5

Reading and Writing Data with SQL

"Any sufficiently advanced technology is indistinguishable from magic." (Arthur C. Clarke)

SQL helps to explain relational database modeling. None of this is magic, and is much easier to understand than you might think.

This chapter shows how the relational database model is used from an application perspective. There is little point in understanding something, such as relational database modeling, without seeing it applied in some way, no matter how simple. With that in mind, this chapter looks at how a database model is accessed when it contains data in a database. A relational database model contains tables, and the records in those tables are accessed using Structured Query Language (SQL). SQL is used to create the database access sections of applications. When a database model is correctly designed, creation of SQL code is a simple process. Great difficulties in coding of SQL queries in particular can often indicate serious database model design flaws.

This book is all about relational database modeling; therefore, it is necessary to explain only the basis of SQL in this book as pertaining directly to helping in the understanding of relational database modeling. How can describing the basic details of SQL help with the understanding of relational database modeling? The answer to this question is very simple. SQL uses a relational database model to change data in a database, and to retrieve data from that database. SQL essentially applies all the structures created by the relational database modeling process. SQL — and particularly SQL queries and their simplicity — is a direct result of the underlying structure of a database model. In other words, the better and more appropriate a database model is, the easier building SQL code for applications will be. Another way to look at this is that a database model is most easily utilized by applications, when matching application structure. It makes perfect sense to describe the basics of using SQL because it demonstrates the use of a relational database model in action.

In this chapter, you learn about the following:

- ❑ What SQL is
- ❑ The origins of SQL
- ❑ Many types of queries
- ❑ Changing data in tables
- ❑ Changing table structure

Defining SQL

SQL is a non-procedural language used for accessing field and record values in relational database tables.

A procedural programming language is a language where there can be dependencies between sequential commands. For example, when setting a variable X = 1 on a line, that variable X can be used to reference the value 1, as the variable X in subsequent lines of programming code. A non-procedural programming language does not allow communication between different lines of programming code. Non-procedural languages also do not allow use of procedures. A procedure allows intra-program communication by passing values in and out of procedures.

Keep in mind the following when considering what SQL is.

- ❑ SQL is structured in the sense that it is used to access structured data, in a structured manner, or retrieve data from organized structures. Those organized structures are tables.
- ❑ Use of the word "language" implies a computer programming language. Computer programming languages are often used to get information from a computer.
- ❑ A *non-procedural language* essentially submits a single command, with a questioning or querying type nature. SQL is a non-procedural language consisting of single commands, where the database itself does a lot of the work in deciding how to get that information. On the other hand, a *procedural language* contains blocks of commands. Those blocks of commands are sequences of distinct steps, typically where each successive step is dependent on the result of the previous command in the sequence.
- ❑ In most relational databases, SQL does have the capability to function in a limited procedural fashion, allowing the programmer to determine partially how a database is accessed. Traditionally, SQL procedural code is used to write what are called *stored procedures*, *triggers*, *database events*, or *database procedures*. There are other names for these code blocks, depending on the database in use. Procedural SQL code is generally simplistic and can sometimes allow the inclusion of basic programming constructs such as `IF` statements, `CASE` statements, and so on.

So, SQL is a simple non-procedural programming language allowing single line commands. These single-line commands are used to access data stored in database table structures.

The Origins of SQL

IBM created the original relational database technology. SQL was created as an uncomplicated, non-procedural way of accessing data from an IBM-created relational database. SQL was initially called "Sequel." Thus, SQL is often pronounced as "sequel." For some other databases, SQL is pronounced by representing each letter, as in "ess-queue-ell." The meaning of the two pronunciations is identical.

The query language used to access an object database is called Object Database Query Language (ODQL). The acronym "QL" thus means "query language," a language used to query a database.

In its most primitive form, SQL stems from the idea of a reporting language, devised in theory by the inventor of the relational database model. The roots of SQL lie in retrieval of sets of data. What this means is that SQL is intended as a language to retrieve many records from one or many tables at once, yielding a result set. SQL was not originally intended to retrieve individual records from a relational database, as exact record matches, common in transactional and OLTP databases. However, SQL can now be used to do precisely just that, and with excellent efficiency. SQL is now used in modern-day relational database engines to retrieve both sets of records, and individual records, in transactional, OLTP, and data warehouse databases.

What does all this mean without using a plethora of nasty long words? In short, SQL was developed as a shorthand method of retrieving information from relational databases. SQL has become the industry standard over the last 20 years. The following is an example of a *query* (a request or question put to the database) written in SQL, in this case all the records in the AUTHOR table will be retrieved:

```
SELECT AUTHOR_ID, NAME FROM AUTHOR;

AUTHOR_ID NAME
---------- --------------------------------
         1 Orson Scott Card
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         5 Jerry Pournelle
         6 William Shakespeare
         7 Kurt Vonnegut
```

SQL for Different Databases

SQL as implemented in different database engines is not standardized. Each database vendor developed a unique relational database, and relational database management system (database management toolkit). The result was different relational databases having different strengths. The vendors often altered and extended the standard form of SQL to take advantage of the way in which their individual products were written. The result is that relational database products from different vendors, although similar in nature, and even appearance, are often very different internally. Additionally, the different relational databases are different both in internal structural characteristics, and in the way they are used. And that's only the database engine itself. There is also use of different computer hardware platforms and operating systems. The larger database vendors service multiple operating systems, with completely different versions of the database software, applicable to different operating systems, even down to different flavors of Unix.

Many of the smaller-scale database engines such as dBase and MSAccess are only written for a single platform and operating system. In the case of dBase and MSAccess, that system is PC-based, running the Windows operating system.

So, what are the basics of SQL?

The Basics of SQL

The basics of SQL consist of a number of parts. This section briefly introduces what simple SQL can do. As you read through this chapter, each is explained individually by example.

- *Query commands* — Querying a database is performed with a single command called the `SELECT` command. The `SELECT` command creates queries, and has various optional clauses that include performing functions such as filtering and sorting. Queries are used to retrieve data from tables in a database. There are various ways in which data can be retrieved from tables:

Some database engines refer to SQL commands as SQL statements. Don't get confused. The two terms mean exactly the same thing. I prefer the term "command." This is merely a personal preference on my part. The term "clause" is usually applied to subset parts of commands.

  - *Basic query* — The most simple of queries retrieves all records from a single table.
  - *Filtered query* — A filtered query uses a `WHERE` clause to include or exclude specific records.
  - *Sorted query* — Sorting uses the `ORDER BY` clause to retrieve records in a specific sorted order.
  - *Aggregated query* — The `GROUP BY` clause allows summarizing, grouping, or aggregating of records into summarized record sets. Typically, aggregated queries contain fewer records than the query would produce, if the `GROUP BY` clause were not used. A `HAVING` clause can be used to filter in, or filter out, records in the resulting summarized aggregated record set. In other words, the `HAVING` clause filters the results of the `GROUP BY` clause, and not the records retrieved before aggregation.
  - *Join query* — A join query joins tables together, returning records from multiple tables. Joins can be performed in various ways, including inner joins and outer joins.
  - *Nested queries* — A nested query is also known as a *subquery*, which is a query contained within another query (a *parent* or *calling query*). Nesting implies that subqueries can be nested in multiple layers and thus a subquery itself can also be a calling query of another subquery.
  - *Composite queries* — A composite query is a query that merges multiple query results together, most often using the `UNION` keyword.
- *Data change commands* — The following commands are used to change data in tables in a database.
  - `INSERT` — The `INSERT` command is used to add new records to a table.
  - `UPDATE` — The `UPDATE` command allows changes to one or more records in a table, at once.
  - `DELETE` — The `DELETE` command allows deletion of one or more records from a table, at once.

- *Database structure change commands* — These commands allow alterations to metadata (the data about the data). Metadata in a simple relational database comprises tables and indexes. Table metadata change commands allow creation of new tables, changes to existing tables, and destruction of existing tables, among other more obscure types of operations — too obscure for this book. Table metadata commands are CREATE TABLE, ALTER TABLE, and DROP TABLE commands.

Querying a Database Using SELECT

The following sections examine database queries using the SELECT command in detail, as well as by example.

Basic Queries

The following syntax shows the structure of the SELECT statement and the FROM clause. The SELECT list is the list of fields, or otherwise, usually retrieved from tables. The FROM clause specifies one or more tables from which to retrieve data.

```
SELECT { [alias.]field | expression | [alias.]* [,the rest of the list of fields] }
FROM table [alias] [ , ... ];
```

Please see the Introduction to this book for syntax conventions, known as Backus-Naur Form syntax notation.

The easiest way to understand SQL is by example. Retrieve all fields from the AUTHOR table using the * (star or asterisk) character. The * character tells the query to retrieve all fields in all tables in the FROM clause:

```
SELECT * FROM AUTHOR;

AUTHOR_ID NAME
---------- --------------------------------
         1 Orson Scott Card
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         5 Jerry Pournelle
         6 William Shakespeare
         7 Kurt Vonnegut
```

A semi-colon and carriage return is used to end the query command, submitting the query to the query engine. Not all relational databases execute on a semi-colon. Some databases use a different character; others just a carriage-return.

Specify an individual field by retrieving only the name of the author from the AUTHOR table:

```
SELECT NAME FROM AUTHOR;

NAME
--------------------------------
Orson Scott Card
```

```
James Blish
Isaac Azimov
Larry Niven
Jerry Pournelle
William Shakespeare
Kurt Vonnegut
```

Retrieving specific field names is very slightly more efficient than retrieving all fields using the * character. The * character requires the added overhead of metadata interpretation lookups into the metadata dictionary — to find the fields in the table. In highly concurrent, very busy databases, continual data dictionary lookups can stress out database concurrency handling capacity.

Execute an expression on a single field of the `AUTHOR` table, returning a small section of the author's name:

```
SELECT AUTHOR_ID, SUBSTR(NAME,1,10) FROM AUTHOR;

AUTHOR_ID SUBSTR(NAM
---------- ----------
         1 Orson Scot
         2 James Blis
         3 Isaac Azim
         4 Larry Nive
         5 Jerry Pour
         6 William Sh
         7 Kurt Vonne
```

Execute an expression, but this time involving more than a single field:

```
SELECT E.ISBN, (E.LIST_PRICE * R.RANK) + R.INGRAM_UNITS
FROM EDITION E JOIN RANK R ON (R.ISBN = E.ISBN);

      ISBN (E.LIST_PRICE*R.RANK)+R.INGRAM_UNITS
---------- ------------------------------------
 198711905                              46072.5
 345308999                                 9728
 345336275                                11860
 345438353                                24200
 553278398                                14430
 553293362                                 7985
 553293370                                14815
 553293389                                 8370
 893402095                              14026.5
1585670081                                34600
5557076654                              37632.5
```

Use aliases as substitutes for table names:

```
SELECT A.NAME, P.TITLE, E.ISBN
FROM AUTHOR A JOIN PUBLICATION P USING (AUTHOR_ID)
```

```
  JOIN EDITION E USING (PUBLICATION_ID);

NAME                 TITLE                                    ISBN
-------------------- ---------------------------------- ----------
William Shakespeare  The Complete Works of Shakespeare   198711905
Isaac Azimov         Foundation                          246118318
Isaac Azimov         Foundation                          345308999
Larry Niven          Footfall                            345323440
Larry Niven          Ringworld                           345333926
Isaac Azimov         Foundation                          345334787
Isaac Azimov         Foundation                          345336275
James Blish          A Case of Conscience                345438353
Larry Niven          Lucifer's Hammer                    449208133
Isaac Azimov         Prelude to Foundation               553278398
Isaac Azimov         Second Foundation                   553293362
Isaac Azimov         Foundation and Empire               553293370
Isaac Azimov         Foundation's Edge                   553293389
Isaac Azimov         Foundation                          893402095
James Blish          Cities in Flight                   1585670081
Isaac Azimov         Foundation                         5553673224
Isaac Azimov         Foundation                         5557076654
```

The `USING` *clause in join syntax allows a vague specification of a join field. This assumes that the two joined tables have the required relationship on the field of the required field name. In the previous query, both the* `AUTHOR` *and* `PUBLICATION` *tables have the field* `PUBLICATION_ID`*, and in both tables the same values, one being a primary key, and the other a directly related foreign key.*

Without the alias, the query would simply have table names, much longer strings, making the query a little more difficult to read and code:

```
SELECT AUTHOR.NAME, PUBLICATION.TITLE, EDITION.ISBN
FROM AUTHOR JOIN PUBLICATION USING (AUTHOR_ID)
  JOIN EDITION USING (PUBLICATION_ID);

NAME                 TITLE                                    ISBN
-------------------- ---------------------------------- ----------
William Shakespeare  The Complete Works of Shakespeare   198711905
Isaac Azimov         Foundation                          246118318
Isaac Azimov         Foundation                          345308999
Larry Niven          Footfall                            345323440
Larry Niven          Ringworld                           345333926
Isaac Azimov         Foundation                          345334787
Isaac Azimov         Foundation                          345336275
James Blish          A Case of Conscience                345438353
Larry Niven          Lucifer's Hammer                    449208133
Isaac Azimov         Prelude to Foundation               553278398
Isaac Azimov         Second Foundation                   553293362
Isaac Azimov         Foundation and Empire               553293370
Isaac Azimov         Foundation's Edge                   553293389
Isaac Azimov         Foundation                          893402095
James Blish          Cities in Flight                   1585670081
Isaac Azimov         Foundation                         5553673224
Isaac Azimov         Foundation                         5557076654
```

Using shorter alias names can help to keep SQL code more easily readable, particularly for programmers in the future having to make changes. Maintainable code is less prone to error and much easier to tune properly.

Filtering with the `WHERE` Clause

A filtered query uses the `WHERE` clause to include, or exclude, specific records. The following syntax adds the syntax for the `WHERE` clause to the `SELECT` command:

```
SELECT ...
FROM table [alias] [, ... ]
[ WHERE [table.|alias.] { field | expression } comparison { ... }
  [ { AND | OR } [ NOT ] ... ] ];
```

The `WHERE` *clause is optional.*

Begin with filtering by retrieving the author whose primary key values is equal to 5:

```
SELECT * FROM AUTHOR WHERE AUTHOR_ID = 5;

AUTHOR_ID NAME
---------- --------------------
         5 Jerry Pournelle
```

This filter is efficient because a single record is found using the primary key. A fast index search can be used to find a single record very quickly, even in an extremely large table.

Now find everything other than authors whose primary key value is 5:

```
SELECT * FROM AUTHOR WHERE AUTHOR_ID != 5;

AUTHOR_ID NAME
---------- --------------------
         1 Orson Scott Card
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         6 William Shakespeare
         7 Kurt Vonnegut
```

Filtering using a negative such as `NOT` or `!=` forces a full table scan and ignores all indexes altogether. Searching for something on the premise that it does not exist is extremely inefficient, especially for a very large table. A full table scan is a physical input/output (I/O) read of all the records in a table. Reading an entire table containing billions of records can take a week. Not many programmers have that long to test their queries.

Some small tables are more efficiently read using only the table (a full table scan) and ignoring indexes.

How about authors whose primary key value is less than or equal to 5:

```
SELECT * FROM AUTHOR WHERE AUTHOR_ID <= 5;

AUTHOR_ID NAME
---------- --------------------
         1 Orson Scott Card
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         5 Jerry Pournelle
```

A range search is more efficient than a full table scan for large tables because certain types of indexes, such as BTree indexes, are read efficiently using range scans. A BTree index is built like an upside down tree and searching requires traversal up and down the tree structure. Therefore, both single record and multiple record range scans are efficient using BTree indexes.

This one finds a range:

```
SELECT * FROM AUTHOR WHERE AUTHOR_ID >= 3 AND AUTHOR_ID <= 5;

AUTHOR_ID NAME
---------- --------------------
         3 Isaac Azimov
         4 Larry Niven
         5 Jerry Pournelle
```

Many relational databases use a special operator called `BETWEEN`, which retrieves between a range inclusive of the end points:

```
SELECT * FROM AUTHOR WHERE AUTHOR_ID BETWEEN 3 AND 5;

AUTHOR_ID NAME
---------- --------------------
         3 Isaac Azimov
         4 Larry Niven
         5 Jerry Pournelle
```

There are other ways of filtering (common to many relational databases), such as the LIKE operator. The LIKE operator is somewhat similar to a very simple string pattern matcher. The following query finds all authors with the vowel "a" in their names:

```
SELECT * FROM AUTHOR WHERE NAME LIKE "%a%";

AUTHOR_ID NAME
---------- --------------------
         3 Isaac Azimov
         2 James Blish
         4 Larry Niven
         1 Orson Scott Card
         6 William Shakespeare
```

The LIKE operator is generally not efficient. Simple string pattern matching tends to full-table scan entire tables, no matter how the string is structured.

IN can be used as set membership operator:

```
SELECT * FROM AUTHOR WHERE AUTHOR_ID IN (1,2,3,4,5);

AUTHOR_ID NAME
---------- --------------------
         1 Orson Scott Card
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         5 Jerry Pournelle
```

Traditionally, the IN operator is most efficient when testing against a list of literal values.

The NOT, AND, and OR operators are known as *logical operators*, or sometimes as *logical conditions*. This depends on the database in use. Logical operators allow for Boolean logic in WHERE clause filtering and various other SQL code commands and clauses. Mathematically, the sequence of precedence is NOT, followed by AND, and finally OR. Precedence, covered in the next section, can be altered using parentheses.

Precedence

Precedence is the order of resolution of an expression and generally acts from left to right, across an expression. In other words, in the following expression, each of the first, second, and third expressions are evaluated one after the other:

```
<expression1> AND <expression2> AND <expression3>
```

An expression is a mathematical term representing any part of a larger mathematical expression. Thus, an expression is an expression in itself, can contain other expressions, and can be a subset part of other expressions. So in the expression (((5 + 3) * 23) – 50), (5 + 3) is an expression, so is (5 + 3) * 23, so is ((5 + 3) * 23), and even the number 50 is an expression, in this context.

In the following expression, however, the conjunction of the second and third expressions is evaluated first; then the result is evaluated against the first expression using the `OR` logical operator. This is because the `AND` operator has higher precedence than the `OR` operator:

```
<expression1> OR <expression2> AND <expression3>
```

Higher precedence implies "executed first."

The precedence of evaluation of expressions in the next expression is changed by using the parentheses. Therefore, use of parentheses as in () has higher precedence than `NOT`, `AND`, and `OR`.

```
(<expression1> OR <expression2>) AND <expression3>
```

Aside from logical operator precedence, there is also the factor of *arithmetical precedence*. Basic arithmetic is something we all learned in grade school mathematics. This is to refresh your memory, rather than to insult your intelligence, by explaining the completely obvious. Addition and subtraction have the lowest level of precedence, but they are equal to each other:

```
5 + 4 – 3 = 6
```

It should be plain to see why addition and subtraction have equal precedence because no matter what order in which the numbers are added and subtracted, the result will always be the same. Try it out yourself in your head, and you will understand better. Asking you to do an exercise like this in your head is once again not intended as an intellectual insult; however, just try it and you will understand how simplicity can be used to explain so many things. Perhaps even the answers to life itself could be answered so easily, by breaking all questions into their constituent little pieces.

The ability to break things into small parts to solve small problems is very important when building anything with computers, including relational database models. Object-oriented design is the most modern of software design methodologies. Breaking things into small things is what object- oriented design using programming languages such as Java are all about — breaking things into smaller constituent parts to make the complexity of the whole much easier to implement. In some respects, relational database modeling has some striking similarities in term of breaking down complexity to introduce simplicity. There is beauty in simplicity because it is easy to understand!

Multiplication and division have higher precedence than addition and subtraction but once again are equal in precedence to each other:

```
3 + 4 * 5 = 23 and not 35
```

Remember that parenthesizing a part of an expression changes precedence, giving priority to the parenthesized section:

```
( 3 + 4 ) * 5 = 35
```

Any function such as raising a number to a power, or using a `SUBSTR` function, has the highest level of precedence. Apart from the parenthesized section of course:

```
3 + 4² * 5 = 83
3 * 4 + LENGTH(SUBSTR(NAME, 1, 10)) = 22
```

Some databases and programming languages may represent raising a number to a power in different ways, such as `4^2`, `4^^2`, `EXP(4,2)`, `POWER(4,2)`. *This depends on the database in use.*

So now go back to the `WHERE` clause and utilize the rules of precedence. The following query has precedence executed from left to right. It finds all Hardcover editions, of all books, regardless of the page count or list price. After `PAGES` and `LIST_PRICE` are checked, the query also allows any hard cover edition. The `OR` operator simply overrides the effect of the filters against `PAGES` and `LIST_PRICE`:

```
SELECT ISBN, PRINT_DATE, PAGES, LIST_PRICE, FORMAT FROM EDITION

WHERE PAGES < 300 AND LIST_PRICE < 50 OR FORMAT = "Hardcover";

      ISBN PRINT_DAT      PAGES LIST_PRICE FORMAT
---------- --------- ---------- ---------- --------------
1585670081                  590       34.5 Hardcover
 345438353                  256         12 Paperback
 198711905                 1232      39.95 Hardcover
 345336275 31-JUL-86        285        6.5
 246118318 28-APR-83        234       9.44 Hardcover
```

The next query changes the precedence of the `WHERE` clause filter, from the previous query, preventing the `OR` operator from simply overriding what has already been selected by the filter on the `PAGES` filter (now page counts are all under 300 pages):

```
SELECT ISBN, PRINT_DATE, PAGES, LIST_PRICE, FORMAT FROM EDITION
WHERE PAGES < 300 AND (LIST_PRICE < 50 OR FORMAT = 'Hardcover');

      ISBN PRINT_DAT      PAGES LIST_PRICE FORMAT
---------- --------- ---------- ---------- --------------------------------
 345438353                  256         12 Paperback
 345336275 31-JUL-86        285        6.5
 246118318 28-APR-83        234       9.44 Hardcover
```

Sorting with the `ORDER BY` Clause

Sorting records in a query requires use of the `ORDER BY` clause, whose syntax is as follows:

```
SELECT ...
FROM table [alias] [, ... ]
[ WHERE ... ]
[ ORDER BY { field | expression [ASC| DESC] [ , ... ] } ];
```

The `ORDER BY` *clause is optional.*

Sorting with the `ORDER BY` clause allows resorting into an order other than the natural physical order that records were originally added into a table. This example sorts by `AUTHOR_ID`, contained within the name of the author (the `NAME` field):

```
SELECT * FROM AUTHOR ORDER BY NAME, AUTHOR_ID;

AUTHOR_ID NAME
---------- -------------------
         3 Isaac Azimov
         2 James Blish
         5 Jerry Pournelle
         7 Kurt Vonnegut
         4 Larry Niven
         1 Orson Scott Card
         6 William Shakespeare
```

Some queries, depending on data retrieved, whether tables or indexes are read, which clause are used — can be sorted without use of the `ORDER BY` clause. It is rare but it is possible. Using the `ORDER BY` clause in all situations can be inefficient.

Different databases allow different formats for `ORDER BY` clause syntax. Some formats are more restrictive than others.

Aggregating with the GROUP BY Clause

An aggregated query uses the `GROUP BY` clause to summarize repeating groups of records into aggregations of those groups. The following syntax adds the syntax for the GROUP BY clause:

```
SELECT ...
FROM table [alias] [, ... ]
 [ WHERE ... ]
[ GROUP BY expression [, ... ] [ HAVING condition ] ]
[ ORDER BY ... ];
```

The `GROUP BY` *clause is optional.*

Some databases allow special expansions to the `GROUP BY` *clause, allowing creation of rollup and cubic query output, even to the point of creating highly complex spreadsheet or On-Line Analytical process (OLAP) type analytical output rollups create rollup totals, such as subtotals for each grouping in a nested groups query. Cubic output allows for reporting such as cross-tabbing and similar cross sections of data. Lookup OLAP, rollup and cubic data on the Internet for more information. OLAP is an immense topic in itself and detailed explanation does not belong in this book.*

Note the sequence of the different clauses in the previous syntax. The `WHERE` clause is always executed first, and the `ORDER BY` clause is always executed last. It follows that the `GROUP BY` clause always appears after a `WHERE` clause, and always before an `ORDER BY` clause.

A simple application of the `GROUP BY` clause is to create a summary, as in the following example, creating an average price for all editions, printed by each publisher:

```
SELECT P.NAME AS PUBLISHER, AVG(E.LIST_PRICE)
FROM PUBLISHER P JOIN EDITION E USING (PUBLISHER_ID)
GROUP BY P.NAME;

PUBLISHER                        AVG(E.LIST_PRICE)
-------------------------------- -----------------
Ballantine Books                        8.49666667
Bantam Books                                   7.5
Books on Tape                                29.97
Del Rey Books                                 6.99
Fawcett Books                                 6.99
HarperCollins Publishers                      9.44
L P Books                                     7.49
Overlook Press                                34.5
Oxford University Press                      39.95
Spectra                                        7.5
```

In this example, an average price is returned for each publisher. Individual editions of books are summarized into each average, for each publisher; therefore, individual editions of each book are not returned as separate records because they are summarized into the averages.

The next example selects only the averages for publishers, where that average is greater than 10:

```
SELECT P.NAME AS PUBLISHER, AVG(E.LIST_PRICE)
FROM PUBLISHER P JOIN EDITION E USING (PUBLISHER_ID)
GROUP BY P.NAME
HAVING AVG(E.LIST_PRICE) > 10;

PUBLISHER                        AVG(E.LIST_PRICE)
-------------------------------- -----------------
Books on Tape                                29.97
Overlook Press                                34.5
Oxford University Press                      39.95
```

The `AS` clause in the preceding query renames a field in a query.

The above example filters out aggregated records.

A common programming error is to get the purpose of the `WHERE` and `HAVING` clause filters mixed up. The `WHERE` clause filters records as they are read (as I/O activity takes place) from the database. The `HAVING` clause filters aggregated groups, after all database I/O activity has completed. Don't use the `HAVING` clause when the `WHERE` clause should be used, and visa versa.

Join Queries

A *join query* is a query retrieving records from more than one table. Records from different tables are usually joined on related key field values. The most efficient and effective forms of join are those between directly related primary and foreign key fields. There are a number of different types of joins:

- *Inner Join* — An intersection between two tables using matching field values, returning records common to both tables only. Inner join syntax is as follows:

```
SELECT ...
FROM table [alias] [, ... ]
[
  INNER JOIN table [alias]
[
    USING (field [, ... ])
  | ON (field = field [{AND | OR} [NOT] [ ... ])
  ]
]
[ WHERE ... ] [ GROUP BY ... ] [ ORDER BY ... ];
```

The following query is an inner join because it finds all publishers and related published editions. The two tables are linked based on the established primary key to foreign key relationship. The primary key is in the `PUBLISHER` table on the one side of the one-to-many relationship, between the `PUBLISHER` and `EDITION` tables. The foreign key is precisely where it should be, on the "many" side of the one-to-many relationship.

```
SELECT P.NAME AS PUBLISHER, E.ISBN
FROM PUBLISHER P JOIN EDITION E USING (PUBLISHER_ID);

PUBLISHER                              ISBN
-------------------------------- ----------
Overlook Press                   1585670081
Ballantine Books                  345333926
Ballantine Books                  345336275
Ballantine Books                  345438353
Bantam Books                      553293362
Spectra                           553278398
Spectra                           553293370
Spectra                           553293389
Oxford University Press           198711905
L P Books                         893402095
Del Rey Books                     345308999
Del Rey Books                     345334787
Del Rey Books                     345323440
Books on Tape                    5553673224
Books on Tape                    5557076654
```

```
HarperCollins Publishers          246118318
Fawcett Books                     449208133
```

- *Cross join* — This is also known mathematically as a *Cartesian product*. A cross join merges all records in one table with all records in another table, regardless of any matching values. Cross join syntax is as follows:

```
SELECT ...
FROM table [alias] [, ... ]
[ CROSS JOIN table [alias] ]
[ WHERE ... ] [ GROUP BY ... ] [ ORDER BY ... ];
```

A cross-join simply joins two tables regardless of any relationship. The result is a query where each record in the first table is joined to each record in the second table (a little like a merge):

```
SELECT P.NAME AS PUBLISHER, E.ISBN
FROM PUBLISHER P CROSS JOIN EDITION E;

PUBLISHER                              ISBN
-------------------------------- ----------
Overlook Press                    198711905
Overlook Press                    246118318
Overlook Press                    345308999
Overlook Press                   1585670081
Overlook Press                   5553673224
Overlook Press                   5557076654
Overlook Press                   9999999999
...
Ballantine Books                  198711905
Ballantine Books                  246118318
Ballantine Books                  345308999
...
```

The previous record output has been edited. Some Overlook Press records have been removed, as well as all records returned after the last Ballantine Books record shown.

- *Outer join* — Returns records from two tables as with an inner join, including both the intersection between the two tables, plus records in one table that are not in the other. Any missing values are typically replaced with `NULL` values. Outer joins can be of three forms:
  - *Left outer join* — All records from the left side table plus the intersection of the two tables. Values missing from the right side table are replaced with `NULL` values. Left outer join syntax is as follows:

```
SELECT ...
FROM table [alias] [, ... ]
[
  LEFT OUTER JOIN table [alias]
  [
    USING (field [, ... ])
  | ON (field = field [{AND | OR} [NOT] [ ... ])
  ]
]
[ WHERE ... ] [ GROUP BY ... ] [ ORDER BY ... ];
```

This query finds the intersection between publishers and editions, plus all publishers currently with no titles in print:

```
SELECT P.NAME AS PUBLISHER, E.ISBN
FROM PUBLISHER P LEFT OUTER JOIN EDITION E USING (PUBLISHER_ID);

PUBLISHER                              ISBN
-------------------------------- ----------
Overlook Press                   1585670081
Ballantine Books                  345333926
Ballantine Books                  345336275
Ballantine Books                  345438353
Bantam Books                      553293362
Spectra                           553278398
Spectra                           553293370
Spectra                           553293389
Oxford University Press           198711905
Bt Bound
L P Books                         893402095
Del Rey Books                     345308999
Del Rey Books                     345334787
Del Rey Books                     345323440
Books on Tape                    5553673224
Books on Tape                    5557076654
HarperCollins Publishers          246118318
Fawcett Books                     449208133
Berkley Publishing Group
```

In this example, any publishers with no titles currently in print have `NULL` valued ISBN numbers.

- *Right outer join* — All records from the right side table plus the intersection of the two tables. Values missing from the left side table are replaced with `NULL` values. Right outer join syntax is as follows:

```
SELECT ...
FROM table [alias] [, ... ]
[
  RIGHT OUTER JOIN table [alias]
  [
    USING (field [, ... ])
  | ON (field = field [{AND | OR} [NOT] [ ... ])
  ]
]
[ WHERE ... ] [ GROUP BY ... ] [ ORDER BY ... ];
```

Now, find the intersection between publishers and editions, plus all self-published titles (no publisher):

```
SELECT P.NAME AS PUBLISHER, E.ISBN
FROM PUBLISHER P RIGHT OUTER JOIN EDITION E USING (PUBLISHER_ID);

PUBLISHER                              ISBN
-------------------------------- ----------
Overlook Press                   1585670081
Ballantine Books                  345333926
Ballantine Books                  345336275
Ballantine Books                  345438353
```

```
Bantam Books                      553293362
Spectra                           553278398
Spectra                           553293389
Spectra                           553293370
Oxford University Press           198711905
L P Books                         893402095
Del Rey Books                     345323440
Del Rey Books                     345334787
Del Rey Books                     345308999
Books on Tape                    5553673224
Books on Tape                    5557076654
HarperCollins Publishers          246118318
Fawcett Books                     449208133
                                 9999999999
```

In this example, books without a publisher would have `NULL` valued publishing house entries.

- *Full outer join* — The intersection plus all records from the right side table not in the left side table, in addition to all records from the left side table not in the right side table. Full outer join syntax is as follows:

```
SELECT ...
FROM table [alias] [, ... ]
[
  FULL OUTER JOIN table [alias]
  [
    USING (field [, ... ])
  | ON (field = field [{AND | OR} [NOT] [ ... ])
  ]
]
[ WHERE ... ] [ GROUP BY ... ] [ ORDER BY ... ];
```

This query finds the full outer join, effectively both the left and the right outer joins at the same time:

```
SELECT P.NAME AS PUBLISHER, E.ISBN
FROM PUBLISHER P FULL OUTER JOIN EDITION E USING (PUBLISHER_ID);

PUBLISHER                              ISBN
-------------------------------- ----------
Overlook Press                   1585670081
Ballantine Books                  345333926
Ballantine Books                  345336275
Ballantine Books                  345438353
Bantam Books                      553293362
Spectra                           553278398
Spectra                           553293370
Spectra                           553293389
Oxford University Press           198711905
Bt Bound
L P Books                         893402095
Del Rey Books                     345308999
Del Rey Books                     345334787
Del Rey Books                     345323440
Books on Tape                    5553673224
Books on Tape                     5557076654
HarperCollins Publishers           246118318
Fawcett Books                      449208133
Berkley Publishing Group
                                  9999999999
```

In this example, missing entries of both publishers and editions are replaced with `NULL` values.

- *Self Join* — A self join simply joins a table to itself, and is commonly used with a table containing a hierarchy of records (a denormalized one-to-many relationship). A self join does not require any explicit syntax other than including the same table in the `FROM` clause twice, as in the following example:

```
SELECT P.NAME AS PARENT, C.NAME
FROM SUBJECT P JOIN SUBJECT C ON (C.PARENT_ID = P.SUBJECT_ID);

PARENT           NAME
---------------- ---------------------
Non-Fiction      Self Help
Non-Fiction      Esoteric
Non-Fiction      Metaphysics
Non-Fiction      Computers
Fiction          Science Fiction
Fiction          Fantasy
Fiction          Drama
Fiction          Whodunnit
Fiction          Suspense
Fiction          Literature
Literature       Poetry
Literature       Victorian
Literature       Shakespearian
Literature       Modern American
Literature       19th Century American
```

Nested Queries

A *nested query* is a query containing other subqueries or queries contained within other queries. It is important to note that use of the term "nested" means that a query can be nested within a query, within a query, and so on — more or less *ad infinitum*, or as much as your patience and willingness to deal with complexity allows. Some databases use the `IN` set operator to nest one query within another, where one value is checked for membership in a list of values. The following query finds all authors, where each author has a publication, each publication has an edition, and each edition has a publisher:

```
SELECT * FROM AUTHOR WHERE AUTHOR_ID IN
 (SELECT AUTHOR_ID FROM PUBLICATION WHERE PUBLICATION_ID IN
  (SELECT PUBLICATION_ID FROM EDITION WHERE PUBLISHER_ID IN
     (SELECT PUBLISHER_ID FROM PUBLISHER)));

AUTHOR_ID NAME
---------- --------------------
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         6 William Shakespeare
```

Some databases also allow use of the EXISTS keyword. The EXISTS keyword returns a Boolean True result if the result is positive (it exists), or False otherwise. Where the IN operator includes expressions on both sides, the EXISTS operator has an expression only on the right side of the comparison. The next query finds all authors where the author exists as a foreign key AUTHOR_ID value in the PUBLISHER table:

```
SELECT * FROM AUTHOR WHERE EXISTS
 (SELECT AUTHOR_ID FROM PUBLICATION);

AUTHOR_ID NAME
---------- --------------------
         1 Orson Scott Card
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         5 Jerry Pournelle
         6 William Shakespeare
         7 Kurt Vonnegut
```

It is often also possible to pass a cross checking or correlation value into a subquery, such as in the following case using EXISTS. The query is a slightly more complex variation on the previous one where the AUTHOR_ID value, for each record found in the AUTHOR table, is passed to the subquery, and used by the subquery, to match with a PUBLISHER record:

A correlation between a calling query and a subquery is a link where variables in calling query and subquery are expected to contain the same values. The correlation link is usually a primary key to foreign key link — but it doesn't have to be.

```
SELECT * FROM AUTHOR WHERE EXISTS
 (SELECT AUTHOR_ID FROM PUBLICATION WHERE AUTHOR_ID = AUTHOR.AUTHOR_ID);

AUTHOR_ID NAME
---------- --------------------
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         6 William Shakespeare
         7 Kurt Vonnegut
```

Sometimes a correlation can be established between the calling query and subquery, using the IN operator as well as the EXISTS operator, although this is not as common. The next query is almost identical to the previous query, except that it uses the IN operator:

```
SELECT * FROM AUTHOR WHERE AUTHOR_ID IN
 (SELECT AUTHOR_ID FROM PUBLICATION WHERE AUTHOR_ID = AUTHOR.AUTHOR_ID);

AUTHOR_ID NAME
---------- --------------------
         2 James Blish
         3 Isaac Azimov
         4 Larry Niven
         6 William Shakespeare
         7 Kurt Vonnegut
```

Subqueries can produce single scalar values. In this query, the subquery passes the AUTHOR_ID value for the filtered author, back to the query on the PUBLICATION table—it passes a single AUTHOR_ID value (a single value is a scalar value):

```
SELECT AUTHOR_ID, TITLE FROM PUBLICATION WHERE AUTHOR_ID =
 (SELECT AUTHOR_ID FROM AUTHOR WHERE NAME = 'James Blish');


AUTHOR_ID TITLE
---------- -------------------------
         2 Cities in Flight
```

The DISTINCT clause is used to return only the unique records in a set of returned records.

Traditionally, the IN set membership operator is regarded as more efficient when testing against a list of literal values. The EXISTS set membership operator is regarded a being better than IN when checking against a subquery, in particular a correlated subquery. A correlated subquery creates a semi-join between the calling query and the subquery, by passing a key value from calling to subquery, allowing a join between calling query and subquery. This may not be true for all relational databases. A semi-join is called a semi-join because it effectively joins two tables but does not necessarily return any field values to the calling query, for return to the user, by the calling query.

Subqueries can also produce and be verified as multiple fields (this query returns no records):

```
SELECT * FROM COAUTHOR WHERE (COAUTHOR_ID, PUBLICATION_ID) IN
 (SELECT A.AUTHOR_ID, P.PUBLICATION_ID
  FROM AUTHOR A JOIN PUBLICATION P
    ON (P.AUTHOR_ID = A.AUTHOR_ID));
```

The ON clause in join syntax allows specification of two fields from different tables to join on. The ON clause is used when join fields in the two joined tables have different names, or in this case, when the complexity of the query, and use of aliases, forces explicit join field specification.

Composite Queries

Set merge operators can be used to combine two separate queries into a merged *composite query*. Both queries must have the same data types for each field, all in the same sequence. The term *set merge* implies a merge or sticking together of two separate sets of data. In the case of the following query, all records from two different tables are merged into a single set of records:

```
SELECT AUTHOR_ID AS ID, NAME FROM AUTHOR
UNION
SELECT PUBLISHER_ID AS ID, NAME FROM PUBLISHER;

ID NAME
---------- --------------------------------
         1 Orson Scott Card
         1 Overlook Press
         2 Ballantine Books
```

```
            2 James Blish
            3 Bantam Books
            3 Isaac Azimov
            4 Larry Niven
            4 Spectra
            5 Jerry Pournelle
            5 Oxford University Press
            6 Bt Bound
            6 William Shakespeare
            7 Kurt Vonnegut
            7 L P Books
            8 Del Rey Books
            9 Books on Tape
           10 HarperCollins Publishers
           11 Fawcett Books
           12 Berkley Publishing Group
           41 Gavin Powell
```

Changing Data in a Database

Changes to a database can be performed using the INSERT, UPDATE, and DELETE commands. Some database have variations on these commands, such as multiple table INSERT commands, MERGE commands to merge current and historical records, among others. These other types of commands are far too advanced for this book.

The INSERT command allows additions to tables in a database. Its syntax is generally as follows:

```
INSERT INTO table [ ( field [, ... ] ) ] VALUES ( expression [ , ... ]);
```

The UPDATE command has the following syntax. The WHERE clause allows targeting of one or more records:

```
UPDATE table SET field = expression [, ... ] [ WHERE ... ];
```

The DELETE command is similar in syntax to the UPDATE command. Again the WHERE clause allows targeting of one or more records for deletion from a table:

```
DELETE FROM table [ WHERE ... ];
```

Understanding Transactions

In a relational database, a *transaction* allows you to temporarily store changes. At a later point, you can choose to store the changes permanently using a COMMIT command. Or, you can completely remove all changes you have made since the last COMMIT command, by using a ROLLBACK command. It is that simple!

You can execute multiple database change commands, storing the changes to the database, localized for your connected session (no other connected users can see your changes until you commit them using the COMMIT command). If you were to execute a ROLLBACK command, rather than a COMMIT command, all new records would be removed from the database. For example, in the following script, the first two new authors are added to the AUTHOR table (they are committed), and the third author is not added (it is rolled back):

```
INSERT INTO AUTHOR(AUTHOR_ID, NAME) VALUES(100, 'Jim Jones');
INSERT INTO AUTHOR(AUTHOR_ID, NAME) VALUES(100, 'Jack Smith');
COMMIT;
INSERT INTO AUTHOR(AUTHOR_ID, NAME) VALUES(100, 'Jenny Brown');
ROLLBACK;
```

Blocks of commands can be executed within a single transaction, where all commands can be committed at once (by a single `COMMIT` command), or removed from the database at once (by a single `ROLLBACK` command). The following script introduces the concept of a block of code in a database. Blocks are used to compartmentalize sections of code, not only for the purposes of transaction control (controlling how transactions are executed) but also to create blocks of independently executable code, such as for stored procedures.

```
BEGIN
 INSERT INTO AUTHOR(AUTHOR_ID, NAME) VALUES(100, 'Jim Jones');
 INSERT INTO AUTHOR(AUTHOR_ID, NAME) VALUES(100, 'Jack Smith');
 INSERT INTO AUTHOR(AUTHOR_ID, NAME) VALUES(100, 'Jenny Brown');
 COMMIT;
TRAP ERROR
 ROLLBACK;
END;
```

The preceding script includes an *error trap*. The error trap is active for all commands between the `BEGIN` and `END` commands. Any error occurring between the `BEGIN` and `END` commands reroutes execution to the `TRAP ERROR` section, executing the `ROLLBACK` command, instead of the `COMMIT` command. The error trap section aborts the entire block of code, aborting any changes made so far. Any of the `INSERT` commands can trigger the error trap condition, if an error occurs.

Changing Database Metadata

Database objects such as tables define how data is stored in a database; therefore, database objects are known as *metadata*, which is the data about the data. In general, all databases include `CREATE`, `ALTER`, and `DROP` commands for all object types, with some exceptions. Exceptions are usually related to the nature of the object type, which is, of course, beside the point for this book. The intention in this chapter is to keep everything simple because SQL is not the focus of this book. Relational database modeling is the focus of this book.

The following command creates the `AUTHOR` table:

```
CREATE TABLE AUTHOR(
       AUTHOR_ID INTEGER NULL,
       NAME VARCHAR(32) NULL);
```

Use the `ALTER TABLE` command to set the `AUTHOR_ID` field as non-nullable:

```
ALTER TABLE AUTHOR MODIFY(AUTHOR_ID NOT NULL);
```

Use the DROP TABLE and CREATE TABLE commands to recreate the AUTHOR table with two constraints:

```
DROP TABLE AUTHOR;
CREATE TABLE AUTHOR (
       AUTHOR_ID INTEGER PRIMARY KEY NOT NULL,
       NAME VARCHAR(32) UNIQUE NOT NULL);
```

This CREATE TABLE command creates the PUBLICATION table, including a foreign key (REFERENCES AUTHOR), pointing back to the primary key field on the AUTHOR table:

```
CREATE TABLE PUBLICATION(
       PUBLICATION_ID INTEGER PRIMARY KEY NOT NULL,
       AUTHOR_ID INTEGER REFERENCES AUTHOR NOT NULL,
       TITLE VARCHAR(64) UNIQUE NOT NULL);
```

A relational database should automatically create an index for a primary key and a unique key field. The reason is simple to understand. When adding a new record, a primary key or unique key must be checked for uniqueness. What better way to maintain unique key values than an index? However, foreign key fields by their very nature can be both NULL valued and duplicated in a child table. Duplicates can occur because, in its most simple form, a foreign key field is usually on the "many" side of a one-to-many relationship. In other words, there are potentially many different publications for each author. Many authors often write more than one book. Additionally, a foreign key value could potentially be NULL valued. An author does not necessarily have to have any publications, currently in print. Creating an index on a foreign key field is not automatically controlled by a relational database. If an index is required for a foreign key field, that index should be manually created:

```
CREATE INDEX XFK_PUBLICATION_AUTHOR ON PUBLICATION (AUTHOR_ID);
```

Indexes can also be altered and dropped using the ALTER INDEX and DROP INDEX commands, respectively.

In general, database metadata change commands are a lot more comprehensive than just creating, altering, and dropping simple tables and indexes. Then again, it is not necessary to bombard you with too much scripting in this book.

Try It Out Creating Tables and Constraints

Figure 5-1 shows the tables of the online bookstore database model. Create all tables, including primary and foreign key constraints:

1. It is important to create the table in the correct sequence because some tables depend on the existence of others.
2. The PUBLISHER, AUTHOR, and SUBJECT tables can be created first because they are at the top of dependency hierarchies.
3. Next, create the PUBLICATION table.
4. Then create the EDITION, REVIEW, and COAUTHOR tables.

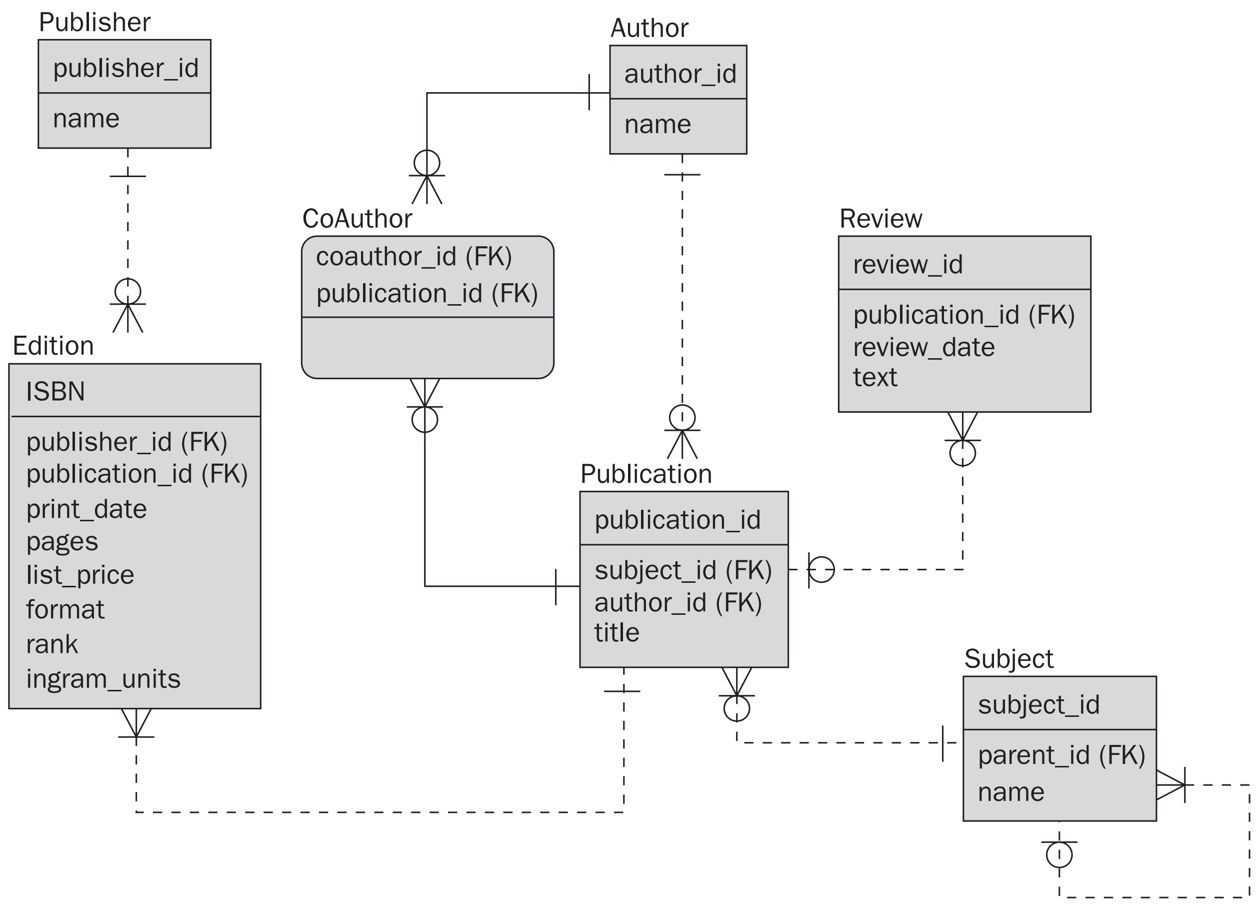

Figure 5-1: The online bookstore relational database model.

How It Works

Create tables as follows:

1. Create the `PUBLISHER`, `AUTHOR`, and `SUBJECT` tables:

```
CREATE TABLE PUBLISHER(
  PUBLISHER_ID INTEGER PRIMARY KEY NOT NULL,
  NAME VARCHAR(32) UNIQUE NOT NULL);

CREATE TABLE AUTHOR(
  AUTHOR_ID INTEGER PRIMARY KEY NOT NULL,
  NAME VARCHAR(32) UNIQUE NOT NULL);

CREATE TABLE SUBJECT(
  SUBJECT_ID INTEGER PRIMARY KEY NOT NULL,
  PARENT_ID INTEGER REFERENCES SUBJECT NULL,
  NAME VARCHAR(32) UNIQUE NOT NULL);
```

2. Create the `PUBLICATION` table (indexes can be created on foreign key fields):

```
CREATE TABLE PUBLICATION(
  PUBLICATION_ID INTEGER PRIMARY KEY NOT NULL,
  SUBJECT_ID INTEGER REFERENCES SUBJECT NOT NULL,
  AUTHOR_ID INTEGER REFERENCES AUTHOR NOT NULL,
  TITLE VARCHAR(64) UNIQUE NOT NULL);

CREATE INDEX XFK_P_AUTHOR ON PUBLICATION(AUTHOR_ID);
CREATE INDEX XFK_P_PUBLISHER ON PUBLICATION(SUBJECT_ID);
```

3. Create the `EDITION`, `REVIEW`, and `COAUTHOR` tables (indexes can be created on foreign key fields):

```
CREATE TABLE EDITION(
  ISBN INTEGER PRIMARY KEY NOT NULL,
  PUBLISHER_ID INTEGER REFERENCES PUBLISHER NOT NULL,
  PUBLICATION_ID INTEGER REFERENCES PUBLICATION NOT NULL,
  PRINT_DATE DATE NULL,
  PAGES INTEGER NULL,
  LIST_PRICE INTEGER NULL,
  FORMAT VARCHAR(32) NULL,
  RANK INTEGER NULL,
  INGRAM_UNITS INTEGER NULL);

CREATE INDEX XFK_E_PUBLICATION ON EDITION(PUBLICATION_ID);
CREATE INDEX XFK_E_PUBLISHER ON EDITION(PUBLISHER_ID);

CREATE TABLE REVIEW(
  REVIEW_ID INTEGER PRIMARY KEY NOT NULL,
  PUBLICATION_ID INTEGER REFERENCES PUBLICATION NOT NULL,
  REVIEW_DATE DATE NOT NULL,
  TEXT VARCHAR(4000) NULL);

CREATE TABLE COAUTHOR(
  COAUTHOR_ID INTEGER REFERENCES AUTHOR NOT NULL,
  PUBLICATION_ID INTEGER REFERENCES PUBLICATION NOT NULL);
  CONSTRAINT PRIMARY KEY (COAUTHOR_ID, PUBLICATION_ID);

CREATE INDEX XFK_CA_PUBLICATION ON COAUTHOR(COAUTHOR_ID);
CREATE INDEX XFK_CA_AUTHOR ON COAUTHOR(PUBLICATION_ID);
```

Summary

In this chapter, you learned about:

- ❑ What SQL is, why it is used, and how and why it originated
- ❑ SQL is a reporting language for relational databases
- ❑ SQL is primarily designed to retrieve sets or groups of related records
- ❑ SQL was not originally intended for retrieving unique records but does this fairly well in modern relational databases

- There are many different types of queries used for extracting and presenting information to the user in different ways
- There are three specific commands (`INSERT`, `UPDATE`, and `DELETE`) used for changing records in tables
- Tables and indexes can themselves be changed using table and index database metadata changing commands
- There are some simple ways of building better written, more maintainable, and faster performing SQL code commands

Above all, this chapter has shown how the relational database model is used from an application perspective. There is little point in understanding something such as relational database modeling without seeing it applied in some way, no matter how simple.

The next chapter returns to the topic of relational database modeling by presenting some advanced relational database modeling techniques.

Exercises

Use the ERD in Figure 5-1 to help you answer these questions.

1. Find all records and fields in the `EDITION` table.
2. Find all ISBN values in the `EDITION` table, for all `FORMAT='Hardcover '`books.
3. Do the same query, but sort records by `LIST_PRICE` contained within `FORMAT`.
4. Which of the two expressions 3 + 4 * 5, and (3 + 4) * 5 yields the greater value?
5. Find the sum of all `LIST_PRICE` values in the `EDITION` table, for each publisher.
6. Join the `SUBJECT` and `PUBLICATION` tables on the `SUBJECT_ID` field, as an intersection.
7. Find the intersection of subjects and publications, where subjects do not necessarily have to have publications.
8. Find subjects with publications, using a semi-join, in two different ways.