresponding XML elements. Instead, the XML elements that correspond to the rows of the measure_table could include attributes that refer to the relevant location and date elements. The following code shows an example of what a measure XML element might look like.

```xml
<measure date="1" location="1"
    cloudhigh="26.0" cloudmid="34.5"
    cloudlow="7.5" ozone="304.0"
    pressure="835.0" surftemp="272.7"
    temperature="272.1" />
```

In order to enforce the data integrity of the attributes date and location, the DTD for this XML design would specify these as IDREF attributes (see Section 6.2.2).

### 5.6.9 Database software

Every different database software product has its own format for storing the database tables on disk, which means that data stored in a database is only accessible via one specific piece of software.

This means that, if we are given data stored in a particular database format, we are forced to use the corresponding software. Something that slightly alleviates this problem is the existence of a standard language for querying databases. We will meet this language, SQL, in the next chapter.

If we are in the position of storing information in a database ourselves, there are a number of fully-featured open source database management systems to choose from. PostgreSQL[^6] and MySQL[^7] are very popular options, though they require some investment in resources and expertise to set up because they have separate client and server software components. SQLite[^8] is much simpler to set up and use, especially for a database that only requires access by a single person working on a single computer.

Section 7.2.14 provides a very brief introduction to SQLite.

The major proprietary database systems include Oracle, Microsoft SQL Server, and Microsoft Access. The default user interface for these software products is based on menus and dialogs so they are beyond the scope and

[^6]: http://www.postgresql.org/
[^7]: http://www.mysql.com/
[^8]: http://www.sqlite.org/
interest of this book. Nevertheless, in all of these, as with the default interfaces for the open source database software, it is possible to write computer code to access the data. Writing these data queries is the topic of the next chapter.

Recap

A database consists of one or more tables. Each column of a database table contains only one type of information, corresponding to one variable from a data set.

A primary key uniquely identifies each row of a table. A primary key is a column in a table with a different value on every row.

A foreign key relates one table to another within a database. A foreign key is a column in a table that refers to the values in the primary key of another table.

A database should be designed so that information about different entities resides in separate tables.

Normalization is a way to produce a good database design.

Databases can handle large data sets and data sets with a complex structure, but databases require specific software and a certain level of expertise.

5.7 Further reading

“Modern Database Management”
by Jeffrey A. Hoffer, Mary Prescott, and Fred McFadden
Comprehensive text book treatment of databases and associated technologies, with more of a business focus. Includes many advanced topics beyond the scope of this book.
Summary

Simple text data is stored using 1 byte per character. Integers are stored using 2 or 4 bytes and real values typically use 4 or 8 bytes.

There is a limit to the size of numbers that can be stored digitally and for real values there is a limit on the precision with which values can be stored.

Plain text files are the simplest data storage solution, with the advantage that they are simple to use, work across different computer platforms, and work with virtually any software. The main disadvantage to plain text files is their lack of standard structure, which means that software requires human input to determine where data values reside within the file. Plain text files are also generally larger and slower than other data storage options.

CSV (commas-separated values) files offer the most standardized plain text format.

Binary formats tend to provide smaller files and faster access speeds. The disadvantage is that data stored in a binary format can only be accessed using specific software.

Spreadsheets are ubiquitous, flexible, and easy to use. However, they lack structure so should be used with caution.

XML is a language that can be used for marking up data. XML files are plain text, but provide structure that allows software to automatically determine the location of data values within the file (XML files are self-describing).

Databases are sophisticated, but relatively complex. They are useful for storing very large or very complex data sets, but require specific software and much greater expertise.
XML Reference

XML (the eXtensible Markup Language) is a data description language that can be used for storing data. It is particularly useful as a format for sharing information between different software systems.

The information in this chapter describes XML 1.0, which is a W3C Recommendation.

Within this chapter, any code written in a sans-serif oblique font represents a general template; that part of the code will vary depending on the names of the elements and the names of the attributes that are used to store a particular data set.

6.1 XML syntax

The first line of an XML document should be a declaration that this is an XML document, including the version of XML being used.

```xml
<?xml version="1.0"?>
```

It is also useful to include a statement of the encoding used in the file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

The main content of an XML document consists entirely of XML elements. An element usually consists of a start tag, an end tag, with plain text content or other XML elements in between.

A start tag is of the form `<elementName>` and an end tag has the form `</elementName>`.

The following code shows an example of an XML element.

```xml
<filename>ISCCPMonthly_avg.nc</filename>
```
The components of this XML element are shown below.

| element:  | `<filename>ISCCPMonthly_avg.nc</filename>` |
| start tag: | `<filename>` | `ISCCPMonthly_avg.nc</filename>` |
| content:  | `<filename>ISCCPMonthly_avg.nc</filename>` |
| end tag:   | `<filename>ISCCPMonthly_avg.nc</filename>` |

The start tag may include attributes of the form `attrName="attrValue"`. The attribute value must be enclosed within double-quotes.

The names of XML elements and XML attributes are case-sensitive.

It is also possible to have an empty element, which consists of a single tag, with attributes. In this case, the tag has the form `<elementName />`.

The following code shows an example of an empty XML element with two attributes.

```xml
<case date="16-JAN-1994" temperature="278.9" />
```

The components of this XML element are shown below.

| element name: | `<case date="16-JAN-1994" temperature="278.9" />` |
| attribute name: | `<case date="16-JAN-1994"` |
| attribute value: | `date="16-JAN-1994"` |
| attribute name: | `temperature="278.9"` |
| attribute value: | `temperature="278.9"` |

XML elements may be nested; an XML element may have other XML elements as its content. An XML document must have a single root element, which contains all other XML elements in the document.

The following code shows a very small, but complete, XML document. The root element of this document is the `temperatures` element. The `filename` and `case` elements are nested within the `temperatures` element.

```xml
<?xml version="1.0"?>
<temperatures>
  <filename>ISCCPMonthly_avg.nc</filename>
  <case date="16-JAN-1994" temperature="278.9"/>
</temperatures>
```
Table 6.1: The predefined XML entities.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>less-than sign</td>
<td><code>&amp;lt;</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater-than sign</td>
<td><code>&amp;gt;</code></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>ampersand</td>
<td><code>&amp;amp;</code></td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>quotation mark</td>
<td><code>&amp;quot;</code></td>
</tr>
<tr>
<td><code>'</code></td>
<td>apostrophe</td>
<td><code>&amp;apos;</code></td>
</tr>
</tbody>
</table>

A comment in XML is anything between the delimiters `<!--` and `-->`.

For the benefit of human readers, the contents of an XML element are usually indented. However, whitespace is preserved within XML so this is not always possible when including plain text content.

In XML code, certain characters, such as the greater-than and less-than signs, have special meanings. Table 6.1 lists these special characters and also gives the escape sequence required to produce the normal, literal meaning of the characters.

A special syntax is provided for escaping an entire section of plain text content for the case where many such special characters are included. Any text between the delimiters `<![CDATA[` and `]]>` is treated as literal.

### 6.2 Document Type Definitions

An XML document that obeys the rules of the previous section is described as **well-formed**.

It is also possible to specify additional rules for the structure and content of an XML document, via a **schema** for the document. If the document is well-formed and also obeys the rules given in a schema, then the document is described as **valid**.

The Document Type Definition language (DTD) is a language for describing the schema for an XML document. DTD code consists of **element declarations** and **attribute declarations**.
6.2.1 Element declarations

An element declaration should be included for every different type of element that will occur in an XML document. Each declaration describes what content is allowed inside a particular element. An element declaration is of the form:

```
<!ELEMENT elementName elementContents>
```

The *elementContents* specifies whether an element can contain plain text, or other elements (and if so, which elements, in what order), or whether the element must be empty. Some possible values are shown below.

**EMPTY**

The element is empty.

**ANY**

The element may contain anything (other elements, plain text, or both).

**(#PCDATA)**

The element may contain plain text.

**elementA)**

The element must contain exactly one *elementA* element. The parentheses, ( and ), are essential in this example and all others below.

**elementA+**

The element may contain zero or more *elementA* elements. The asterisk, *, indicates “zero or more”.

**elementA+**

The element must contain one or more *elementA* elements. The plus sign, +, indicates “one or more”.

**elementA?**

The element must contain zero or one *elementA* elements. The question mark, ?, indicates “zero or one”.

**elementA, elementB**

The element must contain exactly one *elementA* element and exactly one *elementB* element. The element names are separated from each other by commas.

**elementA|elementB**

The element must contain either exactly one *elementA* element or exactly one *elementB* element. The vertical bar, |, indicates alternatives.
The element may contain plain text, or a single elementA element, or zero or more elementB elements. The asterisk, *, is inside the parentheses so only applies to the elementB element.

(#PCDATA | elementA | elementB)*
The element may contain plain text, plus zero or more occurrences of elementA elements and elementB elements. The asterisk, *, is outside the parentheses so applies to all elements within the parentheses.

### 6.2.2 Attribute declarations

An attribute declaration should be included for every different type of element that can have attributes. The declaration describes which attributes an element may have, what sort of values the attribute may take, and whether the attribute is optional or compulsory. An attribute declaration is of the form:

```
<!ATTLIST elementName attrName attrType attrDefault
    ...>
```

The attrType controls what value the attribute can have. It can have one of the following forms:

**CDATA**

The attribute can take any value. Attribute values must always be plain text and escape sequences (XML entities) must be used for special XML characters (see Table 6.1).

**ID**

The value of this attribute must be unique for all elements of this type in the document (i.e., a unique identifier). This is similar to a primary key in a database table.

The value of an ID attribute must *not* start with a digit.

**IDREF**

The value of this attribute must be the value of some other element’s ID attribute. This is similar to a foreign key in a database table.

**(option1 | option2)**

This form provides a list of the possible values for the attribute. The
list of options is given, separated by vertical bars, |. This is a good way to limit an attribute to only valid values (e.g., only "male" or "female" for a gender attribute).

```xml
<!ATTLIST elementName
gender (male|female) #REQUIRED>
```

The attrDefault either provides a default value for the attribute or states whether the attribute is optional or required (i.e., must be specified). It can have one of the following forms:

- **value**
  - This is the default value for the attribute.

- **#IMPLIED**
  - The attribute is optional. It is valid for elements of this type to contain this attribute, but it is not required.

- **#REQUIRED**
  - The attribute is required so it must appear in all elements of this type.

### 6.2.3 Including a DTD

A DTD can be embedded within an XML document or the DTD can be located within a separate file and referred to from the XML document.

The DTD information is included within a `DOCTYPE` declaration following the XML declaration. An inline DTD has the form:

```xml
<!DOCTYPE rootElementName [ DTD code ]>
```

An external DTD stored in a file called `file.dtd` would be referred to as follows:

```xml
<!DOCTYPE rootElementName SYSTEM "file.dtd">
```

The name following the keyword `DOCTYPE` must match the name of the root element in the XML document.
<?xml version="1.0"?>
<!DOCTYPE temperatures [ 
<!ELEMENT temperatures (filename, case)> 
<!ELEMENT filename (#PCDATA)> 
<!ELEMENT case EMPTY> 
<!ATTLIST case 
   date CDATA #REQUIRED 
   temperature CDATA #IMPLIED> ]>
<temperatures>
   <filename>ISCCPMonthly_avg.nc</filename>
   <case date="16-JAN-1994" temperature="278.9"/>
</temperatures>

Figure 6.1: A well-formed and valid XML document, with an embedded DTD. The line numbers (in grey) are just for reference.

6.2.4 An example

Figure 6.1 shows a very small, well-formed and valid XML document with an embedded DTD.

Line 1 is the required XML declaration.

Lines 2 to 9 provide a DTD for the document. This DTD specifies that the root element for the document must be a `temperatures` element (line 2). The `temperatures` element must contain one `filename` element and one `case` element (line 3). The `filename` element must contain only plain text (line 4) and the `case` element must be empty (line 5).

The `case` element must have a `date` attribute (line 7) and may also have a `temperature` attribute (line 8). The value of both attributes can be arbitrary text (CDATA).

The elements within the XML document that mark up the actual data values are on lines 10 to 14.
6.3 Further reading

The W3C XML 1.0 Specification
http://www.w3.org/TR/2006/REC-xml-20060816/
The formal and official definition of XML. Quite technical.

The w3schools XML Tutorial
http://www.w3schools.com/xml/
Quick, basic tutorial-based introduction to XML.

The w3schools DTD Tutorial
http://www.w3schools.com/dtd/
Quick, basic tutorial-based introduction to DTDs.

The W3C Validation Service
http://validator.w3.org/
This will check raw XML files as well as HTML documents.

libxml2
http://xmlsoft.org/
This software includes a command-line tool, xmllint for checking XML code, including validating it against a DTD.