TRAINING WORKBOOK

Oracle Training

Introduction to Oracle

Confidential Business Information

Prepared For: Release 6.x
Prepared By: SCT
4 Country View Road
Malvern, Pennsylvania 19355
United States of America
Issued: July 2004

Section A: Introduction

Overview

Workbook goal

This course introduces the user how to retrieve and manipulate data which is stored within an Oracle relational database. SQL*Plus is the application, or tool, which will be used to access the data.

Participants in this course will be able to:

- Write statements to obtain information from the database
- Write statements to generate reports
- Manipulate data and process transactions
- Create and modify data tables
- Write programs in which SQL statements are enclosed within procedural statements (such as IF statements)
- Create program units, such as procedures and functions, which can be stored in the database.

Intended audience

SQL is used for all types of database activities by many types of users. However, in order for attendees to receive the optimum benefit of this training, SCT recommends that prospective students come from one of the following groups:

- System administrators
- Database administrators
- Security administrators
- Application programmers
- Decision support system personnel
- End users who have extensive contact with the database

Prerequisites

Prerequisites include:

- Each attendee should be familiar with the concepts of relational databases
- Each attendee should have some familiarity with a computer programming language

In this section

These topics are covered in this section.

Course Overview

Overview

This course will instruct attendees as to how SQL and PL/SQL are used to provide information and update data. The goal of this manual is to help developers tackle real world problems faced every day (at least those problems that can be solved with software). SQL and the rest of Oracle's products offer the potential for incredible development productivity.

This workbook is filled with sample code fragments, as well as complete application components, which can be applied immediately to a situation. Using this workbook, it is SCT's intent to guide participants through the analytical process to develop efficient, maintainable code to successfully support the development and maintenance of an Oracle database.

Workbook contents

Workbook contents

This workbook contains the following sections:

- Section A: Introduction
- Section B: Introduction to Oracle SQL and SQL*Plus
- Section C: Introduction to the Query
- Section D: Conditions and Operators
- Section E: Arithmetic Expressions and Functions
- Section F: Clauses
- Section G: Advanced Queries
- Section H: Insert, Update and Delete
- Section I: Creating and Maintaining Database Objects
- Section J: SQL*Loader
- Section K: SQL*Plus Reporting
- Section L: Introduction to PL/SQL
- Section M: Declaring Variables
- Section N: SQL Statements Within PL/SQL
- Section O: Conditional, Iterative, and Sequential Control
- Section P: Declare and Use Cursors
- Section Q: Handle PL/SQL Errors
- Section R: Procedures and Functions
- Section S: Table Descriptions and Contents
- Section T: Related Files

Section B: Introduction to Oracle SQL and SQL*Plus

Overview

Objectives

This section will examine the following:

- Definition of a relational database
- Definition of a schema
- Object names
- Character sets
- Simple and compound symbols
- Login
- Tables
- Columns
- DUAL table
- Data Dictionary
- SQL Buffer

In this section

These topics are covered in this section.

SQL statements

Overview

This training manual contains a list of common SQL (Structured Query Language) statements. SQL is used to create, store, modify, retrieve, and manage information in any Oracle database. This course will address how SQL is used within an Oracle database.

About SQL

SQL is a set of commands governed by the American National Standards Institution (ANSI) and the International Standards Organization (ISO). Many of the commands in this manual are a superset of the ANSI standard which will be used in the environment called SQL*Plus.

For a complete listing of SQL and SQL*Plus commands, refer to Oracle's SQL, SQL*Plus, and PL/SQL language reference manuals.

Oracle's relational database

What is a relational database?

A relational database is a collection of data items which can be accessed or reassembled in many different ways without having to reorganize the tables.

For instance, as a developer, you may be asked to create two reports:

- A report that lists a student ID, name, birth date, social security number, and home address for a group of students
- A report that lists a student ID, name, social security number, and courses being taken during the current term

Although these are two different reports, they do share some things in common. We are trying to access the student ID, name, and social security number for both.

Data access

Relational databases recognize the fact that data needs to be accessed several ways in order to be valuable.

What is a schema?

A schema is a collection of logical structures of data. A schema is owned by a database user and has the same name as the user. Every database object that is created by a user then becomes part of the user's schema.

This will be discussed in more detail in Section I.

Oracle object names

As with any language, there are conventions used in naming objects. The following rules govern naming objects.

- Names must be 1 to 30 characters in length
- Names cannot contain quotation marks
- Names are not case sensitive
- Each name must begin with an alphabetic character
- A name cannot be an Oracle reserved word
- A name cannot be the duplicate of another database object owned by the same user

Continued on the next page

Oracle's relational database, Continued

Character set

SQL is not a case-sensitive language. Uppercase and lowercase letters are treated the same way with the exception of literal strings (alphanumeric characters surrounded by single quotes) and character variables (variables designated with the "&" prefix). Though SQL is case-insensitive, stick to using UPPERCASE for keywords and reserved words. It is much easier to read.

Type	Character
Letters	A-Z, a-z
Digits	0-9
Symbols	~ ! @ # $ % & * ( ) - + = [ ] { } " ‘ , . ? | : ;
Whitespace	tab, space, carriage return

Simple and compound symbols

Symbol	Description
;	statement terminator
%	multibyte wild card symbol
_	singlebyte wild card symbol (underscore)
<> and !=	not equals
||	concatenation operator
<= and >=	relational operator
--	single line comment indicator (dash)
/* and */	multiline comment block delimiters

Login

Logging in

To retrieve and manipulate data, first connect to the database.

There are two ways to connect to the SQL*Plus environment.

Login from Unix/VMS prompt

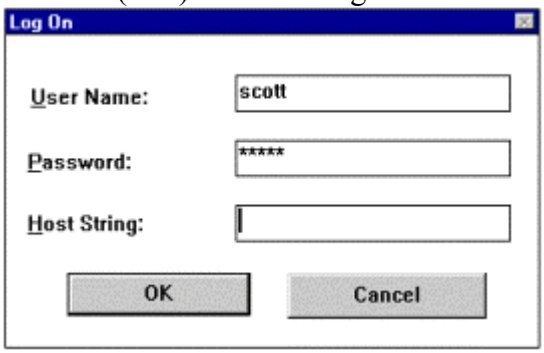
At the operating system prompt, type SQLPLUS. Then enter a username and password. After entering a password, the system goes to the SQL prompt:

```
prompt$    SQLPLUS


SQL*Plus: Release 8.1.7.0.0 - Production on Mon Apr 1
11:36:35 2002

(c) Copyright 2000 Oracle Corporation.  All rights reserved.

Enter user-name: train01
Enter password:

Connected to:
Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production

SQL>
```

Your output may differ from what is shown here, depending on which releases of the software you have installed.

Login from PC

When logging in via client server, the prompt may also include a request for a database (SID) or host string:



Contact the database administrator for proper database names (SID).

Release numbers

Note the Oracle release numbers. They are normally overlooked in day to day activity, but when seeking assistance from SCT or the Oracle Corporation, undoubtedly these version numbers will be requested by the contact person.

SQL Buffer

SQL buffer

SQL*Plus will store the last command in the SQL buffer. Use the following commands to edit statements stored in the buffer.

Command	Abbreviation	Purpose
APPEND *text*	a *text*	Add *text* to the end of a line
CHANGE/*old/new*	c/*old/new*	Change *old* to *new* in line
CHANGE/*text*	c/*text*	Delete *text* from a line
CLEAR BUFFER	CL BUFF	Delete lines
DEL	none	Delete a line
INPUT	I	Add one or more lines
LIST	l	List all lines in buffer
LIST *n*	l *n*	List a specific line
LIST *m n*	l *m n*	List a range of lines

Bypass the Buffer

At the SQL> prompt type the keyword EDIT. This action will open the system editor. Save the file and exit the editor in the manner defined by the editor to return to the SQL> prompt.

To execute the commands specified in the file, type one of these command lines (they are synonymous).

```
SQL>          START <file_name.sql>
SQL>          @<file_name.sql>
```

View editor settings

In order to view the editor settings, type the keyword DEFINE at the SQL prompt.

```
SQL> DEFINE

DEFINE _SQLPLUS_RELEASE = "800060000" (CHAR)
DEFINE _EDITOR          = "Notepad" (CHAR)
DEFINE _O_VERSION       = "Oracle8i Enterprise Edition
Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production" (CHAR)
DEFINE _O_RELEASE       = "801070000" (CHAR)
```

Change the default editor

To change the default editor, use the keyword DEFINE_EDITOR:

```
SQL> DEFINE_EDITOR = <editor name>
```

Tables

Tables

A table is the basic structure to hold user information. Think of a table as a spreadsheet made up of columns and rows. Column definitions fall under three categories: character, numeric, and date datatypes.

Table definition

To view the definition of a table, use the DESC command (abb. for describe):

```
SQL>    DESC <table_name>
```

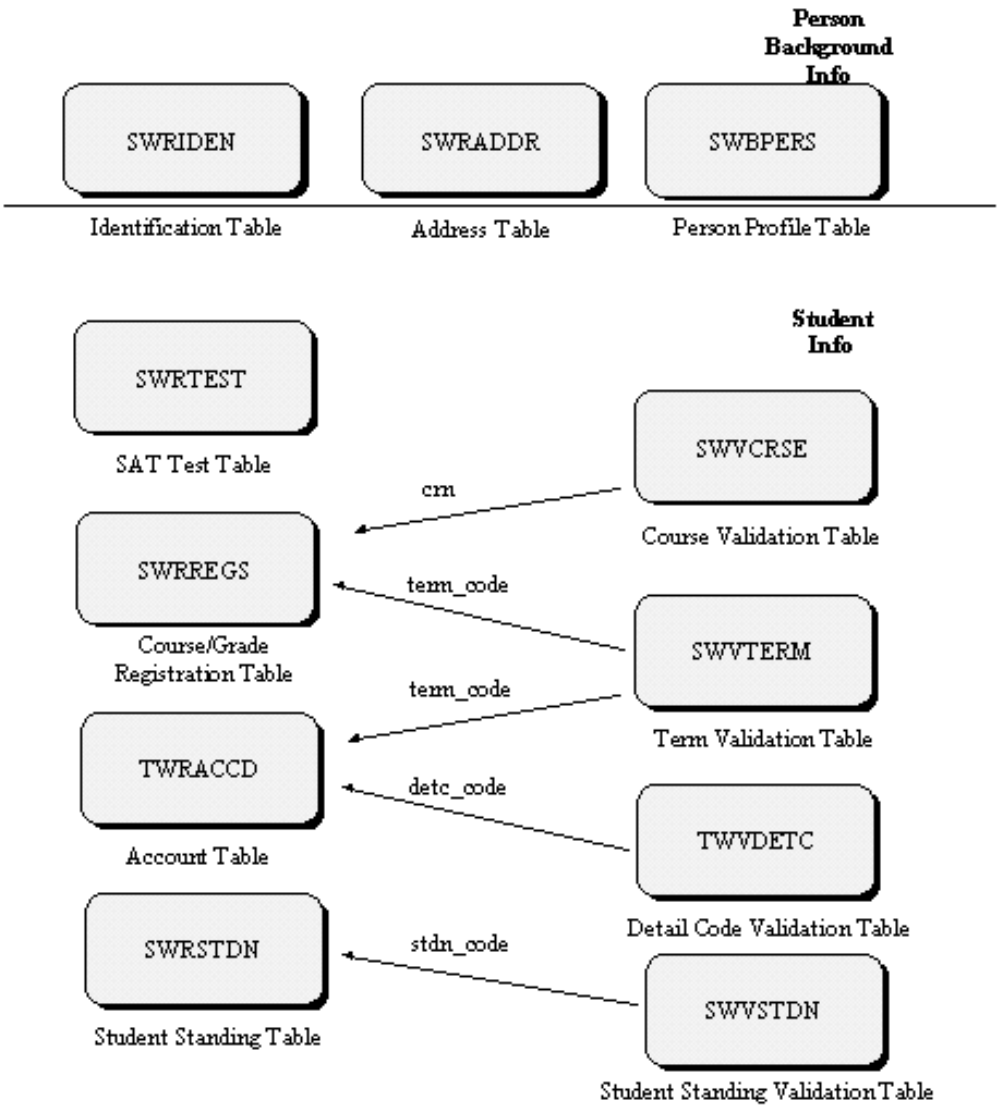
Table examples

Many of the code examples and exercises in this manual use tables designed for this course. Take a moment to become familiar with these tables using the DESC command:

```
SQL>    DESC  swriden
SQL>    DESC  swbpers
SQL>    DESC  swbaddr
SQL>    DESC  swvcrse
SQL>    DESC  swrregs
SQL>    DESC  swrstdn
SQL>    DESC  swrtest
SQL>    DESC  swvstdn
SQL>    DESC  swvtele
SQL>    DESC  swvterm
SQL>    DESC  twraccd
SQL>    DESC  twvdetc
```

Table relationships

Diagram



Naming Conventions

Characters

The first character in the name of all SCT Banner tables identifies the system which owns it.

The second character is system specific and identifies the application module to which the table refers.

The third character identifies the type of table: B(ase), R(epeating), or V(alidation) table.

The fourth through seventh positions represent a four character description for the table.

Example

For example: SWRIDEN

- **S**(tudent)
- **W** (client-developed object)
- **R**(epeating table)
- **IDEN**(tification)

Columns

Tables are made up of columns. The number of columns in a table can range from 1 to 254. Each column is defined using a specific datatype.

The following are the most common datatypes:

- VARCHAR2(n)
  Variable length character string having a maximum size *n* of 2000.
- CHAR(n)+
  Fixed length character data having a maximum size *n* of 255.
- VARCHAR++
  Currently synonymous with VARCHAR2 datatypes.
- LONG
  Variable length data up to 2 gigabytes.
- NUMBER(p,s)
  Numeric datatype having a precision *p* and scale *s*.
- DATE
  Valid dates range from January 1, 4712 BC to December 31, 4712 AD.

+ The CHAR(n) datatype is a backwards compatible datatype associated with DB2 applications.
++ It is highly recommended that this datatype never be used. Oracle may change its definition in a future release.

Data Dictionary

Definition

The data dictionary is a collection of views containing information concerning the database, database objects, users, and events. It is also called "meta data."

To access the views contained in the data dictionary, type the following:

```
SQL>    SELECT * FROM DICTIONARY;
```

Importance

The contents of the dictionary tables and views will be explored as they relate to the topics of discussion. It is highly recommended that users become familiar with the contents of the data dictionary. A great deal of information concerning the SCT Banner database may be gleaned from the data dictionary.

Dictionary views

The dictionary views can be divided into three categories: ALL, USER, and DBA views.

- Dictionary view names beginning with "USER" contain information about the database objects and events owned by a schema (user).
- Dictionary view names beginning with "ALL" contain information about the database objects to which the user has access.

Refer to an Oracle Server Reference manual for a complete listing of data dictionary views and their descriptions.

Selectable views

The following are data dictionary views that can be selected to learn more information about objects in the database:

- USER_TABLES (TABS)
- USER_VIEWS
- ALL_TABLES
- ALL_TAB_COMMENTS
- ALL_COL_COMMENTS

List table information

To list all of the table information within your schema, type the following:

```
SELECT * FROM USER_TABLES;
```

A printed listing of the contents of these tables may be found in Section T..

DUAL

DUAL table

DUAL is a table automatically created by Oracle along with the data dictionary. It is in the schema of the user SYS, but is accessible to all users by the name DUAL. It has one column, DUMMY, defined to be a VARCHAR2, and it contains one row with the value of "X".

Selecting from DUAL

Selecting from DUAL is useful for computing a constant expression with the SELECT command. Because it has only one row, the constant is only returned once.

The following example obtains the current system date:

```
SQL>    SELECT SYSDATE FROM DUAL;

  SYSDATE
---------
19-MAY-97
```

Section C: Introduction to the Query

Overview

Objectives

Upon completion of this section, each attendee will be able to write queries that perform the following:

- Select all or specific columns
- Remove duplicate rows
- Create substitute column headings

In this section

These topics are covered in this section.

Select statements

Overview

The SELECT statement retrieves rows from one or more tables. It pulls all rows, or adds conditions so that only the information needed is retrieved.

```
SELECT <column name> <constant> FROM <table name>;
```

The purpose of a SELECT statement is to display columns and rows from one or more tables. This is also known as querying the database.

Examples

```
SQL> SELECT  last_name
       FROM  swriden;


LAST_NAME
------------------
Brown
Brown
Erickson
Erickson
Johnson
Jones
Jones-Erickson
Smith
White

SQL> SELECT  id
       FROM  swriden;


ID
---------
157834585
176536782
692568211
578549991
3539543
145672112
145672112
5829934
543853339
...
```

Selecting multiple columns

Separator

To select multiple columns, separate them by a comma.

Example

```
SQL> SELECT  last_name, first_name
       FROM  swriden;

LAST_NAME                 FIRST_NAME
------------------------- ----------
Brown                     Julie
Brown                     Julie
Erickson                  Ralph
Erickson                  Susan
Johnson                   Peter
Jones                     Sandy
Jones-Erickson            Sandy
Smith                     Robert
White                     Nancy
```

Selecting a literal

Literals

Selections do not have to be on a column from a table or database object. Select a literal if it is enclosed in quotes.

Example

```
SQL> SELECT 'Student Name is', first_name,
            last_name
       FROM  swriden;

'STUDENTNAMEIS' FIRST_NAME      LAST_NAME
--------------- --------------- --------------------
Student Name is Julie           Brown
Student Name is Julie           Brown
Student Name is Peter           Johnson
Student Name is Robert          Smith
    ...
```

DISTINCT clause

DISTINCT

The DISTINCT clause specifies that duplicate rows should be removed before the rows are returned. A row is considered a duplicate of another if every value for each column of the SELECT clause matches that of another row(s).

Without DISTINCT clause

```
SQL> SELECT  last_name
       FROM  swriden;

LAST_NAME
-----------------
Brown
Brown
Erickson
Erickson
Johnson
Jones
Jones-Erickson
Smith
White
```

With DISTINCT clause

```
SQL> SELECT DISTINCT last_name
       FROM swriden;

LAST_NAME
-----------------
Brown
Erickson
Johnson
Jones
Jones-Erickson
Smith
White
```

Continued on the next page

DISTINCT clause, Continued

Without DISTINCT

```
SQL> SELECT last_name, first_name
       FROM swriden;

LAST_NAME       FIRST_NAME
----------      -----------
Brown           Julie
Brown           Julie
Smith           Robert
Johnson         Peter
Jones           Sandy
Jones-Erickson  Sandy
Erickson        Ralph
Erickson        Susan
White           Nancy
```

With DISTINCT

```
SQL> SELECT DISTINCT last_name, first_name
       FROM swriden;

LAST_NAME       FIRST_NAME
----------      -----------
Brown           Julie
Erickson        Ralph
Erickson        Susan
Johnson         Peter
Jones           Sandy
Jones-Erickson  Sandy
Smith           Robert
White           Nancy
```

Selecting all columns

Separator

To retrieve all columns, specify all columns separated by commas, or use an asterisk (*).

```
SELECT * FROM <table name>;
```

Example

```
SQL> SELECT * FROM swriden;
```

Column heading aliases

Aliases

Aliases are used to substitute column headings in the SELECT statement.

Example

```
SQL>     SELECT  DISTINCT pidm "ID Number",
                 last_name "Last Name",
                first_name "First Name"
           FROM  swriden;

ID Number Last Name                 First Name
--------- ------------------------- ----------
    12340 Brown                     Julie
    12341 Smith                     Robert
    12342 Johnson                   Peter
    12343 Jones                     Sandy
    12343 Jones-Erickson            Sandy
    12344 Erickson                  Ralph
    12345 Erickson                  Susan
    12346 White                     Nancy
```

Multiple words

Enclosing the Heading Alias within double quotes allows the use of multiple-word headings. If the alias is one word, then the quotes are not necessary.

Pseudo-columns

Pseudo-columns

A pseudo-column is a column that yields a value when selected, but is not actually a column in a table. Below are some frequently used pseudo-columns.

ROWNUM

Returns a number indicating the sequence in which a row was selected from a table or set of joined rows.

```
SQL> SELECT ROWNUM, pidm, last_name
       FROM swriden;

   ROWNUM      PIDM LAST_NAME
--------- --------- ------------------
        1     12340 Brown
        2     12340 Brown
        3     12341 Smith
        4     12342 Johnson
      ...
```

ROWID

The ROWID is an internally generated and maintained six-byte binary value which identifies a row of data in a table. The information in ROWID provides the exact physical location of a row in the database. The return value of ROWID provides this value in a readable format.

```
SQL> SELECT rowid, last_name
  2    FROM swriden;

ROWID              LAST_NAME
------------------ -------------------------
AAAKIQAAGAAAabTAAA Brown
AAAKIQAAGAAAabTAAB Brown
AAAKIQAAGAAAabTAAC Smith
AAAKIQAAGAAAabTAAD Johnson
AAAKIQAAGAAAabTAAE Jones
AAAKIQAAGAAAabTAAF Jones-Erickson
AAAKIQAAGAAAabTAAG Erickson
AAAKIQAAGAAAabTAAH Erickson
AAAKIQAAGAAAabTAAI White
AAAKIQAAGAAAabTAAJ Marx
AAAKIQAAGAAAabTAAK Clifford
```

Continued on the next page

Pseudo-columns, Continued

ROWID format

The ROWID format appears as follows:

```
BBBBBB.RRRR.FFFFF
```

BBBBBB	the block in the database
RRRRR	the row in the block (where the first row begins with zero)
FFFFF	the database file

Note: Prior to 8.x, the ROWID was something like a 16-digit string, with each digit being base-16. From 8.x on, the ROWID is something like a 20-digit string, with each digit being base-64.

SYSDATE

The current date and time.

```
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
---------
01-APR-02
```

USER

The name of the current user.

```
SQL> SELECT USER FROM DUAL;

USER
---------
TRAIN01
```

Self Check

Directions

Use the information you have learned in this workbook to complete this self check activity.

For the following exercises, use the Student subject table: **SWRREGS.**

Exercise 1

Write a query to return all the columns.

Exercise 2

Write a query to return the PIDM (Personal identification master), CRN (course number), and GPA (grade point average) for each record.

Continued on the next page

Self Check, Continued

Exercise 3

Write a query to return the unique course numbers.

Exercise 4

Write a query to return the row number, row identification, and PIDM for each record.

Continued on the next page

Self Check, Continued

Exercise 5 Select the system date from the dummy table DUAL.

Self Check – Answer Key

Exercise 1

Write a query to return all the columns.

```
SQL> SELECT * FROM swrregs;

      PIDM TERM_C        CRN        GPA ACTIVITY_
---------- ------ ---------- ---------- ---------
     12340 199701      10001        3.2 27-MAR-02
     12349 199701      10001        3.4 27-MAR-02
     12350 199701      10001        3.1 27-MAR-02
     12346 199602      10001        2.6 27-MAR-02
     12348 199602      10001        2.9 27-MAR-02
     12343 199602      10001          3 27-MAR-02
     12340 199701      10007        2.1 27-MAR-02
     12340 199602      10015        2.8 27-MAR-02
     12340 199702      10017            27-MAR-02
     12340 199702      10004            27-MAR-02
     12340 199602      10005          2 27-MAR-02
  ... 44 rows selected
```

Exercise 2

Write a query to return the PIDM (Personal Identification Master), CRN (course number), and GPA (grade point average) for each record.

```
SQL> SELECT pidm, crn, gpa
       FROM swrregs;

      PIDM        CRN        GPA
---------- ---------- ----------
     12340      10001        3.2
     12349      10001        3.4
     12350      10001        3.1
     12346      10001        2.6
     12348      10001        2.9
     12343      10001          3
     12340      10007        2.1
     12340      10015        2.8
     12340      10017
... 44 rows selected
```

Continued on the next page

Exercise 3

Write a query to return the unique course numbers.

```
SQL> SELECT DISTINCT crn
       FROM swrregs;

       CRN
----------
     10001
     10002
     10003
     10004
     10005
     10006
     10007
     10008
     10009

    ... 22 rows selected
```

Exercise 4

Write a query to return the row number, row identification, and PIDM for each record.

```
SQL> SELECT ROWNUM, ROWID, pidm
       FROM SWRREGS;

    ROWNUM ROWID                  PIDM
---------- ------------------ ----------
         1 AAAKIYAAGAAAadTAAA      12340
         2 AAAKIYAAGAAAadTAAB      12349
         3 AAAKIYAAGAAAadTAAC      12350
         4 AAAKIYAAGAAAadTAAD      12346
         5 AAAKIYAAGAAAadTAAE      12348
         6 AAAKIYAAGAAAadTAAF      12343
         7 AAAKIYAAGAAAadTAAG      12340
         8 AAAKIYAAGAAAadTAAH      12340
         9 AAAKIYAAGAAAadTAAI      12340
        10 AAAKIYAAGAAAadTAAJ      12340
        11 AAAKIYAAGAAAadTAAK      12340

       ... 44 rows selected
```

Continued on the next page

Exercise 5

Select the system date from the dummy table DUAL.

```
SQL> SELECT SYSDATE
       FROM DUAL;

SYSDATE
---------
<today's date>
```

Section D: Conditions and Operators

Overview

Purpose

So far, we have discussed ways to query information from a particular table or database. In this section, we will discuss making queries more selective by only returning rows which meet certain criteria.

Objectives

Upon completion of this section, each attendee will be able to:

- Specify all or specific rows based upon search criteria using:
  - comparison operators
  - logical operators
  - miscellaneous operators
- Use the pseudo-columns ROWNUM and ROWID to narrow searches
- Use parameters so that users are prompted for column or search information

In this section

These topics are covered in this section.

Conditions

What is a condition?

In SQL, a condition is a restriction on a query so that only rows which meet those conditions will be returned. Conditions are added through the use of the WHERE clause.

The WHERE clause

Syntax

```
SELECT    select_list
  FROM    table_name
 WHERE    condition
```

WHERE Clause Components

- Column name or expression
- Comparison operator
- Expression or column name

Example

```
SQL> SELECT    last_name, first_name
       FROM    swriden
      WHERE    last_name  = 'Erickson';

LAST_NAME                 FIRST_NAME
------------------------- ---------------
Erickson                  Ralph
Erickson                  Susan
```

Comparison operators

Operators

Operators are used in WHERE clauses to compare values.

Operator	Function	Examples
( )	Overrides precedence	SELECT (X + Y) / (X - Y)
=	Test for equality	...WHERE last_name = 'SMITH'
!=, ^=, <>	Test for inequality	...WHERE last_name <>'SMITH'
>	Greater than	...WHERE sat_verbal > 450
>=	Greater than or equal to	...WHERE sat_verbal >= 450
<	Less than	...WHERE sat_math < 450
<=	Less than or equal to	...WHERE sat_math <= 450

Using ROWNUM

Take advantage of the pseudo-column ROWNUM to limit the number of rows returned. The ROWNUM is assigned to a row after it has evaluated the WHERE clause.

```
SQL> SELECT    ROWNUM, pidm, last_name
       FROM    swriden
      WHERE    ROWNUM < 5;

ROWNUM          PIDM LAST_NAME
--------- ---------- -----------------
        1      12340 Brown
        2      12340 Brown
        3      12341 Smith
        4      12342 Johnson
```

Using ROWID

Use the pseudo-column ROWID (the actual location of a row in a table) to distinguish between duplicate rows. Then, delete the extra row based on the ROWID.

```
SQL>  SELECT    ROWID, pidm, ssn, birth_date,
                mrtl_code, sex, confid_ind
    FROM  swbpers;

ROWID              PIDM     SSN         BIRTH_DAT M S C
------------------ ---------      ---------   --------- - - -
0000030F.0000.0003    12340       555444412   02-AUG-73 S F Y
0000030F.0001.0003    12341       682082678   12-NOV-70 M M N
0000030F.0002.0003    12343       555444412   02-AUG-73 S F Y
0000030F.0003.0003    12344       198767345   02-AUG-73 S M N
0000030F.0004.0003    12345       555444412   05-JAN-54 M   Y
0000030F.0005.0003    12340       555444412   02-AUG-73 S F Y
```

Logical operators

Definition

Compound logical expressions are two or more expressions connected by logical operators.

Diagram 1

AND	true	false	null
True	true	false	null
False	false	false	false
Null	null	false	null

Example 1

```
SQL> SELECT    stat_code, zip, pidm
       FROM    swbaddr
      WHERE    stat_code = 'PA'
        AND    zip = '19380';

STA ZIP             PIDM
--- ---------- ---------
PA  19380           12340
```

Diagram 2

OR	true	false	null
true	true	true	true
false	true	false	null
null	true	null	null

Example 2

```
SQL> SELECT     last_name, first_name, id
       FROM     swriden
      WHERE     last_name = 'Johnson'
         OR     id  = 543853339;

LAST_NAME     FIRST_NAME         ID
------------ --------------    ---------
Johnson       Peter             3539543
White         Nancy             543853339
```

Between operator

Between operator

The Between condition is used to return rows containing values between two specified values (inclusive).

Example

```
SQL>  SELECT pidm, sat_verbal
        FROM swrtest
       WHERE sat_verbal BETWEEN 520 AND 800;

   PIDM   SAT_VERBAL
--------- ----------
    12340        550
    12341        530
    12342        660
    12341        590
    12343        530
    12345        590
    12346        630
    12346        520
    12346        520
```

With logical operators

Keep in mind that logical operators may evaluate character data as well:

```
SQL>     SELECT detc_code, description
           FROM twvdetc
          WHERE detc_code BETWEEN 'BOOK' AND 'CHEK';

DETC DESCRIPTION
---- -----------------
BOOK Book Charges
CHEK Check Payment
CASH Cash Payment
```

In operator

In operator

The In operator is used in the WHERE clause to retrieve data which matches a value in the list provided.

Example

```
SQL> SELECT pidm, id, last_name, first_name
       FROM swriden
      WHERE last_name IN ('Smith', 'Jones', 'Johnson');

PIDM             ID         LAST_NAME       FIRST_NAME
-------------    ---------- --------------- ----------
12341            5829934    Smith           Robert
12342            3539543    Johnson         Peter
12343            145672112  Jones           Sandy
```

Like operator

Like operator

The Like condition is used to select information based on pattern matching. There can be more than one wildcard in a Like condition.

Wildcard characters

- % matches any number of characters
- _ matches a single character

Example 1

```
SQL> SELECT crn, description
       FROM swvcrse
      WHERE description LIKE 'S%';

CRN      DESCRIPTION
------   -------------------
10012    Statistics
10015    Speech
10020    Swimming
```

Example 2

```
SQL> SELECT crn, description
       FROM swvcrse
      WHERE description LIKE'%l_gy';

CRN      DESCRIPTION
-----    --------------------
10005    Biology
10006    Zoology
10008    Psychology
10011    Anthropology
```

Not operator

Not operator

The Not operator may be used to make a negative condition out of the following operators:

- NOT BETWEEN...AND...
- NOT IN (list)
- IS NOT NULL
- NOT LIKE

Example

```
SQL>   SELECT  id, first_name, last_name, change_ind
         FROM  swriden
        WHERE  change_ind IS NOT NULL;

ID            FIRST_NAME      LAST_NAME         C
----------    --------------- ----------------- -
176536782     Julie           Brown             I
145672112     Sandy           Jones             N
```

Precedence rules

Expression evaluation

When a condition contains more than one expression, Oracle evaluates each expression according to the order of evaluation. The order of evaluation is determined by the precedence of the connecting operators.

Equal precedence

=, !=, >, >=, <, <=, IN, LIKE, IS NULL, BETWEEN...AND...

Logical operators

The logical operators are evaluated in this order:

- 1. NOT
- 2. AND
- 3. OR

Example 1 (Incorrect)

For example, suppose that you wish to retrieve data for a survey for all students who are married or have a birth date before '01-Jan-60'. Do not include confidential records.

```
SQL> SELECT    *
       FROM    swbpers
      WHERE    confid_ind = 'N'
        AND    mrtl_code = 'M'
         OR    birth_date < '01-JAN-60';

PIDM           SSN       BIRTH_DAT M S C ACTIVITY_
---------      --------- --------- - - - ---------
12341          682082678 12-NOV-70 M M N 27-JUN-97
12345          555444412 05-JAN-54 M   Y 24-JUN-97
```

What data are you actually retrieving?

Example 2 (correct)

The correct way:

```
SQL> SELECT    *
       FROM    swbpers
      WHERE    confid_ind = 'N'
        AND    (mrtl_code = 'M'
         OR    birth_date < '01-JAN-60');


PIDM           SSN       BIRTH_DAT M S C ACTIVITY_
---------      --------- --------- - - - ---------
12341          682082678 12-NOV-70 M M N 27-JUN-97
```

Parameters

Ampersand prompt

Use the ampersand (&) to prompt users for parameters. When SQL*Plus encounters an ampersand variable, the user is prompted for input.

Enter substitute variables anywhere in a SQL statement command except in the first word entered at the prompt.

Example 1

```
SQL> SELECT *
       FROM swvcrse
      WHERE crn = '&Course_Num';

Enter value for course_num: 10021

CRN           DESCRIPTION              ACTIVITY_
---------     ----------------------   ---------
10021         Economics                17-APR-97
```

Example 2

```
SQL> SELECT *
       FROM &table;

Enter value for table: SWRREGS

PIDM     TERM_C     CRN        GPA   ACTIVITY_
-------- ---------  --------- ----- -----------
12340    199701     10001     3.2   17-APR-97
12340    199701     10007     2.1   17-APR-97
12340    199701     10015     2.8   17-APR-97
12340    199701     10017           17-APR-97
 ...
```

Continued on the next page

Parameters, Continued

Double ampersand

Avoid repetitious variable entry for the entire session by using the double ampersand (&&).

```
SQL> SELECT &&number, &&number + 10,
            &&number + 20
       FROM DUAL;

Enter value for number: 1

1         1+10       1+20
-------   --------   -------
1         11         21
```

Undefining double ampersand variables

To undefine a double ampersand variable, use the undefine command.

```
SQL> undefine number
```

Self Check

Directions Use the information you have learned in this workbook to complete this self check activity.

Exercise 1 Query the first 5 rows from the **SWRREGS** table.

Try to query rows 3 and higher from **SWRREGS**. What occurred?

Exercise 2 Using **SWBADDR**, find the city, state, and zip code for PIDM (internal identification master) 12340. Remember to describe the table first to see the names of the fields.

Continued on the next page

Self Check, Continued

Exercise 3

From **SWRIDEN**, query the students who do not have the last name of 'Erickson'. Remember to only include the most current record for each student. Return all columns.

Exercise 4

Use the single ampersand in a SQL statement to prompt for a table name, displaying all columns within a table. Run this query for **SWRREGS** and **SWRIDEN.**

Continued on the next page

Self Check, Continued

Exercise 5

Retrieve the first name, last name, and ID from **SWRIDEN** where the ID begins with a 1. (Hint: use the keyword LIKE)

Retrieve the first name, last name, and ID from **SWRIDEN** where the fourth character in the ID field is a 9 and the total length of the column is 7.

Exercise 6

Using the **SWRREGS** table, list the PIDM, CRN, and GPA for students who have taken golf, tennis, or swimming. (Hint: look at the validation table SWVCRSE first to find the code values for the courses.)

Continued on the next page

Self Check, Continued

Note For exercises 7 - 9, refer to the SAT test table, **SWRTEST.**

Exercise 7 Retrieve the student records where the student achieved above 550 in both math and verbal.

Exercise 8 Retrieve the student records where the student achieved above 550 in either math or verbal.

Exercise 9 Retrieve the student records where the student took the SAT between '01-JAN-97' and '31-MAY-97' (if the two-digit year does not work, try entering the four-digit year).

Self Check – Answer Key

Exercise 1

Query the first 5 rows from the **SWRREGS** table.

```
SQL> SELECT * FROM swrregs
                2  WHERE ROWNUM <6;

      PIDM TERM_C        CRN        GPA ACTIVITY_
---------- ------ ---------- ---------- ---------
     12340 199701      10001        3.2 27-MAR-02
     12349 199701      10001        3.4 27-MAR-02
     12350 199701      10001        3.1 27-MAR-02
     12346 199602      10001        2.6 27-MAR-02
     12348 199602      10001        2.9 27-MAR-02

5 rows selected.
```

Try to query rows 3 and higher from **SWRREGS**. What occurred?

```
SQL>  SELECT *
           FROM swrregs
          WHERE ROWNUM > 3;

   no rows selected
```

This will never return any rows because ROWNUM is assigned on order of display. For example, the first row to be displayed is assigned ROWNUM of 1, but then fails the condition of ROWNUM > 3 and is consequently discarded. The second row fetched will therefore be assigned ROWNUM of 1, and will also fail the condition. Subsequently, all rows will fail to meet the condition.

Exercise 2

Using **SWBADDR**, find the city, state, and zip code for PIDM (internal identification master) 12340. Remember to describe the table first to see the names of the fields.

```
SQL> SELECT    city, stat_code, zip
     FROM      swbaddr
    WHERE      pidm = 12340;

CITY                 STA ZIP
-------------------- --- ----------
WEST CHESTER         PA  19380
```

Continued on the next page

Self Check – Answer Key, Continued

Exercise 3

From **SWRIDEN**, query the students who do not have the last name of 'Erickson'. Remember to only include the most current record for each student. Return all columns.

```
SQL> SELECT *
        FROM swriden
       WHERE change_ind IS NULL
         AND last_name <> 'Erickson';

PIDM  ID        LAST_NAME      FIRST_NAME          MI C
ACTIVITY_
----- --------- -------------- --------------- -- - ---------
12340 157834585 Brown          Julie           K    15-JUN-97
12341 5829934   Smith          Robert          E    11-JUN-97
12342 3539543   Johnson        Peter           S    03-JUN-97
  ... 8 rows selected
```

Exercise 4

Use the single ampersand in a SQL statement to prompt for a table name, displaying all columns within a table. Run this query for **SWRREGS** and **SWRIDEN.**

```
SQL> SELECT * FROM &TABLE;
Enter value for table: swrregs
old   1: SELECT * FROM &TABLE
new   1: SELECT * FROM swrregs

SQL> /
Enter value for table: swriden
old   1: SELECT * FROM &TABLE
new   1: SELECT * FROM swriden
```

Continued on the next page

Exercise 5

Retrieve the first name, last name, and ID from **SWRIDEN** where the ID begins with a 1. (Hint: use the keyword LIKE)

```
SQL> SELECT first_name, last_name, id
        FROM swriden
       WHERE id like '1%';

 FIRST_NAME      LAST_NAME      ID
 --------------- -------------- ---------
 Julie           Brown          157834585
 Julie           Brown          176536782
 Sandy           Jones          145672112
 Sandy           Jones-Erickson 145672112
```

Retrieve the first name, last name, and ID from **SWRIDEN** where the fourth character in the ID field is a 9 and the total length of the column is 7.

```
SQL>  SELECT first_name, last_name, id
        FROM swriden
       WHERE id LIKE '___9___';

 FIRST_NAME      LAST_NAME      ID
 --------------- -------------- -------
 Robert          Smith          5829934
 Peter           Johnson        3539543
```

Exercise 6

Using the **SWRREGS** table, list the PIDM, CRN, and GPA for students who have taken golf, tennis, or swimming. (Hint: look at the validation table SWVCRSE first to find the code values for the courses.)

```
SQL> SELECT * FROM swvcrse;

 SQL> SELECT   pidm, crn, gpa
        FROM  swrregs
       WHERE   crn IN (10018, 10019, 10020);

    PIDM       CRN       GPA
 ------- --------- ---------
   12341     10018       2.4
   12342     10020
   12344     10019
```

Continued on the next page

Self Check – Answer Key, Continued

Note For exercises 7 - 9, refer to the SAT test table, **SWRTEST.**

Exercise 7 Retrieve the student records where the student achieved above 550 in both math and verbal.

```
SQL> SELECT *
        FROM swrtest
       WHERE sat_verbal > 550
         AND sat_math > 550;

    PIDM TEST_DATE SAT_VERBAL  SAT_MATH ACTIVITY_
 ------- --------- ---------- --------- ---------
   12341 06-JUN-97        590       610 17-MAY-97
   12345 06-JUN-97        590       620 27-APR-97
   12346 17-MAY-97        630       590 27-APR-97
```

Exercise 8 Retrieve the student records where the student achieved above 550 in either math or verbal.

```
SQL> SELECT *
        FROM swrtest
       WHERE sat_verbal > 550
          OR sat_math > 550;

    PIDM TEST_DATE SAT_VERBAL  SAT_MATH ACTIVITY_
 ------- --------- ---------- --------- ---------
   12341 17-MAY-97        530       580 27-APR-97
   12342 17-MAY-97        660       520 27-APR-97
   12341 06-JUN-97        590       610 17-MAY-97
   12345 06-JUN-97        590       620 27-APR-97
   12346 17-MAY-97        630       590 27-APR-97
```

Exercise 9 Retrieve the student records where the student took the SAT between '01-JAN-97' and '31-MAY-97' (if the two digit year does not work try entering the four digit year).

```
SQL> SELECT * FROM swrtest
       WHERE test_date BETWEEN
'01-JAN-97' AND '31-MAY-97';

    PIDM TEST_DATE SAT_VERBAL   SAT_MATH ACTIVITY_
---------- --------- ---------- ---------- --------
    12340 01-MAR-97        550        480 05-FEB-02
    12341 03-MAR-97        530        580 05-FEB-02
    12342 13-FEB-97        660        520 05-FEB-02
    12343 03-FEB-97        530        420 16-JAN-02
```

Section E: Arithmetic Expressions and Functions

Overview

Purpose

Oracle supports arithmetic expressions in SQL statements to perform various calculations on numbers:

- + Add
- - Subtract
- * Multiply
- / Divide
- ( ) Override Precedence

In addition, Oracle supports a wide range of built-in functions to manipulate data.

Objectives

Upon completion of this section, each attendee will be able to:

- Use arithmetic expressions to perform calculations on data
- Use the following within SQL statements:
  - Numeric functions
  - Character functions
  - Date functions
  - Conversion functions
  - Group functions

In this section

These topics are covered in this section.

Arithmetic Expressions

Purpose

Arithmetic expressions allow you to perform calculations on data.

Example 1

```
SQL> SELECT sat_verbal, sat_verbal + 100
       FROM swrtest;

SAT_VERBAL     SAT_VERBAL+100
----------     --------------
550            650
530            630
660            760
590            690
530            630
370            470
590            690
630            730
520            620
```

Example 2

```
SQL>  SELECT   sat_verbal, sat_math,
              sat_verbal + sat_math total
        FROM  swrtest;

SAT_VERBAL     SAT_MATH   TOTAL
----------     ---------  ---------
550            480        1030
530            580        1110
660            520        1180
590            610        1200
530            420         950
370            420         790
590            620        1210
630            590        1220
520            460         980
```

Other operators

Similarly, you can use the subtraction (-), division (/), and multiplication (*) arithmetic operators.

Order of evaluation

Expression evaluation

The Relational Database Management System evaluates each arithmetic expression. The results are then combined in the order determined by the following precedence:

- * Multiplication
- / Division
- + Addition
- - Subtraction

Overriding precedence

To override the precedence rules, use parentheses. Oracle evaluates expressions within parentheses first.

Examples

The following examples retrieve the balance from the student account balance table. According to the precedence rules, division will occur before addition, so the following examples yield different results.

Example 1

In the first example, 100 is divided by 12 and then added to the balance.

```
SQL> SELECT    balance, balance + 100 /12
       FROM    twraccd;

BALANCE        BALANCE+100/12
---------      --------------
1500.5         1508.8333
 300.2         308.53333
  -700         -691.6667
  1100         1108.3333
   500         508.33333
 -1000         -991.6667
  1200         1208.3333
    50         58.333333
   800         808.33333
   800         808.33333
 -1100         -1091.667
```

Continued on the next page

Order of evaluation, Continued

Example 2

In the second example, 100 is added to the balance, and then the total is divided by 12.

```
SQL> SELECT balance, (balance + 100)/12
       FROM twraccd;

   BALANCE (BALANCE+100)/12
---------- ----------------
    1500.5           133.375
     300.2             33.35
      -700               -50
      1100               100
       500                50
     -1000               -75
      1200         108.33333
        50              12.5
       800                75
       800                75
     -1100         -83.33333
```

Numeric functions

ABS(n)

Returns the absolute value of *n*.

```
SQL> SELECT ABS(-32) FROM DUAL;

 ABS(-32)
---------
       32
```

CEIL(n)

Returns smallest integer greater than or equal to *n*.

```
SQL> SELECT CEIL(12.8) FROM DUAL;

CEIL(12.8)
----------
        13

SQL> SELECT CEIL(-16.2) FROM DUAL;

CEIL(-16.2)
----------
       -16
```

FLOOR(n)

Returns largest integer equal to or less than *n.*

```
SQL> SELECT FLOOR(12.8) FROM DUAL;

FLOOR(12.8)
-----------
         12

SQL> SELECT FLOOR(-17.5) FROM DUAL;

FLOOR(-17.5)
------------
         -18
```

MOD(m, n)

Returns remainder of m divided by *n*. If *n* is negative and greater than *m*, *m* is returned.

```
SQL> SELECT MOD(5,2) FROM DUAL;

 MOD(5,2)
---------
        1
```

Continued on the next page

Numeric functions, Continued

POWER(m, n) Returns *m* raised to the *n*th power. *n* must be an integer; if not, an error will be returned.

```
SQL> SELECT POWER(10,2) FROM DUAL;

POWER(10,2)
-----------
        100

SQL> SELECT POWER(-10,3) FROM DUAL;

POWER(-10,3)
------------
       -1000
```

ROUND(n [,m]) Returns *n* rounded to *m* places right of the decimal point; if *m* is omitted, then *n* is rounded to the nearest whole number.

```
SQL> SELECT ROUND(15.67,1) FROM DUAL;

ROUND(15.67,1)
--------------
          15.7

SQL> SELECT ROUND(152,-1) FROM DUAL;

ROUND(152,-1)
-------------
          150
```

Continued on the next page

Numeric functions, Continued

SIGN(n)

If *n* < 0, the function returns -1; if n = 0, the function returns 0; if n > 0, then the function returns 1.

```
SQL> SELECT SIGN(-15) FROM DUAL;

SIGN(-15)
---------
       -1

SQL> SELECT SIGN(15) FROM DUAL;

 SIGN(15)
---------
        1

SQL> SELECT SIGN(0) FROM DUAL;

  SIGN(0)
---------
        0
```

TRUNC(n [,m])

Returns *n* truncated to *m* decimal places. If *m* is omitted, then the decimal is removed completely. *m* can be negative to truncate *m* digits left of the decimal point.

```
SQL> SELECT TRUNC(16.99,1) FROM DUAL;

TRUNC(16.99,1)
--------------
          16.9

SQL> SELECT TRUNC(16.99,-1) FROM DUAL;

TRUNC(16.99,-1)
---------------
             10
```

Character functions

ASCII(char)

Returns the ASCII value for the given character.

```
SQL> SELECT ASCII('A') FROM DUAL;

ASCII('A')
----------
        65
```

CHR(n)

Returns the character having ASCII value *n*.

```
SQL> SELECT CHR(65) FROM DUAL;

C
-
A
```

CONCAT(m, n) or ||' '||

Merges together two fields specified by *m* and *n*.

```
SQL> SELECT CONCAT(first_name, last_name) "Name"
       FROM swriden;
Name
----------------------------------------
JulieBrown
JulieBrown
RobertSmith
PeterJohnson
SandyJones
...
```

To merge two or more fields, use the pipes:

```
SQL>  SELECT   last_name||', '||first_name "Name"
        FROM   swriden;

Name
----------------------------------------
Brown, Julie
Brown, Julie
Smith, Robert
Johnson, Peter
Jones, Sandy
...
```

Continued on the next page

Character functions, Continued

INITCAP(char) Returns *char* with the first letter of each word in uppercase and all other letters in lowercase.

```
SQL> SELECT INITCAP('BASEBALL GAME') FROM DUAL;

INITCAP(
--------
Baseball Game
```

LOWER(char) Returns *char*, with all letters forced to lowercase.

```
SQL> SELECT LOWER('BASEBALL GAME')
     FROM DUAL;

LOWER('B
--------
baseball game
```

UPPER(char) Returns char, with all letters forced to uppercase.

```
SQL> SELECT UPPER('big letters') "Upper Case"
       FROM DUAL;

Upper Case
-----------
BIG LETTERS


SQL> SELECT last_name, first_name
       FROM swriden
      WHERE UPPER(last_name) = 'WHITE';

LAST_NAME                 FIRST_NAME
------------------------- ---------------
White                     Nancy
```

Continued on the next page

Character functions, Continued

LPAD(char1, n [,char2])

Returns char1, left-padded to length *n* with the sequence of characters in char2; char2 defaults to blanks.

```
SQL> SELECT LPAD(pidm,8,0) "PIDM" FROM swriden;

PIDM
--------
00012340
00012340
00012341
00012342
00012343
...
```

RPAD(char1, n [,char2])

Returns char1, right-padded to length *n* with sequence of characters in char2; if char2 is omitted, right-pad with blanks.

```
SQL> SELECT RPAD(last_name,20,'.')||id "ID Listing"
       FROM swriden
      WHERE change_ind IS NULL;

ID Listing
------------------------------
Brown...............157834585
Smith...............5829934
Johnson.............3539543
Jones-Erickson......145672112
Erickson............692568211
Erickson............578549991
White...............543853339
Marx................543853339
Clifford............543853339
Serum...............543853339
```

LTRIM(char [, set])

Removes characters from the left of char, with initial characters removed up to the first character not in the set; set defaults to ' ', which removes extra blanks.

```
SQL> SELECT LTRIM('123ABC123','123')
       FROM DUAL;

LTRIM(
------
ABC123
```

Continued on the next page

Character functions, Continued

RTRIM(char [, set])

Removes characters from the right of char, with initial characters removed up to the first character not in the set; defaults to ' ', which removes extra blanks.

```
SQL> SELECT RTRIM('123ABC123','123') FROM DUAL;

RTRIM(
------
123ABC
```

SUBSTR(char, m [,n])

Returns a portion of *char*, beginning at character *m, n* characters long (if *n* is omitted, to the end of *char*).

```
SQL> SELECT    SUBSTR('SunGard SCT',1, 7) "Substring"
       FROM    DUAL;

Substring
-------------------------
SunGard
```

LENGTH(char)

Returns the length of char.

```
SQL> SELECT LENGTH('abc') FROM DUAL;

LENGTH('ABC')
-------------
            3
```

Note: If the evaluated field is null, then no value (null) will be returned.

SOUNDEX (char)

Returns character string containing the phonetic representation of char.

```
SQL> SELECT last_name
       FROM swriden
      WHERE SOUNDEX(last_name) = SOUNDEX('SMYTHE');

LAST_NAME
------------------------
Smith
```

Date functions

ADD_MONTHS (d, n)

Returns the date *d* plus *n* months. Can be negative to subtract months.

```
SQL> SELECT ADD_MONTHS('10-DEC-97',2) FROM DUAL;

ADD_MONTH
---------
10-FEB-98
```

LAST_DAY(d)

Returns the date of the last day of the month that contains *d*.

```
SQL> SELECT LAST_DAY('15-JAN-97') FROM DUAL;

LAST_DAY(
---------
31-JAN-97
```

MONTHS_ BETWEEN (d, e)

Returns the number of months between dates *d* and *e*. If *d* is less than *e*, then the result is negative.

```
SQL> SELECT MONTHS_BETWEEN('11-OCT-96','01-JAN-96')
     "Months"
     FROM DUAL;

   Months
---------
9.3225806
```

NEXT_DAY (d, char)

Returns the date of the first day of the week named by *char* that is later than *d*.

```
SQL> SELECT    NEXT_DAY('05-FEB-97','FRIDAY')
                 "Pay Date"
     FROM      DUAL;

Pay Date
---------
07-FEB-97
```

Continued on the next page

Date functions, Continued

TRUNC (d, [,fmt])

Returns *d* with the time portion of the day truncated to the unit specified by the format model *fmt*. See Section S for a complete list of format models.

```
SQL> SELECT    TRUNC(TO_DATE('05-OCT-97'),'YEAR')
                 "Date"
       FROM    DUAL;

Date
---------
01-JAN-97
```

ROUND (d [,fmt])

Returns *d* with the time portion of the day truncated to the unit specified by the format model *fmt*. See Section S for a complete list of format models.

```
SQL> SELECT    ROUND(TO_DATE('05-OCT-97'),'YEAR')
                 "Date"
       FROM    DUAL;

Date
---------
01-JAN-98
```

Format Models

Date format element models

Date format element models for TO_CHAR and TO_DATE

This table lists the date format elements. You can use any combination of these elements as the *fmt* argument of the TO_CHAR and TO_DATE functions. *Fmt* defaults to the default DATE format, 'DD-MON-YY'.

Format element	Value returned
SCC or CC	Century; 'S' prefixes BC date with '-'
YYYY or SYYYY	Year; 'S' prefixes BC date with '-'
YYY or YY or Y	Last 3, 2, or 1 digit(s) of year. Century defaults to current
RR	Last 2 digits of year. 50 - 99 = 20th century, 00 - 49 =21st century
BC or AD	BC/AD indicator
B.C. or A.D.	BC/AD indicators with periods
Q	Quarter of year (1, 2, 3, 4)
MM	Month of year (01 - 12)
RM	Roman Numeral month (I…XII)
MONTH	Name of month padded with blanks to nine characters
MON	Name of month abbreviated (JAN, FEB, etc.)
WW or W	Week of year (1 - 52) or month(1 - 5)
DDD or DD or D	Day of year (1-366) or month (1 - 31) or week (1 - 7)
DAY	Name of day, blank-padded to 9 characters
DY	Name of day, 3 letter abbreviation
J	Julian day (days since December 31, 4713 BC)
AM or PM	Meridian indicator
A.M. or P.M.	Meridian indicator with periods
HH or HH12	Hour of day (1 - 12)
HH24	Hour of day (1 - 23)
MI	Minute (0 - 59)
SS or SSSSS	Seconds (0 - 59) or seconds past midnight (0 - 86399)
- / , . ; :	Punctuation is reproduced in the result

Continued on the next page

Format Models, Continued

Date format element models (cont.)

Format element	Value returned
"…text…"	Quoted string is reproduced in the result
Date format prefixes and suffixes	
FM	"Fill mode." Suppresses blank padding when prefixed to MONTH or DAY.
FX	"Format exact." Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model in the TO_DATE function.

Suffixes

You can add these suffixes to date format elements:

- **TH** Ordinal number ("DDTH" for "4TH")
- **SP** Spelled-out number ("DDSP" for "FOUR")
- **SPTH** and **THSP** Spelled-out ordinal number ("DDSPTH" for "FOURTH")

Date format case control

The following strings specify output in uppercase, initial caps, or lower case.

Uppercase	Initial caps	Lower case
DAY	Day	day
DY	Dy	dy
MONTH	Month	month
YEAR	Year	year
AM	Am	am
PM	Pm	pm
A.M.	A.m.	a.m.
P.M.	P.m.	p.m.

Continued on the next page

Format Models, Continued

Number format models

Number format models for TO_CHAR

Format element	Example	Function
9	'9999'	Number of "9"s determines length of returned character
0	'0999'	Prefixes value with leading zeroes
$	'$9999'	Prefixes value with a dollar sign
B	'B9999'	Returns zero values as blanks, instead of "0"
MI	'9999MI	Returns "-" after negative values
S	'S9999'	Returns "+" for positive values and "-" for negative values
PR	'999PR'	Returns negative values in <angle brackets>
D	99D99	Returns the decimal character
G	9G999	Returns the group separator
L	L999	Returns the local currency symbol
C	C999	Returns the international currency symbol
,	'9,999'	Returns comma in this position
.	'99.99'	Returns period in this position
V	'999V99'	Multiplies value by 10^n where n is "9"s after V
EEEE	'9.99EEEE'	Returns value in scientific notation
RM or rn	RN	Upper or lowercase Roman numerals
DATE	'DATE'	Returns value converted from Julian date

Conversion functions

TRANSLATE (char, from, to)

Returns *char* with all occurrences of each character in *from* replaced by its corresponding character in *to*.

```
SQL> SELECT TRANSLATE('PERRY','P','J') FROM DUAL;

TRANS
-----
JERRY

SQL> SELECT    TRANSLATE('ABC123ABC123','B3','X0')
               "Translate"
       FROM    DUAL;

Translate
-----------
AXC120AXC120
```

NVL (expr1, expr2)

Fields in a database which contain NULL values actually contain no value. Because NULL in a field can give unexpected results, it is best to convert the NULL into another value such as a zero for a number field or a space for a character field.

```
SQL> SELECT    NVL(sat_math,0), NVL(sat_verbal,0),
              NVL(sat_math,0) + NVL(sat_verbal,0)
              "Tot Score"
       FROM    swrtest;

NVL(SAT_MATH,0) NVL(SAT_VERBAL,0)  Tot Score
--------------- -----------------  ---------
            480                550       1030
            580                530       1110
            520                660       1180
            610                590       1200
            420                530        950
            420                370        790
            620                590       1210
            590                630       1220
            460                520        980
              0                520        520
```

Continued on the next page

Conversion functions, Continued

DECODE(expr, search1, result1, [search2, result2,]... [default])

The DECODE function converts the retrieved value from the database (search value)of the expression to a value specified (result). If no match is found, the DECODE function returns the default value. If no default value is specified, then the default will return a null.

```
SQL> SELECT    pidm,
     DECODE    (sex, 'F', 'Female', 'M', 'Male',
               'Unknown') "Gender"
       FROM    swbpers;

PIDM     Gender
-------  -------
12340    Female
12341    Male
12343    Female
12344    Male
12345    Unknown
```

DECODE can be used to perform a calculation. Consider a table which stores an amount as an absolute value and an indicator to specify if the refund is a negative quantity.

```
SQL> SELECT    DECODE(refund, 'Y', amount * -1,
               amount)
       FROM    ...
```

Continued on the next page

Conversion functions, Continued

INSTR (char1, char2 [, n [,m]])

Searches *char1* beginning with its *n*th character for the *m*th occurrence of *char2* and returns the position of the character in *char1* that is the first character of this occurrence.

```
SQL> SELECT INSTR('MISSISSIPPI','SS') "STRING"
       FROM DUAL;

   STRING
---------
         3

SQL> SELECT INSTR('MISSISSIPPI','SS',5) "STRING"
       FROM DUAL;

   STRING
---------
         6
```

INSTR and character strings

Use INSTR to parse a character string. In the view swvtele, the name column is created by the concatenation of last_name, a comma, a space, first name, a space, and mi (middle initial). The name column in this view would be parsed as follows:

```
SQL> SELECT    name "Whole Name",
               SUBSTR (name,1, instr(name,',') -1)
               "Last Name"
       FROM    swvtele;

Whole Name                Last Name
------------------------- ---------------
Brown, Julie K            Brown
Smith, Robert E           Smith
Johnson, Peter S          Johnson
Jones-Erickson, Sandy J   Jones-Erickson
Erickson, Ralph L         Erickson
```

Continued on the next page

Conversion functions, Continued

REPLACE (char, search_string [, replace_string])

Returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted, all occurrences of *search_string* are removed.

```
SQL> SELECT     REPLACE('100 Lakeshore Drive # 3',
                '#', 'No.') "Changes"
       FROM     DUAL;

Changes
-------------------------
100 Lakeshore Drive No. 3
```

TO_DATE (char [,fmt])

Converts a character value to a date value. If the *fmt* clause is omitted, then the character value must have the default format of 'DD-MON-YY'. When the format is not in the default form, you must explicitly tell Oracle the format. For a complete set of format models, see Section S.

```
SQL> INSERT INTO swvterm
       VALUES (20008,'Fall Semester 2000',
              TO_DATE('010197', 'DDMMYY'));

SQL> INSERT INTO swvterm
       VALUES (200101, 'Spring Semester 2001',
              TO_DATE('01-JAN-1997','DD-MON-YYYY'));
```

Continued on the next page

Conversion functions, Continued

TO_CHAR (d, [,fmt])

Converts number and dates into a character. See Section S for a complete list of format models.

```
SQL> SELECT    TO_CHAR(SYSDATE,'DD-MON-YYYY') "DATE"
       FROM    DUAL;

DATE
---------------------------------
20-MAY-1997


SQL> SELECT    TO_CHAR(SYSDATE,'FMMonth DD, YYYY')
                "DATE"
       FROM    DUAL;

DATE
------------
May 20, 1997


SQL> SELECT    TO_CHAR(SYSDATE,'DD-MON-YY
                 HH24:HH:SS') "DATE"
       FROM    DUAL;

DATE
------------------
20-MAY-97 14:02:22


SQL> SELECT    TO_CHAR(SYSDATE,'FMDay, FMMonth
                 FMDdSPTh, Syear') "DATE"
       FROM    DUAL;

DATE
--------------------------------------------------
Tuesday, May Twentieth, Nineteen Ninety-Seven
```

Note: Dates that have been converted using TO_CHAR are no longer sorted by date, but by alphanumeric order. If you are forced to use a converted date in a GROUP BY clause, it is best to convert the date to the YYYYMMDDHH24MISS format.

Continued on the next page

Conversion functions, Continued

RR: Changing Millennia

With the arrival of the 21st century, the need to refer to both the 20th and 21st centuries is necessary. The RR date format will change the pivot so that two-digit years from 50 - 99 refer to the 20th century; years from 00 - 49 refer to the 21st century. If the date is retrieved from a field where the date was inserted as a four-digit year, then changing the pivot year is not necessary.

```
SQL> SELECT    TO_CHAR(TO_DATE('01-JAN-97',
               'DD-MON-RR'),'DD-MON-YYYY') "DATE"
       FROM    DUAL;

DATE
----------------
01-JAN-1997


SQL> SELECT    TO_CHAR(TO_DATE('01-JAN-20',
               'DD-MON-RR'),'DD-MON-YYYY') "DATE"
       FROM    DUAL;

DATE
--------------------
01-JAN-2020
```

TO_NUMBER (char)

Converts *char* to a numeric datatype.

```
SQL> SELECT    TO_NUMBER('001') + TO_NUMBER('002')
               "TOTAL"
     FROM      DUAL;

TOTAL
---------
3
```

Group functions

Group functions

Group functions are used to obtain summary information about groups of rows. All group functions, except COUNT (*), ignore null values. Use the NVL function in the argument to substitute a value of null in a group function.

AVG(n)

Returns the average value of *n*.

```
SQL> SELECT AVG(gpa) FROM swrregs;

AVG(GPA)
---------
2.6052632

SQL> SELECT AVG(NVL(gpa,0)) FROM swrregs;

AVG(NVL(GPA,0))
---------------
      1.9038462
```

COUNT ({ * | expr })

Returns the number of rows in a query. If specifying a count on a particular column, the null values are not included.

```
SQL> SELECT COUNT(*) FROM swrregs;

 COUNT(*)
---------
       26

SQL> SELECT COUNT(gpa) FROM swrregs;

COUNT(GPA)
----------
        19
```

Continued on the next page

Group functions, Continued

MAX(expr) Returns the maximum value of expr.

```
SQL> SELECT MAX(gpa) FROM swrregs;

MAX(GPA)
---------
          4
```

MIN(expr) Returns the minimum value of expr.

```
SQL> SELECT MIN(gpa) FROM swrregs;

 MIN(GPA)
---------
         0
```

SUM(n) Returns sum of values of *n*.

```
SQL> SELECT SUM(gpa) FROM SWRREGS;

 SUM(GPA)
---------
     49.5
```

Self Check

Directions Use the information you have learned in this workbook to complete this self check activity.

Exercise 1 In the **SWRREGS** table, what is the average GPA of all the classes that PIDM 12342 took?

Exercise 2 How many records does PIDM 12343 have in the **SWRIDEN** table?

Exercise 3 Select the PIDM and the combined score of the SAT verbal and math for each record from the **SWRTEST** table.

Continued on the next page

Self Check, Continued

Exercise 4

Return the first name concatenated with the last name from the **SWRIDEN** table. Return only rows where the uppercase value of the first name is 'PETER'.

Exercise 5

What is the lowest and highest SAT verbal scores for students who took the test in March or April 1997 using **SWRTEST**?

Exercise 6

Retrieve the PIDM and age (whole number) from **SWBPERS**. Use the birth date and current system date to obtain the age.

Self Check – Answer Key

Exercise 1

In the **SWRREGS** table, what is the average GPA of all the classes that PIDM 12342 took?

```
SQL>  SELECT AVG(gpa)
        FROM swrregs
       WHERE pidm = 12342;

  AVG(GPA)
 ---------
    2.8625
```

Exercise 2

How many records does PIDM 12343 have in the **SWRIDEN** table?

```
SQL>  SELECT COUNT(*)
        FROM swriden
       WHERE pidm = 12343;

  COUNT(*)
 ---------
         2
```

Exercise 3

Select the PIDM and the combined score of the SAT verbal and math for each record from the **SWRTEST** table.

```
SQL> SELECT pidm, sat_verbal + sat_math
        FROM swrtest;

    PIDM SAT_VERBAL+SAT_MATH
 ------- ------------------
   12340               1030
   12341               1110
   12342               1180
   12341               1200
   12343                950
 ... 9 rows selected
```

Continued on the next page

Self Check – Answer Key, Continued

Exercise 4

Return the first name concatenated with the last name from the **SWRIDEN** table. Return only rows where the uppercase value of the first name is 'PETER'.

```
SQL> SELECT first_name||' '||last_name
       FROM swriden
      WHERE UPPER (first_name) = 'PETER';

 FIRST_NAME||''||LAST_NAME
 ----------------------------------------
 Peter Johnson
```

Exercise 5

What is the lowest and highest SAT verbal scores for students who took the test in March or April 1997 using **SWRTEST**?

```
SQL>  SELECT MAX(sat_verbal), MIN(sat_verbal)
        FROM swrtest
       WHERE test_date BETWEEN '01-MAR-97' AND
             '30-APR-97';

 MAX(SAT_VERBAL) MIN(SAT_VERBAL)
 --------------- ---------------
             550             530
```

Exercise 6

Retrieve the PIDM and age (whole number) from **SWBPERS**. Use the birth date and current system date to obtain the age.

```
SQL>     SELECT pidm,
                TRUNC (MONTHS_BETWEEN(birth_date,
                 SYSDATE)/12) "AGE"
           FROM swbpers;

      PIDM        AGE
---------- ----------
     12340         71
     12341         68
     12343         71
     12344         71
     12345         51
```

Section F: Clauses

Overview

Purpose

Clauses can be added to a SELECT statement to add conditions to the returned data as with the WHERE clause. In addition, clauses may be used to summarize data and change the order in which the data is returned.

Objectives

This section will examine the following clauses:

- WHERE
- ORDER BY
- GROUP BY
- HAVING

In this section

These topics are covered in this section.

The WHERE clause

WHERE clause

As noted in Section D, the WHERE clause is comprised of one or more conditions added to a query or manipulation statements so that only certain records are selected or manipulated.

- **SELECT...**
- **FROM...**
- **WHERE...**

Without WHERE

Without the WHERE clause, all rows will be returned from the specified table:

```
SQL> SELECT * FROM SWBPERS;

PIDM      SSN       BIRTH_DAT MRTL_CODE SEX CONFID ACTIVITY_
--------- --------- --------- --------- --- ------ ---------
12340     555444412 02-AUG-73 S         F   Y      30-MAR-97
12341     682082678 12-NOV-70 M         M   N      09-MAY-97
12343     555444412 02-AUG-73 S         F   Y      04-MAY-97
12344     198767345 02-AUG-73 S         M   N      07-MAY-97
12345     555444412 05-JAN-75 M             N      06-MAY-97
```

With WHERE

In the example below, rows are selected based on the criteria of birth date:

```
SQL> SELECT * FROM SWBPERS WHERE BIRTH_DATE = '02-AUG-73';

PIDM      SSN       BIRTH_DAT MRTL_CODE SEX CONFID ACTIVITY_
--------- --------- --------- --------- --- ------ ---------
12340     555444412 02-AUG-73 S         F   Y      30-MAR-97
12343     555444412 02-AUG-73 S         F   Y      04-MAY-97
12344     198767345 02-AUG-73 S         M   N      07-MAY-97
```

ORDER BY

ORDER BY clause

The ORDER BY clause changes the order in which information is displayed.

Note: The order columns in the ORDER BY clause do not have to appear in the SELECT clause. Without specifying ascending or descending order, ascending is assumed.

- **SELECT...**
- **FROM...**
- **WHERE...**
- **ORDER BY...**

Example 1

```
SQL> SELECT  pidm, birth_date
       FROM  swbpers
   ORDER BY  birth_date;

     PIDM BIRTH_DAT
--------- ---------
    12341 12-NOV-70
    12340 02-AUG-73
    12343 02-AUG-73
    12344 02-AUG-73
    12345 05-JAN-75
```

Example 2

```
SQL> SELECT  pidm, birth_date
       FROM  swbpers
   ORDER BY  birth_date DESC;

     PIDM BIRTH_DAT
--------- ---------
    12345 05-JAN-75
    12340 02-AUG-73
    12343 02-AUG-73
    12344 02-AUG-73
    12341 12-NOV-70
```

Continued on the next page

ORDER BY, Continued

Example 3

```
SQL> SELECT  last_name, first_name
       FROM  swriden
   ORDER BY  last_name,first_name;

LAST_NAME            FIRST_NAME
-------------------- ------------
Brown                Julie
Brown                Julie
Erickson             Ralph
Erickson             Susan
Johnson              Peter
Jones                Sandy
Jones-Erickson       Sandy
Smith                Robert
White                Nancy
```

Ordering by position

Column positions

Rather than specifying column names, refer to the columns by their position in the SELECT statement.

```
SQL> SELECT  last_name, first_name
       FROM  swriden
   ORDER BY  1,2;

LAST_NAME            FIRST_NAME
-------------------- ------------
Brown                Julie
Brown                Julie
Erickson             Ralph
Erickson             Susan
Johnson              Peter
Jones                Sandy
Jones-Erickson       Sandy
Smith                Robert
White                Nancy
```

Note: Although ordering by position requires less programming effort, it should not be used for programs.

GROUP BY

GROUP BY clause

Use the GROUP BY clause to group selected rows and return a single row of summary information. Oracle collects each group of rows based on the values of the expression(s) specified in the GROUP BY clause.

Restrictions

If a SELECT statement contains the GROUP BY clause, the select list can only contain these types of expressions:

- Constants
- Group functions
- Expressions identical to those in the GROUP BY clause
- Expressions involving the above expressions that evaluate to the same value for all rows in a group

Example 1

```
SQL> SELECT       pidm, AVG(gpa)
       FROM       swrregs
   GROUP BY       pidm;

     PIDM  AVG(GPA)
--------- ---------
    12340     3.025
    12341     2.475
    12342      2.45
    12343 3.1333333
    12344         0
    12345       2.5
    12346 2.7333333
```

The SELECT statement contains both a column name and a group function. This would return an error without the GROUP BY clause, which references *pidm* and causes *AVG(gpa)* to average the rows associated with each *pidm*.

Example 2

```
SQL> SELECT term_code, pidm, SUM(amount)
       FROM twraccd
      GROUP BY term_code, pidm
      ORDER BY term_code, pidm;

    TERM_C  PIDM SUM(AMOUNT)
---------- ----- -----------
    199602 12344        1120
    199701 12340      2500.7
    199701 12341        1600
    199701 12343        1900
    199701 12344        2100
    199701 12345         300
    199701 12346        1900
    199702 12342         800
    199702 12344        1150
    199801 12342        1350
    199802 12341        1000
```

HAVING

HAVING clause A HAVING clause places a condition on the GROUP function.

Example 1

```
SQL> SELECT  PIDM, COUNT(*)
       FROM  SWRIDEN
   GROUP BY  PIDM
     HAVING  COUNT(*) > 1;

     PIDM  COUNT(*)
--------- ---------
    12340         2
    12343         2
```

Example 2

```
SQL> SELECT    pidm, AVG(gpa)
       FROM    swrregs
   GROUP BY    pidm
     HAVING    AVG(gpa) < 3.0;

     PIDM  AVG(GPA)
--------- ---------
    12341     2.475
    12342      2.45
    12344         0
    12345       2.5
    12346 2.7333333
```

Example 3

```
SQL>   SELECT  pidm, term_code, SUM(amount)
         FROM  twraccd
        WHERE  term_code >= '199701'
     GROUP BY  pidm, term_code
       HAVING  SUM(amount) > 500
     ORDER BY  pidm, term_code;

     PIDM TERM_C SUM(AMOUNT)
--------- ------ -----------
    12340 199701      2500.7
    12341 199701        1600
    12341 199802        1000
    12342 199702         800
    12342 199801        1350
    12343 199701        1900
    12344 199701        2100
    12344 199702        1150
    12346 199701        1900
```

NOTE: ORDER BY must be the last line in a WHERE clause.

Self Check

Directions

Use the information you have learned in this workbook to complete this self check activity.

Examine the data to find out if the students have been receiving, on the average, low marks for courses. Build the query step by step.

Exercise 1

Examine the Student Grades table (**SWRREGS**) using the DESC command. You will be referring to this table for the rest of the section.

Exercise 2

Find the average GPA for each course number. Group by the course number.

Continued on the next page

Self Check, Continued

Exercise 3

To easily locate the courses with particularly low averages, order your data by the average (lowest first).

Exercise 4

Reduce the list so that only courses with averages below 2.0 are returned using a **WHERE** clause. Did you receive an error? Why?

Continued on the next page

Self Check, Continued

Exercise 5

Try Exercise 4 again, but put the condition in a **HAVING** clause.

Exercise 6

Your institution has changed the testing format for courses 10001 through 10006 from consecutive terms of 199602 and 199701. Examine the effects of the format change. In order to do this, select the course number, term code, and average GPA for the above courses and terms using **SWRREGS**. Group and order by course number and term code.

According to the data, has the new test format had a positive or negative effect on the GPAs?

Continued on the next page

Self Check, Continued

Exercise 7

To ensure that there is enough data to make a valid conclusion, make sure at least 3 students have taken the course in a term for the row to be returned. Use a **HAVING** clause to restrict the data being returned.

Self Check – Answer Key

Exercise 1

Examine the Student Grades table (**SWRREGS**) using the DESC command. You will be referring to this table for the rest of the section.

```
SQL> DESC swrregs

 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 PIDM                                      NOT NULL NUMBER(8)
 TERM_CODE                                 NOT NULL VARCHAR2(6)
 CRN                                       NOT NULL NUMBER(5)
 GPA                                                NUMBER(4,2)
 ACTIVITY_DATE                             NOT NULL DATE
```

Exercise 2

Find the average GPA for each course number. Group by the course number.

```
SQL> SELECT    crn, AVG(gpa)
       FROM    swrregs
   GROUP BY    crn;

      CRN  AVG(GPA)
--------- ---------
    10001 3.0333333
    10002      3.06
    10003       2.2
    10004       3.6
    10005     2.825
    10006       2.9
    ... 22 rows selected
```

Continued on the next page

Exercise 3

To easily locate the courses with particularly low averages, order your data by the average (lowest first).

```
SQL>   SELECT  crn, AVG(gpa)
         FROM  swrregs
     GROUP BY  crn
     ORDER BY  AVG(gpa);

      CRN  AVG(GPA)
--------- ---------
    10016       1.1
    10023       1.2
    10011       1.6
    10007       2.1
    10003       2.2
    10018       2.4
    10013       2.6
    10005     2.825
    10006       2.9
    10001 3.0333333
    10002      3.06
    10015      3.15
    ...
```

Exercise 4

Reduce the list so that only courses with averages below 2.0 are returned using a **WHERE** clause. Did you receive an error? Why?

```
SQL> SELECT    crn, AVG(gpa)
       FROM    swrregs
      WHERE    AVG(gpa) < 2.0
   GROUP BY    crn order by AVG(gpa);


WHERE AVG(gpa) < 2.0
      *
ERROR at line 2:
ORA-00934: group function is not allowed here
```

Continued on the next page

Exercise 5

Try Exercise 4 again, but put the condition in a **HAVING** clause.

```
SQL> SELECT crn, AVG(gpa)
       FROM swrregs
      GROUP BY crn
     HAVING AVG(gpa) < 2.0
      ORDER BY AVG(gpa);

      CRN  AVG(GPA)
--------- ---------
    10016       1.1
    10023       1.2
    10011       1.6
```

Exercise 6

Your institution has changed the testing format for courses 10001 through 10006 from consecutive terms of 199602 and 199701. Examine the effects of the format change. In order to do this, select the course number, term code, and average GPA for the above courses and terms using **SWRREGS**. Group and order by course number and term code.

```
SQL>  SELECT  crn, term_code, AVG(gpa)
        FROM  swrregs
       WHERE  term_code in (199602, 199701)
         AND  crn BETWEEN 10001 AND 10006
    GROUP BY  crn, term_code
    ORDER BY  crn, term_code;

      CRN TERM_C  AVG(GPA)
--------- ------ ---------
    10001 199602 2.8333333
    10001 199701 3.2333333
    10002 199602       2.8
    10002 199701 3.2333333
    10004 199701       3.6
    10005 199602       2.5
    10005 199701      3.15
    10006 199602 2.6666667
    10006 199701 2.9666667
```

According to the data, has the new test format had a positive or negative effect on the GPAs?

The overall average GPAs have risen for each course. Therefore, we are going to assume that the test format has had a positive effect on the student GPAs.

Continued on the next page

Exercise 7

To ensure that there is enough data to make a valid conclusion, make sure at least 3 students have taken the course in a term for the row to be returned. Use a **HAVING** clause to restrict the data being returned.

```
SQL>  SELECT crn, term_code, AVG(gpa)
        FROM swrregs
       WHERE term_code in (199602, 199701)
         AND crn BETWEEN 10001 AND 10006
    GROUP BY crn, term_code
      HAVING COUNT(*) > 2
    ORDER BY crn, term_code;

      CRN TERM_C  AVG(GPA)
--------- ------ ---------
    10001 199602 2.8333333
    10001 199701 3.2333333
    10002 199701 3.2333333
    10006 199602 2.6666667
    10006 199701     3.075
```

Section G: Advanced Queries

Overview

Purpose

Up to this point, we have provided examples of queries that return rows from only a single table. However, on many occasions we are concerned with retrieving data from many tables within the same query. For example, the student name, address, personal information, accounting information, registration information, and test scores are all kept in separate tables.

This section looks at how to join this information, so that data from two or more tables are retrieved with one SQL statement.

Objectives

Upon completion of this section, each attendee will be able to perform the following:

- Create queries to retrieve data from more than one table
- Use the set operators of UNION, INTERSECT, and MINUS to combine two or more queries into one result
- Create subqueries to solve complex queries

In this section

These topics are covered in this section.

Joins

Joins

A join is a SELECT statement that combines rows from two or more tables and/or views. Oracle performs joins whenever multiple tables appear in the FROM clause of a SELECT statement.

Joins and WHERE clause

The optional WHERE clause determines how Oracle combines rows of the tables. If the optional WHERE clause is omitted, the result is called a Cartesian Product.

For instance, if there are 10 rows in the swriden table and 5 rows in the swbaddr table, the resulting join of these two tables without a WHERE condition would return 50 rows. This type of join is rarely useful.

Equi-join

Returns rows from two or more tables based on an equality condition.

```
SELECT   select_list
  FROM   table1 [,table2] [,table3] …
WHERE    table1.column = table2.column
 [AND    table1.column = table3.column] …
```

Example 1

```
SQL> SELECT       last_name||', '||first_name||' '||mi NAME,
                  street_line1||' '|| city||', '||stat_code||
                  ' '|| zip ADDRESS
       FROM       swriden, swbaddr
     WHERE        swriden.pidm = swbaddr.pidm
       AND        change_ind IS NULL;

NAME                      ADDRESS
------------------------- ----------------------------------------
Brown, Julie K            506 BROWN STREET WEST CHESTER, PA 19380
Smith, Robert E           210 PINE STREET SAN FRANCISCO, CA 94082
Johnson, Peter S          PO BOX 1035 BROWNVILLE, KY 67233
Jones-Erickson, Sandy J 23 MARKET STREET WEST CHESTER, PA 19382
Erickson, Ralph L         18 CHESTNUT ROAD NEW ORLEANS, LA 23456
```

Notice how each column in the WHERE clause is preceded by the table name that contains it. If a column name is ambiguous, the column must be preceded by the table name.

Continued on the next page

Joins, Continued

Outer joins

An outer join returns all the rows returned by an equi-join as well as those rows from one table that do not match any rows from the other table. The table that might not contain the matching data is appended with a '(+)' in the WHERE clause.

Example

```
SQL> SELECT       last_name||', '||first_name||' '||mi NAME,
                  street_line1||' '||city||', '||stat_code
                  ||' '|| zip ADDRESS
       FROM       swriden, swbaddr
      WHERE       swriden.pidm = swbaddr.pidm (+)
        AND       change_ind IS NULL;


NAME                      ADDRESS
------------------------  ---------------------------------------
Brown, Julie K            506 BROWN STREET WEST CHESTER PA 19380
Smith, Robert E           210 PINE STREET SAN FRANCISCO CA 94082
Johnson, Peter S          PO BOX 1035 BROWNVILLE KY 67233
Jones-Erickson, Sandy J   23 MARKET STREET WEST CHESTER PA 19382
Erickson, Ralph L         18 CHESTNUT ROAD NEW ORLEANS LA 23456
Erickson, Susan T
White, Nancy Carol
Marx, Joan Elizabeth
```

Continued on the next page

Joins, Continued

Self-joins

There may be cases when you will need to join a table to itself. This is especially helpful when finding duplicates. In order to join a table to itself you must use table aliases. Below is an example of a self-join.

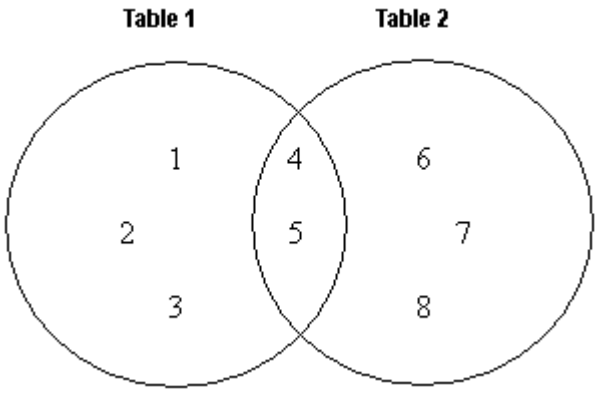
Example

Ensure that no duplicates exist in the validation table SWVTERM. Using a self-join, retrieve the records where the term coded match but the descriptions do not.

```
SQL> SELECT  A.term_code, A.description,
             B.term_code, B.description
       FROM  swvterm A, swvterm B
      WHERE  A.term_code = B.term_code
        AND  A.description <> B.description;

TERM_C   DESCRIPTION              TERM_C     DESCRIPTION
------   -------------------   --------     -------------------
199905   Spring Semester 1999      199905     Summer Semester 1999
199905   Summer Semester 1999      199905     Spring Semester 1999
```

Joins - Union, Union All, Intersect, Minus

Diagram



UNION

Returns all distinct rows for both select statements.

- 1, 2, 3, 4, 5, 6, 7, 8

UNION ALL

Returns all rows for both select statements, regardless of duplicates.

- 1, 2, 3, 4, 5, 4, 5, 6, 7, 8

INTERSECT

Returns only rows returned by both of the queries.

- 4, 5

MINUS

Returns all rows returned by the preceding query that were not present in the second.

- 1, 2, 3

Continued on the next page

Joins - Union, Union All, Intersect, Minus, Continued

Examples

For the following examples, refer to tables SWRIDEN1 AND SWRIDEN2. They have the following data:

SWRIDEN1

```
SQL> SELECT last_name, first_name
       FROM swriden1
      WHERE change_ind IS NULL;


LAST_NAME                 FIRST_NAME
------------------------- ------------
Jones-Erickson            Sandy
White                     Nancy
Marx                      Joan
```

SWRIDEN2

```
SQL> SELECT last_name, first_name
       FROM swriden2
      WHERE change_ind IS NULL;

LAST_NAME                  FIRST_NAME
-------------------------- ------------
Smith                      Robert
Johnson                    Peter
White                      Nancy
Marx                       Joan
```

Continued on the next page

Joins - Union, Union All, Intersect, Minus, Continued

Union & Union All

Union returns all distinct rows returned by both of the two queries. Union All returns all the rows of the two queries, including duplicates.

Union

```
SQL> SELECT last_name, first_name
       FROM swriden1
      WHERE change_ind IS NULL
      UNION
     SELECT last_name, first_name
       FROM swriden2
      WHERE change_ind IS NULL;


LAST_NAME                 FIRST_NAME
------------------------- -------------
Johnson                   Peter
Jones-Erickson            Sandy
Marx                      Joan
Smith                     Robert
White                     Nancy
```

Union All

```
SQL> SELECT last_name, first_name
       FROM swriden1
      WHERE change_ind IS NULL
      UNION ALL
     SELECT last_name, first_name
       FROM swriden2
      WHERE change_ind IS NULL;


LAST_NAME                 FIRST_NAME
------------------------- ---------------
Jones-Erickson            Sandy
White                     Nancy
Marx                      Joan
Marx                      Joan
Smith                     Robert
Johnson                   Peter
White                     Nancy
```

Continued on the next page

Joins - Union, Union All, Intersect, Minus, Continued

Intersect

Intersect returns only rows returned by both of the queries.

```
SQL>   SELECT last_name, first_name
         FROM swriden1
        WHERE change_ind IS NULL
    INTERSECT
       SELECT last_name, first_name
         FROM swriden2
        WHERE change_ind IS NULL;


LAST_NAME                 FIRST_NAME
------------------------- ---------------
Marx                      Joan
White                     Nancy
```

Minus

Minus returns all rows returned by the first query that are not present in the second.

```
SQL>   SELECT last_name, first_name
         FROM swriden1
        WHERE change_ind IS NULL
        MINUS
       SELECT last_name, first_name
         FROM swriden2
        WHERE change_ind IS NULL;

LAST_NAME        FIRST_NAME
---------------  ----------
Jones-Erickson   Sandy
```

Subqueries

Subquery

A subquery is a form of SELECT command that appears inside another SQL statement. A subquery is sometimes called a nested query. The statement containing a subquery is called the parent statement.

Example

```
SQL>  SELECT last_name||' '||first_name "NAME",
             description "COURSE", gpa "GPA"
        FROM swvcrse, swriden, swrregs
       WHERE swrregs.pidm = swriden.pidm
         AND swrregs.crn = swvcrse.crn
         AND gpa > =
            (SELECT  AVG(gpa)
              FROM  swrregs)
         AND change_ind IS NULL
    ORDER BY description, last_name, first_name, mi;

NAME                 COURSE                         GPA
-------------------- ------------------------ ---------
Brown Julie          Biology                          3
Brown Julie          Biology                        3.1
Brown Julie          Biology                        3.2
Jones-Erickson Sandy Calculus                         4
Erickson Susan       European History                 3
Erickson Susan       European History               3.9
Smith Robert         Photography                    3.1
White Nancy          Photography                    3.6
...
```

Continued on the next page

Subqueries, Continued

Limitations

The subquery in the previous example returns only one row for each row evaluated in the parent query. If the following statement were issued, without the AVG function, an error would occur.

```
SQL>  SELECT last_name||' '||first_name "NAME",
             description "COURSE", gpa "GPA"
        FROM swvcrse, swriden, swrregs
       WHERE swrregs.pidm = swriden.pidm
         AND swrregs.crn = swvcrse.crn
         AND gpa > =
               (SELECT gpa
                   FROM swrregs
                  WHERE gpa > 2)
         AND change_ind IS NULL
    ORDER BY description, last_name, first_name;

ERROR:
ORA-01010: invalid OCI operation

no rows selected
```

The SQL statement errors because the single-row subquery returns more than one row.

Subqueries returning multiple values

Multiple row comparisons

To evaluate comparisons that return more than a single row, use the following:

- ANY
- ALL
- IN
- EXISTS

ANY

The example below would calculate the average GPA for each course, and then select the people whose GPAs in a course are below any of the course averages.

```
SQL> SELECT   last_name, first_name, description
              "COURSE", gpa "GPA"
       FROM   swvcrse, swriden, swrregs
      WHERE   swrregs.pidm = swriden.pidm
        AND   swrregs.crn = swvcrse.crn
        AND   gpa < ANY  (SELECT AVG(gpa)
                          FROM swrregs
                       GROUP BY crn)
        AND    swriden.change_ind IS NULL;

  no rows selected
```

ALL

The example below would take the average GPA for each course, and then select the people whose GPAs in a course are below all of the course averages.

```
SQL> SELECT    last_name, first_name,
               description "COURSE", gpa "GPA"
       FROM    swvcrse, swriden, swrregs
      WHERE    swrregs.pidm = swriden.pidm
        AND    swrregs.crn = swvcrse.crn
        AND    gpa < ALL  (SELECT    AVG(gpa)
                           FROM    swrregs
                        GROUP BY    crn)
        AND     swriden.change_ind IS NULL;

no rows selected
```

Continued on the next page

Subqueries returning multiple values, Continued

IN

Use the IN operator to evaluate equality to any member of the test.

```
SQL>   SELECT last_name, first_name, detc_code,
              amount, balance
         FROM swriden, twraccd
        WHERE swriden.pidm = twraccd.pidm
          AND change_ind IS NULL
          AND twraccd.pidm IN (SELECT pidm
                                 FROM swrstdn
                                WHERE stdn_code = 'SS')
     ORDER BY last_name, first_name;

LAST_NAME    FIRST_NAME      DETC AMOUNT   BALANCE
-----------   --------------- ---- --------- ---------
Erickson      Ralph           TUIT       750       750
Erickson      Ralph           BOOK       400       400
Erickson      Ralph           LABS       120       120
Erickson      Ralph           MEAL       900       900
Erickson      Ralph           DORM      1000      1000
Erickson      Ralph           CASH       800      -800
Erickson      Ralph           CRED       400      -400
```

EXISTS

Evaluates to TRUE if the subquery returns a row.

```
SQL> SELECT  last_name, first_name
       FROM  swriden
      WHERE  change_ind IS NULL
        AND NOT EXISTS (SELECT 'X'
                          FROM swbaddr
                         WHERE  swbaddr.pidm=swriden.pidm);
```

Nested Subqueries

Nesting

Nesting is the act of putting several subqueries in serial:

```
SQL>      SELECT last_name, first_name
            FROM swriden
           WHERE change_ind IS NULL
             AND pidm IN   (SELECT pidm
                              FROM swbpers
                             WHERE mrtl_code = 'S'
                               AND pidm IN (SELECT pidm
                                              FROM twraccd
                                             WHERE term_code =
                                                   '199701'));

LAST_NAME            FIRST_NAME
-------------------- ---------------
Brown                Julie
Jones-Erickson       Sandy
Erickson             Ralph
```

Correlated Subqueries

Correlated subqueries

A correlated subquery is a SELECT statement inside the WHERE clause of a SQL statement which is correlated (or makes reference) to one or more columns in the enclosing SQL statement.

In the preceding subquery examples, each subquery was executed once, and the resulting value was used by the WHERE clause of the main query. You can also compose a subquery that is executed repeatedly, once for each candidate row considered for selection by the main query.

Correlated subqueries can also contain tables used by the main query. If this is the case, the main query should define an alias in order to make references.

Example

```
SQL> SELECT    last_name||' '|| first_name
               "NAME", description "COURSE",
               gpa "GPA"
       FROM    swvcrse, swriden, swrregs a
      WHERE    a.pidm = swriden.pidm
        AND    a.crn = swvcrse.crn
        AND    change_ind IS NULL
        AND    gpa <  (SELECT AVG (gpa)
                         FROM swrregs b
                        WHERE b.crn = a.crn);

NAME                COURSE               GPA
------------------  ---------------      ------
Brown Julie         Biology              2
Brown Julie         Speech               2.8
Smith Robert        Photography          3.1
Johnson Peter       Zoology              2.3
```

Continued on the next page

Correlated Subqueries, Continued

Find most recent row

If the swrstdn (student standing) table contains both current and historical data, then a correlated subquery can find the most recent row:

```
SQL> SELECT    first_name, last_name,
               description "COURSE", gpa "GPA"
         FROM  swriden, swrstdn A, swvstdn
        WHERE  swriden.pidm = A.pidm
          AND  A.stdn_code = swvstdn.stdn_code
          AND  change_ind IS NULL
          AND   A.activity_date =
                              SELECT MAX (activity_date)
                                FROM swrstdn b
                               WHERE B.pidm = A.pidm;
```

Performance

Almost without exception, a correlated subquery will be faster than most any other type of subquery evaluation. Depending on many factors, joining many large tables can be impossible, at least from a performance standpoint. After joining about 3 or 4 large tables (250,000 rows each) performance may become reduced exponentially.

The answer to joining tables and performance (or not joining tables) is the correlated subquery. Correlated subqueries are most often used when a column evaluation in a WHERE clause is not necessary in the select_list.

Self Check

Directions

Use the information you have learned in this workbook to complete this self check activity.

Exercise 1

Select the ID and combined SAT scores from the **SWRIDEN** and **SWRTEST** tables, using an equi-join. Join by PIDM.

Exercise 2

Create a report that contains the same information as above (using the same tables), but also include students who have not taken the SAT test. Use an outer join.

Continued on the next page

Self Check, Continued

Exercise 3

Return the PIDM(s) of the students who are in the **SWRIDEN** table but not in the **SWRREGS** table, using the keyword **MINUS**.

Exercise 4

By joining more than two tables, return the first name concatenated with the last name, course number, and course description for every course each student has taken. (**SWRIDEN, SWVCRSE, SWRREGS)**

Self Check – Answer Key

Exercise 1

Select the ID and combined SAT scores from the **SWRIDEN** and **SWRTEST** tables, using an equi-join. Join by PIDM.

```
SQL> SELECT a.id, b.sat_verbal + b.sat_math
       FROM swriden a, swrtest b
      WHERE a.pidm = b.pidm
        AND a.change_ind IS NULL
   ORDER BY a.id,b.sat_verbal + b.sat_math;


ID          SAT_VERBAL+SAT_MATH
--------- --------------------
145672112                  950
157834585                 1030
3539543                   1180
543853339                  980
543853339                 1220
578549991                 1210
5829934                   1110
5829934                   1200
692568211                  790
9 rows selected
```

Exercise 2

Create a report that contains the same information as above (using the same tables), but also include students who have not taken the SAT test. Use an outer join.

```
SQL> SELECT a.id, b.sat_verbal + b.sat_math
       FROM swriden a, swrtest b
      WHERE a.pidm = b.pidm (+)
        AND a.change_ind IS NULL
   ORDER BY a.id,b.sat_verbal + b.sat_math;

ID          SAT_VERBAL+SAT_MATH
--------- --------------------
145672112                  950
157834585                 1030
3539543                   1180
543853339                  980
543853339                 1220
543853339
578549991                 1210
5829934                   1110
5829934                   1200
692568211                  790
```

Continued on the next page

Self Check – Answer Key, Continued

Exercise 3

Return the PIDM(s) of the students who are in the **SWRIDEN** table but not in the **SWRREGS** table, using the keyword **MINUS**.

```
SQL> SELECT pidm
       FROM swriden
      MINUS
     SELECT pidm
       FROM swrregs;


     PIDM
---------
    12347
```

Exercise 4

By joining more than two tables, return the concatenation of the first name, last name, course number, and course description for every course each student has taken. **(SWRIDEN, SWVCRSE, SWRREGS)**

```
SQL> SELECT    first_name||' '||last_name name,
               swrregs.term_code,
               swrregs.crn,
               swvcrse.description
         FROM  swriden, swvcrse, swrregs
        WHERE  change_ind is null
          AND  swriden.pidm = swrregs.pidm
          AND  swvcrse.crn = swrregs.crn
    ORDER BY   last_name, first_name;

NAME                 TERM_C        CRN DESCRIPTION
-------------------- ------ ---------- ---------------
Julie Brown          199701      10001 Writing
Julie Brown          199702      10004 Physics
Julie Brown          199701      10005 Biology
Julie Brown          199602      10008 Psychology
Julie Brown          199701      10009 Calculus
Julie Brown          199602      10015 Speech
Julie Brown          199702      10017 Mgmt Info Sys
Julie Brown          199701      10007 Philosophy
Julie Brown          199602      10005 Biology
Stephanie Clifford   199602      10001 Writing
Ralph Erickson       199701      10004 Physics

... 44 rows selected
```

Section H: Insert, Update and Delete

Overview

Purpose

Throughout the previous sections, we have discussed retrieving data from the database. This section covers data manipulation: inserting, removing, and updating data within a table. More commonly, end users perform these tasks through a different software application, such as an Oracle Forms application.

Through a Forms application (such as SCT Banner), a user can make changes to a few rows of data, but larger changes or mass updates require SQL*Plus. For example, what if 10,000 rows in the swriden table contained an invalid change indicator? An end user could query rows with the faulty indicator through a form and make the corrections. However, making corrections one at a time would certainly be tedious.

Objectives

Upon completion of this section, each attendee will be able to write statements which:

- Insert new records into a table
- Update existing records in a table
- Remove records from a table
- Control data manipulation through the use of transactions to save and undo changes to data

In this section

These topics are covered in this section.

Insert

Purpose

Add a row to a table or a view's base table.

```
INSERT INTO table [view]([column1] [,column2] … )
     VALUES (expr1 [,expr2] …);

SQL> INSERT INTO swvcrse (crn, description,
                          activity_date)
           VALUES   (10025, 'ENGLISH', SYSDATE);
```

Abbreviated version

If all the columns of a table are being inserted in a table, the columns can be omitted from the statement. The above statement might be abbreviated:

```
SQL> INSERT INTO swvcrse
          VALUES   (10025, 'ENGLISH', SYSDATE);
```

Note: Omitting column references is a great shortcut when doing hands-on manipulations, but should never be used in a stored procedure. If the definition of a table changes in the future, the stored statements without column references will become invalid.

Insert via subquery

Inserts can be accomplished using a subquery from another table. The same number of rows returned from the subquery will be inserted into the table.

In the following example, we want to insert students into the SWRSTDN table for those that have an average course GPA of 3.5 or higher (honor students, so they will be marked with 'HS').

```
SQL> INSERT INTO    swrstdn (pidm, stdn_code,
                    stdn_date, activity_date)
     SELECT    pidm,
               'HS',
               '01-JAN-97',
               SYSDATE
       FROM    swrregs
   GROUP BY    pidm
     HAVING   AVG(gpa)    >3.5;
```

Update

Purpose

Change existing column values within a table or in a view's base table.

```
UPDATE table [view]
   SET column = expr [,column = expr] […]
[WHERE condition ];

SQL> UPDATE swrtest
       SET sat_math = 490
     WHERE pidm = 12340
       AND test_date='01-MAR-97';
```

Subqueries

Subqueries may also be used in the update WHERE condition.

```
SQL>  UPDATE swrstdn
        SET stdn_code = 'HS'
      WHERE pidm in
           (  SELECT pidm
                FROM swrregs
              GROUP BY pidm
                HAVING AVG (gpa) > 3.5);
```

Delete

Delete

Deleting rows from a table is quite simple; in fact, it is too simple. Consider the following syntax:

```
DELETE   [FROM]     <table>
[WHERE   <column_name> condition];

SQL>      DELETE     FROM swrtest
           WHERE     pidm = 12341
             AND     sat_verbal = 530
             AND     sat_math = 580;
```

Deleting entire tables

What will occur if the delete statement is run without the optional WHERE clause, or replacing <table> with swriden?

```
SQL> DELETE swriden;

12 rows deleted.

SQL> SELECT * FROM swriden;

no rows selected
```

Transactions

What is a transaction?

A transaction is defined as a change to the database since the last COMMIT.

Commit

Makes permanent the changes made to the database. While working in SQL, changes can be viewed after a command is run on the database. For instance, if a person was deleted from the swriden table while in SQL*Plus, you could view these changes. This is called an implied commit. This does not mean that the change has been made permanent to the database.

To make the change permanent, use the COMMIT command.

```
SQL> DELETE    FROM  swriden
               WHERE last_name = 'JONES';

SQL> COMMIT;
```

Note: Oracle recommends that every transaction end explicitly with a COMMIT before disconnection from Oracle. If a program terminates abnormally, the last uncommitted transaction is rolled back. A normal exit from most Oracle applications causes the current transaction to be committed.

Rollback

Use ROLLBACK to undo work within the current transaction.

```
ROLLBACK [TO SAVEPOINT <savepoint>];

SQL> DELETE     FROM  swriden
                WHERE  last_name = 'JONES';

SQL> ROLLBACK;
```

Continued on the next page

Transactions, Continued

Savepoint

Identifies a point in the current transaction to which you can later roll back.

```
SAVEPOINT <savepoint>
```

```
SQL>      DELETE FROM      swriden
                 WHERE     last_name = 'JONES';

SQL>      SAVEPOINT sp1;

SQL>      DELETE FROM      swriden
                 WHERE     last_name = 'SMITH';

SQL>      SAVEPOINT sp2;

SQL>      ROLLBACK TO SAVEPOINT sp1;
```

Autocommit

SQL*Plus contains a SET command which can be altered in order to commit every transaction after every manipulation statement is issued.

To view the current setting of the AUTOCOMMIT option, type the following:

```
SQL> SHOW AUTOCOMMIT

autocommit OFF
```

To set the autocommit to commit after every insert, update, or delete, type the following:

```
SQL> SET AUTOCOMMIT ON
```

AUTOCOMMIT can also be set at login based on the profile settings of your login.sql.

Self Check

Directions

Use the information you have learned in this workbook to complete this self check activity.

Exercise 1

Insert a new student in the **SWRIDEN** table using your own name, PIDM 2045 and ID 432G. Do not use a middle name.

Exercise 2

Add a new student profile record in **SWBPERS** for the new student added in Exercise 1, using the following information:

- Activity Date — Current system date
- Social Security Number — 124-62-8747
- Birth Date — Unknown (leave null)
- Marital Code — Unknown (leave null)
- Sex — Female
- Confidential Indicator — Y

Note: check the description of the table for column size constraints.

Exercise 3

Create a savepoint named SP1.

Continued on the next page

Self Check, Continued

Exercise 4

Insert another row into the **SWRIDEN** table, but prompt the operator for each variable except for the activity date.

Exercise 5

Update the new student profile record created in step 2 so that the social security number is 635-56-1525 and the marital code is 'S' (**SWBPERS)**.

Exercise 6

Roll back to savepoint SP1.

Exercise 7

Commit your changes.

Continued on the next page

Self Check, Continued

Exercise 8 Delete the student profile record created in Exercise 2 **(SWPBERS)**.

Exercise 9 Delete the **SWRIDEN** record you created in Exercise 1.

Exercise 10 Commit your changes.

Self Check – Answer Key

Exercise 1

Insert a new student in the **SWRIDEN** table using your own name, PIDM 2045 and ID 432G. Do not use a middle name.

```
SQL>  INSERT INTO swriden (pidm, activity_date, id,
      last_name, first_name)
         VALUES  (2045, sysdate, '432G',
                  'My_Last_Name', 'My_First_Name');
```

Exercise 2

Add a new student profile record in **SWBPERS** for the new student added in step 1, using the following information:

- Activity Date — Current system date
- Social Security Number — 124-62-8747
- Birth Date — Unknown (leave null)
- Marital Code — Unknown (leave null)
- Sex — Female
- Confidential Indicator — Y

Note: check the description of the table for column size constraints.

```
SQL> INSERT INTO swbpers (pidm, activity_date,
                   ssn, sex, confid_ind)
         VALUES  (2045,sysdate,124628747,'F','Y');
```

or

```
SQL> INSERT INTO swbpers
        VALUES  (2045,sysdate, '124628747',
                 NULL, NULL,'F','Y');
```

Exercise 3

Create a savepoint named SP1.

```
SQL> SAVEPOINT SP1;
Savepoint created.
```

Exercise 4

Insert another row into the **SWRIDEN** table, but prompt the operator for each variable except for the activity date.

```
SQL>  INSERT INTO swriden (pidm, activity_date, id,
         last_name, first_name, mi, change_ind)
         VALUES (&Internal_ID, SYSDATE, '&ID',
              '&last_name', '&first_name',
              '&middle_name', '&change_ind');
```

Continued on the next page

Self Check – Answer Key, Continued

Exercise 5

Update the new student profile record created in step 2 so that the social security number is 635-56-1525 and the marital code is 'S' (**SWBPERS)**.

```
SQL> UPDATE SWBPERS
        SET ssn = '635561525', mrtl_code = 'S'
      WHERE pidm = 2045;
```

or

```
SQL> UPDATE SWBPERS
        SET ssn = 635561525
      WHERE pidm = 2045;

SQL> UPDATE SWBPERS
        SET mrtl_code = 'S'
      WHERE pidm = 2045;
```

Exercise 6

Roll back to savepoint SP1.

```
SQL> ROLLBACK TO SP1;
```

Exercise 7

Commit your changes.

```
COMMIT;
```

Exercise 8

Delete the student profile record created in Exercise 2 **(SWPBERS)**.

```
SQL> DELETE FROM swbpers
          WHERE pidm = 2045;

 1 row deleted.
```

Exercise 9

Delete the **SWRIDEN** record you created in Exercise 1.

```
SQL> DELETE FROM swriden
          WHERE pidm = 2045;
1 row deleted.
```

Exercise 10

Commit your changes.

```
 COMMIT;
```

Section I: Creating and Maintaining Database Objects

Overview

Purpose

In the previous sections, we have both retrieved and manipulated data from tables. A table is a database object within a schema. In this section, we will discuss how to create and maintain tables and get introduced to other schema objects that can be created.

Objectives

Upon completion of this section, each attendee will be able to create, maintain, and secure:

- Tables
- Views
- Sequences

In this section

These topics are covered in this section.

Schemas

Schemas

Schemas may contain the following types of objects:

- Tables
- Views
- Clusters+
- Database links+
- Stand-alone stored functions and procedures
- Indexes
- Packages
- Database triggers+
- Sequences
- Snapshots+
- Profiles+
- Roles+
- Rollback segments+
- Tablespaces+

\+ These objects are not discussed in this manual; refer to Oracle's SQL language reference manual

Data definition language commands

DDL commands

Data Definition Language (DDL) commands allow you to perform these tasks:

- Create, alter, and drop objects
- Grant and revoke privileges and roles
- Establish auditing options
- Add comments to the data dictionary

Implicit COMMITs

Oracle implicitly commits the current transaction before and after every Data Definition Language statement. Consideration to database manipulations should be considered before using DDL statements.

For a complete listing of DDL commands, see the Oracle SQL Language Reference Manual.

Creating a table

Naming a table or view

- Must begin with a letter, A-Z or a-z
- May contain letters, numerals, and the special character _ (underscore). The characters $ and # are also legal, but their use is discouraged
- Case-insensitive; e.g., grades, GRADES, and GrAdEs are all the same table
- May be up to 30 characters in length
- May not duplicate the name of another table or view under the same schema
- May not duplicate an Oracle reserved word

Name	Valid?
NATION	Yes
1CONTINENT	No; doesn't begin with a letter.
NORTH_AMERICA	Yes
UPDATE	No; Oracle reserved word.
TABLE1	Yes; but poor design for naming conventions.

Naming a column

Column names follow the same rules as those for table names. Columns with exactly the same name in separate tables can be ambiguous, however. For instance, joining two tables both containing "pidm" columns requires the use of TableName.ColumnName notation.

To avoid this ambiguity, SCT has appended the TableName to the front of each column name. For the purposes of demonstration and exercise convenience, this convention is not followed in the training materials. If the SWRIDEN table was to follow SCT standards, the columns would have appeared as such:

```
swriden_pidm
swriden_last_name
swriden_first_name
…
```

Not only does this naming convention remove the ambiguity of the column's table, but makes for very readable code. This is one of several conventions found within a SCT Banner table's design. As you begin to explore your system's tables and their structure, you will begin to recognize many other standards and conventions.

Note: Give your tables and columns meaningful names. You have up to 30 characters - why not use them?

Continued on the next page

Creating a table, Continued

Data type

- VARCHAR2(n)
  Variable length character string with a maximum length, *n*, of 2000 bytes.
- LONG
  Variable length character string containing up to 2 gigabytes, or $2^{31} - 1$ bytes.
- NUMBER(p,s)
  Numeric datatype having a precision *p* and scale *s*.
- DATE
  Valid dates range from Jan 1, 4712 BC to Dec 31, 4712 AD.

Syntax

```
CREATE TABLE [schema.] table
( { column datatype  [DEFAULT expr]
[column_constraint] | table_constraint }
[ , { column datatype  [DEFAULT expr]
[column_constraint] table_constraint}     ] … )
                    [ AS subquery ]
```

Example

```
SQL>  CREATE TABLE hobbies
   (hobbies_pidm          NUMBER(8) NOT NULL,
    hobbies_type_code     VARCHAR2(10),
    hobbies_desc          VARCHAR2(100),
    hobbies_how_long      NUMBER(3),
    hobbies_yearly_cost   NUMBER(11,2),
    hobbies_solo_group    VARCHAR2(2),
    hobbies_activity_date DATE);
```

Altering a table

Methods

A table can be altered via any of the following methods:

- Add a column
- Redefine a column (datatype, size, default value)
- Add an integrity constraint
- Enable, disable, or drop an integrity constraint or trigger

Syntax

```
ALTER TABLE [schema.] table
{ [ ADD ( { column_element | column_constraint}
  [, column_element | column_constraint}  ] …) ]
  [ MODIFY  ( column_element [, column_element]
...) ]
  [ DROP drop_clause ] …
  [ ENABLE enable_clause ] …
  [ DISABLE  disable_clause ] …
```

Adding and removing columns

ALTER TABLE

```
SQL> ALTER TABLE twraccd
               ADD (effective_date DATE);
```

Removing a column

With Oracle 8i, new functionality was added to remove a column:

```
SQL> ALTER TABLE <table_name> DROP COLUMN
<column_name>;
```

Example:

```
      ALTER TABLE twraccd
         DROP COLUMN effective_date;
```

Removing a column prior to Oracle 8i

Prior to Oracle 8i, there was no option in the ALTER TABLE statement to remove a column from a table.

A solution to this limitation may be found using the AS QUERY option:

- Create a new table using the AS QUERY option specifying all the columns except the column to be removed
- Drop the original table
- Rename the new table to the original name

Example

In the following example, we are removing the balance column from the account table, TWRACCD.

```
SQL> CREATE TABLE   new_twraccd
        AS SELECT   pidm, term_code, detc_code,
                    trans_type, bill_date, amount,
                   activity_date
              FROM  twraccd;

SQL>   DROP TABLE   twraccd;

SQL>       RENAME   new_twraccd
               TO   twraccd;
```

Redefining a column

```
SQL>     ALTER TABLE swriden
             MODIFY (last_name VARCHAR2 (30));
Table altered.
```

Constraints

Purpose

Constraints define the conditions under which data is valid.

The table_constraint syntax is a part of the table definition. An integrity constraint defined with this syntax can impose rules on any columns in the table. The table constraint syntax may appear in a CREATE TABLE or ALTER TABLE statement. The syntax can define any type of integrity constraint except a NOT NULL constraint.

The column_constraint syntax is part of a column definition. In most cases, an integrity constraint defined with this syntax can only impose rules on the column in which it is defined.

Column_constraint syntax that appears in a CREATE TABLE statement can define any type of integrity constraint. Column_constraint syntax that appears in an ALTER TABLE statement can only define or remove a NOT NULL constraint. To modify an integrity constraint, you must drop the constraint and redefine it.

NOT NULL constraint

The NOT NULL constraint specifies that a column cannot contain a null value. If you do not specify this constraint, the default is NULL.

```
SQL> ALTER TABLE hobbies
        MODIFY (hobbies_type_code  NOT NULL);
```

CHECK constraint

The CHECK constraint explicitly defines a condition. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to NULL).

Syntax: **`CONSTRAINT constraint_name CHECK (condition)`**

```
SQL> ALTER TABLE twraccd
  ADD CONSTRAINT check_trans_type
           CHECK (trans_type IN ('C', 'P'));

SQL> ALTER TABLE twraccd
  ADD CONSTRAINT check_amount
           CHECK (amount > 0 );
```

Continued on the next page

Constraints, Continued

PRIMARY KEY constraint

A PRIMARY KEY constraint designates a column or combination of columns as the table's primary key. To satisfy a PRIMARY KEY constraint, both of these conditions must be true:

- No primary key value can appear in more than one row in the table
- No column that is part of the primary key can contain a null

A table can have only one primary key.

```
SQL>     ALTER TABLE swvterm
   ADD CONSTRAINT PK_swvterm PRIMARY KEY
                  (term_code);
```

UNIQUE constraint

The UNIQUE constraint designates a column or combination of columns as a unique key. To satisfy the condition, no two rows in the table can have the same value for the unique key. You cannot designate the same column or combination of columns as both a unique key and the primary key. Although you can have only one primary key for a table, a table can have several unique keys.

```
SQL>     ALTER TABLE twvdetc
    ADD CONSTRAINT unq_detc_code UNIQUE
                   (detc_code);
```

Referential integrity constraints

Purpose

A referential integrity constraint designates a column or combination of columns as a foreign key, and establishes a relationship between that foreign key and a specified primary or unique key called the referenced key. In this relationship, the table containing the foreign key is called the child table, and the table containing the referenced key is called the parent table.

Conditions

To satisfy a referential integrity constraint, the following conditions must be met:

- The child and parent tables must be in the same database
- The value of the row's foreign key must appear as a referenced key value in one of the parent table's rows. The row in the child table is said to depend on the referenced key in the parent table

Keywords

A referential integrity constraint is defined in the child table. A referential integrity constraint definition can include any of these keywords:

- Foreign key
  Identifies the column or combination of columns in the child table that makes up the foreign key. Only use this keyword when defining a foreign key with a table constraint clause.
- References
  Identifies the parent table and the column or combination of columns that make up the referenced key. If you only identify the parent table and omit the column names, the foreign key automatically references to the primary key of the parent table. The referenced key columns must be of the same number and datatypes as the foreign key columns.
- On delete cascade
  Allows deletion of referenced key values in the parent table that have dependent rows in the child table. This causes Oracle to automatically delete dependent rows from the child table to maintain referential integrity. If you omit this option, Oracle forbids deletion of referenced key values in the parent table that have dependent table.

  WARNING: ***NEVER*** create a table using this integrity constraint unless it is directly applicable to your business rules and applications.

Continued on the next page

Referential integrity constraints, Continued

Defined constraints

Before defining a referential integrity constraint in the child table, the referenced UNIQUE or PRIMARY KEY constraint on the parent table must already be defined. Also, the parent table must be in your own schema or you must have REFERENCED privileges on the columns of the referenced key in the parent table. You cannot define a referential integrity constraint in a CREATE table statement that contains an AS clause. Instead, create the table without the constraint and add the constrain using the ALTER TABLE statement.

Note: You can define multiple foreign keys in a table. Also, a single column can be part of more that one foreign key.

```
SQL>    ALTER TABLE twraccd
   ADD CONSTRAINT FK1_twraccd_INV_swvterm_KEY
      FOREIGN KEY (term_code)
       REFERENCES swvterm (term_code);
```

Truncate

Purpose	TRUNCATE can be used to quickly remove all rows from a table. Removing all rows with the TRUNCATE command is faster than removing them with the DELETE command. No rollback information is created; thus the rows are permanently removed.
Syntax	**TRUNCATE TABLE [schema.] <table>;**
Example	SQL> TRUNCATE TABLE old_scores;

Creating views

Views

A view is a logical table that allows you to access data from other tables and views. A view contains no data itself. The tables upon which a view is based are called base tables.

Purpose

Views are used for these purposes:

- To provide an additional level of table security, by restricting access to a predetermined set of rows and/or columns of a base table
- To hide data complexity. A view may be used to act as one table when actually several tables are used to construct the results
- To present data from another perspective. For example, views provide a means of renaming columns without actually changing the base table's definition

Syntax

```
CREATE [OR REPLACE] [FORCE | NOFORCE]
  VIEW [schema.] view [ (alias  [, alias] … ) ]
  AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
```

- OR REPLACE
  Recreates the view if it already exists. Use this option to change the definition of an existing view without dropping, recreating, and regranting object privileges previously granted on it.
- FORCE | NOFORCE
  FORCE creates the view regardless of whether the view's base tables already exist within the owner's schema or the owner of the view has privileges to the base tables. NOFORCE creates this view only if the base tables exist within the owners schema or the owner of the view has privileges to the base tables.
- WITH CHECK OPTION
  Specifies that inserts and updates performed through the view must result in rows that the view query can select.

Continued on the next page

Creating views, Continued

Constraint

The name assigned to the CHECK OPTION constraint. If the constraint is omitted, then Oracle automatically assigns the constraint a name of the form, SYS_cn, where *n* is an integer that makes the constraint name unique within the database.

SWVTELE

In our training exercises, there exists one view, **swvtele**.

```
SQL> SELECT * FROM swvtele;

PIDM       NAME                 PHONE
---------  -------------------- ---------
12340      Brown, Julie         (610) 562-4789
12341      Smith, Robert        (215) 795-4323
12342      Johnson, Peter       (610) 562-4789
12343      Jones-Erickson, Sandy (610) 324-6734
12344      Erickson, Ralph      (850) 674-3213
```

Script

The view was created using the following script:

```
SQL> CREATE OR REPLACE VIEW swvtele
          (pidm, name, phone)
       AS SELECT    swriden.pidm, last_name||
                    ', '||first_name||' '||mi,
                    '('||phone_area||')'
                     ||SUBSTR(phone_number,1,3)
                  ||'-'|| SUBSTR(phone_number,4,4)
            FROM    swriden, swbaddr
           WHERE    swriden.pidm = swbaddr.pidm
             AND    change_ind IS NULL;
```

Continued on the next page

Creating views, Continued

Check option

```
SQL> CREATE OR REPLACE VIEW swvsusp
    AS SELECT   *
         FROM  swrstdn
        WHERE  stdn_code = 'SS'
         WITH CHECK OPTION;
```

With the check option on this view, users will get the following error when trying to change a student's standing:

```
SQL> UPDATE    swvsusp
        SET    stdn_code = 'GS';
*
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION
where-clause violation
```

Synonyms

Purpose

Synonyms are used for security and convenience. Creating a synonym for an object allows you to:

- Reference the object without specifying its owner (if the synonym is public)
- Provide another name for the object

Syntax

```
CREATE [PUBLIC] SYNONYM [schema.]synonym
                FOR [schema.] object
```

Example

```
SQL> CREATE PUBLIC SYNONYM standing
        FOR swrstdn;
```

Sequences

Purpose

A sequence is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary keys.

Syntax

```
CREATE SEQUENCE [schema.] sequence
    [INCREMENT BY integer]
    [START WITH integer]
    [MAXVALUE integer | NOMAXVALUE]
    [MINVALUE integer | NOMINVALUE]
    [CYCLE | NOCYCLE]
    [CACHE integer | NOCACHE]
    [ORDER | NOORDER]
```

- INCREMENTED BY
  Specifies the interval between sequence numbers. This value can be any positive or negative integer, but cannot be 0. If the increment is negative, the sequence descends. If the increment is positive, the sequence ascends. If omitted, the interval defaults to 1.
- MAX & MIN VALUE
  Specifies the sequence's minimum or maximum value. These integer values can have 28 or fewer characters. Ranges are 10^{27} to -10^{26}.
- CYCLE
  Specifies that a sequence will continue to generate values after reaching its MIN or MAX. The value generated is the MIN or MAX specification.
- CACHE
  Specifies how many values of the sequence Oracle preallocates and keeps in memory for faster access. The minimum value for this parameter is 2.
- ORDER
  Guarantees sequence numbers are generated in the order specified.
- START WITH
  Specifies the beginning value generated by the sequence. The MIN and MAX values must be less than or equal to START WITH.

Example

```
SQL>  CREATE SEQUENCE pidm_seq
            START WITH 13000;
```

Continued on the next page

Sequences, Continued

Accessing and incrementing

Once a sequence is created, you can access its value in SQL statements using the pseudo-columns CURRVAL and NEXTVAL. CURRVAL returns the current value of the sequence, NEXTVAL increments the sequence and returns the new value.

```
SQL> INSERT INTO swriden
VALUES(pidm_seq.nextval, '254915791',
  'McMahon', 'Stephen', 'J', NULL,
  SYSDATE);
```

Indexes

Purpose	Indexes are database structures that boost the performance of queries. Indexes are used in conjunction with table columns. An index associates each distinct value of a column with the rows in a table that contain that value. The column that has the index is called the key column.
Uniqueness	Some indexes can be used to enforce uniqueness among the values in a column. Such an index is called a unique index. If a unique index is created, no two rows in the table may contain the same value in the indexed column. A query that references an indexed column in its WHERE clause can use the index. When a query uses an index, Oracle searches the index for all the values that meet the condition specified by the WHERE clause. If the query selects only the indexed column, the query can read the indexed column values directly from the index rather than from the table.
ROWIDs	For each value, the index also identifies the locations, or ROWIDs, of rows in the table having that value. If the query selects data in addition to the indexed value, Oracle finds the rows in the table based on the ROWIDs. Searching by ROWID is the fastest way for Oracle to locate a single row.
When to use indexes	Indexes improve the performance of queries that select a small percentage of rows from a table. As a general guideline, you should use indexes for queries that select less than 10% or 15% of table rows.
Full table scans	If a query does not use an index, Oracle must perform a full table scan, reading all rows of a table sequentially. Oracle examines each row to determine whether it meets the criteria of the query's WHERE clause. Indexed queries can be considerably faster than finding the row with a full table scan; however, a query that selects more than 10% or 15% of a table's rows may be performed faster by a full table scan than by an indexed query.

Continued on the next page

Indexes, Continued

Choosing columns to index

- Index columns that are used frequently in WHERE clauses
- Index columns whose MAX and MIN values are selected frequently
- Index columns that are used frequently to join tables in SQL statements
- Index columns with high selectivity. Selectivity is high if few rows have the same value in the key column. Unique indexes are the most selective and the most effective in optimizing query performance
- Do not index columns with few distinct values. Such columns have low selectivity
- Do not index columns in small tables. If a table uses fewer than 5 data blocks, a full table scan may return rows faster than an indexed query. You can determine how many data blocks a table uses by examining the ROWIDs of the table's rows. For example, this query returns the number of blocks used by the swrtest table:

```
SQL> SELECT COUNT
 (DISTINCT(SUBSTR(ROWID,1,8)||SUBSTR(ROWID,15,4)))
       FROM swrtest;
```

- Do not index columns that are frequently modified using the UPDATE, INSERT, and DELETE statements. These statements not only update the rows in the table; they must also update the index as well.

Concatenated indexes

Purpose

An index can be made up of more than one column. Such an index is called a concatenated index.

Concatenated indexes are useful in providing selectivity. Sometimes two columns with low selectivity can be combined to produce a concatenated index with high selectivity. If all the selected columns are included in a concatenated index, the query can be satisfied entirely by an index search and avoid access to the table altogether. The columns that make up a concatenated index are referred to as the concatenated key.

SQL statements and concatenated indexes

Whether or not a SQL statement uses a concatenated index is determined by the column contained in the WHERE clause of the SQL statement and the order of the columns in the CREATE INDEX statement. A query can only use a concatenated index if it references a leading portion of the index in the WHERE clause. The leading portion of a concatenated index refers to the first column specified in the CREATE index statement.

Columns

An index can contain a maximum of 16 columns. You can create several indexes on different columns (or different combinations of columns) in the same table. Oracle imposes no limits on the number of indexes you can create on a single table.

Syntax

```
CREATE [UNIQUE] INDEX [schema.] index
ON [schema.] table (column, [column,] [column,] …)
```

Ordering columns in concatenated indexes

If only one column of the concatenated index is used frequently in WHERE clauses, place that column first in the create INDEX statement.
If more than one column is used frequently in WHERE clauses, place the most selective column first in the CREATE INDEX statement.

```
SQL> CREATE INDEX pidm_term_index
           ON twraccd (pidm, term_code);

SQL> CREATE UNIQUE INDEX swriden_key_index
                  ON swriden (pidm, id,
                              last_name, first_name,
                                mi, change_ind);
```

Security

Brief overview

System security is a complicated and lengthy topic and is beyond the scope of this course. However, there are a few minor aspects of security that should be discussed at this time in association to the creation of database objects.

GRANT privileges

Creating a table or view (or any database object), does not automatically grant other users the access to retrieve or manipulate data from those tables. The owner must explicitly GRANT privileges to other users within the database.

```
GRANT object_priv [ALL] [(column)]
   ON [schema.] object TO user [PUBLIC]
 WITH GRANT OPTION;
```

- Object_priv
  An object privilege to be granted. You can substitute any of these values:

Object Privilege	Tables	Views	Sequences
ALTER	✓		✓
DELETE	✓	✓	
INDEX	✓		
INSERT	✓	✓	
REFERENCES	✓		
SELECT	✓	✓	✓
UPDATE	✓	✓	

- ALL
  Grants all the privileges to the grantee.
- Column
  Specifies a table or view column on which privileges are granted. You can only specify columns when granting the INSERT, REFERENCES, or UPDATE privilege. If you do not list columns, the grantee has the specified privilege on all the columns in the table or view.
- ON
  Identifies the object on which the privileges are granted. If you do not qualify object with schema, Oracle assumes the object is in your own schema.
- TO
  Identifies users to which the object privilege is granted. PUBLIC grants object privileges to all users.

Continued on the next page

Security, Continued

WITH GRANT OPTION

Allows the grantee to grant the object privileges to other users.

```
SQL> GRANT SELECT
        ON swriden
        TO PUBLIC;

SQL> GRANT ALL
        ON swriden
        TO train01;

SQL> GRANT UPDATE (id, last_name, first_name,
                   mi, change_ind, activity_date)
       ON  swriden
       TO  train02
     WITH GRANT OPTION;
```

Self Check

Directions Use the information you have learned in this workbook to complete this self check activity.

Exercise 1 Create a view called **SWVADDR_XX** ***(where XX is your*** user number) which contains a person's first name, last name (combine it into one column called name), city, state, and zip based on the **SWRIDEN** and **SWBADDR** tables.

Exercise 2 Retrieve all the rows from the new view.

Exercise 3 Grant the right to select from the view to a person sitting next to you. Make sure someone gives you the right to select from his/her new view.

Exercise 4 Try to select all columns from the view you were just granted access to in Exercise 3. What happened?

Exercise 5 Now, put the owner name in front of the table, in the syntax below. Did you get results?

- `SELECT * FROM <owner.table_name>`

Continued on the next page

Self Check, Continued

Exercise 6

Because you have to specify the owner each time you are referring to the view, create a synonym to alleviate this.

Exercise 7

Select all columns from the view. You should not have to specify the owner in front of the view.

Exercise 8

To make data retrieval faster, create an index for PIDM on **SWRIDEN**.

Exercise 9

Create a sequence which will be used to generate a new PIDM. Find out what the first value should be by finding the maximum existing PIDM +1. Insert a new row into the **SWRIDEN** using your sequence to generate the PIDM.

Continued on the next page

Self Check, Continued

Exercise 10

Create a relationship between the validation table T**WVDETC** and the repeating table **TWRACCD**. TWVDETC should have the primary key of DETC_CODE and TWRACCD should have the foreign key of DETC_CODE.

Exercise 11

Create a table called **TEMP_XX** (where XX is your user number) with the following structure:

```
MYNUMBER      NUMBER (8)
TEXT          VARCHAR2 (30)
MYDATE        DATE
MESSAGE       VARCHAR2 (50)
```

Self Check – Answer Key

Exercise 1

Create a view called **SWVADDR_XX** ***(where XX is your*** user number) which contains a person's first name, last name (combine it into one column called name), city, state, and zip based on the **SWRIDEN** and **SWBADDR** tables.

```
SQL> CREATE VIEW swvaddr_XX  /* For user train01 */
        AS SELECT first_name||' '||last_name name,
                  city, stat_code state, zip
             FROM swriden, swbaddr
            WHERE swriden.pidm = swbaddr.pidm
              AND change_ind IS NULL;
```

Exercise 2

Retrieve all the rows from the new view.

```
SQL> SELECT * FROM swvaddr_XX
```

Exercise 3

Grant the right to select from the view to a person sitting next to you. Make sure someone gives you the right to select from his/her new view.

```
SQL> GRANT SELECT ON swvaddr_XX TO trainXX;
```

Exercise 4

Try to select all columns from the view you were just granted access to in Exercise 3. What happened?

```
SELECT * FROM swvaddr_XX;

ERROR at line 1:
ORA-00942: table or view does not exist
```

This is because you did not specify the owner name in front of the view.

Exercise 5

Now, put the owner name in front of the table, in the syntax below. Did you get results?

- `SELECT * FROM <owner.table_name>`

```
SELECT * FROM trainXX.swvaddr_XX;
```

Continued on the next page

Exercise 6

Because you have to specify the owner each time you are referring to the view, create a synonym to alleviate this.

```
SQL> CREATE SYNONYM swvaddr_XX
        FOR trainXX.swvaddr_XX;
```

Exercise 7

Select all columns from the view. You should not have to specify the owner in front of the view.

```
SQL> SELECT * FROM swvaddr_XX;
```

Exercise 8

To make data retrieval faster, create an index for PIDM on **SWRIDEN**.

```
SQL> CREATE INDEX swriden_key_index
          ON swriden (pidm);

Index created.
```

Exercise 9

Create a sequence which will be used to generate a new PIDM. Find out what the first value should be by finding the maximum existing PIDM +1. Insert a new row into the **SWRIDEN** using your sequence to generate the PIDM.

```
SQL> SELECT MAX(pidm) + 1 FROM swriden;

MAX(PIDM)+1
-----------
      12350

SQL> CREATE SEQUENCE PIDM_SEQUENCE
      START WITH 12350;

Sequence created.

SQL>  INSERT INTO swriden (pidm, id, last_name,
                           activity_date)
      VALUES   (pidm_sequence.NEXTVAL, '123ME',
                  'Smith', sysdate);
```

Continued on the next page

Self Check – Answer Key, Continued

Exercise 10

Create a relationship between the validation table T**WVDETC** and the repeating table **TWRACCD**. TWVDETC should have the primary key of DETC_CODE and TWRACCD should have the foreign key of DETC_CODE.

```
SQL> ALTER TABLE twvdetc
       ADD CONSTRAINT pk_twvdetc
         PRIMARY KEY (detc_code);

Table altered.

SQL> ALTER TABLE twraccd
     ADD CONSTRAINT fk1_twraccd_inv_twvdetc_key
         FOREIGN KEY (detc_code)
         REFERENCES twvdetc;
Table altered.
```

Exercise 11

Create a table called **TEMP_XX** (where XX is your user number) with the following structure:

```
MYNUMBER     NUMBER (8)
TEXT         NUMBER (8)
MYDATE       DATE
MESSAGE      VARCHAR2 (50)
```

```
SQL> CREATE TABLE temp_xx
    (mynumber  NUMBER(8),
     text      VARCHAR2(15),
     mydate    DATE,
     message   VARCHAR2(50));
```

Section J: SQL*Loader

Overview

Purpose

SQL*Loader gives the capability to easily load data from a flat file to database tables. It is a great tool to use when converting existing legacy data into the SCT Banner tables. Although we will cover only the basics of SQL*Loader in this section, the utility is quite powerful and has many options.

SQL*Loader can:

- Load data from multiple datafiles of different file types
- Handle fixed-format, delimited-format, and variable-length records
- Manipulate data fields with SQL functions before inserting the data into database columns
- Support a wide range of datatypes, including DATE, BINARY, PACKED DECIMAL, and ZONED DECIMAL
- Load multiple tables during the same run, loading selected rows into each table packages
- Combine multiple physical records into a single logical record
- Treat a single physical record as multiple logical records
- Generate unique, sequential key values in specified columns
- Use the operating system's file or record management system to access datafiles
- Load data from disk or tape
- Provide thorough error reporting capabilities, so you can easily adjust and load all records
- Use high-performance "direct" loads to load data directly into database files without Oracle processing.

Objectives

Upon completion of this section, each attendee will be able to:

- Define the basic file types which are required to load data
- Invoke the SQL*Loader
- Analyze output files for errors

Continued on the next page

Overview, Continued

In this section

These topics are covered in this section.

Required input files

Input files

In order to load data from a flat file into a table, there must be two files: a data file and a control file (or they can be combined into one file, as we will see later). The data file contains the data you want to load, and the control file specifies the format and the destination of the data.

The data file

There are two main data file types: fixed format files and variable format files.

Fixed format files:

- Contain records with fixed length, and the data fields in those records have fixed length, type, and position
- Positioning of data is crucial because the position in the file is the only way to distinguish between the data items

Variable format files:

- Have records that are only as long as necessary to contain the data
- The fields' positions and lengths are based on delimiters
  - Terminated fields - followed by a specified character (usually a comma)
  - Enclosed fields - both preceded and followed by specified characters (usually quotation marks). Used for strings that might contain spaces

The control file

The control file contains several kinds of information which are necessary for loading the data (in bold in Control file format). The remaining clauses are all optional; they can be used to describe and manipulate the file data.

The control file uses the Data Definition Language (DDL), which describes:

- Data location
- Data format
- Column definition
- Datatype mapping
- Field specifications

Continued on the next page

Required input files, Continued

Considerations

Things to keep in mind:

- The name of the data field corresponds to the name of the table column into which the data is loaded
- The datatype of the field tells SQL*Loader how to read the data in the datafile. It is not necessarily the same as the column datatype
- Data is converted from the datatype specified in the control file to the datatype of the column in the database
- If any of your column names are one of the following reserved words, then you must enclose them in quotation marks:
  - COUNT
  - DATA
  - DATE
  - FORMAT
  - OPTIONS
  - PART
  - POSITION

Control file format

Listing

```
OPTIONS ({[SKIP=integer] [LOAD=integer]
[ERRORS=integer] [ROWS=integer]
[BINDSIZE=integer] [SILENT=(ALL | [FEEDBACK |
 ERROR | DISCARDS] })

LOAD   [DATA]
[   {   INFILE | INDDN} {file   |   *   }
{  STREAM | RECORD | FIXED length [BLOCKSIZE size]   |
VARIABLE [length]   ]
[   {   BADFILE | BADDN } file   ]
[   {   DISCARDS | DISCARDMAX }   integer   ]

[   {   INDDN | INFILE } ...]

[   APPEND | REPLACE | INSERT ]
[   RECLEN integer  ]

[   {   CONCATENATE integer  |
    CONTINUEIF { [THIS | NEXT ] ( start[   :end] ) | LAST}
          operator { 'string'  | X'hex' } } ]

INTO TABLE [user.]table
          [APPEND | REPLACE | INSERT ]
          [WHEN condition  [ AND condition  ] ...]
          [FIELDS   [delimiter] ]
            (column
            {RECNUM |
         CONSTANT value |
      SEQUENCE ( { integer | MAX | COUNT } [ , increment ] ) |
           [ POSITION ( { start [ end} * [+integer] } ) ] datatype
             [TERMINATED [BY] {WHITESPACE | [X] 'character'}]
             [ [OPTIONALLY] ENCLOSED [BY] [X] 'character' ]
                                [ NULLIF condition ]
                                [ DEFAULTIF condition  ] }
             [   ,... ] )
       [ INTO TABLE ... ]
       [ BEGIN DATA ]
```

Generating data

Functions

The following functions provide the means for SQL*Loader to generate the data stored in the database row rather than reading it from a data file. You can use these functions in your control file after you specify the database column that you want to populate.

- CONSTANT
- RECNUM
- SYSDATE
- SEQUENCE

CONSTANT

To specify a constant for a column, use the keyword CONSTANT followed by a value:

```
column_name CONSTANT value
```

RECNUM

Sets the column to the number of the logical record from which that row was loaded. Records are counted sequentially from the beginning of the first datafile starting with record one. It increments for records that are discarded, skipped, rejected, or loaded.

```
column_name RECNUM
```

SYSDATE

Sets the column to the current system date.

```
column_name SYSDATE
```

SEQUENCE

The SEQUENCE keyword ensures a unique value for a particular column (can be used for PIDMS). It does not increment for records that are discarded or skipped.

The combination of column name and the SEQUENCE function is a complete column specification.

```
column_name SEQUENCE (n | MAX | COUNT ,[increment])
```

where:

- *n*
  The sequence starts with the integer value n.
- COUNT
  The sequence starts with the number of rows already in the table, plus the increment.
- MAX
  The sequence starts with the current maximum value for the column, plus the increment.
- *increment*
  The sequence is incremented by this amount for each successive row. The default increment is 1.

Handling blanks in records

Null values	If you want all inserted values for a given column to be null, omit the column's specifications entirely. To set a column's values conditionally to null based on a test of some condition in the logical record, use the NULLIF clause. To set a numeric column to zero instead of NULL, use the DEFAULTIF clause.
NULLIF	Totally blank fields for numeric or DATE fields cause the record to be rejected. To load one of these fields as null, use the NULLIF clause with the BLANKS keyword. If an all-blank CHAR field is surrounded by enclosure delimiters, then the blanks within the enclosures are loaded. Otherwise, the field is loaded as null.
DEFAULTIF	Using DEFAULTIF on numeric data sets the column to zero when the specified field condition is true. Using the DEFAULTIF on character data (CHAR, DATE, or numeric EXTERNAL) data sets the column to null.

SQL*Loader examples

Basic Person Information

The following Basic Person Information is to be loaded into SWBPERS:

```
SQL> desc swbpers
 Name                      Null?        Type
 ------------------------  --------     ----
 PIDM                      NOT NULL     NUMBER(8)
 SSN                                    VARCHAR2(9)
 BIRTH_DATE                             DATE
 MRTL_CODE                              VARCHAR2(1)
 SEX                                    VARCHAR2(1)
 CONFID_IND                             VARCHAR2(1)
 ACTIVITY_DATE             NOT NULL     DATE
```

Continued on the next page

SQL*Loader examples, Continued

Fixed format

A fixed format data file will look something like this:

```
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890
----------------------------------------------------------------

     6415628629112-OCT-1965SF
     6527965326310-AUG-1972MM
     6616925689405-JAN-1975SMY
     ...
```

(SWBPERS.DAT)

Control file

Your associated control file will look something like this:

```
LOAD DATA
INFILE 'SWBPERS1.DAT'
BADFILE 'SWBPERS1.BAD'
DISCARDFILE 'SWBPERS1.DSC'
APPEND
INTO TABLE SWBPERS
(pidm             POSITION(01:08) INTEGER EXTERNAL,
 ssn              POSITION(09:17) CHAR,
 birth_date       POSITION(18:28) DATE "DD-MON-YYYY"
                        NULLIF birth_date = BLANKS,
 mrtl_code        POSITION(29:29) CHAR,
 sex              POSITION(30:30) CHAR,
 confid_ind       POSITION(31:31) CHAR(1),
 activity_date                    SYSDATE)
```

(SWBPERS.CTL)

Analysis

- We have chosen the option of appending to the table. This is the safest option because the existing records within the table will not be affected by our actions.
- The length of the fields is explicitly specified and remains standard throughout the entire data file. Following the POSITION specification is the field's datatype from the data file.
- The birth_date from the data file is in the format of a four-digit year rather than two-digit. Therefore, a mask is specified so that SQL*Loader can convert this.
- Because the birth_date is not required, using the NULLIF function will insert a null into the column. The DEFAULTIF function would have worked as well.
- Our character fields will automatically default to NULL if it is equal to spaces in the data file; so we have not used the NULLIF or DEFAULTIF function. We would only use the NULLIF or DEFAULTIF function for character fields when our data was enclosed with delimiters.

Continued on the next page

SQL*Loader examples, Continued

Text delimited

Your data file will look something like this:

```
64,156286291,12-OCT-1965,S,F,
65,279653263,10-AUG-1972,M,M,
66,169256894,05-JAN-1975,S,M,Y
...
```

(SWBPERS2.DAT)

Control file

Your associated control file will look something like this:

```
LOAD DATA
INFILE 'SWBPERS2.DAT'
BADFILE 'SWBPERS2.BAD'
DISCARDFILE 'SWBPERS2.DSC'
INTO TABLE SWBPERS
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(pidm                integer external,
 activity_date       SYSDATE,
 ssn                 CHAR(9),
 birth_date          DATE(11) "DD-MON-YYYY"
                      NULLIF birth_date = BLANKS,
 mrtl_code           CHAR(1),
sex                  CHAR(1),
 confid_ind          CHAR(1))
```

(SWBPERS2.CTL)

Analysis

- Neither APPEND, INSERT, or REPLACE has been chosen as how the records should be inserted. Therefore, the default of INSERT will be used. This means that the table must be empty before SQL*Loader is run or it will abort.
- Because the data file is terminating the fields by a comma, each field's length from each record will be evaluated separately. Positions are relative and not fixed.

Continued on the next page

SQL*Loader examples, Continued

Combined files

You can combine the control file and data file into one. The asterisk indicates to SQL*Loader that the data is contained within the file, and the BEGINDATA keyword indicates that the data will begin on the following line.

```
LOAD DATA
INFILE *
BADFILE 'SWBPERS.BAD'
DISCARDFILE 'SWBPERS.DSC'
APPEND
INTO TABLE SWBPERS
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(pidm                integer external,
 activity_date       SYSDATE,
 ssn                 CHAR(9),
 birth_date          DATE(11) "DD-MON-YYYY"
NULLIF birth_date = BLANKS,
 mrtl_code           CHAR(1),
 sex                 CHAR(1),
 confid_ind          CHAR(1))

BEGINDATA
64,156286291,12-OCT-1965,S,F,
65,279653263,10-AUG-1972,M,M,
66,169256894,05-JAN-1975,S,M,Y
...
```

(SWBPERS.CTL)

Invoking SQL*Loader

Running SQL*Loader

Once you have your files ready, it is time to run. At the prompt (for UNIX), type the following:

```
sqlldr username control={control file}
```

You will be prompted for your password.
For our previous example, we would invoke SQL*Loader by typing the following:

```
$sqlldr username control=SWBPERS1.CTL
```

SQL*Loader process

Output files

The following file types will be created if specified in either the control file or the command line. These files are created by SQL*Loader to indicate to you the success or failure of the load.

- Log Files (.LOG)
- Bad Files (.BAD)
- Discard Files (.DSC)

The log file

The log file will provide the statistics of the load. The first section shows the parameters and data layout that it pulled from the control file. The second section lists the rejected record numbers and rejection reasons. The end of the file will list the total number of records read, records that passed, and records that failed.

Log file example

Here is an example of a log file from the fixed format SWBPERS example, as shown previously:

```
SQL*Loader: Release 7.2.3.0.0 - Production on Wed Feb 26
15:04:45 1997

Copyright (c) Oracle Corporation 1979, 1994.  All rights
reserved.

   Control File:   ././SWBPERS1.CTL
   Data File:      ././SWBPERS1.DAT
   Bad File:     ././SWBPERS1.BAD
   Discard File: SWBPERS1.DSC
   (Allow all discards)
   Number to load: ALL
   Number to skip: 0
   Errors allowed: 50
   Bind array:     64 rows, maximum of 65536 bytes
   Continuation:   none specified
   Path used:      Conventional
```

Continued on the next page

SQL*Loader process, Continued

Log file example (cont.)

```
Table SWBPERS, loaded from every logical record.
Insert option in effect for this table: APPEND

Column Name                  Position   Len  Term  Encl Datatype
---------------------- ---------- ---- ----  ---- ---------
--
PIDM                          1:8       9     ,    O(") CHARACTER
SSN                          9:17       9     ,    O(") CHARACTER
BIRTH_DATE                  18:28      11     ,    O(") DATE
     DD-MON-YYYY
MRTL_CODE                   29:29       1     ,    O(") CHARACTER
SEX                         30:30       1     ,    O(") CHARACTER
CONFID_IND                  31:31       1     ,    O(") CHARACTER

ACTIVITY_DATE      SYSDATE
Column BIRTH_DATE is NULL if BIRTH_DATE = BLANKS
Record 4: Rejected - Error on table SWBPERS.
ORA-01400: mandatory (NOT NULL) column is missing or NULL
during insert
Record 5: Rejected - Error on table SWBPERS, column
BIRTH_DATE.
ORA-01841: (full) year must be between -4713 and +4713
Table SWBPERS:
  3 Rows successfully loaded.
  2 Rows not loaded due to data errors.
  0 Rows not loaded because all WHEN clauses were
  failed.
  0 Rows not loaded because all fields were null.


Space allocated for bind array:  3328 bytes(64 rows)
Space allocated for memory
besides bind array:             58658 bytes

Total logical records skipped:          0
Total logical records read:             5
Total logical records rejected:         2
Total logical records discarded:        0

Run began on Wed Feb 26 15:04:45 1997
Run ended on Wed Feb 26 15:04:53 1997
Elapsed time was:     00:00:07.86
CPU time was:         00:00:00.30
```

Continued on the next page

SQL*Loader process, Continued

Analyzing the log file

- Although we did not specify the errors allowed or the bind array in the control file, these values defaulted. You can always check the log to figure out the default values SQL*Loader is using.
- Because the log file is from a fixed format data file, the positions are listed along with the length. For variable length format, position would have FIRST and then NEXT, and the Length would contain an asterisk (*) because the length will be determined on a record by record basis.
- Although five logical records were read by SQL*Loader, only three were successfully inserted into the SWBPERS table. In order to fix these errors, edit the.BAD file and then run the control file against it.
- Record 4 was rejected because a mandatory column was missing.
- Record 5 was rejected because the date format was incorrect.

The bad file

The bad file contains records rejected because of incorrect data. Data can be considered "bad" for a number of reasons. Data is rejected by SQL*Loader when the input format is invalid, such as when the second enclosure delimiter is missing or when a delimited field exceeds its maximum length.

Once SQL*Loader accepts a record for processing, the row is sent to Oracle for insertion. At this point, the row can also be considered "bad". Examples might be because a key is not unique, because a required field is null, or because the field contains invalid data for the Oracle datatype.

The discard file

The discard file contains records that were filtered out of the load because they did not match any of the record-selection criteria (WHEN clause) specified in the control file. This file is created only when it is needed, and only if you have specified that a discard file should be enabled.

The discard field is written in the same format as the datafield. The discard data can be loaded with the existing control file after any necessary editing or correcting.

Self Check

Directions	Use the information you have learned in this workbook to complete this self check activity.
Exercise 1	Examine the SWRIDEN.DAT file which has been provided. What type of data file is it?
Exercise 2	Create a control file that will load the data into the table. For the PIDM, obtain the maximum PIDM number in the table and increment by one. Use the current system date for the activity date.
Exercise 3	Run SQL*Loader.
Exercise 4	Examine your log file. What was the success rate? Which records, if any, did not load correctly?
Exercise 5	What steps would you take to fix the records that had errors and reload the data?

Self Check – Answer Key

Exercise 1

Examine the SWRIDEN.DAT file which has been provided. What type of data file is it?

A comma delimited file.

Exercise 2

Create a control file that will load the data into the table. For the PIDM, obtain the maximum PIDM number in the table and increment by one. Use the current system date for the activity date.

```
LOAD DATA
INFILE 'swriden.dat'
BADFILE 'swriden.bad'
DISCARDFILE 'swriden.dsc'
APPEND
INTO TABLE swriden
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
        TRAILING NULLCOLS
(pidm            SEQUENCE(MAX, 1),
 activity_date SYSDATE,
 id              CHAR,
 last_name       CHAR,
 first_name      CHAR,
 mi              CHAR,
 change_ind      CHAR)
```

Exercise 3

Run SQL*Loader.

```
$sqlldr <username>/<password> CONTROL = swriden.ctl
```

Exercise 4

Examine your log file. What was the success rate? Which records, if any, did not load correctly?

Record 4 should have errored because the ID was too long. All others should have been successfully inserted.

Exercise 5

What steps would you take to fix the records that had errors and reload the data?

You can go into your SWRIDEN.BAD file and fix the errors in the records that contained errors. Then, you could run your control file against the SWRIDEN.BAD file instead of the SWRIDEN.DAT file.

Section K: SQL*Plus Reporting

Overview

Purpose

So far, you have learned how to query the database to return information. However, the data has not been formatted for reporting purposes. SQL*Plus gives you the capability to format the appearance of the data returned from an SQL Statement.

Objectives

Upon completion of this section, each attendee will be able to:

- Change column headings
- Format NUMBER, CHAR, VARCHAR2 (VARCHAR), LONG, DATE, and columns
- Copy, list, and reset column display attributes
- Suppress duplicate values and insert spaces for clarity
- Calculate and print summary lines (totals, averages, minimums, maximums, and more)
- List and remove spacing and summary line definitions
- Set page dimensions
- Place titles at the top and bottom of each page
- Display column values and the current date or page number in titles
- List and suppress page title definitions
- Send query results to a file or printer

Continued on the next page

Overview, Continued

In this section

These topics are covered in this section.

Example query

Example query

Our examples will be based upon the following query. Go ahead and type it into SQL*Plus, and execute it to see your results.

```
SQL> SELECT    term_code, last_name||', '||
              first_name name, bill_date,
               detc_code, amount
         FROM  swriden, twraccd
        WHERE  swriden.pidm = twraccd.pidm
          AND  swriden.change_ind is null
     ORDER BY  term_code,last_name,
               First_name,bill_date,detc_code;
```

Formatting columns

COLUMN command

Through the SQL*Plus COLUMN command, you can change the column headings and reformat the column data in your query results.

Changing column headings

SQL*Plus uses column or expression names as default column headings when displaying query results. Column names are often short and cryptic, however, and expressions can be hard to understand. You can define a more useful column heading with the HEADING clause of the COLUMN command in the format shown below:

```
COLUMN column_name HEADING column_heading
```

The new headings will remain in effect until you enter different headings, reset each column's format or exit from SQL*Plus.

EXAMPLE: Enter the following line:

```
SQL> COLUMN DETC_CODE HEADING 'Category'
```

Quotation marks

To change a column heading to two or more words, enclose the new heading in single or double quotation marks when you enter the COLUMN command. To display a column heading on more than one line, use a vertical bar (|) where you want to begin a new line.

EXAMPLE: Enter the following line:

```
SQL> COLUMN BILL_DATE HEADING 'Billing|Date'
```

Display query

Type in the rest of the column headings so that the query displays:

```
                              Billing
Term   Name                   Date      Category    Amount
------ -------------------- --------- -------- ---------
199602 Erickson, Ralph      21-MAY-97 LABS            120
199602 Erickson, Ralph      21-MAY-97 DORM           1000
199701 Brown, Julie         21-MAY-97 BOOK          300.2
199701 Brown, Julie         21-MAY-97 TUIT         1500.5
199701 Brown, Julie         28-MAY-97 BKSC            700
199701 Erickson, Ralph      21-MAY-97 MEAL            900
...
```

Continued on the next page

Formatting columns, Continued

Underline character

To change the character used to underline each column heading, set the UNDERLINE variable of the SET command to the desired character.

EXAMPLE: Enter the following:

```
SQL> SET UNDERLINE =

SQL> /
```

SQL*Plus displays the following results:

```
                            Billing
Term   Name                 Date      Category    Amount
====== ==================== ========= ======== =========
199602 Erickson, Ralph      21-MAY-97 LABS           120
199602 Erickson, Ralph      21-MAY-97 DORM          1000
199701 Brown, Julie         21-MAY-97 BOOK         300.2
199701 Brown, Julie         21-MAY-97 TUIT        1500.5
199701 Brown, Julie         28-MAY-97 BKSC           700
199701 Erickson, Ralph      21-MAY-97 MEAL           900
199701 Erickson, Ralph      21-MAY-97 CRED           400
...
```

Now change the underline character back to a dash:

```
SQL> SET UNDERLINE '-'
```

Enclose the dash in quotation marks; otherwise, SQL*Plus interprets the dash as a hyphen indicating that you wish to continue the command on another line.

Continued on the next page

Formatting columns, Continued

Formatting your columns

Although we now have customized our headings for the report, we also want to change the appearance of the data itself for the report.

The COLUMN command identifies the column you want to format and the model you want to use, as shown below:

```
COLUMN column_name FORMAT model
```

The format model will stay in effect until you enter a new one, reset the column's format, or exit from SQL*Plus.

Formatting a character column

The default display width for CHAR and VARCHAR2 (VARCHAR) values is the width defined for the column in the database or the width of the column heading, whichever is longer. The default display can be changed using the COLUMN command. Use a format model consisting of the letter A (for alphanumeric), followed by a number representing the width of the column in characters.

You may notice that the Category title is truncated and that the Name column is taking up a lot of space. First, we will decrease the size of the Name column, and then increase the length of the DETC_CODE column.

To set the width of the column NAME to 25 characters, enter:

```
SQL> COLUMN NAME FORMAT A25
```

To set the width of the column DETC_CODE to eight characters and rerun the current query, enter:

```
SQL> COLUMN DETC_CODE FORMAT A8
SQL> /
```

SQL*Plus displays the results:

```
                                Billing
Term   Name                      Date      Category  Amount
------ -------------------- --------- -------- -------
199602 Erickson, Ralph      21-MAY-97 LABS         120
199602 Erickson, Ralph      21-MAY-97 DORM        1000
199701 Brown, Julie         21-MAY-97 BOOK       300.2
199701 Brown, Julie         21-MAY-97 TUIT      1500.5
199701 Brown, Julie         28-MAY-97 BKSC         700
199701 Erickson, Ralph      21-MAY-97 MEAL         900
199701 Erickson, Ralph      21-MAY-97 CRED         400
...
```

Continued on the next page

Formatting columns, Continued

Formatting a number column

Use number format models to add commas, dollar signs, angle brackets (around negative values), and/or leading zeros to numbers in a given column. You can also round the values to a given number of decimal places, display minus signs to the right of negative values (instead of to the left), and display values in exponential notation.

To display AMOUNT with a dollar sign, a comma, and the numeral zero instead of a blank for any zero values, enter the following command:

```
SQL> COLUMN AMOUNT  FORMAT $99,990.00
```

Now rerun the current query:

```
SQL> /
```

SQL*Plus displays the following output:

```
                                Billing
Term   Name                     Date      Category Amount
------ ------------------------ --------- -------- --------
199602 Erickson, Ralph          21-MAY-97 LABS      $120.00
199602 Erickson, Ralph          21-MAY-97 DORM    $1,000.00
199701 Brown, Julie             21-MAY-97 BOOK      $300.20
199701 Brown, Julie             21-MAY-97 TUIT    $1,500.50
199701 Brown, Julie             28-MAY-97 BKSC      $700.00
199701 Erickson, Ralph          21-MAY-97 MEAL      $900.00
199701 Erickson, Ralph          21-MAY-97 CRED      $400.00
199701 Erickson, Ralph          25-MAY-97 CASH      $800.00
...
```

Listing and resetting column display attributes

List a column's current attributes	To list the current display attributes for a given column, use the COLUMN command followed by the column name as shown below: `COLUMN [column_name]`
List all columns' current attributes	To list the current display attributes for all columns, enter the COLUMN command with no column names or clauses after it: `COLUMN`
Reset a column to default values	To reset the display attributes for a column to their default values, use the CLEAR clause of the COLUMN command as shown below. Do not clear the columns now or you will undo the previous steps. `COLUMN [column_name] CLEAR`
Reset all columns to default values	To reset all columns' display attributes to their default values, enter the following command: `CLEAR COLUMNS` You may wish to place the command CLEAR COLUMNS at the beginning of every command file to ensure that previously entered COLUMN commands will not affect queries you run in a given file.

Suppressing and restoring column display attributes

Suppress a column's display attributes

You can suppress and restore the display attributes you have given a specific column. To suppress a column's display attributes, enter a COLUMN command in the following form:

```
COLUMN column_name OFF
```

Restore defined attributes

The OFF clause tells SQL*Plus to use the default display attributes for the column, but does not remove the attributes you have defined through the COLUMN command. To restore the attributes you defined through COLUMN, use the ON clause:

```
COLUMN column_name ON
```

Suppressing duplicate values in break columns

Suppress duplicate values

The BREAK command suppresses duplicate values by default in the column or expression you name. Thus, to suppress the duplicate values in a column specified in an ORDER BY clause, use the BREAK command in its simplest form:

```
BREAK ON column_name
```

Note: Whenever you specify a column or expression in a BREAK command, use an ORDER BY clause specifying the same column or expression. If you do not do this, the breaks may appear to occur randomly.

Example

To suppress the display of duplicate term codes in the previous query results, enter the following commands:

```
SQL> BREAK ON TERM_CODE
```

SQL*Plus displays the following output:

```
                                 Billing
Term   Name                      Date      Category   Amount
------ ------------------------- --------- -------- ----------
199602 Erickson, Ralph           21-MAY-97 LABS        $120.00
       Erickson, Ralph           21-MAY-97 DORM      $1,000.00
199701 Brown, Julie              21-MAY-97 BOOK        $300.20
       Brown, Julie              21-MAY-97 TUIT      $1,500.50
       Brown, Julie              28-MAY-97 BKSC        $700.00
       Erickson, Ralph           21-MAY-97 MEAL        $900.00
       Erickson, Ralph           21-MAY-97 CRED        $400.00
       Erickson, Ralph           25-MAY-97 CASH        $800.00
       Erickson, Susan           21-MAY-97 TUIT        $300.00
       Jones-Erickson, Sandy     18-MAY-97 FAID      $1,100.00
       Jones-Erickson, Sandy     21-MAY-97 TUIT        $800.00
       Smith, Robert             21-MAY-97 DORM        $500.00
       Smith, Robert             21-MAY-97 TUIT      $1,100.00
       White, Nancy              21-MAY-97 CHEK        $950.00
       White, Nancy              21-MAY-97 LABS         $50.00
       White, Nancy              21-MAY-97 TUIT        $900.00
199702 Erickson, Ralph           21-MAY-97 BOOK        $400.00
       Erickson, Ralph           21-MAY-97 TUIT        $750.00
...
```

Inserting space when a break column's value changes

Insert blank lines

You can insert blank lines or begin a new page each time the value changes in the break column.

To insert n blank lines, use the BREAK command in the following form:

```
BREAK ON break_column SKIP n
```

Example

To place one blank line between term codes, enter the following command:

```
SQL> BREAK ON TERM_CODE SKIP 1
```

Now rerun the query:

```
SQL> /
```

SQL*Plus displays the results:

```
                                 Billing
Term   Name                      Date      Category     Amount
------ ------------------------- --------- -------- ----------
199602 Erickson, Ralph           21-MAY-97 LABS        $120.00
       Erickson, Ralph           21-MAY-97 DORM      $1,000.00

199701 Brown, Julie              21-MAY-97 BOOK        $300.20
       Brown, Julie              21-MAY-97 TUIT      $1,500.50
       Brown, Julie              28-MAY-97 BKSC        $700.00
       Erickson, Ralph           21-MAY-97 MEAL        $900.00
       Erickson, Ralph           21-MAY-97 CRED        $400.00
       Erickson, Ralph           25-MAY-97 CASH        $800.00
       Erickson, Susan           21-MAY-97 TUIT        $300.00
       Jones-Erickson, Sandy     18-MAY-97 FAID      $1,100.00
       Jones-Erickson, Sandy     21-MAY-97 TUIT        $800.00
       Smith, Robert             21-MAY-97 DORM        $500.00
       Smith, Robert             21-MAY-97 TUIT      $1,100.00
       White, Nancy              21-MAY-97 CHEK        $950.00
       White, Nancy              21-MAY-97 LABS         $50.00
       White, Nancy              21-MAY-97 TUIT        $900.00

199702 Erickson, Ralph           21-MAY-97 BOOK        $400.00
...
```

Using multiple spacing techniques

Specify multiple columns

Suppose you have more than one column in your ORDER BY clause and wish to insert space when each column's value changes. Each BREAK command you enter replaces the previous one. Thus, if you want to use different spacing techniques in one report or insert space after the value changes in more than one ordered column, you must specify multiple columns and actions in a single BREAK command.

Example

To skip a page when the value of TERM_CODE changes, one line when the value of NAME changes, and 2 lines at the end of the report, enter the following command:

```
SQL> BREAK ON TERM_CODE SKIP PAGE ON NAME SKIP 1
ON REPORT SKIP 2
```

Run the query to see the results:

```
                                 Billing
Term   Name                      Date      Category    Amount
------ ------------------------- --------- -------- ---------
199602 Erickson, Ralph           21-MAY-97 LABS       $120.00
                                 21-MAY-97 DORM     $1,000.00

                                  Billing
 Term   Name                      Date      Category    Amount
 ------ ------------------------- --------- -------- ---------

 199701 Brown, Julie              21-MAY-97 BOOK       $300.20
                                  21-MAY-97 TUIT     $1,500.50
                                  28-MAY-97 BKSC       $700.00

         Erickson, Ralph          21-MAY-97 MEAL       $900.00
                                  21-MAY-97 CRED       $400.00
                                  25-MAY-97 CASH       $800.00

        Erickson, Susan           21-MAY-97 TUIT       $300.00

        Jones-Erickson, Sandy     18-MAY-97 FAID     $1,100.00
                                  21-MAY-97 TUIT       $800.00

        Smith, Robert             21-MAY-97 DORM       $500.00
                                  21-MAY-97 TUIT     $1,100.00

        White, Nancy              21-MAY-97 CHEK       $950.00
                                  21-MAY-97 LABS        $50.00
                                  21-MAY-97 TUIT       $900.00


                                  Billing
Term   Name                      Date      Category    Amount
------ ------------------------- --------- -------- ---------

199702 Erickson, Ralph           21-MAY-97 BOOK       $400.00
                                 21-MAY-97 TUIT       $750.00
...
```

Listing and removing break definitions

List current break definition

You can list your current break definition by entering the BREAK command with no clauses:

```
BREAK
```

Remove current break definition

You can remove the current break definition by entering the CLEAR command with the BREAKS clause:

```
CLEAR BREAKS
```

Note: You may wish to place the command CLEAR BREAKS at the beginning of every command file to ensure that previously entered BREAK commands will not affect queries you run in a given file.

Computing summary lines when a break column's value changes

COMPUTE command

If you organize the rows of a report into subsets with the BREAK command, you can perform various computations on the rows in each subset. You do this with the functions of the SQL*Plus COMPUTE command. Use the BREAK and COMPUTE commands together in the following forms:

```
BREAK ON break_column
 COMPUTE function OF column column column ...
        ON break_column
```

Functions and effects

The following table lists compute functions and their effects:

Function	Effect
SUM	Computes the sum of the values in the column
MIN	Computes the minimum value in the column
MAX	Computes the maximum value in the column
AVG	Computes the average of the values in the column
STD	Computes the standard deviation of the values in the column
VAR	Computes the variance of the values in the column
COUNT	Computes the number of non-null values in the column
NUM	Computes the number of rows in the column

Continued on the next page

Computing summary lines when a break column's value changes, Continued

Example

To compute the total of AMOUNT by name, first list the current BREAK definition:

```
SQL> BREAK

      break on report skip 2 nodup
      on TERM_CODE page nodup
      on NAME skip 1 nodup
```

NOTE: When values in successive rows are duplicates, using `nodup` prevents the duplicates from being printed.

Now enter the following COMPUTE command and run the current query:

```
SQL> COMPUTE SUM OF AMOUNT ON NAME
SQL> /
```

SQL*Plus displays the following output:

```
                                 Billing
Term   Name                      Date      Category    Amount
------ ------------------------- --------- -------- ---------
199602 Erickson, Ralph           21-MAY-97 LABS       $120.00
                                 21-MAY-97 DORM     $1,000.00
       ************************                    -----------
       sum                                         $1,120.00


                                 Billing
Term   Name                      Date      Category    Amount
------ ------------------------- --------- -------- ---------
199701 Brown, Julie              21-MAY-97 BOOK       $300.20
                                 21-MAY-97 TUIT     $1,500.50
                                 28-MAY-97 BKSC       $700.00
       ************************                    -----------
       sum                                         $2,500.70
 ...
```

Computing summary lines at the end of the report

Compute summary lines

You will want to add a total billed amount for each term, and a grand total at the end of the report. First, find out what your existing COMPUTEs are by typing the following:

```
SQL> COMPUTE
```

Because we are summing for the same column, only at different breakpoints, we will revise the existing compute. Do this by typing the following:

```
SQL> COMPUTE SUM OF amount ON name term_code REPORT
```

You should see the total appearing after each name and term and a grand total at the end of the report.

Defining page titles and dimensions

TTITLE and BTITLE

You can set a title to display at the top of each page of a report as well as a title to display at the bottom of each page. The TTITLE command defines the top title; the BTITLE command defines the bottom title.

```
TTITLE position_clause(s) char_value
 position_clause(s) char_value ...
```

or

```
BTITLE position_clause(s) char_value
 position_clause(s) char_value ...
```

Clauses

The most often used clauses of TTITLE and BTITLE are summarized in the following table:

Clause	Example	Description
COL n	COL 72	Makes the next CHAR value appear in the specified column of the line.
SKIP n	SKIP 2	Skips to a new line n times. If n is greater than 1, n-1 blank lines appear before the next CHAR value.
LEFT	LEFT	Left-aligns the following CHAR value.
CENTER	CENTER	Centers the following CHAR value.
RIGHT	RIGHT	Right-aligns the following CHAR value.

Continued on the next page

Defining page titles and dimensions, Continued

Example

To put titles at the top and bottom of each page of a report, enter:

```
SQL> TTITLE CENTER 'ABC University' SKIP 1 CENTER
 'Billing Report' SKIP 2
SQL> BTITLE CENTER 'CONFIDENTIAL'
```

Now run the current query:

```
SQL> /
```

SQL*Plus displays the following output:

```
                                        ABC University
                                        Billing Report

                                 Billing
Term   Name                      Date      Category    Amount
------ ------------------------- --------- -------- ---------
199602 Erickson, Ralph           21-MAY-97 LABS       $120.00
                                 21-MAY-97 DORM     $1,000.00
       ************************                     -----------
       sum                                          $1,120.00

******                                              -----------
   sum                                              $1,120.00
   ...
                                       CONFIDENTIAL
   ...
```

Positioning title elements

Resetting LINESIZE

The report in the preceding exercise might look more attractive if you give the institution name more emphasis by centering the titles more closely around the data. You can accomplish these changes by resetting the system variable LINESIZE, as the following example shows.

```
SQL> SET LINESIZE 70
SQL> /
```

Displaying column values in titles

Changing master column value

You may wish to create a master/detail report that displays a changing master column value at the top of each page with the detail query results for that value below. You can reference a column value in a top title by storing the desired value in a variable and referencing the variable in a TTITLE command. Use the following form of the COLUMN command to define the variable:

```
COLUMN column_name NEW_VALUE variable_name
```

Example

```
SQL> COLUMN TERM_CODE NEW_VALUE TRMVAL NOPRINT
```

Because you will display the term code in the title, you do not want them to print as part of the detail. The NOPRINT clause you entered above tells SQL*Plus not to print the column TERM_CODE.

Next, include a label and the value in your page title, enter the proper BREAK command, and suppress the bottom title from the last example:

```
SQL> TTITLE CENTER 'ABC University'
                        RIGHT 'Page: ' -
             FORMAT 999 SQL.PNO
             SKIP 1 CENTER 'Billing Report'
             SKIP 2 - CENTER 'Term Code: ' TRMVAL
             SKIP 3
```

SQL*Plus displays the following output:

```
                  ABC University            Page:    1

                  Billing Report

                 Term Code: 199602


                           Billing
Name                       Date      Category   Amount
-------------------------- --------- -------- --------
Erickson, Ralph            21-MAY-97 LABS      $120.00
...
```

Changing the page length

Setting PAGESIZE

You will want to adjust the page size according to how many lines fit on a page for your printer. To set 60 lines per page, type in the following:

```
SQL> SET PAGESIZE 60
```

Storing and printing query results

Storing and printing

In most cases, you will want to store the results of the query into a file, or directly send the output to the printer.

Spool to file

To spool to a file:

```
SPOOL  <filename>
<report body>
....
SPOOL OFF
```

Spool to printer

To spool to the printer:

```
SPOOL OUT
<report body>
...
SPOOL OFF
```

Saving the commands to a file

Storing commands

You should now have an almost finished report. However, it is unrealistic that a user would want to type the commands in, line by line, until the desired results are achieved. Instead, you can store all the commands into one file, and then run the file.

Example file

An example file follows that you would have for the report that you created.

```
SET FEEDBACK OFF
SET ECHO OFF
CLEAR COLUMNS
CLEAR COMPUTES
CLEAR BREAKS
COLUMN NAME HEADING 'Name'
COLUMN DETC_CODE HEADING 'Category'
COLUMN BILL_DATE HEADING 'Billing|Date'
COLUMN AMOUNT HEADING 'Amount'
SET LINESIZE 70
SET PAGESIZE 60
COLUMN TERM_CODE NEW_VALUE TRMVAL NOPRINT
TTITLE CENTER 'ABC University' RIGHT 'Page: ' -
FORMAT 999 SQL.PNO SKIP 1 CENTER 'Billing Report' SKIP 2 -
CENTER 'Term Code: ' TRMVAL SKIP 3
BTITLE CENTER 'CONFIDENTIAL'
BREAK ON TERM_CODE SKIP PAGE ON NAME SKIP 1
COMPUTE SUM OF amount ON name term_code REPORT
SPOOL BILLING.RPT
      SELECT  TERM_CODE, LAST_NAME||', '||FIRST_NAME NAME,
              BILL_DATE, DETC_CODE, AMOUNT
        FROM  SWRIDEN, TWRACCD
       WHERE  SWRIDEN.PIDM = TWRACCD.PIDM
         AND  SWRIDEN.CHANGE_IND IS NULL
    ORDER BY  TWRACCD.TERM_CODE,SWRIDEN.LAST_NAME,
              SWRIDEN.FIRST_NAME, TWRACCD.BILL_DATE,
              TWRACCD.DETC_CODE;
SPOOL OFF
SET FEEDBACK ON
SET ECHO ON
```

Section L: Introduction to PL/SQL

Overview

Purpose

At this point in time, you should know all the basics of SQL - how to retrieve and manipulate data, control the scope of transactions through COMMITs and ROLLBACKs, and how to create and maintain schema objects in the database. If you are an experienced programmer, you might be asking yourself, "Where are my IF, THEN, ELSE statements?" and "What about loops?"

Enter PL/SQL - Procedural Language extensions to SQL. PL/SQL gives us the ability to put "standard" programming constructs around SQL statements.

Objectives

- Define the basic structure of PL/SQL
- Understand how PL/SQL interprets and executes statements
- Differentiate between PL/SQL versions

In this section

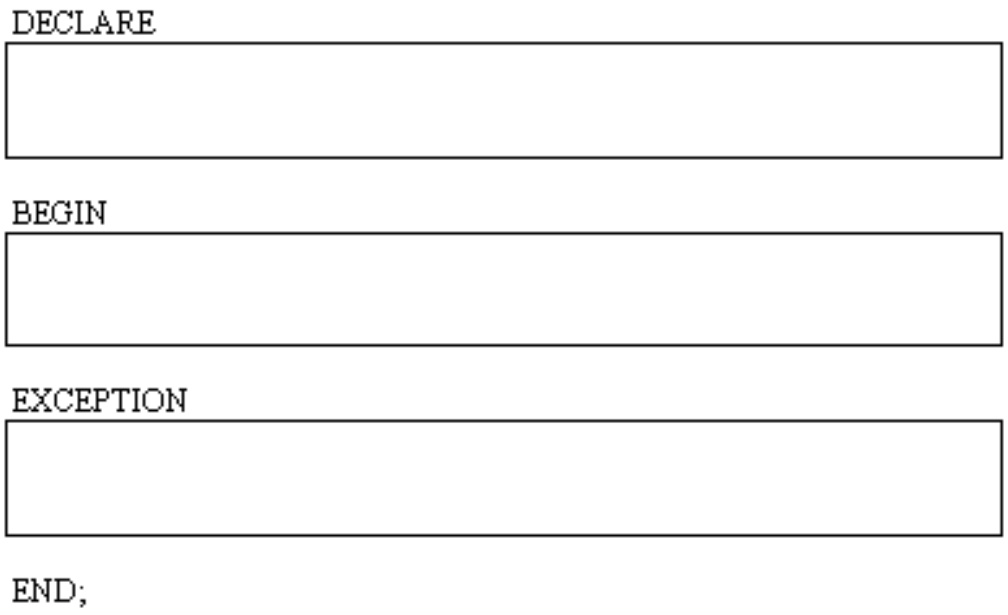
These topics are covered in this section.

PL/SQL Block Structure

Block structure

The foundation for the application of PL/SQL is through the PL/SQL Block Structure. This block structure provides you with the ability to scope related objects and make applications more modular.

Diagram



Continued on the next page

PL/SQL Block Structure, Continued

Conditions

PLSQL allows you to enclose your SQL statements with conditions.

```
/* This procedure will either insert or delete a record
from the swriden table, depending on the parameter of
action_in. */
   PROCEDURE maintain_pidms
   (action_in       IN VARCHAR2,
    pidm_in         IN NUMBER,
    id_in           IN VARCHAR2,
    last_name_in    IN VARCHAR2 := NULL,
    first_name_in   IN VARCHAR2 := NULL,
    mi_in           IN VARCHAR2 := NULL)
   IS
   BEGIN
      IF action_in = 'DELETE'
      THEN
         DELETE FROM swriden WHERE pidm = pidm_in;
      ELSIF action_in = 'INSERT'
      THEN
         INSERT INTO swriden
             (pidm, id, last_name, first_name, mi,
              activity_date)
              VALUES (pidm_in, id_in, last_name_in,
                      first_name_in, mi_in, SYSDATE);
      END IF;
   END;
```

Sections of the PL/SQL Block

Header For named blocks only such as procedures, functions, and packages, the Header section assigns a label to a given block and defines the type of block to be defined.

Declaration The part of the block that declares variables, cursors, and sub-blocks that are referenced in the execution and exception sections.

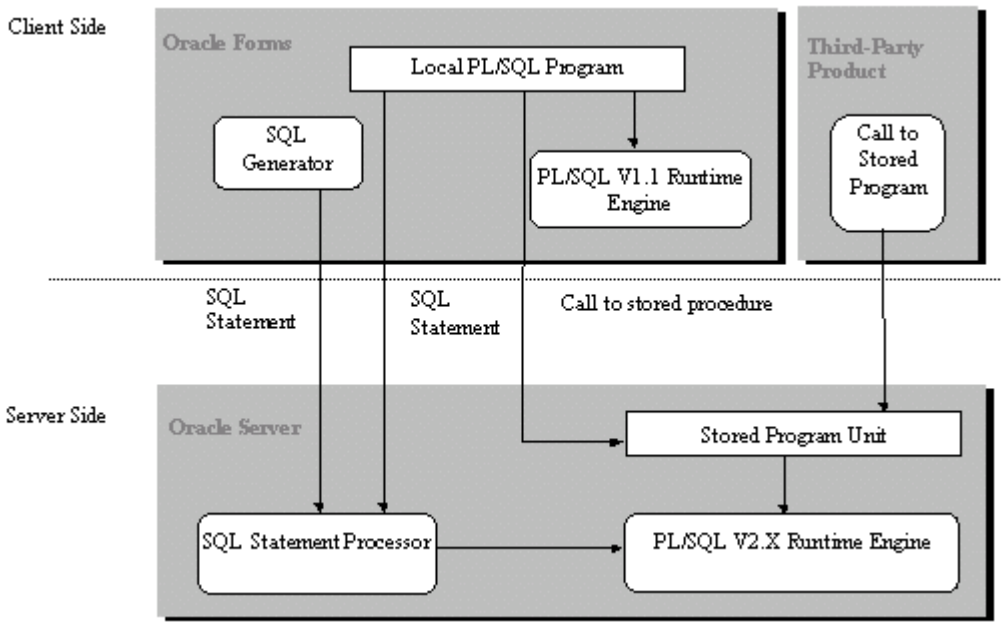
Execution The part of the PL/SQL block containing the executable statements.

Exception The section that handles exceptions to normal processing (warnings and error conditions).

- Blocks can contain sub-blocks
- Statements end with a semi-colon
- Declared objects exist within a certain scope (discussed in the following section)

How does PL/SQL work?

Diagram



1

[1] Feuerstein, Steven. Oracle PL/SQL Programming. Cambridge: O'Reilly & Associates, Inc., 1995.

PL/SQL Versions

Version table

Version/Release	Characteristics
Version 1.0	First available in SQL*Plus as a batch-processing script. Oracle Version 6.0 was released at approximately the same time. PL/SQL was then implemented within SQL*Forms Version 3, the predecessor of Oracle Forms.
Release 1.1	Available only in the Oracle Developer/2000 tools. This upgrade supports client-side packages and allows client-side programs to execute stored code transparently.
Version 2.0	Available with Release 7.0 (Oracle 7 Server). Major upgrade to Version 1. Adds support for stored procedures, functions, packages, programmer-defined records, PL/SQL tables, and many package extensions, including DBMS_OUTPUT and DBMS_PIPE.
Release 2.1	Available with Release 7.1 of the Oracle 7 Server Versions. Supports programmer-defined subtypes, enables the use of stored functions inside SQL statements, and offers dynamic SQL with the DBMS_SQL package. With Version 2.1, you can now execute SQL DDL statements from within PL/SQL programs.
Release 2.2	Available with Release 7.2 of the Oracle7 Server Version. Implements a binary 'wrapper' for PL/SQL programs to protect source code, supports cursor variables for embedded PL/SQL environments such a Pro*C, and makes available database-driven job scheduling with the DBMS_JOB package.
Release 2.3	Available with Release 7.3 of the Oracle7 Server Version. Enhances functionality of PL/SQL tables, offers improved remote dependency management, adds file I/O capabilities to PL/SQL, and completes the implementation of cursor variables.

2

[2] Feuerstein, Steven. Oracle PL/SQL Programming. Cambridge: O'Reilly & Associates, Inc., 1995.

Section M: Declaring Variables

Overview

Purpose

You can use all the datatypes defined within the database, including numeric, character and date. In addition, you can use the Boolean datatype, which returns the values of TRUE, FALSE, or NULL.

Objectives

Upon completion of this section, each attendee will be able to:

- Distinguish between legal and illegal declarations
- Understand the scope of a variable

In this section

These topics are covered in this section.

Variable declaration

Syntax

```
Identifier [CONSTANT] datatype [NOT NULL][:=plsql-expression];
```

Datatypes

Datatype table

Category	Datatype
Number	BINARY_INTEGER
	DEC
	DECIMAL
	DOUBLE PRECISION
	FLOAT
	INT
	INTEGER
	NATURAL
	NUMBER
	NUMERIC
	POSITIVE
	REAL
	SMALLINT
Character	CHAR
	CHARACTER
	LONG
	LONG RAW
	RAW
	ROWID
	STRING
	VARCHAR
	VARCHAR2
Boolean	BOOLEAN
Date-time	DATE

NUMBER

```
pidm      NUMBER;
amount    NUMBER(7,2);
tax       CONSTANT DECIMAL := .06;
fee       NUMBER(7,2) := 0;
```

CHAR/ VARCHAR2

```
last_name      VARCHAR2;
institution    VARCHAR2   NOT NULL
                          := 'ABC University';
```

Continued on the next page

Datatypes, Continued

DATE

```
application_date      DATE;
today                  DATE  := SYSDATE;
bill_date              DATE :=TO_DATE('10/10/97',
                              'MM/DD/YY');
```

BOOLEAN

```
amount_paid    BOOLEAN;
registered     BOOLEAN  NOT NULL := FALSE;
```

Association

You can associate PL/SQL objects with certain attributes from another object.

%TYPE

```
DECLARE
          amount           NUMBER(7,2);
          balance          amount%TYPE;
          internal_id      swriden.pidm%TYPE;
```

%ROWTYPE

```
DECLARE
          swriden_row  swriden%ROWTYPE;
```

Scoping Rules

Scope and visibility

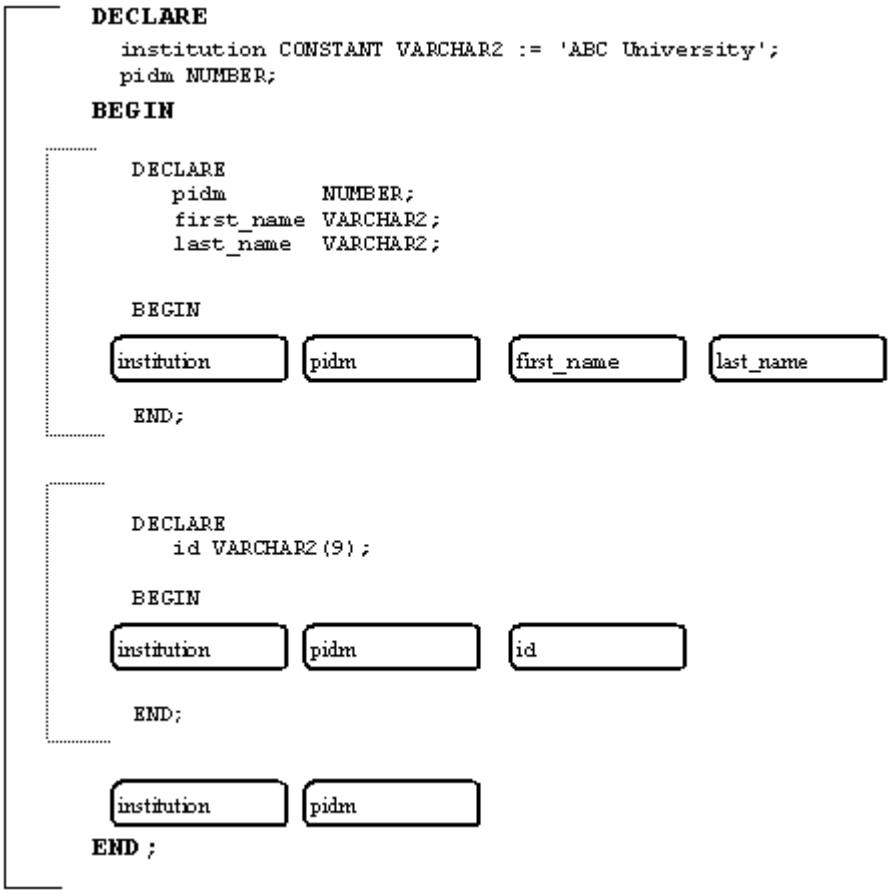
References to an identifier are resolved according to its scope and visibility.

- **Scope**
  The region of a program unit from which you can reference the identifier.
- **Visibility**
  An identifier is visible only in the regions from which you can reference the identifier using an unqualified name.

Rules

- Identifiers declared in a PL/SQL block are considered local to that block and global to all its sub-blocks
- If a global identifier is redefined in a sub-block, then both identifiers remain in scope
- Within the sub-block, however, only the local identifier is visible because you must use a qualified name to reference the global identifier

Example



Self Check

Directions Use the information you have learned in this workbook to complete this self check activity.

Exercise 1 Are the following statements legal or illegal? If illegal, specify why.

-
```
DECLARE
pidm NUMBER(9);
```

-
```
DECLARE
amount, balance NUMBER(7,2) ;
```

-
```
DECLARE
registered BOOLEAN NOT NULL;
```

-
```
DECLARE
institution VARCHAR2(20);
location  VARCHAR2;
```

-
```
DECLARE
max_amount NUMBER CONSTANT;
```

-
```
DECLARE
id      swriden.id%rowtype;
```

Continued on the next page

Self Check, Continued

Exercise 2 What are the values for each of the variables at the specified numbered steps?

```
DECLARE
   name  VARCHAR2(20) := 'Smith';
   balance NUMBER (7,2) := 100;
   fee NUMBER (7,2) := 0;
BEGIN
   DECLARE
      name VARCHAR2(20) := 'Johnson';
      balance NUMBER (7,2) := 200;
   BEGIN
      name := 'Nancy' || name;
      balance := balance + 50 ;
   END;

   DECLARE
      new_name  VARCHAR2(20);
      new_balance NUMBER (7,2) ;
   BEGIN
      new_name := name;
      new_balance := balance + 100;
   END;

   fee := new_balance - balance;
   name := 'Sam ' || name;
END;
```

1 _____

2 _____

3 _____

4 _____

5 _____

Self Check – Answer Key

Exercise 1

Are the following statements legal or illegal? If illegal, specify why.

-
```
DECLARE
pidm NUMBER(9);
```

Legal.

-
```
DECLARE
amount, balance NUMBER(7,2) ;
```

Not legal. You cannot have more than one declaration per line.

-
```
DECLARE
registered BOOLEAN NOT NULL;
```

Not legal. You must initialize a variable that has a NOT NULL constraint.

-
```
DECLARE
institution VARCHAR2(20);
location  VARCHAR2;
```

Both are legal.

-
```
DECLARE
max_amount NUMBER CONSTANT;
```

Not legal. **You must initialize a CONSTANT**.

-
```
DECLARE
id      swriden.id%rowtype;
```

Not legal. **You can only use rowtype when declaring a table row not a single field.**

Continued on the next page

Self Check – Answer Key, Continued

Exercise 2 What are the values for each of the variables at the specified numbered steps?

```
DECLARE
   name  VARCHAR2(20) := 'Smith';
   balance NUMBER (7,2) := 100;
   fee NUMBER (7,2) := 0;
BEGIN
   DECLARE
      name VARCHAR2(20) := 'Johnson';
      balance NUMBER (7,2) := 200;
   BEGIN
      name := 'Nancy' || name;          [1]   Nancy Johnson
      balance := balance + 50 ;         [2]   250
   END;

   DECLARE
      new_name  VARCHAR2(20);
      new_balance NUMBER (7,2) ;
   BEGIN
      new_name := name;
      new_balance := balance + 100;     [3]   200
   END;

   fee := new_balance - balance;        [4]   Error

   name := 'Sam ' || name;              [5]   Sam Smith

END;
```

Section N: SQL Statements Within PL/SQL

Overview

Purpose

As mentioned earlier, PL/SQL is the procedural language that can enclose SQL statements with procedural statements. All data manipulation statements are supported by PL/SQL - SELECT, UPDATE, DELETE, INSERT. Data Definition statements (such as creating tables) are supported only in PL/SQL versions 2.0 and above through the use of the DBMS package.

Objectives

Upon completion of this section, each attendee will be able to:

- Use SQL statements within PL/SQL procedures
- Identify SQL functions supported by PL/SQL

In this section

These topics are covered in this section.

SQL Data manipulations

Version support All PL/SQL versions support SQL data manipulations.

INSERT

```
/* Inserts a record into the swbpers table*/
DECLARE
   pidm_in          NUMBER(8) := 1021;
   ssn_in           VARCHAR2(9) := '987654321';
   birth_date_in    DATE  := TO_DATE('10-FEB-
1973','DD-MON-YYYY');
   mrtl_code_in     VARCHAR2(1) := 'S';
   sex_in           VARCHAR2(1) := 'F';
   confid_ind_in    VARCHAR2(1) := NULL;
BEGIN
   INSERT INTO swbpers (pidm,ssn, birth_date,
                        mrtl_code, sex, confid_ind,
                          activity_date)
      VALUES (pidm_in, ssn_in, birth_date_in,
              mrtl_code_in, sex_in, confid_ind_in,
              SYSDATE);
END;
```

UPDATE

```
/* Updates the old zip code with the new zip code. */
   DECLARE
      zip_old VARCHAR2(10) := '19380';
      zip_new VARCHAR2(10) := '19382';

   BEGIN
      UPDATE swbaddr
         SET zip = zip_new
       WHERE zip = zip_old;
   END;
```

DELETE

```
/* Deletes all rows from the account table where the
term matches '199701' */
   DECLARE
      term VARCHAR2(6) := '199701';
   BEGIN
      DELETE FROM twraccd
            WHERE term_code < term;
   END;
```

Continued on the next page

SQL Data manipulations, Continued

DBMS_DDL

Until PL/SQL version 2.0, Data Definition Language statements were not allowed. This includes creating, altering, and dropping objects from the database. Such statements are made possible through the built-in package DBMS_DDL.

SELECT INTO

A SELECT INTO statement is the only DML that returns data. You will provide the host variable names for this data to be stored, specified in the INTO clause.

A SELECT INTO statement must return only one row. Zero or multiple returned rows returns an error. If you would like to return multiple rows, use cursors (discussed in Section P).

Syntax

```
SELECT  col1, col2, ...
  INTO  var1, var2 ...
  FROM  table_name
 WHERE  ...
```

Example

```
/* Retrieves the last and first name for PIDM 12340 into
the host variables of last_name_in and first_name_in.
*/
DECLARE
   last_name_in swriden.last_name%TYPE;
   first_name_in swriden.first_name%TYPE;
BEGIN
   SELECT last_name, first_name
     INTO last_name_in, first_name_in
     FROM swriden
    WHERE pidm = 12340
      AND change_ind IS NULL;

   -- data manipulation

END;
```

Process Transactions

SAVEPOINT and ROLLBACK TO

SAVEPOINT

- `SAVEPOINT <marker_name>;`

ROLLBACK TO

- `ROLLBACK [WORK] TO [SAVEPOINT] <marker_name>;`

Example

```
BEGIN
   INSERT INTO swriden (pidm, last_name,
                        first_name, activity_date)
      VALUES (pidm_sequence.NEXTVAL, 'Smith',
              'John', SYSDATE);

   SAVEPOINT A;

   INSERT INTO swriden (pidm, last_name,
                        first_name, activity_date)
      VALUES (pidm_sequence.NEXTVAL, 'Patterson',
              'Jim', SYSDATE);

   SAVEPOINT B;
   ...

   ROLLBACK TO SAVEPOINT A;

   COMMIT;
END;
```

Referencing SQL functions

Functions that can be referenced

- Numeric (ROUND, POWER, SQRT, etc.)
- Character (LENGTH, UPPER, etc.)
- DATE (MONTHS_BETWEEN, ADD_MONTHS, etc.)
- GROUP (AVG, MAX, COUNT, etc.)

Using functions within DML statements

```
SQL> INSERT INTO swriden (pidm, id ,last_name,
                          first_name, activity_date)
          VALUES (pidm_sequence.NEXTVAL, '56677G',
                  UPPER('&last'), UPPER('&first'),
                  SYSDATE);
```

Use functions in an Assignment statement

```
first_name_in := UPPER(first_name_in);
lag_time := MONTHS_BETWEEN (date_ordered,
                            date_shipped);
gpa_rounded := ROUND(gpa);
```

Effective coding style

- Indent three spaces
- Use uppercase for reserved words, and lowercase for user-defined objects and variables
- Put only one declaration per line, and only one statement per line
- For multi-line statements, align the subsequent lines offset from the first line
- Take advantage of whitespace
- Right-align reserved words in your SQL statements
- Use comments throughout your code

Section O: Conditional, Iterative, and Sequential Control

Overview

Purpose

Conditional control gives you the ability to direct the flow of execution based on a condition. You will use conditional control frequently throughout your PL/SQL coding. Sequential control, or the GOTO statement, will allow you to transfer control to another part of the program. Iterative control, or loops, allows you to execute the same code repeatedly.

Objectives

Upon completion of this section, each attendee will be able to:

- Demonstrate an understanding of the PL/SQL conditional control by writing a PL/SQL procedure to conditionally execute SQL statements
- Identify three uses for PL/SQL statement labels
- Identify the four types of loops supported by PL/SQL
- Demonstrate an understanding of PL/SQL iterative control by writing a simple PL/SQL procedure using a loop structure

In this section

These topics are covered in this section.

Logical comparisons

Logical comparisons

Logical comparisons form the basis of conditional control in PL/SQL. The results are always TRUE, FALSE, or NULL.

Null comparisons

- Anything compared with NULL results in a NULL value
- A NULL in an expression evaluates to NULL (except for concatenation)

Examples

- `10 - NULL` evaluates to `NULL`
- `'PL/SQL'||NULL||'is fun'` evaluates to `PL/SQL is fun`

Boolean operators

Boolean operators: AND, OR, and NOT

AND	TRUE	FALSE	NULL
TRUE	T	F	N
FALSE	F	F	F
NULL	N	F	N

OR	TRUE	FALSE	NULL
TRUE	T	T	T
FALSE	T	F	N
NULL	T	N	N

NOT	
TRUE	F
FALSE	T
NULL	N

Conditional control

IF-THEN

Use the IF-THEN construct when you want to execute one or more statements if the condition yields TRUE.

```
IF <condition>
THEN
   <TRUE sequence of statements>
END IF;
```

Example

```
IF average_gpa > 3.0 THEN
   student_status := 'HONORS';
END IF;
```

IF-THEN-ELSE

Use the IF-THEN-ELSE statement when you want to choose between two mutually exclusive actions.

```
IF <condition>
THEN
   <TRUE sequence of statements>
ELSE
   <FALSE or NULL sequence of statements>
END IF;
```

Example

```
IF average >= .70
THEN
Student status := 'PASSED';
ELSE
Student_status := 'FAILED';
END IF;
```

Continued on the next page

Conditional control, Continued

IF-THEN-ELSIF

Use the IF-THEN-ELSIF statement when you want to select an action from several mutually exclusive alternatives.

```
IF <condition>
THEN
   <TRUE sequence of statements>
ELSIF <condition>
THEN
   <TRUE sequence of statements>
ELSE
   <FALSE or NULL sequence of statements>
END IF;
```

Example

```
/* This block determines whether the institution
   size is small, medium, or large.  This is
   determined by of the number of records in the
   SWRIDEN table. */
DECLARE
   num_students NUMBER(10);
   institution_size VARCHAR2(7);
BEGIN
   SELECT COUNT(*)
     INTO num_students
     FROM swriden
    WHERE change_ind IS NULL;
   IF num_students < 5000
   THEN
      institution_size := 'Small';
   ELSIF num_students BETWEEN 5000 AND 14999
   THEN
      institution_size := 'Medium';
   ELSE
      institution_size := 'Large';
   END IF;
END;
```

Continued on the next page

Conditional control, Continued

Nested IF statements

You can nest any IF statement within any other IF statement.

```
IF <condition 1>
THEN
   IF <condition 2>
   THEN
       <statements 2>
   ELSE
      IF <condition 3>
      THEN
          <statements 3>
      ELSIF <condition 4>
      THEN
          <statements 4>
      END IF;
   END IF;
END IF;
```

Example

```
/* If the combined score is greater than 1200, the student's
scores are evaluated further.  If the student's verbal score
is greater than 600, a record is inserted into HIGH_VERBAL.
If the math score is greater than 600, then a record is
inserted in HIGH_MATH. */
    DECLARE
       pidm_in         NUMBER;
       id_in           VARCHAR2(9) := '123G';
       test_date_in    DATE := '10-MAY-97';
       sat_verbal_in   NUMBER(3);
       sat_math_in     NUMBER(3);
    BEGIN
       SELECT swrtest.pidm, sat_verbal, sat_math
         INTO pidm_in, sat_verbal_in, sat_math_in
         FROM swrtest, swriden
        WHERE swriden.pidm = swrtest.pidm
          AND swriden.id = id_in
          AND test_date = test_date_in;
       IF sat_verbal_in + sat_math_in > 1200 THEN
          IF sat_verbal_in > 600 THEN
             INSERT INTO HIGH_VERBAL
             (pidm, verbal_score, test_date)
                 VALUES (pidm_in, sat_verbal_in,
        test_date_in);
          END IF;
          IF sat_math_in > 600 THEN
  INSERT INTO HIGH_MATH (pidm, math_score,
         test_date)
                VALUES (pidm_in, sat_math_in, test_date_in);
          END IF;
       END IF;
    END;
```

Iterative control

Loops

Four types of loops:

- Simple Loops
- Numeric FOR Loops
- WHILE Loops
- Cursor FOR Loops

Simple loops

The loop is useful when you want to guarantee that the body (or part of body) executes at least one time. The simple loop will execute until an EXIT statement is executed and the value is TRUE.

```
LOOP
   <sequence of statements>
END LOOP;
```

Exit any loop immediately with the EXIT statement.

```
EXIT WHEN <condition>;
```

Example

```
DECLARE
   my_counter NUMBER := 1;
BEGIN
   LOOP
      INSERT INTO temp (col1)
         VALUES (my_counter);
      my_counter := my_counter + 1;
      EXIT WHEN my_counter > 10;
   END LOOP;
END;
```

Continued on the next page

Iterative control, Continued

Numeric FOR loops

Repeat a sequence of statements a fixed number of times with a Numeric FOR loop.

```
FOR <index> IN <low_value> .. <high_value>
LOOP
   <sequence of statements>
END LOOP;
```

Example

```
BEGIN
   FOR i IN 1..20 LOOP
      INSERT INTO temp (col1)
         VALUES (i);
   END LOOP;
END;
```

Loop index

The loop index takes on each value in the range, one at a time, in either forward or reverse order. The index is implicitly of type NUMBER, and cannot be reassigned within the loop. However, it may be used in an expression.

Examples

```
BEGIN
   FOR countdown IN REVERSE 10..20
   LOOP
      INSERT INTO temp (col1)
         VALUES (countdown);
      /*Counting down from 20 to 10 */
   END LOOP;
END;


DECLARE
   counter NUMBER;
BEGIN
   FOR my_index IN 1..20 LOOP
      INSERT INTO temp (col1)
         VALUES (my_index);
      my_index := my_index + 1;  /* illegal */
      counter := my_index;  /* legal */
   END LOOP;
END;
```

Continued on the next page

Iterative control, Continued

WHILE loops

Use the WHILE Loop when you want to repeat a sequence of statements until specific condition is no longer TRUE.

```
WHILE <condition>
LOOP
  <sequence of statements>
END LOOP;
```

Examples

```
DECLARE
   my_counter NUMBER:=1;

BEGIN
   WHILE my_counter < 50
   LOOP
      my_counter := my_counter + 1;
      INSERT INTO TEMP(col1)
         VALUES (my_counter);
   END LOOP;
END;

DECLARE
   end_of_program BOOLEAN:=FALSE;
   my_counter     NUMBER:=0;

BEGIN
   WHILE NOT end_of_program LOOP
      my_counter := my_counter + 1;
      INSERT INTO TEMP(col1)
         VALUES (my_counter);
      IF my_counter >= 20 THEN
         end_of_program := TRUE;
      END IF;
   END LOOP;
END;
```

Sequential control

The GOTO statement

The GOTO statement performs unconditional branching to a named label. The syntax for a GOTO statement is:

```
GOTO label_name;
```

The GOTO label is defined as follows:

```
<<label_name>>
```

Example

```
IF rating > 80 THEN
   GOTO calc_raise;
END IF;

<<calc_raise>>
IF job_title = 'SALESMAN' THEN
   amount := commission * 0.25;
ELSE
   amount := salary * 0.10;
END IF;
```

Restrictions

- At least one executable statement must follow a label
- The target label must be in the same scope as the GOTO statement
- The target label must be in the same part of the PL/SQL block as the GOTO statement (GOTO in body cannot go to a label in the exception handler)

Labelling blocks and loops

Labelling blocks and loops

In addition to using labels for GOTO statements, you can also use labels for blocks and loops.

Allow declared object referencing

Label a block to allow referencing of declared objects that would otherwise not be visible because of scoping rules.

```
<<label name>>
DECLARE
/* --declarations */
BEGIN
/*   --executable statements */
EXCEPTION
/*   --exception handler */
END label_name;  /* --must include label name */
```

Example

```
<<outer_block>>
DECLARE
  mypidm NUMBER := 1234;
BEGIN
  /* sub-block */
  DECLARE
     mypidm NUMBER := 432;
  BEGIN
    UPDATE swriden
       SET last_name = 'SMITH'
     WHERE mypidm in (mypidm, outer_block.mypidm);
  END;
  /* end of sub-block */
END outer_block;
```

Allow variable referencing

Label a block to allow a variable to be referenced that might be hidden by a column name.

Example

```
<<sample>>
DECLARE
   pidm swriden.pidm%TYPE := 12340;
BEGIN
   UPDATE swriden
      SET last_name = 'Smith'
    WHERE pidm = sample.pidm
      AND change_ind IS NULL;
   COMMIT;
END sample;
```

Continued on the next page

Labelling blocks and loops, Continued

Allow object referencing

Label a LOOP to allow an object to be referenced that would otherwise not be visible because of scoping rules.

Example

```
/* counter_loop */
DECLARE
   x NUMBER:=1;
BEGIN
   FOR x IN 1..20 LOOP
     DECLARE
         x NUMBER := 20;
     BEGIN
        INSERT INTO temp (col1, col2)
          VALUES (x, counter_loop.x);
        x := x - 1;
     END;
   END LOOP counter_loop; /* must include
                             loop name here */
END;
```

Specify exits from outer loops

Label an EXIT as a way to specify exits from outer loops.

```
DECLARE
   end_of_program BOOLEAN:=FALSE;
   x NUMBER:=1;
BEGIN
<<outer_loop>>
   WHILE NOT end_of_program LOOP;
      <<inner_loop>>
      FOR i in 1..10 LOOP
        EXIT outer_loop WHEN end_of_program = TRUE;
        EXIT inner_loop WHEN i > 10;
        x := x + 1;
        IF x > 9
        THEN
           end_of_program := TRUE;
        END IF;
      END LOOP inner_loop;
   END LOOP outer_loop;
END;
```

Self Check

Directions

Use the information you have learned in this workbook to complete this self check activity.

Exercise 1

Select the number of students who are in the **SWRIDEN** table into the host variable NUM_OF_STUDENTS. Make sure that you are only retrieving the most current row for each student.

In the number column of the TEMP_XX (where xx is your user number) table that you created earlier, insert the number of students. The message (MESSAGE column) is determined by the following:

- > 5 'More than 5 students'
- = 5 'Five students'
- < 5 'Less than 5 students'

Continued on the next page

Self Check, Continued

Exercise 2

Delete the contents of the TEMP_XX table.

Exercise 3

Create a **SIMPLE loop** which inserts your name into the MESSAGE column of the TEMP_XX table 5 times. In NUMBER, insert the number of iterations.

Exercise 4

Delete the contents of the TEMP_XX table.

Continued on the next page

Self Check, Continued

Exercise 5

Change your loop to a **Numeric FOR Loop.** Rather than iterating a specified number, allow the user to enter the amount of times the loop should iterate.

Exercise 6

Delete the contents of the TEMP_XX table.

Continued on the next page

Self Check, Continued

Exercise 7

Create an outer FOR loop which iterates 5 times, and an inner FOR loop which iterates 3 times. After each iteration of the inner loop, insert into the **TEMP_XX** table the index for the outer loop in NUMBER, and the index for the inner loop in TEXT. Label each loop.

Self Check – Answer Key

Exercise 1

Select the number of students who are in the **SWRIDEN** table into the host variable NUM_OF_STUDENTS. Make sure that you are only retrieving the most current row for each student.

In the number column of the TEMP_XX (where xx is your user number) table that you created earlier, insert the number of students. The message (MESSAGE column) is determined by the following:

- > 5 'More than 5 students'
- = 5 'Five students'
- < 5 'Less than 5 students'

```
DECLARE
   num_of_students NUMBER;
 BEGIN
   SELECT COUNT(*)
     INTO num_of_students
     FROM swriden
    WHERE change_ind IS NULL;
   IF num_of_students < 5 THEN
     INSERT INTO TEMP_XX (number, message)
        VALUES (num_of_students, 'Less than 5
   Students');
   ELSIF
      num_of_students = 5 THEN
      INSERT INTO TEMP_XX (number, message)
        VALUES (num_of_students, 'Five Students');
   ELSE
      INSERT INTO TEMP_XX (number, message)
        VALUES (num_of_students, 'More than 5
                Students');
   END IF;
 END;
```

(exercise continues)

Continued on the next page

Exercise 1 (cont.)

OR -- the better approach

```
DECLARE
    num_of_students NUMBER;
    my_message VARCHAR2(30);
  BEGIN
    SELECT COUNT(*)
      INTO num_of_students
      FROM swriden
     WHERE change_ind IS NULL;
    IF num_of_students < 5 THEN
      my_message :=  'Less than 5 Students';
    ELSIF num_of_students = 5 THEN
      my_message := 'Five Students';
    ELSE
      my_message := 'More than 5 Students';
    END IF;
    INSERT INTO TEMP_XX (number, message)
       VALUES (num_of_students, my_message);
  END;
```

Exercise 2

Delete the contents of the TEMP_XX table.

```
DELETE temp_xx;
```

Exercise 3

Create a **SIMPLE loop** which inserts your name into the MESSAGE column of the TEMP_XX table 5 times. In NUMBER, insert the number of iterations.

```
DECLARE
  my_counter NUMBER := 1;
BEGIN
LOOP
  EXIT WHEN my_counter > 5;
  INSERT INTO TEMP_XX (number, message)
     VALUES (my_counter, 'My Name');
  my_counter := my_counter + 1;
END LOOP;
END;
```

Exercise 4

Delete the contents of the TEMP_XX table.

```
DELETE temp_xx;
```

Continued on the next page

Self Check – Answer Key, Continued

Exercise 5

Change your loop to a **Numeric FOR Loop.** Rather than iterating a specified number, allow the user to enter the amount of times the loop should iterate.

```
BEGIN
  FOR my_counter IN 1..&n LOOP
   INSERT INTO TEMP_XX (number,message)
      VALUES (my_counter, 'My Name');
  END LOOP;
END;
```

Exercise 6

Delete the contents of the TEMP_XX table.

```
DELETE temp_xx;
```

Continued on the next page

Exercise 7

Create an outer FOR loop which iterates 5 times, and an inner FOR loop which iterates 3 times. After each iteration of the inner loop, insert into the **TEMP_XX** table the index for the outer loop in NUMBER, and the index for the inner loop in TEXT. Label each loop.

```
BEGIN
<<OUTER_LOOP>>
     FOR outer_index IN 1..5 LOOP
     <<INNER_LOOP>>
        FOR inner_index IN 1..3 LOOP
           INSERT INTO temp_xx (number, text)
           VALUES (outer_index, inner_index);
     END LOOP INNER_LOOP;
  END LOOP OUTER_LOOP;
END;
```

OR

```
BEGIN
<<OUTER_LOOP>>
     FOR i IN 1..5 LOOP
     <<INNER_LOOP>>
        FOR i IN 1..3 LOOP
           INSERT INTO temp_xx (number, text)
           VALUES (outer_loop.i, inner_loop.i);
     END LOOP INNER_LOOP;
  END LOOP OUTER_LOOP;
END;
```

Section P: Declare and Use Cursors

Overview

Purpose

A PL/SQL cursor allows you to fetch and process your data, one row at a time.

Objectives

Upon completion of this section, each attendee will be able to:

- Identify the two types of cursors supported by PL/SQL
- Outline the procedural steps for using explicit cursors in PL/SQL blocks
- Demonstrate comprehension of PL/SQL cursors by writing a simple PL/SQL procedure using explicit cursors
- Use implicit cursor attributes

In this section

These topics are covered in this section.

Cursor Basics

SQL statements with associated cursor

- INSERT
- UPDATE
- DELETE
- SELECT...INTO
- COMMIT
- ROLLBACK

Two types of cursors

EXPLICIT
- Multiple row SELECT Statements

IMPLICIT
- All INSERT Statements
- All UPDATE Statements
- All DELETE Statements
- Single row SELECT..INTO Statements

Handling multiple rows

The set of rows returned by a query can consist of zero, one, or many rows, depending upon the number of rows that meet the query's search condition.

When a query returns multiple rows, a cursor can be explicitly defined to:
- Process beyond the first row returned by the query
- Keep track of which rows are currently being processed

Cursor Operations

Steps

- Declare the cursor
- Open the cursor
- Fetch data from the cursor
- Close the cursor

Declaring cursors

```
DECLARE
   CURSOR <cursor name>
      IS <regular_select_statement>;
```

Declared cursors are scoped just as variables are.

```
DECLARE
   pidm_in NUMBER(8);

   CURSOR c1 IS
      SELECT pidm
        FROM swriden
       WHERE change_ind IS NULL;
BEGIN...
```

Opening cursors

Open the cursor to process the SELECT statement and store the returned rows in the cursor.

```
OPEN <cursor name>;
```

Example: `OPEN c1;`

- Evaluates the SELECT statement associated with the cursor
- Allocates the resources used by ORACLE to process the query
- Identifies the active set

Continued on the next page

Cursor Operations, Continued

Fetching from cursors

Fetch data from the cursor and store it in specified variables.

```
FETCH <cursor name> INTO <var1, var2...>;
```

Example: `FETCH c1 INTO pidm_in;`

There must be exactly one INTO variable for each column selected by the SELECT statement.

The first column gets assigned to var1, the second assigned to var2, etc.

Closing cursors

Close the cursor to free up resources.

```
CLOSE <cursor name>;
```

Example: `CLOSE c1;`

- Marks resources held by opened cursor as reusable
- No more rows can be fetched from a closed cursor

Cursor Attributes

%FOUND

After a cursor or cursor variable is opened but before the first fetch, %FOUND yields NULL. Thereafter, it yields TRUE if the last fetched returned a row, or FALSE if the last fetch failed to return a row.

```
WHILE swriden_cursor%FOUND LOOP
   FETCH swriden_cursor INTO last_name, first_name;
   /* --data processing here */
END LOOP;
```

%NOTFOUND

Logical opposite of %FOUND.

```
LOOP
  FETCH swriden_cursor INTO last_name, first_name;
  EXIT WHEN swriden_cursor%NOTFOUND;
  /* --data processing here */
END LOOP;
```

%ROWCOUNT

When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields 0. Thereafter, it yields the number of row fetched so far.

```
LOOP
  FETCH swriden_cursor INTO last_name_in,
      first_name_in;
  EXIT WHEN swriden_cursor%NOTFOUND OR
            swriden_cursor%ROWCOUNT > 100;
  /* --data processing here */
END LOOP;
```

Continued on the next page

Cursor Attributes, Continued

%ISOPEN

Yields TRUE if its cursor or cursor variable is open; otherwise, yields FALSE. Cursors must be closed before they can be reopened, so you can use this attribute to test whether the cursor is open or not.

```
IF swriden_cursor%ISOPEN THEN
   FETCH swriden_cursor INTO last_name_in,
    first_name_in;
ELSE
   OPEN swriden_cursor;
   FETCH swriden_cursor INTO last_name_in,
first_name_in;
END IF;
```

Example

```
/*For each student, the pidm and average gpa is
selected from the SWRREGS table.  The rows are ordered
so that the student with the highest average gpa is the
first row selected.  Each row is fetched and then
assigned a rank.  Then the PIDM and rank are inserted
into the SWRRANK table. */

   DECLARE
      pidm_in swriden.pidm%TYPE;
      gpa_in swrregs.gpa%TYPE;
      rank NUMBER := 0;
      CURSOR c1 IS
          SELECT pidm, AVG(gpa)
            FROM swrregs
        GROUP BY pidm
        ORDER BY AVG(gpa) DESC;
   BEGIN
      OPEN c1;
      LOOP
         FETCH c1 INTO pidm_in, gpa_in;
         EXIT WHEN c1%NOTFOUND;
         rank := rank + 1;
         INSERT INTO swrrank (pidm, class_rank)
            VALUES (pidm_in, rank);
      END LOOP;
   END;
```

Continued on the next page

Cursor Attributes, Continued

Referencing the current row

Reference the current row with the WHERE CURRENT OF statement.

```
WHERE CURRENT OF <cursor_name>
```

The cursor must be declared with a FOR UPDATE clause.

```
/*Purging block.  Retrieves each non-current row from SWRIDEN
and inserts it into SWRIDEN_HISTORY.  After the row has been
inserted, then the row from SWRIDEN is deleted.*/
    DECLARE
       swriden_row swriden%ROWTYPE;
       CURSOR c1 IS
          SELECT *
            FROM swriden
           WHERE change_ind IS NOT NULL
             FOR UPDATE;
    BEGIN
       OPEN c1;
       LOOP
          FETCH c1 INTO swriden_row;
          EXIT WHEN c1%NOTFOUND;
          INSERT INTO swriden_history  (pidm, id, last_name,
                                        first_name, change_ind)
              VALUES (swriden_row.pidm, swriden_row.id,
                      swriden_row.last_name,
                      swriden_row.first_name,
                      swriden_row.change_ind);
          DELETE FROM swriden
                WHERE CURRENT OF c1;
       END LOOP;
    END;
```

Cursor FOR Loops

Cursor FOR loops

Specify a sequence of statements to be repeated once for each row that is returned by the cursor with the Cursor FOR Loop.

```
FOR <record_name> IN <cursor_name> LOOP
   /* --statements to be repeated go here */
END LOOP;
```

- Cursor FOR loops are similar to Numeric FOR loops
- Cursor FOR loops specify a set of rows from a table using the cursor's name
- Numeric FOR loops specify an integer range
- A Cursor FOR loop record takes on the values of each rows
- A Numeric FOR loop index takes on each value in the range
- Record_name is implicitly declared as:

```
record_name cursor_name%ROWTYPE;
```

- To reference an element of the record, use the record_name.column_name notation

Conceptual Cursor loop model

- When a cursor is initiated, an implicit OPEN cursor_name is executed
- For each row that satisfies the query associated with the cursor, an implicit FETCH is executed into the components of record_name
- When there are no more rows left to FETCH, an implicit CLOSE cursor_name is executed and the loop is exited

```
DECLARE
   rank NUMBER := 0;
   CURSOR c1 IS
       SELECT pidm, AVG(gpa)
         FROM swrregs
     GROUP BY pidm
     ORDER BY AVG(gpa) DESC;
BEGIN
   /* --an implicit open is done here */
   FOR cursor_row IN c1 LOOP
      /* --an implicit fetch is done here */
      rank := rank + 1;
      INSERT INTO swrrank (pidm,class_rank)
         VALUES (cursor_row.pidm, rank);
   END LOOP; /* --an implicit close is done here */
END;
```

Declare cursors to use parameters

Pass parameters to cursors

Many times, you may want your cursor to be dependent on variables. The following examples show you how you can pass a parameter to a cursor.

```
DECLARE
   pidm_in swriden.pidm%TYPE;
   gpa_in swrregs.gpa%TYPE;
   rank NUMBER := 0;
   CURSOR c1 (id_in varchar2) IS
      SELECT swrregs.pidm, AVG(gpa)
        FROM swriden, swrregs
       WHERE swriden.pidm = swrregs.pidm
         AND id = id_in
         AND change_ind IS NULL
       GROUP BY swrregs.pidm ORDER BY AVG(gpa) DESC;
BEGIN
   OPEN c1 ('123G');
   LOOP
      FETCH c1 INTO pidm_in, gpa_in;
      EXIT WHEN c1%NOTFOUND;
      rank := rank + 1;
      INSERT INTO swrrank (pidm,class_rank)
         VALUES (pidm_in, rank);
   END LOOP;
END;
```

is the same as:

```
DECLARE
   id_in swriden.id%TYPE;
   pidm_in swriden.pidm%TYPE;
   gpa_in swrregs.gpa%TYPE;
   rank NUMBER := 0;
   CURSOR c1 IS
      SELECT swrregs.pidm, AVG(gpa)
        FROM swriden, swrregs
       WHERE swriden.pidm = swrregs.pidm
         AND id = id_in
         AND change_ind IS NULL
       GROUP BY swrregs.pidm ORDER BY AVG(gpa) DESC;
BEGIN
   id_in := '123G';
   OPEN c1;
   LOOP
      FETCH c1 INTO pidm_in, gpa_in;
      EXIT WHEN c1%NOTFOUND;
      rank := rank + 1;
      INSERT INTO swrrank (pidm,class_rank)
         VALUES (pidm_in, rank);
   END LOOP;
END;
```

Statements associated with Implicit Cursors

Associated statements

- All INSERT Statements
- All UPDATE Statements
- All DELETE Statements
- All SELECT..INTO Statements

Implicit cursors

An implicit cursor is called the 'SQL' cursor; it stores information concerning the processing of the last SQL statement not associated with an explicit cursor.

OPEN, FETCH, and CLOSE do not apply.

All cursor attributes apply.

SQL% NOTFOUND

SQL%NOTFOUND evaluates to TRUE if the most recently executed SQL statement affects no rows.

```
DECLARE
BEGIN
   UPDATE swbpers
      SET ssn = 123456789
    WHERE pidm = 12340;
   IF SQL%NOTFOUND THEN
     INSERT INTO swbpers (pidm, ssn, activity_date)
        VALUES (12340, 123456789, SYSDATE);
   END IF;
END;
```

SQL%FOUND

SQL%FOUND evaluates to TRUE if the most recently executed SQL statement affects one or more rows.

SQL% ROWCOUNT

SQL%ROWCOUNT evaluates to the number of rows affected by a DELETE, UPDATE, or INSERT.

```
DECLARE
BEGIN
   UPDATE sales_staff
      SET status = 'POOR'
    WHERE weekly_sales < 50;

   IF SQL%ROWCOUNT > 5 THEN
     INSERT INTO temp (message)
       VALUES ('**Warning - more than 5 salespeople
                  have a poor status');
   END IF;
END;
```

Self Check

Directions Use the information you have learned in this workbook to complete this self check activity.

Exercise 1 Create a cursor which selects the term, PIDM, and average GPA from **SWRREGS**, where the term is equal to a term code entered by the user (use the '&' variable). Insert the data into a table you create called **TERM_GPA** using the definition below.

```
pidm        not null      number(8)
term_code   not null      varchar2(6)
gpa         not null      number(4,2)
```

(Remember, you can create **TERM_GPA** based on **SWRREGS** using the 'AS SELECT' clause and only selecting the columns you need).

Fetch the data using a simple loop.

Continued on the next page

Self Check, Continued

Exercise 2 Repeat Exercise 1, but use a ***Cursor FOR Loop*** instead.

Self Check – Answer Key

Directions

Use the information you have learned in this workbook to complete this self check activity.

Exercise 1

Create a cursor which selects the term, PIDM, and average GPA from **SWRREGS**, where the term is equal to a term code entered by the user (use the '&' variable). Insert the data into a table you create called **TERM_GPA** using the definition below. (Remember, you can create **TERM_GPA** based on **SWRREGS** using the 'AS SELECT' clause and only selecting the columns you need). ***Fetch the data using a simple loop.***

```
TERM_GPA
pidm            NOT NULL    NUMBER(8)
term_code       NOT NULL    VARCHAR2(6)
gpa             NOT NULL    NUMBER(4,2)

/*Creates empty table term_gpa */
SQL> CREATE TABLE term_gpa
          AS SELECT pidm, term_code, gpa
        FROM swrregs
       WHERE 1=2;
/* no rows match, so the table is not populated */

/*Add NOT NULL constraint to GPA column */
SQL> ALTER TABLE term_gpa MODIFY gpa NOT NULL;

Table altered.
```

OR:

```
SQL> CREATE TABLE term_gpa
                (pidm NUMBER(8) NOT NULL,
          term_code VARCHAR2(6) NOT NULL,
                    gpa NUMBER(4,2) NOT NULL);
```

Continued on the next page

Self Check – Answer Key, Continued

Exercise 2

Repeat Exercise 1, but use a ***Cursor FOR loop instead***.

```
DECLARE
CURSOR C1 IS
      SELECT term_code, pidm, AVG(gpa) avg_gpa
        FROM swrregs
       WHERE term_code = '&term'
    GROUP BY term_code, pidm;
BEGIN
   FOR c1rec IN c1 LOOP
      INSERT INTO term_gpa (term_code, pidm,
                            avg_gpa)
         VALUES (c1rec.term_code,c1rec.pidm,
                 c1rec.avg_gpa);
   END LOOP;
   COMMIT;
END;
```

Section Q: Handle PL/SQL Errors

Overview

Purpose

In PL/SQL, errors are called exceptions. Whenever an error occurs in the PL/SQL body, ORACLE will jump to the exception area which specifies how to handle the error.

Objectives

Upon completion of this section, each attendee will be able to:

- Understand the advantages of exception handling
- Identify the different types of exception handling within PL/SQL
- Identify the functions supplied by PL/SQL that provide error information

In this section

These topics are covered in this section.

Exception Handling

Advantages of Exception Handling

- Event-driven handling of errors
- Separation of error-processing code
- Improved reliability of error handling

Types of Exceptions

- Named system exceptions
- Named programmer-defined exceptions
- Unnamed system exceptions
- Unnamed programmer-defined exceptions

Named system exceptions

Predefined exceptions

An ORACLE error 'raises' an exception automatically. Below are some common errors that have been defined for you.

Exception Name	Oracle Error	SQLCODE Value
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
STORAGE_ERROR	ORA-06500	-6500
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Notes

- INVALID CURSOR is raised if you try an illegal cursor operation. For example, INVALID_CURSOR is raised if you close an unopened cursor.
- INVALID_NUMBER is raised if the conversion of a character string to a number fails because the string does not represent a valid number.
- LOGIN_DENIED is raised if you try logging on to ORACLE with an invalid username/password.
- PROGRAM_ERROR is raised if PL/SQL has an internal problem.
- ROWTYPE_MISMATCH is raised if the host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types.
- STORAGE_ERROR is raised if PL/SQL runs out of memory or if memory is corrupted.
- TIMOUT_ON_RESOURCE is raised if a timeout occurs while Oracle is waiting for a resource.
- VALUE_ERROR is raised if an arithmetic, conversion, truncation, or constraint error occurs.
- NO_DATA_FOUND is raised if a SELECT INTO statement return no rows.
- TOO_MANY_ROWS is raised if a SELECT INTO statement returns more than one row.

Continued on the next page

Named system exceptions, Continued

Syntax

```
WHEN <exception name> [ OR <exception name> ....] THEN
      <sequence of statements>
    ...
[WHEN OTHERS THEN  --if used, must be the last handler
      <sequence of statements>]
```

Example

```
/*Retrieves the last name from swriden based on the ID
that the user enters.  If one row is successfully
retrieved, then the ID and last name are inserted into
the TEMP table.  If no rows or too many rows are found,
then the transaction is rolled back and a row
containing an error message is inserted into the TEMP
table.*/
   DECLARE
      student_lname swriden.last_name%TYPE;
      id_in swriden.id%TYPE;
   BEGIN
      id_in := &id;
      SELECT last_name
        INTO student_lname
        FROM swriden
       WHERE id = id_in
         AND change_ind IS NULL;
      INSERT INTO temp (col1, message)
         VALUES ('ID '||id_in ,student_lname);
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         ROLLBACK;
         INSERT INTO temp (message)
            VALUES ('ID '||id_in||' Not Found.');
         COMMIT;
      WHEN TOO_MANY_ROWS THEN
         ROLLBACK;
         INSERT INTO temp (message)
        VALUES ('More than one row found for
     ID '||id_in);
         COMMIT;
      WHEN OTHERS THEN
         ROLLBACK;
   END;
```

Named Programmer-Defined Exceptions

Syntax

```
DECLARE
   my_exception  EXCEPTION;
   ...
```

RAISE your exception

```
RAISE my_exception;
```

Notes

- Once an exception is raised manually, it is treated the same way as if it were a predefined internal exception
- Declared exceptions are scoped just like variables
- A user-defined exception is checked for manually and then raised, if appropriate

Continued on the next page

Named Programmer-Defined Exceptions, Continued

Example

```
DECLARE
   CURSOR C1 IS
      SELECT order_num, ship_date, order_date
        FROM orders;
   order_num_in NUMBER;
   order_date_in DATE;
   ship_date_in DATE;
   invalid_ship_date EXCEPTION;
      /* --user name exception */
BEGIN
   OPEN c1;
/*Exception is within a loop, so that if an order date
 is later than ship date, then the transaction is
 rolled back and the next row is fetched.  For any
 other error, the loop is exited because the error is
 handled outside the loop. */
   LOOP
      BEGIN
         EXIT WHEN c1%NOTFOUND;
         FETCH C1 INTO order_num_in, order_date_in,
 ship_date_in;
         IF order_date_in > ship_date_in THEN
           RAISE invalid_ship_date;
         END IF;
         UPDATE orders
            SET status = 'VALID'
          WHERE order_num = order_num_in;
         COMMIT;
      EXCEPTION
         WHEN invalid_ship_date THEN
            ROLLBACK;
            INSERT INTO temp (col1, message)
              VALUES (order_num_in, 'Order Date is
                     Later than Ship Date.');
            UPDATE orders
               SET status = 'INVALID'
             WHERE order_num = order_num_in;
            COMMIT;
      END;
   END LOOP;
EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK;
      INSERT INTO temp (message)
         VALUES ('ERROR: Routine Failed.');
      COMMIT;
END;
```

Exception Propagation

Steps

Step	Action
1	The current block is searched for a handler. If not found, then go to step 2.
2	If an enclosing block is found, it is searched for a handler.
3	Steps 1 and 2 are repeated until either there are no more enclosing blocks, or a handler is found. If there are no more enclosing blocks, the exception is passed back to the calling environment. If a handler is found, it is executed. When done, the block in which the handler was found is terminated, and control is passed to the enclosing block (if one exists), or to the environment (if there is no enclosing block).

Notes

- Only one handler per block may be active at a time
- If an exception is raised in a handler, the search for a handler for the new exception begins in the enclosing block of the current block

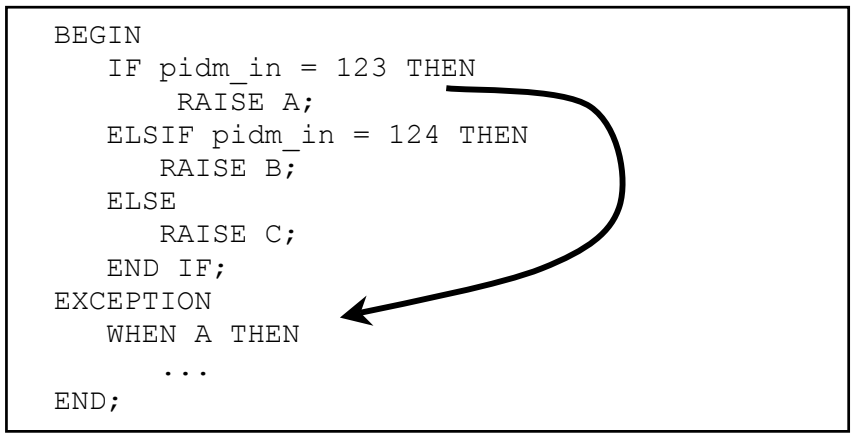
Continued on the next page

Exception Propagation, Continued

Example 1

Example 1 - When A is raised

```
BEGIN
...
   BEGIN
      IF pidm_in = 123 THEN
          RAISE A;
      ELSIF pidm_in = 124 THEN
         RAISE B;
      ELSE
         RAISE C;
      END IF;
   EXCEPTION
      WHEN A THEN
         ...
   END;
EXCEPTION
   WHEN B THEN
      ...
END;
```

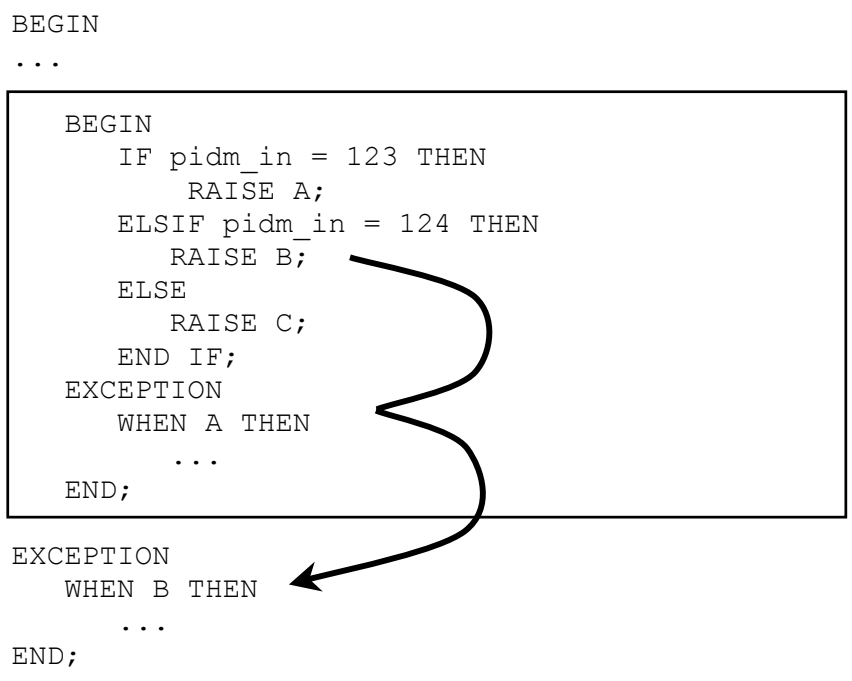


Continued on the next page

Exception Propagation, Continued

Example 2

Example 2 - When B is raised



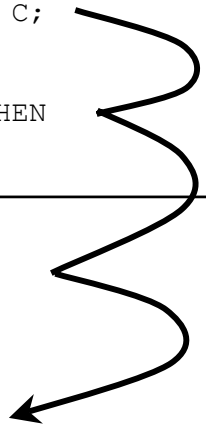
Continued on the next page

Exception Propagation, Continued

Example 3

Example 3 - When C is raised

```
BEGIN
...
   BEGIN
      IF pidm_in = 123 THEN
          RAISE A;
      ELSIF pidm_in = 124 THEN
         RAISE B;
      ELSE
         RAISE C;
      END IF ;
   EXCEPTION
      WHEN A THEN
         ...
   END;

EXCEPTION
   WHEN B THEN
      ...
END;
```



Exception C has no handler and will result in a runtime unhandled exception.

Unnamed System Exceptions

Unnamed exceptions

What do you do when you want to trap an error that has not been predefined? You will need to define the error yourself. Exceptions may only be handled by name (not ORACLE number). To handle undefined errors, use PRAGMA.

Syntax

```
PRAGMA EXCEPTION_INIT (<user_defined_exception_name>,
<ORACLE_error_number>);
```

Example

```
DECLARE
   invalid_session EXCEPTION;
   PRAGMA EXCEPTION_INIT (invalid_session, -22);
   sqlcode_val NUMBER;
   sqlerrm_val CHAR(55);
BEGIN
   INSERT INTO swriden (pidm, id, last_name,
 activity_date)
      VALUES (1234, 56, 'Peterson',SYSDATE);
EXCEPTION
  WHEN invalid_session THEN
     INSERT INTO TEMP(COL1, COL2, MESSAGE)
     VALUES (USER, TO_CHAR(SYSDATE),
'Invalid session ID.  Access Denied');
  WHEN OTHERS THEN
     ROLLBACK;
END;
```

SQLCODE and SQLERRM

Error information

The WHEN OTHERS exception handles all unspecified errors. However, even if you want generically handle all unspecified errors (such as rolling back changes made to the database), you will want to know what error occurred. SQLCODE and SQLERRM will provide error information to you.

SQLCODE

Returns the ORACLE error number of the exceptions, or 1 if it was a user-defined exception.

SQLERRM

Returns the ORACLE error message associated with the current value of SQLCODE. Can also use any ORACLE error number as an argument.

Notes

If no exception is active:

- SQLCODE = 0
- SQLERRM= 'normal, successful completion'

SQLCODE and SQLERRM cannot be used within an SQL statement.

Example

```
DECLARE
   invalid_session EXCEPTION;
   PRAGMA EXCEPTION_INIT (invalid_session, -22);
   sqlcode_val NUMBER;
   sqlerrm_val CHAR(55);
BEGIN
   INSERT INTO swriden (pidm, id, last_name,
 activity_date)
      VALUES (1234, 56, 'Peterson',SYSDATE);
EXCEPTION
  WHEN invalid_session THEN
     INSERT INTO TEMP(COL1, COL2, MESSAGE)
     VALUES (USER, TO_CHAR(SYSDATE),
'Invalid session ID.  Access Denied');
  WHEN OTHERS THEN
      ROLLBACK;
      sqlcode_val := SQLCODE;
      sqlerrm_val := SUBSTR(SQLERRM, 1, 55);
      INSERT INTO temp (col1, message)
         VALUES (sqlcode_val, sqlerrm_val);
      COMMIT;
END;
```

Sending output to the screen

DBMS_ OUTPUT

The traditional way to handle PL/SQL messages was to insert the information into a temporary table. However, with PL/SQL 2.0 and above, Oracle has provided a package called DBMS_OUTPUT that enables you to send output to the screen.

Enabling DBMS_ OUTPUT

For the package to be enabled, you first must type this in at the SQL prompt:

```
SET SERVEROUTPUT ON
```

You can also put this into your profile (login.sql) so that SERVEROUTPUT is on whenever you enter SQL*Plus.

Available output procedures

The following procedures are available for you for output:

Function/Procedure	Description
ENABLE	Enable message output
DISABLE	Disable message output
PUT_LINE	Place a line in the buffer
PUT	Place a partial line in the buffer
NEW_LINE	Terminate a line created with PUT
GET_LINE	Retrieve one line of information from buffer
GET_LINES	Retrieve array of lines from buffer

Continued on the next page

Sending output to the screen, Continued

Example

```
/*Handles the invalid session error differently from all
other errors.  Is not already predefined, so must be
declared. */
DECLARE
   invalid_session EXCEPTION;
   PRAGMA EXCEPTION_INIT (invalid_session, -22);
   sqlcode_val NUMBER;
   sqlerrm_val CHAR(55);
BEGIN
   DBMS_OUTPUT.ENABLE;
   INSERT INTO swriden (pidm, id, last_name,
 activity_date)
      VALUES (1234, 56, 'Peterson',SYSDATE);
EXCEPTION
  WHEN invalid_session THEN
      DBMS_OUTPUT.PUT_LINE
('Invalid session ID.  Access Denied.
        Contact Information Services at 123-4567');
  WHEN OTHERS THEN
      ROLLBACK;
      sqlcode_val := SQLCODE;
      sqlerrm_val := SUBSTR(SQLERRM, 1, 65);
      DBMS_OUTPUT.PUT_LINE(sqlcode_val||'
'||sqlerrm_val);
      COMMIT;
END;
```

You can also use the DBMS_OUTPUT package for basic reports.

Self Check

Directions

Use the information you have learned in this workbook to complete this self check activity.

Exercise 1

Create a PL/SQL block which fits the following scenario.

- Update the **SWBPERS** table with the following information below. If the update fails because the record does not exist, then insert the record into the table instead. In addition, check for all other errors (WHEN OTHERS) and display them on the screen (DBMS_OUTPUT) if an error occurs.

PIDM	**Based on user input**
Social Security No.	**123-45-3267**
Marital Code	**M**
Sex	**M**
Activity Date	**Current System Date**

Self Check – Answer Key

Exercise 1

Create a PL/SQL block which fits the following scenario.

- Update the **SWBPERS** table with the following information below. If the update fails because the record does not exist, then insert the record into the table instead. In addition, check for all other errors (WHEN OTHERS) and display them on the screen (DBMS_OUTPUT) if an error occurs.

PIDM	**Based on user input**
Social Security No.	**123-45-3267**
Marital Code	**M**
Sex	**M**
Activity Date	**Current System Date**

```
DECLARE
   ssn_in swbpers.ssn%type := '123453267';
   mrtl_code_in swbpers.mrtl_code%type := 'M';
   sex_in swbpers.sex%type := 'M';
   sqlcode_val NUMBER;
   sqlerrm_val CHAR(55);
BEGIN
   DBMS_OUTPUT.ENABLE;
   UPDATE swbpers
      SET ssn = ssn_in,
          mrtl_code = mrtl_code_in,
          sex = sex_in,
          activity_date = SYSDATE
    WHERE pidm = &&pidm;
    IF SQL%NOTFOUND THEN
      INSERT INTO SWBPERS (pidm, ssn, mrtl_code,
                           sex, activity_date)
         VALUES (&&pidm, ssn_in, mrtl_code_in,
                 sex_in, SYSDATE);
    END IF;
EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK;
      sqlcode_val := SQLCODE;
      sqlerrm_val := SUBSTR(SQLERRM, 1, 55);
      DBMS_OUTPUT.PUT_LINE(sqlcode_val||'
'||sqlerrm_val);
END;
```

Section R: Procedures and Functions

Overview

Purpose

We have created PL/SQL blocks which have produced results, but which oversimplifies the activities that you will face every day. Processes in your institution are most likely quite complicated, and you are probably wondering how what you learned so far will apply to the real world. Procedures and functions, which are pieces of callable code, will modularize your complicated processes into collections of simpler processes.

Objectives

Upon completion of this section, each attendee will be able to:

- Differentiate between a function and a procedure
- List the advantages of creating functions and procedures
- Create both a function and a procedure in the exercises

In this section

These topics are covered in this section.

Modular Code

Definition	Modularizing code is the process of breaking up large complicated processes into simpler blocks of code.
Advantages	• More reusable • More manageable • More readable • More reliable
Anonymous Block	An anonymous block is a PL/SQL block which has no name. This is the type of block that we have been using thus far.

Procedure

Definition

A procedure is a subprogram that performs a specific action. It returns no value.

Syntax

```
CREATE OR REPLACE PROCEDURE name [(parameter [,
parameter, ...])]
IS
    [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [name];
```

where parameter stands for the following syntax:

```
parameter_name [IN | OUT | IN OUT] datatype
[{:= | DEFAULT} expr]
```

Continued on the next page

Procedure, Continued

Example

```
/*Purging block.  Retrieves each non-current row from SWRIDEN
and inserts it into SWRIDEN_HISTORY.  After the row has been
inserted, then the row from SWRIDEN is deleted.*/
    CREATE OR REPLACE PROCEDURE purge_swriden IS
       swriden_row swriden%ROWTYPE;
       CURSOR c1 is
          SELECT *
            FROM swriden
           WHERE change_ind IS NOT NULL
             FOR UPDATE;
    BEGIN
       OPEN c1;
       LOOP
          FETCH c1 INTO swriden_row;
          EXIT WHEN c1%NOTFOUND;
          INSERT INTO swriden_history
         (pidm, id, last_name, first_name, change_ind)
        VALUES (swriden_row.pidm, swriden_row.id,
                swriden_row.last_name, swriden_row.first_name,
                swriden_row.change_ind);
          DELETE FROM swriden
             WHERE CURRENT OF c1;
       END LOOP;
    END purge_swriden;
```

Notes

- The keyword DECLARE is not used.
- A procedure has two parts - the specification and the body.
  - The procedure specification begins with the keyword PROCEDURE and ends with the procedure name or a parameter list.
  - The procedure body begins with the keyword IS and ends with the keyword END followed by an optional procedure name.
- Parameter declarations are optional.
- You cannot specify a constraint on parameter datatypes. For example, NUMBER(4) is illegal, but NUMBER is fine.

Function

Definition

A function is a subprogram that computes a value. Functions and procedures are structured alike, except that functions have a RETURN clause.

Syntax

```
CREATE OR REPLACE FUNCTION name [(parameter [,
parameter, ...])] RETURN datatype
   IS
       [local declarations]
   BEGIN
       executable statements
   [EXCEPTION
       exception handlers]
   END [name];
```

where parameter stands for the following syntax:

```
parameter_name [IN | OUT | IN OUT] datatype
[{:= | DEFAULT} expr]
```

Continued on the next page

Function, Continued

Example

```
/* Accepts the account id, and returns the account
balance. */
   CREATE OR REPLACE FUNCTION balance (acct_id NUMBER)
RETURN REAL
   IS
     acct_bal REAL;  /* Declare is not used */
   BEGIN
      SELECT bal
        INTO acct_bal
        FROM accts
       WHERE acctno = acct_id;
      RETURN acct_bal;
   END balance;
```

Notes

- The keyword DECLARE is not used.
- Like a procedure, a function has two parts - the specification and the body.
  - The function specification begins with the keyword FUNCTION and ends with RETURN clause.
  - The function body begins with the keyword IS and ends with the keyword END followed by an optional function name.
- Parameter declarations are optional.
- You cannot specify a constraint on parameter datatypes. For example, NUMBER(4) is illegal, but NUMBER is fine.
- The RETURN statement immediately completes the executions of a subprogram, whether it is the last statement or not.

Storing a procedure or function in the database

Storing syntax

To store a procedure or function, you will need to use the following syntax:

```
CREATE {OR REPLACE} PROCEDURE <procedure name> IS
   ...   -- body goes here

CREATE {OR REPLACE} FUNCTION <procedure name>
       RETURN <datatype> IS
   ...   -- body goes here
```

Handling compilation errors

If there were compilation errors, type the following to examine the errors:

```
SQL> SHOW ERRORS
```

You cannot easily fix the stored procedure or function directly when errors occur. Instead, you will rerun your file that originally created the procedure/function, after the errors have been corrected. The REPLACE option will allow the existing stored database procedure/function to be replaced.

Executing a procedure or function

Executing a procedure

A procedure returns no value, so it can be executed as follows:

```
EXECUTE <procedure name>
(<parameter1>,<parameter2>,etc.);
```

Executing a function

A function must be executed within a statement, because a value is returned:

```
my_variable := <function name>;
```

or

```
SELECT <function name>
(<parameter1>,<parameter2>,<etc.)
    FROM <table_name>;
```

Dropping a procedure or function

Syntax

```
SQL> DROP PROCEDURE <procedure name>;
SQL> DROP FUNCTION <function name>;
```

Example

In Section E, Exercise 6, we had created a select statement which would calculate a person's age. We now want to create a stored function so that we can call the function at any time to calculate a person's age.

```
/* Accepts the parameter of pidm_in and returns the age
calculated from SWBPERS where pidm is equal to pidm_in.
If no row is found for the pidm, then a message is
returned to indicate this.  If any other error occurs,
then the error number and error message are returned.
*/
   CREATE  OR REPLACE FUNCTION get_age (pidm_in IN
      NUMBER)
   RETURN VARCHAR2 IS age NUMBER(3);
   BEGIN
   SELECT TRUNC(MONTHS_BETWEEN(SYSDATE,birth_date)/12)
           INTO age
           FROM swbpers
          WHERE pidm = pidm_in;
   RETURN TO_CHAR(AGE);

   EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN 'No data in SWBPERS';
   WHEN  OTHERS THEN
      DECLARE
         sqlcode_val NUMBER(5);
         sqlerrm_val VARCHAR2(30);
   BEGIN
         sqlcode_val := SQLCODE;
         sqlerrm_val := SUBSTR(SQLERRM,1,30);
         RETURN sqlcode_val||sqlerrm_val;
      END;
   END GET_AGE;
```

Continued on the next page

Dropping a procedure or function, Continued

Executing the function

To execute the function, we can do the following:

```
SQL> SELECT get_age(12340) FROM DUAL;


GET_AGE(12340)
----------------------
24
```

Using with SELECT statement

Most likely, we will use the function within a SELECT statement that returns more than just the age:

```
SQL> SELECT first_name,last_name,get_age(pidm)
       FROM swriden
      WHERE change_ind IS NULL;


FIRST_NAME     LAST_NAME        GET_AGE(PIDM)
------------   ---------------  ----------------
Julie          Brown            24
Robert         Smith            26
Peter          Johnson      No data in SWBPERS
```

Self Check

Directions

Use the information you have learned in this workbook to complete this self check activity.

Exercise 1

Create a stored function named **F_GET_ADDRESS** which accepts the parameter of pidm, and returns the address (city, state, and zip concatenated) from **SWBADDR** in the format below:

- `Malvern, PA 19355`

If no record exists for the PIDM, use the predefined exception NO_DATA_FOUND to return the message 'No address exists for this PIDM'.

Continued on the next page

Self Check, Continued

Exercise 2

Create a stored procedure named **P_INSERT_ADDRESS** which accepts the parameter of ID. Using the stored function that you created above, select the PIDM from the **SWRIDEN** table. Here's an example of the syntax you will need in your PL/SQL block:

```
SELECT f_get_address(pidm)
  FROM swriden
 WHERE change_ind IS NULL;
```

Once you have retrieved the address, insert the PIDM into NUMBER, and the address into MESSAGE of the **TEMP_XX** table.

If no record exists in the **SWRIDEN** table for the ID that the user passes to the procedure, then display to the screen 'No swriden record exists for this ID.'. If there is more than one row in the **SWRIDEN** table, then display a message 'More than one swriden record exists for this ID.'. If any other error occurs, display the SQL code and message to the screen.

Continued on the next page

Exercise 3

Execute the procedure, passing the ID of '3539543'. Did the address get inserted into the TEMP_XX table?

Execute the procedure, passing the ID of '578549991', which is in the SWRIDEN table but not in the SWBADDR table. What happened?

Now pass an ID, which does not exist in the SWRIDEN table. Did your output work?

Self Check – Answer Key

Exercise 1

Create a stored function named **F_GET_ADDRESS** which accepts the parameter of pidm, and returns the address (city, state, and zip concatenated) from **SWBADDR** in the format below:

- `Malvern, PA 19355`

If no record exists for the PIDM, use the predefined exception NO_DATA_FOUND to return the message 'No address exists for this PIDM'.

```
CREATE OR REPLACE
FUNCTION F_GET_ADDRESS (pidm_in IN NUMBER)
   RETURN VARCHAR2 IS
   full_address VARCHAR2(50);
BEGIN
  SELECT CITY||', '||STAT_CODE||' '||ZIP
     INTO full_address
     FROM swbaddr
     WHERE pidm = pidm_in;
  RETURN full_address;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN ('No address exists for this PIDM.');
   WHEN OTHERS THEN
      DECLARE
         sqlcode_val NUMBER;
         sqlerrm_val CHAR(55);
      BEGIN
         sqlcode_val := SQLCODE;
         sqlerrm_val := SUBSTR(SQLERRM, 1, 40);
         RETURN (sqlcode_val||' '||sqlerrm_val);
      END;
END F_GET_ADDRESS;
```

Continued on the next page

Exercise 2

Create a stored procedure named **P_INSERT_ADDRESS** which accepts the parameter of ID. Using the stored function that you created above, select the PIDM from the **SWRIDEN** table. Here's an example of the syntax you will need in your PL/SQL block:

```
SELECT f_get_address(pidm)
  FROM swriden
 WHERE change_ind IS NULL;
```

Once you have retrieved the address, insert the PIDM into NUMBER, and the address into MESSAGE of the **TEMP_XX** table.

If no record exists in the **SWRIDEN** table for the ID that the user passes to the procedure, then display to the screen 'No swriden record exists for this ID.'. If there is more than one row in the **SWRIDEN** table, then display a message 'More than one swriden record exists for this ID.'. If any other error occurs, display the SQL code and message to the screen.

```
CREATE OR REPLACE PROCEDURE P_INSERT_ADDRESS (id_in
 IN VARCHAR2) IS
   pidm_in NUMBER(8);
   address_in VARCHAR2(50);
BEGIN
   DBMS_OUTPUT.ENABLE;
   SELECT pidm, f_get_address(pidm)
     INTO pidm_in, address_in
     FROM swriden
    WHERE change_ind IS NULL
      AND id = id_in;
   INSERT INTO temp_xx (col1, message)
       VALUES (pidm_in, address_in);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
     DBMS_OUTPUT.PUT_LINE('No swriden record exists for this ID.');
   WHEN TOO_MANY_ROWS THEN
     DBMS_OUTPUT.PUT_LINE('More than one swriden record exists for
this ID.');
   WHEN OTHERS THEN
   DECLARE
      sqlcode_val NUMBER;
      sqlerrm_val CHAR(55);
   BEGIN
      ROLLBACK;
      sqlcode_val := SQLCODE;
      sqlerrm_val := SUBSTR(SQLERRM, 1, 55);
      DBMS_OUTPUT.PUT_LINE(sqlcode_val||' '||sqlerrm_val);
   END;
END P_INSERT_ADDRESS
```

Continued on the next page

Exercise 3

Execute the procedure, passing the ID of '3539543'. Did the address get inserted into the TEMP_XX table?

```
SQL> EXECUTE P_INSERT_ADDRESS('3539543');
```

The pidm and address for ID '3539543' should have inserted into the TEMP table.

Execute the procedure, passing the ID of '578549991', which is in the SWRIDEN table but not in the SWBADDR table. What happened?

```
SQL> EXECUTE P_INSERT_ADDRESS('578549991');
```

The pidm and message 'No address exists for this ID' should be a row in the TEMP_XX table.

Now pass an ID which does not exist in the SWRIDEN table. Did your output work?

```
SQL> EXECUTE P_INSERT_ADDRESS('XXX');
```

A message should appear on the screen, 'No swriden record exists for this ID.'

Section S: Table Descriptions and Contents

Overview

In this section These topics are covered in this section.

SWBADDR

Description

```
SQL> DESC SWBADDR  - Student address & phone number table.

 Name                           Null?    Type
 ------------------------------ -------- ----
 PIDM                           NOT NULL NUMBER(8)
 STREET_LINE1                   NOT NULL VARCHAR2(30)
 STREET_LINE2                            VARCHAR2(30)
 CITY                                    VARCHAR2(20)
 STAT_CODE                               VARCHAR2(3)
 ZIP                                     VARCHAR2(10)
 PHONE_AREA                              VARCHAR2(3)
 PHONE_NUMBER                            VARCHAR2(7)
 ACTIVITY_DATE                  NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM SWBADDR;
 PIDM   STREET_LINE1     STREET_LINE2  CITY           STA ZIP    PHO PHONE_N  ACTIVITY_DA
-----   ------------     ------------  --------------  --- -----  --- -------  -----------
12340  506 BROWN STREET                WEST CHESTER    PA  19380  610 5624789  17-MAY-97
12341  210 PINE STREET                 SAN FRANCISCO   CA  94082  215 7954323  16-MAY-97
12342  PO BOX 1035       1200 ELM LANE BROWNVILLE      KY  67233  610 5624789  14-MAY-97
12343  23 MARKET STREET                WEST CHESTER    PA  19382  610 3246734  03-JUN-97
12344  18 CHESTNUT ROAD                NEW ORLEANS     LA  23456  850 6743213  07-JUN-97
```

SWBPERS

Listing

```
SQL> DESC SWBPERS  -Student profile table.

 Name                              Null?    Type
 --------------------------------- -------- ----
 PIDM                              NOT NULL NUMBER(8)
 SSN                                        VARCHAR2(9)
 BIRTH_DATE                                 DATE
 MRTL_CODE                                  VARCHAR2(1)
 SEX                                        VARCHAR2(1)
 CONFID_IND                                 VARCHAR2(1)
 ACTIVITY_DATE                     NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM SWBPERS;

     PIDM SSN       BIRTH_DAT M S C ACTIVITY_DA
--------- --------- --------- - - - -----------
    12340 585442212 02-AUG-73 S F Y 02-MAY-97
    12341 682082678 12-NOV-70 M M N 11-JUN-97
    12343 555444412 02-AUG-73 S F Y 06-JUN-97
    12344 198767345 02-AUG-73 S M N 09-JUN-97
    12345 955433412 05-JAN-75 M   N 08-JUN-97
```

SWRIDEN

Description

```
SQL> DESC SWRIDEN  - Student master identification table.

 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 PIDM                                      NOT NULL NUMBER(8)
 ID                                        NOT NULL VARCHAR2(9)
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 FIRST_NAME                                         VARCHAR2(15)
 MI                                                 VARCHAR2(15)
 CHANGE_IND                                         VARCHAR2(1)
 ACTIVITY_DATE                             NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM SWRIDEN;

  PIDM  ID        LAST_NAME           FIRST_NAME      MI        C ACTIVITY_
 ----- --------- ------------------- --------------- --------- - ---------
 12340 157834585 Brown               Julie           K           01-APR-02
 12340 176536782 Brown               Julie           K         I 23-MAR-02
 12341 5829934   Smith               Robert          E           28-MAR-02
 12342 3539543   Johnson             Peter           S           20-MAR-02
 12343 145672112 Jones               Sandy           J         N 03-MAR-02
 12343 145672112 Jones-Erickson      Sandy           J           02-APR-02
 12344 692568211 Erickson            Ralph           L           30-MAR-02
 12345 578549991 Erickson            Susan           T           18-MAR-02
 12346 543853339 White               Nancy           Carol       23-MAR-02
 12347 543853339 Marx                Joan            Elizabeth   03-MAR-02
 12348 543853339 Clifford            Stephanie       Geena       21-FEB-02
 12349 543853339 Serum               Tracy           Paige       23-MAR-02

12 rows selected.
```

SWRREGS

Description

```
SQL> DESC SWRREGS   --Student course registration table.

 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 PIDM                                      NOT NULL NUMBER(8)
 TERM_CODE                                 NOT NULL VARCHAR2(6)
 CRN                                       NOT NULL NUMBER(5)
 GPA                                                NUMBER(4,2)
 ACTIVITY_DATE                             NOT NULL DATE
```

Continued on the next page

SWRREGS, Continued

Contents

```
SQL> SELECT * FROM SWRREGS;

      PIDM TERM_C         CRN         GPA ACTIVITY_
---------- ------ ---------- ---------- ---------
     12340 199701      10001        3.2 02-APR-02
     12349 199701      10001        3.4 02-APR-02
     12349 199602      10007        3.1 02-APR-02
     12346 199602      10001        2.6 02-APR-02
     12348 199602      10001        2.9 02-APR-02
     12343 199602      10001          3 02-APR-02
     12340 199701      10007        2.1 02-APR-02
     12340 199602      10015        2.8 02-APR-02
     12340 199702      10017            02-APR-02
     12340 199702      10004            02-APR-02
     12340 199602      10008          2 02-APR-02
     12340 199602      10005          3 02-APR-02
     12340 199701      10009        3.2 02-APR-02
     12340 199701      10005        3.1 02-APR-02
     12341 199701      10014        3.2 02-APR-02
     12341 199602      10018        2.4 02-APR-02
     12341 199701      10023        1.2 02-APR-02
     12341 199701      10024        3.1 02-APR-02
     12342 199702      10021            02-APR-02
     12342 199702      10020            02-APR-02
     12342 199701      10013        2.6 02-APR-02
     12342 199701      10008        3.3 02-APR-02
     12342 199701      10011        3.3 02-APR-02
     12342 199701      10002        3.4 02-APR-02
     12342 199602      10003        2.3 02-APR-02
     12342 199602      10006        2.8 02-APR-02
     12342 199602      10009        2.9 02-APR-02
     12342 199701      10006        2.3 02-APR-02
     12343 199701      10008        3.8 02-APR-02
     12343 199602      10009          4 02-APR-02
     12343 199701      10011        1.6 02-APR-02
     12344 199702      10012            02-APR-02
     12344 199702      10019            02-APR-02
     12344 199701      10004        3.6 02-APR-02
     12345 199601      10003        2.2 02-APR-02
     12345 199602      10004        2.8 02-APR-02
     12345 199602      10011        2.8 02-APR-02
     12345 199701      10003        3.9 02-APR-02
     12345 199701      10009          3 02-APR-02
     12345 199601      10002        2.8 02-APR-02
     12346 199602      10016        1.1 02-APR-02
     12346 199702      10005            02-APR-02
     12346 199601      10015        3.5 02-APR-02
     12346 199601      10024        3.6 02-APR-02

44 rows selected.
```

SWRSTDN

Description

```
SQL> DESC SWRSTDN  - Student standing table.

 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 PIDM                                      NOT NULL NUMBER(8)
 STDN_CODE                                 NOT NULL VARCHAR2(2)
 STDN_DATE                                 NOT NULL DATE
 ACTIVITY_DATE                             NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM SWRSTDN;

      PIDM ST STDN_DATE ACTIVITY_
---------- -- --------- ---------
     12340 GR 06-JUN-01 06-JUN-01
     12341 GR 06-JUN-01 06-JUN-01
     12342 GH 06-JUN-01 06-JUN-01
     12343 PB 02-APR-02 02-APR-02
     12344 SS 02-APR-02 02-APR-02
     12345 GS 02-APR-02 02-APR-02
     12346 HS 02-APR-02 02-APR-02

7 rows selected.
```

SWRTEST

Description

```
SQL> DESC SWRTEST  - Student test score table.

 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 PIDM                                      NOT NULL NUMBER(8)
 TEST_DATE                                 NOT NULL DATE
 SAT_VERBAL                                         NUMBER(3)
 SAT_MATH                                           NUMBER(3)
 ACTIVITY_DATE                             NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM swrtest;

    PIDM TEST_DATE SAT_VERBAL   SAT_MATH ACTIVITY_
-------- --------- ---------- ---------- ---------
   12340 01-MAR-97        550        480 11-FEB-02
   12341 03-MAR-97        530        580 11-FEB-02
   12342 13-FEB-97        660        520 11-FEB-02
   12341 08-JUN-97        590        610 03-MAR-02
   12343 03-FEB-97        530        420 22-JAN-02
   12344 08-OCT-96        370        420 11-FEB-02
   12345 23-MAR-02        590        620 11-FEB-02
   12346 03-MAR-02        630        590 11-FEB-02
   12346 23-MAR-02        520        460 31-MAR-02
   12347 23-MAR-02        520            31-MAR-02
10 rows selected.
```

SWVCRSE

Description

```
SQL> DESC SWVCRSE  --Course validation table.

 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 CRN                                       NOT NULL NUMBER(5)
 DESCRIPTION                               NOT NULL VARCHAR2(30)
 CREDIT_HOURS                                       NUMBER (3)
 ACTIVITY_DATE                             NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM SWVCRSE;

  CRN DESCRIPTION                    CREDIT_HOURS ACTIVITY_
----- ------------------------------ ------------ ---------
10001 Writing                                   3 02-APR-02
10002 European History                          4 02-APR-02
10003 Algebra                                   4 02-APR-02
10004 Physics                                   3 02-APR-02
10005 Biology                                   2 02-APR-02
10006 Zoology                                   3 02-APR-02
10007 Philosophy                                3 02-APR-02
10008 Psychology                                3 02-APR-02
10009 Calculus                                  2 02-APR-02
10010 Literature                                2 02-APR-02
10011 Anthropology                              4 02-APR-02
10012 Statistics                                3 02-APR-02
10013 Oil Painting                              4 02-APR-02
10014 Pottery                                   3 02-APR-02
10015 Speech                                    3 02-APR-02
10016 C Programming                             4 02-APR-02
10017 Management Information Systems            3 02-APR-02
10018 Tennis                                    3 02-APR-02
10019 Golf                                      3 02-APR-02
10020 Swimming                                  3 02-APR-02
10021 Economics                                 3 02-APR-02
10022 Accounting                                3 02-APR-02
10023 Geometry                                  3 02-APR-02
10024 Photography                               3 02-APR-02
24 rows selected.
```

SWVSTDN

Description

```
SQL> DESC SWVSTDN  - Validation table for standing codes.

 Name                           Null?    Type
 ------------------------------ -------- ----
 STDN_CODE                      NOT NULL VARCHAR2(2)
 DESCRIPTION                             VARCHAR2(30)
 ACTIVITY_DATE                  NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM SWVSTDN;

ST DESCRIPTION                     ACTIVITY_
-- ------------------------------ ---------
GS Good Standing                   02-APR-02
HS Honor Student                   02-APR-02
PB Probation                       02-APR-02
SS Suspended                       02-APR-02
GR Graduate                        02-APR-02
GH Graduate with honors            02-APR-02

6 rows selected.
```

SWVTERM

Description

```
SQL> DESC SWVTERM   - Validation table for term code.

 Name                           Null?    Type
 ------------------------------ -------- ----
 TERM_CODE                      NOT NULL VARCHAR2(6)
 DESCRIPTION                             VARCHAR2(30)
 ACTIVITY_DATE                  NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM SWVTERM;


TERM_C DESCRIPTION                    ACTIVITY_
------ ------------------------------ ---------
199701 Spring Semester 1997           02-APR-02
199702 Spring Semester 1997           02-APR-02
199705 Summer Semester 1997           02-APR-02
199708 Fall Semester 1997             02-APR-02
199808 Fall Semester 1998             02-APR-02
199802 Spring Semester 1998           02-APR-02
199805 Summer Semester 1998           02-APR-02
199808 Fall Semester 1998             02-APR-02
199901 Spring Semester 1999           02-APR-02
199905 Summer Semester 1999           02-APR-02
199905 Spring Semester 1999           02-APR-02
199908 Fall Semester 1999             02-APR-02
200001 Spring Semester 2000           02-APR-02
200005 Summer Semester 2000           02-APR-02
200008 Fall Semester 2000             02-APR-02
199602 Spring Semester 1996           02-APR-02
199702 Spring Semester 1997           02-APR-02

17 rows selected.
```

TWRACCD

Description

```
SQL> DESC TWRACCD   --Account detail table.

 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 PIDM                                      NOT NULL NUMBER(8)
 TERM_CODE                                 NOT NULL VARCHAR2(6)
 DETC_CODE                                 NOT NULL VARCHAR2(4)
 TRANS_TYPE                                NOT NULL VARCHAR2(1)
 BILL_DATE                                 NOT NULL DATE
 AMOUNT                                    NOT NULL NUMBER(7,2)
 BALANCE                                            NUMBER(7,2)
 ACTIVITY_DATE                             NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM TWRACCD;

   PIDM TERM_C DETC T BILL_DATE     AMOUNT    BALANCE ACTIVITY_
------- ------ ---- - --------- ---------- ---------- ---------
  12340 199701 TUIT C 01-DEC-97     1500.5     1500.5 02-APR-02
  12340 199701 BOOK C 01-FEB-97      300.2      300.2 02-APR-02
  12340 199701 BOOK P 23-JUN-97        700       -700 02-APR-02
  12341 199701 TUIT C 06-MAR-97       1100       1100 02-APR-02
  12341 199701 DORM C 15-APR-97        500        500 02-APR-02
  12341 199802 CHEK P 01-OCT-98       1000      -1000 02-APR-02
  12342 199801 TUIT C 01-JUL-98       1300       1200 02-APR-02
  12342 199801 LABS C 02-APR-02         50         50 02-APR-02
  12342 199702 MEAL C 02-APR-02        800        800 02-APR-02
  12343 199701 TUIT C 02-APR-02        800        800 02-APR-02
  12343 199701 FAID P 30-MAR-02       1100      -1100 02-APR-02
  12344 199702 TUIT C 02-APR-02        750        750 02-APR-02
  12344 199702 BOOK C 02-APR-02        400        400 02-APR-02
  12344 199602 LABS C 10-SEP-96        120        120 02-APR-02
  12344 199701 MEAL C 02-APR-02        900        900 02-APR-02
  12344 199602 DORM C 02-APR-02       1000       1000 02-APR-02
  12344 199701 CASH P 06-APR-02        800       -800 02-APR-02
  12344 199701 CRED P 02-APR-02        400       -400 02-APR-02
  12345 199701 TUIT C 02-APR-02        300        300 02-APR-02
  12346 199701 TUIT C 02-APR-02        900        900 02-APR-02
  12346 199701 LABS C 02-APR-02         50         50 02-APR-02
  12346 199701 CHEK P 02-APR-02        950       -950 02-APR-02
22 rows selected.
```

TWVDETC

Description

```
SQL> DESC TWVDETC - Detail code validation table.
 Name                           Null?    Type
 ------------------------------ -------- ----
 DETC_CODE                      NOT NULL VARCHAR2(4)
 DESCRIPTION                             VARCHAR2(30)
 ACTIVITY_DATE                  NOT NULL DATE
```

Contents

```
SQL> SELECT * FROM TWVDETC;

DETC DESCRIPTION                    ACTIVITY_
---- ------------------------------ ---------
TUIT Tuition Charges                02-APR-02
BOOK Book Charges                   02-APR-02
DORM Dorm Charges                   02-APR-02
MEAL Meal Plan Charges              02-APR-02
LABS Lab Fee Charges                02-APR-02
CHEK Check Payment                  02-APR-02
CASH Cash Payment                   02-APR-02
CRED Credit Card Payment            02-APR-02
FAID Financial Aid Payment          02-APR-02

9 rows selected.
```

Section T: Related Files

Overview

In this section

These topics are covered in this section.

Create_exercise_tables.sql

Listing

```
set feedback off
prompt Setting up tables...please wait.
--PERSON INDENTIFICATION TABLE
DROP TABLE SWRIDEN;
CREATE TABLE SWRIDEN
(PIDM NUMBER(8)    NOT NULL,
 ID    VARCHAR2(9) NOT NULL,
 LAST_NAME   VARCHAR2(25) NOT NULL,
 FIRST_NAME  VARCHAR2(15),
 MI      VARCHAR2(15),
 CHANGE_IND  VARCHAR2(1),
 ACTIVITY_DATE DATE NOT NULL);
--
INSERT INTO SWRIDEN VALUES (12340,'157834585','Brown','Julie','K',NULL,SYSDATE-1);
INSERT INTO SWRIDEN VALUES (12340,'176536782','Brown','Julie','K','I',SYSDATE-10);
INSERT INTO SWRIDEN VALUES (12341,'5829934','Smith','Robert','E',NULL,SYSDATE-5);
INSERT INTO SWRIDEN VALUES (12342,'3539543','Johnson','Peter','S',NULL,SYSDATE-13);
INSERT INTO SWRIDEN VALUES (12343,'145672112','Jones','Sandy','J','N',SYSDATE-30);
INSERT INTO SWRIDEN VALUES (12343,'145672112','Jones-Erickson','Sandy','J',NULL,SYSDATE);
INSERT INTO SWRIDEN VALUES (12344,'692568211','Erickson','Ralph','L',NULL,SYSDATE-3);
INSERT INTO SWRIDEN VALUES (12345,'578549991','Erickson','Susan','T',NULL,SYSDATE-15);
INSERT INTO SWRIDEN VALUES (12346,'543853339','White','Nancy','Carol',NULL,SYSDATE-10);
INSERT INTO SWRIDEN VALUES (12347,'543853339','Marx','Joan','Elizabeth',NULL,SYSDATE-30);
INSERT INTO SWRIDEN VALUES
(12348,'543853339','Clifford','Stephanie','Geena',NULL,SYSDATE-40);
INSERT INTO SWRIDEN VALUES (12349,'543853339','Serum','Tracy','Paige',NULL,SYSDATE-10);
--
--
--
--SWRIDEN_HISTORY TABLE
-- This is also an example of how we can create a blank table from one that exists
DROP TABLE SWRIDEN_HISTORY;
CREATE TABLE SWRIDEN_HISTORY
 AS SELECT * FROM SWRIDEN
     WHERE 1=2;

--The 1 = 2 is never true and thus the table creates without any records
--
--
--
--GENERAL PERSON TABLE
DROP TABLE SWBPERS;
CREATE TABLE SWBPERS
(PIDM             NUMBER(8)   NOT NULL PRIMARY KEY,
 SSN              VARCHAR2(9),
 BIRTH_DATE       DATE,
 MRTL_CODE        VARCHAR2(1),
 SEX              VARCHAR2(1),
 CONFID_IND       VARCHAR2(1),
 ACTIVITY_DATE   DATE         NOT NULL);
--
INSERT INTO SWBPERS VALUES (12340,'585442212','02-AUG-1973','S','F','Y',SYSDATE-50);
INSERT INTO SWBPERS VALUES (12341,'682082678','12-NOV-1970','M','M','N',SYSDATE-10);
INSERT INTO SWBPERS VALUES (12343,'555444412','02-AUG-1973','S','F','Y',SYSDATE-15);
INSERT INTO SWBPERS VALUES (12344,'198767345','02-AUG-1973','S','M','N',SYSDATE-12);
INSERT INTO SWBPERS VALUES (12345,'955433412','05-JAN-1954','M',NULL,'Y',SYSDATE-13);
```

Continued on the next page

Create_exercise_tables.sql, Continued

Listing (cont.)

```
--
--PERSON ADDRESS TABLE
--
DROP TABLE SWBADDR;
CREATE TABLE SWBADDR
(PIDM            NUMBER(8)                  NOT NULL,
 STREET_LINE1     VARCHAR2(30)          NOT NULL,
 STREET_LINE2     VARCHAR2(30),
 STREET_LINE3     VARCHAR2(30),
 CITY             VARCHAR2(20),
 STAT_CODE        VARCHAR2(3),
 ZIP              VARCHAR2(10),
 PHONE_AREA       VARCHAR2(3),
 PHONE_NUMBER     VARCHAR2(7),
 ACTIVITY_DATE    DATE               NOT NULL);
--
INSERT INTO SWBADDR VALUES
(12340,'506 BROWN STREET',NULL,NULL,'WEST CHESTER','PA','19380',
'610','5624789',SYSDATE-35);
INSERT INTO SWBADDR VALUES
(12341,'210 PINE STREET',NULL,NULL,'SAN FRANCISCO','CA','94082',
'215','7954323',SYSDATE-36);
INSERT INTO SWBADDR VALUES
(12342,'PO BOX 1035','1200 ELM LANE',NULL,'BROWNVILLE','KY','67233',
'610','5624789',SYSDATE-38);
INSERT INTO SWBADDR VALUES
(12343,'23 MARKET STREET',NULL,NULL,'WEST CHESTER','PA','19382',
'610','3246734',SYSDATE-18);
INSERT INTO SWBADDR VALUES
(12344,'18 CHESTNUT ROAD',NULL,NULL,'NEW ORLEANS','LA','23456',
'850','6743213',SYSDATE-14);
--
--TELEPHONE NUMBER VIEW
CREATE OR REPLACE VIEW SWVTELE
(PIDM, NAME, PHONE)
AS
SELECT SWRIDEN.PIDM,
       LAST_NAME||', '||FIRST_NAME||' 'MI,
       NULL||'('||PHONE_AREA||') '||SUBSTR(PHONE_NUMBER,1,3)||'-'||
       SUBSTR(PHONE_NUMBER,4,4)
  FROM SWRIDEN, SWBADDR
 WHERE SWRIDEN.PIDM = SWBADDR.PIDM
   AND CHANGE_IND IS NULL;
--
--ACCOUNT TRANSACTION TABLE
DROP TABLE TWRACCD;
CREATE TABLE TWRACCD
(PIDM                NUMBER(8)             NOT NULL,
 TERM_CODE           VARCHAR2(6)           NOT NULL,
 DETC_CODE           VARCHAR2(4)           NOT NULL,
 TRANS_TYPE          VARCHAR2(1)           NOT NULL,
 BILL_DATE           DATE                  NOT NULL,
 AMOUNT              NUMBER(7,2)           NOT NULL,
 BALANCE             NUMBER(7,2),
 ACTIVITY_DATE DATE                        NOT NULL);
```

Continued on the next page

Create_exercise_tables.sql, Continued

Listing (cont.)

```
--
INSERT INTO TWRACCD VALUES
(12340,'199701','TUIT','C','01-DEC-1997',1500.50,1500.50,SYSDATE);
INSERT INTO TWRACCD VALUES
(12340,'199701','BOOK','C','01-FEB-1997',300.20,300.20,SYSDATE);
INSERT INTO TWRACCD VALUES
(12340,'199701','BOOK','P','23-JUN-1997',700.00,-700.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12341,'199701','TUIT','C','06-MAR-1997',1100.00,1100.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12341,'199701','DORM','C','15-APR-1997',500.00,500.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12341,'199802','CHEK','P','01-OCT-1998',1000.00,-1000.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12342,'199801','TUIT','C','01-JUL-1998',1300.00,1200.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12342,'199801','LABS','C',SYSDATE,50.00,50.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12342,'199702','MEAL','C',SYSDATE,800.00,800.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12343,'199701','TUIT','C',SYSDATE,800.00,800.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12343,'199701','FAID','P',SYSDATE-3,1100.00,-1100.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12344,'199702','TUIT','C',SYSDATE,750.00,750.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12344,'199702','BOOK','C',SYSDATE,400.00,400.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12344,'199602','LABS','C','10-SEP-1996',120.00,120.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12344,'199701','MEAL','C',SYSDATE,900.00,900.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12344,'199602','DORM','C',SYSDATE,1000.00,1000.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12344,'199701','CASH','P',SYSDATE+4,800.00,-800.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12344,'199701','CRED','P',SYSDATE,400.00,-400.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12345,'199701','TUIT','C',SYSDATE,300.00,300.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12346,'199701','TUIT','C',SYSDATE,900.00,900.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12346,'199701','LABS','C',SYSDATE,50.00,50.00,SYSDATE);
INSERT INTO TWRACCD VALUES
(12346,'199701','CHEK','P',SYSDATE,950.00,-950.00,SYSDATE);
--
-- TRANSACTION TYPE DETAIL CODE TABLE
DROP TABLE TWVDETC;
CREATE TABLE TWVDETC
(DETC_CODE            VARCHAR2(4)           NOT NULL,
 DESCRIPTION          VARCHAR2(30),
 ACTIVITY_DATE        DATE                  NOT NULL);
--
INSERT INTO TWVDETC VALUES ('TUIT', 'Tuition Charges', SYSDATE);
INSERT INTO TWVDETC VALUES ('BOOK', 'Book Charges', SYSDATE);
INSERT INTO TWVDETC VALUES ('DORM', 'Dorm Charges', SYSDATE);
INSERT INTO TWVDETC VALUES ('MEAL', 'Meal Plan Charges', SYSDATE);
INSERT INTO TWVDETC VALUES ('LABS', 'Lab Fee Charges', SYSDATE);
INSERT INTO TWVDETC VALUES ('CHEK', 'Check Payment', SYSDATE);
INSERT INTO TWVDETC VALUES ('CASH', 'Cash Payment', SYSDATE);
INSERT INTO TWVDETC VALUES ('CRED', 'Credit Card Payment', SYSDATE);
 INSERT INTO TWVDETC VALUES ('FAID', 'Financial Aid Payment', SYSDATE);
```

Continued on the next page

Create_exercise_tables.sql, Continued

Listing (cont.)

```
--
-- TERM CODE TABLE
DROP TABLE SWVTERM;
CREATE TABLE SWVTERM
(TERM_CODE             VARCHAR2(6)            NOT NULL,
 DESCRIPTION           VARCHAR2(30),
 ACTIVITY_DATE         DATE                   NOT NULL);
--
INSERT INTO SWVTERM VALUES ('199701', 'Spring Semester 1997', SYSDATE);
INSERT INTO SWVTERM VALUES ('199702', 'Spring Semester 1997', SYSDATE);
INSERT INTO SWVTERM VALUES ('199705', 'Summer Semester 1997', SYSDATE);
INSERT INTO SWVTERM VALUES ('199708', 'Fall Semester 1997', SYSDATE);
INSERT INTO SWVTERM VALUES ('199808', 'Fall Semester 1998', SYSDATE);
INSERT INTO SWVTERM VALUES ('199802', 'Spring Semester 1998', SYSDATE);
INSERT INTO SWVTERM VALUES ('199805', 'Summer Semester 1998', SYSDATE);
INSERT INTO SWVTERM VALUES ('199808', 'Fall Semester 1998', SYSDATE);
INSERT INTO SWVTERM VALUES ('199901', 'Spring Semester 1999', SYSDATE);
INSERT INTO SWVTERM VALUES ('199905', 'Summer Semester 1999', SYSDATE);
INSERT INTO SWVTERM VALUES ('199905', 'Spring Semester 1999', SYSDATE);
INSERT INTO SWVTERM VALUES ('199908', 'Fall Semester 1999', SYSDATE);
INSERT INTO SWVTERM VALUES ('200001', 'Spring Semester 2000', SYSDATE);
INSERT INTO SWVTERM VALUES ('200005', 'Summer Semester 2000', SYSDATE);
INSERT INTO SWVTERM VALUES ('200008', 'Fall Semester 2000', SYSDATE);
INSERT INTO SWVTERM VALUES ('199602', 'Spring Semester 1996', SYSDATE);
INSERT INTO SWVTERM VALUES ('199702', 'Spring Semester 1997', SYSDATE);
--
-- CLASS REGISTRATION TABLE
DROP TABLE SWRREGS;
CREATE TABLE SWRREGS
(PIDM                  NUMBER(8)              NOT NULL,
 TERM_CODE             VARCHAR2(6)            NOT NULL,
 CRN                   NUMBER(5)              NOT NULL,
 GPA                   NUMBER(4,2),
 ACTIVITY_DATE         DATE                   NOT NULL);
--
INSERT INTO SWRREGS VALUES(12340,'199701',10001,3.2,SYSDATE);
INSERT INTO SWRREGS VALUES(12349,'199701',10001,3.4,SYSDATE);
INSERT INTO SWRREGS VALUES(12349,'199602',10007,3.1,SYSDATE);
INSERT INTO SWRREGS VALUES(12346,'199602',10001,2.6,SYSDATE);
INSERT INTO SWRREGS VALUES(12348,'199602',10001,2.9,SYSDATE);
INSERT INTO SWRREGS VALUES(12343,'199602',10001,3.0,SYSDATE);
INSERT INTO SWRREGS VALUES(12340,'199701',10007,2.1,SYSDATE);
INSERT INTO SWRREGS VALUES(12340,'199602',10015,2.8,SYSDATE);
INSERT INTO SWRREGS VALUES(12340,'199702',10017,NULL,SYSDATE);
INSERT INTO SWRREGS VALUES(12340,'199702',10004,NULL,SYSDATE);
INSERT INTO SWRREGS VALUES(12340,'199602',10008,2.0,SYSDATE);
INSERT INTO SWRREGS VALUES(12340,'199602',10005,3.0,SYSDATE);
INSERT INTO SWRREGS VALUES(12340,'199701',10009,3.2,SYSDATE);
INSERT INTO SWRREGS VALUES(12340,'199701',10005,3.1,SYSDATE);
INSERT INTO SWRREGS VALUES(12341,'199701',10014,3.2,SYSDATE);
INSERT INTO SWRREGS VALUES(12341,'199602',10018,2.4,SYSDATE);
INSERT INTO SWRREGS VALUES(12341,'199701',10023,1.2,SYSDATE);
INSERT INTO SWRREGS VALUES(12341,'199701',10024,3.1,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199702',10021,NULL,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199702',10020,NULL,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199701',10013,2.6,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199701',10008,3.3,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199701',10011,3.3,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199701',10002,3.4,SYSDATE);
```

Continued on the next page

Create_exercise_tables.sql, Continued

Listing (cont.)

```
INSERT INTO SWRREGS VALUES(12342,'199602',10003,2.3,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199602',10006,2.8,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199602',10009,2.9,SYSDATE);
INSERT INTO SWRREGS VALUES(12342,'199701',10006,2.3,SYSDATE);
INSERT INTO SWRREGS VALUES(12343,'199701',10008,3.8,SYSDATE);
INSERT INTO SWRREGS VALUES(12343,'199602',10009,4.0,SYSDATE);
INSERT INTO SWRREGS VALUES(12343,'199701',10011,1.6,SYSDATE);
INSERT INTO SWRREGS VALUES(12344,'199702',10012,NULL,SYSDATE);
INSERT INTO SWRREGS VALUES(12344,'199702',10019,NULL,SYSDATE);
INSERT INTO SWRREGS VALUES(12344,'199701',10004,3.6,SYSDATE);
INSERT INTO SWRREGS VALUES(12345,'199601',10003,2.2,SYSDATE);
INSERT INTO SWRREGS VALUES(12345,'199602',10004,2.8,SYSDATE);
INSERT INTO SWRREGS VALUES(12345,'199602',10011,2.8,SYSDATE);
INSERT INTO SWRREGS VALUES(12345,'199701',10003,3.9,SYSDATE);
INSERT INTO SWRREGS VALUES(12345,'199701',10009,3.0,SYSDATE);
INSERT INTO SWRREGS VALUES(12345,'199601',10002,2.8,SYSDATE);
INSERT INTO SWRREGS VALUES(12346,'199602',10016,1.1,SYSDATE);
INSERT INTO SWRREGS VALUES(12346,'199702',10005,NULL,SYSDATE);
INSERT INTO SWRREGS VALUES(12346,'199601',10015,3.5,SYSDATE);
INSERT INTO SWRREGS VALUES(12346,'199601',10024,3.6,SYSDATE);
--
-- COURSE TABLE
DROP TABLE SWVCRSE;
CREATE TABLE SWVCRSE
(CRN           NUMBER(5) NOT NULL,
 DESCRIPTION   VARCHAR2(30)   NOT NULL,
 CREDIT_HOURS     NUMBER(3),
 ACTIVITY_DATE DATE NOT NULL);
--
INSERT INTO SWVCRSE VALUES (10001, 'Writing', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10002, 'European History', 4,SYSDATE);
INSERT INTO SWVCRSE VALUES (10003, 'Algebra', 4,SYSDATE);
INSERT INTO SWVCRSE VALUES (10004, 'Physics', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10005, 'Biology', 2,SYSDATE);
INSERT INTO SWVCRSE VALUES (10006, 'Zoology', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10007, 'Philosophy', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10008, 'Psychology', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10009, 'Calculus', 2,SYSDATE);
INSERT INTO SWVCRSE VALUES (10010, 'Literature', 2,SYSDATE);
INSERT INTO SWVCRSE VALUES (10011, 'Anthropology', 4,SYSDATE);
INSERT INTO SWVCRSE VALUES (10012, 'Statistics', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10013, 'Oil Painting', 4,SYSDATE);
INSERT INTO SWVCRSE VALUES (10014, 'Pottery', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10015, 'Speech', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10016, 'C Programming', 4,SYSDATE);
INSERT INTO SWVCRSE VALUES (10017, 'Management Information Systems',
3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10018, 'Tennis', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10019, 'Golf', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10020, 'Swimming', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10021, 'Economics', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10022, 'Accounting', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10023, 'Geometry', 3,SYSDATE);
INSERT INTO SWVCRSE VALUES (10024, 'Photography', 3,SYSDATE);
--
-- STUDENT STANDING TABLE
DROP TABLE SWRSTDN;
CREATE TABLE SWRSTDN
(PIDM  NUMBER(8) NOT NULL,
 STDN_CODE VARCHAR2(2) NOT NULL,
 STDN_DATE DATE NOT NULL,
 ACTIVITY_DATE DATE NOT NULL);
```

Continued on the next page

Create_exercise_tables.sql, Continued

Listing (cont.)

```
--
INSERT INTO SWRSTDN VALUES(12340,'GR',SYSDATE-300,SYSDATE-300);
INSERT INTO SWRSTDN VALUES(12341,'GR',SYSDATE-300,SYSDATE-300);
INSERT INTO SWRSTDN VALUES(12342,'GH',SYSDATE-300,SYSDATE-300);
INSERT INTO SWRSTDN VALUES(12343,'PB',SYSDATE,SYSDATE);
INSERT INTO SWRSTDN VALUES(12344,'SS',SYSDATE,SYSDATE);
INSERT INTO SWRSTDN VALUES(12345,'GS',SYSDATE,SYSDATE);
INSERT INTO SWRSTDN VALUES(12346,'HS',SYSDATE,SYSDATE);
--
-- STUDENT STANDING CODE TABLE
DROP TABLE SWVSTDN;
CREATE TABLE SWVSTDN
(STDN_CODE     VARCHAR2(2) NOT NULL,
 DESCRIPTION   VARCHAR2(30),
 ACTIVITY_DATE DATE    NOT NULL);
--
INSERT INTO SWVSTDN VALUES ('GS', 'Good Standing', SYSDATE);
INSERT INTO SWVSTDN VALUES ('HS', 'Honor Student', SYSDATE);
INSERT INTO SWVSTDN VALUES ('PB', 'Probation', SYSDATE);
INSERT INTO SWVSTDN VALUES ('SS', 'Suspended', SYSDATE);
INSERT INTO SWVSTDN VALUES ('GR', 'Graduate', SYSDATE);
INSERT INTO SWVSTDN VALUES ('GH', 'Graduate with honors', SYSDATE);
--
-- STUDENT TEST TABLE
DROP TABLE SWRTEST;
CREATE TABLE SWRTEST
(PIDM NUMBER(8) NOT NULL,
 TEST_DATE DATE NOT NULL,
 SAT_VERBAL NUMBER(3),
 SAT_MATH NUMBER(3),
 ACTIVITY_DATE DATE NOT NULL);
--
INSERT INTO SWRTEST VALUES(12340,'01-MAR-1997',550,480,SYSDATE-50);
INSERT INTO SWRTEST VALUES(12341,'03-MAR-1997',530,580,SYSDATE-50);
INSERT INTO SWRTEST VALUES(12342,'13-FEB-1997',660,520,SYSDATE-50);
INSERT INTO SWRTEST VALUES(12341,'08-JUN-1997',590,610,SYSDATE-30);
INSERT INTO SWRTEST VALUES(12343,'03-FEB-1997',530,420,SYSDATE-70);
INSERT INTO SWRTEST VALUES(12344,'08-OCT-1996',370,420,SYSDATE-50);
INSERT INTO SWRTEST VALUES(12345,SYSDATE-10,590,620,SYSDATE-50);
INSERT INTO SWRTEST VALUES(12346,SYSDATE-30,630,590,SYSDATE-50);
INSERT INTO SWRTEST VALUES(12346,SYSDATE-10,520,460,SYSDATE-2);
DROP TABLE TEMP;
CREATE TABLE TEMP
(COL1 NUMBER(8),
 COL2 VARCHAR2(15),
 COL3 DATE,
 MESSAGE VARCHAR2(60));
DROP TABLE HIGH_VERBAL;
CREATE TABLE HIGH_VERBAL
(PIDM NUMBER(8) NOT NULL,
 VERBAL_SCORE NUMBER(3),
 TEST_DATE DATE NOT NULL);

DROP TABLE HIGH_MATH;
CREATE TABLE HIGH_MATH
 (PIDM NUMBER(8) NOT NULL,
  MATH_SCORE NUMBER(3),
  TEST_DATE DATE NOT NULL);
prompt The tables have been successfully created and populated.
set feedback on
```

swriden.ctl

Listing

```
LOAD DATA
INFILE 'swriden.dat'
BADFILE 'swriden.bad'
DISCARDFILE 'swriden.dsc'
APPEND
INTO TABLE swriden
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(pidm      SEQUENCE(MAX, 1),
 activity_date  SYSDATE,
 id             CHAR,
 last_name       CHAR,
 first_name     CHAR,
 mi             CHAR,
 change_ind      CHAR)
```

swriden.dat

Listing

```
443223344,Robertson,Robert,R,
433256789,Thompson,Thomas,T,I
433234566,Thompson,Thomas,T,
66755334533,Appleton,Apple,A,N
667553345,Thompson,Apple,A,
657890007,Jackson,Steve,D,
543678890,Hedges,Mike,R
453678923,Palm,Royal,Z
```