Chapter 4

Selecting Data from the Database

In This Chapter

- SELECT Overview and Syntax
- Choosing Columns: The SELECT Clause
- Specifying Tables: The FROM Clause
- Selecting Rows: The WHERE Clause

SELECT Overview and Syntax

In many ways, the SELECT statement is the real heart of SQL. It lets you find and view your data in a variety of ways. You use it to answer questions based on your data: how many, where, what kind of, even what if. Once you become comfortable with its sometimes dauntingly complex syntax, you'll be amazed at what the SELECT statement can do.

Because SELECT is so important, five chapters focus on it:

- This chapter begins with the bare bones: the SELECT, FROM, and WHERE clauses, search conditions, and expressions.
- Chapter 5 delves into some SELECT refinements: ORDER BY, the DISTINCT keyword, and aggregates.
- Chapter 6 covers the GROUP BY clause, the HAVING clause, and making reports from grouped data. Chapter 6 also summarizes the issues regarding null values in database management.
- Chapter 7 introduces multiple-table queries with a comprehensive discussion of joining tables.
- Chapter 8 moves on to **nested queries**, also known as **subqueries**.

Queries in this chapter use single tables so that you can focus on manipulating the syntax in a simple environment. Following is an example of a SELECT query—don't worry about the syntax yet:

SQL

```
select address
from publishers
where pub_id = '0877'
address
=======================================
2 2nd Ave.
[1 row]
```

Basic SELECT Syntax

Discovering the structure of the SELECT statement begins with this skeleton:

- The SELECT clause identifies the *columns* you want to retrieve.
- The FROM clause specifies the *tables* those columns are in.
- The WHERE clause qualifies the *rows*—it chooses the ones you want to see.

SYNTAX

```
SELECT select_list
FROM table_list
WHERE search_conditions
```

Select_list and Search_condition Expressions Both the SELECT and WHERE clauses (in the select_list or search_conditions) can include

- Plain column names (`price`)
- Column names combined with other elements, such as calculations (`price * 1.085`)
- Constants (character strings or display headings)

Collectively, these are expressions. Because the column name expression is the simplest case, examples often start there and then go on to a more complex expression. This does not mean that a column name is not an expression—it's just the place to start looking at expressions. Syntax that includes "expression" or "expr" or "char_expr" means that you can use a column name or a more complex expression.

pub_id	name	address	city	state
0736	New Age Books	1 1st St.	Boston	MA
0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA

Figure 4.1 Locating a Specific Piece of Data in a Table

Combining SELECT, FROM, and WHERE Artful combinations of the SELECT, FROM, and WHERE clauses produce meaningful answers to your questions and keep you from drowning in a sea of data. Think of the SELECT and WHERE clauses as horizontal and vertical axes on a matrix. (Figure 4.1 illustrates the query you saw at the beginning of the chapter.) The data you get from the SELECT statement is at the intersection of the SELECT (column) and WHERE (row) clauses.

Let's look at a SELECT statement with another `bookbiz` table, `authors`. The `authors` table stores information about authors: ID numbers, names, addresses, and phone numbers. If you want to know just the names of authors who live in California (not their addresses and phone numbers), use the SELECT clause and the WHERE clause to limit the data that the SELECT statement returns.

Here's a query that uses the SELECT clause's select_list to limit the *columns* you see. It lists just the names for the authors, ignoring their ID numbers, addresses, and phone numbers.

SQL

```
select au_lname, au_fname
from authors
au_lname                                 au_fname
================================== ====================
Bennet                                   Abraham
Green                                    Marjorie
Carson                                   Cheryl
Ringer                                   Albert
Ringer                                   Anne
DeFrance                                 Michel
Panteley                                 Sylvia
```

```
McBadden                                 Heather
Stringer                                 Dirk
Straight                                 Dick
Karsen                                   Livia
MacFeather                               Stearns
Dull                                     Ann
Yokomoto                                 Akiko
O'Leary                                  Michael
Gringlesby                               Burt
Greene                                   Morningstar
White                                    Johnson
del Castillo                             Innes
Hunter                                   Sheryl
Locksley                                 Chastity
Blotchet-Halls                           Reginald
Smith                                    Meander
[23 rows]
```

This display still doesn't provide exactly what you want because it lists all authors regardless of the state they live in. You need to refine the data retrieval statement further with the WHERE clause.

SQL

```
select au_lname, au_fname
from authors
where state = 'CA'
```

```
au_lname                                 au_fname
================================ ====================
Bennet                                   Abraham
Green                                    Marjorie
Carson                                   Cheryl
McBadden                                 Heather
Stringer                                 Dirk
Straight                                 Dick
Karsen                                   Livia
MacFeather                               Stearns
Dull                                     Ann
Yokomoto                                 Akiko
O'Leary                                  Michael
Gringlesby                               Burt
```

```
White                          Johnson
Hunter                         Sheryl
Locksley                       Chastity
[15 rows]
```

Now you're looking at just the names of the 15 authors having a California address. The rows for the eight authors living elsewhere are not included in the display.

Full SELECT Syntax

In practice, SELECT syntax can be either simpler or more complex than the example just shown. It can be simpler in that the SELECT and (in most systems) FROM clauses are the only required ones in a SELECT statement. The WHERE clause (and all other clauses) are optional. On the other hand, the full syntax of the SELECT statement includes all of the following phrases and keywords:

SYNTAX

```
SELECT [ALL | DISTINCT] select_list
  FROM table/view_list
  [WHERE search_conditions]
  [GROUP BY group_by_list ]
   [HAVING search_conditions]
  [ORDER BY order_by_list ]
```

SELECT Statement Clause Order Although SQL is a free-form language, you do have to keep the clauses in a SELECT statement in syntactical order (for example, a GROUP BY clause must come before an ORDER BY clause). Otherwise, you'll get syntax errors.

Naming Conventions You may need to qualify the names of database objects (according to the customs of your SQL dialect) if there is any ambiguity about which object you mean. In this database, there are several columns called `title_id` (in the `titles` table, the `titleauthors` table, and the `titleview` view, among others—see Figure 2.13). When you are working with multiple tables, you may have to specify which `title_id` column you're talking about by including the table or view name, usually separated from the column name by a period (`titles.title_id`). If the system allows multiple tables with the same name, add the owner name (`mary.titles.title_id` or `dba.titles.title_id`)—some possible combinations appear in Figure 4.2.

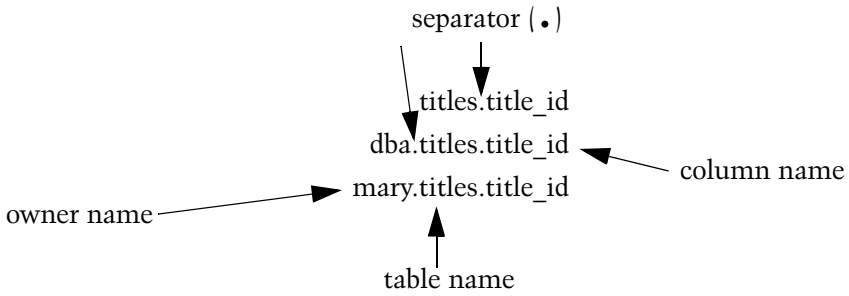

Figure 4.2 Qualifying Columns

You may also see larger elements, such as database and server names, used this way, but that is less common.

The examples in this chapter involve queries on a single table, so qualification is not an important issue here. Qualifiers are also omitted in most books, articles, and reference manuals on SQL because the short forms make SELECT statements more readable. However, it's never wrong to include them.

Choosing Columns: The SELECT Clause

The first clause of the SELECT statement—the one that begins with the keyword SELECT—is required in all SELECT statements. The keywords ALL and DISTINCT, which specify whether duplicate rows are to be included in the results, are optional. DISTINCT and ALL are discussed in the next chapter.

The select_list specifies the columns you want to see in the results. It can consist of these items individually or together:

- An asterisk, shorthand for all the columns in the table, displayed in CREATE TABLE order
- One or more column names, in any order
- One or more character constants (such as "Total") used as display headings or text embedded in the results
- One or more SQL functions (AVG) and arithmetic operators, generally used with columns (`price * 1.085`)

You can mix these elements freely. As mentioned earlier, columns, constants, functions, and combinations of these elements, with or without arith-

metic operators, are collectively called expressions. Separate with a comma each element in a SELECT list from the following element.

Choosing All Columns: SELECT *

The asterisk (*) has a special meaning in the select_list. It stands for *all the column names* in *all the tables* in the table list. The columns are displayed in the order in which they appeared in the CREATE TABLE statement(s). Most people read a SELECT * statement as "select star." Use it when you want to see all the columns in a table.

The general syntax for selecting all the columns in a table is this:

SYNTAX

```
SELECT *
FROM table/view_list
```

Because SELECT * finds all the columns currently in a table, changes in the structure of a table (adding, removing, or renaming columns) automatically modify the results of a SELECT *. Listing the columns individually gives you more precise control over the results, but SELECT * saves typing (and the frustration of typographical errors). SELECT * is most useful for tables with few columns because displays of many columns can be confusing. It also comes in handy when you want to get a quick look at a table's structure (what columns it has and in what order they appear).

The following statement retrieves all columns in the `publishers` table and displays them in the order in which they were defined when the `publishers` table was created. Because no WHERE clause is included, this statement retrieves every row.

SQL

```
select *
from publishers
pub_id pub_name               address       city         state
====== ====================== ============= ============ =====
0736   New Age Books          1 1st St.     Boston       MA
0877   Binnet & Hardley       2 2nd Ave.    Washington   DC
1389   Algodata Infosystems   3 3rd Dr.     Berkeley     CA
[3 rows]
```

You get exactly the same results by listing all the column names in the table in CREATE TABLE order after the SELECT keyword:

SQL

```
select pub_id, pub_name, address, city, state
from publishers
```

```
pub_id pub_name               address           city         state
====== ====================== ================= ============ =====
0736   New Age Books          1 1st St.         Boston       MA
0877   Binnet & Hardley       2 2nd Ave.        Washington   DC
1389   Algodata Infosystems   3 3rd Dr.         Berkeley     CA
[3 rows]
```

Choosing Specific Columns

To select a subset of the columns in a table, as some of the previous examples have demonstrated, simply list the columns you want to see in the SELECT list:

SYNTAX

```
SELECT column_name[, column_name]...
FROM table_list
```

Separate each column name from the following column name with a comma.

Rearranging Result Columns The order in which columns appear in a display is completely up to you: Use the SELECT list to order them in any way that makes sense.

Following are two examples. Both of them find and display the publisher names and identification numbers from all three of the rows in the `publishers` table. The first one prints `pub_id` first, followed by `pub_name`. The second reverses that order. The information is exactly the same; only the display format changes.

SQL

```
select pub_id, pub_name
from publishers
```

```
pub_id pub_name
====== ========================================
0736   New Age Books
0877   Binnet & Hardley
1389   Algodata Infosystems
[3 rows]


select pub_name, pub_id
from publishers
pub_name                                 pub_id
======================================== ======
New Age Books                            0736
Binnet & Hardley                         0877
Algodata Infosystems                     1389
[3 rows]
```

More Than Column Names

The SELECT statements you've seen so far show exactly what's stored in a table. This is useful, but often not useful enough. SQL lets you add to and manipulate these results to make them easier to read or to do "what if" queries. This means you can use strings of characters, mathematical calculations, and functions provided by your system in the SELECT list, with or without column names.

Display Label Conventions When the results of a query are displayed, each column has a default heading—its name as defined in the database. Column names in databases are often cryptic (so they'll be easy to type) or have no meaning to users unfamiliar with departmental acronyms, nicknames, or project jargon.

You can solve this problem by specifying **display labels** (sometimes called **column aliases** or **headings**) to make query results easier to read and understand. To get the heading you want, simply type `column_name column_heading`, or `column_name as column_heading` in the SELECT clause in place of the column name. For example, to change the `pub_name` column heading to `Publisher`, try one of the following statements:

SQL

```
select pub_name Publisher, pub_id
from publishers
```

SQL

```
select pub_name as Publisher, pub_id
from publishers
```

Some systems also allow this syntax:

Adaptive Server Anywhere

```
select Publisher = pub_name, pub_id
from publishers
```

The results of all three methods show a new column heading:

Results

```
Publisher                              pub_id
====================================== ======
New Age Books                          0736
Binnet & Hardley                       0877
Algodata Infosystems                   1389
[3 rows]
```

TIP

For consistency, pick one of these formats and stick with it. Many users prefer the AS convention—it has the advantage of being simple and unambiguous.

SQL VARIANTS

Check to see how your system handles column headings that are longer than defined column size. For example, what happens when you change the `pub_id` column heading to a string such as "Identification #"? Does your system increase the display size of the column or shorten the new column heading to the size of the column data? The following queries show two possibilities:

Adaptive Server Anywhere

```
select pub_name as Publisher, pub_id as Identification#
from publishers
Publisher                              Identification#
====================================== ===============
New Age Books                          0736
Binnet & Hardley                       0877
Algodata Infosystems                   1389
[3 rows]
```

Oracle

```
PUBLISHER                              IDEN
-------------------------------------- ----
New Age Books                          0736
Binnet + Hardley                       0877
Algodata Infosystems                   1389
```

(Oracle SQL Plus shows display headings as uppercase by default. Enclose the heading text in double quotes to preserve case.) If you use a smaller heading, however, SQL doesn't shrink the display size to less than its datatype-defined size.

Display Label Limitations Most SQL dialects that allow you to add display labels have some restrictions. Check your reference guide for details on

- Quotes (single and double)
- Embedded spaces
- Special characters

For example, Adaptive Server Anywhere allows single and double quotes around column headings. The quotes are not needed unless there is an embedded space in the column heading.

Adaptive Server Anywhere

```
select pub_name as 'Publisher #', pub_id as "Identification #"
from publishers;
Publisher #                            Identification #
====================================== ===============
New Age Books                          0736
Binnet & Hardley                       0877
Algodata Infosystems                   1389
```

However, other systems are not as forgiving.

SQL VARIANTS

Oracle SQL Plus rejects single quotes around column headings.

```
Oracle
SQL> select pub_name as Publisher, pub_id as 'Identification #'
  2   from publishers;
ERROR at line 1:
ORA-00923: FROM keyword not found where expected
```

Change the single quotes to double, and the query works fine. In addition, the original case of the heading is preserved.

```
Oracle
SQL> select pub_name as "Publisher #", pub_id as "Identification #"
  2  from publishers;
Publisher #                              Iden
---------------------------------------- ----
New Age Books                            0736
Binnet & Hardley                         0877
Algodata Infosystems                     1389
```

Other implementations object to spaces or special characters.

```
Informix
select pub_name as Publisher, pub_id as Identification#
from publishers
SQL Error. An illegal character has been found.
```

The illegal character is the pound sign (#). Quotation marks don't help in this case.

Character Strings in Query Results Sometimes a little text can make query results easier to understand. That's where **strings** (of characters) come in handy.

Let's say you want a listing of publishers with something like "The publisher's name is" in front of each item. All you have to do is insert the string in

the correct position in the SELECT list. Be sure to enclose the entire string in quotes (single quotes are standard, but some dialects allow both single and double quotes) so your system can tell it's not a column name and separate it with commas from other elements in the select_list .

Follow your system's rules for protecting embedded apostrophes and quotes, if any appear in the string. In most cases, double single quotes do the trick and prevent the apostrophe from being interpreted as a close quote.

SQL

```
select 'The publisher''s name is', pub_name as Publisher
from publishers

'The publisher''s name is' Publisher
========================== ============================
The publisher's name is    New Age Books
The publisher's name is    Binnet & Hardley
The publisher's name is    Algodata Infosystems
[3 rows]
```

The constants create a new column in the display only—what you see doesn't affect anything that's physically in the database.

Combining Columns, Display Headings, and Text You can combine columns, display headings, and text in a SELECT list.

Remember to put quotes around the text but not around the column names. You need quotes around display headings only if they contain spaces (or other special characters). Figure 4.3 illustrates mixing several techniques.

Computations with Constants The SELECT list is the place where you indicate computations you want to perform on numeric data or constants.

Here are the available **arithmetic operators**:

Symbol	Operation
+	addition
–	subtraction
/	division
*	multiplication

Adaptive Server Anywhere:

```
select 'The name for publisher #' as 'Long Text',
    pub_id as Num, 'is' as Text, pub_name
from publishers
```

Display label with embedded space—needs quotes

Display heading with no embedded space—no quotes

Text—needs quotes

Column name—no quotes

```
Long Text                  Num  Text pub_name
========================== ==== ==== ====================
The name for publisher # 0736 is   New Age Books
The name for publisher # 0877 is   Binnet & Hardley
The name for publisher # 1389 is   Algodata Infosystems
```

Figure 4.3 Column Names, Text, and Display Headings

The arithmetic operators—addition, subtraction, division, and multiplication—can be used on any numeric column.

Certain arithmetic operations can also be performed on date columns, if your system provides date functions.

You can use all of these operators in the SELECT list with column names and numeric constants in any combination. For example, to see what a projected sales increase of 100 percent for all the books in the `titles` table looks like, type this:

SQL

```
select title_id, ytd_sales, ytd_sales * 2
from titles
```

```
title_id   ytd_sales titles.ytd_sales*2
======== =========== ==================
PC8888          4095               8190
BU1032          4095               8190
PS7777          3336               6672
PS3333          4072               8144
BU1111          3876               7752
MC2222          2032               4064
TC7777          4095               8190
TC4203         15096              30192
PC1035          8780              17560
BU2075         18722              37444
PS2091          2045               4090
```

```
PS2106          111              222
MC3021        22246            44492
TC3218          375              750
MC3026        (NULL)           (NULL)
BU7832         4095             8190
PS1372          375              750
PC9999        (NULL)           (NULL)
[18 rows]
```

Notice the null values in the `ytd_sales` column and the computed column. When you perform any arithmetic operation on a null value, the result is NULL.

SQL VARIANTS

The null value may show up as a blank, as the word NULL, or as some other symbol determined by the system. Check your vendor's documentation: You may have a way to change the default NULL display.

Oracle

```
SQL> select title_id, ytd_sales, ytd_sales * 2
  2  from titles
  3  where title_id > 'M' and title_id < 'PS';
TITLE_ YTD_SALES YTD_SALES*2
------ --------- -----------
MC2222      2032        4064
MC3021     22246       44492
MC3026
PC1035      8780       17560
PC8888      4095        8190
PC9999
6 rows selected.
```

Computed Column Display Headings You can give the computed column a heading (for example, `Projected_Sales`):

SQL

```
select title_id, ytd_sales, ytd_sales * 2 as Projected_Sales
from titles
```

For a fancier display, try adding character strings such as "Current sales =" and "Projected sales are" to the SELECT statement.

Sometimes, as in the previous example, you'll want both the original data and the computed data in your results. But you don't have to include the column on which the computation takes place in the SELECT list. To see just the computed values, type this:

SQL

```
select title_id, ytd_sales * 2
from titles

title_id titles.ytd_sales*2
======== ==================
PC8888                 8190
BU1032                 8190
PS7777                 6672
PS3333                 8144
BU1111                 7752
MC2222                 4064
TC7777                 8190
TC4203                30192
PC1035                17560
BU2075                37444
PS2091                 4090
PS2106                  222
MC3021                44492
TC3218                  750
MC3026               (NULL)
BU7832                 8190
PS1372                  750
PC9999               (NULL)
[18 rows]
```

Computations with Column Names You can also use arithmetic operators for computations on the data in two or more columns, with no constants involved. Here's an example:

SQL

```
select title_id, ytd_sales * price
from titles
```

```
title_id    titles.ytd_sales*titles.price
======== ================================
PC8888                          81900.00
BU1032                          81859.05
PS7777                          26654.64
PS3333                          81399.28
BU1111                          46318.20
MC2222                          40619.68
TC7777                          61384.05
TC4203                         180397.20
PC1035                         201501.00
BU2075                          55978.78
PS2091                          22392.75
PS2106                            777.00
MC3021                          66515.54
TC3218                           7856.25
MC3026                            (NULL)
BU7832                          81859.05
PS1372                           8096.25
PC9999                            (NULL)
[18 rows]
```

Finally, you can compute new values on the basis of columns from more than one table. (Chapter 7, on joining, and Chapter 8, on subqueries, give information on how to work with multiple-table queries, so check them for details.)

Arithmetic Operator Precedence When there is more than one arithmetic operator in an expression, the system follows rules that determine the order in which the operations are carried out (Figure 4.4). According to commonly used precedence rules, multiplication and division are calculated first, followed by subtraction and addition. When more than one arithmetic operator in an expression has the same level of precedence, the order of execution is left to right. Expressions within parentheses take precedence over all other operations.

Here's an example: The following SELECT statement subtracts the advance on each book from the gross revenues realized on its sales (`price` multiplied by

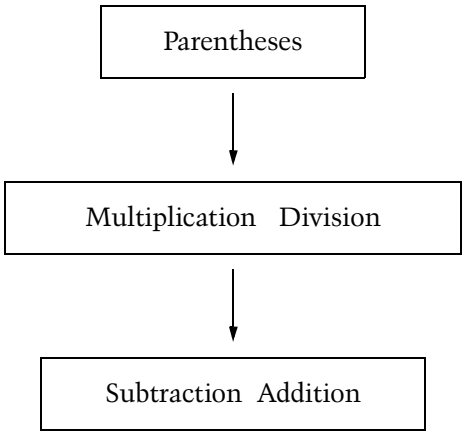

Figure 4.4 Precedence Hierarchy for Arithmetic Operators

ytd_sales). The product of ytd_sales and price is calculated first because the operator is multiplication.

SQL

```
select title_id, ytd_sales * price - advance
from titles
```

To avoid misunderstandings, use parentheses. The following query has the same meaning and gives the same results as the previous one, but it is easier to understand:

SQL

```
select title_id, (ytd_sales * price) - advance
from titles
title_id titles.ytd_sales*titles.price
======== ============================
PC8888                       155800.00
BU1032                       117809.05
PS7777                        56014.64
PS3333                       120119.28
BU1111                        80078.20
MC2222                        60939.68
```

```
TC7777                          114809.05
TC4203                          327357.20
PC1035                          370101.00
BU2075                          233073.78
PS2091                           42612.75
PS2106                           -4113.00
MC3021                          273975.54
TC3218                            8356.25
MC3026                             (NULL)
BU7832                          117809.05
PS1372                            8596.25
PC9999                             (NULL)
[18 rows]
```

Another important use of parentheses is changing the order of execution: Calculations inside parentheses are handled first. If parentheses are nested (one set of parentheses inside another), the most deeply nested calculation has precedence. For example, the result and meaning of the query just shown can be changed if you use parentheses to force evaluation of the subtraction before the multiplication:

SQL

```
select title_id, ytd_sales * (price - advance)
from titles
title_id titles.ytd_sales*(titles.pric
======== ==============================
PC8888                     -32596200.00
BU1032                     -20352190.95
PS7777                     -13283985.36
PS3333                      -8021880.72
BU1111                     -19294921.80
MC2222                         60939.68
TC7777                     -32637190.95
TC4203                     -60052642.80
PC1035                     -61082899.00
BU2075                    -189317051.22
PS2091                      -4607487.25
PS2106                       -664113.00
MC3021                    -333401024.46
TC3218                      -2609643.75
```

```
MC3026                              (NULL)
BU7832                        -20352190.95
PS1372                         -2609403.75
PC9999                              (NULL)
[18 rows]
```

Specifying Tables: The FROM Clause

The **table list** names the table(s), the view(s), or both, that contain columns included in the SELECT list and in the WHERE clause. (Views are covered in Chapter 9—for now, just consider them a kind of table.) Separate table names in the table list with commas. The FROM syntax looks like this:

SYNTAX

```
SELECT select_list
FROM [qualifier.]{table_name | view_name} [alias]
  [, [qualifier.]{table_name | view_name} [alias] ]...
```

The full naming syntax for tables and views, with qualifying database and owner names, is always permitted in the table list. It's necessary, however, only when there might be some confusion about the name.

Using Table Aliases

In many SQL dialects, you can give table names **aliases** to save typing. Assign an alias in the table list by giving the alias after the table name, like this:

SQL

```
select p.pub_id, p.pub_name
from publishers p
```

The **p** in front of each of the column names in the SELECT list acts as a substitute for the full table name (`publishers`). This query is equivalent to

SQL

```
select publishers.pub_id, publishers.pub_name
from publishers
```

You can't combine the two naming conventions. Once you assign an alias, you must use the alias or no qualifier—alternately using the alias and the full table name in a given query isn't allowed because the alias actually substitutes for the table or view name during the query. In effect, the table name does not exist. Here's an example of assigning an alias but also using the full name:

SQL

```
select publishers.pub_id, p.pub_name
from publishers p
Correlation name 'publishers' not found.
```

Since only one table is involved in these queries, there is no ambiguity about which `pub_id` column you're referencing, so using the table name—either its alias or its full name—as a qualifier is optional. Aliases are really useful only in multiple-table queries where you need to qualify columns from different tables. You'll see examples of their use in Chapters 7 and 8.

Skipping FROM

Some systems allow you to write queries *without* a FROM clause. For example, a query for the current date and time (information not stored in a table) may work fine, like this:

Adaptive Server Anywhere

```
select current date
current date
============
Mar 01 2000 12:00am
[1 row]
```

SQL VARIANTS Other systems don't allow you to skip FROM. When you retrieve nontable information, you must use FROM with a dummy table that you create or the system supplies (for Oracle, `dual`).

```
Oracle
SQL> select sysdate
  2  from dual;
SYSDATE
--------------------
Mar 01 2000 12:00 AM
```

Selecting Rows: The WHERE Clause

The WHERE clause is the part of the SELECT statement that specifies the search conditions. These conditions determine exactly which rows are retrieved. The general format is this:

SYNTAX

```
SELECT select_list
FROM table_list
WHERE search_conditions
```

When you run a SELECT statement with a WHERE clause, your system searches for the rows in the table that meet your conditions (also called **qualifications**).

SQL provides a variety of operators and keywords for expressing the search conditions, including these:

- **Comparison operators** (=, <, >, and so on)
```
select title
from titles
where advance * 2 > ytd_sales * price
```
- Combinations or logical negations of conditions (AND, OR, NOT)
```
select title
from titles
where advance < 5000 or ytd_sales > 2000
```
- Ranges (BETWEEN and NOT BETWEEN)
```
select title
from titles
where ytd_sales between 4095 and 12000
```

- Lists (IN, NOT IN)
```
select pub_name
from publishers
where state in ('CA', 'IN', 'MD')
```
- Unknown values (IS NULL and IS NOT NULL)
```
select title
from titles
where advance is null
```
- Character matches (LIKE and NOT LIKE)
```
select au_lname
from authors
where phone not like '415%'
```

Each of these keywords and operators is explained and illustrated in this chapter. In addition, the WHERE clause can include join conditions (see Chapter 7) and subqueries (see Chapter 8).

Comparison Operators

You often want to look at values in relation to one another to find out which is "larger" or "smaller" or "lower" in the alphabet sort or "equal" to some other database value or to a constant. SQL provides a set of comparison operators for these purposes. In most dialects, the comparison operators are these:

Operator	Meaning
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
< >	not equal to

The operators are used in the syntax:

SYNTAX

```
WHERE expression comparison_operator expression
```

An expression can be a plain column name or something more complex—a character string, a function or calculation (usually involving a column name),

or any combination of these elements connected by arithmetic operators. When evaluated, an expression produces a single value per row.

In contexts other than SQL, the comparison operators are usually used with numeric values. In SQL, they are also used with *char* and *varchar* data (< means earlier in the dictionary order and > means later) and with dates (< means earlier in chronological order and > means later). When you use character and date values in a SQL statement, be sure to put quotes around them.

TIP

The order in which uppercase and lowercase characters and special characters are evaluated depends on the character-sorting sequence you are using, imposed by your database system or by the machine you are using. (There are more details on sort order in "Character Sets and Sort Orders"). Check your system to see how it handles trailing blanks in comparisons. Is "Dirk" considered the same as "Dirk "?

Comparing Numbers The following SELECT statements and their results should give you a good sense of how the comparison operators are used. The first query finds the books that cost more than $25.00.

SQL

```
select title, price
from titles
where price > $25.00
```

```
title                                                              price
================================================================== =====
Secrets of Silicon Valley                                          40.00
The Busy Executive's Database Guide                                29.99
Prolonged Data Deprivation: Four Case Studies                      29.99
Silicon Valley Gastronomic Treats                                  29.99
Sushi, Anyone?                                                     29.99
But Is It User Friendly?                                           42.95
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean    40.95
Straight Talk About Computers                                      29.99
Computer Phobic and Non-Phobic Individuals: Behavior Variations    41.59
[9 rows]
```

SQL VARIANTS Check your system to see if it allows dollar signs with money values. Most do not. Transact-SQL is an exception, and so is Adaptive Server Anywhere.

Comparing Character Values The next SELECT statement finds the authors whose last names follow McBadden in the alphabet. Notice the name is in single quotes. (Some systems allow both single and double quotes around character and date constants in the WHERE clause, but most allow single quotes only.)

SQL

```
select au_lname, au_fname
from authors
where au_lname >'McBadden'
```

```
au_lname                                 au_fname
======================================== ===============
O'Leary                                  Michael
Panteley                                 Sylvia
Ringer                                   Albert
Ringer                                   Anne
Smith                                    Meander
Straight                                 Dick
Stringer                                 Dirk
White                                    Johnson
Yokomoto                                 Akiko
[9 rows]
```

(Your results may differ, depending on the sort order your system uses. See Chapter 5 for more on this issue.)

Comparing Imaginary Values The next query displays hypothetical information—it calculates double the price of all books for which advances over $10,000 were paid and displays the title identification numbers and calculated prices:

SQL

```
select title_id, price * 2
from titles
where advance > 10000
title_id titles.price*2
======== ==============
BU2075             25.98
MC3021             25.98
[2 rows]
```

Finding Values Not Equal to Some Value Following is a query that finds the telephone numbers of authors who don't live in California, using the not equal comparison operator (in some SQL dialects, you can use != as the not equal operator).

SQL

```
select au_id, phone
from authors
where state <> 'CA'
au_id       phone
=========== ============
998-72-3567 801 826-0752
899-46-2035 801 826-0752
722-51-5454 219 547-9982
807-91-6654 301 946-8853
527-72-3246 615 297-2723
712-45-1867 615 996-8275
648-92-1872 503 745-6402
341-22-1782 913 843-0462
[8 rows]
```

Connecting Conditions with Logical Operators

Use the **logical operators** AND, OR, and NOT when you're dealing with more than one condition in a WHERE clause. The logical operators are also called **Boolean operators**.

AND AND joins two or more conditions and returns results only when all of the conditions are true. For example, the following query will find only the rows in which the author's last name is Ringer and the author's first name is Anne. It will not find the row for Albert Ringer.

SQL

```
select au_id, au_lname, au_fname
from authors
where au_lname = 'Ringer'
  and au_fname = 'Anne'
```

```
au_id       au_lname                            au_fname
=========== =================================== ========
899-46-2035 Ringer                              Anne
[1 row]
```

The next example finds business books with a price higher than $20.00 and for which an advance of less than $20,000 was paid:

SQL

```
select title, type, price, advance
from titles
where type = 'business'
  and price > 20.00
  and advance < 20000
```

```
title                                    type     price  advance
=======================================  ======== ====== ========
The Busy Executive's Database Guide      business 29.99   5000.00
Cooking with Computers: Surreptitious
  Balance Sheets                         business 21.95   5000.00
Straight Talk About Computers            business 29.99   5000.00
[3 rows]
```

OR OR also connects two or more conditions, but it returns results when any of the conditions is true. The following query searches for rows containing Anne or Ann in the `au_fname` column:

SQL

```
select au_id, au_lname, au_fname
from authors
where au_fname = 'Anne'
  or au_fname = 'Ann'
```

```
au_id       au_lname                          au_fname
=========== ================================= ==========
899-46-2035 Ringer                            Anne
427-17-2319 Dull                              Ann
[2 rows]
```

The following query searches for books with a price higher than $20.00 *or* an advance less than $5,000:

SQL

```
select title, type, price, advance
from titles
where price > $30.00
  or advance < $5000
```

```
title                                     type         price advance
========================================= ============ ===== =======
Secrets of Silicon Valley                 popular_comp 40.00 8000.00
Emotional Security: A New Algorithm       psychology   17.99 4000.00
Prolonged Data Deprivation: Four Case
   Studies                                psychology   29.99 2000.00
Silicon Valley Gastronomic Treats         mod_cook     29.99    0.00
Fifty Years in Buckingham Palace Kitchens trad_cook    21.95 4000.00
But Is It User Friendly?                  popular_comp 42.95 7000.00
Is Anger the Enemy?                       psychology   21.95 2275.00
Onions, Leeks, and Garlic: Cooking
   Secrets of the Mediterranean           trad_cook    40.95 7000.00
Computer Phobic and Non-Phobic
   Individuals: Behavior Variations       psychology   41.59 7000.00
[9 rows]
```

Semantic Issues with OR and AND One more example using OR will demonstrate a potential for confusion. Let's say you want to find all the business books, as well as any books with a price higher than $10 and any books with an advance less than $20,000. The English phrasing of this problem suggests

the use of the operator AND, but the logical meaning dictates the use of OR because you want to find all the books in all three categories, not just books that meet all three characteristics at once. Here's the SQL statement that finds what you're looking for:

SQL

```
select title, type, price, advance
from titles
where type = 'business'
  or price > $20.00
  or advance < $20000
```

```
title                                    type         price advance
======================================== ============ ===== =======
Secrets of Silicon Valley                popular_comp 40.00  8000.00
The Busy Executive's Database Guide      business     29.99  5000.00
Emotional Security: A New Algorithm      psychology   17.99  4000.00
Prolonged Data Deprivation:
  Four Case Studies                      psychology   29.99  2000.00
Cooking with Computers:
  Surreptitious Balance Sheets           business     21.95  5000.00
Silicon Valley Gastronomic Treats        mod_cook     29.99     0.00
Sushi, Anyone?                           trad_cook    29.99  8000.00
Fifty Years in Buckingham Palace Kitchens trad_cook   21.95  4000.00
But Is It User Friendly?                 popular_comp 42.95  7000.00
You Can Combat Computer Stress!          business     12.99 10125.00
Is Anger the Enemy?                      psychology   21.95  2275.00
Life Without Fear                        psychology   17.00  6000.00
The Gourmet Microwave                    mod_cook     12.99 15000.00
Onions, Leeks, and Garlic:
  Cooking Secrets of the Mediterranean   trad_cook    40.95  7000.00
Straight Talk About Computers            business     29.99  5000.00
Computer Phobic and Non-Phobic
   Individuals: Behavior Variations      psychology   41.59  7000.00
[16 rows]
```

Compare this query and its results to the earlier example that is identical except for the use of AND instead of OR.

NOT The logical operator NOT negates an expression. When you use it with comparison operators, put it before the expression rather than before the comparison operator. The following two queries are equivalent:

SQL

```
select au_lname, au_fname, state
from authors
where state <> 'CA'
```

SQL

```
select au_lname, au_fname, state
from authors
where not state = 'CA'
```

Here are the results:

Results

```
au_lname                          au_fname        state
================================= =============== =====
Ringer                            Albert          UT
Ringer                            Anne            UT
DeFrance                          Michel          IN
Panteley                          Sylvia          MD
Greene                            Morningstar     TN
del Castillo                      Innes           MI
Blotchet-Halls                    Reginald        OR
Smith                             Meander         KS
[8 rows]
```

Logical Operator Precedence Like the arithmetic operators, logical operators are handled according to precedence rules. When both kinds of operators occur in the same statement, arithmetic operators are handled before logical operators. When more than one logical operator is used in a statement, NOT is evaluated first, then AND, and finally OR. Figure 4.5 shows the hierarchy.

Some examples will clarify the situation. The following query finds all the business books in the `titles` table, no matter what their advances are, as well as all psychology books that have an advance greater than $5,500. The advance condition pertains to psychology books and not to business books because the AND is handled before the OR.

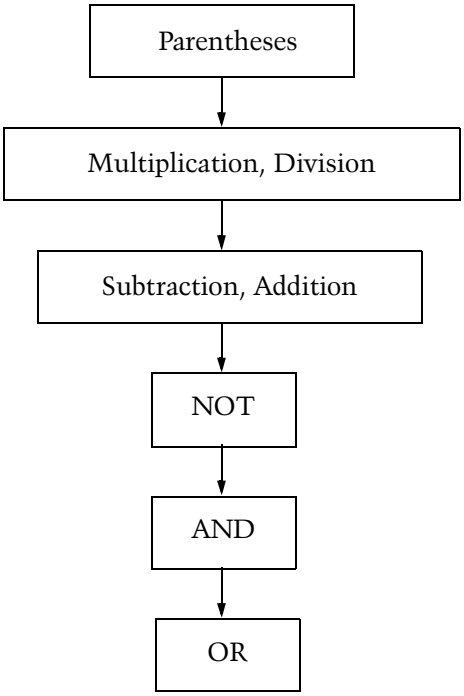

Figure 4.5 Precedence Hierarchy for Logical Operators

SQL

```
select title_id, type, advance
from titles
where type = 'business'
  or type = 'psychology'
  and advance > 5500
```

```
title_id type             advance
======== ============ ============
BU1032   business          5000.00
BU1111   business          5000.00
BU2075   business         10125.00
PS2106   psychology        6000.00
BU7832   business          5000.00
PS1372   psychology        7000.00
[6 rows]
```

The results include three business books with advances less than $5,500 because the query was evaluated according to the following precedence rules:

1. Find all psychology books with advances greater than $5,500.
2. Find all business books (never mind about advances).
3. Display both sets of rows in the results.

You can change the meaning of the previous query by adding parentheses to force evaluation of the OR first. With parentheses added, the query executes differently:

1. Find all business and psychology books.
2. Locate those that have advances over $5,500.
3. Display only the final subset.

SQL

```
select title_id, type, advance
from titles
where (type = 'business' or type = 'psychology')
  and advance > 5500
```

```
title_id type              advance
======== ============ ============
BU2075   business         10125.00
PS2106   psychology        6000.00
PS1372   psychology        7000.00
[3 rows]
```

The parentheses cause SQL to find all business and psychology books and, from among those, to find those with advances greater than $5,500.

Here's a query that includes arithmetic operators, comparison operators, and logical operators. It searches for books that are not bringing in enough money to offset their advances. Specifically, the query searches for any books with gross revenues (that is, `ytd_sales` times `price`) less than twice the advance paid to the author(s). The user who constructed this query has tacked on another condition: She wants to include in the results only books published before October 15, 2000, because those books have had long enough to establish a sales pattern. The last condition is connected with the logical operator

AND; according to the rules of precedence, it is evaluated after the arithmetic operations.

SQL

```
select title_id, type, price, advance, ytd_sales
from titles
where price * ytd_sales <  2 * advance
  and pubdate < '10/15/2000'
title_id type        price     advance     ytd_sales
======== =========== ========= =========== ===========
PS2106   psychology  17.00     6000.00     111
[1 row]
```

SQL VARIANTS

If you run this query on a system with a different date format, you may need to change the `pubdate` value to correspond to that format. For example, if your SQL engine expects dates to look like DD-MON-YYYY, you could write the query like this:

Oracle

```
SQL> select title_id, type, price, advance, ytd_sales
  2   from titles
  3   where price * ytd_sales <  2 * advance
  4  and pubdate < '21 OCT 2000';
TITLE_ TYPE             PRICE   ADVANCE YTD_SALES
------ ------------ --------- --------- ---------
PS2106 psychology          17      6000       111
```

Ranges (BETWEEN and NOT BETWEEN)

Another common search condition is a range. There are two different ways to specify ranges:

- With the comparison operators > and <
- With the keyword BETWEEN

Use BETWEEN to specify an **inclusive range**, in which you search for the lower value and the upper value as well as the values they bracket. For example, to find all the books with sales between (and including) 4,095 and 12,000, you could write this query:

SQL

```
select title_id, ytd_sales
from titles
where ytd_sales between 4095 and 12000
title_id   ytd_sales
======== ===========
PC8888          4095
BU1032          4095
TC7777          4095
PC1035          8780
BU7832          4095
[5 rows]
```

Notice that books with sales of 4,095 are included in the results. If there were any with sales of 12,000, they would be included too. In this way, the BETWEEN range is different from the greater-than/less-than (> <) range. The same query using the greater-than and less-than operators returns different results because the range is not inclusive:

SQL

```
select title_id, ytd_sales
from titles
where ytd_sales > 4095 and ytd_sales < 12000
title_id   ytd_sales
======== ===========
PC1035          8780
[1 row]
```

NOT BETWEEN The phrase NOT BETWEEN finds all the rows that are not inside the range. To find all the books with sales outside the range of 4,095 to 12,000, type this:

SQL

```
select title_id, ytd_sales
from titles
where ytd_sales not between 4095 and 12000
title_id   ytd_sales
======== ===========
PS7777          3336
PS3333          4072
BU1111          3876
MC2222          2032
TC4203         15096
BU2075         18722
PS2091          2045
PS2106           111
MC3021         22246
TC3218           375
PS1372           375
[11 rows]
```

You can get the same results with comparison operators, but notice in this query that you use OR between the two `ytd_sales` comparisons rather than AND.

SQL

```
select title_id, ytd_sales
from titles
where ytd_sales < 4095 or ytd_sales > 12000
title_id   ytd_sales
======== ===========
PS7777          3336
PS3333          4072
BU1111          3876
MC2222          2032
TC4203         15096
BU2075         18722
PS2091          2045
PS2106           111
MC3021         22246
TC3218           375
PS1372           375
[11 rows]
```

This is another case where it's easy to get confused because of the way the question can be phrased in English. You might ask to see all books whose sales are less than 4,095 *and* all books whose sales are greater than 12,000. The logical meaning, however, calls for the use of the Boolean operator OR. If you substitute AND, you'll get no results at all because no book can have sales that are simultaneously less than 4,095 and greater than 12,000.

Lists (IN and NOT IN)

The IN keyword allows you to select values that match any one of a list of values. For example, without IN, if you want a list of the names and states of all the authors who live in California, Indiana, or Maryland, you can type this query:

SQL

```
select au_lname, state
from authors
where state = 'CA' or state = 'IN' or state = 'MD'
```

However, you get the same results with less typing if you use IN. The items following the IN keyword must be

- inside parentheses
- separated by commas
- enclosed in quotes, if they are character or date values

SQL

```
select au_lname, state
from authors
where state in ('CA', 'IN', 'MD')
```

Following is what results from either query:

Results

```
au_lname                                 state
======================================== =====
Bennet                                   CA
Green                                    CA
Carson                                   CA
DeFrance                                 IN
Panteley                                 MD
```

```
McBadden                                 CA
Stringer                                 CA
Straight                                 CA
Karsen                                   CA
MacFeather                               CA
Dull                                     CA
Yokomoto                                 CA
O'Leary                                  CA
Gringlesby                               CA
White                                    CA
Hunter                                   CA
Locksley                                 CA
[17 rows]
```

The more items in the list, the greater the savings in typing by using IN rather than specifying each condition separately.

An important use for the IN keyword is in nested queries, also referred to as subqueries. For a full discussion of subqueries, see Chapter 8.

Selecting Null Values

From earlier chapters ("NULLs" in Chapter 1), you may recall that NULL is a placeholder for unknown information. It does not mean zero or blank.

To clarify this NULL–zero difference, take a look at the following listing showing title and advance amount for books belonging to one particular publisher.

SQL

```
select title, advance
from titles
where pub_id = '0877'
```

```
title                                                    advance
======================================================== =======
Silicon Valley Gastronomic Treats                           0.00
Sushi, Anyone?                                           8000.00
Fifty Years in Buckingham Palace Kitchens                4000.00
The Gourmet Microwave                                   15000.00
Onions, Leeks, and Garlic: Cooking Secrets of the
  Mediterranean                                          7000.00
The Psychology of Computer Cooking                       (NULL)
[6 rows]
```

A cursory perusal shows that one book (*Silicon Valley Gastronomic Treats*) has an advance of $0.00, probably due to extremely poor negotiating skills on the author's part. This author will receive no money until the royalties start coming in. Another book (*The Psychology of Computer Cooking*) has a NULL advance: Perhaps the author and the publisher are still working out the details of their deal, or perhaps the data entry clerk hasn't made the entry yet. Eventually, in this case, an amount will be known and recorded. Maybe it will be zero, maybe millions, maybe a couple of thousand dollars. The point is that right now the data does not disclose what the advance for this book is, so the advance value in the table is NULL.

What happens in the case of comparisons involving NULLs? Since a NULL represents the unknown, it doesn't match anything, even another NULL. For example, a query that finds all the title identification numbers and advances for books with moderate advances (under $5,000) will not find the row for MC3026, *The Psychology of Computer Cooking*.

SQL

```
select title_id, advance
from titles
where advance < $5000
title_id      advance
======== ============
PS7777        4000.00
PS3333        2000.00
MC2222           0.00
TC4203        4000.00
PS2091        2275.00
[5 rows]
```

Neither will a query for all books with an advance over $5,000:

SQL

```
select title_id, advance
from titles
where advance > $5000
title_id      advance
======== ============
PC8888        8000.00
TC7777        8000.00
PC1035        7000.00
```

```
BU2075       10125.00
PS2106        6000.00
MC3021       15000.00
TC3218        7000.00
PS1372        7000.00
[8 rows]
```

TIP

NULL is neither above nor below (nor equal to) $5,000 because NULL is unknown.

IS NULL But don't despair! You can retrieve rows on the basis of their NULL/NOT NULL status with the following special pattern:

SYNTAX

```
WHERE column_name IS [NOT] NULL
```

Use it to find the row for books with null advances like this:

SQL

```
select title_id, advance
from titles
where advance is null
title_id      advance
======== ============
MC3026         (NULL)
PC9999         (NULL)
[2 rows]
```

SQL VARIANTS

Some systems allow the equal sign, in addition to "is".

Adaptive Server Enterprise

```
select title_id, advance
from titles
where advance = null
```

Since IS NULL is specified in the ANSI standard, it makes sense to use it, rather than use the less common = NULL.

IS NULL and Other Comparison Operators You can use the IS NULL pattern in combination with other comparison operators. Here's how a query for books with an advance under $5,000 *or* a null advance would look:

SQL

```
select title_id, advance
from titles
where advance < $5000
  or advance is null
title_id      advance
======== ============
PS7777        4000.00
PS3333        2000.00
MC2222           0.00
TC4203        4000.00
PS2091        2275.00
MC3026         (NULL)
PC9999         (NULL)
[7rows]
```

Matching Character Strings: LIKE

Some problems can't be solved with comparisons. Here are a few examples:

- "His name begins with 'Mc' or 'Mac'—I can't remember the rest."
- "We need a list of all the 415 area code phone numbers."
- "I forget the name of the book, but it has a mention of exercise in the notes."
- "Well, it's Carson, or maybe Karsen—something like that."
- "His first name is 'Dirk' or 'Dick.' Four letters, starts with a *D* and ends with a *k*."

In each of these cases, you know a pattern embedded somewhere in a column, and you need to use the pattern to retrieve all or part of the row. The LIKE keyword is designed to solve this problem. You can use it with character fields (and on some systems, with date fields). It doesn't work with numeric fields defined as integer, money, and decimal or float. The syntax is this:

SYNTAX

```
WHERE column_name [NOT] LIKE 'pattern'
   [ESCAPE escape_char]
```

The pattern must be enclosed in quotes and must include one or more **wildcards** (symbols that take the place of missing letters or strings in the pattern). You use the ESCAPE keyword when your pattern includes one of the wildcards and you need to treat it as a literal.

ANSI SQL provides two wildcard characters for use with LIKE, the percent sign (%) and the underscore or underbar (_).

Wildcard	Meaning
%	any string of zero or more characters
_	any single character

SQL VARIANTS

Many systems offer variations (notations for single characters that fall within a range or set, for example). Check your system's reference guide to see what's available.

LIKE Examples Following are answers to the questions just posed and the queries that generated them. First, the search for Scottish or Irish surnames:

SQL

```
select au_lname, city
from authors
where au_lname like 'Mc%' or au_lname like 'Mac%'
```

```
au_lname                                 city
======================================== ================
McBadden                                 Vacaville
MacFeather                               Oakland
[2 rows]
```

The LIKE pattern instructs the system to search for a name that begins with "Mc" and is followed by a string of any number of characters (%) or that begins with "Mac" and is followed by any number of characters. Notice that the wildcard is inside the quotes.

Now the 415 area code list:

```
SQL
select au_lname, phone
from authors
where phone like '415%'
au_lname                                 phone
======================================== ============
Bennet                                   415 658-9932
Green                                    415 986-7020
Carson                                   415 548-7723
Stringer                                 415 843-2991
Straight                                 415 834-2919
Karsen                                   415 534-9219
MacFeather                               415 354-7128
Dull                                     415 836-7128
Yokomoto                                 415 935-4228
Hunter                                   415 836-7128
Locksley                                 415 585-4620
(11 rows affected)
```

Here again, you're looking for some known initial characters followed by a string of unknown characters.

The book with "exercise" somewhere in its notes is a little trickier. You don't know if it's at the beginning or end of the column, and you don't know whether the first letter of the word is capitalized. You can cover all these possibilities by leaving the first letter out of the pattern and using the same "string of zero or more characters" wildcard at the beginning and end of the pattern.

```
SQL
select title_id, notes
from titles
where notes like '%xercise%'
title_id   notes
======== ====================================================
PS2106   New exercise, meditation, and nutritional techniques
         that can reduce the shock of daily interactions.
         Popular audience.  Sample menus included, exercise
         video available separately.
[1 row]
```

When you know the number of characters missing, you can use the single-character wildcard, (_). In the next example, the first letter is either *K* or *C* and the next to the last is either *e* or *o*. If the authors table contained the last name Karson, it would also be included in the results. Starson or Karstin would not.

SQL

```
select au_lname, city
from authors
where au_lname like '_ars_n'
```

```
au_lname                                 city
======================================== ====================
Carson                                   Berkeley
Karsen                                   Oakland
(2 rows affected)
```

The next example is similar to the previous one. It looks for four-letter first names starting with *D* and ending with *k*.

SQL

```
select au_lname, au_fname, city
from authors
where au_fname like 'D__k'
```

```
au_lname                                 au_fname             city
======================================== ==================== ==============
Stringer                                 Dirk                 Oakland
Straight                                 Dick                 Oakland
[2 rows]
```

NOT LIKE You can also use NOT LIKE with wildcards. To find all the phone numbers in the `authors` table that do *not* have 415 as the area code, you could use either of these queries (they are equivalent):

SQL

```
select phone
from authors
where phone not like '415%'

select phone
from authors
where not phone like '415%'
```

Escaping Wildcard characters are almost always used together with the LIKE keyword. Without LIKE, the wildcard characters are interpreted literally and represent exactly their own values. The query that follows finds any phone numbers that consist of the four characters "415%" only. It will not find phone numbers that start with 415:

SQL

```
select phone
from authors
where phone = '415%'
```

What if you want to search for a value that contains one of the wildcard characters? For example, in one row in the `titles` table, the `notes` column contains a claim to increase readers' friends by some percentage. You can search for the percent mark by using ESCAPE to appoint a character to strip the percent sign of its magic meaning and convert it to an ordinary character. A wildcard directly after the **escape character** has only its literal meaning. Other wildcards continue to have their special significance. In the following LIKE expression, you are looking for a literal percent sign somewhere in the `notes` column. Since it's probably not the first or last character, you use wildcard percent signs at the beginning and end of the expression and a percent sign preceded by the escape character in the middle.

SQL

```
select title_id, notes
from titles
where notes like '%@%%' escape '@'
title_id  notes
======= ====================================================
TC7777   Detailed instructions on improving your position in
         life by learning how to make authentic Japanese sushi
         in your spare time. 5-10% increase in number of
         friends per recipe reported from beta test.
[1 row]
```

Following are some examples of LIKE with escaped and unescaped wildcard character searches (the @ sign is the designated escape character):

Symbol	Meaning
LIKE '27%'	27 followed by any string of 0 or more characters
LIKE '27@%'	27%
LIKE '_n'	an, in, on, etc.
LIKE '@_n'	_n

Like, Is IN LIKE Equals . . . ?

Don't get confused by the similarities of equal, IN, and LIKE.

Equals Use the equal comparison operator when you want all data that exactly matches a single value—you know just what you are looking for. You can use the equal comparison operator with any kind of data—character, date, or numeric. Put quotes around character and date data. In this query, you are looking for authors named "Meander."

SQL

```
select au_lname, au_fname,  phone
from authors
where au_fname = 'Meander'
```

```
au_lname                      au_fname      phone
============================= ============ ============
Smith                         Meander       913 843-0462
[1 row]
```

IN Use IN when you have two or more values and are looking for data that exactly matches any one of these values. IN works with any kind of data—character, date, or numeric. Put quotes around character and date data. Here, you are trying to find any writers called "Meander," "Malcolm," or "Stearns."

SQL

```
select au_lname, au_fname,  phone
from authors
where au_fname in ( 'Meander', 'Malcolm', 'Stearns')
```

```
au_lname                      au_fname      phone
============================= ============ ============
MacFeather                    Stearns       415 354-7128
Smith                         Meander       913 843-0462
[2 rows]
```

LIKE Use LIKE when you want to find data that matches a pattern. For example, if you are trying to locate all the people with the letters "ea" in their names, you could write code like this:

SQL

```
select au_lname, au_fname,  phone
from authors
where au_fname like '%ea%'
au_lname                       au_fname      phone
============================== ============ ============
McBadden                       Heather       707 448-4982
MacFeather                     Stearns       415 354-7128
Smith                          Meander       913 843-0462
[3 rows]
```

In most cases, LIKE works with character and date data only.

SQL VARIANTS

Some systems support autoconvert capabilities that allow you to use LIKE with numeric data. Notice that you have to put quotes around the pattern, just as if it were character:

Oracle

```
SQL> select title_id, price
  2  from titles
  3  where price like '%.99'
TITLE_     PRICE
------ ---------
BU1032     29.99
PS7777     17.99
PS3333     29.99
MC2222     29.99
TC7777     24.99
BU2075     12.99
MC3021     12.99
BU7832     29.99
8 rows selected.
```

Other systems give an error for the same code:

```
SQL Server
select title_id, price
from titles
where price like '%.99'
Server: Msg 257, Level 16, State 3, Line 1
Implicit conversion from data type money to varchar is not allowed.
Use the CONVERT function to run this query.
```

Comparing the Three The guidelines for differentiating among equal, IN, and LIKE are compared and summarized in Figure 4.6.

Keyword	Use	Example	Notes
=	Exact matches to a single value	`where fname = 'Meander'`	All datatypes. Use quotes around character and date data.
IN	Exact matches to one or more values in a set of values—another way of specifying a series of OR clauses	`where au_fname in ('Meander', 'Malcolm', 'Stearns')`	All datatypes. Use quotes around character and date data. Separate elements with commas.
LIKE	Matches to a pattern, always used with wildcards (%, _)	`where au_fname like '%ea%'`	Character and date datatypes—others if the system does some autoconversion. ESCAPE neutralizes the wildcards.

Figure 4.6 Equal, IN, LIKE

Summary

This chapter concentrates on the basic clauses of the SELECT statement. Now you are familiar with the SELECT statement basics. These include:

- Using the asterisk for all columns in CREATE TABLE order, or listing individual column names, in any order, for a tailored report. You've also learned how to modify display labels, add text, and perform calculations in the SELECT clause.
- Specifying tables in the FROM clause, and assigning aliases as needed.
- Selecting rows in the WHERE clause, using comparison operators, logical operators, IN, IS NULL, and BETWEEN to zero in on just the values you want.

The next chapter covers some refinements on selection: ordering results with ORDER BY, eliminating duplicates in results with DISTINCT, and using aggregate functions for creating summary values.