CHAPTER 16

Selecting Data with SQL

The SQL `SELECT` statement is the key to getting data out of your database. And if the `SELECT` statement is the key to getting the data, the `WHERE` clause is the key to getting the correct data. Programmers must master both these constructs if they expect to do productive programming for a database. As a database administrator, you probably don't need quite the same level of mastery as a programmer, but you do need to be familiar with the basic concepts of selecting data, joining tables, and summarizing data.

Note All of the examples in this chapter are based on the sample tables owned by the user named `AMY`.

Using the Basic SELECT Query

You use `SELECT` statements to retrieve data from an Oracle database. You use them to specify which tables you are interested in and which columns you want back from those tables. The general form of a `SELECT` statement looks like this:

```
SELECT column_list
FROM table_list
WHERE conditions
GROUP BY column_list
HAVING conditions
ORDER BY column_list;
```

Each of the clauses in this example has a specific function:

✥ `SELECT` *`column_list`*—Specifies which columns you want to see

✥ `FROM` *`table_list`*—Specifies which tables contain those columns

- ✦ `WHERE` *`conditions`*—Restricts the data that Oracle returns so that you see only rows that match the specified conditions
- ✦ `GROUP BY` *`column_list`*—Causes Oracle to summarize data over the values in the specified columns
- ✦ `HAVING` *`conditions`*—Restricts the summarized data so that you see only summary rows that match the specified conditions
- ✦ `ORDER BY` *`column_list`*—Specifies how to sort the data

You'll learn about each of these clauses as you read this chapter.

Selecting data

A simple `SELECT` statement might look like this:

```
SELECT last_name, first_name, specialty
FROM artist;
```

This statement returns the three named columns from the `ARTIST` table. If you were to execute it using SQL*Plus, your results would look like this:

```
LAST_NAME      FIRST_NAME     SPECIALTY
-------------- -------------- ---------------
Smith          Ken            DIGITAL
Palionis       Lina           OTHER
Ahrens         Lorri          WATERCOLOR
Mosston        Nikki          ACRYLIC
Perry          Robert         MIXED MEDIA
Stone          Sharron        DIGITAL
Michael        Sherry         WATERCOLOR
Roberts        Stephen        MIXED ON PAPER
Joyce          Terence        DIGITAL
Taylor         Thomas         OIL
```

Notice that these rows aren't sorted in any particular order. That's because an `ORDER BY` clause wasn't used.

Sorting data

You can use the `ORDER BY` clause to specify sorting the data returned by a query. The clause goes at the end of the query and consists of a comma-delimited list of columns. Oracle will sort the query results based on those columns. For example, to sort the artist list by last name and first name, you could write a query like this:

```
SELECT last_name, first_name, specialty
FROM artist
ORDER BY last_name, first_name;
```

Oracle will sort the results in ascending order on the columns that you specify. You can also sort in descending order by using the DESC keyword after a column name. The query in Listing 16-1 sorts in descending order by specialty and then in ascending order by name within specialty.

Listing 16-1: Using the ORDER BY clause

```
SQL> SELECT last_name, first_name, specialty
  2  FROM artist
  3  ORDER BY specialty DESC, last_name, first_name;

LAST_NAME     FIRST_NAME    SPECIALTY
------------- ------------- --------------
Ahrens        Lorri         WATERCOLOR
Michael       Sherry        WATERCOLOR
Palionis      Lina          OTHER
Taylor        Thomas        OIL
Roberts       Stephen       MIXED ON PAPER
Perry         Robert        MIXED MEDIA
Joyce         Terence       DIGITAL
Smith         Ken           DIGITAL
Stone         Sharron       DIGITAL
Mosston       Nikki         ACRYLIC
```

You can use an ASC keyword to indicate ascending order, but it's rarely used because an ascending sort is the default.

Caution

If you want data to be returned in a certain order, you must use an ORDER BY clause to specify the sort order that you want. Some common misconceptions are that Oracle returns records in primary key order, in index order, or in the order in which records were inserted. All of these arrangements are true sometimes, under the right circumstances, but you can't count on them being true all of the time. If the order of your query results is important, use an ORDER BY clause.

Restricting the results

Chances are good that you won't want to see all the data in a table when you issue a query. You can use the WHERE clause to restrict the data returned by Oracle to those rows that interest you. The WHERE clause, if one exists, follows immediately after the FROM clause. The following query, for example, returns only those artists whose specialty is 'DIGITAL':

```
SELECT last_name, first_name, specialty
FROM artist
WHERE specialty = 'DIGITAL'
ORDER BY specialty DESC, last_name, first_name
```

This WHERE clause is very simple and contains a test only for equality. Oracle supports a number of operators that you may use, and these are listed in Table 16-1.

Table 16-1
Comparison Operators

Operator	***Purpose***	***Example***
`=`	Equality.	`Specialty = 'DIGITAL'`
`<>`	Inequality.	`Specialty <> 'DIGITAL'`
`<`	Less than.	`book_due < sysdate`
`>`	Greater than.	`book_due > sysdate`
`<=`	Less than or equal to.	`book_due <= sysdate`
`>=`	Greater than or equal to.	`book_due >= sysdate`
`LIKE`	Limited wildcard comparisons. Use `'%'` to match any string. Use `'_'` to match any one character.	`Specialty LIKE 'DIGI%'` `Specialty LIKE 'DIG_TAL'`
`BETWEEN`	Tests to see if one value is between two others. The range is inclusive.	`Specialty BETWEEN 'D' and 'E'`
`IN`	Tests to see if a value is contained within a list of values.	`Specialty IN ('DIGITAL', 'WATERCOLOR')`
`IS NULL`	Tests to see if a value is null.	`Specialty IS NULL`
`IS NOT NULL`	Tests to see if a value is not null.	`Specialty IS NOT NULL`

Keep these points in mind while writing WHERE clauses:

✦ Quoted strings must be within single quotes, not double quotes.

✦ If you need to place an apostrophe inside a string, you should double it. For example, if you need to compare a column to the string "Won't," write it in the WHERE clause as 'Won''t'.

✦ Always think about the effects of null values. For each column in a WHERE clause, you have to ask yourself what would happen if that column was null. See the section "Beware of Nulls!" later in this chapter.

✦ Use care when applying a function to a database column because this can prevent indexes from being used. See the section "Using SQL Functions in Queries" later in this chapter.

- If you are summarizing data, try to eliminate all the data that you can in the `WHERE` clause rather than the `HAVING` clause. See the section "Summarizing Data" later in this chapter.

The `WHERE` clause may be the most important part of a SQL statement. It's usually the most difficult to write, yet it's also where you can be the most creative.

Joining Tables

When a query combines data from two tables, it is called a *join*. The ability to join tables is one of the most fundamental operations of a relational database. That's where the term *relational* comes from — joining two tables that contain related data.

There are different ways of classifying joins, and you'll often hear people bandy about terms such as "right outer join," "left outer join," "equi-join," and so forth. Most of those terms don't really matter or are useful only to academics who spend a lot of time theorizing about SQL. When it comes to Oracle, there are two keys to understanding joins:

- You need to understand the concept of a Cartesian product.
- You need to understand the difference between an inner join and an outer join.

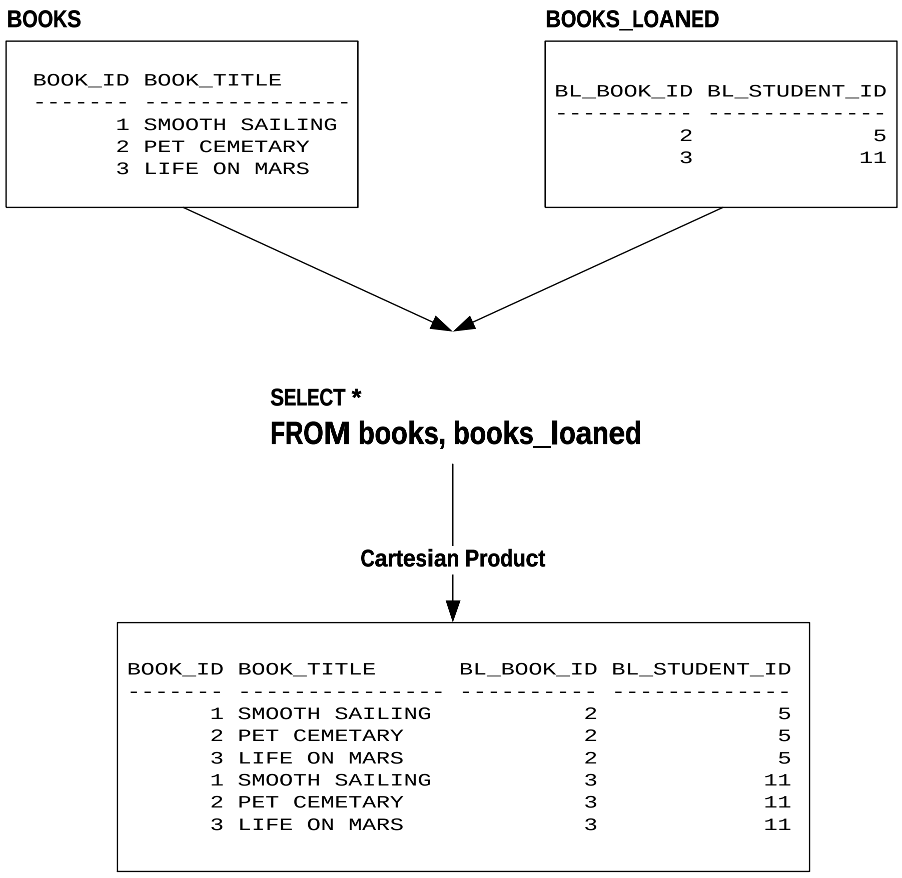
If you get these two concepts down, you won't have any trouble joining tables together in your queries.

Deriving a Cartesian product

Whenever you join two tables in a relational database, conceptually, you begin working with the Cartesian product of those two tables. The *Cartesian product* is defined as every possible combination of two rows from the two tables. Figure 16-1 illustrates this concept using the `BOOKS` and `BOOKS_LOANED` table.

As you can see in Figure 16-1, the Cartesian product has a multiplying effect. The `BOOKS` table contains three rows, and the `BOOKS_LOANED` table contains two rows. The resulting Cartesian product consists of six rows. This has some serious performance implications when you are dealing with large tables, and companies such as Oracle expend a great deal of time, money, and effort looking for ways to optimize table joins.

Note In practice, Cartesian products are rarely generated. Conceptually, though, all joins start with the Cartesian product and are then winnowed down by conditions in the `WHERE` clause. You might find visualizing this Cartesian product to be an effective aid to writing correct join conditions in your `WHERE` clauses.



```
BOOK_ID BOOK_TITLE
------- --------------
      1 SMOOTH SAILING
      2 PET CEMETARY
      3 LIFE ON MARS
```

```
BL_BOOK_ID BL_STUDENT_ID
---------- -------------
         2             5
         3            11
```

```
BOOK_ID BOOK_TITLE      BL_BOOK_ID BL_STUDENT_ID
------- -------------- ---------- -------------
      1 SMOOTH SAILING          2             5
      2 PET CEMETARY            2             5
      3 LIFE ON MARS            2             5
      1 SMOOTH SAILING          3            11
      2 PET CEMETARY            3            11
      3 LIFE ON MARS            3            11
```

Figure 16-1: The Cartesian product of the BOOKS and BOOKS_LOANED tables

You would rarely ever really want the Cartesian product to be returned as the result when you join two tables. You will almost always end up adding conditions to your WHERE clause to eliminate combinations of rows that don't make sense.

Using equi-joins to compare columns

Equi-joins are the most common type of join, and they are most often used to join two tables that have a parent-child relationship. Equi-joins compare one or more columns in a set of tables for equality.

For example, the BOOKS table is related to the BOOKS_LOANED table by a parent-child relationship. Any one book may be loaned out several times over its lifetime, and each loan will be recorded with an entry in the BOOKS_LOANED table. If you want to report on the loan history of your books, you could start by querying the BOOKS_LOANED table, as follows:

```
SQL> SELECT bl_book_id, bl_student_id, bl_loan_date
  2  FROM books_loaned;

BL_BOOK_ID BL_STUDENT_ID BL_LOAN_D
---------- ------------- ---------
         2             5 04-MAY-98
```

This report really doesn’t tell you much, because unless you have a good memory for ID numbers, you won’t know the name of the book or student involved with any particular loan. However, you can get the book name from the BOOKS table. To do that, you must join the BOOKS and BOOKS_LOANED tables. The query in this next example does this. Notice that the BOOKS table has been added to the FROM clause, and the BOOK_TITLE and BOOK_ID columns have been added to the SELECT list.

```
SQL> SELECT book_id, book_title,
  2         bl_book_id, bl_student_id, bl_loan_date
  3  FROM books_loaned, books;

  BOOK_ID BOOK_TITLE      BL_BOOK_ID BL_STUDENT_ID BL_LOAN_D
--------- --------------- ---------- ------------- ---------
        1  SMOOTH SAILING          2             5 04-MAY-98
        2  PET CEMETARY            2             5 04-MAY-98
        3  LIFE ON MARS            2             5 04-MAY-98
```

Yuck! These results clearly aren’t what you want. There was only one loan, yet the report now lists three loans, one for each book. What you are seeing here is the Cartesian product. The two tables were joined without the proper join condition being added to the WHERE clause. If you look at the BOOK_ID and BL_BOOK_ID columns, you’ll see that the results make sense in only one case, and that’s where the two columns are equal. Add a WHERE clause with that condition to the query, and it will now return the result you’re after, as shown in the following example:

```
SQL> SELECT book_id, book_title,
  2         bl_book_id, bl_student_id, bl_loan_date
  3  FROM books_loaned, books
  4  WHERE book_id = bl_book_id;

  BOOK_ID BOOK_TITLE      BL_BOOK_ID BL_STUDENT_ID BL_LOAN_D
--------- --------------- ---------- ------------- ---------
        2 PET CEMETARY             2             5 04-MAY-98
```

Now that you have the query returning the loan history, including the book name, it’s time to think about adding the student name into the mix. The relationship between STUDENTS and BOOKS_LOANED is the same as between BOOKS and

BOOKS_LOANED, so you can add the STUDENTS table to the query using the same technique as you used for BOOKS. The resulting query looks like this:

```
SELECT book_id, book_title,
       bl_book_id, bl_student_id,
       student_id, first_name, last_name,
       bl_loan_date
FROM books_loaned, books, students
WHERE book_id = bl_book_id
AND student_id = bl_student_id;
```

The queries in this section include the book ID number from both the BOOKS table and the BOOKS_LOANED table. It just so happens that the column names were slightly different between the two tables. However, what if the column names were identical? Would you write WHERE book_id = book_id in your query? How would Oracle tell the difference? These are good questions, and form the subject of the next section.

Using table aliases

When you join two tables, you may run into column names that are identical in each table. That begs the question of how to uniquely identify the columns when you refer to them in your query. One way is to prefix each column name with its respective table name. Consider this example:

```
SELECT books.book_id, books.book_title,
       books_loaned.bl_book_id, books_loaned.bl_student_id,
       students.student_id, students.first_name,
       students.last_name,
       books_loaned.bl_loan_date
FROM books_loaned, books, students
WHERE books.book_id = books_loaned.bl_book_id
AND students.student_id = books_loaned.bl_student_id;
```

Qualifying each column name with a table name is effective, but it often makes the query difficult to read, especially if your table names are on the long side. Because of this, in SQL, you are allowed to define aliases for the tables within a query. An *alias* is simply an alternate, and usually shorter, name for a table. You specify table aliases in the FROM clause by placing them immediately after the table names, separated by at least one space, as shown in this example:

```
FROM books_loaned bl, books b, students s
```

This FROM clause specifies aliases of BL for BOOKS_LOANED, B for BOOKS, and S for STUDENTS. Now you can use these aliases to qualify the column names within the query, as shown in this example:

```
SELECT b.book_id, b.book_title,
        bl.bl_book_id, bl.bl_student_id,
        s.student_id, s.first_name, s.last_name,
        bl.bl_loan_date
FROM books_loaned bl, books b, students s
WHERE b.book_id = bl.bl_book_id
AND s.student_id = bl.bl_student_id;
```

Short one- or two-letter aliases are typical because they make the query easier to read. They allow you to focus more on the column names than the table names.

Tip

When you define table aliases, try to make them mnemonic. Sometimes, you will see people using “a,” “b,” “c,” and so forth. Avoid that because it becomes difficult to remember which table is “a” and which table is “b.” The example in this section uses “bl” for “books loaned,” making it easy to associate the two.

Using outer joins

An *outer join* allows you to join two tables and return results even when the second table doesn’t have any records corresponding with the first. To really understand this, you have to know what an inner join is. An *inner join* between two tables returns rows where there are corresponding records in *both* tables. The join between BOOKS and BOOKS_LOANED is such a join. It returns a record only for the one book that had been loaned out, not for all books. Those results look like this:

```
  BOOK_ID BOOK_TITLE      BL_BOOK_ID BL_STUDENT_ID BL_LOAN_D
--------- --------------- ---------- ------------- ---------
        2 PET CEMETARY             2             5 04-MAY-98
```

Books that had never been loaned out weren’t returned by the query. Why not? Because the join condition requires that the BOOK_ID column in the BOOKS table equal a BL_BOOK_ID value in the BOOKS_LOANED table. Since there were no loans recorded for books 1 and 3, that condition could never be true.

That’s all well and good, but what if you want to produce a report listing all books, including loan history if there was any, but not excluding any books without a loan history? The solution is to write an outer join so you’ll have one table that is the anchor table. You will always get at least one row back for each row in the anchor table. In this case, the BOOKS table will be the anchor. When you write the query, you indicate that you want an outer join by placing a (+) in the WHERE clause after the column names from the optional table. That’s probably clear as mud, so here’s an example:

```
SELECT book_id, book_title,
       bl_book_id, bl_student_id, bl_loan_date
FROM books_loaned, books
WHERE book_id = bl_book_id (+);
```

The (+) notation following the BL_BOOK_ID column specifies that you are doing an outer join. In return, whenever a book is encountered that doesn't have a loan history, Oracle will return one row for the book, but with null values for all the loan-related columns. The output will look like this:

```
  BOOK_ID BOOK_TITLE      BL_BOOK_ID BL_STUDENT_ID BL_LOAN_D
--------- --------------- ---------- ------------- ---------
        1 SMOOTH SAILING
        2 PET CEMETARY             2             5 04-MAY-98
        3 LIFE ON MARS
```

Note

Oracle is the only database vendor to use this rather strange (+) notation. Other database vendors allow you to specify outer joins in the FROM clause. This actually makes more sense because an outer join is table-specific, not column-specific.

Outer joins can become increasingly complex as you add more tables to the join, especially if these additional tables relate to the optional table. If you want to join the STUDENTS table to the query shown here, you need to also make that an outer join, because STUDENTS relates to BOOKS_LOANED. If BOOKS_LOANED is optional, then STUDENTS must be too. The following query will return all books, their loan records, if any exist, and the name of the student who goes with each loan record.

```
  SELECT book_id, book_title,
         bl_book_id, bl_student_id,
         student_id, first_name, last_name,
         bl_loan_date
  FROM books_loaned, books, students
  WHERE book_id = bl_book_id (+)
  AND student_id (+) = bl_student_id;
```

With the addition of the STUDENTS table, the (+) notation has been appended following the STUDENT_ID column in the WHERE clause. This makes the STUDENTS table optional. That way, when no BOOKS_LOANED record exists for a book, the lack of a student record won't matter, and the query will still return information for that book. Here are the results of executing this query:

```
BOOK_ID BOOK_TITLE      BL_BOOK_ID BL_STUDENT_ID STUDENT_ID...
------- --------------- ---------- ------------- ----------...
     2  PET CEMETARY             2             5          5...
     1  SMOOTH SAILING                                     ...
     3  LIFE ON MARS                                       ...
```

Oracle places one restriction on outer joins: One table can't be the optional table in two outer joins. You could not, for example, change the last line in the previous query to read AND student_id = bl_student_id (+).

Summarizing Data

Several SQL functions exist that allow you to summarize data. These are referred to as *aggregate functions* because they return values aggregated from more than one row of data. Table 16-2 describes each of these aggregate functions.

Table 16-2
SQL Aggregate Functions

Function	*Description*
AVG	Returns the average of all values in the column
COUNT	Returns a count of rows or a count of values in a column
MAX	Returns the maximum value of a column
MIN	Returns the minimum value of a column
STDDEV	Returns the standard deviation of all values in the column
SUM	Returns the sum of all values in a column
VARIANCE	Returns the variance (related to standard deviation) of all values in a column

The summary functions listed in Table 16-2 are almost always used in conjunction with a GROUP BY clause. The one exception is when you are summarizing all the data returned by a query.

Performing query-level summaries

The simplest form of summarization is when you use one of the aggregate functions to summarize the results of an entire query. This is often done to return a count of rows in a table. The following example shows the COUNT function being used to return the number of rows in the ARTISTS table:

```
SQL> SELECT COUNT(*)
  2  FROM artist;

 COUNT(*)
---------
       10

1 row selected.
```

The asterisk used as an argument to COUNT indicates that you want the function to include every row in the table. You can pass in a column name instead, in which case, Oracle will count the number of values in that one column. The results may be different because null values in a column are not counted. Here's an example illustrating that point:

```
SQL> SELECT COUNT(city)
  2  FROM artist;

COUNT(CITY)
-----------
          9

1 row selected.
```

It seems that one artist has a null value for the CITY column. Since nulls represent the lack of a value, they aren't counted. In this case, the ARTIST table contains nine city values.

Using the GROUP BY clause

In addition to summarizing data for an entire table, you can use the GROUP BY clause to summarize data for a specific set of columns. This is almost better explained by example. Suppose you want to know how many artists of each specialty you have. You could determine this by writing a query like the one shown in Listing 16-2.

Listing 16-2: **Using the GROUP BY clause to summarize data**

```
SQL> SELECT specialty, COUNT(*)
  2  FROM artist
  3  GROUP BY specialty;

SPECIALTY       COUNT(*)
-------------- ---------
ACRYLIC                1
DIGITAL                3
MIXED MEDIA            1
MIXED ON PAPER         1
OIL                    1
OTHER                  1
WATERCOLOR             2

7 rows selected.
```

Notice what's going on here. The select list contains one column not used in an aggregate function. That column is also listed in the GROUP BY clause. The aggregate function COUNT(*) is also in the select list, causing Oracle to count up the number of rows for each distinct value in the specialty column.

Oracle evaluates a GROUP BY query like this by going through these steps:

1. Oracle retrieves the data for the query and saves it in temporary storage (possibly in memory, or possibly on disk).
2. Oracle sorts the data enough to group all records with like GROUP BY values together. The results might look like this:

   ```
   WATERCOLOR
   WATERCOLOR
   DIGITAL
   DIGITAL
   DIGITAL
   MIXED MEDIA
   MIXED ON PAPER
   ACRYLIC
   OIL
   OTHER
   ```

   Note that the sort is not a complete sort. Oracle is concerned only with grouping like values together. A strict alphabetical sort isn't necessary at this point.
3. Oracle applies the specified aggregate function to each group of records and returns one row for each group.

Because summary queries return one row for each group, you can't have columns in your select list that aren't also listed in your GROUP BY clause, unless those columns have an aggregate function applied to them. You cannot, for example, write this query:

```
SELECT specialty, last_name, first_name, COUNT(*)
FROM artist
GROUP BY specialty;
```

Why can't you do this? Because one specialty may represent many artists. Three artists have a specialty of 'DIGITAL'. Since the query is returning only one row for a specialty, how is it supposed to choose which first and last names go with that row? The answer is that there is no automatic way to do this. You either need to apply an aggregate function to the first and last name fields, or you must remove them from the select list.

You can group query results by more than one column, and COUNT is certainly not the only aggregate function at your disposal. The following example shows a GROUP BY query consisting of a join and two aggregate functions:

```
SELECT book_title, bl_student_id, COUNT(*), SUM(bl_fine)
FROM books, books_loaned
WHERE bl_fine > 0
AND books.book_id = bl_book_id
GROUP BY book_title, bl_student_id;
```

This query lists all books for which fines were paid, uses the COUNT function to tell you how many fines were paid for each book, and uses the SUM function to tell you the total dollar amount of fines for any one book. It does this for each combination of book and student.

Using the HAVING clause

The HAVING clause is a clause that you can add to a GROUP BY query to restrict the rows returned by that query. HAVING is just like WHERE, except that the conditions in the HAVING clause are applied to the data after it is summarized. You can use HAVING when you are interested only in specific groups. A good example might be if you wanted to see a list of artist specialties that were represented only by one artist. You could generate that list using the query shown in Listing 16-3.

Listing 16-3: Restricting rows in a query

```
SQL> SELECT specialty
  2  FROM artist
  3  GROUP BY specialty
  4  HAVING COUNT(*) = 1;

SPECIALTY
--------------
ACRYLIC
MIXED MEDIA
MIXED ON PAPER
OIL
OTHER

5 rows selected.
```

Notice that the HAVING clause in this query restricted the results based on an aggregate function. It has to do that because the HAVING clause is applied only after the data has been summarized. The HAVING clause can use only the GROUP BY columns and aggregate functions on other columns. Column values for individual rows aren't available. The aggregate functions used in the HAVING clause do not necessarily need to be the ones listed in the SELECT clause.

While it's possible to use the GROUP BY columns in the HAVING clause, it's often more efficient if you take another approach. For example, you could write the following query, which excludes the specialty named 'OTHER':

```
SELECT specialty
FROM artist
GROUP BY specialty
HAVING COUNT(*) = 1
AND specialty <> 'OTHER';
```

This query will work as expected, but think about what Oracle needs to do to resolve it. All the rows containing 'OTHER' must be read from the database, then grouped and summarized like those for any other value. Only after that data is summarized can the result be excluded. You could make this query much more efficient by avoiding the need to read those rows in the first place. You can do that by moving the specialty condition to the WHERE clause:

```
SELECT specialty
FROM artist
WHERE specialty <> 'OTHER'
GROUP BY specialty
HAVING COUNT(*) = 1;
```

This query will return the same results but will execute more efficiently because rows containing 'OTHER' never need to be read, placed in temporary storage, sorted, and summarized. Any time that you find yourself using GROUP BY columns in the HAVING clause, you should consider the possibility of moving those conditions to the WHERE clause.

Using the DISTINCT keyword vs. the ALL keyword

You can use two keywords with aggregate functions that affect how the data is summarized. These keywords are DISTINCT and ALL. The difference lies in whether the aggregate function summarizes each value in a group or each distinct value in a group. Let's use artist specialties as the basis for an example. You can count the values in the SPECIALTY column like this:

```
SQL> SELECT COUNT(specialty)
  2  FROM artist;

COUNT(SPECIALTY)
----------------
              10
```

This query returns a value of 10 because there are 10 artists in the table and each has a specialty. That isn't a very useful number, though. You might be more interested in the number of distinctly different specialties that exist. For that, you could use the DISTINCT keyword, as in this example:

```
SQL> SELECT COUNT(DISTINCT specialty)
  2  FROM artist;

COUNT(DISTINCTSPECIALTY)
------------------------
                       7
```

Because the DISTINCT keyword is used, Oracle counts each distinct specialty value only once. In this case, there are seven different specialties. You can use the ALL keyword in place of DISTINCT, but that's rarely done since ALL represents the default behavior anyway.

Note

Given this scenario, you'd probably be better off selecting the SPECIALTY column and grouping by that column, since your next question after finding out that there are seven specialties is going to be, "What are they?"—unless, of course, you were using this as a subquery.

When developing SQL queries, you can follow the steps outlined in the following sidebar. It discusses a methodology for query development.

Using Subqueries

Subqueries are queries that are nested within a larger query. You can use subqueries in a WHERE clause to express conditions that can't be expressed using simple comparison expressions. You should be aware of three types of subqueries:

- ✦ Noncorrelated subqueries, which execute independently of the parent query
- ✦ Correlated subqueries, which are executed once for each row returned by a parent query, and which use values from that row
- ✦ Inline views, which are subqueries that appear in the FROM clause

A Methodology for Query Development

I would like to share with you my personal methodology for developing SQL queries. I've often thought that standard SQL syntax encourages a backwards approach to query development. Since `SELECT` comes first, people tend to write that clause first. If the syntax were to match my order of query development, it would look like this:

```
FROM table_list
WHERE conditions
GROUP BY column_list
HAVING conditions
SELECT column_list
```

I generally find it helpful to follow the steps listed here when I'm developing a new SQL query. I start my work with the `FROM` clause. As I work through each step, I continuously test and modify the query. Usually, I do that by executing the partially completed query using SQL*Plus, and viewing the results. Here are my steps:

1. I figure out which tables contain the data that I want.
2. I work out the join conditions for these tables. At this point, I write the `WHERE` clause for those join conditions, and I place all the join columns in my select list. If I'm joining a number of tables, I add one at a time to the query and test the results after each addition.
3. Once I'm sure of having the joins correct, I add any other needed conditions to the `WHERE` clause. I never do this earlier because doing so could mask problems with the joins.
4. Having narrowed the results down to the data that I need, I look at any grouping that needs to be done.
5. If I did use a `GROUP BY` clause, I write the `HAVING` clause only after I know that I have the `GROUP BY` clause correct.
6. Finally, I go back and make the `SELECT` list look right. I remove columns that I had listed, to verify that joins were working correctly, I apply any needed SQL functions, and I add any other miscellaneous columns that I wasn't forced to add in any earlier steps.

There you have it. I've found that this process can reduce the development of some complex queries to a relatively routine task. The key here is to build up your queries a piece at a time, testing and validating each piece as you go.

Noncorrelated subqueries

A *noncorrelated subquery* doesn't depend on any values from rows returned by the parent query. You can use a noncorrelated subquery to obtain a list of values for use with the IN operator. Say you're interested in generating a list of students who don't appear to be using the school library. You might decide to use borrowing a book as a basis for that list, and you assume that if a student has never borrowed a book, then that student isn't using the library. That may not always be a correct assumption, but it's the only data that you have to work with. The following query uses the BOOKS_LOANED table to return a list of students who *have* borrowed at least one book:

```
SELECT DISTINCT bl_student_id
FROM books_loaned;
```

The DISTINCT keyword eliminates duplicate values from the result list. Even if a student borrowed a hundred books, you need only one copy of his or her ID number. Now that you have this query, you can use it as part of a larger query that answers the original question. For example:

```
SELECT student_id, first_name, last_name
FROM students
WHERE student_id NOT IN (
   SELECT DISTINCT bl_student_id
   FROM books_loaned
   );
```

The outer query here returns a list of students not represented by the inner query; in other words, it returns a list of students who have never borrowed a book. The subquery in the WHERE clause is independent of the parent query, making it noncorrelated. The acid test here is that you can pull the subquery out as it stands, execute it on its own, and get results.

Correlated subqueries

A *correlated subquery* is one in which the result depends on values returned by the parent query. Because of this dependency, Oracle must execute these queries once for each row examined by the parent query: hence, the term correlated.

We can also use the problem of identifying students who have never borrowed books to demonstrate the use of correlated subqueries. We want a list of students who have never borrowed books, so for any given student, we need to check the BOOKS_LOANED table to see if any records exist. Here, a correlated subquery is used to do that:

```
SELECT student_id, first_name, last_name
```

```
  FROM students s
  WHERE NOT EXISTS (
     SELECT *
     FROM books_loaned bl
     WHERE bl.bl_student_id = s.student_id
     );
```

The NOT EXISTS operator is used here, and it will return a value of TRUE if no rows are returned by the subquery that goes with it. Notice the WHERE clause of this subquery. It contains a reference to a value in the STUDENTS table, which is listed in the FROM clause of the outer query. That makes this a correlated subquery. It must be executed once for each student. You couldn't pull this subquery out and execute it as it stands because the STUDENT_ID column will not be recognized. The subquery makes sense only in the context of the outer query.

Tip

Specify table aliases when you write correlated subqueries, and use them to qualify all your column names. This is necessary if your subquery is against the same table as the parent query, but even when it's not necessary, the aliases will help you keep straight to which table a column belongs.

You've now seen the same query implemented in two ways: using NOT IN and using NOT EXISTS. Which is better? That really depends on the execution plan that the optimizer puts together for the query. You might hear debates between those who feel that NOT IN is always the best approach and those who believe that NOT EXISTS is always best. Don't waste your time with debates. If you're ever in doubt about which approach to take, just test it out. Chapter 19, "Tuning SQL Statements," shows you how to discover the optimizer's execution plan for a statement. Arm yourself with that information, and perhaps with some statistics from a few trial executions, and let the numbers speak for themselves.

Inline views

An *inline view* is a subquery used in the FROM clause in place of a table name. Such a subquery functions the same as if you had created and used a view based on the query. Here's an example:

```
  SELECT student_id, books_borrowed
  FROM (
     SELECT s.student_id student_id,
            COUNT(bl.bl_student_id) books_borrowed
     FROM students s, books_loaned bl
     WHERE s.student_id = bl.bl_student_id (+)
     GROUP BY s.student_id
     )
  WHERE books_borrowed = 0;
```

The inline view in this example returns a complete list of students, together with the number of books that they have borrowed. You accomplish this by using an outer join from STUDENTS to BOOKS_LOANED and counting up the number of BL_STUDENT_ID values. Remember, nulls don't count. Whenever a student has no corresponding loan records, the BL_STUDENT_ID value will be null, and that student will get a value of 0. Column aliases are specified for both columns returned by the inline view, and these become the column names visible to the parent SELECT statement. The outer SELECT statement, then, selects only those records where the BOOKS_BORROWED count is equal to zero.

Note The number of different solutions to the problem of finding students who have never borrowed books shows just how creative you can get when writing SQL queries. You haven't seen the last of this problem yet, either. There's at least one more solution shown later in this chapter.

Inline views function just as any other view, and you can join them to other tables, views, or even other inline views. You can join the inline view just discussed to the STUDENTS table to allow the parent SELECT statement to retrieve the first and last names for each student. Here's an example showing how to do that:

```
SELECT bb.student_id, bb.books_borrowed,
       s.last_name, s.first_name
FROM (
   SELECT s.student_id student_id,
          COUNT(bl.bl_student_id) books_borrowed
   FROM students s, books_loaned bl
   WHERE s.student_id = bl.bl_student_id (+)
   GROUP BY s.student_id
   ) bb, students s
WHERE books_borrowed = 0
AND bb.student_id = s.student_id;
```

In this case, because the inline view and the STUDENTS table both contain columns named STUDENT_ID, you use table aliases of BB and S to make it clear just which table is being referred to at any time.

Beware of Nulls!

Null values have a strangely unintuitive effect on the values of expressions, especially those returning Boolean values such as those that you often find in a WHERE clause. To write queries that properly handle null values, you need to understand three-valued logic. You also need to understand the built-in operators and functions that have been specifically designed to deal with null values.

The two primary problems that people encounter when using nulls are three-valued logic and the effect of nulls on expressions.

Recognizing three-valued logic

The primary problem with nulls is that SQL deals with them using a system known as *three-valued logic*. In our day-to-day lives, most of us become accustomed to two-valued logic. A statement may be either true or false. Three-valued logic introduces another option referred to as *unknown*. The following statement demonstrates this nicely:

```
SELECT first_name, last_name
FROM artist
WHERE city = 'Madison'
OR city <> 'Madison';
```

Intuitively, most people would expect this statement to return every row in the `ARTIST` table. After all, we are asking for the records for Madison, and also for other cities. In reality, though, if you execute this statement against the data in the sample database, you'll only get nine of the ten artists' records back because one artist has a null value for the `CITY` column. Is null equal to `'Madison'`? The answer is unknown. Is null not equal to `'Madison'`? That answer is also unknown. Is unknown the same as true? No, it's not, so the record with a null is never returned.

Understanding the impact of nulls on expressions

Another problem with nulls is that if any value in an expression happens to be a null, then the result of the expression will also be a null. Theoretically, this is reasonable behavior. Practically, it's often not what you need to get the job done.

You typically need to deal with null values in two parts of a query: the select list and the `WHERE` clause.

Using nulls in the SELECT list

Your primary concern with the `SELECT` list is whether you want `NULL` values to be returned to your program. Or, if you are writing a SQL*Plus report, you need to be concerned with what the report displays when a value is null. The following query returns three columns, all of which could be null:

```
SELECT bl_book_id, bl_student_id, bl_fine
FROM books_loaned;
```

Assuming for a moment that you would never have a loan record without a student and a book, how do you want to deal with the `BL_FINE` field? One choice is to leave it as it is and allow nulls to be returned by the query. Your other choice is to supply a reasonable alternate value to use in place of nulls. You can use Oracle's built-in NVL function to do this. The following example shows the query rewritten to return 0 whenever the fine field is null:

```
SELECT bl_book_id, bl_student_id, NVL(bl_fine,0)
FROM books_loaned;
```

The NVL function takes two arguments. One is a column name or an expression. The other is the value to be returned when the first argument evaluates to null. In this example, NVL will return 0 whenever BL_FINE is null. Otherwise, it will return the value of BL_FINE.

Using nulls in the WHERE clause

With WHERE clauses (and HAVING clauses, too), your primary concern is whether you want records with null values included in the query results. Take a look at this query, for example:

```
SELECT *
FROM books_loaned
WHERE bl_fine <= 10;
```

This query is asking for all records where the fine was less than or equal to $10. That seems straightforward enough, but what about records where the BL_FINE column is null? Should those be included? If your answer is yes, you can include them by adding a condition that uses the IS NULL operator. Consider this example:

```
SELECT *
FROM books_loaned
WHERE bl_fine <= 10
OR bl_fine IS NULL;
```

You could also use NVL in the WHERE clause and write WHERE NVL(bl_fine,0) <= 10, but as you'll find out in the next section, using functions on columns in the WHERE clause isn't always a good idea.

In addition to IS NULL, there is the IS NOT NULL operator. Use IS NULL when you want to check for a value being null; use IS NOT NULL when you want to check a value to see if it is something other than null.

Tip Always use IS NULL or IS NOT NULL when testing for nullity. Don't ever write anything like WHERE bl_fine = NULL. The equality operator can't test for nullity, and that expression will always evaluate to unknown.

Using SQL Functions in Queries

Oracle implements a number of built-in functions that you can use in your queries. You've already seen one example in the NVL function used to translate null values to something other than null. A few of the more interesting and useful functions are described here. See Appendix B, "SQL Built-in Function Reference," for a complete list.

Using functions on indexed columns

When you use functions in a query, especially in the WHERE clause of a query, be aware of the potential for those functions to preclude the use of an index. Say, for example, that you have indexed the SPECIALTY column in the ARTIST table. A query such as the following, which selects for a specific specialty, could take advantage of that index:

```
SELECT last_name, first_name
FROM artist
WHERE specialty = 'WATERCOLOR';
```

Applying a function to the specialty column, however, changes the rules. Look at this example, where the UPPER function has been applied to make the query case-insensitive:

```
SELECT last_name, first_name
FROM artist
WHERE UPPER(specialty) = 'WATERCOLOR';
```

This query no longer searches for a specific value in the SPECIALTY column; rather, it is searching for rows where the expression UPPER(specialty) returns a specific value. This distinction is important to make, because this normally precludes Oracle from using any indexes on the SPECIALTY column. Instead, in this case, Oracle will be forced to read every row in the table, evaluate the expression for each row, and test the result. On a large table, this can be a significant performance issue.

New Feature

Oracle8i implements a new feature that allows you to index the results of an expression. You can use this feature to create an index on an expression such as UPPER(specialty), which Oracle will then be able to use when you make such a function call in your queries. See Chapter 14, "Managing Indexes," for information on creating function-based indexes.

If you're not using function-based indexes, you need to carefully consider your use of functions in the WHERE clause. Avoid applying them to indexed columns, if at all possible.

Using the DECODE function

You can use the DECODE function to work as an inline IF statement inside a query. A call to DECODE includes a variable number of arguments used to transfer an input argument into some other value. The first argument is the input argument. The second and third arguments form a pair. If the input argument matches the second argument, the function returns the third argument. You can have as many argument pairs as you like. The final argument, which is optional, represents the value to be returned if the input argument doesn't match any of the pairs.

The FISH table contains a death date column that can be used to illustrate the use of DECODE. Suppose that you are producing a report of fish, and that you want to know whether each fish is alive. The actual death date, if one exists, doesn't matter to you. You can use the DECODE function to translate the DEATH_DATE column from a date field to a text field containing either 'ALIVE' or 'DEAD', as shown in this example:

```
DECODE(death_date,NULL,'ALIVE','DEAD')
```

Note The DECODE function is one of the few functions that can properly test for nulls.

The input argument is DEATH_DATE. The first value pair is NULL, 'ALIVE'. If the DEATH_DATE is null, meaning that the fish hasn't died yet, the DECODE function will return the character string 'ALIVE'. If the date doesn't match any value pair — in this case, there is only one — then the default value of 'DEAD' will be returned. The following example shows this DECODE function being used in a query to retrieve a list of fish:

```
SQL> SELECT name_of_fish,
  2         DECODE(death_date,null,'ALIVE','DEAD')
  3  FROM fish;

NAME_OF_FI DECOD
---------- -----
Fish Two   DEAD
Fish Three DEAD
Fish Four  ALIVE
Wesley     ALIVE
```

If this is the first time you've seen the DECODE function being used, you may think it's a rather strange function. It is a strange function, but it's extremely useful. As you work more with Oracle, you'll see a number of creative applications for the DECODE function.

Using the INSTR function

The INSTR (short for INSTRING) function is a function that you apply to a character-datatype column. The INSTR function hunts down a text string or letter and tells you exactly where it starts within that column. If the text string you are searching for isn't found, the INSTR function will return a value of 0.

For example, you might want to find all the fish with comments referring to the fish named Wesley. You could do this using the following query:

```
   SELECT name_of_fish, comment_text
   FROM fish
   WHERE INSTR(comment_text,'Wesley') <> 0;
```

In this case, it doesn't matter where in the COMMENT_TEXT column the string occurs. All that matters is that it does occur. Any nonzero value indicates that. If you execute this query, the results will look like this:

```
NAME_OF_FI COMMENT_TEXT
---------- ----------------------------------------
Fish Two   Eaten by Wesley
Fish Three Eaten by Wesley
Fish Four  Died while I was on vacation, probably
           eaten by Wesley
```

Using the SUBSTR function

You can use the SUBSTR function to return a specific portion of a character column. The SUBSTR function takes three arguments. The first is the string itself. The second argument indicates the starting position. The third argument indicates the number of characters that you want to extract.

Following is an example showing how you can use the SUBSTR function. This is the same query that you saw in the previous section, but the SUBSTR function has been used to make the fish names display without "Fish" in front of each one:

```
SQL> SELECT SUBSTR(name_of_fish,6,5), comment_text
  2  FROM fish
  3  WHERE INSTR(comment_text,'Wesley') <> 0;

SUBST COMMENT_TEXT
----- ----------------------------------------
Two   Eaten by Wesley
Three Eaten by Wesley
Four  Died while I was on vacation, probably
      eaten by Wesley
```

As you can see, this query displays only the last five characters of each fish's name.

Using the concatenation operator

The concatenation operator is not really a function, but this seems as good a place as any to talk about it. You can use the concatenation operator to combine any two text strings into one. If you want to insert the words "The Honorable" in front of your fish's names, for example, you could use the concatenation operator, as shown in the following example:

```
SQL> SELECT 'The Honorable ' || name_of_fish
  2  FROM fish;

'THEHONORABLE'||NAME_OF_
------------------------
The Honorable Fish Two
The Honorable Fish Three
The Honorable Fish Four
The Honorable Wesley
```

When concatenating strings, take care to insert spaces between them so that your words don't run together. In this example, a space is appended to the word "Honorable."

Using the NVL function

The `NVL` function enables you to substitute a phrase, number, or date for a null value in a query. Here's an easy example:

```
   SELECT name_of_fish, NVL(sex,'Unknown') sex
   FROM fish;
```

In this example, when a fish has a null value for sex, the word "Unknown" will be returned in its place. The results look like this:

```
NAME_OF_FI SEX
---------- ----------
Fish Two   Female
Fish Three Male
Fish Four  Unknown
Wesley     Male
```

Notice that in this example, a column alias was used to give the `SEX` column back its original name. Otherwise, Oracle generates a name based on the expression that was used. If you're writing ad-hoc SQL*Plus reports, it's easier to set up column headings if you name columns that are the result of expressions.

Combining Query Results

A *union* combines the results of several queries into one result set. Unions are useful when you find it difficult to write just one query to get the results that you want. With unions, you can combine the results of two or more queries. SQL supports four types of unions. The operators for the four types are as follows:

`UNION`	Combines the results of two `SELECT` statements, and weeds out duplicate rows
`UNION ALL`	Combines the results of two `SELECT` statements, and doesn't weed out duplicate rows
`MINUS`	Returns rows that are from one `SELECT` statement that are not also returned by another
`INTERSECT`	Returns rows that are returned by each of two `SELECT` statements

Using the UNION and UNION ALL keywords

The `UNION` operator combines the results of two `SELECT` statements and weeds out duplicate rows at the same time. For example, the following `UNION` query returns a list of students who have either borrowed or reserved a book:

```
SELECT bl_student_id
FROM books_loaned
UNION
SELECT reserved_student_id
FROM books_reserved;
```

Because `UNION` is used in this example, no student will be listed twice in the results. If you don't want duplicate records to be eliminated, use `UNION ALL` instead.

Using the MINUS operator

The `MINUS` operator subtracts the results of one query from another. Remember the queries earlier in the chapter that produced a list of students who had never borrowed books? Here is yet one more approach to solving that problem, this time using the `MINUS` operator:

```
SQL> SELECT student_id
  2  FROM students
  3  MINUS
  4  SELECT bl_student_id
  5  FROM books_loaned;
```

```
STUDENT_ID
----------
         1
         7
        11
```

The first SELECT statement, from the STUDENTS table, provides a complete list of ID numbers. The second SELECT statement, from the BOOKS_LOANED table, provides a list of ID numbers for students who have borrowed books. The MINUS operator causes Oracle to look at the results of the first query and to delete any rows where those rows match a row returned by the second query. Oracle looks at all the columns in the row when it does this.

Using the INTERSECT operator

The INTERSECT operator causes Oracle to return rows only when they occur in both SELECT statements. The following query, for example, returns a list of students who have borrowed books:

```
SQL> SELECT student_id
  2  FROM students
  3  INTERSECT
  4  SELECT bl_student_id
  5  FROM books_loaned;

STUDENT_ID
----------
         5
```

As with the MINUS example presented earlier, the first query returns a list of all student ID numbers. The second query returns ID numbers only for students who have borrowed books. The INTERSECT operator causes Oracle to return rows represented only in both result sets.

Summary

In this chapter, you learned:

✦ Learning to write queries is a basic SQL skill. The same skills are used when writing other commands, such as INSERT and UPDATE. The SQL SELECT command is the mechanism for queries. The SELECT command has five basic components: the SELECT, FROM, WHERE, GROUP BY, and ORDER BY clauses.

- Oracle supports two types of joins. Inner joins return results only when both tables contain corresponding records. Outer joins can be used to make one table an optional table.
- You can use the `GROUP BY` clause to summarize data. Aggregate functions such as `COUNT` are used when doing this. You can also use the `HAVING` clause when summarizing data, to eliminate summary rows that you don't want to see in the results.
- Functions modify column contents within the context of the query. When used in a query, the underlying data doesn't change. You can use functions anywhere that you can use a column.
- You should develop your SQL queries systematically and test them each step of the way to be sure the results are what you expect.
- Unions allow you to combine the results of two `SELECT` statements. There are four types of unions: `UNION`, `UNION ALL`, `MINUS`, and `INTERSECT`. The `UNION` operator combines the rows from two `SELECT` statements and eliminates any duplicates. The `UNION ALL` operator performs the same functionbut without eliminating duplicates. The `MINUS` operator removes any rows that are returned by the second query. The `INTERSECT` operator returns only rows that are returned by both queries.

✦ ✦ ✦