ORACLE SQL*Plus:
An Introduction and Tutorial

Richard Holowczak

Prof. Richard Holowczak
Department of Statistics and Computer Information Systems
Baruch College, City University of New York (Box E-0435)
E-Mail: richard_holowczak@baruch.cuny.edu

Contents

- 1. Introduction
- 2. Oracle Products: An Overview
  - 2.1 Application Development Tools
  - 2.2 Database Utilities
  - 2.3 Connectivity Products
  - 2.4 Core Database Engine
- 3. SQL*Plus Basics
  - 3.1 Running SQL*Plus
  - 3.2 SQL*Plus Commands
  - 3.3 SQL*Plus Help Facilities
- 4. The SQL Language
  - 4.1 SQL Statements
  - 4.2 SQL Data Definition Language
    - 4.2.1 Create, Modify and Drop Tables, Views, Indexes and Sequences
    - 4.2.2 GRANT and REVOKE Statements
    - 4.2.3 Synonyms
  - 4.3 SQL Data Manipulation Language
    - 4.3.1 Select, Insert, Update, Delete, Commit and Rollback Data
    - 4.3.2 Displaying Table Metadata
    - 4.3.3 Oracle Pseudo-Columns
    - 4.3.4 Oracle SQL Functions
    - 4.3.5 Examples of SQL DML Statements
      - Basic Select Statements
      - Selecting from 2 or More Tables
      - Recursive Queries and Table Aliases
      - Tree Queries
      - WHERE Clause Expressions
      - Examples of SQL Functions
- 5. Advanced SQL*Plus Commands

    - 5.1 Editing The SQL Buffer
    - 5.2 Formatting SQL*Plus Output
    - 5.3 Saving SQL*Plus Output using the SPOOL Command
    - 5.4 Prompting and Accepting user Input
    - 5.5 Collecting Statistics On SQL Statements
    - 5.6 Using the ALTER SESSION Statement for Date Formats
- 6. Stored Procedures and Triggers
    - 6.1 Trigger Syntax
    - 6.2 Trigger Example
    - 6.3 Stored Procedure Example
- 7. Acknowledgements

---

1. Introduction

The Oracle Relational Database Management System (RDBMS) is an industry leading database system designed for mission critical data storage and retrieval. The RDBMS is responsible for accurately storing data and efficiently retrieving that data in response to user queries.

The Oracle Corporation also supplies interface tools to access data stored in an Oracle database. Two of these tools are known as SQL*Plus, a command line interface, and Developer/2000 (now called simply *Developer*), a collection of forms, reports and graphics interfaces. This technical working paper introduces the features of the SQL*Plus tool and provides a tutorial that demonstrates its salient features.

This tutorial is intended for students and database practitioners who require an introduction to SQL, an introduction to working with the Oracle SQL*Plus tool, or both.

This document is organized as follows. A brief overview of the suite of Oracle products is first presented in Section 2. In Section 3, we discuss the basics of working with the SQL*Plus tool. Structured Query Language (SQL), including data definition language (DDL) and data manipulation language (DML) is discussed in section 4. Advanced SQL*Plus commands are discussed in section 5 and a brief introduction to stored procedures and triggers is given in section 6.

---

2. Oracle Products: An Overview

The Oracle products suite includes the following tools and utilities:

2.1 Application Development Tools

- **SQL*Plus** - A command line tool used to manipulate tables and other database objects in an Oracle database.
- **Developer/2000 and Developer** A suite of application development tools including Forms, Reports and Graphics.
    - **Oracle*Forms** - A screen based tool used to develop data entry forms and menus that access tables in an Oracle database.
    - **Oracle*Reports** - A screen based tool used to develop reports that access tables in an Oracle database.
    - **Oracle*Graphics** - A graphical tool used to develop charts and reports that access tables in an Oracle database.
- **CASE*Designer and Oracle Designer/2000** - A graphical tool used to create and display models contained in the CASE*Dictionary.
- **CASE*Dictionary** - A repository for business rules, functional models and data models used for organizing and documenting an application development effort.
- **CASE*Generator** - A code generating tool that uses information stored in CASE*Dictionary to develop data entry forms, reports and graphics.
- **Oracle*Book** - A graphical tool used to develop on-line documentation with hypertext capabilities.
- **SQL*TextRetrieval and Oracle Context** - A suite of tools and API used to develop sophisticated text search and retrieval applications.
- **Programmer/2000** - Including the Pro* precompilers - Libraries of routines and utilities that can be linked with ``C'', C++, FORTRAN, Java, ADA, COBOL or other host languages to allow access to Oracle databases.

2.2 Database Utilities

- **Enterprise Manager** - A GUI based collection of utilities for managing Oracle Databases.
- **SQL*DBA and SVRMGR** - A utility that allows the database administrator (DBA) to monitor database activity and to tune the database for optimal performance.
- **Export/Import** - Command line utilities that allow a user or the DBA to export data from an Oracle database into a machine readable file or to import data from a machine readable file into an Oracle database.
- **SQL*Loader** - A command line utility to load ASCII or binary data files into an Oracle database.
- **Oracle*Terminal** - A utility program used to customize the user interface and keyboard mappings for all Oracle tools. This utility allows all Oracle tools to have a similar ``look and feel'' across many different hardware and operating system platforms.

2.3 Connectivity and Middleware Products

- **SQL*Net and Net8** - A communications driver that allows an Oracle tool running on a client machine to access Oracle data on a separate server machine.
- **SQL*Connect and Oracle Gateways** - A communications driver that allows an Oracle tool running on a client machine to access Non-Oracle data on a server machine such as data residing in a DB2 database or MS SQL Server database.

- **ORACLE Server** - Typically a part of the Oracle RDBMS running on a database server, this component receives requests from client machines and submits them to the Oracle RDBMS. The results are then passed back to the client machines.
- **Oracle ODBC Drivers** - Open DataBase Connectivity drivers for connecting software to Oracle databases using the ODBC standard.

2.4 Core Database Engine

- **ORACLE RDBMS** - The Oracle Relational Database Engine. Now called the Oracle Universal Server with several options in addition to managing relational data. These options are now called *Cartridges*:
    - **Oracle Web Applications Server** - A WWW Server (HTTP server) linked into the Oracle RDBMS. Allows web based applications using HTML forms and JAVA to access and manipulate data.
    - **Spatial Data Cartridge**- Allows storage of temporal and spatial data in the Oracle RDBMS. Useful for Geographic Information Systems (GIS).
    - **Video Cartridge** - Provides storage and real-time serving of streaming video.
    - **ConText Cartridge** - Provides storage and retrieval of text documents.
    - **Messaging Option** - A groupware architecture built on top of the RDBMS.
    - **OLAP Option** - Tools and database support for On-Line Analytical Processing.
    - **Objects Option** - Allows complex objects to be modeled and stored in the DBMS. Includes Object Oriented features such as encapsulation, inheritance, server and client side methods, etc.
- **Integrated Data Dictionary** - Stores and manages access to all of the tables owned by all users in a system.
- **SQL** - The language used to access and manipulate database data.
- **PL/SQL** - A procedural extension to the SQL language unique to the Oracle line of products.

2.5 Typical Development Environments

Developing applications using an Oracle database requires access to a copy of the Oracle RDBMS (or a central Oracle RDBMS server), and one or more of the development tools. Third party development tools such as PowerBuilder, Visual Basic or Java can also be used for applications development.

Stand-alone development in a single user environment can be accomplished using the Personal Oracle or Personal Oracle Lite RDBMS in conjunction with Oralce Developer or a third party development tool.

Muli-user development in a shared environment can be accomplished using an Oracle RDBMS server running on a server machine. Distributed client PCs can develop the applications using any of the tools mentioned above.

Regardless of the development environment, used, the Oracle SQL*Plus utility is a convenient and capable tool for manipulating data in an Oracle database. In the following section, the SQL*Plus tool is introduced.

3. SQL*Plus Basics

Oracle's SQL*Plus is a command line tool that allows a user to type SQL statements to be executed directly against an Oracle database. SQL*Plus has the ability to format database output, save often used commands and can be invoked from other Oracle tools or from the operating system prompt.

In the following sections, the basic functionality of SQL*Plus will be demonstrated along with sample input and output to demonstrate some of the many features of this product.

3.1 Running SQL*Plus

In this section, we give some general directions on how to get into the SQL*Plus program and connect to an Oracle database. Specific instructions for your installation may vary depending on the version of SQL*Plus being used, whether or not SQL*Net or Net8 is in use, etc.

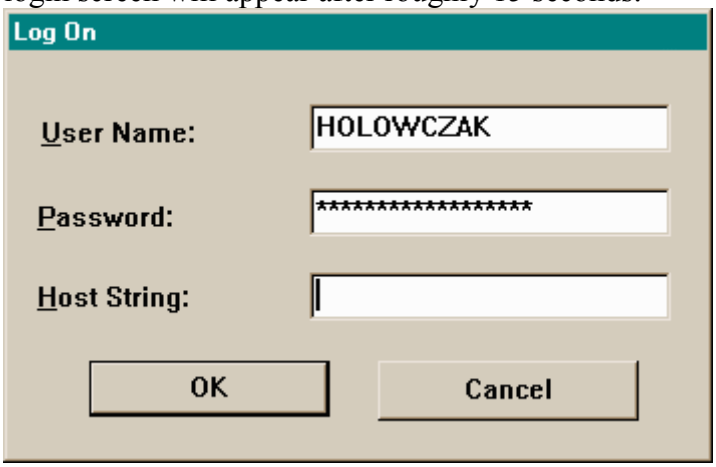
Before using the SQL*Plus tool or any other development tool or utility, the user must obtain an Oracle account for the DBMS. This account will include a username, a password and, optionally, a host string indicating the database to connect to. This information can typically be obtained from the database administrator.

The following directions apply to two commonly found installations: Windows 95/98 or NT client with an Oracle server, and a UNIX installation.

3.1.1 Running SQL*Plus under Windows 95/98 and Windows NT

To run the SQL*Plus command line program from **Windows 95/98** or **Windows NT**, click on the 

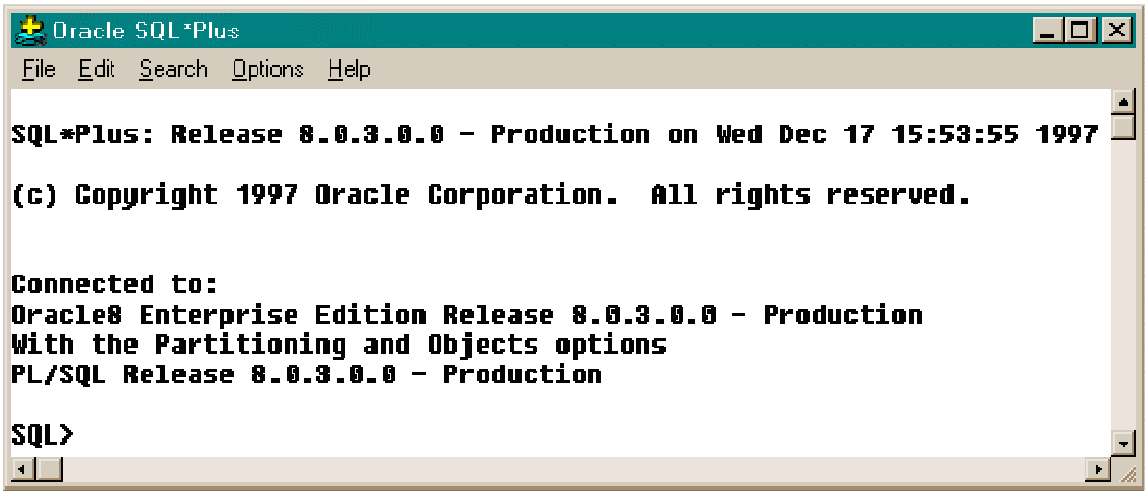
button, Programs, Oracle for Windows 95 and then SQL*Plus. The SQL*Plus login screen will appear after roughly 15 seconds.



In the **User Name:** field, type in your Oracle username.
Press the TAB key to move to the next field.
In the **Password:** field, type your Oracle password.

Press the TAB key to move to the next field.
In the **Host String:** field, type in the *Service Name* of the Oracle host to connect to. If the DBMS is Personal Oracle lite then this string might be ODBC:POLITE. If the DBMS is Personal Oracle8, then the host string might be beq-local. For Client/Server installations with SQL*Net or Net8, this string will be the service name set up by the SQL*Net assistant software.

Finally, click on the OK button to complete the Oracle log in process. SQL*Plus will then establish a *session* with the Oracle DBMS and the SQL*Plus prompt (SQL>) will appear. The following figure shows the results of logging into Oracle using SQL*Plus:



There are a number of situations in which an error may occur:

- You might mistype your username, password and/or the Host String
- SQL*Plus and SQL*Net may not be configured properly on your Windows client.
- The network between your Windows client and the Oracle server may have a problem
- The Oracle server may be temporarily shut down or otherwise unavailable

In any of the above cases, an error message will be returned. If the Oracle server is not available or if you supply the wrong username or password, an error will be returned right away. If there is a networking problem, SQL*Plus may take several minutes before returning an error.

Here are some common error messages and some suggestions on how to resolve them:

`ERROR: ORA-12154: TNS:could not resolve service name`
: Either the Host string was mis-typed or SQL*Net is not configured properly. Exit SQL*Plus and try logging in again. If the error still occurs, try another PC.

`ERROR: ORA-01017: invalid username/password; logon denied`
: Either the username or password was typed incorrectly. Exit SQL*Plus and try again.

Unfortunately, most versions of SQL*Plus will not re -display the login screen if your attempt to connect is unsuccessful. You should exit SQL*Plus completely by pulling down the File menu and choosing the Exit menu item. Then run SQL*Plus again from the beginning.

For users of Personal Oracle Lite, there is a default database schema created upon installation of the software. To log into Personal Oracle Lite using SQL*Plus, supply the following values on the SQL*Plus login screen:

In the **User Name:** field, type in `OOT_SCH`
In the **Password:** field, type in `OOT_SCH`
In the **Host String:** field, type in `ODBC:POLITE`.

3.1.2 Running SQL*Plus under UNIX

To run SQL*Plus under **UNIX**, log into your UNIX account and at the UNIX command prompt (shown as `unix%` below), type the `sqlplus` command followed by a carriage return. When prompted for a username, supply your Oracle username (This may be the same as or different from your UNIX account name). When prompted for a password, supply your Oracle account password (this should not be the same as your UNIX account password).

```
unix%  sqlplus

SQL*Plus: Release 3.3.2.0.0 - Production on Sun Dec 21 13:32:53 1997
Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

Enter user-name: holowczak
Enter password: ****************

Connected to:
Oracle7 Server Release 7.3.2.3.0 - Production Release
With the distributed, replication, and parallel query options
PL/SQL Release 2.3.2.3.0 - Production

SQL>
```

To exit the SQL*Plus program (in any operating system), type `EXIT` and press Enter or carriage return:

```
 SQL>   EXIT
```

Once a session has been established using the SQL*Plus tool, any SQL statements or SQL*Plus Commands may be issued. In the following section, the basic SQL*Plus Commands are introduced.

3.2 SQL*Plus Commands

SQL*Plus commands allow a user to manipulate and submit SQL statements. Specifically, they enable a user to:

- Enter, edit, store, retrieve, and run SQL statements
- List the column definitions for any table
- Format, perform calculations on, store, and print query results in the form of reports

- Access and copy data between SQL databases

The following is a list of SQL*Plus commands and their functions. The most commonly used commands are emphasized in italics:

- / - Execute the current SQL statement in the buffer - same as RUN
- ACCEPT - Accept a value from the user and place it into a variable
- APPEND - Add text to the end of the current line of the SQL statement in the buffer
- AUTOTRACE - Trace the execution plan of the SQL statement and gather statistics
- BREAK - Set the formatting behavior for the output of SQL statements
- BTITLE - Place a title on the bottom of each page in the printout from a SQL statement
- CHANGE - Replace text on the current line of the SQL statement with new text
- CLEAR - Clear the buffer
- COLUMN - Change the appearance of an output column from a query
- COMPUTE - Does calculations on rows returned from a SQL statement
- CONNECT - Connect to another Oracle database or to the same Oracle database under a different user name
- COPY - Copy data from one table to another in the same or different databases
- DEL - Delete the current line in the buffer
- DESCRIBE - List the columns with datatypes of a table
- EDIT - Edit the current SQL statement in the buffer using an external editor such as `vi` or `emacs`
- EXIT - Exit the SQL*Plus program
- GET - Load a SQL statement into the buffer but do not execute it
- *HELP* - Obtain help for a SQL*Plus command (In some installations)
- HOST - Drop to the operating system shell
- INPUT - Add one or more lines to the SQL statement in the buffer
- LIST - List the current SQL statement in the buffer
- QUIT - Exit the SQL*Plus program
- REMARK - Place a comment following the REMARK keyword
- *RUN* - Execute the current SQL statement in the buffer
- *SAVE* - Save the current SQL statement to a script file
- SET - Set a variable to a new value
- SHOW - Show the current value of a variable
- *SPOOL* - Send the output from a SQL statement to a file
- *START* - Load a SQL statement located in a script file and then run that SQL statement
- TIMING - Used to time the execution of SQL statements for performance analysis
- TTITLE - Place a title on the top of each page in the printout from a SQL statement
- UNDEFINE - Delete a user defined variable

Examples of these SQL*Plus commands are given in the following sections.

Note the distinction made between *SQL*Plus Commands* and *SQL Statements*. SQL*Plus commands are proprietary to the Oracle SQL*Plus tool. SQL is a standard language that can be used is just about any Relational Database Management System (RDBMS).

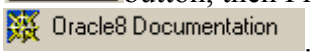
3.3 SQL*Plus Help Facilities

Some versions of SQL*Plus store the help documentation in the database and make it available via the SQL*Plus command line. Newer installations have changed this and now store the documentation in HTML format which can be read using a World Wide Web Browser such as MS Internet Explorer or Netscape Navigator.

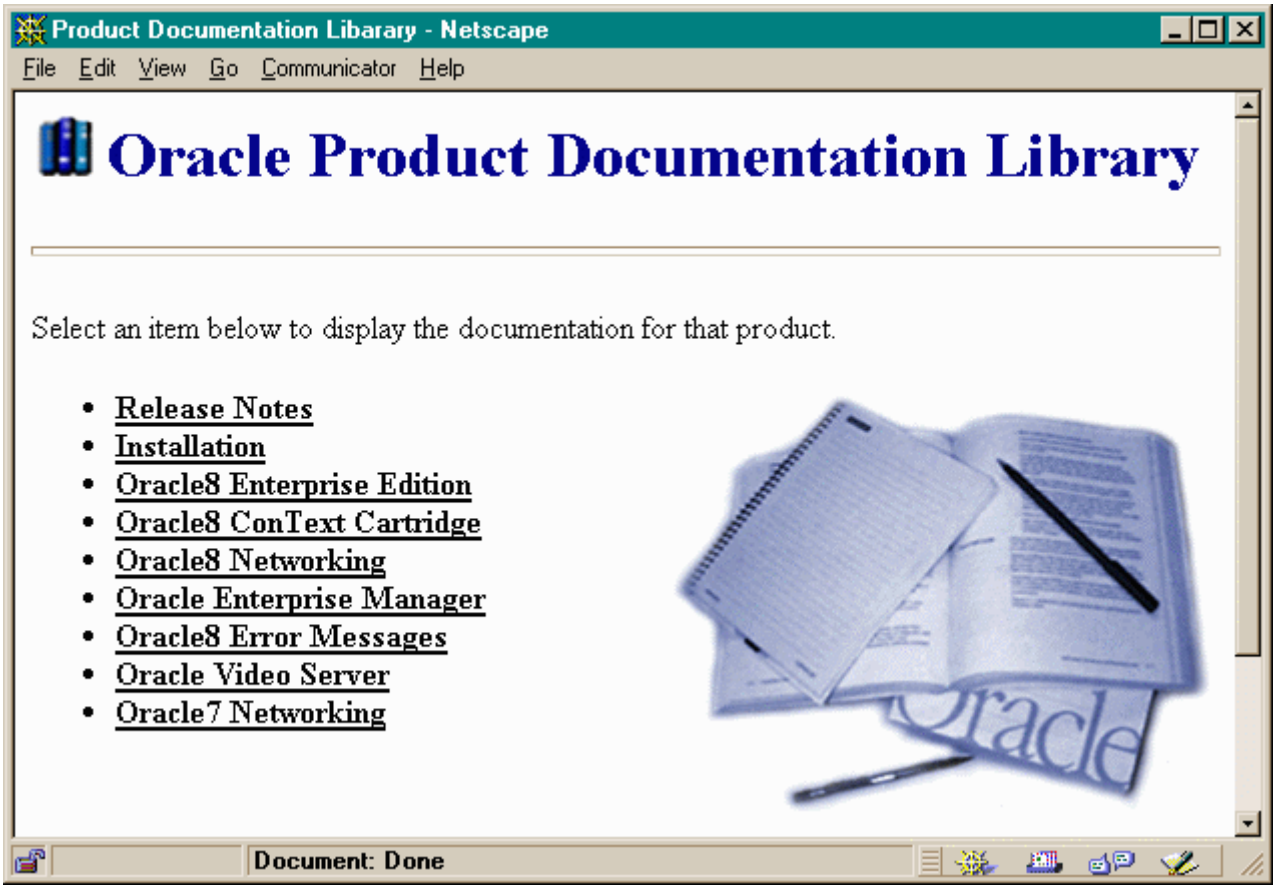
The following two sections describe how to invoke help in SQL*Plus under Windows 95/NT and under UNIX. The method you use to access help may differ according to how your software was installed.

3.3.1 Getting Help Under Windows 95/98/NT

To get HELP on any of the oracle tools, use the Oracle8 Documentation which is accessible through a web browser. To access the Oracle8 Documentation, click on the Windows 95 

button, then Programs, Oracle for Windows 95 and finally Oracle8 Documentation: 

. This will launch your local Web Browser (Netscape Navigator/Communicator or Microsoft Internet Explorer) and the Welcome to the Oracle8 Documentation Library! screen will be displayed. From here, click on the "TEXT VERSION" link to get to the Oracle Product Documentation Library.



Once in the Oracle8 Documentation main screen, click on Oracle8 Enterprise Edition and then SQL*Plus Getting Started for Windows NT/95. Other documentation you may find useful are:

Help File/Link	Contents
SQL Reference	Comprehensive syntax for all SQL statements
SQL*Plus Getting Started for Windows NT/95	Specific SQL*Plus commands and options for Windows 95 and NT users.
SQL*Plus Quick Reference	Quick reference guide to SQL*Plus commands.
SQL*Plus User's Guide and Reference	Comprehensive guide to using SQL*Plus.

Each of these can be found on the same Oracle8 Enterprise Edition page.

3.3.2 Getting Help Under UNIX

Under the UNIX operating system, help on SQL statements and SQL*Plus commands can be retrieved at the `SQL>` prompt by typing HELP followed by the command or statement. For example, to get help on the SELECT statement, type `HELP SELECT` as follows:

```
SQL> HELP SELECT
    SELECT command

PURPOSE:
    To retrieve data from one or more tables, views,
    or snapshots.

SYNTAX:

SELECT [DISTINCT | ALL] { *
                      | { [schema.]{table | view | snapshot}.*
                        | expr }  [ [AS] c_alias ]
                     [, { [schema.]{table | view | snapshot}.*
                        | expr } [ [AS] c_alias ]  ] ... }
    FROM [schema.]{table | view | subquery | snapshot} [t_alias]
      [, [schema.]... ] ...
    [WHERE condition ]
    [ [START WITH condition] CONNECT BY condition]
    [GROUP BY expr [, expr] ... [HAVING condition] ]
    [{UNION | UNION ALL | INTERSECT | MINUS} SELECT command ]
    [ORDER BY {expr|position} [ASC | DESC]
           [, {expr|position} [ASC | DESC]] ...]
    [FOR UPDATE [OF [[schema.]{table | view}.]column
                 [, [[schema.]{table | view}.]column] ...] ]

etc.
```

---

4. The SQL Language

Structured Query Language (SQL) is the language used to manipulate relational databases. SQL is tied very closely with the relational model.

In the relational model, data is stored in structures called relations or *tables*. Each table has one or more attributes or *columns* that describe the table. In relational databases, the table is the fundamental building block of a database application. Tables are used to store data on Employees, Equipment, Materials, Warehouses, Purchase Orders, Customer Orders, etc. Columns in the Employee table, for example, might be Last Name, First Name, Salary, Hire Date, Social Security Number, etc.

SQL statements are issued for the purpose of:

- Data definition - Defining tables and structures in the database (DB).
- Data manipulation - Inserting new data, Updating existing data, Deleting existing data, and Querying the Database (Retrieving existing data from the database).

Another way to say this is the SQL language is actually made up of 1) the Data Definition Language (DDL) used to create, alter and drop scema objects such as tables and indexes, and 2) The Data Manipulation Language (DML) used to manipulate the data within those schema objects.

The SQL language has been standardized by the ANSI X3H2 Database Standards Committee. Two of the latest standards are SQL-89 and SQL-92. Over the years, each vendor of relational databases has introduced new commands to extend their particular implementation of SQL. Oracle's implementation of the SQL language conforms to the basic SQL-92 standard and adds some additional commands and capabilities.

4.1 SQL Statements

The following is an alphabetical list of SQL statements that can be issued against an Oracle database. These commands are available to any user of the Oracle database. Emphasized items are most commonly used.

- ALTER - Change an existing table, view or index definition
- AUDIT - Track the changes made to a table
- COMMENT - Add a comment to a table or column in a table
- COMMIT - Make all recent changes permanent
- *CREATE* - Create new database objects such as tables or views
- *DELETE* - Delete rows from a database table
- *DROP* - Drop a database object such as a table, view or index
- GRANT - Allow another user to access database objects such as tables or views
- *INSERT* - Insert new data into a database table
- No AUDIT - Turn off the auditing function
- REVOKE - Disallow a user access to database objects such as tables and views
- ROLLBACK - Undo any recent changes to the database
- *SELECT* - Retrieve data from a database table
- *UPDATE* - Change the values of some data items in a database table

Some examples of SQL statements follow. For all examples in this tutorial, key words used by SQL and Oracle are given in all uppercase while user-specific information, such as table and column names, is given in lower case.

To create a new table to hold employee data, we use the CREATE TABLE statement:

```
CREATE TABLE employee
(fname           VARCHAR2(8),
 minit           VARCHAR2(2),
 lname           VARCHAR2(8),
 ssn             VARCHAR2(9) NOT NULL,
 bdate           DATE,
 address         VARCHAR2(27),
 sex             VARCHAR2(1),
 salary          NUMBER(7) NOT NULL,
 superssn        VARCHAR2(9),
 dno             NUMBER(1) NOT NULL) ;
```

To insert new data into the employee table, we use the INSERT statement:

```
INSERT INTO employee
VALUES ('BUD', 'T', 'WILLIAMS', '132451122',
       '24-JAN-54', '987 Western Way, Plano, TX',
       'M', 42000, NULL, 5);
```

To retrieve a list of all employees with salary greater than 30000 from the employees table, the following SQL statement might be issued (Note that all SQL statements end with a semicolon):

```
SELECT fname, lname, salary
FROM   employee
WHERE  salary > 30000;
```

To give each employee in department 5 a 4 percent raise, the following SQL statement might be issued:

```
UPDATE employee
SET    salary = salary * 1.04
WHERE  dno = 5;
```

To delete an employee record from the database, the following SQL statement might be issued:

```
DELETE FROM employee
WHERE  empid = 101 ;
```

The above statements are just an example of some of the many SQL statements and variations that are used with relational database management systems. The full syntax of these commands and additional examples are given below.

4.2 SQL Data Definition Language

In this section, the basic SQL Data Definition Language statements are introduced and their syntax is given with examples.

An Oracle database can contain one or more *schemas*. A schema is a collection of database objects that can include: tables, views, indexes and sequences. By default, each user has their own the schema which has the same name as the Oracle username. For example, a single Oracle database can have separate schemas for HOLOWCZAK, JONES, SMITH and GREEN.

Any object in the database must be created in only one schema. The object name is prefixed by the schema name as in: `schema.object_name`
By default, all objects are created in the user's own schema. For example, when JONES creates a database object such as a table, it is created in her own schema. If JONES creates an EMPLOYEE table, the full name of the table becomes: `JONES.EMPLOYEE`. Thus database objects with the same name can be created in more than one schema. This feature allows each user to have their own EMPLOYEE table, for example.

Database objects can be shared among several users by specifying the schema name. In order to work with a database object from another schema, a user must be granted authorization. See the section below on GRANT and REVOKE for more details.

Please note that many of these database objects and options are not available under Personal Oracle Lite. For example, foreign key constraints are not supported. Please see the on-line documentation for Personal Oracle Lite for more details.

4.2.1 Create, Modify and Drop Tables, Views and Sequences

SQL*Plus accepts SQL statements that allow a user to create, alter and drop table, view and sequence definitions. These statements are all standard ANSI SQL statements with the exception of CREATE SEQUENCE.

- **ALTER TABLE** - Change an existing table definition. The table indicated in the ALTER statement must already exist. This statement can be used to add a new column or remove an existing column in a table, modify the data type for an existing column, or add or remove a constraint.

  ALTER TABLE has the following syntax for adding a new column to an existing table:

```
ALTER TABLE <table name>
ADD ( <column name>  <data type> <[not]null> ) ;
```

  Another ALTER TABLE option can change a data type of column. The syntax is:

```
ALTER TABLE <table name>
MODIFY ( <column name>  <new data type> <[not]null> ) ;
```

  Finally, ALTER TABLE can also be used to add a constraint to a table such as for a PRIMARY KEY, FOREIGN KEY or CHECK CONSTRAINT. The syntax to add a PRIMARY KEY is:

```
ALTER TABLE <table name>
ADD CONSTRAINT <constraint-name>
PRIMARY KEY (<column-name>);
```

The syntax to add a FOREIGN KEY constraint is:

```
ALTER TABLE <table-name>
ADD CONSTRAINT <constraint-name>
FOREIGN KEY (<column-name>)
REFERENCES <parent-table-name> (column-name);
```

In Oracle, you must use an ALTER TABLE statement to define a composite PRIMARY KEY (a key made up of two or more columns).

NOTE: In Oracle, there is no single command to drop a column of a table. In order to drop a column from a table, you must create a temporary table containing all of the columns and records that will be retained. Then drop the original table and rename the temporary table to the original name. This is demonstrated below in the section on Creating, Altering and Dropping Tables.

- **CREATE TABLE** - Create a new table in the database. The table name must not already exist. CREATE TABLE has the following syntax:

```
CREATE TABLE <table_name>
( <column1_name>  <data type> <[not]null>,
  <column2_name>  <data type> <[not]null>,
  . . .
) ;
```

An alternate syntax can be used to create a table with a subset of rows or columns from an existing table.

```
CREATE TABLE <table_name> AS
<sql select statement> ;
```

- **DROP TABLE** - Drop a table from the database. The table name must already exist in the database. The syntax for the DROP TABLE statement is:

```
DROP TABLE <table name> ;
```

- **CREATE INDEX** - Create a new Index that facilitates rapid lookup of data. An index is typically created on the primary and/or secondary keys of the table. The basic syntax for the CREATE INDEX statement is:

```
CREATE INDEX <index name>
ON           <table name>
```

```
                        ( <column name>, <column name> ) ;
```

- **DROP INDEX** - Drop an index from the database. The syntax for the DROP INDEX statement is:

```
DROP INDEX <index name> ;
```

- **CREATE SEQUENCE** - Create a new Oracle Sequence of values. The new sequence name must not exist. CREATE SEQUENCE has the following syntax:

```
CREATE SEQUENCE <sequence name>
INCREMENT BY    <increment number>
START WITH      <start number>
MAXVALUE        <maximum value>
CYCLE ;
```

- **DROP SEQUENCE** - Drop an Oracle Sequence. The sequence name must exist. DROP SEQUENCE has the following syntax:

```
DROP SEQUENCE <sequence name> ;
```

- **CREATE VIEW** - Create a new view based on existing tables in the database. The table names must already exist. The new view name must not exist. CREATE VIEW has the following syntax:

```
CREATE VIEW <view name> AS
<sql select statement> ;
```

where *sql select statement* is in the form:

```
SELECT <column names>
FROM   <table name>
WHERE  <where clause>
```

Additional information on the SELECT statement and SQL queries can be found in the next section.

Note that an ORDER BY clause may not be added to the *sql select statement* when defining a view.

In general, views are read-only. That is, one may query a view but it is normally the case that views can not be operated on with INSERT, UPDATE or DELETE. This is especially true in cases where views joing two or more tables together or when a view contains an aggregate function.

- **DROP VIEW** - Drop a view from the database. The view name must already exist in the database. The syntax for the DROP VIEW command is:

```
DROP VIEW <view name> ;
```

In the following section, each of the SQL DDL commands will be discussed in more detail.

Creating, Altering and Dropping Tables

A table is made up of one or more columns (also called attributes in relational theory). Each column is given a name and a data type that reflects the kind of data it will store. Oracle supports four basic data types called CHAR, NUMBER, DATE and RAW. There are also a few additional variations on the RAW and CHAR data types. The basic datatypes, uses and syntax, are as follows:

- **VARCHAR2** - Character data type. Can contain letters, numbers and punctuation. The syntax for this data type is: `VARCHAR2(size)` where *size* is the maximum number of alphanumeric characters the column can hold. For example `VARCHAR2(25)` can hold up to 25 alphanumeric characters. In Oracle8, the maximum size of a VARCHAR2 column is 4,000 bytes.

  The VARCHAR data type is a synonym for VARCHAR2. It is recommended to use VARCHAR2 instead of VARCHAR.

- **NUMBER** - Numeric data type. Can contain integer or floating point numbers only. The syntax for this data type is: `NUMBER(precision, scale)` where *precision* is the total size of the number including decimal point and *scale* is the number of places to the right of the decimal. For example, `NUMBER(6,2)` can hold a number between `-999.99` and `999.99`.
- **DATE** - Date and Time data type. Can contain a date and time portion in the format: `DD-MON-YY HH:MI:SS`. No additional information is needed when specifying the DATE data type. If no time component is supplied when the date is inserted, the time of 00:00:00 is used as a default. The output format of the date and time can be modified to conform to local standards.
- **RAW** - Free form binary data. Can contain binary data up to 255 characters. Data type LONG RAW can contain up to 2 gigabytes of binary data. RAW and LONG RAW data cannot be indexed and can not be displayed or queried in SQL*Plus. Only one RAW column is allowed per table.
- **LOB** - Large Object data types. These include BLOB (Binary Large OBject) and CLOB (Character Large OBject). More than one LOB column can appear in a table. These data types are the prefferred method for storing large objects such as text documents (CLOB), images, or video (BLOB).

A column may be specified as NULL or NOT NULL meaning the column may or may not be left blank, respectively. This check is made just before a new row is inserted into the table. By default, a column is created as NULL if no option is given.

In addition to specifying NOT NULL constraints, tables can also be created with constraints that enforce referential integrity (relationships among data between tables). Constraints can be added to one or more columns, or to the entire table.

Each table may have one PRIMARY KEY that consists of a single column containing no NULL values and no repeated values. A PRIMARY KEY with multiple columns can be identified using the ALTER TABLE command.

Up to 255 columns may be specified per table. Column names and table names must start with a letter and may not contain spaces or other punctuation except for the underscore character. Column names and table names are case insensitive. This means that you can specify the names of columns and tables in any way you like. For example, the following three SELECT statements are all identical:

```
SELECT lname, fname, address FROM employee;
SELECT LNAME, FNAME, ADDRESS FROM EMPLOYEE;
SELECT Lname, Fname, Address FROM Employee;
```

In the following example, a new table called ``employee'' is created with ten columns of a variety of types. The columns indicated by NOT NULL will be mandatory while the other columns, by default, will be optional.

```
SQL>   CREATE TABLE employee
   2   (fname           VARCHAR2(8),
   3    minit           VARCHAR2(2),
   4    lname           VARCHAR2(8),
   5    ssn             VARCHAR2(9) NOT NULL,
   6    bdate           DATE,
   7    address         VARCHAR2(27),
   8    sex             VARCHAR2(1),
   9    salary          NUMBER(7) NOT NULL,
  10    superssn        VARCHAR2(9),
  11    dno             NUMBER(1) NOT NULL) ;

 Table created.

 SQL>
```

The numbers 2 through 11 before each line indicate the line number supplied by the SQL*Plus program as this statement was typed in. We will omit these numbers in the rest of the examples to facilitate copying and pasting this material directly into a live SQL*Plus session.

A new table can also be created with a subset of the columns in an existing table. In the following example, a new table called `emp_department_1` is created with only the *fname,*

minit, lname and *bdate* columns from the `employee` table. This new table is also populated with data from the `employee` table where the employees are from department number 1.

```
SQL> CREATE TABLE emp_department_1
     AS SELECT fname, minit, lname, bdate
     FROM employee
     WHERE dno = 1 ;

 Table created.

SQL> DESCRIBE emp_department_1
 Name                           Null?    Type
 ------------------------------ -------- ----
 FNAME                                   VARCHAR2(8)
 MINIT                                   VARCHAR2(2)
 LNAME                                   VARCHAR2(8)
 BDATE                                   DATE

SQL>
```

One can also create a new table with all of the columns from the original table, but with only a subset of the rows form the original table:

```
SQL> CREATE TABLE high_pay_emp
     AS SELECT *
     FROM employee
     WHERE salary > 50000 ;

 Table created.
```

DESCRIBE is an SQL*Plus command that displays the columns of a table and their data types. The syntax for the DESCRIBE command is:

```
   DESCRIBE <table name> ;
```

The copying of data can be suppressed by giving a WHERE clause that always evaluates to FALSE for each record in the source table. The following example makes a duplicate of the `employee` table but does not copy any data into it.

```
SQL>  CREATE TABLE copy_of_employee
      AS SELECT *
      FROM employee
      WHERE 3=5 ;

 Table created.

SQL> DESCRIBE copy_of_employee
 Name                           Null?    Type
 ------------------------------ -------- ----
 FNAME                                   VARCHAR2(8)
 MINIT                                   VARCHAR2(2)
 LNAME                                   VARCHAR2(8)
 SSN                            NOT NULL VARCHAR2(9)
 BDATE                                   DATE
```

```
 ADDRESS                                   VARCHAR2(27)
 SEX                                       VARCHAR2(1)
 SALARY                           NOT NULL NUMBER(7)
 SUPERSSN                                  VARCHAR2(9)
 DNO                              NOT NULL NUMBER(1)
```

Constraints can be added to the table at the time it is created, or at a later time using the ALTER TABLE statement. Constraints can include:

- Primary key and Unique key constraints.
- Foreign key constraints (for referential integrity).
- Check constraints.

Here is an example of creating a primary key constraint on the `empid` column:

```
CREATE TABLE employee
(fname           VARCHAR2(8),
 minit           VARCHAR2(2),
 lname           VARCHAR2(8),
 ssn             VARCHAR2(9) NOT NULL,
 bdate           DATE,
 address         VARCHAR2(27),
 sex             VARCHAR2(1),
 salary          NUMBER(7) NOT NULL,
 superssn        VARCHAR2(9),
 dno             NUMBER(1) NOT NULL,
 CONSTRAINT pk_emp PRIMARY KEY (ssn) );
```

Referential integrity constraints can also be added. In the following example, the `dno` column in the `employee` table references the `dnumber` column in the `department` table. If a department is deleted, all employees that reference the department are also deleted. This is given by the `ON DELETE CASCADE` option:

```
CREATE TABLE department
(dnumber      NUMBER(1),
 dname        VARCHAR2(15),
 mgrssn       VARCHAR2(9),
 mgrstartdate DATE
 CONSTRAINT pk_department PRIMARY KEY (dnumber) );

CREATE TABLE employee
(fname           VARCHAR2(8),
 minit           VARCHAR2(2),
 lname           VARCHAR2(8),
 ssn             VARCHAR2(9) NOT NULL,
 bdate           DATE,
 address         VARCHAR2(27),
 sex             VARCHAR2(1),
 salary          NUMBER(7) NOT NULL,
 superssn        VARCHAR2(9),
 dno             NUMBER(1) NOT NULL,
 CONSTRAINT pk_emp PRIMARY KEY (ssn),
 CONSTRAINT fk_dno FOREIGN KEY (dno)
```

```
 REFERENCES department (dnumber) ON DELETE CASCADE);
```

In order to specify a foreign key constraint, the column in the child (or detail) table (e.g., the `dnumber` column in the `department` table in the above example) must be either the primary key or a unique key for the table. Thus, the child (or detail) table must be created first before the parent (or master) table is created using the above constraints.

Additional CREATE TABLE constraint statements allow the specification of what should happen when a row is deleted or updated in a parent table. In the above example, deleting a department causes all employees in that department to also be deleted. Other options include `ON DELETE SET DEFAULT` and `ON DELETE SET NULL`. In addition, the behavior of child tables when a parent table is updated can also be specified using an `ON UPDATE` clause.

CHECK constraints can be added to check the values for a given column. This can be used to allow only a specific set of valid values for a column. In the following example, CHECK constraints are added to limit the valid values for the `sex` column and to check if the salary is greater than 10,000 (be sure to DROP TABLE employee before you try the next one).

```
CREATE TABLE employee
(fname           VARCHAR2(8),
 minit           VARCHAR2(2),
 lname           VARCHAR2(8),
 ssn             VARCHAR2(9) NOT NULL,
 bdate           DATE,
 address         VARCHAR2(27),
 sex             VARCHAR2(1)
     CONSTRAINT ck_sex CHECK (sex IN ('M', 'F')),
 salary          NUMBER(7) NOT NULL
     CONSTRAINT ck_salary CHECK (salary > 10000),
 superssn        VARCHAR2(9),
 dno             NUMBER(1) NOT NULL,
 CONSTRAINT pk_emp PRIMARY KEY (ssn),
 CONSTRAINT fk_dno FOREIGN KEY (dno)
    REFERENCES department (dnumber) ON DELETE CASCADE);
```

The CHECK constraints are activated when inserting a new row or when updating existing data. In the following example, the value given for sex is 'm':

```
SQL> insert into employee values
  2  ('Joe', 'M', 'Smith', '123456789', '01-JUN-45',
  3   '123 Smith St.', 'm', 45000, '123456789', 1) ;
  insert into employee values
  *
  ERROR at line 1:
  ORA-02290: check constraint (HOLOWCZAK.CK_SEX) violated
```

In the previous examples, constraints were given names with the following prefixes:

- Primary key constraints: `pk_`
- Foreign key constraints: `fk_`

- Check constraints: `ck_`

Naming constraints in this fashion is simply a convenience. Any name may be given to a constraint.

The ALTER TABLE command can be used to add a new column to an existing table or to change the data type of an existing column. The following examples add a new column *manager* to an existing table named `emp_department_1` and then modify the data type of the *fname* column.

```
SQL> DESCRIBE emp_department_1
 Name                            Null?    Type
 ------------------------------- -------- ----
 FNAME                                    VARCHAR2(8)
 MINIT                                    VARCHAR2(2)
 LNAME                                    VARCHAR2(8)
 BDATE                                    DATE

SQL>  ALTER TABLE emp_department_1
      ADD (manager VARCHAR2(8)) ;

Table altered.

SQL>  ALTER TABLE emp_department_1
      MODIFY (fname VARCHAR2(15));

Table altered.

SQL> DESCRIBE emp_department_1
 Name                            Null?    Type
 ------------------------------- -------- ----
 FNAME                                    VARCHAR2(15)
 MINIT                                    VARCHAR2(2)
 LNAME                                    VARCHAR2(8)
 BDATE                                    DATE
 MANAGER                                  VARCHAR2(8)
```

The ALTER TABLE command can also be used to change the datatype of column *provided there is no data in the table*. To get around this if there is data in the table, create a temporary table using all of the data from the existing table, delete the existing records from the original table, alter the datatype, and then insert the records from the temporary table back into the original table. For example, assume the `emp_department_1` table has some records in it and we want to change the datatype for the MANAGER column:

```
CREATE TABLE temp AS SELECT * FROM emp_department_1;

DELETE FROM emp_department_1;

ALTER TABLE emp_department_1
MODIFY (manager VARCHAR2(15));

INSERT INTO emp_department_1
SELECT * FROM temp;
```

```
DROP TABLE temp;
```

This trick can also be used to drop a column from a table. Assume the Employee table has the following columns: `fname, minit, lname, ssn, bdate, address, sex, salary, superssn` and `dno`, and we want to drop the `salary` column from the table:

```
CREATE TABLE temp AS
SELECT fname, minit, lname, ssn, bdate,
address, sex, superssn, dno FROM employee;

DROP TABLE employee;

CREATE TABLE employee AS
SELECT * FROM temp;
```

The DROP TABLE command can be used to drop a table definition and all of its data from the database. In the following example, the table `emp_department_1` created previously, is dropped from the database.

```
SQL> DROP TABLE emp_department_1 ;

Table dropped.
```

Exercise 1: Creating and Altering Tables

As an exercise, create a table called STUDENTS with the following columns and data types:

```
Column Name      Data Type
StudentID        NUMBER(5,0)       NOT NULL
Name             VARCHAR2(25)
Major            VARCHAR2(15)
GPA              NUMBER(6,3)
```

Create another table called COURSES with the following columns and data types:

```
Column Name      Data Type
StudentID        NUMBER(5,0)       NOT NULL
CourseNumber     VARCHAR2(15)      NOT NULL
CourseName       VARCHAR2(25)
Semester         VARCHAR2(10)
Year             NUMBER(4,0)
Grade            VARCHAR2(2)
```

Use the DESCRIBE command to display the data types of the columns after each table is created.

Next, use the ALTER TABLE statement to add the following column to the STUDENTS table:

```
Column Name      Data Type
TutorID          NUMBER(5,0)
```

Use the ALTER TABLE statement to define the StudentID as the PRIMARY KEY for the STUDENTS table.

Use the ALTER TABLE statement to define the StudentID and CourseNumber as the PRIMARY KEY for the COURSES table. To do this, list both of the column names separated by a comma.

Use the ALTER TABLE statement to define StudentID in the COURSES table as a FOREIGN KEY that references the StudentID in the STUDENTS table.

Finally, add some data to the STUDENTS and COURSES tables (simply copy and paste these statements into SQL*Plus to add the data):

```
INSERT INTO students VALUES (101, 'Bill', 'CIS', 3.45,  102);
INSERT INTO students VALUES (102, 'Mary', 'CIS', 3.10,  NULL);
INSERT INTO students VALUES (103, 'Sue',  'Marketing', 2.95, 102);
INSERT INTO students VALUES (104, 'Tom',  'Finance', 3.5, 106);
INSERT INTO students VALUES (105, 'Alex', 'CIS', 2.75, 106);
INSERT INTO students VALUES (106, 'Sam',  'Marketing', 3.25, 103);
INSERT INTO students VALUES (107, 'Jane', 'Finance', 2.90, 102);

INSERT INTO courses VALUES (101, 'CIS3400', 'DBMS I', 'FALL', 1997, 'B+');
INSERT INTO courses VALUES (101, 'CIS3100', 'OOP I', 'SPRING', 1999, 'A-');
INSERT INTO courses VALUES (101, 'MKT3000', 'Marketing', 'FALL', 1997,
'A');
INSERT INTO courses VALUES (102, 'CIS3400', 'DBMS I', 'SPRING', 1997, 'A-
');
INSERT INTO courses VALUES (102, 'CIS3500', 'Network I', 'SUMMER', 1997,
'B');
INSERT INTO courses VALUES (102, 'CIS4500', 'Network II', 'FALL', 1997,
'B+');
INSERT INTO courses VALUES (103, 'MKT3100', 'Advertizing', 'SPRING', 1998,
'A');
INSERT INTO courses VALUES (103, 'MKT3000', 'Marketing', 'FALL', 1997,
'A');
INSERT INTO courses VALUES (103, 'MKT4100', 'Marketing II', 'SUMMER', 1998,
'A-');
```

Creating and Dropping Indexes

An index is a data structure that afford rapid lookup of data in a table. An index is normally created on those columns of a table used to look up data. For example, in the employee table, the key *ssn* can be used to look up the rest of an employee's information. Creating a index on the *ssn* field would be accomplished by the following statement:

```
SQL>  CREATE INDEX employee_ssn_idx
      ON employee (ssn) ;

Index created.
```

It is also possible to create indexes on other columns of a table. For example, if the `employee` table is frequently accessed by *superssn*, an index can be created on that column as well:

```
SQL>  CREATE INDEX employee_superssn_idx
      ON employee (superssn) ;
```

```
Index created.
```

Indexes can be dropped using the DROP INDEX statement: For example, to drop just the *employee_superssn_idx* index, one could submit:

```
DROP INDEX employee_superssn_idx ;

Index Dropped.
```

Note that dropping a table (using the DROP TABLE statement) automatically drops all indexes on that table.

Exercise 2: Creating and Altering Tables

For this exercise, create an index on the STUDENTS table for the `Name` column. Be sure to give this index an appropriate name.

Create an index on the COURSES table for the `semester` and `year` columns (together).

Creating and Dropping Views

In the SQL language, a view is a representation of one or more tables. A view can be used to hide the complexity of relationships between tables or to provide security for sensitive data in tables. In the following example, a limited view of the `employee` table is created. When a view is defined, a SQL statement is associated with the view name. Whenever the view is accessed, the SQL statement will be executed.

In the following example, the view `emp_dno_1` is created as a limited number of columns (*fname, lname, dno*) and limited set of data (`WHERE dno=1`) from the `employee` table.

```
     CREATE VIEW emp_dno_1
     AS SELECT fname, lname, dno
     FROM employee
     WHERE dno = 1;

View created.
```

Once the view is created, it can be queried with a SELECT statement as if it were a table.

```
SELECT * FROM emp_dno_1 ;

FNAME    LNAME          DNO
-------- -------- ---------
JAMES    BORG             1
```

Views can be dropped in a similar fashion to tables. The DROP VIEW command provides this facility. In the following example, the view just created is dropped.

```
DROP VIEW emp_dno_1 ;
```

```
View dropped.
```

Views can also be created to join several tables together. The following is an example of creating a view that joins two tables:

```
SQL> CREATE VIEW dept_managers AS
  2  SELECT dnumber, dname, mgrssn, lname, fname
  3  FROM   employee, department
  4  WHERE  employee.ssn = department.mgrssn ;

View created.

SQL> SELECT * FROM dept_managers ;

DNUMBER DNAME               MGRSSN LNAME    FNAME
------- --------------- ---------- -------- --------
      5 RESEARCH         333445555 WONG     FRANKLIN
      4 ADMINISTRATION   987654321 WALLACE  JENNIFER
      1 HEADQUARTERS     888665555 BORG     JAMES
```

This view can then be used as part of other queries or as the basis for developing applications.

As a final example, a view can be created that contains an aggregate function. In the following example, a view is created that returns the average salary of all employees per department.

```
SQL> CREATE VIEW dept_average_salary AS
  2  SELECT dnumber, dname, AVG(salary) AS average_salary
  3  FROM   department, employee
  4  WHERE  employee.dno = department.dnumber
  5  GROUP BY dnumber, dname ;

 View created.

SQL> SELECT * FROM dept_average_salary ;

   DNUMBER DNAME           AVERAGE_SALARY
---------- --------------- --------------
         1 HEADQUARTERS             55000
         4 ADMINISTRATION           31000
         5 RESEARCH                 33250
```

Note the use of the column alias `AS average_salary` and the mandatory GROUP BY clause.

Note that in general, views are read-only as in the above cases.

To see which views are defined in a schema, submit a query to the USER_VIEWS view:

```
SQL> SELECT view_name FROM user_views ;

VIEW_NAME
-------------------------
DEPT_AVERAGE_SALARY
```

```
DEPT_MANAGERS
EMP_DNO_1
```

Exercise 3: Creating Views

For this exercise, create a view called V_CIS_MAJORS basd upon the following SQL SELECT statement:

```
SELECT * FROM students WHERE major = 'CIS';
```

Query the view and show the output.

Create another view called V_COURSES_TAKEN based upon the following SQL SELECT statement:

```
SELECT name, major, coursenumber, coursename,
       semester, year, grade
FROM   students, courses
WHERE  students.studentid = courses.studentid;
```

Before querying this view, format the output column by submitting the following SQL*Plus COLUMN FORMAT commands:

```
COLUMN name       FORMAT A8
COLUMN coursename FORMAT A15
COLUMN major      FORMAT A10
COLUMN year       FORMAT 9999
```

As discussed in Section 5.2, the format command changes the way data is displayed in SQL*Plus. It does not change how the data is stored in the tables.

Query the V_COURSES_TAKEN view and show the output.

Creating, Altering and Dropping Sequences

The Oracle database provides a database object known as a Sequence. Sequences are used to automatically generate a series of unique numbers such as those used for Employee Id or Part Number columns. Sequences are not part of the ANSI SQL-92 standard. In the following example, an Oracle Sequence for Employee Id is created. The numbers to be generated will be between 1001 and 9999. As a rule of thumb, sequences can be named with the suffix `seq` to differentiate them from other database objects.

```
      CREATE SEQUENCE department_number_seq
      START WITH  1
      MAXVALUE    9999
      NOCYCLE ;

Sequence created.
```

In this example, the sequence will begin its numbering at 1 and count up (in increments of 1 which is the default) until it reaches 9999. Once the MAXVALUE is reached, accessing the sequence will return an error.

Sequences are accessed using a SELECT statement with a special table called `DUAL`. The DUAL table is a placeholder that exists in all schemas by default. In the following example, the next value in the `employee_id_seq` sequence is retrieved:

```
SELECT department_number_seq.nextval FROM dual ;

  NEXTVAL
---------
     6
```

Sequences can also be used in INSERT statements to automatically provide the next value for a key. For example, to insert a new employee row with the next employee id in the sequence, the following statement would be issued:

```
INSERT INTO department VALUES
  (department_number_seq.nextval, 'Finance',
   '123456789', '01'-JAN-90');

1 Row Created.
```

As with most database objects, Oracle Sequences can be dropped using a DROP SEQUENCE command. Dropping a sequence and then re-creating it has the effect of resetting the sequence to its START WITH number. In the following example, the Employee Id sequence created previously is dropped.

```
DROP SEQUENCE department_number_seq ;

Sequence dropped.
```

Sequences can also be altered to change the INCREMENT BY, MAXVALUE or START WITH values. The ALTER SEQUENCE statement achieves these changes.

Exercise 4: Working with Sequences

For this exercise, start by creating an Oracle Sequence called `student_id_seq`. Have the sequence start numbering at 120 and go up to 999.

Then, write an SQL INSERT statement to insert a new record for the following person:

```
Name:     Joe
Major:    CIS
GPA:      3.85
TutorID:  103
```

Use the `student_id_seq.nextval` as the StudentID. Finally, use a SELECT statement to query the V_CIS_MAJOR and see if the record was inserted properly.

4.2.2 Grant and Revoke Statements

The GRANT and REVOKE statements allow a user to control access to objects (Tables, Views, Sequences, Procedures, etc.) in their schema. The Grant command grants authorization for a *subject* (another user or group) to perform some *action* (SELECT, INSERT, UPDATE, DELETE, ALTER, INDEX) on an *object* (Table, View, stored procedure, sequence or synonym).

The actions are defined as follows:

- **SELECT** - allows a subject to select rows from the object.
- **INSERT** - allows a subject to insert rows into the object.
- **UPDATE** - allows a subject to update rows in the object.
- **DELETE** - allows a subject to delete rows from the object.
- **ALTER** - allows a subject to alter the object. For example, add a column or change a constraint.
- **INDEX** - allows a subject to create an index on the object.
- **EXECUTE** - allows a subject to execute a stored procedure or trigger.

In addition to objects such as tables, the SELECT and UPDATE actions may also be granted on individual columns in a table or view.

The general syntax for the GRANT statement is:

```
GRANT <action1>, <action2>, ...
ON    tablename
TO    subject;
```

For example, assume user ALICE wishes to allow another user BOB to view the rows in the employee table. ALICE would execute the following GRANT statement:

```
GRANT SELECT
ON    employee
TO    BOB ;
```

At this point, user BOB may now issue SQL SELECT statements on the table `ALICE.employee`. For example, user BOB may execute:

```
SELECT * FROM ALICE.employee ;

FNAME    MI LNAME          SSN
-------- -- -------- ---------
JOHN     B  SMITH    123456789
FRANKLIN T  WONG     333445555
ALICIA   J  ZELAYA   999887777
JENNIFER S  WALLACE  987654321
RAMESH   K  NARAYAN  666884444
JOYCE    A  ENGLISH  453453453
AHMAD    V  JABBAR   987987987
JAMES    E  BORG     888665555
etc.
```

The REVOKE statement reverses the authorization by removing privileges from a subject (user). The syntax for REVOKE is:

```
REVOKE <action>
ON     <object>
FROM   <subject>
```

For example, to revoke Bob's privileges to read the employee table, Alice might execute:

```
REVOKE SELECT
ON     employee
FROM   BOB ;
```

The current authorizations in effect can be viewed by selecting from the USER_TAB_PRIVS view. In the following example, the columns are first formatted (more examples of this are given in a later section), and then the privileges for the user (table owner) ALICE are displayed.

```
 COLUMN grantee    FORMAT A10
 COLUMN grantor    FORMAT A10
 COLUMN owner      FORMAT A10
 COLUMN table_name FORMAT A10
 COLUMN privilege  FORMAT A10
 SELECT * FROM USER_TAB_PRIVS ;

GRANTEE    OWNER      TABLE_NAME GRANTOR    PRIVILEGE  GRA
---------- ---------- ---------- ---------- ---------- ---
BOB        ALICE       EMPLOYEE  ALICE      SELECT     NO
```

A quick way to generate a list of GRANT statements for every table in your schema is to run a query on the catalog that forms the GRANT statements:

```
SELECT 'GRANT SELECT, INSERT, UPDATE, DELETE ON '
        || table_name ||  ' TO username;'
        FROM   cat
        WHERE  table_type = 'TABLE';
```

In the above example, *username* is the name of the user you would like to grant access to.

The result of this query is something like the following:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON DEPARTMENT TO username;
GRANT SELECT, INSERT, UPDATE, DELETE ON DEPENDENT TO username;
GRANT SELECT, INSERT, UPDATE, DELETE ON DEPT_LOCATIONS TO username;
GRANT SELECT, INSERT, UPDATE, DELETE ON EMPLOYEE TO username;
GRANT SELECT, INSERT, UPDATE, DELETE ON PROJECT TO username;
GRANT SELECT, INSERT, UPDATE, DELETE ON WORKS_ON TO username;
```

This output can then be copied and pasted back in to put the grant statements into effect.

Exercise 5: GRANT and REVOKE

GRANT SELECT privileges to another member of your group. Have them query your STUDENTS table.

THen REVOKE the SELECT privilege from the STUDENTS table. Have your group member try to query the table after you have revoked access and see what happens.

4.2.3 Synonyms

In many cases, a schema is created under a single username but other users must have access to the tables, sequences and stored procedures. One possibility is to code all queries and applications to specifically access these database objects by providing a schema name. For example;

```
SELECT * FROM alice.employee;
```

This query selects all of the columns and rows from the employee table in user alice's schema.

One problem with this method is that if the tables move to another user's schema, all of the references will need to change.

An alternative is to use *Synonyms* to provide a pointer to the schema and database objects. A Synonym is like a pointer in that is has a name that is recognized in the local schema that, when addressed, will resolve to the schema.object name in another user's schema.

Synonyms are created with the CREATE SYNONYM command:

```
CREATE SYNONYM <synonym_name> FOR  <schema>.<object> ;
```

One can create synonyms for tables, views, sequences and stored procedures.

For example, if Bob wishes to have access to Alice's employee table, first, Alice would need to GRANT access to her table using the GRANT command, and then Bob would create a synonym using:

```
CREATE SYNONYM employee FOR alice.employee;
```

Now Bob can execute the following query:

```
SELECT * FROM employee ;
```

The above query will return all columns and rows from the employee table in Alice's schema.

If the tables are moved to another schema such as Abe's schema, then only the synonyms need to be dropped and recreated. All applications will run the same.

To generate a list of CREATE SYNONYM statements, use the following type of query:

```
SELECT 'CREATE SYNONYM ' || table_name ||
       ' FOR schema.' || table_name || ' ;'
FROM   cat
WHERE  table_type = 'TABLE' ;
```

Where *schema* is the username containing the tables. This results in the following output that can be pasted back into SQL*Plus by each user to create the set of synonyms:

```
CREATE SYNONYM DEPARTMENT FOR schema.DEPARTMENT ;
CREATE SYNONYM DEPENDENT FOR schema.DEPENDENT ;
CREATE SYNONYM DEPT_LOCATIONS FOR schema.DEPT_LOCATIONS ;
CREATE SYNONYM EMPLOYEE FOR schema.EMPLOYEE ;
CREATE SYNONYM PROJECT FOR schema.PROJECT ;
CREATE SYNONYM STUDENTS FOR schema.STUDENTS ;
CREATE SYNONYM WORKS_ON FOR schema.WORKS_ON ;
```

In this section, the SQL commands for creating, altering and deleting tables, views and sequences, and granting and revoking access to database objects have been introduced. A typical database may have a dozen or more related tables with several columns each. To facilitate the creation and deletion of a large number of tables, the CREATE statements can be placed into a file and executed using the SQL*Plus *START* command.

4.3 SQL Data Manipulation Language

In this section, we discuss SQL statements that can be used to manipulate data in tables and views.

4.3.1 Select, Insert, Update, Delete, Commit and Rollback Data

SQL*Plus allows the user to enter SQL statements to select, insert, update and delete rows in database tables. These are all standard SQL statements.

- COMMIT - Make all recent changes to the database permanent. Changes that have occurred since the last commit are made permanent. A commit can be done explicitly using the following syntax:

  ```
  COMMIT ;
  ```

  A commit is also done implicitly when the next SQL statement is executed or the user exits SQL*Plus.

- DELETE - Delete one or more rows from a table. The syntax for this SQL statement is:
-
-
  ```
  DELETE FROM <table name>
  WHERE <where clause>
  ```

  If the WHERE clause is omitted, all rows in the table will be deleted.

- INSERT - Insert a row of data into a table. The syntax for this SQL statement is:
-

```
INSERT INTO <table name>
(column1, column2 . . .)
VALUES (value1, value2, . . .)
```

If a value for each column in the table is supplied, then the columns do not need to be listed in the first set of parenthesis. Values can be of 3 types: Character, Number or Date. Each one requires a slightly different format when inserting:

- Character - Must be enclosed within single quotes
  For example: 'Bill Smith'
- Number - No quotes are required
  For example: 123, 44000.12
- Date - Enclosed in single quotes in the format 'DD-MON-YY'
  For example: '26-JUN-96'

All values, regardless of data type, must be separated by commas.

Another option for the INSERT statement is to pull some data from another table. The syntax is:

```
INSERT INTO <table name>
SELECT <columns>
FROM   <table>
WHERE  <where-clause>
```

For example, assume table1 and table2 have the same number of columns and the corresponding columns have the same data types. To insert all data currently in table1 into table2:

```
INSERT INTO table2
SELECT * FROM table1;
```

- ROLLBACK - Undo all recent changes to the database. A rollback can only undo changes made since the last commit. The syntax for the ROLLBACK command is:

```
ROLLBACK ;
```

- SELECT - Retrieve existing rows from a table. If the table is empty, a message indicating that no rows were found will be displayed. A simplified syntax for the SELECT statement is:

```
SELECT <column1, column2, . . .>
FROM   <table1, table2, . . .>
WHERE  <where clause>
GROUP BY <column1, column2, . . .>
HAVING   <having clause>
ORDER BY <column1, column2, . . .>
```

The WHERE clause, GROUP BY, HAVING and ORDER BY statements are optional. If a WHERE clause if omitted, all rows in the table will be retrieved. If the ORDER BY statement is omitted, there is no specific order in which the rows will be displayed. GROUP BY and HAVING are used in conjunction with aggregate functions (functions that operate on more than one record). If all columns in the table are to be retrieved, an asterisk (*) may be substituted for the entire list of columns after the SELECT key word.

More than one table can be specified in the FROM clause. The WHERE clause typically contains logic expressions (such as `WHERE salary > 40000`) that are evaluated for each row in the table.

A more complete syntax for the SELECT statement is:

```
SELECT     <column1, column2, . . .>
FROM       <schema.table1, schema.table2, . . .> | <view>
WHERE      <where clause>
CONNECT BY <connect by expression>
GROUP BY   <group by expression>
HAVING     <having clause>
ORDER BY   <column1, column2, ....> ASC | DESC
```

- UPDATE - Change the values of existing rows in a table in the database. One or more rows must exist in the table in order to successfully update data. The syntax for this SQL statement is:
-
-
```
UPDATE <table name>
SET    <column name> = <expression>
WHERE  <where clause>
```

The *expression* can be either a single value or an arithmetic expression including another column in the table. More than one column can be updated at a time by adding additional *column name = expression* pairs separated by commas. If the WHERE clause is omitted, the update is applied to all rows in the table.

For example, to give all employees in the marketing department a 3% raise:

```
UPDATE employee
SET    salary = salary * 1.03
WHERE  dno =
(SELECT dno
 FROM   department
 WHERE  dname = 'MARKETING');
```

In the following example, a new row is inserted into the `employee` table. Since a value is supplied for each column, the columns do not need to be explicitly listed.

```
DESCRIBE employee ;
Name                            Null?    Type
------------------------------- -------- ----
FNAME                                    VARCHAR2(8)
MINIT                                    VARCHAR2(2)
LNAME                                    VARCHAR2(8)
SSN                             NOT NULL NUMBER
BDATE                                    DATE
ADDRESS                                  VARCHAR2(27)
SEX                                      VARCHAR2(1)
SALARY                          NOT NULL NUMBER(7)
SUPERSSN                                 NUMBER(9)
DNO                             NOT NULL NUMBER(1)

   INSERT INTO employee
    VALUES ('JOHN', 'B', 'SMITH', 123456789, '09-JAN-55',
            '731 FONDREN, HOUSTON, TX', 'M', 30000, 333445555, 5) ;

 1 row created.
```

To check the contents of the `employee` table, a SELECT statement is done on the table.

```
SELECT * FROM EMPLOYEE ;

FNAME    MI LNAME          SSN BDATE     ADDRESS                    S SALARY
SUPERSSN DNO
-------- -- -------- --------- --------- ------------------------- - ------
--------- ---
JOHN     B  SMITH    123456789 09-JAN-55 731 FONDREN, HOUSTON, TX  M  30000
333445555   5
FRANKLIN T  WONG     333445555 08-DEC-45 638 VOSS,HOUSTON TX       M  40000
888665555   5
ALICIA   J  ZELAYA   999887777 19-JUL-58 3321 CASTLE, SPRING, TX   F  25000
987654321   4
JENNIFER S  WALLACE  987654321 20-JUN-31 291 BERRY, BELLAIRE, TX   F  43000
888665555   4
RAMESH   K  NARAYAN  666884444 15-SEP-52 975 FIRE OAK, HUMBLE, TX  M  38000
333445555   5
JOYCE    A  ENGLISH  453453453 31-JUL-62 5631 RICE, HOUSTON, TX    F  25000
333445555   5
AHMAD    V  JABBAR   987987987 29-MAR-59 980 DALLAS, HOUSTON, TX   M  25000
987654321   4
JAMES    E  BORG     888665555 10-NOV-27 450 STONE, HOUSTON, TX    M  55000
1

8 rows selected.
```

In the next example, a row in the `employee` table is updated.

```
      UPDATE employee
      SET    salary = salary * 1.04
```

```
      WHERE  dno = 4;

3 rows updated.


SELECT * FROM employee ;

FNAME    MI LNAME          SSN BDATE     ADDRESS                    S SALARY
SUPERSSN DNO
-------- -- -------- --------- --------- ------------------------- - ------
--------- ---
JOHN     B  SMITH    123456789 09-JAN-55 731 FONDREN, HOUSTON, TX  M  30000
333445555   5
FRANKLIN T  WONG     333445555 08-DEC-45 638 VOSS,HOUSTON TX       M  40000
888665555   5
ALICIA   J  ZELAYA   999887777 19-JUL-58 3321 CASTLE, SPRING, TX   F  26000
987654321   4
JENNIFER S  WALLACE  987654321 20-JUN-31 291 BERRY, BELLAIRE, TX   F  44720
888665555   4
RAMESH   K  NARAYAN  666884444 15-SEP-52 975 FIRE OAK, HUMBLE, TX  M  38000
333445555   5
JOYCE    A  ENGLISH  453453453 31-JUL-62 5631 RICE, HOUSTON, TX    F  25000
333445555   5
AHMAD    V  JABBAR   987987987 29-MAR-59 980 DALLAS, HOUSTON, TX   M  26000
987654321   4
JAMES    E  BORG     888665555 10-NOV-27 450 STONE, HOUSTON, TX    M  55000
1

8 rows selected.
```

In the final example, a row is deleted from the `employee` table.

```
SQL>  DELETE FROM employee
      WHERE dno = 5;

4 rows deleted.

SQL> COMMIT ;
Commit complete.
```

In the final example, if the ROLLBACK command was given instead of the COMMIT command, the rows would have been undeleted.

4.3.2 Displaying Table Metadata (Data about the data)

Once database objects have been created, it is often useful to query the data dictionary to see the various characteristics of the objects. In this section, we describe several ways to query the data dictionary to retrieve this information.

Note that many of these statements and commands will not work properly under Personal Oracle Lite.

The Oracle Data Dictionary maintains a collection of `USER_` views that are accessible from each user's schema. The following table summarizes these views:

USER View	Contents	Typical Query
USER_TABLES	Table names and storage details about tables a user owns	SELECT table_name FROM USER_TABLES;
CAT or TAB	Brief list of tables and views for a user	SELECT * FROM CAT; *or* SELECT * FROM TAB;
COL	Column names and NOT NULL constraints.	SELECT colno, cname, coltype, width, scale, precision, nulls FROM col WHERE tname = 'EMPLOYEE' ORDER BY col.colno;
USER_INDEXES	Indexes defined on tables the user owns	COLUMN table_owner FORMAT A12 SELECT index_name, table_owner, table_name FROM USER_INDEXES ;
USER_VIEWS	View names and view definitions (queries) a user owns	SELECT view_name, text FROM USER_VIEWS;
USER_SEQUENCES	Sequence definitions and current values for sequences a user owns	SELECT * FROM USER_SEQUENCES ;
USER_TRIGGERS	Trigger names and definitions for triggers a user owns	SELECT trigger_name, trigger_body FROM USER_TRIGGERS;
USER_ERRORS	Contains information about the last error that occurred in a user's schema due to a trigger or procedure compilation error.	SELECT * FROM USER_ERRORS;
USER_CONSTRAINTS	Constraints on tables a user owns. Includes column constraints such as NOT NULL, CHECK and foreign key constraints.	SELECT constraint_name, table_name, search_condition FROM USER_CONSTRAINTS WHERE table_name = 'EMPLOYEE';
USER_OBJECTS	All database objects a user owns. Includes tables, views, sequences, indexes, procedures, triggers, etc.	COLUMN object_name FORMAT A35 SELECT object_name, object_type FROM USER_OBJECTS ;
USER_SOURCE	Source code for stored procedures owned by the user.	To see which procedures exist: SELECT DISTINCT NAME from USER_SOURCE; To see the actual code: SELECT

		TEXT FROM USER_SOURCE WHERE NAME = 'procedure_name' ORDER BY LINE; Note: You may have to reduce the ARRAYSIZE variable to avoid overflowing the bufer. e.g., `SET ARRAYSIZE 2`
USER_TS_QUOTAS	Quotas on tablespaces accessible to a user.	SELECT * FROM USER_TS_QUOTAS ;

A comprehensive list of user catalog views can be found in the Oracle Server Reference guide.

Many of the view contain columns of type LONG. In order to display their content, set the SQL*Plus variable `LONG` to a large number such as 4096 as follows:

```
SQL>   SET LONG 4096
```

You may have to reduce the ARRAYSIZE variable to avoid overflowing the bufer. e.g.,

```
SET ARRAYSIZE 2
```

To find out the names of tables you have created, use the system view called `CAT` in a SELECT statement: `SELECT * FROM cat;` . The following is an example:

```
SELECT * FROM cat ;

TABLE_NAME                     TABLE_TYPE
------------------------------ ----------
EMPLOYEE                       TABLE
DEPARTMENT                     TABLE
PROJECT                        TABLE
DEPENDENTS                     TABLE
```

The `TAB` view was supported in older versions of Oracle and may not be available in future releases of Oracle. In that case, try using the `CAT` view instead of `TAB`.

The column definitions for a table can be displayed using the `DESCRIBE` command in SQL*Plus:

```
DESCRIBE employee ;
Name                           Null?    Type
------------------------------ -------- ----
FNAME                                   VARCHAR2(8)
MINIT                                   VARCHAR2(2)
LNAME                                   VARCHAR2(8)
SSN                            NOT NULL NUMBER
BDATE                                   DATE
ADDRESS                                 VARCHAR2(27)
SEX                                     VARCHAR2(1)
SALARY                         NOT NULL NUMBER(7)
SUPERSSN                                NUMBER(9)
```

```
 DNO                                NOT NULL NUMBER(1)
```

More detailed metadata can be retrieved from the tables `COL` and `user_constraints`.

To get information on columns of a table, use the following (substitute 'EMPLOYEE' with the name of the table in question):

```
SQL> COLUMN coltype FORMAT A10
SQL> COLUMN cname FORMAT A15
SQL> SELECT   colno, cname, coltype, width, scale, precision, nulls
     FROM     col
     WHERE    tname = 'EMPLOYEE'
     ORDER BY col.colno;

COLNO CNAME           COLTYPE    WIDTH SCALE PRECISION NULLS
----- --------------- ---------- ----- ----- --------- ---------
    1 FNAME           VARCHAR2       8                 NULL
    2 MINIT           VARCHAR2       2                 NULL
    3 LNAME           VARCHAR2       8                 NULL
    4 SSN             NUMBER        22                 NOT NULL
    5 BDATE           DATE           7                 NULL
    6 ADDRESS         VARCHAR2      27                 NULL
    7 SEX             VARCHAR2       1                 NULL
    8 SALARY          NUMBER        22     0          7 NOT NULL
    9 SUPERSSN        NUMBER        22     0          9 NULL
   10 DNO             NUMBER        22     0          1 NOT NULL

10 rows selected.
```

To see any constraints that are presently in effect on a table, use the following (substitute 'EMPLOYEE' with the name of the table in question):

```
SQL> COLUMN search_condition FORMAT A21
SQL> SELECT constraint_name, constraint_type,
            search_condition, delete_rule
     FROM   user_constraints
     WHERE  table_name = 'EMPLOYEE';

CONSTRAINT_N CONSTRAINT_T SEARCH_CONDITION      DELETE_RULE
------------ ------------ -------------------- -----------
FK_DNO       R                                 CASCADE
SYS_C00886   C            EMPID IS NOT NULL
SYS_C00887   C            SSN IS NOT NULL
SYS_C00888   C            SALARY IS NOT NULL
SYS_C00889   C            DNO IS NOT NULL
CK_SEX       C            sex IN ('M', 'F')
CK_SALARY    C            salary > 10000
PK_EMP       P
```

A list of Indexes defined on tables in the user's schema can be displayed by querying the `USER_INDEXES` table:

```
SQL> COLUMN table_owner FORMAT A12
SQL> SELECT index_name, table_owner, table_name FROM USER_INDEXES ;

INDEX_NAME                     TABLE_OWNER  TABLE_NAME
------------------------------ ------------ ------------------------------
ACCOUNTS_PK                    HOLOWCZA     ACCOUNTS
AT_PK                          HOLOWCZA     ACCOUNT_TYPES
COURSES_PK                     HOLOWCZA     COURSES
CUSTOMER_PK                    HOLOWCZA     CUSTOMERS
PK_DEPARTMENT                  HOLOWCZA     DEPARTMENT
PK_EMP                         HOLOWCZA     EMPLOYEE
UNQ_RNAME                      HOLOWCZA     LOGREPORT
```

Finally, a list of Views the user owns can be displayed by querying the `USER_VIEWS` table:

```
SQL> SET LONG 4096
SQL> SELECT view_name, text FROM USER_VIEWS;

VIEW_NAME
------------------
TEXT
--------------------------------------------------------------
VACCOUNTS
SELECT c.fname, c.lname, ac.account_number, at.account_typeid,
       at.interest_rate, at.minimum_balance,
       ac.date_opened, ac.current_balance
FROM   customers c, accounts ac, account_types at
WHERE  c.customerid = ac.customerid
  AND  ac.account_typeid = at.account_typeid

V_COURSES_TAKEN
SELECT name, major, coursenumber, coursename,
       semester, year, grade
FROM   students, courses
WHERE  students.studentid = courses.studentid
```

4.3.3 Oracle Pseudo-Columns

The Oracle implementation of SQL adds several *pseudo columns* to each table. These columns do not exist in a physical table, yet they can be used in any SQL statement for a variety of purposes.

The following table lists the major pseudo columns:

- **CURRVAL** - Returns the current value of an Oracle sequence.
- **NEXTVAL** - Returns the current value of an Oracle sequence and then increments the sequence.
- **LEVEL** - The current level in a hierarchy for a query using STARTWITH and CONNECT BY.
- **ROWID** - An identifier (data file, block and row) for the physical storage of a row in a table.
- **ROWNUM** - The integer indicating the order in which a row is returned from a query.

Exercise 6: Dispaying Metadata

For this exercise, query the `USER_` tables and display the following metadata:

- List the tables in the presently in the schema
- List the Indexes
- List the Views
- For the STUDENT and COURSES tables, display the columns in each table, their data types and whether or not they allow NULL values
- For the STUDENT and COURSES tables, display constraints on each table

4.3.4 Oracle SQL Functions

The Oracle implementation of SQL provides a number of functions that can be used in SELECT statements. Functions are typically grouped into the following:

- **Single row functions** - Operate on column values for each row returned by a query.
- **Group functions** - Operate on a collection (group) of rows.

The following is an overview and brief description of **single row functions.** *x* is some number, *s* is a string of characters and *c* is a single character.

- Math functions include:
  ABS (*x*) - Absolute Value of *x*
  CEIL (*x*) - Smallest integer greater than or equal to *x*. **COS (*x*)** - Cosine of *x*
  FLOOR (*x*) - Largest integer less than or equal to *x*. **LOG (*x*)** - Log of *x*
  LN (*x*) - Natural Log of *x*
  ROUND (*x*, *n*) - Round *x* to *n* decimal places to the right of the decimal point.
  SIN (*x*) - Sine of *x*
  TAN (*x*) - Tangent of *x*
  TRUNC (*x*, *n*) - Truncate *x* to *n* decimal places to the right of the decimal point.
- Character functions include:
  CHR (*x*) - Character for ASCII value *x*.
  INITCAP (*s*) - String *s* with the first letter of each word capitalized.
  LOWER (*s*) - Converts string *s* to all lower case letters.
  LPAD (*s*, *x*) - Pads string *s* with *x* spaces to the left.
  LTRIM (*s*) - Removes leading spaces from *s*.
  REPLACE (*s1*, *s2*, *s3*) - Replace occurrences of *s1* with *s2* in string *s*.
  RPAD (*s*, *x*) - Pads string *s* with *x* spaces to the right.
  RTRIM (*s*) - Removes trailing spaces from *s*.
  SUBSTR (*s*, *x1*, *x2*) - Return a portion of string *s* starting at position *x1* and ending with position *x2*. If *x2* is omitted, it's value defaults to the end of *s*.
  UPPER (*s*) - Converts string *s* to all upper case letters.
- Character functions that return numbers include:
  ASCII (*c*) - Returns the ASCII value of *c*
  INSTR (*s1*, *s2*, *x*) - Returns the position of *s2* in *s1* where the search starts at position *x*.
  LENGTH (*s*) - Length of *s*
- Conversion functions include:
  TO_CHAR (*date*, *format*) - Converts a date column to a string of characters. *format* is a set of Date formatting codes where:
  `YYYY` is a 4 digit year.
  `NM` is a month number.
  `MONTH` is the full name of the month.

`MON` is the abbreviated month.
`DDD` is the day of the year.
`DD` is the day of the month.
`D` is the day of the week.
`DAY` is the name of the day.
`HH` is the hour of the day (12 hour clock)
`HH24` is the hour of the day (24 hour clock)
`MI` is the minutes.
`SS` is the seconds.

TO_CHAR (*number*, *format*) - Converts a numeric column to a string of characters. *format* is a set of number formatting codes where:
`9` indicates a digit position. Blank if position value is 0.
`0` indicates a digit position. Shows a 0 if the position value is 0.
`$` displays a leading currency indicator.
TO_DATE (*s*, *format*) - Converts a character column (string *s* to a date. *format* is a set of Date formatting codes as above.
TO_NUMBER (*s*, *format*) - Converts a character column (string *s* to a Number. *format* is a set of Number formatting codes as above.

- Date functions include:
  SYSDATE - Returns the current date (and time if the TO_CHAR function is used) from the system clock.
- Some additional function are:
  DECODE (*s*, *search1*, *result1*, *search2*, *result2*) - Compares *s* with *search1*, *search2*, etc. and returns the corresponding *result* when there is a match.

  NVL (*s*, *expression*) - If *s* is NULL, return *expression*. If *s* is not null, then return *s*.

  USER - Returns the username of the current user.

The following is an overview and brief description of **multiple row (group) functions.** *col* is the name of a table column (or expression) of type NUMBER.

- **AVG (*col*)** - Returns the average of a group of rows for *col*
- **MAX (*col*)** - Returns the maximum of a group of rows for *col*
- **MIN (*col*)** - Returns the minimum of a group of rows for *col*
- **STDEV (*col*)** - Returns the standard deviation of a group of rows for *col*
- **SUM (*col*)** - Returns the sum (total) of a group of rows for *col*
- **VARIANCE (*col*)** - Returns the variance of a group of rows for *col*

In addition the COUNT group function counts instances of values. These values can be any type (CHAR, DATE or NUMBER):

- **COUNT (*columns*)** - Returns the number of instances of a group of rows for (*columns*)

To use an aggregate function, a GROUP BY clause must be added to the SELECT statement.

Examples of functions are given in the following section.

Exercise 7: Functions

For this exercise, use the various functions to display the following:

- Display the average, minimum, and maximum grade point average for all of the students
- For each student, write a sentence like the following:
  `Congratulations Bill, your grade point average is 3.45`
  You'll need to use the TO_CHAR function to convert the GPA column (which is a NUMBER data type) to a set of characters.
- For each student, count the number of courses he or she has taken.
- Modify the above query to only count CIS courses.
  Hint: You'll need to use the `SUBSTR` function on the `COURSENUMBER` column to extract the first three letters. Then compare this to 'CIS'.

4.3.5 Examples of SQL DML Statements

In this section, several examples of SQL DML statements are given. Variations on WHERE clause, FROM clause and using SQL functions are all demonstrated.

Basic Select Statements

Example Table STUDENTS:

```
CREATE TABLE students (studentid NUMBER(5,0), name VARCHAR2(25),
                       major VARCHAR2(15), gpa NUMBER(6,3),
                       tutorid NUMBER(5,0));
INSERT INTO students VALUES (101, 'Bill', 'CIS', 3.45,  102);
INSERT INTO students VALUES (102, 'Mary', 'CIS', 3.10,  NULL);
INSERT INTO students VALUES (103, 'Sue',  'Marketing', 2.95, 102);
INSERT INTO students VALUES (104, 'Tom',  'Finance', 3.5, 106);
INSERT INTO students VALUES (105, 'Alex', 'CIS', 2.75, 106);
INSERT INTO students VALUES (106, 'Sam',  'Marketing', 3.25, 103);
INSERT INTO students VALUES (107, 'Jane', 'Finance', 2.90, 102);
Example table COURSES:

Create table courses(studentid  NUMBER(5,0) NOT NULL,
coursenumber VARCHAR2(15) NOT NULL,
coursename VARCHAR2(25), semester  VARCHAR2(10),
year NUMBER(4,0), grade VARCHAR2(2));

INSERT INTO courses VALUES (101, 'CIS3400', 'DBMS I', 'FALL', 1997, 'B+');
INSERT INTO courses VALUES (101, 'CIS3100', 'OOP I', 'SPRING', 1999, 'A-');
INSERT INTO courses VALUES (101, 'MKT3000', 'Marketing', 'FALL', 1997,
'A');
INSERT INTO courses VALUES (102, 'CIS3400', 'DBMS I', 'SPRING', 1997, 'A-
');
INSERT INTO courses VALUES (102, 'CIS3500', 'Network I', 'SUMMER', 1997,
'B');
INSERT INTO courses VALUES (102, 'CIS4500', 'Network II', 'FALL', 1997,
'B+');
INSERT INTO courses VALUES (103, 'MKT3100', 'Advertizing', 'SPRING', 1998,
'A');
INSERT INTO courses VALUES (103, 'MKT3000', 'Marketing', 'FALL', 1997,
'A');
```

```
INSERT INTO courses VALUES (103, 'MKT4100', 'Marketing II', 'SUMMER', 1998,
'A-');
```

StudentID	Name	Major	GPA	TutorId
101	Bill	CIS	3.45	102
102	Mary	CIS	3.1	
103	Sue	Marketing	2.95	102
104	Tom	Finance	3.5	106
105	Alex	CIS	2.75	106
106	Sam	Marketing	3.25	103
107	Jane	Finance	2.9	102

- Average GPA of all students:

```
SELECT AVG(gpa)
FROM   students;

  AVG(GPA)
  ----------
  3.12857143
```

- Average GPA of Finance and CIS students:

```
SELECT AVG(gpa)
FROM   students
WHERE  major = 'CIS' OR major = 'Finance';

  AVG(GPA)
  ----------
       3.14
```

- Give the name of the student with the highest GPA:
  This is an example of a *subquery*

```
SELECT name, gpa
FROM   students
WHERE  gpa =
  (  SELECT MAX(gpa) FROM students  );

NAME            GPA
-------- ----------
Tom             3.5
```

Another option is to enclose some text in quotes and concatenate that text with the output of the SQL statement:

```
SELECT 'The student with the highest GPA is ' || name
FROM   students
WHERE  gpa =
  (  SELECT MAX(gpa) FROM students   );


NAME
----------------------------------------
The student with the highest grade is Tom
```

- Show the students with the GPA grades in each major:

```
SELECT    name, major, gpa
FROM      students s1
WHERE     gpa =
   (
     SELECT max(gpa)
     FROM   students s2
     WHERE  s1.major = s2.major
   );

NAME     MAJOR             GPA
-------- ---------- ----------
Bill     CIS              3.45
Tom      Finance           3.5
Sam      Marketing        3.25
```

  Note the two *aliases* given to the students table: **s1** and **s2**. These allow us to refer to different views of the same table.

You may wish to sort the output based on the GPA. In this case, the output is ordered by GPA in decending order (highest GPA will come first, etc.):

```
SELECT    name, major, gpa
FROM      students s1
WHERE     gpa =
   (
     SELECT max(gpa)
     FROM   students s2
     WHERE  s1.major = s2.major
   )
ORDER BY gpa DESC;

NAME     MAJOR             GPA
-------- ---------- ----------
Tom      Finance           3.5
Bill     CIS              3.45
Sam      Marketing        3.25
```

Selecting from 2 or More Tables

- In the `FROM` portion, list all tables separated by commas. Called a *Join*.
- The `WHERE` part becomes the *Join Condition*

Example table EMPLOYEE:

```
FNAME    MI LNAME         SSN BDATE     ADDRESS                    S SALARY
SUPERSSN DNO
-------- -- ------- --------- --------- ------------------------- - ------
--------- ---
JOHN     B  SMITH   123456789 09-JAN-55 731 FONDREN, HOUSTON, TX  M  30000
333445555 5
FRANKLIN T  WONG    333445555 08-DEC-45 638 VOSS,HOUSTON TX       M  40000
888665555 5
ALICIA   J  ZELAYA  999887777 19-JUL-58 3321 CASTLE, SPRING, TX   F  25000
987654321 4
JENNIFER S  WALLACE 987654321 20-JUN-31 291 BERRY, BELLAIRE, TX   F  43000
888665555 4
```

```
RAMESH   K  NARAYAN 666884444 15-SEP-52 975 FIRE OAK, HUMBLE, TX  M  38000
333445555 5
JOYCE    A  ENGLISH 453453453 31-JUL-62 5631 RICE, HOUSTON, TX    F  25000
333445555 5
AHMAD    V  JABBAR  987987987 29-MAR-59 980 DALLAS, HOUSTON, TX   M  25000
987654321 4
JAMES    E  BORG    888665555 10-NOV-27 450 STONE, HOUSTON, TX    M  55000
1
```

Example table DEPARTMENT:

```
DNAME              DNUMBER    MGRSSN MGRSTARTD
--------------- --------- --------- ---------
RESEARCH                 5 333445555 22-MAY-78
ADMINISTRATION           4 987654321 01-JAN-85
HEADQUARTERS             1 888665555 19-JUN-71
```

Example Table DEPT_LOCATIONS:

```
DNUMBER DLOCATION
------- ---------------
      1 HOUSTON
      4 STAFFORD
      5 BELLAIRE
      5 SUGARLAND
      5 HOUSTON
```

Example table DEPENDENT:

```
     ESSN DEPENDENT_NAME  SEX BDATE     RELATIONSHIP
--------- --------------- --- --------- ------------
333445555 ALICE           F   05-APR-76 DAUGHTER
333445555 THEODORE        M   25-OCT-73 SON
333445555 JOY             F   03-MAY-48 SPOUSE
123456789 MICHAEL         M   01-JAN-78 SON
123456789 ALICE           F   31-DEC-78 DAUGHTER
123456789 ELIZABETH       F   05-MAY-57 SPOUSE
987654321 ABNER           M   26-FEB-32 SPOUSE
```

- List all of the employees working in Houston:

```
SELECT  employee.fname, employee.lname
FROM    employee, dept_locations
WHERE   employee.dno = dept_locations.dnumber
AND     dept_locations.dlocation = 'HOUSTON' ;

FNAME    LNAME
-------- --------
JOHN     SMITH
FRANKLIN WONG
RAMESH   NARAYAN
JOYCE    ENGLISH
JAMES    BORG
```

- List each employee name and the location they work in. List them in order of location and name:

```
SELECT    dept_locations.dlocation, department.dname,
          employee.fname, employee.lname
FROM      employee, department, dept_locations
WHERE     employee.dno = department.dnumber
   AND    department.dnumber = dept_locations.dnumber
   AND    employee.dno = dept_locations.dnumber
ORDER BY  dept_locations.dlocation, employee.lname;
```

Results:

```
DLOCATION       DNAME           FNAME    LNAME
--------------- --------------- -------- --------
BELLAIRE        RESEARCH        JOYCE    ENGLISH
BELLAIRE        RESEARCH        RAMESH   NARAYAN
BELLAIRE        RESEARCH        JOHN     SMITH
BELLAIRE        RESEARCH        FRANKLIN WONG
HOUSTON         HEADQUARTERS    JAMES    BORG
HOUSTON         RESEARCH        JOYCE    ENGLISH
HOUSTON         RESEARCH        RAMESH   NARAYAN
HOUSTON         RESEARCH        JOHN     SMITH
HOUSTON         RESEARCH        FRANKLIN WONG
STAFFORD        ADMINISTRATION  AHMAD    JABBAR
STAFFORD        ADMINISTRATION  JENNIFER WALLACE
STAFFORD        ADMINISTRATION  ALICIA   ZELAYA
SUGARLAND       RESEARCH        JOYCE    ENGLISH
SUGARLAND       RESEARCH        RAMESH   NARAYAN
SUGARLAND       RESEARCH        JOHN     SMITH
SUGARLAND       RESEARCH        FRANKLIN WONG

16 rows selected.
```

What is the highest paid salary in Houston ?

```
SELECT MAX(employee.salary)
FROM   employee, dept_locations
WHERE  employee.dno = dept_locations.dnumber
 AND   dept_locations.dlocation = 'HOUSTON';

MAX(EMPLOYEE.SALARY)
--------------------
               55000
```

To obtain the Cartesian Product of two tables, use a SELECT statement with no WHERE clause:

```
SELECT *
FROM department, dept_locations ;

DNAME           DNUMBER    MGRSSN MGRSTARTD DNUMBER DLOCATION
--------------- ------- --------- --------- ------- ----------
RESEARCH              5 333445555 22-MAY-78       1 HOUSTON
ADMINISTRATION        4 987654321 01-JAN-85       1 HOUSTON
HEADQUARTERS          1 888665555 19-JUN-71       1 HOUSTON
RESEARCH              5 333445555 22-MAY-78       4 STAFFORD
ADMINISTRATION        4 987654321 01-JAN-85       4 STAFFORD
HEADQUARTERS          1 888665555 19-JUN-71       4 STAFFORD
RESEARCH              5 333445555 22-MAY-78       5 BELLAIRE
ADMINISTRATION        4 987654321 01-JAN-85       5 BELLAIRE
HEADQUARTERS          1 888665555 19-JUN-71       5 BELLAIRE
RESEARCH              5 333445555 22-MAY-78       5 SUGARLAND
ADMINISTRATION        4 987654321 01-JAN-85       5 SUGARLAND
HEADQUARTERS          1 888665555 19-JUN-71       5 SUGARLAND
RESEARCH              5 333445555 22-MAY-78       5 HOUSTON
ADMINISTRATION        4 987654321 01-JAN-85       5 HOUSTON
HEADQUARTERS          1 888665555 19-JUN-71       5 HOUSTON
```

```
                  15 rows selected.
```

In which states do our employees work ?

```
SELECT    DISTINCT dlocation
FROM      dept_locations;

DLOCATION
---------------
BELLAIRE
HOUSTON
STAFFORD
SUGARLAND
```

List the Department name and the total salaries for each department:

```
SELECT   department.dname, SUM( employee.salary )
FROM     employee, department
WHERE    employee.dno = department.dnumber
GROUP BY department.dname
```

Results:

```
DNAME           SUM(EMPLOYEE.SALARY)
--------------- --------------------
ADMINISTRATION                 93000
HEADQUARTERS                   55000
RESEARCH                      133000
```

We can also use a *Column Alias* to change the title of the columns

```
SELECT   department.dname, SUM( employee.salary ) AS TotalSalaries
FROM     employee, department
WHERE    employee.dno = department.dnumber
GROUP BY department.dname
```

Results:

```
DNAME           TOTALSALARIES
--------------- -------------
ADMINISTRATION          93000
HEADQUARTERS            55000
RESEARCH               133000
```

Here is a combination of a function and a column alias:

```
SELECT   fname, lname,
         salary  AS CurrentSalary,
         (salary * 1.03)  AS ProposedRaise
FROM     employee;

FNAME    LNAME    CURRENTSALARY PROPOSEDRAISE
-------- -------- ------------- -------------
JOHN     SMITH            30000         30900
FRANKLIN WONG             40000         41200
ALICIA   ZELAYA           25000         25750
JENNIFER WALLACE          43000         44290
RAMESH   NARAYAN          38000         39140
JOYCE    ENGLISH          25000         25750
AHMAD    JABBAR           25000         25750
JAMES    BORG             55000         56650
```

```
8 rows selected.
```

Recursive Queries and Table Aliases

- Recall some of the E-R diagrams and relations we dealt with had a recursive relationship.
- For example: A student can tutor one or more other students. A student has only one tutor.
  STUDENTS (studentid, name, major, grade, student_tutorid)
- Provide a listing of each student and the name of their tutor:

```
SELECT   s1.name AS Student,  tutors.name AS Tutor
FROM     students s1,  students tutors
WHERE    s1.tutorid = tutors.studentid;

STUDENT                   TUTOR
------------------------- -----------
Bill                      Mary
Sue                       Mary
Jane                      Mary
Sam                       Sue
Tom                       Sam
Alex                      Sam
```

- The above is called a "recursive" query because it access the same table two times.
- We give the table two *aliases* called **s1** and **tutors** so that we can compare different aspects of the same table.
- However, as is, the table is missing something: We don't see who is tutoring Mary. Use a left outer join to see the rest of the information. In MS Access, we use the `LEFT JOIN` command. In Oracle, we place a `(+)` after the join condition to indicate an outer join:

  In MS Access:

```
SELECT   s1.name AS Student,  tutors.name AS Tutor
FROM     students s1   LEFT JOIN   students tutors
ON       s1.tutorid = tutors.studentid;
```

  In Oracle:

```
SELECT   s1.name AS Student,  tutors.name AS Tutor
FROM     students s1, students tutors
WHERE    s1.tutorid = tutors.studentid (+) ;

STUDENT                   TUTOR
------------------------- -------------
Bill                      Mary
Sue                       Mary
Jane                      Mary
Sam                       Sue
Tom                       Sam
Alex                      Sam
Mary
```

- Here is one more twist: Suppose we were interested in those students who do not tutor anyone? Use a right outer join (`RIGHT JOIN` in MS Access).
- How many students does each tutor work with ?

```
SELECT   s1.name AS TutorName,
         COUNT(tutors.tutorid) AS NumberTutored
FROM     students s1, students tutors
WHERE    s1.studentid = tutors.tutorid
GROUP BY s1.name;

TUTORNAME                 NUMBERTUTORED
------------------------- -------------
Mary                                  3
Sam                                   2
Sue                                   1
```

Tree Queries

Another form of recursive query is the tree query. A tree query decomposes the table such that each row is a node the tree and nodes are related in levels. Consider the Students table defined above.

- Bill tutors Alex, Mary and Sue.
- Mary tutors Liz and Ed
- Sue tutors Petra

Using the SQL SELECT statements CONNECT BY and START WITH clauses, we can form a set of relationships between the rows of the table that form a tree structure.

- START WITH - indicates which row the tree should start with.
- CONNECT BY - indicates how successive related rows are to be identified and included in the result.
- LEVEL - a pseudo-column that indicates which level of the tree the current row is assigned to.

The following example prints a tree structure modeled after the tutoring relationships in the Students table. We will start with Mary's student id (102) since no one tutors her.

```
SELECT            LPAD(' ',2*(LEVEL-1)) || students.name
                  As TutorTree
FROM              students
START WITH        studentid = '102'
CONNECT BY PRIOR  studentid = tutorid;

TUTORTREE
--------------------------------------------------------------------------------
-----
Mary
  Bill
  Sue
    Sam
      Tom
      Alex
  Jane
```

```
7 rows selected.
```

From the tree we can see that Mary tutors Bill, Sue and Jane. In turn, Sue tutors Sam. Finally, Sam tutors both Tom and Alex.

WHERE Clause Expressions

- There are a number of expressions one can use in a WHERE clause.
- Subqueries using = (equals):

```
SELECT name, grade
FROM   students
WHERE  grade =
  (  SELECT MAX(grade) FROM students   );
```

This assumes the subquery returns only one tuple as a result.
Typically used when aggregate functions are in the subquery.

- Subqueries using the `IN` operator are used whenever the value of a column should be found in a *set* of values. The set of values can be explicitly listed (as in the first example) or they can be created on the fly using a subquery.

```
SELECT    employee.fname, department.dname
FROM      employee, department
WHERE     employee.dno = department.dnumber
  AND     department.dname IN ('HEADQUARTERS', 'RESEARCH');

FNAME    DNAME
-------- ---------------
JAMES    HEADQUARTERS
JOHN     RESEARCH
JOYCE    RESEARCH
RAMESH   RESEARCH
FRANKLIN RESEARCH


SELECT    employee.fname
FROM      employee
WHERE     employee.dno IN
          (SELECT dept_locations.dnumber
           FROM dept_locations
           WHERE dept_locations.dlocation = 'STAFFORD');

FNAME
-------
ALICIA
JENNIFER
AHMAD
```

In the above case, the subquery returns a set of tuples. The IN clause returns true when a tuple matches a member of the set.

- Subqueries using EXISTS. EXISTS will return TRUE if there is at least one row resulting from the subquery.

```
SELECT   fname, lname, salary
FROM     employee
WHERE    EXISTS
          (SELECT  fname
           FROM    EMPLOYEE e2
           WHERE   e2.salary > employee.salary)
       AND EXISTS
          (SELECT  fname
           FROM    EMPLOYEE e3
           WHERE   e3.salary < employee.salary);

FNAME    LNAME       SALARY
-------- -------- ---------
JOHN     SMITH        30000
FRANKLIN WONG         40000
JENNIFER WALLACE      43000
RAMESH   NARAYAN      38000
```

The above query shows all employees names and salaries where there is at least one person who makes more money (the first exists) and at least one person who makes less money (second exists).

- Subqueries with NOT EXISTS. NOT EXISTS will return TRUE if there are no rows returned by the subquery.

```
SELECT   fname, lname, salary
FROM     employee
WHERE    NOT EXISTS
          (SELECT  fname
           FROM    EMPLOYEE e2
           WHERE   e2.salary > employee.salary);

FNAME    LNAME       SALARY
-------- -------- ---------
JAMES    BORG         55000
```

The above query shows all employees for whom there does not exist an employee who is paid less. In other words, the highest paid employee.

- The **HAVING** clause is similar to the WHERE clause. The difference is that WHERE is used to filter individual rows while HAVING is used to filter groups according to the GROUP BY clause.

  Show the departments with average salary greater than 33000.

```
SELECT    department.dname, AVG(salary)
FROM      employee, department
```

```
WHERE     employee.dno = department.dnumber
GROUP BY  department.dname
HAVING    AVG(salary) > 33000 ;

DNAME           AVG(SALARY)
--------------- -----------
HEADQUARTERS          55000
RESEARCH              33250
```

Show departments with 3 or more employees:

```
SELECT   department.dname, COUNT(employee.dno)
  FROM   department, employee
 WHERE   department.dnumber = employee.dno
GROUP BY department.dname
HAVING   COUNT(employee.dno) >= 3;

DNAME           COUNT(EMPLOYEE.DNO)
--------------- -------------------
ADMINISTRATION                    3
RESEARCH                          4
```

Examples of SQL Functions

- Text, dates and numbers can be combined using the various conversion functions. In the following example, the TO_CHAR function is used to convert the date BDATE into a character string.

```
  SELECT 'The oldest employee was born on ' ||
         TO_CHAR( MIN(bdate), 'DD/MM/YY')
         AS Sentence
  FROM   employee;

  SENTENCE
  ----------------------------------------
  The oldest employee was born on 10/11/27
```

- Date math results in a numerical answer that must be converted to characters to concatenate with other character strings as in this next example:

```
SELECT 'The oldest employee was born on ' ||
       TO_CHAR( MIN(bdate), 'DD/MM/YY') || ' and is now' ||
       TO_CHAR( (SYSDATE - MIN(bdate)) / 365, '99') ||
       ' years old.'
       AS Sentence
FROM   employee;

SENTENCE
-----------------------------------------------------------------
The oldest employee was born on 10/11/27 and is now 70 years old.
```

- The DECODE function can be used to provide a variety of lookup values. In the following example, the string concatenation operator || is used to put together a sentence about each employee. The DECODE command takes the COUNT of dependents as the first argument. Then, depending on the COUNT for a given employee, it returns an appropriate ending to the sentence.

```
  SELECT   fname || ' '|| lname || ' has ' ||
           DECODE(COUNT(essn),
```

```
                  0, 'no dependents.',
                  1, 'one dependent.',
                  2, 'two dependents.',
                  3, 'three dependents.')
          AS Sentence
FROM     employee, dependent
WHERE    employee.ssn = dependent.essn (+)
GROUP BY employee.fname, lname;

SENTENCE
--------------------------------------
AHMAD JABBAR has no dependents.
ALICIA ZELAYA has no dependents.
FRANKLIN WONG has three dependents.
JAMES BORG has no dependents.
JENNIFER WALLACE has one dependent.
JOHN SMITH has three dependents.
JOYCE ENGLISH has no dependents.
RAMESH NARAYAN has no dependents.

8 rows selected.
```

Deleting Tuples with DELETE

- DELETE is used to remove tuples from a table.
- With no `WHERE` clause, DELETE will remove all tuples from a table.
- Remove all employees:

```
DELETE employee;
```

- Remove only employees making more than $50,000

```
DELETE employee
WHERE  salary  > 50000;
```

- Remove all employees working in Houston:

```
DELETE employee
WHERE  employee.dno IN
  (SELECT dept_locations.dnumber
   FROM   dept_locations
   WHERE  dlocation = 'HOUSTON');
```

- `DELETE` will not be successful if a constraint would be violated.
  For example, consider the `DNO` attribute in the Employee table as a Foreign Key.
  Removing a department would then be contingent upon no employees working in that department.
  This is what we call enforcing *Referential Integrity*

Change Values using UPDATE

- The `UPDATE` command is used to change attribute values in the database.

- UPDATE uses the `SET` clause to overwrite the value.
- Change the last name of an Employee:

```
UPDATE employee
SET    lname = 'SMITH'
WHERE  lname = 'JONES';
```

- Give an Employee a raise:

```
UPDATE employee
SET    salary = salary * 1.05
WHERE  fname = 'JOYCE' AND lname = 'ENGLISH' ;
```

- Give all employees over the age of 50 a raise:

```
    UPDATE EMPLOYEE
    SET SALARY = SALARY * 1.02
    WHERE TO_NUMBER( ( SYSDATE - bdate) / 365) >= 50;
```

Exercise 8: Update and Delete

For this exercise, write the SQL UPDATE and DELETE statements to:

- Add .05 to all of the Marketing major's GPA's.
- Change Sam's tutor from Sue to Jane
- For any student who is currently majoring in CIS and who has a GPA of less than 3.0, change their major to Marketing.

5. Advanced SQL*Plus Commands

This section introduces some of the advanced features of SQL*Plus including editing the SQL command buffer, formatting output from SQL SELECT statements, saving the output from SQL statements and collecting performance statistics on the execution of SQL statements.

5.1 Editing The SQL Buffer

SQL*Plus has several commands to allow the user to edit or modify SQL statements. Once a new SQL statement has been typed in (ending with a **;**) this statement is placed into a **buffer** and is considered to be the *current SQL statement*. All of the following commands operate on the current SQL statement in the buffer.

- / - Execute the current SQL statement in the buffer
- APPEND - Add text to the end of the current line of the SQL statement in the buffer
- CHANGE - Replace text on the current line of the SQL statement with new text
- CLEAR - Clear the buffer
- DEL - Delete the current line in the buffer
- INPUT - Add one or more lines to the SQL statement in the buffer
- LIST - List the current SQL statement in the buffer
- RUN - Execute the current SQL statement in the buffer
- SAVE - Save the current SQL statement to a script file
- START - Load a SQL statement located in a script file and then run that SQL statement

SQL statements may be typed with a free format. Spaces and <cr> characters may be used to separate key words in a SQL statement. SQL*Plus displays line numbers in the left hand margin indicating the current line for a SQL statement that spans multiple lines.

In the following example, an erroneous SQL statement has been entered. After the **;** was typed, an error message was displayed indicating the approximate location of the error and a brief error message description.

```
SQL> SELECT tname, tabtype
  2  FRO
  3  tab;
FRO
*
ERROR at line 2:
ORA-00923: FROM keyword not found where expected
```

To correct line number 2, the user can type the line number followed by the correct portion of the SQL statement. This corrects the SQL statement in the buffer. The last step is to execute the SQL statement in the buffer by typing the RUN command.

```
SQL> 2 FROM
SQL> RUN

TNAME                          TABTYPE
------------------------------ -------
MACHINE                        TABLE
EMPLOYEE                       TABLE
```

The LIST command can be used to display the current contents of the SQL buffer. An asterisk (*) is used to mark the current line of the SQL statement within the buffer.

```
SQL> LIST
  1  SELECT tname, tabtype
  2  FROM
  3* tab
```

The current line of the SQL statement in the buffer can be appended using the APPEND command. The syntax is: APPEND *new text*. In the following example the new text ``xyz'' is appended to line number 3 which is the current line in the buffer.

```
SQL> LIST
  1  SELECT tname, tabtype
  2  FROM
  3* tab

SQL> APPEND xyz
  3* tabxyz

SQL> LIST
  1  SELECT tname, tabtype
  2  FROM
  3* tabxyz
```

Text on a line in the SQL statement can also be replaced using the CHANGE command. The syntax for the CHANGE command is: CHANGE / *old text* / *new text* /

In the following example, text on the current line number 3 will be replaced with blank text:

```
SQL> LIST
  1  SELECT tname, tabtype
  2  FROM
  3* tabxyz

SQL> CHANGE/xyz//
  3* tab

SQL> LIST
  1  SELECT tname, tabtype
  2  FROM
  3* tab
```

To move to a different line of the SQL statement in the buffer, simply type the line number.

```
SQL> LIST
  1  SELECT tname, tabtype
  2  FROM
  3* tab

SQL> 2
  2* FROM

SQL> LIST
  1  SELECT tname, tabtype
  2* FROM
  3  tab
```

The DEL command can be used to delete the current line of the SQL statement out of the buffer as in the following example.

```
SQL> LIST
  1  SELECT tname, tabtype
  2* FROM
  3  tab

SQL> DEL
  2

SQL> LIST
  1  SELECT tname, tabtype
  2* tab
```

A SQL statement in the buffer can be saved to a file for later use. The SAVE command serves this purpose. The syntax for the SAVE command is: SAVE *filename*

In this example, the current contents of the buffer are saved to a file called query.sql:

```
SQL> LIST
  1  SELECT tname, tabtype
  2* FROM
  3  tab
```

```
SQL> SAVE query.sql
```

A directory and/or drive letter (for those using MS DOS or MS Windows) can be placed in front of the file name in order to re-direct the file to another drive or directory. For example, to save the current statement to a floppy disk:

```
SQL> SAVE a:\query.sql
```

A SQL statement saved in a file can then be loaded and executed using the START command. The syntax for the START command is: START *filename*

Here, the file query.sql created in the previous example is loaded and executed using the START command:

```
SQL> START query.sql

TNAME                          TABTYPE
------------------------------ -------
MACHINE                        TABLE
EMPLOYEE                       TABLE

SQL>
```

Again, a drive letter and/or directory name can be placed in front of the file name.

In many cases, it is easiest to create and edit a set of text files containing the queries and then use the START command to execute them. Instructions for this vary depending on the operating system. For example, under a UNIX system, one can use a text editor such as `VI`, `Emacs` or `Pico` to create text files with the create statements to create the tables, insert statements to add data and select statements to perform some queries.

Under MS Windows 95 or NT, one can use the Windows NotePad editor to create these same types of files. If the files are stored on a floppy disk (for example, the `a:` drive), then the START command can be used as follows:

```
SQL> START a:\query.sql
```

When working with SQL statements and SQL*Plus commands in a script file, be sure and make backups of your disks and files.

5.2 Formatting SQL*Plus Output

SQL*Plus contains several commands that can alter the appearance of the output. These commands are only in effect for the current SQL*Plus session. They can also be included in SQL script files and can be executed using the START command.

The formatting commands include:

- BREAK - Set the formatting behavior for the output of SQL statements
- BTITLE - Place a title on the bottom of each page in the printout from a SQL statement
- COLUMN - Change the appearance of an output column from a query
- REMARK - Place a comment following the REMARK keyword
- SET - Set a SQL*Plus variable to a new value
- SHOW - Show the current value of a SQL*Plus variable
- TTITLE - Place a title on the top of each page in the printout from a SQL statement
- UNDEFINE - Delete a user defined variable

Note that none of these SQL*Plus formatting commands changes the underlying table structures.

Perhaps the most useful command is `COLUMN` which changes the appearance of data for a given column. The syntax for the `COLUMN` command is as follows:

```
COLUMN column_name option1 option2 ...
```

Where `option` can be one or more of the following:

- `FORMAT` *`format`* - Changes the format for the column. For example, to only display the first 10 characters of an employee's last name (column LNAME), use the following:

  ```
  COLUMN lname FORMAT A10
  ```

  This indicates to format the lname column as an Ascii column with only 10 characters. Numbers can be formatted using "9" to indicate digits. For example:

  ```
  COLUMN salary FORMAT $9,999,990.99
  ```

- `HEADING` *`heading_text`* - changes the heading for a column.
- `JUSTIFY LEFT` or `JUSTIFY CENTER` or `JUSTIFY RIGHT` - aligns the output with the left, center or right of the column.
- `NULL` *`text`* - Indicates the text that should be displayed in place of a NULL value.
- `WRAPPED` or `WORD_WRAPPED` or `TRUNCATED` - Indicates how text that is longer than the displayed column width should be handled. `TRUNCATED` means it will be cut off at the maximum width of the field. `WRAPPED` will wrap the value down to the next line of output. `WORD_WRAPPED` does the same as Wrapped but breaks the value up on white space.

The `SET` and `SHOW` commands can also be useful. To see a listing of all of the SQL*Plus variables, type `SHOW ALL`

```
SQL> show all
appinfo is ON and set to "SQL*Plus"
arraysize 15
autocommit OFF
autoprint OFF
autotrace OFF
shiftinout INVISIBLE
blockterminator "." (hex 2e)
btitle OFF and is the 1st few characters of the next SELECT
```

```
cmdsep OFF
colsep " "
compatibility version NATIVE
concat "." (hex 2e)
copycommit 0
copytypecheck is ON
define "&" (hex 26)
echo OFF
editfile "afiedt.buf"
embedded OFF
escape OFF
feedback ON for 6 or more rows
flagger OFF
flush ON
heading ON
headsep "|" (hex 7c)
linesize 100
lno 24
loboffset 1
long 80
longchunksize 80
newpage 1
null ""
numformat ""
numwidth 9
pagesize 24
pause is OFF
pno 0
recsep WRAP
recsepchar " " (hex 20)
release 800030000
repfooter OFF and is NULL
repheader OFF and is NULL
serveroutput OFF
showmode OFF
spool OFF
sqlcase MIXED
sqlcode 0
sqlcontinue "> "
sqlnumber ON
sqlprefix "#" (hex 23)
sqlprompt "SQL> "
sqlterminator ";" (hex 3b)
suffix "SQL"
tab ON
termout ON
time OFF
timing OFF
trimout ON
trimspool OFF
ttitle OFF and is the 1st few characters of the next SELECT
underline "-" (hex 2d)
user is "HOLOWCZAK"
verify ON
wrap : lines will be wrapped
```

Some of the SQL*Plus variables of interest include `BTITLE` and `TTITLE` described above, Other useful variables include:

- SET COLSEP - The separator characters between columns (default is a space)
- SET ECHO - Determines if SQL*Plus commands should be echoed or not. The default is `OFF` meaning SQL*Plus commands will not be echoed.
- SET HEADING - Determines if column headings should be displayed or not. Default is `ON`.
- SET LINESIZE - Determines the maximum number of characters in a line of output. The default is 80.
- SET LONG - Determines how much data in a `LONG` column will be displayed. The default is 80 bytes.
- SET NULL - Determines what value should be displayed in place of a NULL value.
- SET PAGESIZE - The number of lines for one page (before the headers repeat). The default is 24 lines.
- SET PAUSE - Determines if output should be paused after `PAGESIZE` lines have been displayed. The default is `OFF`.
- SET TRIMOUT - Determines if each line should be padded with blanks out to a length of `LINESIZE`. The default is `ON` meaning the output will be trimmed (e.g., will not be padded with spaces).
- SET WRAP - Determines if the output will be wrapped to the next line or truncated if the line is longer than `LINESIZE`. The default is `ON` meaning long output will be wrapped.

The following example shows an SQL*Plus script file (`myquery.sql`) that utilizes some of the above formatting commands. The output of this script is displayed afterwards.

```
TTITLE 'Employees, Departments and Department Managers'
SET PAGESIZE 36
COLUMN address      FORMAT A20 WORD_WRAPPED
COLUMN dept_manager FORMAT A13 WORD_WRAPPED HEADING 'Dept. Manager'
COLUMN dno          FORMAT 999
SELECT employee.fname, employee.lname, employee.address,
       employee.dno, department.dname,
       employee2.fname || ' ' || employee2.lname  dept_manager
FROM   employee, department, employee employee2
WHERE  employee.dno = department.dnumber
   AND department.mgrssn = employee2.ssn ;
```

When the above script is executed, the following output is displayed:

```
SQL> START a:\myquery.sql

Mon Jan 05
                        Employees, Departments and Department Managers

FNAME    LNAME    ADDRESS              DNO DNAME          Dept. Manager
-------- -------- ------------------- --- -------------- -------------
JOHN     SMITH    731 FONDREN,           5 RESEARCH       FRANKLIN WONG
                  HOUSTON, TX

FRANKLIN WONG     638 VOSS,HOUSTON TX    5 RESEARCH       FRANKLIN WONG
RAMESH   NARAYAN  975 FIRE OAK,          5 RESEARCH       FRANKLIN WONG
                  HUMBLE, TX

JOYCE    ENGLISH  5631 RICE, HOUSTON,    5 RESEARCH       FRANKLIN WONG
                  TX

JAMES    BORG     450 STONE, HOUSTON,    1 HEADQUARTERS   JAMES BORG
```

```
                  TX

ALICIA   ZELAYA   3321 CASTLE, SPRING,    4 ADMINISTRATION JENNIFER
                  TX                                      WALLACE

JENNIFER WALLACE  291 BERRY, BELLAIRE,    4 ADMINISTRATION JENNIFER
                  TX                                      WALLACE

AHMAD    JABBAR   980 DALLAS, HOUSTON,    4 ADMINISTRATION JENNIFER
                  TX                                      WALLACE

8 rows selected.
```

5.3 Saving SQL*Plus Output using the SPOOL Command

SQL*Plus has a command called SPOOL that can send the output from any SQL statement to a file. Indeed, anything that is displayed in SQL*Plus can be echoed to this spool file.

The SPOOL command is invoked with the name of a file that will contain the output. Once this has been executed, the output from all subsequent SQL statements will be copied to the file. To end capturing the output, issue the SPOOL OFF command. The following is an example:

```
SQL> SPOOL a:\myfile.out

SQL> SELECT * from EMPLOYEE;

etc.     Any SQL statements typed here will show up in the output.

SQL> SPOOL OFF
```

The SPOOL OFF command turns the output off. Everything between SPOOL a:\myfile.out and SPOOL OFF will be in the file myfile.out. This is a simple ASCII text file that can be read by Windows Notepad, MS Word, or just about any word processor, e-mail package, etc. To print, load this file into MS Word, set the font to Courier and print as you would with any other document.

Note that some SQL*Plus commands will not show up in the SPOOL file. To have them echo to the SPOOL file, use the SET ECHO ON option.

Also, when SPOOLing to a file, SQL*Plus makes each line 80 characters long by padding with spaces. This can be shortened to fewer characters using the SET LINESIZE option. For example, SET LINESIZE 70 will pad each line of output to 70 characters.

The SET TRIMOUT and SET TABS options offer other ways to change the spooled output.

5.4 Prompting and Accepting user Input

SQL*Plus has several commands that can be used to prompt the user for input, accept input from the user and store it in a variable, and then use that variable in a query.

The following example shows the prompt/accept sequence for a query.

```
PROMPT Type the department you are looking for
ACCEPT dept NUMBER PROMPT "Department Number: "
SELECT fname, lname, dno
FROM   employee
WHERE  dno = &dept ;
```

When this script is executed, the following output is shown:

```
SQL>  START a:\empquery.sql
Type the department you are looking for
Department Number: 5
old   3: WHERE dno = &dept
new   3: WHERE dno =         5

FNAME    LNAME          DNO
-------- -------- ---------
JOHN     SMITH            5
FRANKLIN WONG             5
RAMESH   NARAYAN          5
JOYCE    ENGLISH          5
```

In the above example, the user typed "5" in response to the *Department Number:* prompt.

The first PROMPT command simply echoes out a line to the display. The second command, ACCEPT, accepts input from the user. In this case, the variable that will hold the input is called `dept`. The input should be of type `NUMBER`, and the PROMPT `Department Number:` should be displayed.

Once the user types the department number and presses enter, the variable `dept` takes on the value. The following two lines (starting with old and new) are verifying the values used for the `dept` variable. To suppress the display of this verification, use the `SET VERIFY OFF` command before running the script or as one of the first commands in the script.

In the following example, additional commands have been added including the SET VERIFY and REMARK commands to improve the script.

```
REMARK   This script accepts a department number as
REMARK   input from the user and then displays the
REMARK   last name, first name, address and department number
REMARK   of the employees in that department
REMARK   Turn off VERIFY
SET VERIFY OFF

REMARK   Format some columns
COLUMN dno     FORMAT 99999 HEADING 'Dept.|Number'
COLUMN address FORMAT A25   HEADING 'Address'
COLUMN fname   FORMAT A10   HEADING 'First|Name'
COLUMN lname   FORMAT A12   HEADING 'Last|Name'

REMARK   Prompt the user and get a department number
PROMPT Type the department you are looking for
```

```
ACCEPT dept NUMBER PROMPT "Department Number: "

REMARK   Perform the query
SELECT fname, lname, address, dno
FROM   employee
WHERE  dno = &dept ;
```

When this script is executed, the following output is shown:

```
SQL>  START a:\empquery.sql
Type the department you are looking for
Department Number: 4

First      Last                                    Dept.
Name       Name         Address                   Number
---------- ------------ ------------------------- ------
ALICIA     ZELAYA       3321 CASTLE, SPRING, TX        4
JENNIFER   WALLACE      291 BERRY, BELLAIRE, TX        4
AHMAD      JABBAR       980 DALLAS, HOUSTON, TX        4
```

5.5 Collecting Statistics On SQL Statements

SQL*Plus has several commands that monitor the execution of SQL statements. The commands can be used to gather statistical information for performance monitoring purposes. The first command is called `AUTOTRACE` and is used to trace the execution plan for an SQL statement. To use `AUTOTRACE`, a special table must be created in the schema to hold the statistical information. Execute the following `CREATE TABLE` command in your schema:

```
create table PLAN_TABLE (
 statement_id    varchar2(30),
 timestamp       date,
 remarks         varchar2(80),
 operation       varchar2(30),
 options         varchar2(30),
 object_node     varchar2(128),
 object_owner    varchar2(30),
 object_name     varchar2(30),
 object_instance numeric,
 object_type     varchar2(30),
 optimizer       varchar2(255),
 search_columns  number,
 id              numeric,
 parent_id       numeric,
 position        numeric,
 cost            numeric,
 cardinality     numeric,
 bytes           numeric,
 other_tag       varchar2(255),
 partition_start varchar2(255),
 partition_stop  varchar2(255),
 partition_id    numeric,
 other           long);
```

This table need only be created once. No data is permanently stored in PLAN_TABLE so it will not take much if any space.

To check the execution plan for each SQL statement, turn the AUTOTRACE option on with the following SQL*Plus command:

```
   SET AUTOTRACE ON
```

Then execute an SQL statement:

```
SQL>  SELECT * FROM employee;
FNAME    MI LNAME          SSN BDATE     ADDRESS
-------- -- -------- --------- --------- ------------------------
JOHN     B  SMITH    123456789 09-JAN-55 731 FONDREN, HOUSTON, TX
FRANKLIN T  WONG     333445555 08-DEC-45 638 VOSS,HOUSTON TX
ALICIA   J  ZELAYA   999887777 19-JUL-58 3321 CASTLE, SPRING, TX
JENNIFER S  WALLACE  987654321 20-JUN-31 291 BERRY, BELLAIRE, TX
RAMESH   K  NARAYAN  666884444 15-SEP-52 975 FIRE OAK, HUMBLE, TX
JOYCE    A  ENGLISH  453453453 31-JUL-62 5631 RICE, HOUSTON, TX
AHMAD    V  JABBAR   987987987 29-MAR-59 980 DALLAS, HOUSTON, TX
JAMES    E  BORG     888665555 10-NOV-27 450 STONE, HOUSTON, TX

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'EMPLOYEE'

Statistics
----------------------------------------------------------
          0  recursive calls
          3  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
       1891  bytes sent via SQL*Net to client
        651  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          8  rows processed
```

Notice the regular output from the SQL command is given followed by the Execution plan and the statistics.

To turn off AUTOTRACE, issue the following command:

```
  SET AUTOTRACE OFF
```

The next useful command for performance monitoring is called TIMING. This command starts a timer that can be read at any interval, similar to how a stopwatch operates. To set up a timer, issue the following command:

```
SQL> TIMING START select_emp
SQL> SELECT * FROM employee ;

FNAME    MI LNAME          SSN BDATE
-------- -- -------- --------- ---------
JOHN     B  SMITH    123456789 09-JAN-55
FRANKLIN T  WONG     333445555 08-DEC-45
ALICIA   J  ZELAYA   999887777 19-JUL-58
JENNIFER S  WALLACE  987654321 20-JUN-31
RAMESH   K  NARAYAN  666884444 15-SEP-52
```

```
JOYCE    A  ENGLISH  453453453 31-JUL-62
AHMAD    V  JABBAR   987987987 29-MAR-59
JAMES    E  BORG     888665555 10-NOV-27

8 rows selected.

SQL> TIMING SHOW select_emp
timing for: select_emp
 real: 1760
```

Thus the above query took 1.76 seconds to complete.

To stop a timer, issue the `TIMING STOP` command. Note that AUTOTRACE and TIMING should probably not be used in conjunction as it would be difficult to separate the execution time for the SQL statement from the time taken to generate the plan and statistics.

5.6 Using the ALTER SESSION Statement for Date Formats

In the previous examples of SQL statements, the default format of data of type DATE has been in the form: `DD-MON-YY`

The TO_CHAR and TO_DATE functions can be used to convert dates to other formats, however, this may become inconvenient, especially when inserting a large number of rows.

The ALTER SESSION statement can be used to alter various characteristics of the current SQL*Plus session including the default date format. This statement is often used to format dates to conform to regional customs. The syntax of ALTER SESSION for use with changing the default date format is as follows:

```
ALTER SESSION
SET NLS_DATE_FORMAT = <date_format>
```

The *date_format* can include the following codes:

`YY` A 2 digit year such as 98.
`YYYY` A 4 digit year such as 1998.
`NM` A month number.
`MONTH` The full name of the month.
`MON` The abbreviated month (Jan, Feb, Mar).
`DDD` The day of the year. For use is Julian dates.
`DD` The day of the month.
`D` The day of the week.
`DAY` The name of the day.
`HH` The hour of the day (12 hour clock)
`HH24` The hour of the day (24 hour clock)
`MI` The minutes.

`SS` The seconds.

For example, to change the default date to include a full four digit year, issue the following ALTER SESSION statement:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY'
```

From this point, all INSERT, UPDATE and DELETE statements must format the date accordingly. Also, any SELECT statements will return the date formatted accordingly.

Note that this change only remains in effect for the current session. Logging out of SQL*Plus and logging back in (or re-connecting to the Oracle database using the `connect` command) will reset the date format back to its default.

---

6. Stored Procedures and Triggers

The Oracle RDBMS has the ability to store procedures within the data dictionary and execute procedures in the RDBMS. Procedures (Program Units) are written in the PL/SQL language (Procedural Language), which is proprietary to Oracle. PL/SQL runs in both the database engine as well as in many of Oracle's development tools such as Oracle Developer.

The PL/SQL language has all of the conditional (IF ...THEN) looping (WHILE), assignment, variable declaration and other language constructs of a complete programming language. SQL statements may be freely mixed in with the other programming statements. The major change to SQL is the syntax of the SELECT statement. All SELECT statements in PL/SQL must use the INTO clause to redirect the rows returned by the SELECT into variables. The syntax of the SELECT statement is:

```
SELECT <column1, column2, . . .>
INTO   <var1, var2, . . .>
FROM   <table1, table2, . . .>
WHERE  <where clause>
GROUP BY <column1, column2, . . .>
HAVING   <having clause>
ORDER BY <column1, column2, . . .>
```

Variables named in the INTO clause correspond to the order of columns selected in the SELECT clause. For example:

```
DECLARE
   empsalary     NUMBER;
   empdepartment NUMBER;
BEGIN
   SELECT   employee.salary, employee.dno
   INTO     empsalary, empdepartment
   FROM     employee
   WHERE    employee.lname = 'SMITH';
```

```
   IF (empdepartment = 1) THEN
      UPDATE employee
      SET    salary = empsalary * 1.03
      WHERE  employee.lname = 'SMITH';
   END IF;
END;
```

The above PL/SQL block declares two variables and then executes a SELECT statement returning the salary in PL/SQL variable *empsalary* and the department number in PL/SQL variable *empdepartment* for employee SMITH. If the *empdepartment* is equal to 1 then an SQL UPDATE statement is executed.

It is possible that a SELECT...INTO statement can return more than on row or record, or no records at all. In such situations, the entire SELECT statement will fail resulting in what is called an EXCEPTION. EXCEPTIONs in PL/SQL must be handled (taken care of) by some code. Most all triggers and stored procedures that use SELECT...INTO have EXCEPTION handling code.

The EXCEPTION code the following syntax:

```
 EXCEPTION
   WHEN <exception_name> THEN
   BEGIN
   ...
   END;
   WHEN <exception_name> THEN
   BEGIN
   ...
   END;
   WHEN OTHERS THEN
   BEGIN
   ...
   END;
```

To continue the above example, the exception code would appear as follows at the end of the regular stored procedure code:

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
   BEGIN
     RAISE_APPLICATION_ERROR(-20610,
           'No employee with last name SMITH found');
   END;
   WHEN TOO_MANY_ROWS THEN
   BEGIN
     RAISE_APPLICATION_ERROR(-20612,
           'More than one employee with last name SMITH found');

   END;
```

There are two main ways of storing PL/SQL code in the Oracle database: CREATE PROCEDURE and CREATE TRIGGER. Triggers are procedures that are executed in response to some event. Events include the execution of a DML statement on a table (such as INSERT, UPDATE, DELETE, MODIFY). Procedures are typically used to implement general program logic that can be shared across applications, triggers and utilities. A procedure must be explicitly called by an application, trigger or program.

It is common practice to store general business rule checking in procedures. This allows applications to check data validity before a transaction is submitted to the database. Triggers can also call the procedures to check data at the database level. Since the business rules are coded in a single set of procedures, maintenance of this code is simplified. In this section, we will introduce the syntax for creating triggers and demonstrate the use of a trigger to enforce a business rule.

6.1 Trigger Syntax

Creating a trigger is accomplished with the CREATE TRIGGER statement. There are numerous options for a trigger that specify when the trigger should fire. These options include:

- The SQL statement (INSERT, UPDATE, DELETE, SELECT) that causes the event. An event can include more than one SQL statement per trigger.
- The timing when the trigger code is executed. Options here include
    - BEFORE - The trigger code is executed before the effects of the SQL statement are put into place.
    - INSTEAD OF - The trigger code is executed instead of the normal SQL statement.
    - AFTER - The trigger code is executed after the normal SQL statement is processed.
- Some SQL statements such as UPDATE, DELETE and SELECT may affect more than one row. Triggers may be specified to fire once for the SQL statement or once for each row affected by the SQL statement.

6.2 Trigger Example

Here is an example trigger called *check_age* used to check if an employee is over the age of 16. This trigger will be executed in response to the events of INSERT or DELETE on the employee table. The *check_age* trigger code will be executed BEFORE the affects of the SQL statement are put into place. Finally, *check_age* will execute FOR EACH ROW affected by the SQL statement.

Lines starting with the double minus sign -- are comments and are ignored by the trigger.

```
CREATE OR REPLACE TRIGGER check_age
BEFORE INSERT OR UPDATE ON employee
FOR EACH ROW
DECLARE
   -- Declare two variables.
   years_old NUMBER;
   error_msg CHAR(180);
```

```
BEGIN
   --  The variable :new.bdate will be holding the new birth date
   --  of the record to be inserted or updated.  Subtract from
   --  the system date and divide by 365 to get years.
   years_old := ( (sysdate - :new.bdate) / 365);
   -- Now check to see if the new employee is under age.
   -- If so, then show an error.
   IF (years_old < 16) THEN
      error_msg :=  'Do not hire '  || :new.fname || ' ' ||
      :new.lname || '. They are only ' ||
      TO_CHAR(years_old, '99.9') || ' years old.';
      -- Signal the user there is a problem with this data.
      -- This also aborts the affects of the SQL statement
      -- for the current row.
      RAISE_APPLICATION_ERROR ( -20601, error_msg);
   END IF;
END;
```

After this code has been entered in SQL*Plus, an additional line will appear as if the SQL statement should continue. To complete entering the trigger code, type a forward slash `/` and the code will be submitted.

One of three things will happen when a new procedure or trigger is created:

1. If there are no syntax errors, the code will be compiled and the trigger will be stored in the user's schema. In this case, SQL*Plus will respond with a message that the trigger was created.
2. If there are some minor syntax errors within the code itself (between DECLARE and the last END statement), the trigger will still be created and stored in the database, however a message will be returned: `Trigger created with compilation errors.`

   To view the compilation errors check the USER_ERRORS view:
   `SELECT * FROM USER_ERRORS`
   or use the `SHOW ERRORS` SQL*Plus command.

3. Finally, if there are syntax errors in the CREATE OR REPLACE TRIGGER statement itself (such as if the employee table does not exist or one of the key words was misspelled), then the entire statement will be rejected and the trigger code will not be saved in the schema.

To see if the trigger compiled correctly, look in view USER_ERRORS as follows:

```
SQL> SELECT * FROM user_errors;

  no rows selected
```

If the message `no rows selected` appears, then no errors were found in the trigger.

Alternately, use the SQL*Plus command `SHOW ERRORS`.

To view the trigger code:

```
SQL> SET LONG 4096
SQL> SET PAGESIZE 90
SQL> SELECT * FROM user_triggers;
```

Or, in a more compact form:

```
SQL> SELECT trigger_name, trigger_body
     FROM   user_triggers
     WHERE  trigger_name = 'CHECK_AGE';
```

Once the trigger has been entered without syntax errors, it can be tested. See what happens when we attempt to insert a new employee record where the employee's birthdate is less than 16 years ago:

```
SQL> INSERT INTO employee VALUES ('Joe', 'K', 'Smith', 12332199,
  2  '08-JUN-81', '123 Smith St,', 'M', 32000, 888665555, 1);
INSERT INTO employee VALUES ('Joe', 'K', 'Smith', 12332199,
            *
ERROR at line 1:
ORA-20601: Do not hire Joe Smith. They are only  15.6 years old.
ORA-06512: at "HOLOWCZAK.CHECK_AGE", line 8
ORA-04088: error during execution of trigger 'HOLOWCZAK.CHECK_AGE'
```

6.3 Stored Procedure Example

The following example implements a simple inventory system. The Products table holds a list of products with a productid as the key and a description. The inventory location table holds a series of locations in the warehouse including an identifier and the aisle, tier and bin. Finally, the intersection of these two tables is the inventory table which takes a locationid and a productid and gives the quantity of the product present at the location.

SQL Statements to create and populate tables

The SQL code to create and populate the three tables is given below:

First step: Create three tables and add constraints

```
CREATE TABLE inventory_locations (
  locationid    NUMBER(10) NOT NULL,
  aisle         NUMBER(10),
  tier          NUMBER(10),
  bin           NUMBER(10) );

ALTER TABLE inventory_locations
ADD CONSTRAINT il_pk PRIMARY KEY (locationid);

CREATE TABLE products (
   productid    VARCHAR(10) NOT NULL,
   description  VARCHAR(35) );
```

```
ALTER TABLE products ADD CONSTRAINT prod_pk
PRIMARY KEY (productid);

CREATE TABLE inventory (
    locationid  NUMBER(10) NOT NULL,
    productid   VARCHAR(10) NOT NULL,
    quantity    NUMBER(10) );

ALTER TABLE inventory ADD CONSTRAINT inventory_pk
PRIMARY KEY (locationid, productid);
```

Next step: Add some data to the three tables

```
INSERT INTO inventory_locations VALUES (101, 1, 1, 1);
INSERT INTO inventory_locations VALUES (102, 1, 1, 2);
INSERT INTO inventory_locations VALUES (103, 1, 1, 3);
INSERT INTO inventory_locations VALUES (104, 1, 2, 1);
INSERT INTO inventory_locations VALUES (105, 1, 2, 2);
INSERT INTO inventory_locations VALUES (106, 1, 2, 3);
INSERT INTO inventory_locations VALUES (107, 2, 1, 1);
INSERT INTO inventory_locations VALUES (108, 2, 1, 2);

INSERT INTO products VALUES ('P500', 'HP LaserJet 6L');
INSERT INTO products VALUES ('P510', 'HP DeskJet 855');
INSERT INTO products VALUES ('P520', 'IBM Aptiva');
INSERT INTO products VALUES ('P530', 'Compaq Presario');

INSERT INTO inventory VALUES (101, 'P500', 5);
INSERT INTO inventory VALUES (102, 'P510', 10);
INSERT INTO inventory VALUES (103, 'P500', 10);
INSERT INTO inventory VALUES (104, 'P520', 1);
INSERT INTO inventory VALUES (105, 'P530', 5);
```

The following query shows the current state of the inventory:

```
SELECT i.locationid, aisle, tier, bin, i.productid, description, quantity
FROM   inventory i, inventory_locations il, products p
WHERE  i.locationid = il.locationid
  AND  i.productid = p.productid;
```

We can create a view to implement this query:

```
CREATE VIEW vinventory AS
SELECT i.locationid, aisle, tier, bin, i.productid, description, quantity
FROM   inventory i, inventory_locations il, products p
WHERE  i.locationid = il.locationid  AND  i.productid = p.productid;
```

To see the current state of the inventory, simply query the view:

```
SELECT * FROM vinventory;
```

To see the output from the stored procedures, set the following options (Note: You must do this each time you log into SQL*Plus).

```
SET SERVEROUTPUT ON
SET ARRAYSIZE 2
```

Stored Procedures to Add and Remove items From inventory

Adding a new product to an existing location requires the following:

1. Check to see if some quantity of that product is already in the location. If so, then UPDATE the quantity already there.

2. If that product is not currently in the location, then INSERT a new inventory record with the locationid, productid and new quantity.

The following Oracle PL/SQL implements the add_to_inventory procedure:

```
CREATE OR REPLACE PROCEDURE add_to_inventory (
    new_locationid IN NUMBER,
    new_productid  IN VARCHAR,
    new_quantity   IN NUMBER) AS
  current_quantity NUMBER;

BEGIN
  current_quantity := 0;

  -- See if some quantity exists at the current location
  -- If not, then raise EXCEPTION and insert a new record
  -- If so, then continue on to the UPDATE statement
  SELECT quantity
  INTO   current_quantity
  FROM   inventory
  WHERE  inventory.locationid = new_locationid
   AND   inventory.productid  = new_productid;

  -- If we get this far, then there must already exist
  -- an inventory record with this locationid and productid
  -- So update the inventory by adding the new quantity.
  IF (current_quantity > 0) THEN
    UPDATE inventory
    SET    quantity = quantity + new_quantity
    WHERE  inventory.locationid = new_locationid
      AND  inventory.productid  = new_productid;
  END IF;

  -- If the first SELECT statement above fails to return any
  -- records at all, then the NO_DATA_FOUND exception will be
  -- signalled. The following code reacts to this exception.
  EXCEPTION
     WHEN NO_DATA_FOUND THEN
     BEGIN
       -- Since an inventory record mathcing the locationid and
       -- productid can not be found, we must INSERT a new
       -- inventory record.
       INSERT INTO inventory
       (locationid, productid, quantity)
       VALUES (new_locationid, new_productid, new_quantity);
     END;
END;
```

Removing an existing product from inventory requires the following:

1. Check to see if the requested quantity of that product is available in the location. If so, then UPDATE the quantity.
2. If the resulting quantity falls to 0, then DELETE the inventory record.

```
CREATE OR REPLACE PROCEDURE remove_from_inventory (
    current_locationid IN NUMBER,
    current_productid  IN VARCHAR,
    quantity_to_remove IN NUMBER) AS
  current_quantity NUMBER;
```

```
  error_msg CHAR(180);
BEGIN
  current_quantity := 0;

  -- See if some quantity exists at the current location
  -- If not, then raise EXCEPTION and exit the procedure.
  -- If so, then continue on to the UPDATE statement
  SELECT quantity
  INTO   current_quantity
  FROM   inventory
  WHERE  inventory.locationid = current_locationid
   AND   inventory.productid  = current_productid;

  -- If we get this far, then there must already exist
  -- an inventory record with this locationid and productid
  -- So update the inventory by removing the quantity.
  IF (current_quantity - quantity_to_remove > 0) THEN
    UPDATE inventory
    SET    quantity = quantity - quantity_to_remove
    WHERE  inventory.locationid = current_locationid
      AND  inventory.productid  = current_productid;
  END IF;
  -- If the quantity to remove is the same as the current
  -- quantity in the location, then simply delete the
  -- entire record.
  IF (current_quantity - quantity_to_remove = 0) THEN
    DELETE FROM inventory
    WHERE  inventory.locationid = current_locationid
      AND  inventory.productid  = current_productid;
  END IF;
  IF (current_quantity - quantity_to_remove < 0) THEN
    error_msg := 'ERROR: Insufficient quantity in that location';
    RAISE_APPLICATION_ERROR (-20602, error_msg);
  END IF;

  -- If the first SELECT statement above fails to return any
  -- records at all, then the NO_DATA_FOUND exception will be
  -- signalled. The following code reacts to this exception.
  EXCEPTION
     WHEN NO_DATA_FOUND THEN
     BEGIN
       error_msg := 'ERROR: Product ' || current_productid ||
                             ' not found at this location';
       RAISE_APPLICATION_ERROR (-20603, error_msg);
     END;
END;
```

Note: When creating stored procedures in SQL*Plus, you must type a slash character (`/`) on the line after the last `END;` of the procedure. This lets SQL*Plus know to submit the CREATE PROCEDURE statement to the database.

To see any errors from the compilation of the procedure use the `show errors` command. To run the stored procedures, use the EXECUTE command followed by the name of the procedure and any parameters. For example, to add 10 units of product P500 to location 106, execute the following:

```
EXECUTE add_to_inventory(106, 'P500', 10)
```

To remove 10 units of procedure P500 from location 106, execute the following:

```
EXECUTE remove_from_inventory(106, 'P500', 10)
```

In this section, we have shown some basic forms of triggers and stored procedures. For additional information and examples on the PL/SQL language, please refer to the Oracle PL/SQL User's Guide and Reference.

---

7. Acknowledgements

Portions of this tutorial are based on the technical working paper:
Adam, N. R., Holowczak, R. H., Maeng, J-H.,
ORACLE SQL*Plus and SQL*Forms: An Introduction and Tutorial
Technical Working Paper 89, CRAMTD Project, Rutgers University.
September, 1994
