PART 1

■■■

Foundations of Data Manipulation

CHAPTER 1

The Basics

This chapter presents lots of basic recipes to get you started—or rekindle old memories—on the core building blocks of SQL statements. We'll show you recipes for selecting, changing, and removing data from Oracle database tables, plus some common options you usually want to include when doing this kind of work.

Those of you with a firm grounding in SQL should feel free to delve into this chapter in an *à la carte* fashion. We've included one or two clever recipes at this early stage to make sure you get the most from Oracle SQL Recipes from the very first chapter. To continue the menu metaphor, feel free to consume the recipes herein in any order you like, and mix and match code fragments and techniques across the chapters to derive your own recipes.

Note We'll move through the basics quickly, and avoid such niceties as syntax diagrams and discussing every permutation of every recipe's options. Don't fret if you see commands and techniques you don't immediately understand. Try them out yourself in an Oracle test instance to get a feel for them, and remember that complementary books like *Begining Oracle SQL* (de Haan, Fink, Jørgensen, Morton) and *Beginning SQL Queries* (Churcher) help with learning SQL itself.

1-1. Retrieving Data from a Table

Problem

You want to retrieve specific row and column data from a table.

Solution

Issue a `SELECT` statement that includes a `WHERE` clause. The following is a straightforward `SELECT` statement querying a database table for particular column values, for those rows that match defined criteria:

```
select employee_id, first_name, last_name, hire_date, salary
from hr.employees
where department_id = 50
  and salary < 7500;
```

The SELECT statement returns 45 rows when run against an Oracle 11*g* database loaded with the sample HR schema. Here are the first 10 rows you will see.

```
EMPLOYEE_ID FIRST_NAME LAST_NAME   HIRE_DATE SALARY
----------- ---------- ----------- --------- ------
        198 Donald     OConnell    21-JUN-99   2600
        199 Douglas    Grant       13-JAN-00   2600
        123 Shanta     Vollman     10-OCT-97   6500
        124 Kevin      Mourgos     16-NOV-99   5800
        125 Julia      Nayer       16-JUL-97   3200
        126 Irene      Mikkilineni 28-SEP-98   2700
        127 James      Landry      14-JAN-99   2400
        128 Steven     Markle      08-MAR-00   2200
        129 Laura      Bissot      20-AUG-97   3300
        130 Mozhe      Atkinson    30-OCT-97   2800
…
```

How It Works

The SELECT statement was formatted to help you understand (or refresh your understanding of) the basic elements that constitute a working query. The first line of the query explicitly states the five columns we wish to include in the results.

```
select employee_id, first_name, last_name, hire_date, salary
```

The next line is the FROM clause, naming the table or tables to be referenced to pull out the columns we want to see:

```
from hr.employees
```

In this example, we use two-part object naming, specifying both the table name, EMPLOYEES, and the schema to which employees belongs, HR. This disambiguates the EMPLOYEES table in the HR schema from any other employees table that may be present in any other schema, and more importantly avoids the implicit selection of the table in the user's own schema or the currently set schema, which is the default where no schema is specified.

Next we list the WHERE clause, with two criteria that must be satisfied in order for rows to be included in our results.

```
where department_id = 50
  and salary < 7500
```

Rows in the EMPLOYEES table must satisfy both tests to be included in our results. Meeting one, or the other, but not both will not satisfy our query, as we've used the AND Boolean operator to combine the criteria. In non-technical terms, this means an employee must be listed in the department with an ID of 50, as well as have a salary below 7500.

Other Boolean operators are OR and NOT, and these follow normal Boolean precedence rules and can be modified using parenthesis to alter logic explicitly. We can modify our first example to illustrate these operators in combination

```
select employee_id, first_name, last_name, hire_date, salary
from hr.employees
where department_id = 50
  and (salary < 2500 or salary > 7500);
```

his query seeks the same columns as the first, this time focusing on those members of department 50 whose salary is either less than 2500, or more than 7500.

```
EMPLOYEE_ID FIRST_NAME LAST_NAME  HIRE_DATE SALARY
120         Matthew    Weiss      18/JUL/96 8000
121         Adam       Fripp      10/APR/97 8200
122         Payam      Kaufling   01/MAY/95 7900
127         James      Landry     14/JAN/99 2400
128         Steven     Markle     08/MAR/00 2200
132         TJ         Olson      10/APR/99 2100
135         Ki         Gee        12/DEC/99 2400
136         Hazel      Philtanker 06/FEB/00 2200
```

Only 8 rows in the sample HR schema match these criteria.

1-2. Selecting All Columns from a Table

Problem

You want to retrieve all columns from a table, but you don't want to take the time to type all the column names as part of your SELECT statement.

Solution

Use the asterisk (*) placeholder, to represent all columns for a table. For example:

```
select *
from hr.employees
where department_id = 50
  and salary < 7500;
```

Our results show wrapped lines and only a partial listing to save space, but you can see the effect of the asterisk on the columns selected:

```
EMPLOYEE_ID FIRST_NAME LAST_NAME EMAIL PHONE_NUMBER HIRE_DATE JOB_ID SALARY
 COMMISSION_PCT MANAGER_ID DEPARTMENT_ID
----------- ---------- --------- -------- ------------ ---------- --------- ------
 -------------- ---------- -------------
        198 Donald     OConnell  DOCONNEL 650.507.9833 21-JUN-99  SH_CLERK    2600
                       124            50
        199 Douglas    Grant     DGRANT   650.507.9844 13-JAN-00  SH_CLERK    2600
                       124            50
        123 Shanta     Vollman   SVOLLMAN 650.123.4234 10-OCT-97  ST_MAN      6500
                       100            50
```

How It Works

In SQL, the * is a shortcut that stands for all of the column names in a table. When Oracle's parser sees `SELECT *` as part of a query, the parser replaces the * with a list of all the possible column names, except those that have been marked hidden. Writing `SELECT *` is a quick way to build ad-hoc queries, because you avoid having to look up and type out all the correct column names.

Before you succumb to the temptation to use the asterisk for all your queries, it's important to remember that good design usually means explicitly naming only those columns in which you are interested. This results in better performance, as Oracle otherwise must read the system catalog to determine the fields to include in the query. Specifying explicit columns also protects you from problems in the future where columns may be added to a table. If you write code expecting the implicit order from using the asterisk, you could be in for a nasty surprise when that order changes unexpectedly, such as when a table is dropped and re-created in a different way, or a new column is added to a table. For the most part, the asterisk is best reserved for interactive querying, and ad-hoc analysis.

1-3. Sorting Your Results

Problem

Users want to see data from a query sorted in a particular way. For example, they would like to see employees sorted alphabetically by surname, and then first name.

Solution

Oracle uses the standard `ORDER BY` clause to allow you to sort the results of your queries.

```
select employee_id, first_name, last_name, hire_date, salary
from hr.employees
where salary > 5000
order by last_name, first_name;
```

The results are as follows.

```
EMPLOYEE_ID    FIRST_NAME    LAST_NAME    HIRE_DATE       SALARY
-----------    ----------    ---------    ---------       ------
174            Ellen         Abel         11/MAY/96       11000
166            Sundar        Ande         24/MAR/00        6400
204            Hermann       Baer         07/JUN/94       10000
167            Amit          Banda        21/APR/00        6200
172            Elizabeth     Bates        24/MAR/99        7300
151            David         Bernstein    24/MAR/97        9500
169            Harrison      Bloom        23/MAR/98       10000
148            Gerald        Cambrault    15/OCT/99       11000
154            Nanette       Cambrault    09/DEC/98        7500
110            John          Chen         28/SEP/97        8200
…
```

How It Works

The `ORDER BY` clause in this solution instructs Oracle to sort by the `LAST_NAME` column, and where values for `LAST_NAME` match, to then sort by the `FIRST_NAME` column. Oracle implicitly uses ascending ordering unless instructed otherwise, so numbers sort from zero to nine, letters from A to Z, and so on. You can explicitly control sorting direction with the `ASC` and `DESC` options for ascending and descending sorting. Here is an example:

```
select employee_id, first_name, last_name, hire_date, salary
from hr.employees
where salary > 5000
order by salary desc;
```

Our explicit descending sort on salary has these results:

```
EMPLOYEE_ID FIRST_NAME LAST_NAME HIRE_DATE SALARY
----------- ---------- --------- --------- ------
        100 Steven     King      17-JUN-87 24000
        102 Lex        De Haan   13-JAN-93 17000
        101 Neena      Kochhar   21-SEP-89 17000
        145 John       Russell   01-OCT-96 14000
        146 Karen      Partners  05-JAN-97 13500
…
```

1-4. Adding Rows to a Table

Problem

You need to add new rows of data to a table. For example, a new employee joins the company, requiring his data to be added to the `HR.EMPLOYEES` table.

Solution

Use the INSERT statement to add new rows to a table. To add a new row to a table, provide values for all mandatory columns, as well as any values for optional columns. Here's a statement to add a new employee:

```
insert into hr.employees
(employee_id, first_name, last_name, email, phone_number, hire_date, job_id,
 salary, commission_pct, manager_id, department_id)
values
(207, 'John ', 'Doe ', 'JDOE ', '650.555.8877 ', '25-MAR-2009 ', 'SA_REP ',
 3500, 0.25, 145, 80);
```

How It Works

The INSERT statement associates a list of values with a list of columns. It creates a row based upon that association, and inserts that row into the target table.

Oracle will check NULL constraints, as well as primary keys, foreign keys, and other defined constraints to ensure the integrity of your inserted data. See Chapter 10 for recipes on determining the state of constraints on your tables, and how they might affect inserting new data.

You can check which fields are mandatory, defined as not null, by examining the description of the table. You can do this from SQL Developer or SQL*Plus by issuing the DESCRIBE command, which you can abbreviate to DESC. For example:

```
desc hr.employees;

Name                           Null     Type
------------------------------ -------- ------------
EMPLOYEE_ID                    NOT NULL NUMBER(6)
FIRST_NAME                              VARCHAR2(20)
LAST_NAME                      NOT NULL VARCHAR2(25)
EMAIL                          NOT NULL VARCHAR2(25)
PHONE_NUMBER                            VARCHAR2(20)
HIRE_DATE                      NOT NULL DATE
JOB_ID                         NOT NULL VARCHAR2(10)
SALARY                                  NUMBER(8,2)
COMMISSION_PCT                          NUMBER(2,2)
MANAGER_ID                              NUMBER(6)
DEPARTMENT_ID                           NUMBER(4)

11 rows selected
```

You can write a shorter INSERT statement by not enumerating the list of column names, and providing data for *every* column in the right order for the current table definition. Here's an example:

```
insert into hr.employees
values
(208, 'Jane ', 'Doe ', 'JADOE ', '650.555.8866 ', '25-MAR-2009 ', 'SA_REP ',↩
 3500, 0.25, 145, 80)
```

■ **Caution** It is rarely, if ever, a good idea to omit the list of column names—and this is for some quite serious reasons. You have no idea what changes might be made to the table in future, and you make your SQL brittle to future schema changes by assuming an implicit column order. Perhaps the best example of what can go wrong is silent logical corruption. If someone rebuilds the underlying table with a different column order, but your INSERT statement passes data type and other checks, you could find yourself silently inserting data into the wrong columns, with disastrous consequences. Our strong recommendation is to always enumerate the columns in your INSERT statement.

1-5. Copying Rows from One Table to Another

Problem

You want to copy information from one table to another.

Solution

Use the INSERT statement with the SELECT option to copy data from one table to another. Suppose you have a table of candidates applying for jobs at your company, with many of the same details as the HR.EMPLOYEES table. This INSERT statement will insert into the HR.EMPLOYEES table based on a SELECT statement on the CANDIDATES table.

```
insert into hr.employees
(employee_id, first_name, last_name, email, phone_number, hire_date, job_id, salary, ↩
commission_pct, manager_id, department_id)
select 210, first_name, last_name, email, phone_number, sysdate, 'IT_PROG', 3500, ↩
NULL, 103, 60
from hr.candidates
where first_name = 'Susan'
  and last_name = 'Jones';
```

How It Works

This recipe seeds the values to be inserted into the HR.EMPLOYEES table with the results of a SELECT on a CANDIDATES table. The SELECT statement can be run by itself to see the data passed to the INSERT statement.

```
select 210, first_name, last_name, email, phone_number, sysdate, job_id, 3500, NULL, ↩
'IT_PROG', 103
from hr.candidates
where first_name = 'Susan'
  and last_name = 'Jones';
```

```
210 FIRST_NAME LAST_NAME EMAIL  PHONE_NUMBER SYSDATE   'IT_PRO 3500 N 103 60
--- ---------- --------- ------ ------------ --------- ------- ---- - --- --
210 Susan      Jones     SJONES 650.555.9876 30-MAR-09 IT_PROG 3500   103 60
```

We use literal (hard-coded) values for EMPLOYEE_ID, SALARY, and DEPARTMENT_ID. We also use the NULL place-holder to indicate that Susan has no COMMISSION_PCT—that is, she's not a sales person and therefore isn't part of a sales commission scheme. We want the HIRE_DATE to reflect the day on which we run the INSERT statement, so we use the built-in SYSDATE function to return the current system date.

1-6. Copying Data in Bulk from One Table to Another

Problem

You want to copy multiple rows from one table to another.

Solution

The INSERT INTO … SELECT … approach is capable of inserting multiple rows. The key is using desired criteria in the SELECT statement to return the rows you wish to insert. We can amend our previous recipe to handle multiple rows. Here's is an example of multi-row INSERT in action.

```
select candidate_id, first_name, last_name, email, phone_number, sysdate, job_id, 3500, ➥
NULL, 'IT_PROG', 103
from hr.candidates;
```

This recipe relies on the existence of the same HR.CANDIDATES table used in the previous recipe. If you're playing along at home, be sure you create this table first.

How It Works

This recipe also seeds the values to be inserted from the HR.CANDIDATES table. As there is no WHERE clause in the SELECT portion of the statement, all the rows in the CANDIDATES table will be selected, and thus all have equivalent rows inserted into the HR.EMPLOYEES table.

1-7. Changing Values in a Row

Problem

You want to change some of the data in one or more rows of a table.

Solution

The UPDATE statement is designed to change data, as its name suggests. For this problem, let's assume we want to increase the salaries of everyone in department 50 by five percent. This UPDATE statement achieves the change:

```
update hr.employees
set salary = salary * 1.05
where department_id = 50;
```

How It Works

The basic design of the UPDATE statement starts with the UPDATE clause itself

```
update hr.employess
```

This tells Oracle what table is to have its rows updated. The SET clause then specifies which columns are to change and how they are to be updated—either by a literal value, calculation, function result, subselect, or other method.

```
set salary = salary * 1.05
```

In this case, we're using a self-referencing calculation. The new SALARY value will be 1.05 times the existing value, relative to each row (that is, Oracle will perform this calculation for each row affected). Finally, the WHERE clause acts to provide the normal filtering predicates you're already familiar with from the SELECT statement. Only those rows that match a DEPARTMENT_ID of 50 will be affected.

1-8. Updating Multiple Fields with One Statement

Problem

You want to change multiple columns for one or more rows of a table.

Solution

The update statement is designed to update as many rows as you need in one statement. This means you can use multiple column = value clauses in the UPDATE statement to change as many fields as you wish in one statement. For example, to change the phone number, job role, and salary of James Marlow, EMPLOYEE_ID 131, we can use a single UPDATE statement.

```
update hr.employees
set job_id = 'ST_MAN',
  phone_number = '650.124.9876',
  salary = salary * 1.5
where employee_id = 131;
```

How It Works

Oracle evaluates the predicates of the UPDATE statement normally. In this case, it targets the row with the EMPLOYEE_ID of 131. It then changes each field based on the SET criteria given. It performs this action in one pass of the row.

Oracle also supports the grouping the columns and values for updates in parenthesis, in a similar fashion to other databases, but with one quirk.

```
update hr.employees
set (job_id,Phone_number,Salary)
 = (select 'ST_MAN','650.124.9876',salary * 1.5 from dual)
where employee_id = 131;
```

■ **Note** If the subquery returns no data, the values specified in the `set` clause will be set to null.

Note that the value group has to be specified as a well-formed SELECT—other databases would let you get away in this instance with the shortened form like this.

```
Set (job_id,Phone_number,Salary) = ('ST_MAN','650.124.9876',salary * 1.5)
```

This style of value grouping won't work with Oracle, so remember to use a well-formed SELECT.

1-9. Removing Unwanted Rows from a Table

Problem

An employee has taken a job with another company and you need to remove his details from the HR.EMPLOYEE table.

Solution

Let's assume that James Landry, EMPLOYEE_ID 127, has taken a job with another company. You can use the DELETE statement to remove James' details.

```
delete
from hr.employees
where employee_id = 127;
```

How It works

This recipe illustrates the basic DELETE statement, targeting the table HR.EMPLOYEES in the FROM clause. A WHERE clause provides the predicates to use to affect only those rows that match the specified criteria.

It's possible to construct the DELETE statement with no WHERE clause, in effect matching all rows of the table. In that case, all the table's rows are deleted.

```
delete
from hr.employees;
```

■ **Caution** If you do run either of those DELETE statements, don't forget to roll back if you want to continue using HR.EMPLOYEES data for the following recipes.

1-10. Removing All Rows from a Table

Problem

You want to remove all of the data for a given table.

Solution

Deleting all rows from a table using the DELETE statement allows Oracle to fully log the action, so that you can roll back if you issue this statement by accident. But that same logging, and the time it takes, means that using the DELETE statement is sometimes too slow for the desired result.

Oracle provides a complementary technique for removing all data from a table, called TRUNCATE. To truncate the HR.EMPLOYEES table, let's use the SQL statement:

```
truncate table hr.employees;
```

How It Works

No WHERE clause predicates or other modifiers are used when issuing the TRUNCATE statement. This statement is treated as DDL (Data Definition Language) SQL, which has other implications for transactions, such as implicitly committing other open transactions. See Chapter 9 for more details on recipes for transaction control and monitoring.

Using the TRUNCATE command also resets the high-water mark for a table, which means when Oracle does optimizer calculations that include judging the effect of the used capacity in a table, it will treat the table as if it had never had data consume any space.

1-11. Selecting from the Results of Another Query

Problem

You need to treat the results of a query as if they were the contents of a table. You don't want to store the intermediate results as you need them freshly generated every time you run your query.

Solution

Oracle's inline view feature allows a query to be included in the FROM clause of a statement, with the results referred to using a table alias.

```
select d.department_name
from
 (select department_id, department_name
  from hr.departments
  where location_id != 1700) d;
```

The results will look like this.

```
DEPARTMENT_NAME
----------------
Marketing
Human Resources
Shipping
IT
Public Relations
Sales

6 rows selected.
```

How It Works

This recipe treats the results of a SELECT statement on the HR.DEPARTMENTS table as an inline view, giving it the name “d”. Though providing an alias in this case is optional, it makes for more readable, testable, and standards-compliant SQL. You can then refer to the results of that statement as if it were a normal statement in almost all ways. Here are the results of the inline view's SELECT statement.

```
DEPARTMENT_ID DEPARTMENT_NAME
------------- -----------------
           20 Marketing
           40 Human Resources
           50 Shipping
           60 IT
           70 Public Relations
           80 Sales

6 rows selected.
```

The inline view now acts as if you had a table, D, defined with this structure.

```
Name             Null     Type
---------------- -------- -------------
DEPARTMENT_ID    NOT NULL NUMBER(4)
DEPARTMENT_NAME  NOT NULL VARCHAR2(30)
```

These are the definitions from the underlying HR.DEPARTMENTS table, which are implicitly inherited by our inline view. The outer SELECT statement queries the result set, just as if there were a real table d and you'd written this statement against it.

```
select department_name
from d;
```

1-12. Basing a Where Condition on a Query

Problem

You need to query the data from a table, but one of your criteria will be dependent on data from another table at the time the query runs—and you want to avoid hard-coding criteria. Specifically, you want to find all departments with offices in North America for the purposes of reporting.

Solution

In the HR schema, the DEPARTMENTS table lists departments, and the LOCATIONS table lists locations.

```
select department_name
from hr.departments
where location_id in
  (select location_id
   from hr.locations
   where country_id = 'US'
     or country_id = 'CA');
```

Our results are as follows:

```
DEPARTMENT_NAME
---------------
IT
Shipping
Administration
Purchasing
Executive
…
```

How It Works

The right-hand side of our in predicate reads the results of the sub-SELECT to provide values to drive the comparison with the LOCATION_ID values from the HR.DEPARTMENTS table. You can see what values are generated by running the sub-SELECT query itself.

```
select location_id
from hr.locations
where country_id = 'US'
  or country_id = 'CA';
```

Here are the results:

```
LOCATION_ID
-----------
       1400
       1500
       1600
       1700
       1800
       1900

6 rows selected.
```

The outer SELECT compares LOCATION_ID values against this dynamically queried list. It is similar to running this static query.

```
select department_name
from hr.departments
where location_id in (1400,1500,1600,1700,1800,1900);
```

The key advantage of this recipe is that, should we open new offices and change all our LOCATION_ID's for North America, we don't have to rewrite our query: the sub-select will output the necessary new values dynamically.

1-13. Finding and Eliminating NULLs in Queries

Problem

You need to report how many of your employees have a commission percentage as part of their remuneration, together with the number that get only a fixed salary. You track this using the COMMISSION_PCT field of the HR.EMPLOYEES table.

Solution

The structure of the HR.EMPLOYEE tables allows the COMMISSION_PCT to be NULL. Two queries can be used to find those whose commission percent is NULL and those whose commission percent is non-NULL. First, here's the query that finds employees with NULL commission percent:

```
select first_name, last_name
from hr.employees
where commission_pct is null;

FIRST_NAME           LAST_NAME
-------------------- -------------------------
Donald               OConnell
Douglas              Grant
Jennifer             Whalen
Michael              Hartstein
Pat                  Fay
…

72 rows selected.
```

Now here's the query that finds non-NULL commission-percentage holders:

```
select first_name, last_name
from hr.employees
where commission_pct is not null;

FIRST_NAME           LAST_NAME
-------------------- -------------------------
John                 Russell
Karen                Partners
Alberto              Errazuriz
Gerald               Cambrault
Eleni                Zlotkey
…

35 rows selected.
```

How It Works

Our first SELECT statement uses the COMMISSION_PCT IS NULL clause to test for NULL entries. This has only two outcomes: either the column has a NULL entry, thus possessing no value, and satisfies the test; or it has some value.

The second statement uses the COMMISSION_PCT IS NOT NULL clause, which will find a match for any employee with an actual value for COMMISSION_PCT.

Oracle's Non-standard Treatment of the Empty String

Oracle deviates from the SQL standard in implicitly treating an empty, or zero-length, string as a surrogate for NULL. This is for a range of historical and pragmatic reasons, but it's important to remember. Almost all other implementations of SQL treat the empty string as a separate, known value.

Those of you with a programming background will find analogous the idea of a zero length string being well defined, with the memory for the string having a string terminator (\0) as its only component. In contrast, an uninstantiated string has no known state … not even a terminator. You wouldn't use zero-length and uninstantiated strings interchangeably, but this is analogous to what Oracle does with NULLs.

NULL Has No Equivalents

One aspect of recipes (and indeed day-to-day work) involving NULL in SQL often stumps people. SQL expressions are tri-valued, meaning every expression can be true, false, or NULL. This affects all kinds of comparisons, operators, and logic as you've already seen. But a nuance of this kind of logic is occasionally forgotten, so we'll repeat it explicitly. NULL has no equivalents. No other value is the same as NULL, *not even other NULL values*. If you run the following query, can you guess your results?

```
select first_name, last_name
from hr.employees
where commission_pct = NULL;
```

The answer is no rows will be selected. Even though you saw from the above SELECT statement in this recipe that 72 employees have a NULL `COMMISSION_PCT`, no NULL value equals another, so the `COMMISSION_PCT = NULL` criterion will never find a match, and you will never see results from this query. Always use `IS NULL` and `IS NOT NULL` to find or exclude your NULL values.

1-14. Sorting as a Person Expects

Problem

Your textual data has been stored in a mix of uppercase, lowercase, and sentence case. You need to sort this data alphabetically as a person normally would, in a case-insensitive fashion.

Solution

To introduce some mixed case data into our `HR.EMPLOYEES` table, let's run the following `UPDATE` to uppercase William Smith's last name.

```
update hr.employees
set last_name = 'SMITH'
where employee_id = 171;
```

This select statement shows Oracle’s default sorting of employee last names.

```
select last_name
from hr.employees
order by last_name;

LAST_NAME
---------
…
Rogers
Russell
SMITH
Sarchand
Sciarra
Seo
Sewall
Smith
Stiles
Sullivan
…
```

Astute readers will have anticipated these results. Oracle, by default, sorts using a binary sort order. This means that in a simple example like this one, text is sorted according to the numeric equivalent on the code page in use (US7ASCII, WEISO8859P1, and so on). In these code pages, upper- and lowercase letters have different values, with uppercase coming first. This is why the uppercase SMITH has sorted before all other names starting with a capital S.

What most people would expect to see are the two “Smith” values sorted together, regardless of case. This `NLS` directive achieves that result.

```
alter session set NLS_SORT='BINARY_CI';

select last_name
from hr.employees
order by last_name;

LAST_NAME
---------
…
Rogers
Russell
Sarchand
Sciarra
Seo
Sewall
Smith
SMITH
Stiles
Sullivan
…
```

How It Works

Oracle supports both case-sensitive and case-insensitive sort orders. By default, you operate in a case-sensitive sort environment called BINARY. For every such sort order, an equivalent insensitive order exists using a suffix of _CI. We changed to using the BINARY_CI sort order, and reran the query to see results in the order a normal user would expect.

As the name suggests, the NLS_SORT option affects sorting only. It doesn't affect some other aspects of case sensitivity. With NLS_SORT='BINARY_CI', attempts to compare data in a case-insensitive fashion still exhibit Oracle's default behavior.

```
select first_name, last_name
from hr.employees
where last_name like 's%';

no rows selected
```

Don't despair. Oracle provides a similar option to allow case-insensitive comparisons just like this.

■ **Tip** A more traditional approach tackles this problem without changing any NLS parameters. You can cast the column name, literals, or both to upper- or lowercase for comparison using Oracle's UPPER and LOWER functions. This has the disadvantage of preventing the optimizer from using standard indexes, but you can also create function-based indexes to counter this.

1-15. Enabling Other Sorting and Comparison Options

Problem

You need to perform case-insensitive comparisons and other sorting operations on textual data that has been stored in an assortment of uppercase, lowercase, and sentence case.

Solution

By activating Oracle's linguistic comparison logic, you can use the same statement to retrieve the data a human would expect to see, without the burden of artificial case sensitivity.

```
alter session set NLS_COMP='LINGUISTIC';

select first_name, last_name
from hr.employees
where last_name = 'smith';
```

```
FIRST_NAME           LAST_NAME
-------------------- -------------------------
William              SMITH
Lindsey              Smith
```

This recipe relies on the same case-altering statements used in the previous recipes. Be sure to run those statements prior to testing this recipe in action.

How It Works

Historically, there were two ways to treat Oracle's normal case-sensitive handling of text when designing applications. One approach was to design the application logic to ensure data would be stored in a consistent fashion, such as the initial-caps data you see in the `HR.EMPLOYEES` table we've used in several recipes. The other approach allowed data to be stored as users liked, with database and application logic used to hide any case differences from users where they wouldn't be expected.

This once took a great deal of effort, but you can now achieve it with remarkably little fuss. The following statement will retrieve no rows, because no `LAST_NAME` entered in the `HR.EMPLOYEES` table is recorded in lowercase.

```
Select first_name, last_name
From hr.employees
Where last_name = 'smith';

no rows selected
```

But our recipe manages to return the right data because it changes the session settings, instructing Oracle to perform comparisons in linguistic fashion. Neither version of *Smith* is stored in the lowercase form we specified in the query, but the `NLS_COMP` parameter controls comparison and other behaviors in Oracle, and set to `LINGUISTIC` induces what the typical person would consider normal comparison behavior.

1-16. Conditional Inserting or Updating Based on Existence

Problem

You want to insert rows into a table with a key identifier that may already be present. If the identifier is not present, a new row should be created. If the identifier is present, the other columns for that row should be updated with new data, rather than a new row created.

Solution

The `MERGE` statement provides the ability to insert new data into a table, and if the proposed new primary key does not already exist in the table, a newly created row is inserted. If the primary key matches an existing row in a table, the statement instead updates that row with the additional details matching that key.

For our recipe, we'll assume the HR.COUNTRIES table is to be loaded with amended country details sourced from a NEW_COUNTRIES table.

```
merge into hr.countries c
using
  (select country_id, country_name
   from hr.new_countries) nc
on (c.country_id = nc.country_id)
when matched then
  update set c.country_name = nc.country_name
when not matched then
  insert (c.country_id, c.country_name)
  values (nc.country_id, nc.country_name);
```

How It Works

Rather than simply inserting the source data from the HR.NEW_COUNTRIES table directly into the target HR.COUNTRIES table—and potentially failing on a primary key duplication error—the MERGE statement sets up a logic branch to handle matched and unmatched rows based on the ON clause.

Typically, the ON clause specifies how to match primary or unique key data between the source and target. In this recipe, that's the matching of COUNTRY_ID values like this.

```
on (c.country_id = nc.country_id)
```

This is followed by two additional clauses, the WHEN MATCHED THEN clause for values that match the ON clause, and the WHEN NOT MATCHED THEN clause, for unmatched new rows that need to be treated as new data to be inserted.

The matched and not-matched clauses can also include further filtering criteria, and even criteria that when satisfied result in rows being deleted.

```
merge into hr.countries c
using
  (select country_id, country_name, region_id
   from hr.new_countries) nc
on (c.country_id = nc.country_id)
when matched then
  update set c.country_name = nc.country_name,
    c.region_id = nc.region_id
  delete where nc.region_id = 4
when not matched then
  insert (c.country_id, c.country_name, c.region_id)
  values (nc.country_id, nc.country_name, nc.region_id)
  where (nc.region_id != 4);
```

In this modified version of the recipe, matched rows will have their COUNTRY_NAME updated unless the REGION_ID of the new data is equal to 4, in which case the row in HR.COUNTRIES will ultimately be deleted. Unmatched rows will be inserted into HR.EMPLOYEES unless their REGION_ID is 4, in which case they will be ignored.

CHAPTER 2

Summarizing and Aggregating Data

In this chapter we'll introduce recipes for working with your data at a higher level, where grouping, summaries, and the bigger picture are important. Many of the recipes we'll explore cover common or tricky reporting scenarios, the kind you encounter in a business or professional setting. Most organizations ask for reports about who sold what to whom, how many people, sales, or activities happened in a given time frame, trends across regions or customer groups, and the like.

Many people are familiar with some of the basic methods Oracle provides for performing summaries and aggregates. But often developers and DBAs will try to execute more complex calculations in their applications—and tie themselves in knots. You can spot this in your own work if you see any combination of duplicating database capabilities in code, shuffling temporary data and results around needlessly, and similar less-than-efficient tasks.

Oracle is exploding with this kind of functionality now, especially with the introduction of On-Line Analytical Processing capabilities (OLAP) over the last few major versions. After you've explored these recipes for summarizing and aggregating data, you'll realize that Oracle is the number one tool at your disposal to satisfy a huge range of reporting and other requirements.

2-1. Summarizing the Values in a Column

Problem

You need to summarize data in a column in some way. For example, you have been asked to report on the average salary paid per employee, as well as the total salary budget, number of employees, highest and lowest earners, and more.

Solution

You don't need to calculate a total and count the number of employees separately to determine the average salary. The AVG function calculates average salary for you, as shown in the next SELECT statement.

```
select avg(salary)
from hr.employees;

AVG(SALARY)
-----------
 6473.36449
```

Note there is no WHERE clause in our recipe, meaning all rows in the HR.EMPLOYEES table are assessed to calculate the overall average for the table's rows.

Functions such as AVG are termed *aggregate functions*, and there are many such functions at your disposal. For example, to calculate the total salary paid, use the SUM function, as shown here:

```
select sum(salary)
from hr.employees;

SUM(SALARY)
-----------
     692650
```

To tally the number of people receiving a salary, you can simply count the number of rows in the table using the COUNT function.

```
Select count(salary)
From hr.employees;

COUNT(SALARY)
-------------
          107
```

Maximum and minimum values can be calculated using the MAX and MIN functions. By now you're probably thinking Oracle uses very simple abbreviated names for the statistical functions, and by and large you are right. The MAX and MIN functions are shown next.

```
select min(salary), max(salary)
from hr.employees;

MIN(SALARY) MAX(SALARY)
----------- -----------
       2100       24000
```

How It Works

Oracle has numerous built-in statistical and analytic functions for performing common summarizing tasks, such as average, total, and minimum and maximum value. There's no need for you to manually perform the intermediate calculations, though you can do so if you want to confirm Oracle's arithmetic.

The following statement compares Oracle's average calculation for salary with our own explicit total divided by the number of employees.

```
select avg(salary), sum(salary)/count(salary)
from hr.employees;

AVG(SALARY) SUM(SALARY)/COUNT(SALARY)
----------- -------------------------
 6473.36449                6473.36449
```

It's pleasing to know Oracle gets this right. To complete the picture on Oracle's aggregation capabilities, we need to consider what happens when our data includes NULL data. Our current recipe aggregates employee's salaries, and it so happens that every employee has a salary. But only sales people have a commission for their sales efforts, reflected in the HR.EMPLOYEES table by a value in the COMMISSION_PCT. Non-sales staff have no commission, reflected by a NULL value in COMMISSION_PCT. So what happens when we try to average or count the COMMISSION_PCT values? The next SQL statement shows both of these aggregates.

```
select count(commission_pct), avg(commission_pct)
from hr.employees;

COUNT(COMMISSION_PCT) AVG(COMMISSION_PCT)
--------------------- -------------------
                   38                .225
```

Even though we saw 107 employees with a salary, the COUNT function has ignored all NULL values for COMMISSION_PCT, tallying only the 38 employees with a commission. Equally, when calculating the average for the employees' commissions, Oracle has again only considered those rows with a real value, ignoring the NULL entries.

There are only two special cases where Oracle considers NULL values in aggregate functions. The first is the GROUPING function, used to test if the results of an analytic function that includes NULL values generated those values directly from rows in the underlying table or as a final aggregate "NULL set" from the analytic calculation. The second special case is the COUNT(*)function. Because the asterisk implies all columns within the table, Oracle handles the counting of rows independently of any of the actual data values, treating NULL and normal values alike in this case.

To illustrate, the next SQL statement shows the difference between COUNT(*) and COUNT(COMMISSION_PCT) side by side.

```
select count(*), count(commission_pct)
from hr.employees;

  COUNT(*) COUNT(COMMISSION_PCT)
---------- ---------------------
       107                    38
```

Our use of COUNT(*) tallies all rows in the table, whereas COUNT(COMMISSION_PCT) counts only the non-NULL values for COMMISSION_PCT.

2-2. Summarizing Data for Different Groups

Problem

You want to summarize data in a column, but you don't want to summarize over all the rows in a table. You want to divide the rows into groups, and then summarize the column separately for each group. For example, you need to know the average salary paid per department.

Solution

Use SQL's `GROUP BY` feature to group common subsets of data together to apply functions like `COUNT`, `MIN`, `MAX`, `SUM`, and `AVG`. This SQL statement shows how to use an aggregate function on subgroups of your data with `GROUP BY`.

```
select department_id, avg(salary)
from hr.employees
group by department_id;
```

Here are the results showing averages by `DEPARTMENT_ID`.

```
DEPARTMENT_ID AVG(SALARY)
------------- -----------
          100        8600
           30        4150
                     7000
           20        9500
           70       10000
           90  19333.3333
          110       10150
           50  3503.33333
           40        6500
           80  8955.88235
           10        4400
           60        5760

12 rows selected.
```

Note that the third result row indicates a null `DEPARTMENT_ID`.

How It Works

The `GROUP BY` clause determines what groups the target table's rows should be put into for any subsequence aggregate functions. For performance reasons, Oracle will implicitly sort the data to match the grouping desired. From the first line of the `SELECT` statement, you can normally glean the columns that will be required in the `GROUP BY` clause.

```
select department_id, avg(salary)
```

We've told Oracle that we want to aggregate individual salary data into an average, but we haven't told it what to do with the individual DEPARTMENT_ID values for each row. Should we show every DEPARTMENT_ID entry, including duplicates, with the average against each one? Obviously, this would result in wasted, duplicate output—and also leave you wondering if there was something more complex you needed to understand. By using the "unaggregated" fields in the GROUP BY clause, we instruct Oracle how to collapse, or group, the singular row values against the aggregated values it has calculated.

```
group by department_id
```

This means that all values in our SELECT statement are either aggregates or covered by the GROUP BY clause. The key to writing syntactically correct GROUP BY statements is to always remember, values are either grouped or aggregated—no "stragglers" are allowed. You can see this clearly in a query that groups by multiple values.

2-3. Grouping Data by Multiple Fields

Problem

You need to report data grouped by multiple values simultaneously. For example, an HR department may need to report on minimum, average, and maximum SALARY by DEPARTMENT_ID and JOB_ID.

Solution

Oracle's GROUP BY capabilities extend to an arbitrary number of columns and expressions, so we can extend the previous recipe to encompass our new grouping requirements. We know what we want aggregated: the SALARY value aggregated three different ways. That leaves the DEPARTMENT_ID and JOB_ID to be grouped. We also want our results ordered so we can see different JOB_ID values in the same department in context, from highest SALARY to lowest. The next SQL statement achieves this by adding the necessary criteria to the GROUP BY and ORDER BY clauses.

```
Select department_id, job_id, min(salary), avg(salary), max(salary)
From hr.employees
Group by department_id, job_id
Order by department_id, max(salary) desc;

DEPARTMENT_ID JOB_ID     MIN(SALARY) AVG(SALARY) MAX(SALARY)
------------- ---------- ----------- ----------- -----------
           10 AD_ASST           4400        4400        4400
           20 MK_MAN           13000       13000       13000
           20 MK_REP            6000        6000        6000
           30 PU_MAN           11000       11000       11000
           30 PU_CLERK          2500        2780        3100
…

20 rows selected.
```

How It Works

When ordering by aggregate or other functions, you can take advantage of Oracle's shorthand notation for ordering columns. Thus you can write the statement to order by the column-based positions of your data, rather than having to write the cumbersome full text of the aggregate expression.

```
Select department_id, job_id, min(salary), avg(salary), max(salary)
From hr.employees
Group by department_id, job_id
Order by 1, 5 desc;
```

You can mix column names, aggregate expressions, numeric positions, and even column aliases in the SELECT clause within your ordering when working with grouped results, as shown in the next SQL statement.

```
Select department_id, job_id, min(salary), avg(salary), max(salary) Max_Sal
From hr.employees
Group by department_id, job_id
Order by 1, job_id, Max_Sal desc;
```

Flexibility is the key here, and as you create and use more complex expressions for ordering and grouping, you'll find aliases and ordinal notation helpful. However, for readability's sake, you should try to stay consistent.

2-4. Ignoring Groups in Aggregate Data Sets

Problem

You want to ignore certain groups of data based on the outcome of aggregate functions or grouping actions. In effect, you'd really like another WHERE clause to work after the GROUP BY clause, providing criteria at the group or aggregate level.

Solution

SQL provides the HAVING clause to apply criteria to grouped data. For our recipe, we solve the problem of finding minimum, average, and maximum salary for people performing the same job in each of the departments in the HR.EMPLOYEES table. Importantly, we only want to see these aggregate values where more than one person performs the same job in a given department. The next SQL statement uses an expression in the HAVING clause to solve our problem.

```
select department_id, job_id, min(salary), avg(salary), max(salary), count(*)
from hr.employees
group by department_id, job_id
having count(*) > 1;
```

Our recipe results in the following summary.

```
DEPARTMENT_ID JOB_ID     MIN(SALARY) AVG(SALARY) MAX(SALARY)   COUNT(*)
------------- ---------- ----------- ----------- ----------- ----------
           90 AD_VP            17000       17000       17000          2
           50 ST_CLERK          2100        2800        3600         19
           80 SA_REP            6100  8396.55172       11500         29
           50 ST_MAN            3750  6691.66667        8200          6
           80 SA_MAN           10500       12200       14000          5
           50 SH_CLERK          2500        3215        4200         20
           60 IT_PROG           4200        5760        9000          5
           30 PU_CLERK          2500        2780        3100          5
          100 FI_ACCOUNT        6900        7920        9000          5

9 rows selected.
```

How It Works

You can immediately see that we have results for only nine groups of employees. Compared with the 20 groups returned in the previous recipe, it's obvious the HAVING clause has done something—but what?

The HAVING clause is evaluated after all grouping and aggregation has taken place. Internally, Oracle will first generate results like this.

```
DEPARTMENT_ID JOB_ID     MIN(SALARY) AVG(SALARY) MAX(SALARY)   COUNT(*)
------------- ---------- ----------- ----------- ----------- ----------
          110 AC_ACCOUNT        8300        8300        8300          1
           90 AD_VP            17000       17000       17000          2
           50 ST_CLERK          2100        2800        3600         19
           80 SA_REP            6100  8396.55172       11500         29
          110 AC_MGR           12000       12000       12000          1
…
```

This, in effect, is our recipe without the HAVING clause. Oracle then applies the HAVING criteria

```
having count(*) > 1
```

The bolded rows in the previous results have a count of 1 (there's only one employee of that JOB_ID in the respective DEPARTMENT_ID), which means they fail the HAVING clause criterion and are excluded from the final results, leaving us with the solution we saw above.

The HAVING clause criteria can be arbitrarily complex, so you can use multiple criteria of different sorts.

```
select department_id, job_id, min(salary), avg(salary), max(salary), count(*)
from hr.employees
group by department_id, job_id
having count(*) > 1
and min(salary) between 2500 and 17000
and avg(salary) != 5000
and max(salary)/min(salary) < 2
;
```

```
DEPARTMENT_ID JOB_ID     MIN(SALARY) AVG(SALARY) MAX(SALARY)   COUNT(*)
------------- ---------- ----------- ----------- ----------- ----------
           90 AD_VP            17000       17000       17000          2
           80 SA_REP            6100  8396.55172       11500         29
           80 SA_MAN           10500       12200       14000          5
           50 SH_CLERK          2500        3215        4200         20
           30 PU_CLERK          2500        2780        3100          5
          100 FI_ACCOUNT        6900        7920        9000          5

6 rows selected.
```

2-5. Aggregating Data at Multiple Levels

Problem

You want to find totals, averages, and other aggregate figures, as well as subtotals in various dimensions for a report. You want to achieve this with as few statements as possible, preferably just one, rather than having to issue separate statements to get each intermediate subtotal along the way.

Solution

You can calculate subtotals or other intermediate aggregates in Oracle using the CUBE, ROLLUP and grouping sets features. For this recipe, we'll assume some real-world requirements. We want to find average and total (summed) salary figures by department and job category, and show meaningful higher-level averages and subtotals at the department level (regardless of job category), as well as a grand total and company-wide average for the whole organization.

```
select department_id, job_id, avg(salary), sum(salary)
from hr.employees
group by rollup (department_id, job_id);
```

Our results (partial output shown) include sum and average by DEPARTMENT_ID and JOB_ID, rolled-up aggregates by DEPARTMENT_ID, and grand totals across all data.

```
DEPARTMENT_ID JOB_ID     AVG(SALARY) SUM(SALARY)
------------- ---------- ----------- -----------
…
           80 SA_MAN           12200       61000
           80 SA_REP      8396.55172      243500
           80             8955.88235      304500
           90 AD_VP            17000       34000
           90 AD_PRES          24000       24000
           90             19333.3333       58000
          100 FI_MGR           12000       12000
          100 FI_ACCOUNT        7920       39600
          100                   8600       51600
          110 AC_MGR             12000        12000
          110 AC_ACCOUNT          8300         8300
          110                    10150        20300
                            6473.36449       692650

33 rows selected.
```

How It Works

The ROLLUP function performs grouping at multiple levels, using a right-to-left method of rolling up through intermediate levels to any grand total or summation. In our recipe, this means that after performing normal grouping by DEPARTMENT_ID and JOB_ID, the ROLLUP function rolls up all JOB_ID values so that we see an average and sum for the DEPARTMENT_ID level across all jobs in a given department. ROLLUP then rolls up to the next (and highest) level in our recipe, rolling up all departments, in effect providing an organization-wide rollup. You can see the rolled up rows in bold in the output.

Performing this rollup would be the equivalent of running three separate statements, such as the three that follow, and using UNION or application-level code to stitch the results together.

```
select department_id, job_id, avg(salary), sum(salary)
from hr.employees
group by department_id, job_id;
select department_id, avg(salary), sum(salary)
from hr.employees
group by department_id;
select avg(salary), sum(salary)
from hr.employees;
```

Of course, in doing this, you're responsible for your own interspersing of subtotals at the intuitive points in the output. You could try writing a three-way UNION subselect with an outer SELECT to do the ordering. If this sounds more and more complicated, be thankful the ROLLUP command, and its associated command CUBE, have displaced the need to perform such awkward computations.

Careful observers will note that because ROLLUP works from right to left with the columns given, we don't see values where departments are rolled up by job. We could achieve this using this version of the recipe.

```
select department_id, job_id, avg(salary), sum(salary)
from hr.employees
group by rollup (job_id, department_id);
```

In doing so, we get the DEPARTMENT_ID intermediate rollup we want, but lose the JOB_ID intermediate rollup, seeing only JOB_ID rolled up at the final level. To roll up in all dimensions, change the recipe to use the CUBE function.

```
Select department_id, job_id, min(salary), avg(salary), max(salary)
From hr.employees
Group by cube (department_id, job_id);
```

The results show our rollups at each level, shown in bold in the partial results that follow.

```
DEPARTMENT_ID JOB_ID     MIN(SALARY) AVG(SALARY) MAX(SALARY)
------------- ---------- ----------- ----------- -----------
                                7000        7000        7000
                                2100  6392.27273       24000
              AD_VP            17000       17000       17000
              AC_MGR           12000       12000       12000
              FI_MGR           12000       12000       12000
...
           10                   4400        4400        4400
           10 AD_ASST           4400        4400        4400
           20                   6000        9500       13000
           20 MK_MAN           13000       13000       13000
           20 MK_REP            6000        6000        6000
...
```

The power of both the ROLLUP and CUBE functions extends to as many “dimensions” as you need for your query. Admittedly, the term *cube* is meant to allude to the idea of looking at intermediate aggregations in three dimensions, but your data can often have more dimensions than that. Extending our recipe, we could “cube” a calculation of average salary by department, job, manager, and starting year.

```
Select department_id, job_id, manager_id,
   extract(year from hire_date) as "START_YEAR", avg(salary)
From hr.employees
Group by cube (department_id, job_id,  manager_id, extract(year from hire_date));
```

This recipe results in an examination of average salary in four dimensions!

2-6. Using Aggregate Results in Other Queries

Problem

You want to use the output of a complex query involving aggregates and grouping as source data for another query.

Solution

Oracle allows any query to be used as a subquery or inline view, including those containing aggregates and grouping functions. This is especially useful where you’d like to specify group-level criteria compared against data from other tables or queries, and don’t have a ready-made view available.

For our recipe, we’ll use an average salary calculation with rollups across department, job, and start year, as shown in this SELECT statement.

```
select * from (
  select department_id as "dept", job_id as "job", to_char(hire_date,'YYYY') as
  "Start_Year", avg(salary) as "avsal"
  from hr.employees
  group by rollup (department_id, job_id, to_char(hire_date,'YYYY'))) salcalc
where salcalc.start_year > '1990'
or salcalc.start_year is null
order by 1,2,3,4;
```

Our recipe results in the following (abridged) output:

```
      dept job        Start      avsal
---------- ---------- ----- ----------
        10 AD_ASST                4400
        10                        4400
        20 MK_MAN      1996      13000
        20 MK_MAN                13000
        20 MK_REP      1997       6000
        20 MK_REP                 6000
…
                            6473.36449
                                  7000

79 rows selected.
```

How It Works

Our recipe uses the aggregated and grouped results of the subquery as an inline view, which we then select from and apply further criteria. In this case, we could avoid the subquery approach by using a more complex HAVING clause like this.

```
having to_char(hire_date,'YYYY') > '1990'
or to_char(hire_date,'YYYY') is null
```

Avoiding a subquery here works only because we're comparing our aggregates with literals. If we wanted to find averages for jobs in departments where someone had previously held the job, we'd need to reference the HR.JOBHISTORY table. Depending on the business requirement, we might get lucky and be able to construct our join, aggregates, groups, and having criteria in one statement. By treating the results of the aggregate and grouping query as input to another query, we get better readability, and the ability to code even more complexity than the HAVING clause allows.

2-7. Counting Members in Groups and Sets

Problem

You need to count members of a group, groups of groups, and other set-based collections. You also need to include and exclude individual members and groups dynamically based on other data at the same time. For instance, you want to count how many jobs each employee has held during their time at the organization, based on the number of promotions they've had within the company.

Solution

Oracle's COUNT feature can be used to count materialized results as well as actual rows in tables. The next SELECT statement uses a subquery to count the instances of jobs held across tables, and then summarizes those counts. In effect, this is a count of counts against data resulting from a query, rather than anything stored directly in Oracle.

```
select jh.JobsHeld, count(*) as StaffCount
from
  (select u.employee_id, count(*) as JobsHeld
   from
    (select employee_id from hr.employees
     union all
     select employee_id from hr.job_history) u
   group by u.employee_id) jh
group by jh.JobsHeld;
```

From that SELECT statement, we get the following concise summary.

```
  JOBSHELD STAFFCOUNT
---------- ----------
         1         99
         2          5
         3          3
```

Most staff have had only one job, five have held two positions, and three have held three positions each.

How It Works

The key to our recipe is the flexibility of the COUNT function, which can be used for far more than just physically counting the number of rows in a table. You can count anything you can represent in a *result*. This means you can count derived data, inferred data, and transient calculations and determinations. Our recipe uses nested subselects and counts at two levels, and is best understood starting from the inside and working out.

We know an employee's current position is tracked in the HR.EMPLOYEES table, and that each instance of previous positions with the organization is recorded in the HR.JOB_HISTORY table. We can't just count the entries in HR.JOB_HISTORY and add one for the employees' current positions, because staff who have never changed jobs don't have an entry in HR.JOB_HISTORY.

Instead, we perform a UNION ALL of the EMPLOYEE_ID values across both HR.EMPLOYEES and HR.JOB_HISTORY, building a basic result set that repeats an EMPLOYEE_ID for every position an employee has held. Partial results of just the inner UNION ALL statement are shown here to help you follow the logic.

```
EMPLOYEE_ID
-----------
        100
        101
        101
        101
        102
        102
        103
…
```

Union vs. Union All

It's useful to remember the difference between `UNION` and `UNION ALL`. `UNION` will remove duplicate entries in result sets (and perform a sort on the data as part of deduplication), whereas `UNION ALL` preserves all values in all source sets, even the duplicates. In our recipe, the use of `UNION` would have resulted in a count of one job for every employee, regardless of how many promotions or jobs they'd actually had.

You'll be pleased to know `UNION` operators are useful ingredients for many recipes in other chapters of the book.

The next subselect aggregates and groups the values derived in our innermost subselect, counting the occurrences of each `EMPLOYEE_ID` to determine how many jobs each person has held. This is the first point where we use the `COUNT` function on the results of another query, rather than raw data in a table. Partial output at this subselect would look like this.

```
EMPLOYEE_ID   JOBSHELD
----------- ----------
        100          1
        101          3
        102          2
        103          1
        104          1
…
```

Our outermost query also performs straightforward aggregation and grouping, once again employing the `COUNT` function on the results of a subselect—which itself was producing counts of derived data.

2-8. Finding Duplicates and Unique Values in a Table

Problem

You need to test if a given data value is unique in a table—that is, it appears only once.

Solution

Oracle supports the standard HAVING clause for SELECT statements, and the COUNT function, which together can identify single instances of data in a table or result. The following SELECT statement solves the problem of finding if the surname Fay is unique in the HR.EMPLOYEES table.

```
select last_name, count(*)
from hr.employees
where last_name = 'Fay'
group by last_name
having count(*) = 1;
```

With this recipe, we receive these results:

```
LAST_NAME                   COUNT(*)
------------------------- ----------
Fay                                1
```

Because there is exactly one LAST_NAME value of Fay, we get a count of 1 and therefore see results.

How It Works

Only unique combinations of data will group to a count of 1. For instance, we can test if the surname King is unique:

```
select last_name, count(*)
from hr.employees
where last_name = 'King'
group by last_name
having count(*) = 1;
```

This statement returns no results, meaning that the count of people with a surname of King is not 1; it's some other number like 0, 2, or more. The statement first determines which rows have a LAST_NAME value of King. It then groups by LAST_NAME and counts the hits encountered. Lastly, the HAVING clause tests to see if the count of rows with a LAST_NAME of King was equal to 1. Only those results are returned, so a surname is unique only if you see a result.

If we remove the HAVING clause as in the next SELECT statement, we'll see how many Kings are in the HR.EMPLOYEES table.

```
select last_name, count(*)
from hr.employees
where last_name = 'King'
group by last_name;
```

```
LAST_NAME                   COUNT(*)
------------------------- ----------
King                               2
```

Two people have a surname of King, thus it isn't unique and didn't show up in our test for uniqueness.

The same technique can be extended to test for unique combinations of columns. We can expand our recipe to test if someone's complete name, based on the combination of FIRST_NAME and LAST_NAME, is unique. This SELECT statement includes both columns in the criteria, testing to see if Lindsey Smith is a unique full name in the HR.EMPLOYEES table.

```
select first_name, last_name, count(*)
from hr.employees
where first_name = 'Lindsey'
and last_name = 'Smith'
group by first_name, last_name
having count(*) = 1;

FIRST_NAME           LAST_NAME                   COUNT(*)
-------------------- ------------------------- ----------
Lindsey              Smith                              1
```

You can write similar recipes that use string concatenation, self-joins, and a number of other methods.

2-9. Calculating Totals and Subtotals

Problem

You need to calculate totals and subtotals in a variety of environments, using the lowest common denominator of SQL. For instance, you need to count the number of people in each department, as well as a grand total, in a way that can run across a variety of editions of Oracle without change.

Solution

In situations where you feel you can't use analytic functions like ROLLUP and CUBE, or are restricted by licensing or other factors, you can use traditional aggregation and grouping techniques in separate SQL statements, and combine the results with a UNION to fold all the logic into a single statement. This SELECT combines counts of employees by department in one query, with the count of all employees in another query.

```
select nvl(to_char(department_id),'-') as "DEPT.", count(*) as "EMP_COUNT"
from hr.employees
group by department_id
union
select 'All Depts.', count(*)
from hr.employees;
```

The recipe results appear as follows, with abridged output to save space.

```
DEPT.        EMP_COUNT
----------- ----------
-                    1
10                   1
100                  6
110                  2
…
90                   3
All Depts.         107

13 rows selected.
```

How It Works

This recipe uses separate queries to calculate different aggregates and different levels, combining the results into a report-style output using UNION. In effect, two distinct sets of results are generated. First, the count of employees by department is accomplished using this SELECT statement.

```
select nvl(to_char(department_id),'-') as "DEPT.", count(*) as "EMP_COUNT"
from hr.employees
group by department_id;
```

Note that the TO_CHAR function is used to convert the integer DEPARTMENT_ID values to character equivalents. This is to ensure the eventual UNION is not plagued by implicit casting overhead, or even casting errors. In this recipe, we know we're going to want to use the literal phrase "All Depts." in conjunction with the overall employee count, and have it appear in line with the DEPARTMENT_ID values. Without casting, this results in an attempt to form a union from a column defined as an integer and a literal string. We'll receive this error if we don't perform the casting.

```
ORA-01790: expression must have same datatype as corresponding expression
```

Obviously not a useful outcome. You will often see this error when dealing with unions that must handle NULL values.

We also need the TO_CHAR conversion to work in conjunction with the NVL null-testing function, to map the employees with a null DEPARTMENT_ID to a "-" to indicate no department. This is for purely cosmetic reasons, but you can see how it can provide clarity for people reading your recipe results. This way, they don't have to wonder what a blank or null DEPARTMENT_ID means.

The second query in the union calculates only the total count of employees in the HR.EMPLOYEES table, and utilizes the flexible literal handling Oracle provides to implicitly group the full count with the value "All Depts."

```
select 'All Depts.', count(*)
from hr.employees
```

The UNION clause then simply stitches the two results together, giving the output you see. This is equivalent to using the ROLLUP clause covered in the recipe *Aggregating Data at Multiple Levels* earlier in this chapter.

2-10. Building Your Own Aggregate Function

Problem

You need to implement a custom aggregation to work in conjunction with Oracle's existing aggregate functions and grouping mechanisms. You specifically want to aggregate strings in multiple rows to one row so that the text from each row is concatenated, one after the other. You've seen this supported in other databases but you can't find an equivalent Oracle function.

Solution

Oracle provides a framework for writing your own aggregate functions, either to provide an aggregation not otherwise available in Oracle or to develop your own custom version of a common aggregate function with differing behavior. Our recipe creates a new aggregate function, STRING_TO_LIST, and the supporting type definition and type body based on Oracle's template for custom aggregates.

The following type definition defines the mandatory four functions Oracle requires for a custom aggregate type.

```
create or replace type t_list_of_strings as object (
  string_list varchar2(4000),

  static function odciaggregateinitialize
    (agg_context in out t_list_of_strings)
    return number,

  member function odciaggregateiterate
    (self in out t_list_of_strings,
     next_string_to_add in varchar2 )
    return number,

  member function odciaggregatemerge
    (self in out t_list_of_strings,
     para_context in t_list_of_strings)
    return number,

  member function odciaggregateterminate
    (self in t_list_of_strings,
     final_list_to_return out varchar2,
     flags in number)
    return number
);
/
```

We've limited the STRING_LIST parameter to 4000, even though PL/SQL supports up to 32000, to ensure we don't pass the threshold supported by plain SQL in Oracle. Each of the four required functions implements the various stages of our aggregation of a set of strings into one list.

```
create or replace type body t_list_of_strings is
  static function odciaggregateinitialize
    (agg_context in out t_list_of_strings)
    return number is
  begin
    agg_context := t_list_of_strings(null);
    return odciconst.success;
  end;

  member function odciaggregateiterate
    (self in out t_list_of_strings,
     next_string_to_add in varchar2 )
    return number is
  begin
    self.string_list := self.string_list || ' , ' || next_string_to_add;
    return odciconst.success;
  end;

  member function odciaggregatemerge
    (self in out t_list_of_strings,
     para_context in t_list_of_strings)
    return number is
  begin
    self.string_list := self.string_list || ' , ' || para_context.string_list;
    return odciconst.success;
  end;

  member function odciaggregateterminate
    (self in t_list_of_strings,
     final_list_to_return out varchar2,
     flags in number)
    return number is
  begin
    final_list_to_return := ltrim(rtrim(self.string_list, ' , '), ' , ');
    return odciconst.success;
  end;
end;
/
```

With the type and type body in place, we build the function we'll use in our actual SQL statements to produce custom aggregated data.

```
create or replace function string_to_list
  (input_string varchar2)
  return varchar2
  parallel_enable
  aggregate using t_list_of_strings;
/
```

We can now use our custom aggregate function to compose a list from any set or subset of strings that we generate in a query. In this recipe, we want to return the surnames of all the employees assigned to a given manager.

```
select manager_id, string_to_list (last_name) as employee_list
from hr.employees
group by manager_id;
```

Our new custom aggregation function does the trick, producing the surnames in a list-like output along with the manager's ID.

```
MANAGER_ID EMPLOYEE_LIST
---------- ------------------------------------------------------------
       100 Hartstein , Kochhar , De Haan , Fripp , Kaufling , Weiss ...
       101 Whalen , Greenberg , Higgins , Baer , Mavris
       102 Hunold
       103 Ernst , Pataballa , Lorentz , Austin
       108 Faviet , Chen , Sciarra , Urman , Popp
...
```

How It Works

Don't let the jump to PL/SQL, or the length of the recipe, scare you away. In fact, the majority of the code above is boiler-plate template offered by Oracle to assist with building custom aggregates quickly and easily.

This recipe builds the three components necessary to create a custom aggregation, and then uses it in a SQL statement to list the employee surnames by manager. First it defines a custom type defining the four prescribed functions Oracle requires for a custom aggregate. These are:

ODCIAggregateInitialize: This function is called at the very beginning of processing and sets up the new instance of the aggregate type—in our case, `T_LIST_OF_STRINGS`. This new instance has its member variables (if any) and is ready for the main aggregation phase

ODCIAggregateIterate: This holds the core functionality of your custom aggregation. The logic in this function will be called for each data value passed over in your driving query. In our case, this is the logic that takes each string value and appends it to the existing list of strings.

ODCIAggregateMerge: This is the semi-optional function that controls Oracle's behavior, if Oracle decides to perform the aggregation in parallel, using multiple parallel slaves working against portions of the data. While you don't have to enable the parallel option (see below), you still need to include this function. In our case, should our list creation be split into parallel tasks, all we need to do is concatenate the sublists at the end of the parallel process.

ODCIAggregateTerminate; This is the final function and is required to return the result to the calling SQL function. In our case, it returns the global variable `STRING_LIST` that has been built up in the iteration and parallel merging stages.

We now have the mechanics of aggregation built. The next part of the recipe builds the function that you can call to actually get your great new aggregation feature working on data. We create a function that has many of the normal features you'd expect:

```
create or replace function string_to_list
  (input_string varchar2)
  return varchar2
  parallel_enable
  aggregate using t_list_of_strings;
/
```

The function gets a name, STRING_TO_LIST, and it takes a VARCHAR2 string as input and returns one as output. So far, very mundane. It's the next two lines that are key to our recipe's aggregation behavior.

The PARALLEL_ENABLE clause is entirely optional, and instructs Oracle that it is safe to perform the underlying aggregation logic in parallel if the optimizer decides to take that path. Depending on the logic in your ODCIAGGREGATEITERATE function, you may have particular actions that must happen in a particular order and thus want to avoid parallelism. Enabling parallel processing also implies that logic is in place in the ODCIAGGREGATEMERGE member function to deal with merged subsets of results.

The AGGREGATE USING T_LIST_OF_STRINGS clause is where all the magic happens. This line instructs the function that it is aggregate in nature, and an object of type T_LIST_OF_STRINGS should be instantiated with the input parameter, kicking off the actual aggregation work.

■ **Note** Oracle only supports the creation of custom aggregate functions that take exactly one input parameter, and return exactly one output parameter. That's why you don't see any explicit instruction in our recipe to pass the INPUT_STRING parameter to the instantiated T_LIST_OF_STRINGS type, nor one mapping the return value from the T_LIST_OF_STRINGS.ODCIAGGREGATETERMINATE member function back to the return value of the STRING_TO_LIST function. They are the only things Oracle can do when it sees the aggregate clause, so they are implicit when you use the aggregate feature.

From there, calling and use of the new STRING_TO_LIST function behaves much like any other aggregate function, like AVG, MAX, MIN, and so on.

2-11. Accessing Values from Subsequent or Preceding Rows

Problem

You would like to query data to produce an ordered result, but you want to include calculations based on preceding and following rows in the result set. For instance, you want to perform calculations on event-style data based on events that occurred earlier and later in time.

Solution

Oracle supports the LAG and LEAD analytical functions to provide access to multiple rows in a table or expression, utilizing preceding/following logic—and you won't need to resort to joining the source data to itself. Our recipe assumes you are trying to tackle the business problem of visualizing the trend in hiring of staff over time. The LAG function can be used to see which employee's hiring followed another, and also to calculate the elapsed time between hiring.

```
select first_name, last_name, hire_date,
  lag(hire_date, 1, '01-JUN-1987') over (order by hire_date) as Prev_Hire_Date,
  hire_date - lag(hire_date, 1, '01-JUN-1987') over (order by hire_date)
    as Days_Between_Hires
from hr.employees
order by hire_date;
```

Our query returns 107 rows, linking the employees in the order they were hired (though not necessarily preserving the implicit sort for display or other purposes), and showing the time delta between each joining the organization.

```
FIRST_NAME  LAST_NAME  HIRE_DATE PREV_HIRE DAYS_BETWEEN
----------- ---------- --------- --------- ------------
Steven      King       17-JUN-87 01-JUN-87           16
Jennifer    Whalen     17-SEP-87 17-JUN-87           92
Neena       Kochhar    21-SEP-89 17-SEP-87          735
Alexander   Hunold     03-JAN-90 21-SEP-89          104
Bruce       Ernst      21-MAY-91 03-JAN-90          503
...
David       Lee        23-FEB-00 06-FEB-00           17
Steven      Markle     08-MAR-00 23-FEB-00           14
Sundar      Ande       24-MAR-00 08-MAR-00           16
Amit        Banda      21-APR-00 24-MAR-00           28
Sundita     Kumar      21-APR-00 21-APR-00            0

107 rows selected.
```

You can calculate for yourself the day differences to confirm the LAG function and difference arithmetic are indeed working as claimed. For instance, there really are 503 days between January 3, 1990 and May 21, 1991.

How It Works

The LAG and LEAD functions are like most other analytical and windowing functions in that they operate once the base non-analytic portion of the query is complete. Oracle performs a second pass over the intermediate result set to apply any analytical predicates. In effect, the non-analytic components are evaluated first, as if this query had been run.

```
select first_name, last_name, hire_date
  -- placeholder for Prev_Hire_Date,
  -- placehodler for Days_Between_Hires
from hr.employees;
```

The results at this point would look like this if you could see them:

```
FIRST_NAME  LAST_NAME  HIRE_DATE PREV_HIRE DAYS_BETWEEN
----------- ---------- --------- --------- ------------
Steven      King       17-JUN-87  ( To Be Determined )
Jennifer    Whalen     17-SEP-87  ( To Be Determined )
Neena       Kochhar    21-SEP-89  ( To Be Determined )
Alexander   Hunold     03-JAN-90  ( To Be Determined )
Bruce       Ernst      21-MAY-91  ( To Be Determined )
...
```

The analytic function(s) are then processed, providing the results you've seen. Our recipe uses the LAG function to compare the current row of results with a preceding row. The general format is the best way to understand LAG, and has the following form.

```
lag (column or expression, preceding row offset, default for first row)
```

The *column or expression* is mostly self-explanatory, as this is the table data or computed result over which you want LAG to operate. The *preceding row offset* portion indicates the relative row prior to the current row the LAG should act against. In our case, the value '1' means the row that is one row before the current row. The default for LAG indicates what value to use as a precedent for the first row, as there is no row *zero* in a table or result. We've chosen the arbitrary date of 01-JUN-1987 as a notional date on which the organization was founded. You could use any date, date calculation, or date-returning function here. Oracle will supply a NULL value if you don't specify the first row's precedent value.

The OVER analytic clause then dictates the order of data against which to apply the analytic function, and any partitioning of the data into windows or subsets (not shown in this recipe). Astute readers will realize that this means our recipe could have included a general ORDER BY clause that sorted the data for presentation in a different order from the HIRE_DATE ordering used for the LAG function. This gives you the most flexibility to handle general ordering and analytic lag and lead in different ways for the same statement. We'll show an example of this later in this chapter. And remember, you should never rely on the implicit sorting that analytic functions use. This can and will change in the future, so you are best advised to always include ORDER BY for sorting wherever explicitly required.

The LEAD function works in a nearly identical fashion to LAG, but instead tracks following rows rather than preceding ones. We could rewrite our recipe to show hires along with the HIRE_DATE of the next employee, and a similar elapsed-time window between their employment dates, as in this SELECT statement.

```
select first_name, last_name, hire_date,
  lead(hire_date, 1, sysdate) over (order by hire_date) as Next_Hire_Date,
  lead(hire_date, 1, sysdate) over (order by hire_date) - hire_date
    as Days_Between_Hires
from hr.employees;
```

The pattern of dates is very intuitive now that you've seen the LAG example. With LEAD, the key difference is the effect of the default value in the third parameter.

```
FIRST_NAME  LAST_NAME  HIRE_DATE NEXT_HIRE DAYS_BETWEEN
----------- ---------- --------- --------- ------------
Steven      King       17-JUN-87 17-SEP-87           92
Jennifer    Whalen     17-SEP-87 21-SEP-89          735
Neena       Kochhar    21-SEP-89 03-JAN-90          104
Alexander   Hunold     03-JAN-90 21-MAY-91          503
```

```
Bruce       Ernst      21-MAY-91 13-JAN-93          603
...
David       Lee        23-FEB-00 08-MAR-00           14
Steven      Markle     08-MAR-00 24-MAR-00           16
Sundar      Ande       24-MAR-00 21-APR-00           28
Amit        Banda      21-APR-00 21-APR-00            0
Sundita     Kumar      21-APR-00 21-APR-09      3287.98

107 rows selected.
```

In contrast to LAG, where the default provides a notional starting point for the first row's comparison, LEAD uses the default value to provide a hypothetical end point for the last row in the forward-looking chain. In this recipe, we are comparing how many days have elapsed between employees being hired. It makes sense for us to compare the last employee hired (in this case, Sundita Kumar) with the current date using the SYSDATE function. This is a quick and easy finishing flourish to calculate the days that have elapsed since hiring the last employee.

2-12. Assigning Ranking Values to Rows in a Query Result

Problem

The results from a query need to be allocated an ordinal number representing their positions in the result. You do not want to have to insert and track these numbers in the source data.

Solution

Oracle provides the RANK analytic function to generate a ranking number for rows in a result set. RANK is applied as a normal OLAP-style function to a column or derived expression. For the purposes of this recipe, we'll assume that the business would like to rank employees by salary, from highest-paid down. The following SELECT statement uses the rank function to assign these values.

```
select employee_id, salary, rank() over (order by salary desc) as Salary_Rank
from hr.employees;
```

Our query produces results from the highest earner at 24000 per month, right down to the employee in 107th place earning 2100 per month, as these abridged results show.

```
EMPLOYEE_ID     SALARY SALARY_RANK
----------- ---------- -----------
        100      24000           1
        101      17000           2
        102      17000           2
        145      14000           4
        146      13500           5
        201      13000           6
        205      12000           7
        108      12000           7
        147      12000           7
…
        132       2100         107

107 rows selected.
```

How It Works

RANK acts like any other analytic function, operating in a second pass over the result set once non-analytic processing is complete. In this recipe, the EMPLOYEE_ID and SALARY values are selected (there are no WHERE predicates to filter the table's data, so we get everyone employed in the organization). The analytic phase then orders the results in descending order by salary, and computes the rank value on the results starting at 1.

Note carefully how the RANK function has handled equal values. Two employees with salary of 17000 are given equal rank of 2. The next employee, at 14000, has a rank of 4. This is known as *sparse* ranking, where tied values "consume" place holders. In practical terms, this means that our equal second-place holders consume both second and third place, and the next available rank to provide is 4.

You can use an alternative to sparse ranking called *dense* ranking. Oracle supports this using the DENSE_RANK analytical function. Observe what happens to the recipe when we switch to dense ranking.

```
select employee_id, salary, dense_rank() over (order by salary desc)
  as Salary_Rank
from hr.employees;
```

We now see the "missing" consecutive rank values.

```
EMPLOYEE_ID     SALARY SALARY_RANK
----------- ---------- -----------
        100      24000           1
        101      17000           2
        102      17000           2
        145      14000           3
        146      13500           4
        201      13000           5
        205      12000           6
        108      12000           6
        147      12000           6
        168      11500           7
…
        132       2100          58

107 rows selected.
```

The classic examples of when the different kinds of ranking are used are in sporting competitions. Football and other sports typically use sparse ranking when tracking team win/loss progress on a ladder or table. The Olympic Games, on the other hand, tend to use dense ranking when competitors tie in events like swimming and others. They like to ensure that there are always gold, silver, and bronze medalists, even if there are tied entries and they have to give out more medals.

■ **Note** The authors live in hope that, one day, writing SQL statements will be an Olympic event. We'll be sure to use a dense ranking approach to maximize our chance of getting a medal.

Our recipe uses a simple ranking across all employees to determine salary order. Both RANK and DENSE_RANK support normal analytic extensions, allowing us to partition our source data so we can generate a rank for each subset of data. Continuing our recipe's theme, this means we could allocate a rank for salary earners from highest to lowest within each department. Introducing that partitioning to the query looks like this:

```
Select department_id, employee_id, salary, rank() over
  (partition by department_id order by salary desc) as Salary_Rank
From hr.employees
;
```

Our results now show per-department ranking of employees by salary.

```
DEPARTMENT_ID EMPLOYEE_ID     SALARY SALARY_RANK
------------- ----------- ---------- -----------
           10         200       4400           1
           20         201      13000           1
           20         202       6000           2
           30         114      11000           1
           30         115       3100           2
           30         116       2900           3
           30         117       2800           4
           30         118       2600           5
           30         119       2500           6
           40         203       6500           1
…
```

As the DEPARTMENT_ID value ticks over, the PARTITION clause drives the RANK function to start its calculation again for the next subset of results.

2-13. Finding First and Last Values within a Group

Problem

You want to calculate and display aggregate information like minimum and maximum for a group, along with detail information for each member. You want don't want to repeat effort to display the aggregate and detail values.

Solution

Oracle provides the analytic functions FIRST and LAST to calculate the leading and ending values in any ordered sequence. Importantly, these do not require grouping to be used, unlike explicit aggregate functions such as MIN and MAX that work without OLAP features.

For our recipe, we'll assume the problem is a concrete one of displaying an employee's salary, alongside the minimum and maximum salaries paid to the employee's peers in their department. This SELECT statement does the work.

```
select department_id, first_name, last_name,
  min(salary)
    over (partition by department_id) "MinSal",
  salary,
  max(salary)
    over (partition by department_id) "MaxSal"
from hr.employees
order by department_id, salary;
```

This code outputs all employees and displays their salaries between the lowest and highest within their own department, as shown in the following partial output.

```
DEPARTMENT_ID FIRST_NAME LAST_NAME      MinSal     SALARY     MaxSal
------------- ---------- ---------- ---------- ---------- ----------
           10 Jennifer   Whalen           4400       4400       4400
           20 Pat        Fay              6000       6000      13000
           20 Michael    Hartstein        6000      13000      13000
           30 Karen      Colmenares       2500       2500      11000
           30 Guy        Himuro           2500       2600      11000
           30 Sigal      Tobias           2500       2800      11000
           30 Shelli     Baida            2500       2900      11000
           30 Alexander  Khoo             2500       3100      11000
           30 Den        Raphaely         2500      11000      11000
           40 Susan      Mavris           6500       6500       6500
…
107 rows selected.
```

How It Works

The key to both the FIRST and LAST analytic functions is their ability to let you perform the grouping and ordering on one set of criteria, while leaving you free to order differently in the main body of the query, and optionally group or not as desired by other factors.

The OLAP window is partitioned over each department with the OVER clause

```
over (partition by department_id) “MinSal”
```

2-14. Performing Aggregations over Moving Windows

Problem

You need to provide static and moving summaries or aggregates based on the same data. For example, as part of a sales report, you need to provide a monthly summary of sales order amounts, together with a moving three- month average of sales amounts for comparison.

Solution

Oracle provides moving or rolling window functions as part of the analytical function set. This gives you the ability to reference any number of preceding rows in a result set, the current row in the result set, and any number of following rows in a result set. Our initial recipe uses the current row and the three preceding rows to calculate the rolling average of order values.

```
select to_char(order_date, 'MM') as OrderMonth, sum(order_total) as MonthTotal,
avg(sum(order_total))
  over
    (order by to_char(order_date, 'MM') rows between 3 preceding and current row)
    as RollingQtrAverage
from oe.orders
where order_date between '01-JAN-1999' and '31-DEC-1999'
group by to_char(order_date, 'MM')
order by 1;
```

We see the month, the associated total, and the calculated rolling three-month average in our results.

```
OR MONTHTOTAL ROLLINGQTRAVERAGE
-- ---------- -----------------
02   120281.6          120281.6
03   200024.1         160152.85
04       1636          107313.9
05   165838.2        121944.975
06   350019.9         179379.55
07   280857.1          199587.8
08   152554.3        237317.375
09   460216.1         310911.85
10    59123.6        238187.775
11   415875.4         271942.35
12     338672        318471.775

11 rows selected.
```

You might notice January (OrderMonth 01) is missing. This isn't a quirk of this approach: rather it's because the `OE.ORDERS` table has no orders recorded for this month in 1999.

How It Works

Our SELECT statement for a rolling average starts by selecting some straightforward values. The month number is extracted from the ORDER_DATE field using the TO_CHAR() function with the MM format string to obtain the month's number. We choose the month number rather than the name so that the output is sorted as a person would expect.

Next up is a normal aggregate of the ORDER_TOTAL field using the traditional SUM function. No magic there. We then introduce an OLAP AVG function, which is where the detail of our rolling average is managed. That part of the statement looks like this.

```
avg(sum(order_total)) over (order by to_char(order_date, 'MM')
  rows between 3 preceding and current row) as RollingQtrAverage
```

All of that text is to generate our result column, the ROLLINGQTRAVERAGE. Breaking the sections down will illustrate how each part contributes to the solution. The leading functions, AVG(SUM(ORDER_TOTAL)), suggest we are going to sum the ORDER_TOTAL values and then take their average. That is correct to an extent, but Oracle isn't just going to calculate a normal average or sum. These are OLAP AVG and SUM functions, so their scope is governed by the OVER clause.

The OVER clause starts by instructing Oracle to perform the calculations based on the order of the formatted ORDER_DATE field—that's what ORDER BY TO_CHAR(ORDER_DATE, 'MM') achieves—effectively ordering the calculations by the values 02 to 12 (remember, there's no data for January 1999 in the database). Finally, and most importantly, the ROWS element tells Oracle the size of the window of rows over which it should calculate the driving OLAP aggregate functions. In our case, that means over how many months should the ORDER_TOTAL values be summed and then averaged. Our recipe instructs Oracle to use the results from the third-last row through to the current row. This is one interpretation of three-month rolling average, though technically it's actually generating an average over four months. If what you want is really a three-month average —the last two months plus the current month—you'd change the ROWS BETWEEN element to read

```
rows between 2 preceding and current row
```

This brings up an interesting point. This recipe assumes you want a rolling average computed over historic data. But some business requirements call for a rolling window to track trends based on data not only prior to a point in time, but also after that point. For instance, we might want to use a three-month window but base it on the previous, current, and following months. The next version of the recipe shows exactly this ability of the windowing function, with the key changes in bold.

```
select to_char(order_date, 'MM') as OrderMonth, sum(order_total) as MonthTotal,
avg(sum(order_total)) over (order by to_char(order_date, 'MM')
  rows between 1 preceding and 1 following) as AvgTrend
from oe.orders
where order_date between '01-JAN-1999' and '31-DEC-1999'
group by to_char(order_date, 'MM')
order by 1
/
```

Our output changes as you'd expect, as the monthly ORDER_TOTAL values are now grouped differently for the calculation.

```
OR MONTHTOTAL   AVGTREND
-- ---------- ----------
02   120281.6  160152.85
03   200024.1   107313.9
04       1636 122499.433
05   165838.2 172498.033
06   350019.9 265571.733
07   280857.1 261143.767
08   152554.3 297875.833
09   460216.1 223964.667
10    59123.6 311738.367
11   415875.4 271223.667
12     338672   377273.7

11 rows selected.
```

The newly designated AVGTREND value is calculated as described, using both preceding and following rows. Both our original recipe and this modified version are rounded out with a WHERE clause to select only data from the OE.ORDERS table for the year 1999. We group by the derived month number so that our traditional sum of ORDER_TOTAL in the second field of the results aggregates correctly, and finish up ordering logically by the month number.

2-15. Removing Duplicate Rows Based on a Subset of Columns

Problem

Data needs to be cleansed from a table based on duplicate values that are present only in a subset of rows.

Solution

Historically there were Oracle-specific solutions for this problem that used the ROWNUM feature. However, this can become awkward and complex if you have multiple groups of duplicates and want to remove the excess data in one pass. Instead, you can use Oracle's ROW_NUMBER OLAP function with a DELETE statement to efficiently remove all duplicates in one pass.

To illustrate our recipe in action, we'll first introduce several new staff members that have the same FIRST_NAME and LAST_NAME as some existing employees. These INSERT statements create our problematic duplicates.

```
insert into hr.employees
(employee_id, first_name, last_name, email, phone_number, hire_date, job_id,
 salary, commission_pct, manager_id, department_id)
Values
(210, 'Janette', 'King', 'JKING2', '650.555.8880', '25-MAR-2009', 'SA_REP',
 3500, 0.25, 145, 80);
```

```
Insert into hr.employees
(employee_id, first_name, last_name, email, phone_number, hire_date, job_id,
 salary, commission_pct, manager_id, department_id)
Values
(211, 'Patrick', 'Sully', 'PSULLY2', '650.555.8881', '25-MAR-2009', 'SA_REP',
 3500, 0.25, 145, 80);

Insert into hr.employees
(employee_id, first_name, last_name, email, phone_number, hire_date, job_id,
 salary, commission_pct, manager_id, department_id)
Values
(212, 'Allen', 'McEwen', 'AMCEWEN2', '650.555.8882', '25-MAR-2009', 'SA_REP',
 3500, 0.25, 145, 80);

commit;
```

To show that we do indeed have some duplicates, a quick SELECT shows the rows in question.

```
select employee_id, first_name, last_name
from hr.employees
where first_name in ('Janette','Patrick','Allan')
and last_name in ('King','Sully','McEwen')
order by first_name, last_name;

EMPLOYEE_ID FIRST_NAME  LAST_NAME
----------- ----------- ----------
        158 Allan       McEwen
        212 Allan       McEwen
        210 Janette     King
        156 Janette     King
        211 Patrick     Sully
        157 Patrick     Sully
```

If you worked in HR, or were one of these people, you might be concerned with the unpredictable consequences and want to see the duplicates removed. With our problematic data in place, we can introduce the SQL to remove the “extra” Janette King, Patrick Sully, and Allen McEwen.

```
delete from hr.employees
where rowid in
  (select rowid
   from
     (select first_name, last_name, rowid,
      row_number() over
       (partition by first_name, last_name order by employee_id)
       staff_row
      from hr.employees)
   where staff_row > 1);
```

When run, this code does indeed claim to remove three rows, presumably our duplicates. To check, we can repeat our quick query to see which rows match those three names. We see this set of results.

```
EMPLOYEE_ID FIRST_NAME  LAST_NAME
----------- ----------- -------------------------
        158 Allan       McEwen
        156 Janette     King
        157 Patrick     Sully
```

Our DELETE has succeeded, based on finding duplicates for a subset of columns only.

How It Works

Our recipe uses both the ROW_NUMBER OLAP function and Oracle's internal ROWID value for uniquely identifying rows in a table. The query starts with exactly the kind of DELETE syntax you'd assume.

```
delete from hr.employees
where rowid in
  (… nested subqueries here …)
```

As you'd expect, we're asking Oracle to delete rows from HR.EMPLOYEES where the ROWID value matches the values we detect for duplicates, based on criteria evaluating a subset of columns. In our case, we use subqueries to precisely identify duplicates based on FIRST_NAME and LAST_NAME.

To understand how the nested subqueries work, it's easiest to start with the innermost subquery, which looks like this.

```
select first_name, last_name, rowid,
  row_number() over
  (partition by first_name, last_name order by employee_id)
  staff_row
      from hr.employees
```

We've intentionally added the columns FIRST_NAME and LAST_NAME to this innermost subquery to make the recipe understandable as we work through its logic. Strictly speaking, these are superfluous to the logic, and the innermost subquery could be written without them to the same effect. If we execute just this innermost query (with the extra columns selected for clarity), we see these results.

```
FIRST_NAME  LAST_NAME    ROWID              STAFF_ROW
----------- ------------ ------------------ ----------
...
Alexander   Khoo         AAARAgAAFAAAABYAAP          1
Janette     King         AAARAgAAFAAAABXAAD          1
Janette     King         AAARAgAAFAAAABYAA4          2
Steven      King         AAARAgAAFAAAABYAAA          1
...
Samuel      McCain       AAARAgAAFAAAABYABe          1
Allan       McEwen       AAARAgAAFAAAABXAAF          1
Allan       McEwen       AAARAgAAFAAAABYAA6          2
Irene       Mikkilineni  AAARAgAAFAAAABYAAa          1
...
```

```
Martha      Sullivan     AAARAgAAFAAAABYABS          1
Patrick     Sully        AAARAgAAFAAAABXAAE          1
Patrick     Sully        AAARAgAAFAAAABYAA5          2
Jonathon    Taylor       AAARAgAAFAAAABYABM          1
...

110 rows selected.
```

All 110 staff from the HR.EMPLOYEES table have their FIRST_NAME, LAST_NAME and ROWID returned. The ROW_NUMBER() function then works over sets of FIRST_NAME and LAST_NAME driven by the PARTITION BY instruction. This means that for every unique FIRST_NAME and LAST_NAME, ROW_NUMBER will start a running count of rows we've aliased as STAFF_ROW. When a new FIRST_NAME and LAST_NAME combination is observed, the STAFF_ROW counter resets to 1.

In this way, the first Janette King has a STAFF_ROW value of 1, the second Janette King entry has a STAFF_ROW value of 2, and if there were a third and fourth such repeated name, they'd have STAFF_ROW values of 3 and 4 respectively. With our identically-named staff now numbered, we move to the next outermost subselect, which queries the results from above.

```
select rowid
from select
  (select first_name, last_name, rowid,
     row_number() over
     (partition by first_name, last_name order by first_name, last_name)
     staff_row
   from hr.employees)
where staff_row > 1
```

This outer query looks simple, because it is! We simply SELECT the ROWID values from the results of our innermost query, where the calculated STAFF_ROW value is greater than 1. That means that we only select the ROWID values for the second Janette King, Allan McEwen, and Patrick Sully, like this.

```
ROWID
------------------
AAARAgAAFAAAABYAA4
AAARAgAAFAAAABYAA6
AAARAgAAFAAAABYAA5
```

Armed with those ROWID values, the DELETE statement knows exactly which rows are the duplicates, based on only a comparison and count of FIRST_NAME and LAST_NAME.

The beauty of this recipe is the basic structure translates very easily to deleting data based on any such column-subset duplication. The format stays the same, and only the table name and a few column names need to be changed. Consider this a pseudo-SQL template for all such cases.

```
delete from <your_table_here>
where rowid in
  (select rowid
   from
     (select rowid,
      row_number() over
       (partition by <first_duplicate_column>, <second_duplicate_column>, <etc.>
        order by <desired ordering column>)
        duplicate_row_count
       from <your_table_here>)
   where duplicate_row_count > 1)
/
```

Simply plug in the value for your table in place of the marker <*your_table_here*>, and the columns you wish to use to determine duplication in place of equivalent column placeholders, and you're in business!

2-16. Finding Sequence Gaps in a Table

Problem

You want to find all gaps in the sequence of numbers or in dates and times in your data. The gaps could be in dates recorded for a given action, or in some other data with a logically consecutive nature.

Solution

Oracle's `LAG` and `LEAD` OLAP functions let you compare the current row of results with a preceding row. The general format of `LAG` looks like this

```
Lag (column or expression, preceding row offset, default for first row)
```

The *column or expression* is the value to be compared with lagging (preceding) values. The *preceding row offset* indicates how many rows prior to the current row the `LAG` should act against. We've used '1' in the following listing to mean the row one prior to the current row. The default for `LAG` indicates what value to use as a precedent for the first row, as there is no row zero in a table or result. We instruct Oracle to use 0 as the default anchor value, to handle the case where we look for the day prior to the first of the month.

The `WITH` query alias approach can be used in almost all situations where a subquery is used, to relocate the subquery details ahead of the main query. This aids readability and refactoring of the code if required at a later date.

This recipe looks for gaps in the sequence of days on which orders were made for the month of November 1999:

```
with salesdays as
 (select extract(day from order_date) next_sale,
  lag(extract(day from order_date),1,0)
    over (order by extract(day from order_date)) prev_sale
  from oe.orders
  where order_date between '01-NOV-1999' and '30-NOV-1999')
select prev_sale, next_sale
from salesdays
where next_sale - prev_sale > 1
order by prev_sale;
```

Our query exposes the gaps, in days, between sales for the month of November 1999.

```
PREV_SALE  NEXT_SALE
---------- ----------
         1         10
        10         14
        15         19
        20         22
```

The results indicate that after an order was recorded on the first of the month, no subsequent order was recorded until the 10th. Then a four-day gap followed to the 14th, and so on. An astute sales manager might well use this data to ask what the sales team was doing on those gap days, and why no orders came in!

How It Works

The query starts by using the `WITH` clause to name a subquery with an alias in an out-of-order fashion. The subquery is then referenced with an alias, in this case `SALESDAYS`.

The `SALESDAYS` subquery calculates two fields. First, it uses the `EXTRACT` function to return the numeric day value from the `ORDER_DATE` date field, and labels this data as `NEXT_SALE`. The lag OLAP function is then used to calculate the number for the day in the month (again using the `EXTRACT` method) of the `ORDER_DATE` of the preceding row in the results, which becomes the `PREV_SALE` result value. This makes more sense when you visualize the output of just the subquery select statement

```
select extract(day from order_date) next_sale,
  lag(extract(day from order_date),1,0)
    over (order by extract(day from order_date)) prev_sale
from oe.orders
where order_date between '01-NOV-1999' and '30-NOV-1999'
```

The results would look like this if executed independently.

```
NEXT_SALE  PREV_SALE
---------- ----------
         1          0
        10          1
        10         10
        10         10
        14         10
        14         14
        15         14
        19         15
…
```

Starting with the anchor value of 0 in the lag, we see the day of the month for a sale as `NEXT_SALE`, and the day of the previous sale as `PREV_SALE`. You can probably already visually spot the gaps, but it's much easier to let Oracle do that for you too. This is where our outer query does its very simple arithmetic.

The driving query over the `SALESDAYS` subquery selects the `PREV_SALE` and `NEXT_SALE` values from the results, based on this predicate.

```
where next_sale - prev_sale > 1
```

We know the days of sales are consecutive if they're out by more than one day. We wrap up by ordering the results by the `PREV_SALE` column, so that we get a natural ordering from start of month to end of month.

Our query could have been written the traditional way, with the subquery in the `FROM` clause like this.

```
select prev_sale, next_sale
from (select extract(day from order_date) next_sale,
  lag(extract(day from order_date),1,0)
    over (order by extract(day from order_date)) prev_sale
  from oe.orders
  where order_date between '01-NOV-1999' and '30-NOV-1999')
where next_sale - prev_sale > 1
order by prev_sale
/
```

The approach to take is largely a question of style and readability. We prefer the WITH approach on those occasions where it greatly increases the readability of your SQL statements.

CHAPTER 3

Querying from Multiple Tables

Querying data from Oracle tables is probably the most common task you will perform as a developer or data analyst, and maybe even as a DBA—though probably not as the ETL (Extraction, Transformation, and Loading) tool expert. Quite often, you may query only one table for a small subset of rows, but sooner or later you will have to join multiple tables together. That's the beauty of a relational database, where the access paths to the data are not fixed: you can join tables that have common columns, or even tables that do not have common columns (at your own peril!).

In this chapter we'll cover solutions for joining two or more tables and retrieving the results based on the existence of desired rows in both tables (equi-join), rows that may exist only in one table or the other (left or right outer joins), or joining two tables together and including all rows from both tables, matching where possible (full outer joins).

But wait, there's more! Oracle (and the SQL language standard) contains a number of constructs that help you retrieve rows from tables based on the existence of the same rows in another table with the same column values for the selected rows in a query. These constructs include the `INTERSECT`, `UNION`, `UNION ALL`, and `MINUS` operators. The results from queries using these operators can in some cases be obtained using the standard table-join syntax, but if you're working with more than just a couple of columns, the query becomes unwieldy, hard to read, and hard to maintain.

You may also need to update rows in one table based on matching or non-matching values in another table, so we'll provide a couple of recipes on correlated queries and correlated updates using the `IN`/`EXISTS` SQL constructs as well.

Of course, no discussion of table manipulation would be complete without delving into the unruly child of the query world, the Cartesian join. There are cases where you want to join two or more tables without a join condition, and we'll give you a recipe for that scenario.

Most of the examples in this chapter are based on the schemas in the `EXAMPLE` tablespace created during an Oracle Database installation when you specify "Include Sample Schemas." Those sample schemas aren't required to understand the solutions in this chapter, but they give you the opportunity to try out the solutions on a pre-populated set of tables and even delve further into the intricacies of table joins.

3-1. Joining Corresponding Rows from Two or More Tables

Problem

You want to return rows from two or more tables that have one or more columns in common. For example, you may want to join the EMPLOYEES and the DEPARTMENTS table on a common column, but not all common columns, and return a list of employees and their department names.

Solution

If you are using Oracle Database 9*i* or later, you can use the ANSI SQL 99 join syntax with the USING clause. For example, the EMPLOYEES and DEPARTMENTS table in the Oracle sample schemas have the DEPARTMENT_ID column in common, so the query looks like this:

```
select employee_id, last_name, first_name, department_id, department_name
from employees
   join departments using(department_id)
;

EMPLOYEE_ID  LAST_NAME      FIRST_NAME     DEPARTMENT_ID      DEPARTMENT_NAME
------------ -------------- -------------- ------------------ --------------------
200          Whalen         Jennifer       10                 Administration
201          Hartstein      Michael        20                 Marketing
202          Fay            Pat            20                 Marketing
114          Raphaely       Den            30                 Purchasing
115          Khoo           Alexander      30                 Purchasing
. . .
113          Popp           Luis           100                Finance
205          Higgins        Shelley        110                Accounting
206          Gietz          William        110                Accounting

106 rows selected
```

The query retrieves most of the results from the EMPLOYEES table, and the DEPARTMENT_NAME column from the DEPARTMENTS table.

How It Works

The sample schemas supplied with a default installation of Oracle Database 11*g* provide a good starting point for trying out some Oracle features. Oracle Database comes with several sample schemas such as HR, OE, and BI to show not only the relationships between database schemas, but also to show some of the varied features such as index-organized tables (IOTs), function-based indexes, materialized views, large objects (BLOBs and CLOBs), and XML objects.

Here is the structure of the EMPLOYEES and DEPARTMENTS tables:

```
describe employees

Name                           Null     Type
------------------------------ -------- -----------------
EMPLOYEE_ID                    NOT NULL NUMBER(6)
FIRST_NAME                              VARCHAR2(20)
LAST_NAME                      NOT NULL VARCHAR2(25)
EMAIL                          NOT NULL VARCHAR2(25)
PHONE_NUMBER                            VARCHAR2(20)
HIRE_DATE                      NOT NULL DATE
JOB_ID                         NOT NULL VARCHAR2(10)
SALARY                                  NUMBER(8,2)
COMMISSION_PCT                          NUMBER(2,2)
MANAGER_ID                              NUMBER(6)
DEPARTMENT_ID                           NUMBER(4)

11 rows selected

describe departments

Name                           Null     Type
------------------------------ -------- -----------------
DEPARTMENT_ID                  NOT NULL NUMBER(4)
DEPARTMENT_NAME                NOT NULL VARCHAR2(30)
MANAGER_ID                              NUMBER(6)
LOCATION_ID                             NUMBER(4)

4 rows selected
```

There are three other basic ways to join these two tables on the DEPARTMENT_ID column. One of them is pre-ANSI SQL 99, one is more suitable when the column names in the joined tables are not identical, and one is outright dangerous, as you will see.

Using the "old style" join syntax, you include the join condition in the WHERE clause, like this:

```
select employee_id, last_name, first_name, e.department_id, department_name
from employees e, departments d
where e.department_id = d.department_id
;
```

While this approach works and is as efficient from an execution plan point of view, the older syntax can be hard to read, as you are mixing the table-join conditions with any filter conditions. It also forces you to specify a qualifier for the join columns in the SELECT clause.

Another ANSI SQL 99 method for joining tables uses the ON clause as follows:

```
select employee_id, last_name, first_name, e.department_id, department_name
from employees e
   join departments d
      on e.department_id = d.department_id
;
```

■ **Note** You can also use the `INNER JOIN` keywords instead of just `JOIN` in a multi-table query if you want to be more verbose or want to make it very clear that the query is an equi-join instead of an outer join or Cartesian product.

The `ON` clause is less readable (and usually requires more typing!) compared to the `USING` clause when the joined columns have the same name. It has the same downside as using the pre-ANSI SQL 99 syntax in that you must qualify the join columns or any other columns with the same name with an alias.

Finally, you can make the syntax for the `EMPLOYEE/DEPARTMENTS` query even simpler by using the `NATURAL JOIN` clause instead of the `JOIN . . . USING` clause, as in this example:

```
select employee_id, last_name, first_name, department_id, department_name
from employees natural join departments
;
```

When you use `NATURAL JOIN`, Oracle automatically joins the two tables on columns with the same name, which in this case is the `DEPARTMENT_ID` column. This makes the query even simpler and more readable, but has a very dark side in some circumstances. Here's an example. The query we've just shown with `NATURAL JOIN` returns 32 rows, while the earlier queries that used `USING` or `ON` each return 106 rows. What happened? Why the difference?

If you look closely at the table definitions, you'll see another common column called `MANAGER_ID`. `NATURAL JOIN` includes that column in the join criteria. Thus, the preceding query is really equivalent to the following:

```
select employee_id, last_name, first_name, department_id, department_name
from employees join departments using(department_id, manager_id)
;
```

This join is almost certainly not what you want, as the `MANAGER_ID` in the `EMPLOYEES` table has a slightly different meaning than the `MANAGER_ID` in the `DEPARTMENTS` table: `EMPLOYEES.MANAGER_ID` is the employee's manager, whereas `DEPARTMENTS.MANAGER_ID` is the manager of the entire department. As a result, the query does not return employees and their managers. Instead, it produces a list of employees whose department has the same manager as they do, which will not be the case in many organizations. Furthermore, a large number of departments in the `DEPARTMENTS` table do not have a manager assigned, thus the query using `NATURAL JOIN` will leave out employees who do have a reporting manager, but whose department does not have a manager assigned. Use `NATURAL JOIN` with caution!

Later in this chapter, we'll look at two techniques that can help you uncover these logical errors: using optional joins and dealing with `NULL` values in queries.

3-2. Stacking Query Results Vertically

Problem

You want to combine the results from two `SELECT` statements into a single result set.

Solution

Use the UNION operator. UNION combines the results of two or more queries and removes duplicates from the entire result set. In Oracle's mythical company, the employees in the EMPLOYEES_ACT table need to be merged with employees from a recent corporate acquisition. The recently acquired company's employee table EMPLOYEES_NEW has the same exact format as the existing EMPLOYEES_ACT table, so it should be easy to use UNION to combine the two tables into a single result set as follows:

```
select employee_id, first_name, last_name from employees_act;

EMPLOYEE_ID            FIRST_NAME           LAST_NAME
---------------------- -------------------- -------------------------
102                    Lex                  De Haan
105                    David                Austin
112                    Jose Manuel          Urman
118                    Guy                  Himuro
119                    Karen                Colmenares
205                    Shelley              Higgins

6 rows selected

select employee_id, first_name, last_name from employees_new;

EMPLOYEE_ID            FIRST_NAME           LAST_NAME
---------------------- -------------------- -------------------------
101                    Neena                Kochhar
105                    David                Austin
112                    Jose Manuel          Urman
171                    William              Smith
201                    Michael              Hartstein

5 rows selected

select employee_id, first_name, last_name from employees_act
union
select employee_id, first_name, last_name from employees_new
order by employee_id
;

EMPLOYEE_ID            FIRST_NAME           LAST_NAME
---------------------- -------------------- -------------------------
101                    Neena                Kochhar
102                    Lex                  De Haan
105                    David                Austin
112                    Jose Manuel          Urman
118                    Guy                  Himuro
119                    Karen                Colmenares
171                    William              Smith
201                    Michael              Hartstein
205                    Shelley              Higgins

9 rows selected
```

Using UNION removes the duplicate rows. You can have one ORDER BY at the end of the query to order the results. In this example, the two employee tables have two rows in common (some people need to work two or three jobs to make ends meet!), so instead of returning 11 rows, the UNION query returns nine.

How It Works

Note that for the UNION operator to remove duplicate rows, all columns in a given row must be equal to the same columns in one or more other rows. When Oracle processes a UNION, it must perform a sort/merge to determine which rows are duplicates. Thus, your execution time will likely be more than running each SELECT individually. If you know there are no duplicates within and across each SELECT statement, you can use UNION ALL to combine the results without checking for duplicates.
If there are duplicates, it will not cause an error; you will merely get duplicate rows in your result set.

3-3. Writing an Optional Join

Problem

You are joining two tables by one or more common columns, but you want to make sure to return all rows in the first table regardless of a matching row in the second. For example, you are joining the employee and department tables, but some employees lack department assignments.

Solution

Use an outer join. In Oracle's sample database, the HR user maintains the EMPLOYEES and DEPARTMENTS tables; assigning a department to an employee is optional. There are 107 employees in the EMPLOYEES table. Using a standard join between EMPLOYEES and DEPARTMENTS only returns 106 rows, however, since one employee is not assigned a department. To return all rows in the EMPLOYEES table, you can use LEFT OUTER JOIN to include all rows in the EMPLOYEES table and matching rows in DEPARTMENTS, if any:

```
select employee_id, last_name, first_name, department_id, department_name
from employees
   left outer join departments using(department_id)
;
```

```
EMPLOYEE_ID  LAST_NAME       FIRST_NAME      DEPARTMENT_ID     DEPARTMENT_NAME
------------ --------------- --------------- ----------------- -------------------
200          Whalen          Jennifer        10                Administration
202          Fay             Pat             20                Marketing
201          Hartstein       Michael         20                Marketing
119          Colmenares      Karen           30                Purchasing
. . .
206          Gietz           William         110               Accounting
205          Higgins         Shelley         110               Accounting
178          Grant           Kimberely

107 rows selected
```

There are now 107 rows in the result set instead of 106; Kimberely Grant is included even though she does not currently have a department assigned.

How It Works

When two tables are joined using `LEFT OUTER JOIN`, the query returns all the rows in the table to the left of the `LEFT OUTER JOIN` clause, as you might expect. Rows in the table on the right side of the `LEFT OUTER JOIN` clause are matched when possible. If there is no match, columns from the table on the right side will contain `NULL` values in the results.

As you might expect, there is a `RIGHT OUTER JOIN` as well (in both cases, the `OUTER` keyword is optional). You can rewrite the solution as follows:

```
select employee_id, last_name, first_name, department_id, department_name
from departments
   right outer join employees using(department_id)
;
```

The results are identical, and which format you use depends on readability and style.

The query can be written using the `ON` clause as well, just as with an equi-join (inner join). And for versions of Oracle before 9*i*, you must use Oracle's somewhat obtuse and proprietary outer-join syntax with the characters (+) on the side of the query that is missing rows, as in this example:

```
select employee_id, last_name, first_name, e.department_id, department_name
from employees e, departments d
where e.department_id = d.department_id (+)
;
```

Needless to say, if you can use ANSI SQL-99 syntax, by all means do so for clarity and ease of maintenance.

3-4. Making a Join Optional in Both Directions

Problem

All of the tables in your query have at least a few rows that don't match rows in the other tables, but you still want to return all rows from all tables and show the mismatches in the results. For example, you want to reduce the number of reports by including mismatches from both tables instead of having one report for each scenario.

Solution

Use `FULL OUTER JOIN`. As you might expect, a full outer join between two or more tables will return all rows in each table of the query and match where possible. You can use `FULL OUTER JOIN` with the `EMPLOYEES` and `DEPARTMENTS` table as follows:

```
select employee_id, last_name, first_name, department_id, department_name
from employees
   full outer join departments using(department_id)
;

EMPLOYEE_ID  LAST_NAME         FIRST_NAME    DEPARTMENT_ID    DEPARTMENT_NAME
------------ ----------------- ------------- ---------------- --------------------
100          King              Steven        90               Executive
101          Kochhar           Neena         90               Executive
102          De Haan           Lex           90               Executive
. . .
177          Livingston        Jack          80               Sales
178          Grant             Kimberely
179          Johnson           Charles       80               Sales
. . .
206          Gietz             William       110              Accounting
                                             180              Construction
                                             190              Contracting
                                             230              IT Helpdesk

123 rows selected
```

■ **Note** The OUTER keyword is optional when using a FULL, LEFT, or RIGHT join. It does add documentation value to your query, making it clear that mismatched rows from one or both tables will be in the results.

Using `FULL OUTER JOIN` is a good way to view, at a glance, mismatches between two tables. In the preceding output, you can see an employee without a department as well as several departments that have no employees.

How It Works

Trying to accomplish a full outer join before Oracle9*i* was a bit inelegant: you had to perform a `UNION` of two outer joins (a left and a right outer join) using the proprietary Oracle syntax as follows:

```
select employee_id, last_name, first_name, e.department_id, department_name
from employees e, departments d
where e.department_id = d.department_id (+)
union
select employee_id, last_name, first_name, e.department_id, department_name
from employees e, departments d
where e.department_id (+) = d.department_id
;
```

Running two separate queries, then removing duplicates, takes more time to execute than using the `FULL OUTER JOIN` syntax, where only one pass on each table is required.

You can tweak the FULL OUTER JOIN to produce only the mismatched records as follows:

```
select employee_id, last_name, first_name, department_id, department_name
from employees
   full outer join departments using(department_id)
where employee_id is null or department_name is null
;
```

3-5. Removing Rows Based on Data in Other Tables

Problem

You want to delete rows from a table if corresponding rows exist in a second table. For example, you want to delete rows from the EMPLOYEES_RETIRED table for any employees that exist in the EMPLOYEES table.

Solution

Use the IN or EXISTS clause with a subquery. You have a table called EMPLOYEES_RETIRED that should contain only—you guessed it—retired employees. However, the EMPLOYEES_RETIRED table erroneously includes some active employees, so you want to remove any active employees from the EMPLOYEES_RETIRED table. Here's how you can do that:

```
delete from employees_retired
where employee_id
   in (select employee_id from employees)
;
```

How It Works

When you use SELECT, you have the relative luxury of using a join condition to return results. When deleting rows, you can't perform an explicit join unless the join conditions are in a subquery or you use an inline view as follows:

```
delete (
        select employee_id
        from employees_retired join employees using(employee_id)
       );
```

The SQL standard treats views much like tables, in that you can not only run a SELECT statement against a view, but also INSERT, UPDATE and DELETE under certain circumstances. If these circumstances are met (for example, you have a key-preserved table, no aggregates, and so forth), DELETE will delete only from the first table in the FROM clause. Be careful: if your DELETE looks like the following, you will not get the intended results:

```
delete (
        select employee_id
        from employees join employees_retired using(employee_id)
       );
```

The rows will be deleted from the EMPLOYEES table instead, and that is not the desired result!
Another potential solution to the problem uses an EXISTS clause to determine which rows to delete:

```
delete from employees_retired er
where exists (
              select 1 from employees e
              where er.employee_id = e.employee_id
             )
;
```

This is not as elegant as the first solution, but might be more efficient. It appears contrived, and it is, because Oracle never uses the results of the query anywhere. It only uses the subquery to verify the existence of a match, and then deletes the corresponding row(s). You can use a "1", an "X", or even NULL; internally, Oracle translates the result to a zero and does not use it.

Whether you use IN or EXISTS depends on the sizes of the driving table (the outer table referenced in the SELECT, UPDATE, or DELETE) and the size of the result set in the subquery. Using IN is most likely better if the results of the subquery are small or is a list of constants. However, using EXISTS may run a lot more efficiently since the implicit JOIN may take advantage of indexes. The bottom line is, it depends. Look at the execution plans for each method (IN versus EXISTS), and if the relative row counts remain stable whenever the query is run, you can use whichever method is faster.

3-6. Finding Matched Data Across Tables

Problem

You want to find the rows in common between two or more tables or queries.

Solution

Use the INTERSECT operator. When you use INTERSECT, the resulting row set contains only rows that are in common between the two tables or queries:

```
select count(*) from employees_act;

COUNT(*)
----------------------
6

select count(*) from employees_new;

COUNT(*)
----------------------
5
```

```
select * from employees_act
intersect
select * from employees_new
;

EMPLOYEE_ID            FIRST_NAME           LAST_NAME                     . . .
---------------------- -------------------- -------------------------
105                    David                Austin
112                    Jose Manuel          Urman

2 rows selected
```

How It Works

The INTERSECT operator, along with UNION, UNION ALL, and MINUS, joins two or more queries together. As of Oracle Database 11*g*, these operators have equal precedence, and unless you override them with parentheses, they are evaluated in left-to-right order. Or, more intuitively since you usually don't have two queries on one line, top-to-bottom order!

■ **Tip** Future ANSI SQL standards give the INTERSECT operator higher precedence than the other operators. Thus, to "bulletproof" your SQL code, use parentheses to explicitly specify evaluation order where you use INTERSECT with other set operators.

To better understand how INTERSECT works, Figure 3-1 shows a Venn diagram representation of the INTERSECT operation on two queries.

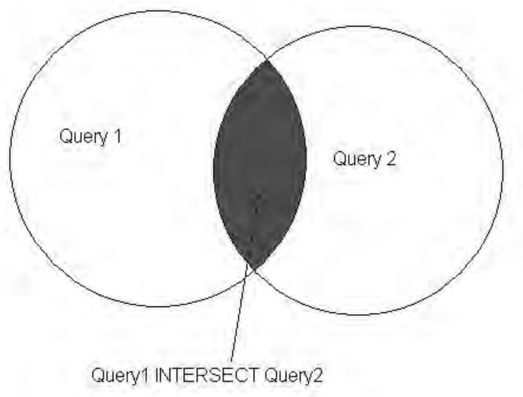


Figure 3-1. *An Oracle INTERSECT operation*

Finally, for an INTERSECT operation to return the intended results, the corresponding columns in each query of the INTERSECT operation should have data types within the same class: numbers, characters, or date/time. For example, a NUMBER column and a BINARY_DOUBLE column will compare correctly if the conversion of the NUMBER column value to a BINARY_DOUBLE produces the same result. So, a NUMBER(3) column containing the value 250 will compare exactly to a BINARY_DOUBLE having the value 2.5E2.

Using set operators is one of the rare cases where a value of NULL in one column is considered equal to another column containing a NULL. Therefore, running this query returns one row:

```
select 'X' C1, NULL C2 from DUAL
intersect
select 'X' C1, NULL C2 from DUAL
;
```

3-7. Joining on Aggregates

Problem

You want to compare values in individual rows to aggregate values computed from the same table; you also want to filter rows that are used for the aggregate. For example: you want to find all employees whose salary is 20% less than the average salary for the company, except for employees who are in the executive department.

Solution

Use a subquery with an aggregate, and compare the employee's salary to the average salary retrieved in the aggregate. For example, this query uses a subquery to calculate the average salary for employees outside of department 90, and compares every employee's salary to 80% of the result from the subquery:

```
select employee_id, last_name, first_name, salary
from employees
where salary < 0.8 * (
                      select avg(salary)
                      from employees
                      where department_id != 90
                     )
;

EMPLOYEE_ID       LAST_NAME           FIRST_NAME           SALARY
----------------- ------------------- -------------------- ----------------
105               Austin              David                4800
106               Pataballa           Valli                4800
107               Lorentz             Diana                4200
115               Khoo                Alexander            3100
. . .
198               OConnell            Donald               2600
199               Grant               Douglas              2600
200               Whalen              Jennifer             4400

49 rows selected
```

How It Works

The presented solution returns one row in the subquery. The Oracle optimizer calculates the average salary once, multiplies it by 0.8, and compares it to the salary of all other employees in the EMPLOYEE table other than those in department 90 (the executive group).

If your subquery returns more than one row, the main query will return an error since you are using the < operator; operators such as <, >, =, >=, <=, and != compare a column's or expression's value to another single value. To compare a value in the main query to a list of one or more values in a subquery, you can use the ANY or SOME operator (they are equivalent) in the WHERE clause. For example, if you wanted to return all employees whose salary matches any of the employees in the executive department, you can do this:

```
select employee_id, last_name, first_name, salary
from employees
where salary = any (
                    select salary
                    from employees
                    where department_id = 90
                   )
;
```

You can also use subqueries in the HAVING clause of a query that uses aggregates. In this example, you can retrieve all departments and average salaries whose average salary is greater than the average salary of the IT department:

```
select department_id, avg(salary) avg_salary
from employees
group by department_id
having avg(salary) > (
                      select avg(salary)
                      from employees
                      where department_id = 60
                     )
;
```

3-8. Finding Missing Rows

Problem

You have two tables, and you must find rows in the first table that are not in the second table. You want to compare all rows in each table, not just a subset of columns.

Solution

Use the MINUS set operator. The MINUS operator will return all rows in the first query that are not in the second query. The EMPLOYEES_BONUS table contains employees who have been given bonuses in the past, and you need to find employees in the EMPLOYEES table who have not yet received bonuses. Use the MINUS operator as follows to compare three selected columns from two tables:

```
select employee_id, last_name, first_name from employees
minus
select employee_id, last_name, first_name from employees_bonus
;

EMPLOYEE_ID            LAST_NAME                 FIRST_NAME
---------------------- ------------------------- --------------------
100                    King                      Steven
109                    Faviet                    Daniel
110                    Chen                      John
120                    Weiss                     Matthew
140                    Patel                     Joshua

5 rows selected
```

How It Works

Note that unlike the INTERSECT and UNION operators, the MINUS set operator is not commutative: the order of the operands (queries) is important! Changing the order of the queries in the solution will produce very different results.

If you wanted to note changes for the entire row, you could use this query instead:

```
select * from employees
minus
select * from employees_bonus
;
```

A Venn diagram may help to show how the MINUS operator works. Figure 3-2 shows the result of Query1 MINUS Query2. Any rows that overlap between Query1 and Query2 are removed from the result set along with any rows in Query2 that do not overlap Query1. In other words, only rows in Query1 are returned less any rows in Query1 that exist in Query2.

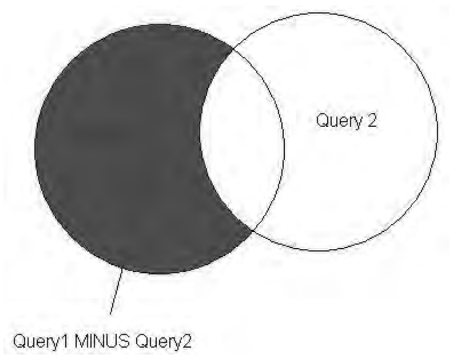


Figure 3-2. *An Oracle MINUS operation*

Rows in Query2 that do not exist in Query1 can be identified by reversing the order of the operands in the query:

```
select * from employees_bonus
minus
select * from employees
;
```

What if Query2 is a proper subset of Query1? In other words, all the rows in Query2 are already in Query1? The query still works as advertised; `MINUS` only removes rows from Query1 that are in common with Query2, and never returns any rows in Query2 that are not in Query1. Figure 3-3 shows the Venn diagram for the scenario where Query2 is a proper subset of Query1.

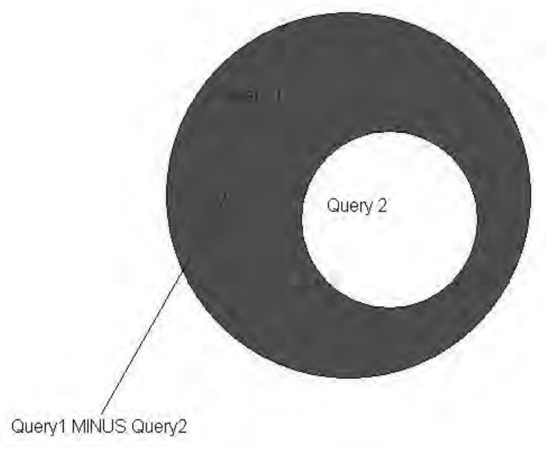


Figure 3-3. *An Oracle MINUS operation with proper subsets*

You may ask why an outer join or an `IN/EXISTS` query might solve most problems like this. For sets of tables with a single primary key, it may be more efficient to use a join. However, for query results with a large number of columns or for comparing entire rows of data even when the primary keys match, `MINUS` is the more appropriate operator to use.

3-9. Finding Rows that Tables Do Not Have in Common

Problem

You want to find rows from two queries or tables that the two queries or tables do NOT have in common. The analysts have provided you with Venn diagrams and the set notation for retrieving the required data, so you must use a combination of Oracle set operators to retrieve the desired result.

Solution

Use aggregate functions and the `UNION ALL` operator together in a compound query, parenthesizing for clarity and correctness. For example, you want to find the list of employees that are in the `EMPLOYEES_ACT` table but not in the `EMPLOYEES_NEW` table, and vice versa. Execute the following queries to show the contents of each table and the results of using Oracle set operators to find the unique rows in each table:

```
select employee_id, first_name, last_name from employees_act
order by employee_id;

EMPLOYEE_ID            FIRST_NAME           LAST_NAME
---------------------- -------------------- -------------------------
102                    Lex                  De Haan
105                    David                Austin
112                    Jose Manuel          Urman
118                    Guy                  Himuro
119                    Karen                Colmenares
205                    Shelley              Higgins

6 rows selected

select employee_id, first_name, last_name from employees_new
order by employee_id;

EMPLOYEE_ID            FIRST_NAME           LAST_NAME
---------------------- -------------------- -------------------------
101                    Neena                Kochhar
105                    David                Austin
112                    Jose Manuel          Urman
171                    William              Smith
201                    Michael              Hartstein

5 rows selected

select employee_id, first_name, last_name,
   count(act_emp_src) act_emp_row_count,
   count(new_emp_src) new_emp_row_count
from
   (
    select ea.*, 1 act_emp_src, to_number(NULL) new_emp_src
    from employees_act ea
    union all
    select en.*, to_number(NULL) act_emp_src, 1 new_emp_src
    from employees_new en
   )
group by employee_id, first_name, last_name
having count(act_emp_src) != count(new_emp_src)
;
```

```
EMPLOYEE_ID FIRST_NAME  LAST_NAME       ACT_EMP_ROW_COUNT NEW_EMP_ROW_COUNT
----------- ----------- --------------- ----------------- -----------------
101         Neena       Kochhar         0                 1
205         Shelley     Higgins         1                 0
102         Lex         De Haan         1                 0
171         William     Smith           0                 1
201         Michael     Hartstein       0                 1
118         Guy         Himuro          1                 0
119         Karen       Colmenares      1                 0

7 rows selected
```

How It Works

To get the correct result, you must first combine both result sets using `UNION ALL` in the subquery, assigning a "tag" column to indicate where the row came from—the first table or the second. Here, we use a "1", but it would work fine with a "2", or an "X".

The `GROUP BY` and `HAVING` clauses pick out the common rows. If a given row exists in both tables once, or several times, the count for that row will be the same, and thus will be excluded by the condition in the `HAVING` clause. If the counts are not the same, the row will show up in the result along with how many times it appears in one table but not the other, and vice versa. Because we are including primary keys in this query, you will not see more than one non-common row in each table.

Figure 3-4 shows the Venn diagram for the result set.

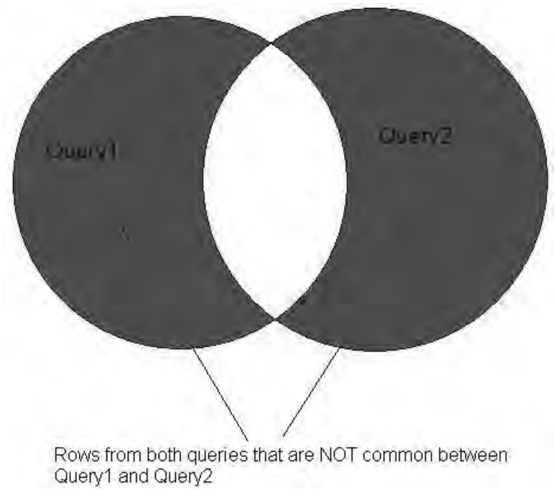

Query1
Query2
Rows from both queries that are NOT common between Query1 and Query2

Figure 3-4. *An Oracle set operation to find unique rows between component queries*

Using standard set notation, the result you are looking for can be more succinctly expressed as follows:

```
NOT (Query1 INTERSECT Query2)
```

Although Oracle syntax does have NOT and INTERSECT, NOT is not supported for use with Oracle set operators. Thus, you must write the query as shown in the solution with UNION ALL and GROUP BY to provide the result identified in Figure 3-4. Even if Oracle supported NOT with set operators, you would still have to use the solution provided to identify how many of each non-common row exists in each component query.

3-10. Generating Test Data

Problem

You want to combine two or more tables without common columns to generate test data or a template for all possible combinations of values from two or more tables.

For example, you are writing an embedded Oracle database application to help you count cards at the next blackjack game, and you want to generate the template table for the 52-card deck with a minimum of effort.

Solution

Use a CROSS JOIN construct between the set of four suits (hearts, clubs, diamonds, and spades) and the 13 rank values (Ace, 2-10, Jack, Queen, and King). For example:

```
create table card_deck
   (suit_rank        varchar2(5),
    card_count       number)
;

insert into card_deck
      select rank || '-' || suit, 0 from
         (select 'A' rank from dual
          union all
          select '2' from dual
          union all
          select '3' from dual
          union all
          select '4' from dual
          union all
          select '5' from dual
          union all
          select '6' from dual
          union all
          select '7' from dual
          union all
          select '8' from dual
          union all
          select '9' from dual
          union all
          select '10' from dual
          union all
          select 'J' from dual
          union all
          select 'Q' from dual
          union all
          select 'K' from dual)
      cross join
         (select 'S' suit from dual
          union all
          select 'H' from dual
          union all
          select 'D' from dual
          union all
          select 'C' from dual)
;

select suit_rank from card_deck;

SUIT_RANK
---------
A-S
2-S
3-S
4-S
5-S
6-S
. . .
10-C
J-C
Q-C
K-C

52 rows selected
```

The Dual Table

The DUAL table comes in handy in many situations. It is available in every version of Oracle from the last 20 years, and contains one row and one column. It is handy when you want to return a row when you don't have to retrieve data from any particular table—you just want to return a value. You would also use the DUAL table to perform an ad-hoc calculation using a built-in function, as in this example:

```
select sqrt(144) from dual;

SQRT(144)

-------------------

12
```

This SQL statement would work just as well if you already had the suits and ranks in two existing tables. In that case, you would reference the two tables rather than write those long subqueries involving UNION operations against the DUAL table. For example:

```
select rank || '-' || suit, 0 from
    card_ranks
cross join
    card_suits
;
```

How It Works

Using a CROSS JOIN, otherwise known as a Cartesian product, is rarely intentional. If you specify no join conditions between two tables, the number of rows in the results is the product of the number of rows in each table. Usually this is not the desired result! As a general rule, for a query with *n* tables, you need to specify at least *n-1* join conditions to avoid a Cartesian product.

If your card games usually involve using one or two jokers in the deck, you can tack them onto the end as easily as this:

```
select rank || '-' || suit, 0 from
   (select 'A' rank from dual
    union all
    select '2' from dual
. . .
    select 'C' from dual)
union (select 'J1', 0 from dual)
union (select 'J2', 0 from dual)
;
```

Using the pre-Oracle9*i* syntax (Oracle proprietary syntax), you can rewrite the original query as follows:

```
select rank || '-' || suit, 0 from
    card_ranks, card_suits
;
```

It's odd to see a multi-table join using the proprietary Oracle syntax and no WHERE clause, but it produces the results you want!

3-11. Updating Rows Based on Data in Other Tables

Problem

You want to update some or all rows in your table based on individual values or aggregates in the same or another table. The values used to update the query are dependent on column values in the table to be updated.

Solution

Use a correlated update to link the values in the subquery to the main table in the UPDATE clause. In the following example, you want to update all employee salaries to 10% more than the average salary for their department:

```
update employees e
   set salary = (select avg(salary)*1.10
                 from employees se
                 where se.department_id = e.department_id)
;

107 rows updated
```

How It Works

For each row in the EMPLOYEES table, the current salary is updated with 10% more than the average salary for their department. The subquery will always return a single row since it has an aggregate with no GROUP BY. Note that this query shows Oracle's read consistency feature in action: Oracle preserves the original salary value for each employee to compute the average while updating each employee's salary in the same UPDATE statement.

Correlated updates are much like correlated subqueries (discussed elsewhere in this chapter): you link one or more columns in the main part of the query with the corresponding columns in the subquery. When performing correlated updates, take special note of how many rows are updated; coding the subquery incorrectly can often update most values in the table with NULL or incorrect values!

In fact, the solution provided will fail if an employee does not have a department assigned. The correlated subquery will return NULL in this scenario because a NULL department value will never match any other department values that are NULL, and thus the employee will be assigned a SALARY of NULL. To fix this problem, add a filter in the WHERE clause:

```
update employees e
   set salary = (select avg(salary)*1.10
                 from employees se
                 where se.department_id = e.department_id)
where department_id is not null
;

106 rows updated
```

All employees without an assigned department will keep their existing salary. Another way to accomplish the same thing is by using the NVL function in the SET clause to check for NULL values:

```
update employees e
   set salary = nvl((select avg(salary)*1.10
                     from employees se
                     where se.department_id = e.department_id),salary)
;

107 rows updated
```

This option may not be as desirable from an I/O perspective or in terms of redo log file usage, since all rows will be updated whether they have a department assigned or not.

3-12. Manipulating and Comparing NULLs in Join Conditions

Problem

You want to map NULL values in a join column to a default value that will match a row in the joined table, thus avoiding the use of an outer join.

Solution

Use the NVL function to convert NULL values in the foreign key column of the table to be joined to its parent table. In this example, holiday parties are scheduled for each department, but several employees do not have a department assigned. Here is one of them:

```
select employee_id, first_name, last_name, department_id
from employees
where employee_id = 178
;

EMPLOYEE_ID   FIRST_NAME      LAST_NAME       DEPARTMENT_ID
------------- --------------- --------------- -----------------
178           Kimberely       Grant

1 rows selected
```

To ensure that each employee will attend a holiday party, convert all NULL department codes in the EMPLOYEES table to department 110 (Accounting) in the query as follows:

```
select employee_id, first_name, last_name, d.department_id, department_name
from employees e join departments d
   on nvl(e.department_id,110) = d.department_id
;

EMPLOYEE_ID   FIRST_NAME      LAST_NAME       DEPARTMENT_ID     DEPARTMENT_NAME
------------- --------------- --------------- ----------------- ------------------
200           Jennifer        Whalen          10                Administration
201           Michael         Hartstein       20                Marketing
202           Pat             Fay             20                Marketing
114           Den             Raphaely        30                Purchasing
115           Alexander       Khoo            30                Purchasing
. . .
```

```
113           Luis          Popp          100             Finance
178           Kimberely     Grant         110             Accounting
205           Shelley       Higgins       110             Accounting
206           William       Gietz         110             Accounting

107 rows selected
```

How It Works

Mapping columns with `NULL`s to non-`NULL` values in a join condition to avoid using an `OUTER JOIN` might still have performance problems, since the index on `EMPLOYEES.DEPARTMENT_ID` will not be used during the join (primarily because `NULL` columns are not indexed). You can address this new problem by using a function-based index (FBI). An FBI creates an index on an expression, and may use that index if the expression appears in a join condition or a `WHERE` clause. Here is how to create an FBI on the `DEPARTMENT_ID` column:

```
create index employees_dept_fbi on employees(nvl(department_id,110));
```

■ **Tip** As of Oracle Database 11*g*, you can now use *virtual columns* as an alternative to FBIs. Virtual columns are derived from constants, functions, and columns from the same table. You can define indexes on the virtual columns, which the optimizer will use in any query as it would a regular column index.

Ultimately, the key to handling `NULL`s in join conditions is based on knowing your data. If at all possible, avoid joining on columns that may have `NULL`s, or ensure that every column you will join on has a default value when it is populated. If the column must have `NULL`s, use functions like `NVL`, `NVL2`, and `COALESCE` to convert `NULL`s before joining; you can create function-based indexes to offset any performance issues with joining on expressions. If that is not possible, understand the business rules about what `NULL`s mean in your database columns: do they mean zero, unknown, or not applicable? Your SQL code must reflect the business definition of columns that can have `NULL`s.

CHAPTER 4

■ ■ ■

Creating and Deriving Data

Storing your data in the database is only half the story. The other half is making that data available to users in ways they can comprehend. When you extract data from the database, either in a report format or one row at a time on a data maintenance page, the values in each row are often not suitable for display for a number of reasons. Usually, your data is highly normalized for performance and maintenance reasons—storing each part of a name in a different column or using a numeric code set that references the full text descriptions in another table. Other normalization-related scenarios include storing numeric quantities or prices across several rows or within several columns in the same row, and not storing a sum or other transformations within the table.

As a result of a highly normalized database, you often must perform some transformations to make it easy for end-users to interpret the results of a report by creating new values on the fly. You may want to perform transformations on single columns, on several columns within a single row, or on one or more columns across multiple rows within one or many tables. This chapter has recipes to cover all of these scenarios.

Also, your data might not be in the correct format or have erroneous values; you are obligated as a DBA or data analyst to ensure that the data is clean when it is entered into your database. Thus, there are some recipes in this chapter that will help you cleanse the data before it is stored in your database, or transform it to a format suitable for export to a downstream system.

Finally, this chapter will provide some insight on how to generate random data for your database. For load testing, it is often useful to create a random distribution of data to thoroughly test execution paths in your application before going live.

4-1. Deriving New Columns

Problem

You don't want to store redundant data in your database tables, but you want to be able to create totals, derived values, or alternate formats for columns within a row.

Solution

In your SELECT statements, apply Oracle built-in functions or create expressions on one or more columns in the table, creating a virtual column in the query results. For example, suppose you want to summarize total compensation for an employee, combining salary and commissions.

In the sample tables included for the OE user from a default installation of Oracle, the ORDER_ITEMS table contains UNIT_PRICE and QUANTITY columns as follows:

```
select * from order_items;

ORDER_ID  LINE_ITEM_ID PRODUCT_ID UNIT_PRICE QUANTITY
--------- ------------ ---------- ---------- --------
2423      7            3290       65         33
2426      5            3252       25         29
2427      7            2522       40         22
2428      11           3173       86         28
2429      10           3165       36         67
. . .
2418      6            3150       17         37
2419      9            3167       54         81
2420      10           3171       132        47
2421      9            3155       43         185
2422      9            3167       54         39

665 rows selected
```

To provide a line-item total in the query results, add an expression that multiplies unit price by quantity, as follows:

```
select order_id, line_item_id, product_id,
   unit_price, quantity, unit_price*quantity line_total_price
from order_items;

ORDER_ID  LINE_ITEM_ID PRODUCT_ID UNIT_PRICE QUANTITY LINE_TOTAL_PRICE
--------- ------------ ---------- ---------- -------- ----------------
2423      7            3290       65         33       2145
2426      5            3252       25         29       725
2427      7            2522       40         22       880
2428      11           3173       86         28       2408
2429      10           3165       36         67       2412
. . .
2418      6            3150       17         37       629
2419      9            3167       54         81       4374
2420      10           3171       132        47       6204
2421      9            3155       43         185      7955
2422      9            3167       54         39       2106

665 rows selected
```

How It Works

In a SELECT statement (or any DML statement, such as INSERT, UPDATE, or DELETE), you can reference any Oracle built-in function (or one of your own functions) on any or all columns in the statement. You can include aggregate functions such as SUM() and AVG() as well, as long as you include the non-aggregated columns in a GROUP BY clause.

You can also nest your functions within a SELECT statement. The column LINE_TOTAL_PRICE contains the desired result, but is not formatted appropriately for displaying to a customer service representative or a customer. To improve the readability of the output, add the TO_CHAR function to the calculation already in the query as follows:

```
select order_id, line_item_id, product_id,
   unit_price, quantity,
   to_char(unit_price*quantity,'$9,999,999.99') line_total_price
from order_items;

ORDER_ID  LINE_ITEM_ID PRODUCT_ID UNIT_PRICE QUANTITY LINE_TOTAL_PRICE
--------- ------------ ---------- ---------- -------- ----------------
2423      7            3290       65         33            $2,145.00
2426      5            3252       25         29              $725.00
2427      7            2522       40         22              $880.00
2428      11           3173       86         28            $2,408.00
2429      10           3165       36         67            $2,412.00
. . .
2418      6            3150       17         37              $629.00
2419      9            3167       54         81            $4,374.00
2420      10           3171       132        47            $6,204.00
2421      9            3155       43         185           $7,955.00
2422      9            3167       54         39            $2,106.00

665 rows selected
```

The TO_CHAR function can also be used to convert date or timestamp columns to a more readable or non-default format; you can even use TO_CHAR to convert data types in the national character set (NCHAR, NVARCHAR2, NCLOB) and CLOBs to the database character set, returning a VARCHAR2.

If your users need to run this query often, you might consider creating a database *view*, which will make it even easier for your analysts and other end-users to access the data. The new view will look like another table with one more column than the original table. Use the CREATE VIEW statement to create a view on the ORDER_ITEMS table, including the derived column:

```
create view order_items_subtotal_vw as
select order_id, line_item_id, product_id,
   unit_price, quantity,
   to_char(unit_price*quantity,'$9,999,999.99') line_total_price
from order_items
;
```

To access the table's columns and the derived column from the view, use this syntax:

```
select order_id, line_item_id, product_id,
   unit_price, quantity, line_total_price
from order_items_subtotal_vw;
```

The next time you need to access the derived LINE_TOTAL_PRICE column, you don't need to remember the calculations or the function you used to convert the total price to a currency format!

Using a database view in this scenario is useful, but adds another object to the data dictionary and thus creates another maintenance point if the base table changes, as when you modify or remove the UNIT_PRICE or QUANTITY columns. If you are using Oracle Database 11*g* or later, you can use a feature called *virtual columns*, which essentially creates the metadata for a derived column within the table definition itself. You can create a virtual column when you create a table, or add it later as you would any other column. The virtual column definition takes up no space in the table itself; it is calculated on the fly when you access the table containing the virtual column. To add the virtual column LINE_TOTAL_PRICE to ORDER_ITEMS, use this syntax:

```
alter table order_items
   add (line_total_price as (to_char(unit_price*quantity,'$9,999,999.99')));
```

The virtual column appears to be a regular column like any of the real columns in the table:

```
describe order_items;

Name                           Null     Type
------------------------------ -------- -----------------
ORDER_ID                       NOT NULL NUMBER(12)
LINE_ITEM_ID                   NOT NULL NUMBER(3)
PRODUCT_ID                     NOT NULL NUMBER(6)
UNIT_PRICE                              NUMBER(8,2)
QUANTITY                                NUMBER(8)
LINE_TOTAL_PRICE                        VARCHAR2(14)

6 rows selected
```

The data type of the virtual column is implied by the result of the calculation in the virtual column definition; the result of the expression in the LINE_TOTAL_PRICE virtual column is a character string with a maximum width of 14, thus the data type of the virtual column is VARCHAR2(14).

■ **Tip** To find out if a column is a virtual column, you can query the VIRTUAL_COLUMN column of the *_TAB_COLS data dictionary views.

There are a number of advantages to using virtual columns, such not having to create a view on the table, as we did in the previous example. In addition, you can create indexes on virtual columns and collect statistics on them. Here is an example of creating an index on the virtual column from the previous example:

```
create index ie1_order_items on order_items(line_total_price);
```

■ **Note** For versions of Oracle that do not support virtual columns (before Oracle Database 11*g*), you can somewhat simulate the indexing feature of a virtual column by creating a function-based index on the expression used in the virtual column.

There are a few restrictions on virtual columns, some of them obvious, some not so much. If your virtual column contains a built-in or user function, the referenced function must be declared as DETERMINISTIC; in other words, the function must return the same results for the same set of input values (for example, other columns in the row). Therefore, you can't use functions such as SYSDATE or SYSTIMESTAMP in a virtual column, as they will almost always return different results every time the row is retrieved (unless the results are retrieved within the same second for SYSDATE or millionth of a second for SYSTIMESTAMP!). If you declare a user function to be DETERMINISTIC when it's not and reference it in a virtual column, you are asking for trouble; your query results that include the virtual column will most likely be incorrect or will produce different results depending on when you run it, who is running it, and so forth.

Other restrictions on virtual columns are not so obvious. A virtual column can't reference another virtual column in the table (although you can certainly include one virtual column's definition within the definition of the second virtual column). In addition, the virtual column can only reference columns within the same table. Overall, virtual columns can enhance your database usability while somewhat reducing the amount of metadata you might otherwise have to maintain in a database view.

4-2. Returning Nonexistent Rows

Problem

You have gaps in the sequence numbers used for a table's primary key, and you want to identify the intermediate sequence numbers.

Solution

Although primary keys are generally (and preferably) invisible to the end user, there are times you may want to reuse sequence numbers where the number of digits in the sequence number must be kept as low as possible to satisfy the requirements of downstream systems or processes. For example, the company baseball team uses the number in EMPLOYEE_ID column as the number on the back of the uniform jersey. The manager wants to reuse old jersey numbers if possible, since the number on the back of the jersey is limited to three digits.

To find the employee numbers that were skipped or are currently not in use, you can use a query with two embedded subqueries as follows:

```
with all_used_emp_ids as
   (select level poss_emp_id from (select max(employee_id) max_emp_num
                                   from employees)
    connect by level <= max_emp_num)
select poss_emp_id
from all_used_emp_ids
```

```
where poss_emp_id not in (select employee_id from employees)
order by poss_emp_id
;

POSS_EMP_ID
----------------------
1
2
3
4
5
. . .
99
179
183

101 rows selected
```

The query retrieves any unused employee numbers up to the highest existing employee number currently in the EMPLOYEES table.

How It Works

The solution uses subquery factoring and two explicit subqueries to find the list of skipped employee numbers. The query in the WITH clause creates a series of numbers (with no gaps) from one to the value of the highest employee number in the EMPLOYEES table. The highlighted part of the subquery that follows retrieves the highest number:

```
(select level poss_emp_id from (select max(employee_id) max_emp_num
                                       from employees)
    connect by level <= max_emp_num)
```

The rest of the query in the WITH clause generates the gapless sequence of numbers. It leverages a subtle feature of Oracle's hierarchical query features (the CONNECT BY clause and the LEVEL pseudo-column) to generate each number in the sequence. See Chapter 13 for other useful recipes that require traversing hierarchical data.

Here is the main part of the SELECT query:

```
select poss_emp_id
from all_used_emp_ids
where poss_emp_id not in (select employee_id from employees)
```

It retrieves all rows from the gapless sequence with possible employee numbers that do not already exist in the EMPLOYEES table. The final result set is ordered by POSS_EMP_ID to make it easy to assign the lowest available number to the next employee. If you want to always exclude one- or two-digit numbers, you can add another predicate to the WHERE clause as follows:

```
with all_used_emp_ids as
   (select level poss_emp_id from (select max(employee_id) max_emp_num
                                   from employees)
    connect by level <= max_emp_num)
```

```
select poss_emp_id
from all_used_emp_ids
where poss_emp_id not in (select employee_id from employees)
  and poss_emp_id > 99
order by poss_emp_id
;

POSS_EMP_ID
----------------------
179
183

2 rows selected
```

In the revised query's results, the only three-digit numbers that can be reused below the highest employee number currently in use are 179 and 183.

4-3. Changing Rows into Columns

Problem

Your transaction data is in a highly normalized database structure, and you want to easily create crosstab-style reports for the business analysts, returning totals or other aggregate results as multiple columns within a single row.

Solution

Use the `PIVOT` keyword in your `SELECT` statement to spread values (in one or many columns) from multiple rows aggregated into multiple columns in the query output. For example, in the Oracle order entry sample schema, `OE`, you want to pick five key products from all orders and find out which customers are buying these products, and how many they bought.

The `ORDERS` table and `ORDER_ITEMS` table are defined as follows:

```
describe orders;

Name                           Null     Type
------------------------------ -------- ---------------------------------
ORDER_ID                       NOT NULL NUMBER(12)
ORDER_DATE                     NOT NULL TIMESTAMP(6) WITH LOCAL TIME ZONE
ORDER_MODE                              VARCHAR2(8)
CUSTOMER_ID                    NOT NULL NUMBER(6)
ORDER_STATUS                            NUMBER(2)
ORDER_TOTAL                             NUMBER(8,2)
SALES_REP_ID                            NUMBER(6)
PROMOTION_ID                            NUMBER(6)

8 rows selected
```

```
describe order_items;

Name                           Null     Type
------------------------------ -------- ---------------------------------
ORDER_ID                       NOT NULL NUMBER(12)
LINE_ITEM_ID                   NOT NULL NUMBER(3)
PRODUCT_ID                     NOT NULL NUMBER(6)
UNIT_PRICE                              NUMBER(8,2)
QUANTITY                                NUMBER(8)
LINE_TOTAL_PRICE                        VARCHAR2(14)

6 rows selected
```

Here is the pivot query to retrieve quantity totals for the top five products by customer:

```
with order_item_query as
   (select customer_id, product_id, quantity
    from orders join order_items using(order_id))
select * from order_item_query
pivot (
       sum(quantity) as sum_qty
          for (product_id) in (3170 as P3170,
                               3176 as P3176,
                               3182 as P3182,
                               3163 as P3163,
                               3165 as P3165)
      )
order by customer_id
;

CUSTOMER_ID P3170_SUM_QTY P3176_SUM_QTY P3182_SUM_QTY P3163_SUM_QTY P3165_SUM_QTY
----------- ------------- ------------- ------------- ------------- -------------
101                                                    208
102
103
104         70            72            77             61            64
105
106                       62                           55
107                                                                  76
108                                                    45            31
109                                     115                          112
116         48            24                           5             10
117                                                    63            67
118         42
119         36                                         30
. . .
167
168
169
170                                                    92

47 rows selected
```

From the results of this query, you can easily see that customer number 104 is the top buyer of all of the products specified in the PIVOT clause.

How It Works

The PIVOT operator, as the name implies, pivots rows into columns. In the solution, you will notice a few familiar constructs, and a few not so familiar. First, the solution uses the WITH clause (subquery factoring) to more cleanly separate the base query from the rest of the query; here is the WITH clause used in the solution:

```
with order_item_query as
   (select customer_id, product_id, quantity
    from orders join order_items using(order_id))
```

Whether you use the WITH clause or an inline view is a matter of style. In either case, be sure to only include columns that you will be grouping by, aggregate columns, or columns used in the PIVOT clause. Otherwise, if you have any extra columns in the query, your results will have a lot more rows than you expected! This is much like adding extra columns to a GROUP BY clause in a regular SELECT statement with aggregates, typically increasing the number of rows in the result.

The PIVOT clause itself generates the additional columns in the query results. Here is the first part of the PIVOT clause in the solution:

```
sum(quantity) as sum_qty
```

You specify one or more aggregates that you want to appear in the new columns and assign a label suffix that Oracle will use for each new column. (We'll show you how to pivot on more than one column or aggregate more than one column later in this section.) In this case, all of the derived columns will end in SUM_QTY.

The column or columns in the FOR clause filter the rows that are used to aggregate the results. Each value of the column in the FOR clause, or each specified combination of two or more columns, will produce an additional derived column in the results. The FOR clause in the solution is as follows:

```
          for (product_id) in (3170 as P3170,
                               3176 as P3176,
                               3182 as P3182,
                               3163 as P3163,
                               3165 as P3165)
```

The query will only use rows whose PRODUCT_ID is in the list, and thus acts like a WHERE clause to filter the results before aggregation. The text string specified after each pivoted value is used as the prefix for the new column name, as you can see in the solution query's results.

Finally, the ORDER BY clause does the same thing as it would in any other query—order the final result set by the columns specified.

4-4. Pivoting on Multiple Columns

Problem

You have implemented the solution shown in the previous recipe. Now you wish to pivot not only on the total quantity of products, but also on the total dollar value purchased.

Solution

The following query extends the previous recipe's solution to add a total dollar amount column for each product number:

```
with order_item_query as
   (select customer_id, product_id, quantity, unit_price
    from orders join order_items using(order_id))
select * from order_item_query
pivot (
       sum(quantity) as sum_qty,
       sum(quantity*unit_price) as sum_prc
          for (product_id) in (3170 as P3170,
                               3176 as P3176,
                               3182 as P3182,
                               3163 as P3163,
                               3165 as P3165)
      )
order by customer_id
;
```

```
CUSTOMER_ID P3170_SUM_QTY P3170_SUM_PRC P3176_SUM_QTY P3176_SUM_PRC  . . .
----------- ------------- ------------- ------------- ------------- ---------
101
102
103
104         70            10164         72            8157.6
105
106                                     62            7024.6
107
108
109
116         48            6969.6        24            2719.2
117
118         42            6098.4
119         36            5227.2
. . .
```

This report further reinforces the findings from the original solution: customer number 104 is a big buyer in terms of total dollars paid as well.

How It Works

Once you've mastered the basics of the PIVOT clause, taking it a step further is easy. Just add as many additional aggregates to your PIVOT clause as you need. Remember, though, that adding a second aggregate doubles the number of derived columns, adding a third triples it, and so forth.

Let's extend the solution even more, adding another pivot column. In the ORDERS table, the column ORDER_MODE contains the string direct if the sale was over the phone and ONLINE if the order was placed on the company's Internet site. To pivot on ORDER_MODE, just add the column to the FOR clause and specify pairs of values instead of single values in the FOR list as follows:

```
with order_item_query as
   (select customer_id, product_id, quantity, unit_price, order_mode
    from orders join order_items using(order_id))
select * from order_item_query
pivot (
       sum(quantity) as sum_qty,
       sum(quantity*unit_price) as sum_prc
          for (product_id, order_mode) in ((3170,'direct') as P3170_DIR,
                                           (3170,'online') as P3170_ONL,
                                           (3176,'direct') as P3176_DIR,
                                           (3176,'online') as P3176_ONL,
                                           (3182,'direct') as P3182_DIR,
                                           (3182,'online') as P3182_ONL,
                                           (3163,'direct') as P3163_DIR,
                                           (3163,'online') as P3163_ONL,
                                           (3165,'direct') as P3165_DIR,
                                           (3165,'online') as P3165_ONL)
      )
order by customer_id
;

CUSTOMER_ID P3170_DIR_SUM_QTY P3170_DIR_SUM_PRC P3170_ONL_SUM_QTY P3170_ONL_SUM_PRC
----------- ----------------- ----------------- ----------------- -----------------
101
102
103
104         70                10164
105
106
107
108
109
116         24                3484.8            24                3484.8
117
118         42                6098.4
119
. . .
```

The output has been truncated horizontally for readability. There is one column for each combination of product number, order mode, and the two aggregates. Looking at the breakdown by order mode, it appears that the company could save money by shifting some or all of customer 104's orders to Internet orders.

Finally, you might not know your pivot criteria ahead of time, as new products may appear on a daily basis, making maintenance of your queries an issue. One possible solution is to dynamically build your queries using Java or PL/SQL, though this solution may have potential security and performance issues. However, if the consumer of your query output can accept XML output, you can use the PIVOT XML clause instead of just PIVOT, and include filtering criteria in the IN clause as in this example, revising the original solution:

```
with order_item_query as
   (select customer_id, product_id, quantity
    from orders join order_items using(order_id))
select * from order_item_query
pivot xml (
           sum(quantity) as sum_qty
              for (product_id) in (any)
          )
order by customer_id
;

CUSTOMER_ID            PRODUCT_ID_XML
---------------------- ------------------------------------------------------------
101                    <PivotSet><item><column name =
        "PRODUCT_ID">2264</column><column name = "SUM_QTY">29</column></item> . . .
102                    <PivotSet><item><column name =
        "PRODUCT_ID">2976</column><column name = "SUM_QTY">5</column></item> . . .
103                    <PivotSet><item><column name =
        "PRODUCT_ID">1910</column><column name = "SUM_QTY">6</column></item> . . .
. . .
170                    <PivotSet><item><column name =
         "PRODUCT_ID">3106</column><column name = "SUM_QTY">170</column></item>

47 rows selected
```

The ANY keyword is shorthand for SELECT DISTINCT PRODUCT_ID FROM ORDER_ITEM_QUERY. Instead of ANY, you can put in just about any SQL statement that returns values in the domain you're pivoting on, as in this example where you only want to display results for products that are currently orderable:

```
with order_item_query as
   (select customer_id, product_id, quantity
    from orders join order_items using(order_id))
select * from order_item_query
pivot xml (
           sum(quantity) as sum_qty
              for (product_id) in (select distinct product_id
                                   from product_information
                                   where product_status = 'orderable')
          )
order by customer_id
;
```

Note that you need to include the DISTINCT keyword in your IN clause to ensure that you have only unique values on the pivot columns. Otherwise, the pivot operation will return an error.

4-5. Changing Columns into Rows

Problem

You have a table with multiple columns containing data from the same domain, and you want to convert these columns into rows of a table with a more normalized design.

Solution

Use the UNPIVOT operator after the FROM clause to convert columns into rows.

In the following example, a web form allows a registered account user to sign up for a free vacation holiday with up to three friends. As you might expect, the direct marketing department wants to use the e-mail addresses for other promotions as well. The format of the table (designed by analysts with minimal database design skills!) is as follows:

```
create table email_signup
   (user_account     varchar2(100),
    signup_date      date,
    user_email       varchar2(100),
    friend1_email    varchar2(100),
    friend2_email    varchar2(100),
    friend3_email    varchar2(100))
;
```

The registration data in the first two rows of the table looks like this:

```
USER_ACCOUNT SIGNUP_DATE USER_EMAIL    FRIEND1_EMAIL  FRIEND2_EMAIL  FRIEND3_EMAIL
------------ ----------- ------------- -------------- -------------- --------------

rjbryla      21-AUG-09   rjbryla@      rjbdba@        pensivepenman@ unclebob@
                         example.com   example.com    example.com    example.com

johndoe      22-AUG-09   janedoe@                     dog@
                         example.com                  example.com

2 rows selected
```

Unfortunately, the direct marketing application needs the e-mail address list in a more normalized format as follows:

```
REQUEST_ACCOUNT | REQUEST_DATE | EMAIL_ADDRESS
```

To generate the e-mail list in the format that the direct marketing group needs, you can use the UNPIVOT command like so:

```
select user_account, signup_date, src_col_name, friend_email
from email_signup
unpivot (
         (friend_email) for src_col_name
              in (user_email, friend1_email, friend2_email, friend3_email)
        )
;

USER_ACCOUNT SIGNUP_DATE  SRC_COL_NAME     FRIEND_EMAIL
------------ ------------ ---------------- -------------------------
rjbryla      21-AUG-09    USER_EMAIL       rjbryla@example.com
rjbryla      21-AUG-09    FRIEND1_EMAIL    rjbdba@example.com
rjbryla      21-AUG-09    FRIEND2_EMAIL    pensivepenman@example.com
rjbryla      21-AUG-09    FRIEND3_EMAIL    unclebob@example.com
johndoe      22-AUG-09    USER_EMAIL       janedoe@example.com
johndoe      22-AUG-09    FRIEND2_EMAIL    dog@example.com

6 rows selected.
```

How It Works

In contrast to the PIVOT operator, the UNPIVOT operator changes columns into multiple rows, although you can't reverse engineer any aggregated column totals without the original data—and if you had the original data, you would not need to UNPIVOT! There are many uses for UNPIVOT, as in converting data in a denormalized table or spreadsheet to individual rows for each column or set of columns.

The direct marketing department does not need the column SRC_COL_NAME, but it can come in handy when you want to find out which source column the e-mail address came from. You can also use UNPIVOT to create more than one destination column; for example, the web form may include a field for each friend's name in addition to their e-mail address:

```
select user_account, signup_date, src_col_names, friend_email, friend_name
from email_signup
unpivot (
         (friend_email,friend_name) for src_col_names
              in ((user_email,user_name),
                  (friend1_email,friend1_name),
                  (friend2_email,friend2_name),
                  (friend3_email,friend3_name))
        )
;
```

Before Oracle Could PIVOT and UNPIVOT

Before Oracle Database 11*g*, your options were somewhat limited and painful, as the PIVOT and UNPIVOT clauses did not exist! To perform a PIVOT query, you had to use a series of DECODE statements or make a very complicated foray into the MODEL analytical clause.

Performing an UNPIVOT operation before Oracle Database 11*g* was nearly as painful. Here is a pre-Oracle Database 11*g* UNPIVOT operation on the e-mail address list processing table from the previous section:

```
select user_account, signup_date,
   'USER_EMAIL' as src_col_name, user_email as friend_email
from email_signup
where user_email is not null
union
select user_account, signup_date, 'FRIEND1_EMAIL', friend1_email
from email_signup
where friend1_email is not null
union
select user_account, signup_date, 'FRIEND2_EMAIL', friend2_email
from email_signup
where friend2_email is not null
union
select user_account, signup_date, 'FRIEND3_EMAIL', friend3_email
from email_signup
where friend3_email is not null
;
```

The maintenance cost of this query is much higher and the execution time much longer due to the sorting and merging required with the series of UNION statements and multiple passes over the same table.

4-6. Concatenating Data for Readability

Problem

For reporting and readability purposes, you want to combine multiple columns into a single output column, eliminating extra blank space and adding punctuation where necessary.

Solution

Use Oracle string concatenation functions or operators to save space in your report and make the output more readable. For example, in the EMPLOYEES table, you can use the || (two vertical bars) operator or the CONCAT function to combine the employee's first and last name:

```
select employee_id, last_name || ', ' || first_name full_name, email
from employees
;

EMPLOYEE_ID      FULL_NAME                        EMAIL
---------------- -------------------------------- ------------------
100              King, Steven                     SKING
101              Kochhar, Neena                   NKOCHHAR
102              De Haan, Lex                     LDEHAAN
103              Hunold, Alexander                AHUNOLD
104              Ernst, Bruce                     BERNST
```

```
105             Austin, David                    DAUSTIN
. . .
204             Baer, Hermann                    HBAER
205             Higgins, Shelley                 SHIGGINS
206             Gietz, William                   WGIETZ

107 rows selected
```

The query concatenates the last name, a comma, and the first name into a single string, aliased as FULL_NAME in the results. If your platform's character set does not support using || as a concatenation operator (as some IBM mainframe character sets do), or you might soon migrate your SQL to such a platform, you can make your code more platform-independent by using the CONCAT functions instead:

```
select employee_id, concat(concat(last_name,', '),first_name) full_name, email
from employees
;
```

Because the CONCAT function only supports two operands as of Oracle Database 11*g*, concatenating more than two strings can make the code unreadable very fast!

How It Works

You can apply a number of different Oracle built-in functions to make your output more readable; some Oracle shops I've worked in relied solely on SQL*Plus for their reporting. Applying the appropriate functions and using the right SQL*Plus commands makes the output extremely readable, even with queries returning Oracle objects, currency columns, and long character strings.

If your text-based columns have leading or trailing blanks (which of course should have been cleaned up on import or by the GUI form), or the column is a fixed-length CHAR column, you can get rid of leading and trailing blanks using the TRIM function, as in this example:

```
select employee_id, trim(last_name) || ', ' || trim(first_name) full_name, email
from employees
;
```

If you only want to trim leading or trailing blanks, you can use LTRIM or RTRIM respectively. If some of your employees don't have first names (or last names), your output using any of the previous solutions would look a little odd:

```
select employee_id, last_name || ', ' || first_name full_name, email
from celebrity_employees
;

EMPLOYEE_ID     FULL_NAME                        EMAIL
--------------- -------------------------------- ------------------
1001            Cher,                            CHER
1021            Kajol,                           KAJOL
1032            Madonna,                         MADONNA
2033            Bono,                            BONO
1990            Yanni,                           YANNI

5 rows selected
```

To remedy this situation, you can add the NVL2 function to your SELECT statement:

```
select employee_id,
   trim(last_name) ||
   nvl2(trim(first_name),', ','') ||
   trim(first_name) full_name,
   email
from employees
;
```

The NVL2 function evaluates the first argument TRIM(FIRST_NAME). If it is not NULL, it returns ', ', otherwise it returns a NULL (empty string), so that our celebrity employees won't have a phantom comma at the end of their name:

```
EMPLOYEE_ID      FULL_NAME                        EMAIL
---------------- -------------------------------- ------------------
1001             Cher                             CHER
1021             Kajol                            KAJOL
1032             Madonna                          MADONNA
2033             Bono                             BONO
1990             Yanni                            YANNI

5 rows selected
```

Finally, you can also take advantage of the INITCAP function to fix up character strings that might have been entered in all caps or have mixed case. If your EMPLOYEES table had some rows with last name and first name missing, you could derive most of the employee names from the e-mail address by using a combination of SUBSTR, UPPER, and INITCAP as in this example:

```
select employee_id, email,
   upper(substr(email,1,1)) || ' ' || initcap(substr(email,2)) name
from employees
;
```

```
EMPLOYEE_ID            EMAIL                     NAME
---------------------- ------------------------- --------------------------
100                    SKING                     S King
101                    NKOCHHAR                  N Kochhar
102                    LDEHAAN                   L Dehaan
103                    AHUNOLD                   A Hunold
104                    BERNST                    B Ernst
105                    DAUSTIN                   D Austin
106                    VPATABAL                  V Patabal
. . .
```

This solution is not ideal if the e-mail address does not contain the complete employee name or if the employee's last name has two parts, such as McDonald, DeVry, or DeHaan.

4-7. Translating Strings to Numeric Equivalents

Problem

In an effort to centralize your domain code management and further normalize the structure of your database tables, you want to clean up and convert some text-format business attributes to numeric equivalents. This will enhance reporting capabilities and reduce data entry errors in the future.

Solution

Use the CASE function to translate business keys or other intelligent numeric keys to numeric codes that are centrally stored in a domain code table. For example, the ORDERS table of the OE schema contains a column ORDER_MODE that currently has four possible values, identified in Table 4-1.

Table 4-1. *Mapping the Text in the ORDER_MODE Column to Numeric Values*

Text (source column)	Numeric (destination column)
Direct	1
Online	2
Walmart	3
Amazon	4

The second column of Table 4-1 contains the numeric value we want to map to for each of the possible values in the ORDER_MODE column. Here is the SQL you use to add the new column to the table:

```
alter table orders add (order_mode_num   number);
```

Oracle versions 9*i* and later include the CASE statement, which is essentially a way to more easily execute procedural code within the confines of the typically non-procedural SQL command language. The CASE statement has two forms: one for simpler scenarios with a single expression that is compared to a list of constants or expressions, and a second that supports evaluation of any combination of columns and expressions. In both forms, CASE returns a single result that is assigned to a column in the SELECT query or DML statement.

The recipe solution using the simpler form of the CASE statement is as follows:

```
update orders
set order_mode_num =
   case order_mode
      when 'direct' then 1
      when 'online' then 2
      when 'walmart' then 3
      when 'amazon' then 4
      else 0
   end
;
```

Once you run the UPDATE statement, you can drop the ORDER_MODE column after verifying that no other existing SQL references it.

How It Works

The CASE statement performs the same function as the older (but still useful in some scenarios) DECODE function, and is a bit more readable as well. For more complex comparisons, such as those evaluating more than one expression or column, you can use the second form of the CASE statement. In the second form, the CASE clause does not contain a column or expression; instead, each WHEN clause contains the desired comparison operation. Here is an example where we want to assign a special code of 5 when the order is an employee order (the CUSTOMER_ID is less than 102):

```
update orders
set order_mode_num =
   case
      when order_mode = 'direct' and
           customer_id < 102 then 5
      when order_mode = 'direct' then 1
      when order_mode = 'online' then 2
      when order_mode = 'walmart' then 3
      when order_mode = 'amazon' then 4
      else 0
   end
;
```

Note that in this scenario you need to check for the employee order first in the list of WHEN clauses, otherwise the ORDER_MODE column will be set to 1 and no customer orders will be flagged, since both conditions check for ORDER_MODE = 'direct'. For both DECODE and CASE, the evaluation and assignment stops as soon as Oracle finds the first expression that evaluates to TRUE.

In the solution, the string 'direct' is translated to 1, 'online' is translated to 2, and so forth. If the ORDER_MODE column does not contain any of the strings in the list, the ORDER_MODE_NUM column is assigned 0.

Finally, reversing the mapping in a SELECT statement for reporting purposes is very straightforward: we can use CASE or DECODE with the text and numeric values reversed. Here is an example:

```
select order_id, customer_id, order_mode_num,
   case order_mode_num
       when 1 then 'Direct, non-employee'
       when 2 then 'Online'
       when 3 then 'WalMart'
       when 4 then 'Amazon'
       when 5 then 'Direct, employee'
       else 'unknown'
   end order_mode_text
```

```
from orders
where order_id in (2458,2397,2355,2356)
;

ORDER_ID       CUSTOMER_ID     ORDER_MODE_NUM    ORDER_MODE_TEXT
-------------- --------------- ----------------- --------------------
2355           104             2                 Online
2356           105             2                 Online
2397           102             1                 Direct, non-employee
2458           101             5                 Direct, employee

4 rows selected
```

The older DECODE statement is the most basic of the Oracle functions that converts one set of values to another; you can convert numeric codes to human-readable text values or vice versa, as we do in the previous solutions. DECODE has been available in Oracle since the earliest releases. DECODE has a variable number of arguments, but the arguments can be divided into three groups:

- The column or expression to be translated
- One or more pairs of values; the first value is the existing value and the second is the translated value
- A single default value if the column or expression to be translated does not match the first value of any of the specified pairs

Here is the UPDATE statement using DECODE:

```
update orders
set order_mode_num =
   decode(order_mode,
             'direct',1,
             'online',2,
             'walmart',3,
             'amazon',4,
          0)
;

105 rows updated
```

DECODE translates the ORDER_MODE column just as CASE does. If the values in the column do not match any of the values in the first of each pair of constants, DECODE returns 0.

4-8. Generating Random Data

Problem

You need to generate random numbers to simulate real-world events that do not follow a discernible pattern.

Solution

Use the Oracle built-in PL/SQL package DBMS_RANDOM. The RANDOM function returns an integer in the range $[-2^{31},2^{31})$ (-2^{31} can be returned, but 2^{31} will not), and the VALUE function returns a decimal number in the range [0,1) with 38 digits of precision.

For example, the merchandising department wants to lower pricing below list price on catalog items on a daily basis to potentially stimulate sales from customers who perceive a bargain when the price is at least 10 percent below list price. However, the merchandising analysts want to vary the discount from 10 percent to 20 percent on a random basis. To do this, first create a DAILY_PRICE column in the PRODUCT_INFORMATION table as follows:

```
alter table product_information add (daily_price  number);
```

Next, use the DBMS_RANDOM.VALUE function to adjust the DAILY_PRICE to a value between 10 percent and 20 percent below the list price:

```
update product_information
set daily_price =
   round(list_price*(0.9-(dbms_random.value*0.1)))
;

288 rows updated.
```

Here is the query to retrieve the calculated daily price:

```
select product_id, list_price, daily_price
from product_information
;

PRODUCT_ID             LIST_PRICE             DAILY_PRICE
---------------------- ---------------------- ----------------------
1772                   456                    380
2414                   454                    379
2415                   359                    293
2395                   123                    99
1755                   121                    104
2406                   223                    200
2404                   221                    182
. . .
```

Running the UPDATE statement a second time will adjust the prices randomly again, with a different discount for each product, but still ranging from 10 to 20 percent off:

```
update product_information
set daily_price =
   round(list_price*(0.9-(dbms_random.value*0.1)))
;

select product_id, list_price, daily_price
from product_information
;
```

```
PRODUCT_ID             LIST_PRICE             DAILY_PRICE
---------------------- ---------------------- ----------------------
1772                   456                    383
2414                   454                    378
2415                   359                    319
2395                   123                    111
1755                   121                    107
2406                   223                    193
2404                   221                    184
. . .
```

The random number value returned is multiplied by 0.1 (10 percent), subtracted from 0.9 (90 percent of the list price is 10 percent off), resulting in a discount of between 10 and 20 percent. The final result is rounded to the nearest dollar.

How It Works

Most, if not all, random number generators start generating the results with a *seed* value, in other words, a value that the random number generator uses to calculate a starting place in the random number sequence. The DBMS_RANDOM package is no exception. If you do not specify a seed value, Oracle uses the current date, user ID, and process ID to calculate the seed. To reproduce a sequence of random numbers from the same starting point, you can specify your own seed, as in this example:

```
begin
   dbms_random.seed(40027);
end;
```

The DBMS_RANDOM.SEED procedure will also accept a string value to initialize the random number sequence:

```
begin
   dbms_random.seed('The Rain in Spain');
end;
```

The DBMS_RANDOM.VALUE function has even a bit more flexibility: you can specify that the random number fall within the specified range. For example, if you only want random numbers between 50 and 100, you can do this:

```
select dbms_random.value(50,100) from dual;

DBMS_RANDOM.VALUE(50,100)
-------------------------
95.2813813550075919354971754613182604395

1 rows selected
```

As a result, you can modify the recipe solution a bit as follows to take advantage of the range specification:

```
update product_information
```

```
set daily_price =
   round(list_price*(0.9-(dbms_random.value(0.0,0.1))))
;
```

Finally, the DBMS_RANDOM package also includes the STRING function so you can retrieve random string values. You specify two parameters: the type of string returned, and the length. Here is an example:

```
select dbms_random.string('U',20) from dual;

DBMS_RANDOM.STRING('U',20)
--------------------------
FTFGYZPCYWNCOKYCKDJI

1 rows selected
```

The first parameter can be one of these string constants:

- 'u' or 'U': uppercase alpha
- 'l' or 'L': lowercase alpha
- 'a' or 'A': mixed case alpha
- 'x' or 'X': upper case alphanumeric
- 'p' or 'P': any printable characters

Thus, to generate a very cryptic and random set of 25 characters, you can do this:

```
select dbms_random.string('p',25) from dual;

DBMS_RANDOM.STRING('P',25)
--------------------------
n!Wuew+R$$Qhf^mbGR,2%tr@a

1 rows selected
```

4-9. Creating a Comma-Separated Values File

Problem

You need to export an Oracle table or view to CSV (Comma-Separated Values) for import into Microsoft Excel or another application that can import data in CSV format.

Solution

Use SQL*Plus, a custom query against one or more tables, and a set of carefully selected SQL*Plus commands to create a text file that will need no further processing before import into Microsoft Excel.

The query you use to retrieve the data returns the first line with the column names, then the rows in the table with commas separating each column and double quotes around each string. Here is the query along with the required SQL*Plus commands that you can store in a file called `emp_sal.sql`, for example:

```
-- suppress sql output in results
set echo off
-- eliminate row count message at end
set feedback off
-- make line long enough to hold all row data
set linesize 1000
-- suppress headings and page breaks
set pagesize 0
-- eliminate SQL*Plus prompts from output
set sqlprompt ''
-- eliminate trailing blanks
set trimspool on
-- send output to file
spool emp_sal.csv
select '"EMPLOYEE_ID","LAST_NAME","FIRST_NAME","SALARY"' from dual
union all
select employee_id || ',"' ||
       last_name || '","' ||
       first_name || '",' ||
       salary
from employees
;
spool off
exit
```

The output from running this script looks like this:

```
C:\>sqlplus hr/hr@recipes @emp_sal.sql

SQL*Plus: Release 11.1.0.6.0 - Production on Sun Sep 27 22:12:01 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


Connected to:
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

"EMPLOYEE_ID","LAST_NAME","FIRST_NAME","SALARY"
100,"King","Steven",21266.67
101,"Kochhar","Neena",44098.56
102,"De Haan","Lex",21266.67
. . .
203,"Mavris","Susan",7150
204,"Baer","Hermann",11000
205,"Higgins","Shelley",11165
206,"Gietz","William",11165
```

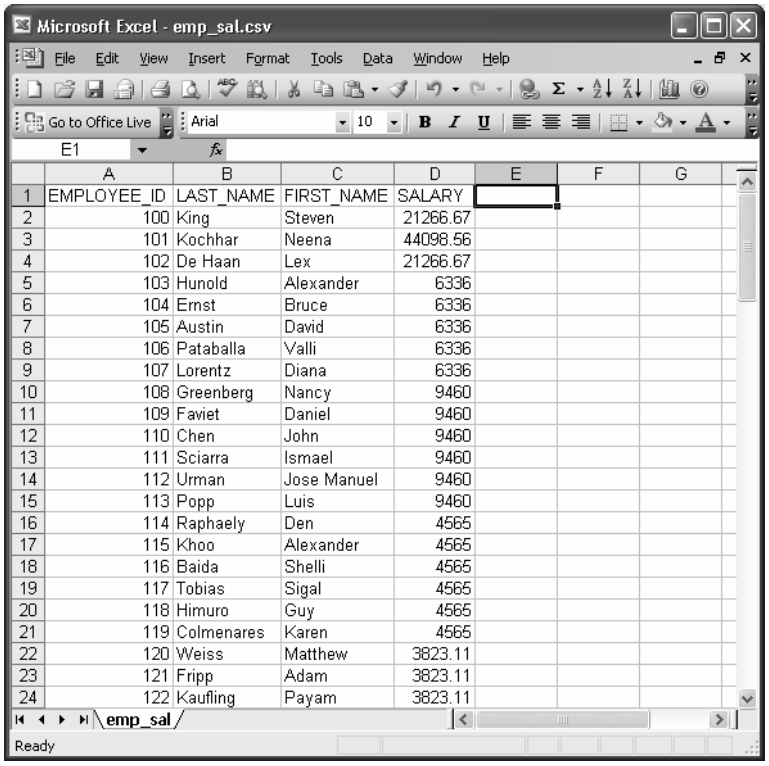
```
Disconnected from Oracle Database 11g Enterprise Edition
    Release 11.1.0.6.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

C:\>
```

How It Works

Many ETL (Extract, Transform, and Load) tools include an option to read from an Oracle database and create an output file in CSV format. If your data movement needs are rather modest and a six-figure ETL tool license is not within your budget, the SQL*Plus version may suffice. Even SQL*Developer, a free tool from Oracle, will export the results of a query or a table to a number of formats including Microsoft Excel, but SQL*Plus is more amenable to scripting and scheduling within a batch job.

The SQL*Plus commands ensure that the default column headers are off, the output width is sufficient, and so forth. The `SPOOL` and `SPOOL OFF` commands send the output to a file on disk that can be read by Microsoft Excel or another program that can read CSV files.

When you open the file `emp_sal.csv` generated in the recipe solution with Microsoft Excel, it looks much like any spreadsheet, as you can see in Figure 4-1.


Microsoft Excel - emp_sal.csv
File Edit View Insert Format Tools Data Window Help
Go to Office Live
Arial
10
E1
emp_sal
Ready

	A	B	C	D
1	EMPLOYEE_ID	LAST_NAME	FIRST_NAME	SALARY
2	100	King	Steven	21266.67
3	101	Kochhar	Neena	44098.56
4	102	De Haan	Lex	21266.67
5	103	Hunold	Alexander	6336
6	104	Ernst	Bruce	6336
7	105	Austin	David	6336
8	106	Pataballa	Valli	6336
9	107	Lorentz	Diana	6336
10	108	Greenberg	Nancy	9460
11	109	Faviet	Daniel	9460
12	110	Chen	John	9460
13	111	Sciarra	Ismael	9460
14	112	Urman	Jose Manuel	9460
15	113	Popp	Luis	9460
16	114	Raphaely	Den	4565
17	115	Khoo	Alexander	4565
18	116	Baida	Shelli	4565
19	117	Tobias	Sigal	4565
20	118	Himuro	Guy	4565
21	119	Colmenares	Karen	4565
22	120	Weiss	Matthew	3823.11
23	121	Fripp	Adam	3823.11
24	122	Kaufling	Payam	3823.11

Figure 4-1. *A SQL*Plus-generated CSV file opened in Microsoft Excel*

CHAPTER 5

■ ■ ■

Common Query Patterns

In this chapter we introduce recipes to handle many common query patterns, often for the sort of problems that you know should have an elegant solution, but for which no obvious SQL command or operator exists. Whether it's predicting the trend in data for extrapolation and planning purposes, paginating your data for display on a web site, or finding lost text in your database, you'll find a recipe to suit you in this chapter.

Many of the problems we address here have multiple solutions. Where space allows, we've tried to cover as many options as possible. Having said that, if you invent or know of a different approach, consider yourself a budding SQL chef too!

5-1. Changing Nulls into Real Values

Problem

You want to substitute null values in a table or result with a meaningful alternative.

Solution

Oracle provides several functions to allow manipulation and conversion of null values into arbitrary literals, including `NVL`, `NVL2`, `COALESCE`, and `CASE`-based tests. Each of these allows for different logic when handling null values, translating their inclusion in results in different ways. The goal is to turn `NULL` values into something more meaningful to the non-database professional, or more appropriate to the business context in which the data will be used.

Our recipe's first `SELECT` statement uses the `NVL` function to return an employee's id and surname, and either the commission percentage or the human-readable "no commission" for staff with a null `COMMISSION_PCT`.

```
select employee_id, last_name,
  nvl(to_char(commission_pct), 'No Commission') as COMMISSION
from hr.employees;
```

```
EMPLOYEE_ID                    LAST_NAME           COMMISSION
-----------                    -----------         -------------
…
        177                    Livingston              .2
        178                    Grant                   .15
        179                    Johnson                 .1
        180                    Taylor              No Commission
        181                    Fleaur              No Commission
        182                    Sullivan            No Commission
…
```

The second version of our recipe targets the same information, but uses the NVL2 function to split the employees into those that are paid a commission and those that aren't.

```
select employee_id, last_name,
  nvl2(commission_pct, 'Commission Based', 'No Commission') as COMMISSION
from hr.employees;
```

```
EMPLOYEE_ID               LAST_NAME             COMMISSION
-----------               ----------            ----------------
…
        177               Livingston            Commission Based
        178               Grant                 Commission Based
        179               Johnson               Commission Based
        180               Taylor                No Commission
        181               Fleaur                No Commission
        182               Sullivan              No Commission
…
```

Our third recipe returns the product of the COMMISSION_PCT and SALARY for commission-based employees, or the SALARY value for those with NULL COMMISSION_PCT, using the COALESCE function.

```
select employee_id, last_name,
  coalesce((1 + commission_pct) * salary, salary) as SAL_INC_COMM
from hr.employees;
```

```
EMPLOYEE_ID              LAST_NAME          COMM_OR_SAL
-----------              ---------          -----------
…
        210              King                    4375
        211              Sully                   4375
        212              McEwen                  4375
        100              King                    24000
        101              Kochhar                 17000
        102              De Haan                 17000
…
```

Our fourth recipe uses the CASE feature to return the salary of non-commissioned employees.

```
select employee_id, last_name,
case
  when commission_pct is null then salary
  else (1 + commission_pct) * salary
  end total_pay
from hr.employees;
```

The results are the same as our recipe code using the COALESCE function.

How It Works

Each of the NVL, NVL2, and COALESCE functions let you handle NULL values in different ways, depending on requirements. Our recipe using NVL selects the EMPLOYEE_ID and LAST_NAME in basic fashion, and then uses NVL to identify whether COMMISSION_PCT has a value or is NULL. Those rows with a genuine value have that value returned, and those rows with NULL return the text "No Commission" instead of nothing at all.

The general structure of the NVL function takes this basic form.

```
nvl(expression, <value to return if expression is null>)
```

By contrast, the second recipe takes the NULL handling slightly further using the NVL2 function. Instead of just substituting a placeholder value for NULL, the NVL2 function acts as a switch. In our example, NVL2 evaluates the value of COMMISSION_PCT. If a real value is detected, the second expression is returned. If a NULL is detected by NVL2, the third expression is returned.

The general form of the NVL2 function is shown next.

```
nvl(expression,
    <value to return if expression is not null>,
    <value to return if expression is null>)
```

Third, we used the COALESCE function to find the first non-NULL value among COMMISSION_PCT and SALARY for each employee. The COALESCE function can take an arbitrary number of expressions, and will return the first expression that resolves to a non-NULL value. The general form is:

```
coalesce(expression_1, expression_2, expression_3 … expression_n)
```

Lastly, we use the CASE expression to evaluate the COMMISSION_PCT field and to switch logic if the value found is NULL. The general form of the CASE expression is:

```
case
  when <expression> then <value, expression, column>
  <optional additional when clauses>
  else <some default value, expression, column>
end
```

5-2. Sorting on Null Values

Problem

Results for a business report are sorted by department manager, but you need to find a way to override the sorting of nulls so they appear where you want at the beginning or end of the report.

Solution

Oracle provides two extensions to the ORDER BY clause to enable SQL developers to treat NULL values separately from the known data, allowing any NULL entries to sort explicitly to the beginning or end of the results.

For our recipe, we'll assume that the report desired is based on the department names and manager identifiers from the HR.DEPARTMENTS table. This SQL selects this data and uses the NULLS FIRST option to explicitly control NULL handling.

```
select department_name, manager_id
from hr.departments
order by manager_id nulls first;
```

The results present the "unmanaged" departments first, followed by the departments with managers by MANAGER_ID. We've abbreviated the results to save trees.

```
DEPARTMENT_NAME                        MANAGER_ID
--------------------                   ----------
Control And Credit
Recruiting
…
Shareholder Services
Benefits
Executive                                100
IT                                       103
…
Public Relations                         204
Accounting                               205

27 rows selected.
```

How It Works

Normally, Oracle sorts NULL values to the end of the results for default ascending sorts, and to the beginning of the results for descending sorts. The NULLS FIRST ORDER BY option, together with its complement, NULLS LAST, overrides Oracle's normal sorting behavior for NULL values and places them exactly where you specify: either at the beginning or end of the results.

Your first instinct when presented with the problem of NULL values sorting to the "wrong" end of your data might be to simply switch from ascending to descending sort, or vice versa. But if you think about more complex queries, subselects using ROWNUM or ROWID tricks, and other queries that need to preserve data order while getting NULL values moved, you'll see that NULLS FIRST and NULLS LAST have real utility. Using them *guarantees* where the NULL values appear, regardless of how the data values are sorted.

5-3. Paginating Query Results

Problem

You need to display query results on web pages, showing a subset of results on each page. Users will be able to navigate back and forth through the pages of results.

Solution

Solving pagination problems requires thinking of the process in more generic terms—and using several fundamental Oracle features in combination to tackle problems like this. What we're really attempting to do is find a *defined subset* of a set of results, whether we display this subset on a web page or report, or feed it in to some subsequent process. There is no need to alter your data to add explicit page numbers or partitioning details. Our solutions will use Oracle's ROWNUM pseudo-column and ROW_NUMBER OLAP function to handle implicit page calculations, and nested subselects to control which page of data we see.

We'll use the OE.PRODUCT_INFORMATION table as the source for our solution, supposing we'll use the data therein to publish an online shopping web site. In the sample schema Oracle provides, the OE.PRODUCT_INFORMATION table holds 288 rows. That's too many for a single web page aesthetically, even though you could technically display a list that long. Your poor customers would tire of scrolling before they bought anything!

Thankfully, we can save our clients with the next SELECT statement. We'll control our SELECT statement to return 10 results only.

```
select product_id, product_name, list_price from
  (select prodinfo.*, rownum r
   from
     (select product_id, product_name, list_price
      from oe.product_information
      order by product_id) prodinfo
   where rownum <= 10)
where r >= 1;
```

For once, we won't abbreviate the results the solution so you can see we have 10 rows of results for display on our hypothetical web page.

```
PRODUCT_ID                 PRODUCT_NAME                      LIST_PRICE
----------                 --------------------              ----------
      1726                 LCD Monitor 11/PM                 259
      1729                 Chemicals - RCP                   80
      1733                 PS 220V /UK                       89
      1734                 Cable RS232 10/AM                  6
      1737                 Cable SCSI 10/FW/ADS               8
      1738                    PS 110V /US                       86
      1739                    SDRAM - 128 MB                   299
      1740                    TD 12GB/DAT                      134
      1742                    CD-ROM 500/16x                   101
      1743                    HD 18.2GB @10000 /E              800

10 rows selected.
```

An alternative technique is to use the ROW_NUMBER OLAP function to perform equivalent row numbering, and similarly filter by a predicate on the numbers it produces.

```
select product_id, product_name, list_price from
  (select product_id, product_name, list_price,
     row_number() over (order by product_id) r
   from oe.product_information)
where r between 1 and 10
```

The results are the same, including the historical artifacts like 18 gigabyte hard disk drives for $800! Those were the days.

How It Works

Let's examine the ROWNUM technique first. The core of this solution is using ROWNUM in a subselect to tag the data we really want with a handy number that controls the pagination. The ROWNUM values aren't stored anywhere: they're not a column in your table or hidden somewhere else. ROWNUM is a pseudo-column of your *results*, and it only comes into being when your results are gathered, but before any sorting or aggregation.

Looking at the subselects at the heart of our recipe, you'll see the format has this structure.

```
select <columns I actually want>, rownum r
from
  (select <columns I actually want>
  from oe.product_information
  order by product_id)
where rownum <= 10
```

We select the actual data and columns we want and wrap those in a similar SELECT that adds the ROWNUM value. We give this an alias for later use in the outer SQL statement, which we'll explain shortly. The FROM and ORDER BY clauses are self-explanatory, leaving us the ROWNUM predicate. In this case, we look for ROWNUM values less than 10, because ultimately we want to show the rows from 1 to 10. In its general form, we are really asking for all matching result rows with a ROWNUM up to the end point we want for our page. So if we wanted to show items 41 to 50 on page 5 of our hypothetical web site, this clause would read WHERE ROWNUM <= 50. In your application code or stored procedure, it's natural to replace this with a bind variable.

```
where rownum <= :page-end-row
```

This means the subselect will actually produce results for all rows up to the last row you intend to use in pagination. If we use our query to ultimately display the page for items 41 to 50 with the WHERE

ROWNUM <= 50 option, the subselect will gather the results for 50 rows, not just the 10 you intend to display. At this point, the outer query comes in to play.

The structure of the outer query has this general form:

```
select <columns I actually want> from
  (<rows upto the end-point of pagination provided by subselect>)
where r >= 1;
```

We're performing a quite normal subselect, where we take the meaningful columns from the subselect for display in the SELECT portion of the statement, and use our WHERE predicate to lop off any unnecessary leading rows from the results of the subselect, leaving us with the perfect set of data for pagination. For our recipe modification targeting rows 41 to 50, this predicate would change to WHERE R >= 41. Again, using this in a prepared fashion or through a stored procedure would typically be done with a bind variable, giving this general form of the WHERE clause.

```
where r >=  :page-start-row
```

At this point, we hope you find yourself with two lingering questions. Why did we introduce the column alias "r" for ROWNUM in the subselect rather than just using the ROWNUM feature again in the outer SELECT statement, and why didn't we just use a BETWEEN clause to simplify the whole design? The alias is used to preserve the ROWNUM values from the subselect for use in the outer select. Suppose instead we'd tried to use ROWNUM again as in the next SQL statement.

```
-- Note!  Intentionally flawed rownum logic
select product_id, product_name, list_price from
  (select prodinfo.*, rownum from
     (select product_id, product_name, list_price
     from oe.product_information
     order by product_id) prodinfo
   where rownum <= 50)
where rownum >= 41;
```

Try running that yourself and you'll be surprised to find no rows returned. Equally, if we stripped out the subselect constructs and just tried a BETWEEN clause, we'd construct SQL like the next example.

```
-- Note!  Intentionally flawed rownum logic
select product_id, product_name, list_price
from oe.product_information
where rownum between 41 and 50;
```

Oops! Once again, no rows returned. Why are these failing? It's because the ROWNUM mechanism only assigns a value after the basic (non-sorting/aggregating) criteria are satisfied, and always starts with a ROWNUM value of 1. Only after that value is assigned does it increment to 2, and so on. By removing the alias, we end up generating *new* ROWNUM values in the outer SELECT and then asking if the first candidate value, 1, is greater than or equal to 41. It's not, so that row is discarded and our ROWNUM value doesn't increment. No subsequent rows satisfy the same predicate—and you end up with no results. Using the BETWEEN variant introduces the same problem. That's why we use the alias in the first place, and why we refer to the preserved "r" values in our successful recipe.

■ **Caution** Developers often fail to test non-obvious cases when using this style of logic. There is one subset of data where our problematic examples would return results. This would be for the first page of data, where the page starting row does have a ROWNUM value of 1. This is a classic case of the boundary condition working but all other possible data ranges failing. Make sure your testing covers both the natural end points of your data, as well as ranges in between, to save yourself from such pitfalls.

Our second recipe takes a different tack. Using the ROW_NUMBER OLAP function, we achieve the same outcome in one pass over the data. Like all OLAP functions, the ROW_NUMBER function is applied to the query after the normal predicates, joins, and the like are evaluated. For the purposes of pagination, this means that the subquery effectively queries all of the data from the OE.PRODUCT_INFORMATION table, and ROW_NUMBER then applies an incrementing number by the order indicated, in this case by PRODUCT_ID.

```
select product_id, product_name, list_price,
  row_number() over (order by product_id) r
from oe.product_information
```

If we evaluated the subquery by itself, we'd see results like this (abbreviated to save paper).

```
PRODUCT_ID              PRODUCT_NAME                      LIST_PRICE          R
----------             ------------------                ----------         ---
      1726             LCD Monitor 11/PM                     259              1
      1729             Chemicals - RCP                        80              2
      1733             PS 220V /UK                            89              3
...
      3511             Paper - HQ Printer                      9            287
      3515            Lead Replacement                         2            288

288 rows selected.
```

Every row of the table is returned, as our SELECT statement hasn't qualified the data with any filtering predicates. The ROW_NUMBER function has numbered all 288 rows, from 1 to 288.

At this point, the outer query comes into play and its role is very straightforward. Our recipe uses an outer query statement that looks like the next SQL snippet.

```
select product_id, product_name, list_price from
  (<subselect returning desired columns and row number “r”>)
where r between 1 and 10
```

This outer SQL is as obvious as it looks. Select the desired columns returned from the subselect, where the r value (generated by the subselect's ROW_NUMBER function) is between 1 and 10. In comparison to our other solution that used ROWNUM, if we wanted to represent a different page of results for products 41 to 50, the outer query's WHERE clause would simply change to WHERE R BETWEEN 41 AND 50. In the general form, we'd recommend parameterizing the page-start-row and page-end-row, as illustrated in the following SQL pseudo code.

```
select <desired columns>
```

```
  (select <desired columns>,
     row_number() over (order by <ordering column>) r
   from <source table, view, etc.>)
where r between :page-start-row and :page-end-row
```

By now you're probably asking yourself which of the two methods, ROWNUM and ROW_NUMBER, you should use. There's no clear-cut answer to that question. Instead, it's best to remember some of the qualities of both recipes. The ROWNUM approach enjoys historical support and has built-in optimization in Oracle to make the sort induced by the ordering faster than a normal sort. The ROW_NUMBER approach is more versatile, enabling you to number your rows in an order different from the order produced by a standard ORDER BY clause. You can also alter the recipe to use other OLAP functions like RANK and DENSE_RANK to handle different paging requirements, such as needing to show items in "tied" positions.

Pagination Outside the Database

Our recipe drives the pagination of data where it is generally best handled—in the database! You may have used or experienced other techniques, such as cursor-driven pagination and even result-caching at the application tier, with data discarded or hidden to provide pagination.

Our advice is to eschew those techniques in pretty much all cases. The application caching technique inevitably requires excessive network traffic with associated performance delay and cost, which the end user won't appreciate. Cursor-driven approaches do work at the database tier to minimize the network issue, but you are always susceptible to "stale" pagination with cursor techniques. Don't be in any doubt: use the database to paginate for you—and reap the benefits!

5-4. Testing for the Existence of Data

Problem

You would like to compare the data in two related tables, to show where matching data exists, and to also show where matching data doesn't exist.

Solution

Oracle supports the EXISTS and NOT EXISTS predicates, allowing you to correlate the data in one table or expression with matching or missing data in another table or expression. We'll use the hypothetical situation of needing to find which departments currently have managers. Phrasing this in a way that best illustrates the EXISTS solution, the next SQL statement finds all departments where a manager is known to exist.

```
select department_name
from hr.departments d
where exists
 (select e.employee_id
  from hr.employees e
  where d.manager_id = e.employee_id);
```

The complement, testing for non-existence, is shown in the next statement. We ask to find all departments in HR.DEPARTMENTS with a manager that does not exist in the data held in HR.EMPLOYEES.

```
select department_name
from hr.departments d
where not exists
 (select e.employee_id
  from hr.employees e
  where d.manager_id = e.employee_id);
```

How It Works

In any database, including Oracle, the EXISTS predicate answers the question, “Is there a relationship between two data items, and by extension, what items in one set are related to items in a second set?” The NOT EXISTS variant tests the converse, “Can it definitively be said that no relationship exists between two sets of data, based on a proposed criterion?” Each approach is referred to as correlation or a correlated subquery (literally, co-relation, sharing a relationship).

Interestingly, Oracle bases its decision on whether satisfying data exists solely on this premise: was a matching row found that satisfied the subquery’s predicates? It’s almost too subtle, so we’ll point out the obvious thing Oracle isn’t seeking. What you select in the inner correlated query doesn’t matter—it’s only the criteria that matter. So you’ll often see versions of existence tests that form their subselect by selecting the value 1, the entire row using an asterisk, a literal value, or even NULL. Ultimately, it’s immaterial in this form of the recipe. The key point is the correlation expression. In our case, it’s WHERE D.MANAGER_ID = E.EMPLOYEE_ID.

This also helps explain what Oracle is doing in the second half of the recipe, where we’re looking for DEPARTMENT_NAME values for rows where the MANAGER_ID doesn’t exist in the HR.EMPLOYEES table. Oracle drives the query by evaluating, for each row in the outer query, whether no rows are returned by the inner correlated query. Oracle doesn’t care what data exists in other columns not in the correlation criteria. It pays to be careful using such NOT EXISTS clauses on their own—not because the logic won’t work but because against large data sets, the optimizer can decided to repeatedly scan the inner data in full, which might affect performance. In our example, so long as a manager’s ID listed for a department is not found in the HR.EMPLOYEES table, the NOT EXISTS predicate will be satisfied, and department included in the results.

■ **Caution** Correlated subqueries satisfy an important problem-solving niche, but it’s crucial to remember the nature of NULL values when using either EXISTS or NOT EXISTS. NULL values are not equivalent to any other value, including other NULL values. This means that a NULL in the outer part of a correlated query will never satisfy the correlation criterion for the inner table, view, or expression. In practice, this means you’ll see precisely the opposite effect as the one you might expect because the EXISTS test will always return false, even if both the inner and outer data sources have NULL values for the correlation, and NOT EXISTS will always return true. Not what the lay person would expect.

5-5. Conditional Branching In One SQL Statement

Problem

In order to produce a concise result in one query, you need to change the column returned on a row-by-row basis, conditional on a value from another row. You want to avoid awkward mixes of unions, subqueries, aggregation, and other inelegant techniques.

Solution

For circumstances where you need to conditionally branch or alternate between source data, Oracle provides the CASE statement. CASE mimics the traditional switch or case statement found in many programming languages like C or Java.

To bring focus to our example, we'll assume our problem is far more tangible and straightforward. We want to find the date employees in the shipping department (with the DEPARTMENT_ID of 50) started their current job. We know their initial hire date with the firm is tracked in the HIRE_DATE column on the HR.EMPLOYEES table, but if they've had a promotion or changed roles, the date when they commenced their new position can be inferred from the END_DATE of their previous position in the HR.JOB_HISTORY table. We need to branch between HIRE_DATE or END_DATE for each employee of the shipping department accordingly, as shown in the next SQL statement.

```
select e.employee_id,
  case
    when old.job_id is null then e.hire_date
    else old.end_date end
  job_start_date
from hr.employees e left outer join hr.job_history old
  on e.employee_id = old.employee_id
where e.department_id = 50
order by e.employee_id;
```

Our results are very straightforward, hiding the complexity that went into picking the correct JOB_START_DATE.

```
EMPLOYEE_ID             JOB_START
-----------             ---------
        120             18-JUL-96
        121             10-APR-97
        122             31-DEC-99
        123             10-OCT-97
        124             16-NOV-99
…
```

How It Works

Our recipe uses the CASE feature, in Search form rather than Simple form, to switch between HIRE_DATE and END_DATE values from the joined tables. In some respects, it's easiest to think of this CASE operation as a combination of two SELECT statements in one. For employees with no promotions, it's as if we were selecting as follows:

```
select e.employee_id, e.hire_date…
```

Whereas for employees that have had promotions, the CASE statement switches the SELECT to the following form:

```
select e.employee_id, old.end_date…
```

The beauty is in not having to explicitly code these statements yourself, and for far more complex uses of CASE, not having to code many dozens or hundreds of statement combinations.

To explore the solution from the data's perspective, the following SQL statement extracts the employee identifier and the hire and end dates using the same left outer join as our recipe.

```
select e.employee_id, e.hire_date, old.end_date end
from hr.employees e left outer join hr.job_history old
  on e.employee_id = old.employee_id
where e.department_id = 50
order by e.employee_id;

EMPLOYEE_ID                HIRE_DATE             END_DATE
-----------                ---------             ---------
        120                18-JUL-96
        121                10-APR-97
        122                01-MAY-95             31-DEC-99
        123                10-OCT-97
        124                16-NOV-99
…
```

The results show the data that drove the CASE function's decision in our recipe. The values in bold were the results returned by our recipe. For the first, second, fourth, and fifth rows shown, END_DATE from the HR.JOB_HISTORY table is NULL, so the CASE operation returned the HIRE_DATE. For the third row, with EMPLOYEE_ID 122, END_DATE has a date value, and thus was returned in preference to HIRE_DATE when examined by our original recipe. There is a shorthand form of the CASE statement known as the Simple CASE that only operates against one column or expression and has THEN clauses for possible values. This wouldn't have suited us as Oracle limits the use of NULL with the Simple CASE in awkward ways.

5-6. Conditional Sorting and Sorting By Function

Problem

While querying some data, you need to sort by an optional value, and where that value is not present, you'd like to change the sorting condition to another column.

Solution

Oracle supports the use of almost all of its expressions and functions in the ORDER BY clause. This includes the ability to use the CASE statement and simple and complex functions like arithmetic operators to dynamically control ordering. For our recipe, we'll tackle a situation where we want to show employees ordered by highest-paid to lowest-paid.

For those with a commission, we want to assume the commission is earned but don't want to actually calculate and show this value; we simply want to order on the implied result. The following SQL leverages the CASE statement in the ORDER BY clause to conditionally branch sorting logic for those with and without a COMMISSION_PCT value.

```
select employee_id, last_name, salary, commission_pct
from hr.employees
order by
  case
    when commission_pct is null then salary
    else salary * (1+commission_pct)
  end desc;
```

We can see from just the first few rows of results how the conditional branching while sorting has worked.

```
EMPLOYEE_ID              LAST_NAME              SALARY        COMMISSION_PCT
-----------              ---------              ------         --------------
        100               King                  24000
        145               Russell               14000             .4
        146               Partners              13500             .3
        101               Kochhar               17000
        102               De Haan               7000
…
```

Even though employees 101 and 102 have a higher base salary, the ORDER BY clause using CASE has correctly positioned employees 145 and 146 based on their included commission percentage.

How It Works

The selection of data for our results follows Oracle's normal approach, so employee identifiers, last names, and so forth are fetched from the HR.EMPLOYEES table. For the purposes of ordering the data using our CASE expression, Oracle performs additional calculations that aren't shown in the results. All the candidate result rows that have a non-NULL commission value have the product of COMMISSION_PCT and SALARY calculated and then used to compare with the SALARY figures for all other employees for ordering purposes.

The next SELECT statement helps you visualize the data Oracle is deriving for the ordering calculation.

```
select employee_id, last_name, commission_pct, salary,
  salary * (1+commission_pct) sal_x_comm
from hr.employees;
```

```
EMPLOYEE_ID             LAST_NAME        COMMISSION_PCT            SALARY              SAL_X_COMM
-----------             ---------        --------------            ------              ----------
        100              King                                      24000
        145              Russell               .4                  14000                 19600
        146              Partners              .3                  13500                 17550
        101              Kochhar                                   17000
        102              De Haan                                   17000
…
```

The values in bold show the calculations Oracle used for ordering when evaluating the data via the CASE expression in the ORDER BY clause. The general form of the CASE expression can be expressed simply as follows.

```
case
  when <expression, column, etc.> then <expression, column, literal, etc.>
  when <expression, column, etc.> then <expression, column, literal, etc.>
  …
  else <default expression for unmatched cases>
end
```

We won't needlessly repeat what the Oracle manual covers in plenty of detail. In short, the CASE expression evaluates the first WHEN clause for a match and if satisfied, performs the THEN expression. If the first WHEN clause isn't satisfied, it tries the second WHEN clause, and so on. If no matches are found, the ELSE default expression is evaluated.

5-7. Overcoming Issues and Errors when Subselects Return Unexpected Multiple Values

Problem

In working with data from a subselect, you need to deal with ambiguous situations where in some cases the subselect will return a single (scalar) value, and in other cases multiple values.

Solution

Oracle supports three expressions that allow a subselect to be compared based on a single column of results. The operators ANY, SOME, and ALL allow one or more single-column values from a subselect to be compared to data in an outer SELECT. Using these operators allows you to deal with situations where you'd like to code your SQL to handle comparisons with flexible set sizes.

Our recipe focuses on using these expressions for a concrete business problem. The order-entry system tracks product information in the OE.PRODUCT_INFORMATION table, including the LIST_PRICE value. However, we know discounts are often offered, so we'd like to get an approximate idea of which items have never sold at full price. To do this, we could do a precise correlated subquery of every sale against list price. Before doing that, a very quick approximation can be done to see if any LIST_PRICE value is higher than any known sale price for any item, indicated by the UNIT_PRICE column of the OE.ORDER_ITEMS table. Our SELECT statement takes this form.

```
select product_id, product_name
from oe.product_information
where list_price > ALL
  (select unit_price
   from oe.order_items);
```

From this query, we see three results:

```
PRODUCT_ID              PRODUCT_NAME
----------              ------------------------
      2351              Desk - W/48/R
      3003              Laptop 128/12/56/v90/110
      2779              Desk - OS/O/F
```

These results mean at least three items—two desks and a laptop—have never sold a full price.

How It Works

Our example's use of the ALL expression allows Oracle to compare the UNIT_PRICE values from every sale, to see if any known sale price was greater than the LIST_PRICE for an item. Breaking down the steps that make this approach useful, we can first look at the subselect.

```
select unit_price
from oe.order_items
```

This is a very simple statement that returns a single-column result set of zero or more items, shown next in abbreviated form.

```
UNIT_PRICE
----------
        13
        38
        43
        43
     482.9
…
665 rows selected.
```

Based on those results, our outer SELECT statement compares each LIST_PRICE from the OE.PRODUCTION_INFORMATION table with every item in the list, as we are using the ALL expression. If the LIST_PRICE is greater than all of the returned values, the expression resolves to true, and the product is included in the results. Where even one of the UNIT_PRICE values returned exceeds the LIST_PRICE of an item, the expression is false and that row is discarded from further consideration.

If you review that logic, you'll realize this is not a precise calculation of every item that didn't sell for full price. Rather, it's just a quick approximation, though one that shows off the ALL technique quite well.

The alternatives SOME and ANY, which are effectively synonyms, resolve the true/false determination based on only needing one item in the subselect to satisfy the SOME/ANY condition. Oracle will happily accept more than one item matching the SOME/ANY criteria, but only needs to determine one value to evaluate the subselect.

5-8. Converting Numbers Between Different Bases

Problem

You need to find a generic way to turn decimal representations of numbers such as IP addresses and MAC addresses into hexadecimal, octal, binary, and other unusual number bases.

Solution

Oracle ships with a small number of base-conversion features, the most notable of which is the TO_CHAR function's ability to convert from decimal to hexadecimal. We can convert an arbitrary value, such as 19452, to hexadecimal using the next SELECT statement:

```
select to_char(19452,'xxxxx')-oi-0ij
from dual;
```

Our output shows the correctly calculated value of 4BFC. Run that query yourself to confirm Oracle's accuracy at hexadecimal conversion. While this is useful for this specific case, often we'll want to see decimals converted to binary, octal, and other uncommon bases.

■ **Tip** You may ask yourself, what other bases might possibly be used? One of the authors once worked on software that stored all numbers in base 36, using the digits 0 to 9 and the letters A to Z to deal with limitations in certain hardware products that could only store strings. You may find yourself needing number conversion to unusual bases when you least expect it!

Other base conversions are not natively addressed out of the box by Oracle, so we'll create our own generic decimal-conversion function that can take a given number and convert it to any base from 2 to 36! We'll build our REBASE_NUMBER function using the fundamental arithmetic operations that do ship with Oracle, as shown here.

```
create or replace function rebase_number
  (starting_value in integer, new_base in number)
return varchar2
is
  rebased_value varchar2(4000) default NULL;
  working_remainder integer default starting_value;
  char_string varchar2(36) default '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  sign varchar2(1) default '';
begin
```

```
  if (starting_value < 0) then
    sign := '-';
    working_remainder := abs(working_remainder);
  end if;
  loop
    rebased_value := substr(char_string, mod(working_remainder,new_base)+1, 1)
      || rebased_value;
    working_remainder := trunc(working_remainder/new_base);
    exit when (working_remainder = 0);
  end loop;
  rebased_value := sign || rebased_value;
  return rebased_value;
end rebase_number;
/
```

With the REBASE_NUMBER function now available, we can perform conversions of our original test number, 19452, to a variety of bases. The first example shows conversion to hexadecimal, to prove we're getting the same result as Oracle.

```
select rebase_number(19452,16) as DEC_TO_HEX
from dual;
```

We successfully calculate the correct result.

```
DEC_TO_HEX
----------
4BFC
```

The same number converted to octal also succeeds.

```
select rebase_number(19452,8) as DEC_TO_OCTAL
from dual;

DEC_TO_OCTAL
------------
45774
```

A final example converting to binary similarly produces the correct result.

```
select rebase_number(19452,2) as DEC_TO_BINARY
from dual;

DEC_TO_BINARY
---------------
100101111111100
```

How It Works

Our function models the classic discrete mathematics used to perform a conversion between two bases for any number. The basic algorithm takes this form: For each significant digit in the original number (thousands, hundreds, and so forth)

- Perform modulo division on that part of the number using the new base.
- Use the whole part of the result as the new "digit" in the converted number.
- Take the remainder, and repeat until the remainder is less than the new base.

Return the new digits and the last remainder as the converted number.

There are two not-so-obvious aspects of our implementation of this logic that warrant further explanation. The first involves dealing with negative numbers. Our function sets up various local variables, some of which we'll discuss shortly, the last of which is called SIGN. We set this to an empty string, which implicitly means a positive outcome reported to the caller unless we detect that a negative number is passed to the function for conversion. In the first part of the logic body of our function, we test for the sign of the STARTING_VALUE and change the SIGN value to '-' for any negative values detected.

```
if (starting_value < 0) then
  sign := '-';
  working_remainder := abs(working_remainder);
end if;
```

Detecting negative numbers and switching the WORKING_NUMBER to positive using the ABS function isn't part of normal decimal-to-hexadecimal conversion. In fact, we don't use it in our arithmetic either! It exists purely to help with the second non-obvious aspect of our implementation, a trick we utilize in our function that we'll cover in a moment. At the end of our function, we reinstate the SIGN as it should be, so that the user doesn't notice this data massaging happening.

```
rebased_value := sign || rebased_value;
```

So why the two-step with the number's sign in the first place? It's because we use a trick to "look up" the conversion of a decimal digit (or digits) into its new base by finding how the whole part of our modulo division maps to a position in a string. In this case, the string is the unusual local variable you see at the start of the function, CHAR_STRING.

```
char_string varchar2(36) default '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
```

You may have thought, why are the authors practicing their typing in this book? But we're not! The trick is to use the SUBSTR function to walk this string to find the matching value for our conversion. Here's how it works at its basic level. We take a source number, like 13, and the desired base, like 16, and plug it into our SUBSTR call, as shown in the next fragment.

```
substr(char_string, mod(13 /* our remainder */, 16 /* our new base */)+1, 1)
```

Performing the arithmetic is easy. 13 modulo 16 gives 0 with a remainder of 13. We add one to this value, because our string is zero-based, so we need to account for the "zeroth" position, giving us a value of 14. We then take the substring of the CHAR_STRING starting at this position (14), for 1 character. The 14th character in the string is D. Voilà: we have converted the decimal number 13 to its hexadecimal equivalent, D. From there, it's just a case of repeating the process for successive modulo division results.

The reverse logic can also be incorporated into a function, so values in any base can be converted to decimal. We can also wrap these functions in a helper function that would allow the conversion from any arbitrary base to any other arbitrary base. These functions are available at www.oraclesqlrecipes.com.

5-9. Searching for a String Without Knowing the Column or Table

Problem

You need to find where in the database a particular text value is stored, but don't have access to the application code or logic that would let you determine this from seeing the data via the application. You have only the Oracle catalog tables and SQL to find what you seek.

Solution

No database currently available includes an omniscient "search everywhere for what I'm seeking" operator. However, using the basic building blocks of Oracle's string-searching LIKE operator, and the ability to output one SELECT statement as the result of another, we can execute a series of steps that will automatically track down the schema, table, and column holding the text data we seek.

To give our recipe a concrete flavor, we'll assume that we're looking for the text "Greene" somewhere in the database, but we don't know—and can't find any documentation that would tell us—if this text is a person's name, a supplier or client company name, part of a product name, or part of some other text.

Our solution works in two parts. The first SELECT statement, shown next, produces as results individual SELECT queries to search all columns of all tables in all schemata for our unidentified text, Greene.

```
Select
  'select ''' || owner || ''',
  ''' || table_name || ''',
  ''' || column_name || ''',
  ' || column_name ||
  ' from ' ||
  owner ||
  '.' ||
  table_name ||
  ' where ' ||
  column_name || '
  like ''%Greene%'';' as Child_Select_Statements
from all_tab_columns
where owner in ('BI', 'HR', 'IX', 'OE', 'PM', 'SH')
and data_type in ('VARCHAR2','CHAR','NVARCHAR2','NCHAR')
and data_length >= length('Greene');
```

In our example, the output of this command is a list of 271 SELECT statements, one for each text-like column in each of the tables in Oracle's sample schemata. A subset is shown here, with formatting to make the output readable on this printed page.

```
CHILD_SELECT_STATEMENTS
--------------------------------------------------------------------------------
…
select 'HR','COUNTRIES','COUNTRY_NAME', COUNTRY_NAME
  from HR.COUNTRIES
  where COUNTRY_NAME like '%Greene%';
select 'HR','DEPARTMENTS','DEPARTMENT_NAME', DEPARTMENT_NAME
  from HR.DEPARTMENTS
  where DEPARTMENT_NAME like '%Greene%';
select 'HR','EMPLOYEES','FIRST_NAME', FIRST_NAME
  from HR.EMPLOYEES
 where FIRST_NAME like '%Greene%';
select 'HR','EMPLOYEES','LAST_NAME', LAST_NAME
  from HR.EMPLOYEES
  where LAST_NAME like '%Greene%';
select 'HR','EMPLOYEES','EMAIL', EMAIL
  from HR.EMPLOYEES
  where EMAIL like '%Greene%';
…
```

As you're looking at these output statements, first stop and think if there are performance implications to running statements across many (if not all) of your tables. If your database is multi-terabyte in size, it might be worth some planning to execute these when you won't affect performance. Once you're happy with the logistics of when it's best to use them, run them against the database. They will elicit the schema, table, and column name, plus the column data, where the sought-after text resides. A portion of the results are shown next.

```
select 'HR','EMPLOYEES','FIRST_NAME', FIRST_NAME
  from HR.EMPLOYEES
  where FIRST_NAME like '%Greene%';

no rows selected

select 'HR','EMPLOYEES','LAST_NAME', LAST_NAME
  from HR.EMPLOYEES
  where LAST_NAME like '%Greene%';

'H 'EMPLOYEE 'LAST_NAM LAST_NAME
-- --------- --------- -------------------------
HR EMPLOYEES LAST_NAME Greene
```

```
select 'HR','EMPLOYEES','EMAIL', EMAIL
  from HR.EMPLOYEES
  where EMAIL like '%Greene%';

no rows selected
```

With the completion of the second set of statements, we've found the text "Greene" in the `LAST_NAME` column of the `HR.EMPLOYEES` table.

How It Works

This solution takes a two-pass approach to the problem. When you run the first query, it searches for all the columns in the `ALL_TAB_COLUMNS` system view to find the schema names, table names, and column names for columns that have a textual data type, such as `VARCHAR2` or `CHAR`. It's a little hard to see the logic through all the literal formatting and string concatenation, so it's best thought of using this general structure.

```
select
  <escaped column, table and schema names for later presentation>,
  <actual column name for later querying>,
  <escaped from clause for later querying >
  <escaped where clause for later querying >
from all_tab_columns
where owner in (<list of schemata in which we're interested>)
and data_type in ('VARCHAR2','CHAR','NVARCHAR2','NCHAR')
and data_length >= <length of text sought>;
```

For each table with at least one relevant textual data type, this query will emit a result that takes the form of a `SELECT` statement that generally looks like this:

```
select
  <literal schema name>,
  <literal table name>,
  <literal column name>,
  <column_name>
where <column_name> like '%<text sought>%';
```

It's then just a matter of running those `SELECT` statements and noting which ones actually produce results. All of the elaborate literal escaping and quoting is used to preserve the object names right down to this level, so the results include not only the text you seek in context, but also the human-readable data for schema, table, and column, as you can see in this example row.

```
HR EMPLOYEES LAST_NAME Greene
```

At least one instance of the text "Greene" can be found in `HR.EMPLOYEES`, in the `LAST_NAME` column.

ALL_TAB_COLUMNS versus ALL_TAB_COLS

Careful observers will note that we've crafted our solution to work with the ALL_TAB_COLUMNS system view. Oracle also includes a system view named ALL_TAB_COLS. The two seem almost identical, having the same fields, but slightly different row counts. So why choose ALL_TAB_COLUMNS in this recipe?

The answer has to do with the slightly different definitions Oracle uses in defining the two views. The ALL_TAB_COLS system view includes hidden columns not normally visible to users or developers. As our recipe seeks ordinary data used every day by end-users, we rightly assume that the developers have not played tricks by either hiding columns of their own or abusing Oracle's system-controlled hidden columns.

The recipe so far has not made any assumptions about the tool or tools you'll use when working with the solution. You can happily run the first statement in SQL*Plus, SQL Developer, or through a programmable API, and retrieve the second set of statements to run. Similarly, you can use any tool to then execute those statements and view the results revealing the hiding place of your lost text. But some extra niceties are possible. For instance, you could wrap much of the logic in a PL/SQL function or stored procedure, or use the formatting capabilities of a query tool like SQL*Plus to make the solution even more elegant.

5-10. Predicting Data Values and Trends Beyond a Series End

Problem

From a large set of data, you need to predict the behavior or trend of the information beyond the bounds of your current data set.

Solution

Predicting trends and extrapolating possibilities based on patterns in the data, such as relationships between two values, can be addressed in many ways. One of the most popular techniques is linear regression analysis, and Oracle provides numerous linear regression functions for many contemporary trending and analysis algorithms.

As part of the sample schemata included with the Oracle database, a Sales History data warehouse example is provided in the SH schema. This includes a fact table of around a million entries of individual items sold with dimensions for time, sales channel, and so on, and the item price for a given sale.

Our recipe supposes that we're solving the problem of predicting what would happen if we introduced more expensive items than those currently sold. In effect, we'd like to extrapolate beyond the current most expensive item, based on the current sales trend of volume sold compared with item price. We're answering the fundamental question, would we actually sell an item if it were more expensive, and if so, how many of those items would we sell?

The following SELECT statement uses three Oracle linear regression functions to help guide our extrapolation. We're asking what's likely to happen if we start selling items in the Electronics category at higher prices, will we sell more or fewer, and how quickly will sales volume change with price.

```
select
  s.channel_desc,
  regr_intercept(s.total_sold, p.prod_list_price)  total_sold_intercept,
  regr_slope (s.total_sold, p.prod_list_price)  trend_slope,
  regr_r2(s.total_sold, p.prod_list_price)  r_squared_confidence
from sh.products p,
  (select c.channel_desc, s.prod_id, s.time_id, sum(s.quantity_sold) total_sold
   from sh.sales s inner join sh.channels c
     on s.channel_id = c.channel_id
   group by c.channel_desc, s.prod_id, s.time_id) s
where s.prod_id=p.prod_id
and p.prod_category='Electronics'
and s.time_id between to_date('01-JAN-1998') and to_date('31-DEC-1999')
group by s.channel_desc
order by 1;

CHANNEL_DESC        TOTAL_SOLD_INTERCEPT        TREND_SLOPE        R_SQUARED_CONFIDENCE
------------        -------------------         -----------        --------------------
Direct Sales              24.2609713            -.02001389           .087934647
Internet                   6.30196312           -.00513654           .065673194
Partners                  11.738347             -.00936476           .046714001
Tele Sales                36.1015696            -.11700913           .595086378
```

For those unfamiliar with interpreting linear regressions, the extrapolated trends can be read in the general form:

```
Y = Intercept + Slope(X)
```

For our data, Y is the total number of items sold and X is the list price. We can see that for Direct Sales, Internet, and Partners, there's a gentle decrease in sales as price increases, whereas sales volume plummets dramatically for Tele Sales as cost increases. However, our confidence values suggest the first three predictions are a poorly fit extrapolation for these channels. The R-squared confidence value indicates how well the extrapolated line of fit suits the data (also known as goodness of fit), where a value of 1.0 means "perfect fit", and 0 means "absolutely no correlation".

How It Works

To feed data to our linear regression functions, such as `REGR_SLOPE`, we need to ensure we are providing the actual values we want to compare. Our recipe compares the total number of items sold to the list price of an item. The `SH.SALES` table tracks the sale of each individual item, with one entry per item sold. It's for this reason that we use the inline view to aggregate all of these individual sale entries into aggregate number of sales for a certain product, date, and channel. You can run the subselect by itself, as shown in the next SQL statement.

```
select c.channel_desc, s.prod_id, s.time_id, sum(s.quantity_sold) total_sold
from sh.sales s inner join sh.channels c
  on s.channel_id = c.channel_id
group by c.channel_desc, s.prod_id, s.time_id;
```

No surprises there, and the results provide a simple rolled-up summary ready for use in our linear regression functions.

```
CHANNEL_DESC                  PROD_ID          TIME_ID          TOTAL_SOLD
--------------------          ----------       ---------        ----------
…
Direct Sales                  30                24-OCT-01          1
Internet                      32                30-NOV-01          4
Direct Sales                  35                28-OCT-01         12
Partners                      47                09-NOV-01          1
Direct Sales                  14                06-OCT-01          5
…
```

Our results provide this summary for every type of item we sell. We join the inline view to the SH.PRODUCTS table, and then we filter by PROD_CATEGORY of Electronics and a two-year date range to focus on our supposed problem. The fun then starts with the linear regression functions in the SELECT clause.

Each of the three statistical functions takes the calculated TOTAL_SOLD amounts and LIST_PRICE values for each channel (thanks to the GROUP BY clause), and performs the relevant calculations. A good statistics textbook will tell you how these formulae were derived, but Table 5-1 shows you each formula's method.

Table 5-1. *Oracle Statistical Methods Supporting Linear Regression Analysis*

Function	Formula
REGR_INTERCEPT	$(\Sigma y)/n - (\Sigma xy - (\Sigma x)(\Sigma y)/n)/(\Sigma x^2 - (\Sigma x)^2/n)\ (\Sigma x)/n$
REGR_SLOPE	$(\Sigma xy - (\Sigma x)(\Sigma y)/n)/(\Sigma x^2 - (\Sigma x)^2/n)$
REGR_R2	$(\Sigma xy - (\Sigma x)(\Sigma y)/n)^2/(\Sigma x^2 - (\Sigma x)^2/n)(\Sigma y^2 - (\Sigma y)^2/n)$

Just looking at those formulae should make you glad you don't have to code the calculations yourself. Just call the appropriate function and Oracle does the hard work for you. Armed with the data returned by those functions, you can then visualize how the sales volume changes with price. Figure 5-1 shows our linear regression data expressed as hypothetical extrapolated lines showing the relationship between sales volume and list price.

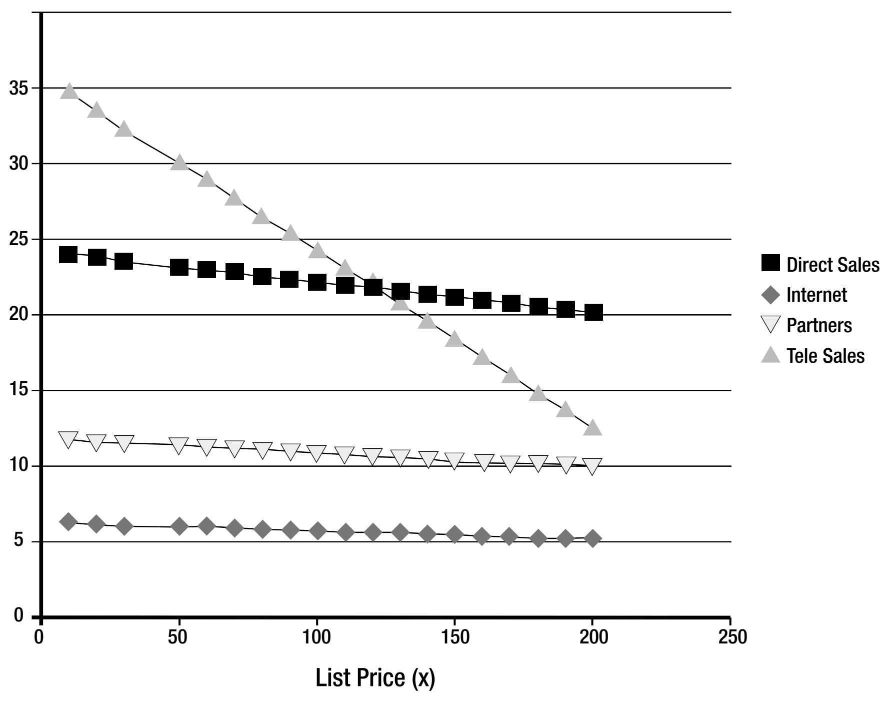


Figure 5-1. *An extrapolation of sales volume and list price using linear regression*

This gives you a clear indication of how your sales volume might change as you alter price. Missing from this is an indication of goodness of fit: literally, how reliable is the estimated relationship between the data and the extrapolation taken from it. This is the r-squared value calculated using the `REGR_R2` function.

5-11. Explicitly (Pessimistically) Locking Rows for an Update

Problem

To protect a particularly sensitive and special business case, you have been asked to ensure that updates to employee salaries are protected from lost updates, and also asked not to use an optimistic locking approach.

Solution

Our problem models one of the classic issues faced by developers and DBAs. While Oracle provides excellent concurrency control and locking mechanisms and supports both optimistic and pessimistic design approaches, often a political or business decision forces you to use one technique.

At some stage in your work with Oracle, you may find your most coherent arguments about Oracle's excellent concurrency technology brushed aside by a manager who says "That's all well and good, but I want to grant pay rises once, and *I don't want to see* any messages indicating another manager updated the same employee's salary and I should try again."

Faced with such a career-threatening imperative, you'll be glad to know Oracle provides the `SELECT … FOR UPDATE` option to explicitly lock data and provide pessimistic concurrency control. For our recipe, we'll use the next `SELECT` statement to find the salary data for employee 200, Jennifer Whalen, in preparation for a later update.

```
select employee_id, last_name, salary
from hr.employees
where employee_id = 200
for update;

EMPLOYEE_ID               LAST_NAME           SALARY
-----------               ---------           ------
      200                 Whalen              4400
```

So far, so good. If a second user (or the same user via a different connection) attempts to perform a `SELECT`, `UPDATE`, or `DELETE` of this row, that user will be blocked. Try issuing the same `SELECT` statement from another connection and you'll see no results as the session waits for the locks related to the `SELECT … FOR UPDATE` pessimistic locking to be released.

```
select employee_id, last_name, salary
from hr.employees
where employee_id = 200
for update;

(no results yet … waiting on blocking exclusive lock)
```

From the first session, we complete our update.

```
update hr.employees
set salary = salary * 1.1
where employee_id = 200;

commit;
```

Upon commit from the first session, the second session's `SELECT` statement finally runs, returning these results.

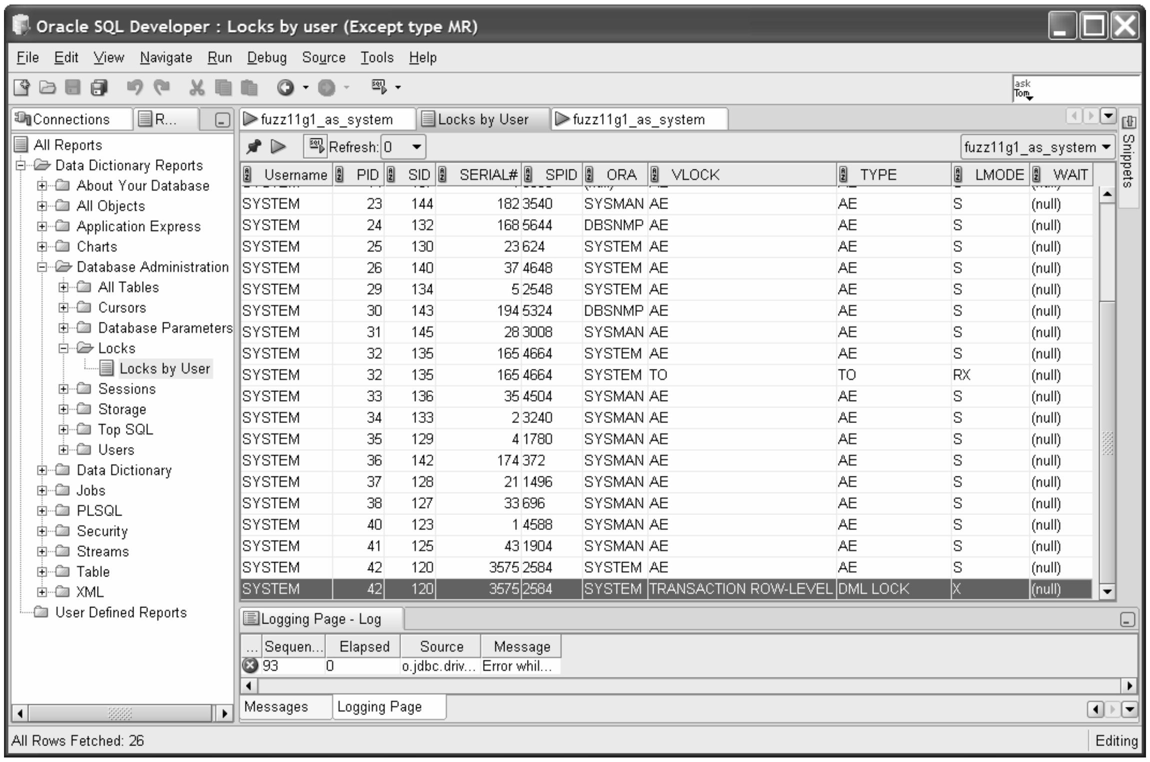
```
EMPLOYEE_ID                LAST_NAME            SALARY
-----------                ---------            ------
        200                Whalen               4840
```

Note the second session did not interrupt the update, and never saw the pre-updated data.

How It Works

The key to the pessimistic locking effect of the `FOR UPDATE` clause is in the way it forces Oracle to take locks on the data covered by the `SELECT` query. Oracle attempts to take a mode X (exclusive) transaction row-level lock on each row that satisfies a `SELECT … FOR UPDATE` query.

When the first connection issues the `SELECT … FOR UPDATE` statement, we can query the `V$LOCK` dynamic view or use a graphical tool like SQL Developer to see the lock taken, as in Figure 5-2.



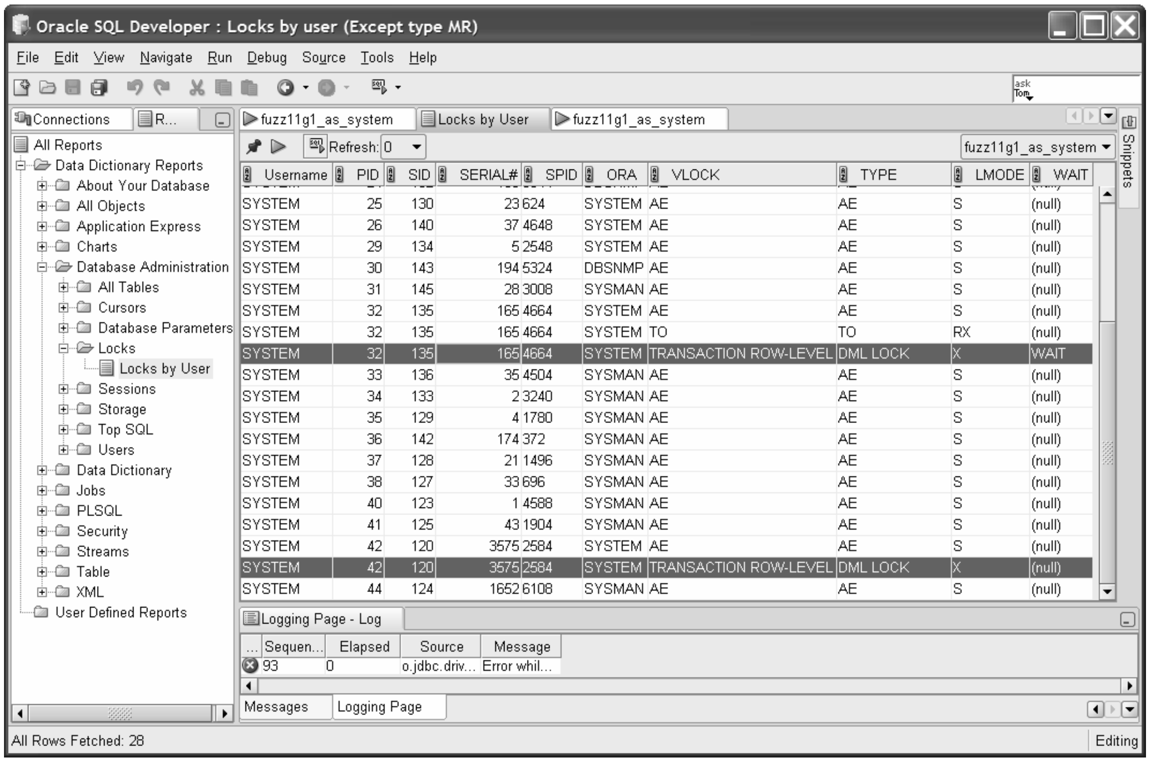
Username	PID	SID	SERIAL#	SPID	ORA	VLOCK	TYPE	LMODE	WAIT
SYSTEM	23	144	182	3540	SYSMAN	AE	AE	S	(null)
SYSTEM	24	132	168	5644	DBSNMP	AE	AE	S	(null)
SYSTEM	25	130	23	624	SYSTEM	AE	AE	S	(null)
SYSTEM	26	140	37	4648	SYSTEM	AE	AE	S	(null)
SYSTEM	29	134	5	2548	SYSTEM	AE	AE	S	(null)
SYSTEM	30	143	194	5324	DBSNMP	AE	AE	S	(null)
SYSTEM	31	145	28	3008	SYSMAN	AE	AE	S	(null)
SYSTEM	32	135	165	4664	SYSTEM	AE	AE	S	(null)
SYSTEM	32	135	165	4664	SYSTEM	TO	TO	RX	(null)
SYSTEM	33	136	35	4504	SYSMAN	AE	AE	S	(null)
SYSTEM	34	133	2	3240	SYSMAN	AE	AE	S	(null)
SYSTEM	35	129	4	1780	SYSMAN	AE	AE	S	(null)
SYSTEM	36	142	174	372	SYSMAN	AE	AE	S	(null)
SYSTEM	37	128	21	1496	SYSMAN	AE	AE	S	(null)
SYSTEM	38	127	33	696	SYSMAN	AE	AE	S	(null)
SYSTEM	40	123	1	4588	SYSMAN	AE	AE	S	(null)
SYSTEM	41	125	43	1904	SYSMAN	AE	AE	S	(null)
SYSTEM	42	120	3575	2584	SYSTEM	AE	AE	S	(null)
SYSTEM	42	120	3575	2584	SYSTEM	TRANSACTION ROW-LEVEL	DML LOCK	X	(null)

...	Sequen...	Elapsed	Source	Message
⊗	93	0	o.jdbc.driv...	Error whil...

All Rows Fetched: 26

Figure 5-2. *Select for update explicit locks for first session in SQL Developer*

The highlighted row shows the lock on the row of the `HR.EMPLOYEES` table for `EMPLOYEE_ID` 200. As soon as the second connection attempts the same statement, it tries to acquire the same exclusive lock, even though the first consequence of doing so would normally only be to select the data. Figure 5-3 shows how Oracle's normal "writes don't block reads" behavior has been overridden by the `FOR UPDATE` technique.



Oracle SQL Developer : Locks by user (Except type MR)

Username	PID	SID	SERIAL#	SPID	ORA	VLOCK	TYPE	LMODE	WAIT
SYSTEM	25	130	23	624	SYSTEM	AE	AE	S	(null)
SYSTEM	26	140	37	4648	SYSTEM	AE	AE	S	(null)
SYSTEM	29	134	5	2548	SYSTEM	AE	AE	S	(null)
SYSTEM	30	143	194	5324	DBSNMP	AE	AE	S	(null)
SYSTEM	31	145	28	3008	SYSMAN	AE	AE	S	(null)
SYSTEM	32	135	165	4664	SYSTEM	AE	AE	S	(null)
SYSTEM	32	135	165	4664	SYSTEM	TO	TO	RX	(null)
SYSTEM	32	135	165	4664	SYSTEM	TRANSACTION ROW-LEVEL	DML LOCK	X	WAIT
SYSTEM	33	136	35	4504	SYSMAN	AE	AE	S	(null)
SYSTEM	34	133	2	3240	SYSMAN	AE	AE	S	(null)
SYSTEM	35	129	4	1780	SYSMAN	AE	AE	S	(null)
SYSTEM	36	142	174	372	SYSMAN	AE	AE	S	(null)
SYSTEM	37	128	21	1496	SYSMAN	AE	AE	S	(null)
SYSTEM	38	127	33	696	SYSMAN	AE	AE	S	(null)
SYSTEM	40	123	1	4588	SYSMAN	AE	AE	S	(null)
SYSTEM	41	125	43	1904	SYSMAN	AE	AE	S	(null)
SYSTEM	42	120	3575	2584	SYSTEM	AE	AE	S	(null)
SYSTEM	42	120	3575	2584	SYSTEM	TRANSACTION ROW-LEVEL	DML LOCK	X	(null)
SYSTEM	44	124	1652	6108	SYSMAN	AE	AE	S	(null)

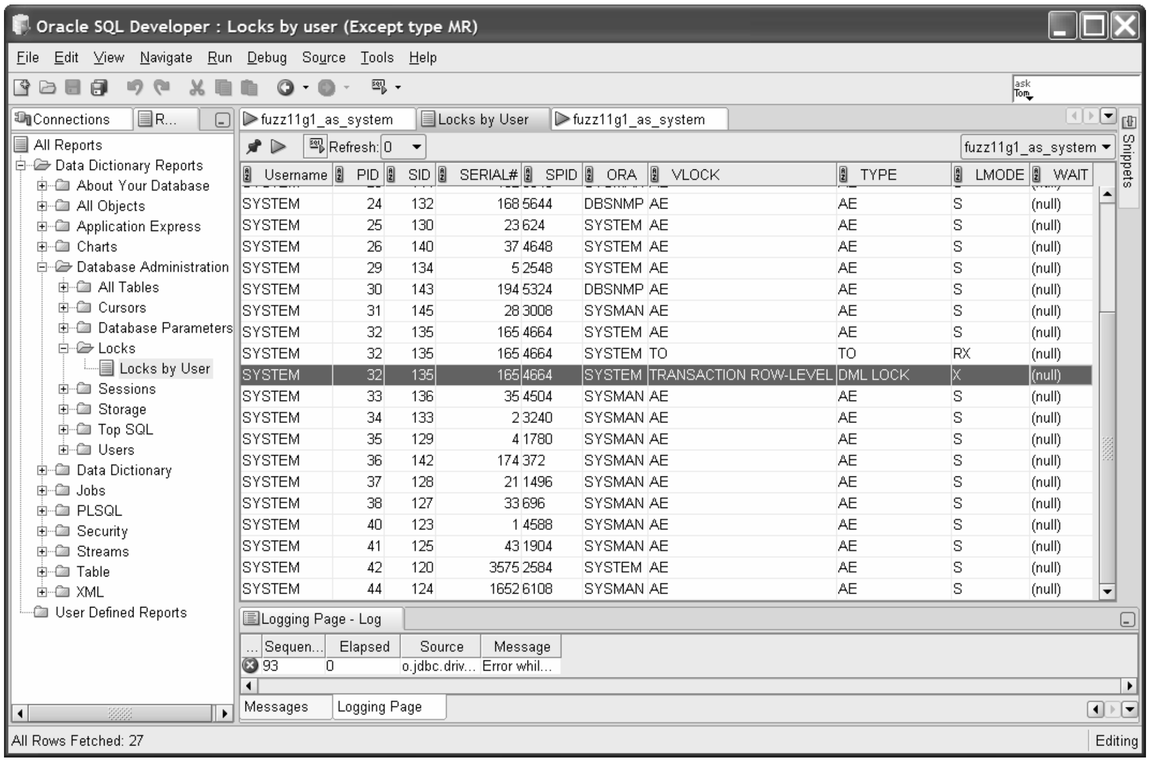
Logging Page - Log

...	Sequen...	Elapsed	Source	Message
⊗	93	0	o.jdbc.driv...	Error whil...

All Rows Fetched: 28

Figure 5-3. *The second select for update blocked, waiting for session 1*

The highlighted row shows the second session in a `WAIT` state, blocked from taking any action until the first session commits or rolls back its work. As soon as we perform our update and commit the results from session 1, our blocking condition clears and the second session takes the exclusive row-level transaction lock it wanted, and can in turn perform its desired update. Figure 5-4 shows the cleared `WAIT` state and the second session now holding the lock.



Username	PID	SID	SERIAL#	SPID	ORA	VLOCK	TYPE	LMODE	WAIT
SYSTEM	24	132	168	5644	DBSNMP	AE	AE	S	(null)
SYSTEM	25	130	23	624	SYSTEM	AE	AE	S	(null)
SYSTEM	26	140	37	4648	SYSTEM	AE	AE	S	(null)
SYSTEM	29	134	5	2548	SYSTEM	AE	AE	S	(null)
SYSTEM	30	143	194	5324	DBSNMP	AE	AE	S	(null)
SYSTEM	31	145	28	3008	SYSMAN	AE	AE	S	(null)
SYSTEM	32	135	165	4664	SYSTEM	AE	AE	S	(null)
SYSTEM	32	135	165	4664	SYSTEM	TO	TO	RX	(null)
SYSTEM	32	135	165	4664	SYSTEM	TRANSACTION ROW-LEVEL	DML LOCK	X	(null)
SYSTEM	33	136	35	4504	SYSMAN	AE	AE	S	(null)
SYSTEM	34	133	2	3240	SYSMAN	AE	AE	S	(null)
SYSTEM	35	129	4	1780	SYSMAN	AE	AE	S	(null)
SYSTEM	36	142	174	372	SYSMAN	AE	AE	S	(null)
SYSTEM	37	128	21	1496	SYSMAN	AE	AE	S	(null)
SYSTEM	38	127	33	696	SYSMAN	AE	AE	S	(null)
SYSTEM	40	123	1	4588	SYSMAN	AE	AE	S	(null)
SYSTEM	41	125	43	1904	SYSMAN	AE	AE	S	(null)
SYSTEM	42	120	3575	2584	SYSTEM	AE	AE	S	(null)
SYSTEM	44	124	1652	6108	SYSMAN	AE	AE	S	(null)

...	Sequen...	Elapsed	Source	Message
⊗	93	0	o.jdbc.driv...	Error whil...

***Figure 5-4.** Cleared from the wait state, the second select for update now holds an exclusive lock*

In this way, you can take explicit pessimistic control of locking in Oracle in those specific circumstances where you can't avoid it.

■ **Caution** While this approach might keep your boss happy, you might want to gently educate him or her over time to the great technology in Oracle, especially the non-blocking capabilities where readers don't block writers and everyone gets a consistent view of the data. They may not want to hear it the first time, but keep dropping it into conversations using the phrase, "You were so clever to buy a database that can do this, Boss." That way, their ego starts to work for you rather than against you, and you can avoid these pessimistic locking techniques as much as possible.

5-12. Synchronizing the Contents of Two Tables

Problem

You manage a system that keeps two or more tables with identical structures, but each table only stores some of the other's rows. You need to synchronize these tables so they each contain the full set of rows.

Solution

Table synchronization is a problem with countless potential solutions. All of Oracle's fundamental SQL capabilities can be used to compare, insert, or delete rows. For our recipe, we'll attempt to use one of the more elegant approaches to synchronization, often overlooked in the rush to tackle the problem in a procedural fashion.

Rather than code elaborate loops, tests, and more in SQL or PL/SQL, Oracle's `MINUS` union operator can be used to determine the difference in two sets of data in a set-wise fashion.

To illustrate our recipe, we'll imagine that our company has two subsidiaries, each of which is tracking a subset of employees in tables with the same structure as the `HR.EMPLOYEES` table. We create these tables, `EMPA` and `EMPB`, with the next `CREATE TABLE` statements.

```
create table hr.empa
as
select *
from hr.employees
where employee_id < 175;

create table hr.empb
as
select *
from hr.employees
where employee_id > 125;
```

The `EMPLOYEE_ID` cutoff chosen is arbitrary and was selected only to ensure there were not only some duplicates in the two tables, but also some rows unique to each table. We can quickly prove that our tables have different data with the next two SQL statements.

```
select min(employee_id), max(employee_id), count(*)
from hr.empa;

MIN(EMPLOYEE_ID)                  MAX(EMPLOYEE_ID)                   COUNT(*)
----------------                  ----------------                   ----------
             100                        174                              75

select min(employee_id), max(employee_id), count(*)
from hr.empb;

MIN(EMPLOYEE_ID)                 MAX(EMPLOYEE_ID)                   COUNT(*)
----------------                 ----------------                   ----------
             126                   212                                  84
```

So we've established our subsidiary tables and can see they have overlapping, but in places different, data. Our solution uses the MINUS operation to bring the tables in to synchronization in the next two statements.

```
insert into hr.empa
select *
from hr.empb
minus
select *
from hr.empa;

insert into hr.empb
select *
from hr.empa
minus
select *
from hr.empb;
```

With our synchronization performed, a quick check shows we seem to now have identical sets of data.

```
select min(employee_id), max(employee_id), count(*)
from hr.empa;

MIN(EMPLOYEE_ID)                  MAX(EMPLOYEE_ID)                    COUNT(*)
----------------                  ----------------                    ----------
             100                      212                             110

select min(employee_id), max(employee_id), count(*)
from hr.empb;

MIN(EMPLOYEE_ID)                    MAX(EMPLOYEE_ID)                   COUNT(*)
----------------                    ----------------                   ----------
             100                     212                                110
```

Naturally, you are welcome to browse the entire set of data in both tables, but you'll be glad to know you won't be disappointed and all of the data will show as synchronized.

How It Works

This recipe uses the `MINUS` operation twice, once in each direction. First, we work on the table `HR.EMPA`, using an `INSERT INTO … SELECT` method. It's the `SELECT` that is key, as shown in the next statement fragment.

```
select *
from hr.empb
minus
select *
from hr.empa;
```

You can run this in isolation, to see how the contents of HR.EMPA itself are “subtracted” from the contents of HR.EMPB to leave us with the rows unique to HR.EMPB. These sample results are abbreviated to save space.

```
EMPLOYEE_ID             FIRST_NAME           AST_NAME             EMAIL   …
-----------             ----------           --------             --------
        175             Alyssa               Hutton               AHUTTON   …
        176             Jonathon             Taylor               JTAYLOR   …
        177             Jack                 Livingston           JLIVINGS …
        178             Kimberely            Grant                KGRANT    …
        179             Charles              Johnson              CJOHNSON …
…
```

These results from the MINUS operation are then fed into the INSERT, and HR.EMPA then has all of its original rows, plus all those rows HR.EMPB had that were not present in HR.EMPA.

We then simply reverse the order of data flow in the second statement, using the MINUS operation to select only those rows in HR.EMPA that were missing from HR.EMPB, and insert them appropriately. It doesn’t matter in which order these two INSERT statements are performed, so long as they both complete successfully.

You can also replace the second SQL statement with an alternative. Because we know HR.EMPA has the complete set of data after the first INSERT, we can remove all the data from HR.EMPB and copy everything over from HR.EMPA without the need for any predicates or union operators. The next code shows the necessary statements for the second step.

```
truncate table HR.EMPB;
insert into HR.EMPB select * from HR.EMPA;
```

This can sometimes be faster, as you avoid the implied sort that using the MINUS operator means.