Chapter 2

More "Beginning" SQL Commands and Statements

In this chapter, we start by expanding the power of the SELECT statement. Then we show how to create tables, insert values into tables, as well as change values and delete rows from tables. The next section in this chapter discusses the transaction processing commands ROLLBACK, COMMIT, and SAVEPOINT, which can be used in multi-user environments. Transaction processing commands can be used to undo changes in the database within a transaction. The chapter closes with a discussion of common datatypes available in Oracle and includes an extended discussion of the DATE datatype.

2.1 An Extended SELECT Statement

The SELECT is *usually* the first word in a SQL statement. The SELECT statement instructs the database engine to return information from the database as a set of rows, a "result set." The SELECT displays the result on the computer screen but does not save the results. (Chapter 1 demonstrated how to save queries and results.)

The simplest form of the SELECT syntax is:

```
SELECT      attributes
FROM        Table;
```

A database consists of a collection of tables and each table consists of rows of data. The above statement returns a result set of zero or more rows which is drawn from specified columns (attributes) available in a table. In the above statement, "Table" is the name of the table in the database from which the data will be taken, and "attributes" are the selected columns (attributes) in the table. The keywords SELECT and FROM are *always* present in a SELECT statement in Oracle.

For example,

```
SELECT      name, address
FROM        Old_Customer;
```

would select the attributes *name* and *address* from the table, **Old_Customer**. The result set would consist of rows of names and addresses in no particular row-order.

Begin Note
The table, **Old_Customer**, has not been created for you. You would have to create this table before you could try out this statement.
End Note

An asterisk (*) in place of the *attributes* would mean "list all the attributes (or columns) of the table" (i.e., whole rows).

An example of using the asterisk would be:

```
SELECT      *
FROM        Student;
```

where the "*" means return "all columns" from the table, **Student**.

2.1.1 SELECTing Attributes (Columns)

All of the attributes (columns) and/or all of the rows do not have to be retrieved with a SELECT statement. When the SELECT is executed, what *is* shown is called the result set. Attribute names may be SELECTed from a table, provided the exact name of the attribute is known. To find out the exact name of the attributes, use "DESC *tablename*" (DESCribe) as discussed in Chapter 1.

As an example of using DESC in our **Student-Course** database, we have a table called **Student**. And, suppose we want to list all the student names from this **Student** table. First, use

```
DESC Student
```

and find that the attribute name for "student name" is *"sname"*:

```
SQL> DESC Student
```

Gives:

```
Name            Null?           Type
--------------- --------------- ------------------------
STNO            NOT NULL        NUMBER (38)
SNAME                           VARCHAR2 (20)
MAJOR                           VARCHAR2 (4)
CLASS                           NUMBER (38)
BDATE                           DATE
```

*****Begin Note*****
All SQL statements require semicolons or slashes to execute them. SQLPLUS commands do **not** require semicolons or a terminating character, but if you use one, SQLPLUS is usually forgiving and will execute the command correctly anyway. Hence, DESC does not require a semicolon, while the SELECT does.
*****End Note*****

To show a listing of student names, we use the following query:

```
SELECT      sname
FROM        Student;
```

This would give:

```
SNAME
------------------------------
Lineas
Mary
Brenda
Richard
Kelly
Lujack
Reva
```

```
.
.
.
Smith
Jake

48 rows selected.
```

Note the result set is unordered. *Why?*

2.1.1.1 Using ORDER BY

The result set (the output) of the above query contains all the *sname*-values (student names) in the **Student** table. But, the *sname*-rows in the result set are not ordered since a relational table does not guarantee its rows are in any particular order. Rows in relational database tables are supposed to be mathematical sets (no particular ordering of rows and no duplicate rows); result sets are similar to mathematical sets because no order is implied, but in SQL duplicate values may occur. To show the contents of a table in a specific order, we can force the ordering of the result set using the ORDER BY clause in the SELECT.

For example, the query below will show the name (*sname*) and *major* from the **Student** table, ordered by *sname*. Here the output will be ordered in ascending order of *sname* because ascending order is the default of the ORDER BY clause. So if we type:

```
SELECT      sname, major
FROM        Student
ORDER BY    sname;
```

This will give:

```
SNAME       MAJO
----------- -----------
Alan        COSC
Benny       CHEM
Bill        POLY
Brad        COSC
Brenda      COSC
Cedric      ENGL
Chris       ACCT
Cramer      ENGL
Donald      ACCT
Elainie     COSC
Fraiser     POLY
.
.
.
Susan       ENGL
Thornton
Zelda       COSC

48 rows selected.
```

To order the output in descending order, the keyword DESC can be appended to the appropriate attribute in the ORDER BY clause as follows:

```
SELECT      sname, major
FROM        Student -- this time we ask for descending order
ORDER BY    sname DESC;
```

This will give:

```
SNAME              MAJO
-------------------- ----------
Zelda              COSC
Thornton
Susan              ENGL
Steve              ENGL
Stephanie          MATH
Smithly            ENGL
Smith
Sebastian          ACCT
Sadie              MATH
Romona             ENGL
Richard            ENGL
.
.
.
Bill               POLY
Benny              CHEM
Alan               COSC

48 rows selected.
```

Begin Note
In the above query, the comment cannot be inserted after the semicolon. The semicolon has to be the last thing in a query, marking the end of a query.
End Note

The collection of names and majors are the same in both the previous result sets, but the names in the latter result are in descending (reverse) order by *sname*.

We may also order within an order. For example, suppose we type:

```
SELECT          sname, major
FROM            Student
ORDER BY major DESC, sname;
```

The result is:

```
SNAME              MAJO
-------------------- ---------
Lionel
Smith
Thornton
Genevieve          UNKN
Lindsay            UNKN
Bill               POLY
Fraiser            POLY
George             POLY
Harley             POLY
Holly              POLY
Jessica            POLY
```

```
Ken             POLY
Lynette         POLY
Jake            MATH
Kelly           MATH
Mario           MATH
Monica          MATH
Reva            MATH
Sadie           MATH
Stephanie       MATH
Cedric          ENGL
Cramer          ENGL
.
.
.
Donald          ACCT
Francis         ACCT
Harrison        ACCT
Sebastian       ACCT

48 rows selected.
```

Here the output is principally ordered by *major* in descending order and then by *sname* within *major* with the names in ascending order.

*****Begin Note*****
The ascending order is the default order of the ORDER BY clause.
*****End Note*****

*****Begin Note*****
In this example, some names have no value for *major*. Unless prohibited by the table creator, nulls are allowed for values. Notice in the example null values sort last because in the example the values are presented in descending order.
*****End Note*****

2.1.2 SELECTing Rows

The output of rows in the result set may be restricted by adding a WHERE clause to the SELECT. When the WHERE clause is used, the database engine selects the rows from the table that meet the conditions given in the WHERE clause. If no WHERE clause is used, the query will return all rows from the table. In other words, the WHERE clause acts as a "row filter."

The simplest format of the SELECT with a WHERE clause would be:

```
SELECT      attribute(s)
FROM        Table
WHERE       criteria;
```

For example, to list the *sname* of only those students who are seniors we would type:

```
SELECT      sname
FROM        Student -- we add a row filter in the next line to show only seniors
WHERE       class = 4;
```

This will give:

```
SNAME
--------------------
Mary
Kelly
Donald
Chris
Holly
Jerry
Harrison
Francis
Jake
Benny

10 rows selected.
```

All of the comparison operators:

```
> (greater than),
<> not equal,
= equal,
> = greater than or equal to, and so on
```

are available for WHERE conditions.

Multiple conditions can be included in a WHERE clause by using logical operators, AND and OR. In addition there is also a BETWEEN operator. The following sections discuss the use of the AND, OR, and BETWEEN operators in the WHERE clause.

2.1.3 Using AND

By using AND in the WHERE clause of a SELECT, we may combine conditions. The result set with WHERE .. AND .. can never contain more rows than the SELECT with either of the conditions by themselves.

For example, consider the following query:

```
SELECT      sname, class, major
FROM        Student
WHERE       class = 4 -- both conditions must be true to retrieve a row
AND         major = 'MATH';
```

This gives us:

```
SNAME                CLASS      MAJOR
-------------------- ---------- ----------
Kelly                4          MATH

1 row selected.
```

The AND clause means both the conditions, "WHERE class = 4 **AND** major = 'MATH'," have to be met for the row to be included in the result set.

2.1.4 Using OR

Another way to combine conditions in a WHERE clause is by using the OR operator. The OR operator can be used when either of the conditions can be met for a row to be included in the result set. For example, consider the following query:

```
SELECT      sname, class, major
FROM        Student
WHERE       class = 4 -- either condition being true gives us a row
OR          major = 'MATH';
```

This gives us:

```
SNAME              CLASS           MAJOR
-------------------  -----------     ----------
Mary                   4             COSC
Kelly                  4             MATH
Reva                   2             MATH
.
.
.
Benny                  4             CHEM
Mario                                MATH
Jake                   2             MATH

16 rows selected.
```

This result set is a tabulation of all students who are either MATH majors OR seniors (*class* = 4). The OR means either of the criteria, "WHERE class = 4 **OR** major = 'MATH'," may be met for the row to be included in the result set.

It is not necessary to include all of the attributes used in the WHERE clause in the result set. It is a good idea to include the attributes when checking a query, but the following query is also legal:

```
SELECT      sname -- this is allowable, but why not include class and major too?
FROM        Student
WHERE       class = 4
OR          major = 'MATH';
```

Giving:

```
SNAME
--------------------
Mary
Kelly
Reva
Donald
.
.
.
Benny
Mario
Jake

16 rows selected.
```

2.1.5 Using BETWEEN

The BETWEEN operator returns rows when a value occurs within a given range of values. The general syntax of the BETWEEN operator is:

```
SELECT ...
FROM ...
WHERE       attribute
BETWEEN     value1 AND value2;
```

To find all the student rows with *class* values between 1 and 3 (inclusive), type:

```
SELECT      sname, class
FROM        Student
WHERE       class -- class = 1, 2 and 3 will satisfy the WHERE condition
BETWEEN     1 and 3;
```

This gives us:

```
SNAME               CLASS
------------------- ----------
Lineas              1
Brenda              2
Richard             1
Lujack              1
Reva                2
Elainie             1
Harley              2
.
.
.
Gus                 3
Jake                2

28 rows selected.
```

In Oracle SQL, *value1* has to be less than *value2*. The end points are included in the result set (BETWEEN is "inclusive"). The same "between *result*" could also be obtained using the query:

```
SELECT      sname, class
FROM        Student
WHERE       class >=1
AND         class <=3;
```

2.2 A Simple CREATE TABLE Statement

The CREATE TABLE statement allows us to create a table in which we can store data. A minimal syntax for this statement is as follows (we will expand this CREATE TABLE syntax with more options in a later chapter):

```
CREATE TABLE Tablename (attribute_name datatype, attribute_name datatype, ...);
```

Tablename and *attribute names* are your choice, but keywords should be avoided (words like "table" or "select"). A datatype defines the kind of data allowable for an attribute, e.g., numbers, alphanumeric characters, or dates. (We have a whole section on datatypes later in the chapter.)

In Example 1 below, we will be creating a table called **Customer**. Suppose the table has two attributes: customer number (we choose to abbreviate this as *cno*) and *balance*. The datatypes we choose are:

The *cno* attribute is a fixed-length character attribute with a length of 3 and will represent the customer number. The *balance* attribute is numeric with five digits and no decimals. The appendage "DEFAULT 0" means if no value is specified for *balance* when rows are inserted into the table, *balance* will be set equal to zero rather than null.

Example 1:

```
CREATE TABLE Customer
 (cno CHAR(3),
 balance NUMBER(5) DEFAULT 0);
```

Begin Note
In this example, if "DEFAULT 0" were not used, *balance* would default to NULL if no value for *balance* was supplied. NULL means "empty" and is Oracle's way of signifying no value is present.
End Note

We could have also used other datatypes for the attributes. For example, another common character datatype is VARCHAR2(n), which is a variable-length character string of length *n*. (Again, we will go into more in depth on datatypes later in the chapter.)

Example 2:
In this example, we are creating a table of names:

```
CREATE TABLE Names
 (name VARCHAR2(20));
```

This table, **Names**, has one attribute called, *"name." "name"* is of datatype VARCHAR2 (which means varying length character), and each *"name"* in the table can have a maximum size of 20 characters.

Begin Note
Older versions of SQL used a datatype called VARCHAR, but Oracle now uses and recommends the use of VARCHAR2.
End Note

2.2.1 Inserting Values into a Created Table

Values may be inserted into a created table using several methods. We will illustrate two of the three common ways to populate tables:

- INSERT INTO .. VALUES
- INSERT INTO .. SELECT
- SQLLOADER (for bulk loading of larger tables -- we do this later)

In this chapter we will look at INSERT INTO .. VALUES and INSERT INTO .. SELECT. We will discuss SQLLOADER later in the text because SQLLOADER is not a command *per se*, but rather a special Oracle procedure for loading tables.

2.2.1.1 INSERT INTO .. VALUES

The INSERT INTO with the VALUES option is a way of creating one row of a table. The following example inserts one row into the **Names** table:

```
INSERT INTO Names
VALUES ('Joe Smith');
```

where

- INSERT is the name of the command.
- INTO is a necessary keyword.
- "Names" is the name of the existing table.
- VALUES is another necessary keyword.
- 'Joe Smith' is a string of letters in agreement with the datatype.

'Joe Smith' is surrounded by single quotes. "Joe Smith" would be invalid.

If you created a table with n attributes, you usually would have n values in the INSERT INTO .. VALUES part of the command. For example, if you have created a table called **Employee**, like this:

```
CREATE TABLE Employee
    (name              VARCHAR2 (20),
    address            VARCHAR2 (20),
    employee_number    NUMBER (3),
    salary             NUMBER (6,2));
```

then the INSERT INTO .. VALUES to insert a row would match column for column and would look like this:

```
INSERT INTO Employee
VALUES ('Joe Smith', '123 4th St.', 101, 2500);
```

The values in the VALUES part of the statement correspond to the attribute names by their ordering as defined by the way the table was created. Character types must be enclosed in single quotes and numeric types are not in quotes. 'Joe Smith' corresponds to the attribute, *name*, and 2500 corresponds to *salary*. An INSERT like the following is incorrect because it does not include all four attributes of the **Employee** table:

```
INSERT INTO Employee
VALUES ('Joe Smith', '123 4th St.');
```

However, if you do not have data values for all four attributes of the **Employee** table, and you wish to insert values into only two of the four attributes, you can name the attributes you want to insert and provide values for only those attributes you name. For example, you can use an INSERT like this:

```
INSERT INTO Employee (name, address)
VALUES ('Joe Smith', '123 4th St.');
```

In this case, the inserted row will contain NULL values for the attributes you did not use.

An INSERT like the following is also incorrect because it does not have the values in the same order as the definition of the table:

```
INSERT INTO Employee
VALUES (2500, 'Joe Smith', 101, '123 4th St.');
```

If the data had to be specified in this order, the statement could be corrected by specifying the column names like this:

```
INSERT INTO Employee (salary, name, employee_number, address)
VALUES (2500, 'Joe Smith', 101, '123 4th St.');
```

The following INSERT would also be legal if the *address* and the *salary* were unknown when the row was created and if the *address* and *salary* attributes allowed nulls:

```
INSERT INTO Employee
VALUES ('Joe Smith', null, 101, null);
```

*****Begin Note*****

If you use non-numeric datatypes like CHAR (fixed character size) or VARCHAR2 (variable character size), you must use single quotes in the INSERT statement. If you use numbers, you should not use quotes. Oracle will convert character strings to numbers, but it is never good to let a system do something you should do yourself.

*****End Note*****

2.2.1.2 INSERT INTO .. SELECT

With the INSERT INTO ..VALUES option, you insert only one row at a time into a table. With the INSERT INTO .. SELECT option, you may (and usually do) insert many rows into a table at one time. The syntax of the INSERT INTO .. SELECT is:

```
INSERT INTO Table-name
 "SELECT clause"
```

For example, the following statement will insert all the values from the table **Customer** into another table called **Newcustomer**:

```
INSERT INTO Newcustomer
 SELECT      *
 FROM        Customer;
```

Before using this above INSERT INTO .. SELECT statement, you would have had to first create the **Newcustomer** table. Although the attributes of **Newcustomer** do not have to be named exactly what they are named in **Customer**, the datatypes and sizes have to match. The size of the attributes you are inserting (the size of the attributes of **Newcustomer**) have to be at *least* as big as the size of the attributes of **Customer**.

You may limit the SELECT and load to less than the whole table -- fewer rows or columns, as necessary. Following are some examples of restricted SELECTs for the INSERT statement:

Suppose you had a table with only one attribute, a customer name, which was created as follows:

```
CREATE TABLE Namelist
      (customer_name VARCHAR2(20));
```

Assume a second table existed with the following structure:

Customer1 (cname, cnumber, amount) where cname is VARCHAR2(20).

You can populate the **Namelist** table with the *cname* from the **Customer1** table as follows:

```
INSERT INTO Namelist
 SELECT      cname
 FROM        Customer1;
```

This would copy all the names from **Customer1** to **Namelist**. But, you do not have to copy all the names from **Customer1** because you can restrict the SELECT as illustrated in the following example:

```
INSERT INTO Namelist
 SELECT      cname
 FROM        Customer1
 WHERE       amount > 100
```

As with the INSERT INTO .. VALUES, if you create a table with n attributes, you usually would have n values in the INSERT INTO .. SELECT in the order of table definition. Suppose you have the following table:

Employee (name, address, employee_number, salary)

Then suppose you wanted to load a table called **Emp1** from **Employee** with the following attributes, *addr*, *sal*, and *empno*, which stand for address, salary, and employee number, respectively:

```
Emp1 (addr, sal, empno)
```

As with INSERT INTO .. VALUES, the INSERT INTO .. SELECT with no named attributes must match column for column and would look like the following:

```
INSERT INTO Emp1
 SELECT        address, salary, employee_number
 FROM          Employee;
```

The following INSERT would fail because the **Employee** table has four attributes while the **Emp1** table has only three:

```
INSERT INTO Emp1
 SELECT * FROM Employee;
```

The following INSERT would also fail because the attribute order of the SELECT must match the order of definition of attributes in the **Emp1** table:

```
INSERT INTO Emp1
 SELECT        address, employee_number, salary
 FROM          Employee;
```

As illustrated in the last INSERT INTO .. SELECT example, you can load fewer attributes than the whole row of the **Emp1** table using named attributes in the INSERT with a statement like:

```
INSERT INTO Emp1 (address, salary)
 SELECT        address, salary
 FROM          Employee;
```

However, this would leave the other attribute, *employee_number*, with a value of NULL or with a default value. Therefore, although loading less than a "full row" is syntactically correct, you must be aware of the result.

One final point: INSERT INTO .. SELECT could succeed if the datatypes of the SELECT matched the datatypes of the attributes in the table to which you are INSERTing. For example, if you had another table called **Emp2** with *name*, *address* as attributes (both defined as VARCHAR2), and if you executed the following, the statement *could* succeed, but you would have an address in a *name* attribute and vice versa:

```
INSERT INTO Emp2
 SELECT        address, name
 FROM          Employee;
```

*****Begin Note*****
We say "could" here because there are ways to prevent integrity violations of this type, but we have not introduced them yet.
*****End Note*****

Be careful with this INSERT INTO .. SELECT statement. Unlike INSERT INTO ..VALUES, which inserts one row at a time, you almost always insert multiple rows with INSERT INTO .. SELECT. If types match, the insert will take place regardless of whether it makes sense or not.

Finally, the previous two statements (CREATE TABLE and INSERT INTO) may be combined for creating backup copies of tables like this:

```
CREATE TABLE Course_copy AS
SELECT * FROM COURSE
```

You will get:

```
Table created.
```

2.2.2 The UPDATE Statement

Another common statement used for setting/changing data values in tables is the UPDATE statement. As with the INSERT INTO .. SELECT option, you often update more than one row at a time with UPDATE. To illustrate the UPDATE statement, you may create a table called **Customer2**, like this:

```
CREATE TABLE Customer2
 (cno CHAR(3), balance NUMBER(5), date_opened DATE);
```

Now suppose some values are inserted into the table using one of the above techniques. And then suppose you would like to set *all balances* in the new table to zero. You can do this with an UPDATE statement, as follows:

```
UPDATE Customer2
SET balance = 0;
```

This statement sets all balances in all rows of the table to zero, regardless of their previous value.

*****Begin Warning*****
Beware, this can be a dangerous. Later in the chapter we will discuss a method to safeguard against accidental misuse using the ROLLBACK statement.
*****End Warning*****

It is often useful and appropriate to include a WHERE clause on the UPDATE statement so values are set selectively. For example, the updating of a particular customer in our new table, **Customer2**, might be done with the following statement:

```
UPDATE      Customer2
SET         balance = 0          -- this updates only one customer
WHERE       cno = 101;
```

This would update only customer 101's row(s). You could also set specific balances to zero with other criteria in the WHERE clause with a statement like the following:

```
UPDATE      Customer2
SET         balance = 0
WHERE       date > '01-JAN-23';
```

However, in this last example, multiple rows might be updated.

2.2.3 The DELETE Statement

The DELETE statement is used to delete rows from tables. A sample of syntax for the DELETE statement is:

```
DELETE FROM Table
        WHERE (condition)
```

The (condition) determines which rows to delete.

*****Begin Warning*****
With UPDATE and DELETE, multiple rows can be affected; therefore, these can be dangerous commands. Be careful when using them and please wait until you learn to use ROLLBACK before applying these statements.
*****End Warning*****

An example of a multi-row delete from our sample **Customer2** table might be:

```
DELETE FROM  Customer2
WHERE        balance < 10;
```

or

```
DELETE FROM  Customer2
WHERE        date_opened < '01-JAN-17';
```

2.3 Deleting a Table

To remove a table from the database, you would use the DROP TABLE command as follows:

DROP TABLE *Table_name*;

Once you drop a table, you cannot bring the table or its data back. DROPping tables cannot be undone.

2.4 ROLLBACK, COMMIT, and SAVEPOINT

When we make modifications to our database -- when we use one or more INSERT, SELECT, DELETE, or UPDATE statements -- we perform a *transaction*. A transaction is defined as "a logical unit of work." A transaction ends with a COMMIT statement (either implied or explicit). There are statements containing implied COMMITs; i.e., if you execute these statements, it comes with a COMMIT; there is also an explicit COMMIT statement.

If you make a mistake in a transaction not including an implied COMMIT, you can undo whatever modification you have made to your database with a ROLLBACK statement. In this section we will discuss how and when to perform a COMMIT and a ROLLBACK, as well as discuss the conditions for undoing a transaction.

All transactions have a beginning and an end. A common beginning point for a transaction is when you log on to the database. Provided you have not issued a command with an implied COMMIT, the end point of the transaction is when you sign off. You may end a transaction by:

- Logging off of your database session (an implied COMMIT)
- Issuing a command containing an implied COMMIT (like the DROP TABLE command discussed in the previous section)
- Issuing an explicit COMMIT statement
- Executing a ROLLBACK statement

If you issue a COMMIT during a session, your transaction ends at that point and a new one begins. Data definition commands contain implicit COMMITs -- they end the current transaction and start a new one. Data definition commands we have seen are: CREATE TABLE and DROP TABLE. If either of these two commands are issued, an implied COMMIT ensues, the current transaction ends and a new transaction begins. Several transactions may take place within a single session. A "session" is bounded by when you log in to SQL and when you log off.

COMMIT and ROLLBACK are explicit transaction-handling statements to allow you to divide your work into separate transactions without signing off and on. Suppose you had a table of values and you deleted some of the rows. You can undo the delete action by issuing a ROLLBACK statement. For example, suppose while updating the **Customer** table, you issued the following DELETE statement:

```
DELETE FROM Customer
WHERE balance < 500;
```

This deletes all rows in the **Customer** table where *balances* are less than 500. Then, after the DELETE executes, you find your boss actually asked you to delete customers where the *balance* was less than 50, not 500. You can ROLLBACK the previous statement with:

```
ROLLBACK;
```

The ROLLBACK command resets the **Customer** table to whatever the values were at the beginning of the transaction. If you are sure you have successfully executed the correct command, you may execute:

```
COMMIT;
```

and the table will not be "ROLLBACK-able." After the COMMIT, the transaction is history.

The ROLLBACK command will not work under certain conditions because some statements contain implied COMMITs. Implied COMMITs are:

- When you use Data Definition Language (DDL) commands (DDL commands define or delete database objects). Examples of such commands include CREATE VIEW, CREATE TABLE, CREATE INDEX, DROP TABLE, RENAME TABLE, ALTER TABLE.
- When you log off of SQL, implicitly COMMITting your work.

For valuable tables, an explicit backup (or two) should be made and permissions for update and delete should be judiciously managed by the table owner. (We demonstrated backing-up tables earlier in the chapter.)

As an intermediate COMMIT/ROLLBACK action, you can also name a transaction milestone called a SAVEPOINT. For example, you can use the following command to mark a point in a transaction with the name, point1:

```
SAVEPOINT point1
```

You can then ROLLBACK to point1 with the following statement:

```
ROLLBACK TO SAVEPOINT point1
```

The naming of SAVEPOINTs allow you to have several ROLLBACK places in a transaction -- "milestones," if you will. These milestones allow partial ROLLBACKs. COMMIT is much stronger than a SAVEPOINT because it commits all actions and wipes out the SAVEPOINTs if there are any.

Following is an example of a transaction that includes a SAVEPOINT, ROLLBACK, and COMMIT:

Suppose we had a table, **CustA**, populated with attributes *name* and *balance*, defined as VARCHAR2(20) and NUMBER(5,2), respectively. Further suppose we type:

```
SELECT      *
FROM        CustA;
```

And we get:

```
NAME        BALANCE
----------  ----------
Mary Jo     25.53
Sikha       44.44
Richard     33.33
```

If we insert another row into **CustA**, as follows:

```
INSERT INTO    CustA
VALUES ('Brenda', 40.02);
```

We will get the following message:

```
1 row created.
```

We can now use this as a milestone, creating a SAVEPOINT by typing:

```
SAVEPOINT pointA;
```

We will then get the following message:

```
Savepoint created.
```

Now if we type:

```
SELECT         *
FROM           CustA;
```

We will get:

```
NAME           BALANCE
----------     ---------------
Mary Jo        25.53
Sikha          44.44
Richard        33.33
Brenda         40.02
```

If we type:

```
DELETE FROM  CustA
WHERE        balance < 35;
```

We will get the following:

```
2 rows deleted.
```

If we type:

```
SELECT         *
FROM           CustA;
```

We will get:

```
NAME           BALANCE
----------     ---------------
Sikha          44.44
Brenda         40.02
```

We could make this our next milestone, calling it **pointB**, by typing:

```
SAVEPOINT pointB;
```

Again, we will get the following message:

```
Savepoint created.
```

Now if we type:

```
DELETE FROM CustA;
```

We will get this message:

```
2 rows deleted.
```

If we now type:

```
SELECT      *
FROM        CustA;
```

We will get this:

```
no rows selected
```

If we feel we have made a mistake, we can, at this point, ROLLBACK the transaction as follows:

```
ROLLBACK TO SAVEPOINT pointB;
```

We will get the following message:

```
Rollback complete.
```

If we now type:

```
SELECT      *
FROM        CustA;
```

We will get:

```
NAME        BALANCE
----------  ---------------
Sikha       44.44
Brenda      40.02
```

We can update **CustA** by typing:

```
UPDATE          CustA
SET BALANCE =   55.55
WHERE           name LIKE 'Si%';
```

This will give us the following message:

```
1 row updated.
```

If we now type:

```
SELECT      *
FROM        CustA;
```

We will get:

```
NAME        BALANCE
----------  ---------------
Sikha       55.55
Brenda      40.02
```

If we want to now rollback to pointA, we type:

```
ROLLBACK TO pointA;
```

We will get the following message:

```
Rollback complete.
```

If we now type:

```
SELECT      *
FROM        CustA;
```

We will get:

```
NAME        BALANCE
----------  ----------
Mary Jo     25.52
Sikha       44.44
Richard     33.33
Brenda      40.02
```

At this point, if we COMMIT, we will basically wipe out the SAVEPOINTs and won't be able to ROLLBACK again. Suppose we issue a COMMIT:

```
COMMIT;
```

We get the following message:

```
Commit complete.
```

This completes the transaction in terms of making it impossible to ROLLBACK because the transaction is ended.

Most database situations occur in a multi-user environment. For example, suppose a **CustomerN** table exists with the following attributes:

```
CustomerN (name, address, credit_limit, balance)
```

Now suppose there are two departments using the **CustomerN** table: credit and billing. In a database, data is shared. Both the credit and billing departments use the **CustomerN** table. Now suppose we have two users: Richard and Sikha. Richard works for the credit department and Sikha works for billing. At the same time, Richard is updating the **CustomerN** table with new credit limits for some customers and Sikha is checking balances and credit limits.

Here is the point: Although both are using the same table, as Richard updates **CustomerN** rows, Sikha will not see Richard's changes until Richard COMMITs the changes (ends his transaction). The judicial use of COMMIT is an underlying principal of database and shared tables.

2.5 The ALTER TABLE Statement

The ALTER TABLE statement is used to alter the structure of a table. With the ALTER TABLE statement you can add/delete columns from tables and/or alter the size or datatypes of columns.

The simplified syntax to add a column to an existing table would be:

```
ALTER TABLE Tablename
ADD         column-name datatype
```

For example, the following will add the *address* attribute (of datatype VARCHAR2) to the **Customer** table created earlier in this chapter:

```
ALTER TABLE  Customer
ADD address VARCHAR2(20);
```

To change a column's type, the simplified syntax would be:

```
ALTER TABLE tablename
MODIFY column-name new_datatype
```

For example, the following will modify the *balance* attribute of the **Customer** table, making it a size of up to eight digits with two decimal places:

```
ALTER TABLE  Customer
MODIFY       balance NUMBER (8,2);
```

We can only make attributes larger and cannot violate any existing data with this statement.

Using the ALTER TABLE statement, we can define or change a default column value, enable or disable integrity constraints, manage internal space, and do some other useful things we will cover later.

*****Begin Note*****
If you modify a column, you can only make it bigger, not smaller, unless there is no data; all the data in the existing database must conform to your modified type.
If you add a column, it will contain null values until you put data into it with an UPDATE or INSERT command to change the values in the new column.
*****End Note*****

A little while ago you created a backup copy of the **Course** table like this:

```
CREATE TABLE Course_copy
AS SELECT    * FROM COURSE;
```

Now, we'll use the backup to provide an example of ALTER TABLE.

First, execute this command:

```
DESC Course_copy
```

This will give:

```
Name                      Null?              Type
------------------------  --------           --------------------------------
COURSE_NAME                                  VARCHAR2(20)
COURSE_NUMBER             NOT NULL           VARCHAR2(8)
CREDIT_HOURS                                 NUMBER(38)
OFFERING_DEPT                                VARCHAR2(4)
```

Now, to alter the table, type:

```
ALTER TABLE  Course_copy
MODIFY       offering_dept VARCHAR2(6);
```

This will give:

```
Table altered.
```

Now,

```
DESC Course_copy
```

Will give:

```
Name                    Null?           Type
----------------------- --------        --------------------------------
COURSE_NAME                             VARCHAR2(20)
COURSE_NUMBER           NOT NULL        VARCHAR2(8)
CREDIT_HOURS                            NUMBER(38)
OFFERING_DEPT                           VARCHAR2(6)
```

Now,

```
SQL> ALTER TABLE Course_copy
       MODIFY offering_dept CHAR(2)
SQL> /
```

Will give:

```
modify offering_dept char(2)
       *
ERROR at line 2:

ORA-01441: cannot decrease column length because some value is too big
```

2.6 Datatypes

A datatype of an attribute defines the allowable values as well as the operations we can perform on the attribute. We commonly use the NUMBER datatype for numbers and the CHAR and VARCHAR2 datatypes for character strings. In this section, we will explore these and other commonly and uncommonly used datatypes.

2.6.1 Common Number Datatypes

The most commonly used numeric datatypes in Oracle are NUMBER and INTEGER. The NUMBER datatype, with no parentheses, defaults to a number of up to 38 digits long with 8 decimal places. NUMBER may also be defined as having some maximum number of digits, such as NUMBER(5). Here, the (5) is referred to as the "precision" of the datatype and may be from 1 to 38. If a second number is included in the definition, for example NUMBER (12,2), the second number is called the "scale." The second number defines how many digits will come after a decimal point. Here, with NUMBER (12,2) we may have up to ten digits before the decimal point and two after.

A very common datatype used in programming languages is type INTEGER. INTEGER holds whole numbers and is equivalent to NUMBER(38).

Usually, you enter a precision and/or a scale for your numbers with entries such as NUMBER(3) or NUMBER(6,2). The NUMBER(3) implies you will have three digits and no decimal places. NUMBER(6,2) means the numbers you store will be similar to 1234.56 or 12.34, with a decimal before the last two digits in a field with a maximum of six numbers overall.

Here is an example to illustrate precision and scale with a numeric attribute. Type the following:

```
CREATE TABLE Testnum (x NUMBER(5,2));
```

Here we are creating a one column table with a numeric column called *"x."*

```
INSERT INTO Testnum VALUES (20);
INSERT INTO Testnum VALUES (200);
INSERT INTO Testnum VALUES (2000);
```

The first two INSERTs work fine, but the last INSERT gives an error:

```
INSERT INTO Testnum VALUES (2000);
                    *
ERROR at line 1:
ORA-01438: value larger than specified precision allows for this column
-
```

Then try typing:

```
INSERT INTO Testnum VALUES (200.12);
INSERT INTO Testnum VALUES (200.123);
```

Now,

```
SELECT *
FROM Testnum;
```

Will give:

```
         X
----------
        20
       200
    200.12
    200.12
```

If a number is inserted that is too large for the precision, an error results. If a number with too many decimal places is added, the decimal values beyond the scale are rounded up automatically, as shown by the following inserts:

```
INSERT INTO Testnum VALUES (123.99778);
INSERT INTO Testnum VALUES (333.333);
INSERT INTO Testnum VALUES (555.499999);
INSERT INTO Testnum VALUES (666.500004);
```

Now,

```
SELECT *
FROM Testnum;
```

Gives:

```
         X
----------
        20
       200
    200.12
    200.12
       124
    333.33
     555.5
     666.5
```

In addition to the above numeric datatypes, there are other specialty numeric types (SMALLINT, BINARY DOUBLE, and others). There are also the FLOAT datatypes, which allow large exponential numbers to be stored, but they are rarely used. Float datatypes include FLOAT, REAL, and DOUBLE PRECISION.

2.6.2 CHAR Datatype (commonly used)

CHAR (pronounced "care") is a fixed-length character datatype. This datatype is normally used when the data will always contain a fixed number of characters. For example, suppose major codes are always exactly four characters long; they should be encoded as CHAR(4). Social security numbers are also good candidates for this datatype because they always contain nine digits, so CHAR(9) can be used. Although Social Security (SS) numbers are digits, they are not used for calculation; hence, SS numbers may be stored as characters. In a field defined as CHAR, if the requisite number of characters is not inserted when loading data, the attribute will be padded on the right with blanks. For example, if we defined the social security number as CHAR(10) instead of CHAR(9), there would be one blank space on the right of every 9-digit social security number character string. The storage size limits for attributes of the CHAR datatype are from 1 to 2000 bytes.

2.6.3 VARCHAR2 Datatype (very commonly used)

As we mentioned earlier in the chapter, VARCHAR2 (pronounced "var-care") is Oracle's variable-length character datatype when loading data. For this datatype, maximum lengths are specified, as in VARCHAR2(20), for a string of from zero to 20 characters. When varying sizes of data are stored in an Oracle VARCHAR2, only the necessary amount of storage is allocated. This practice makes the internal storage of Oracle data more efficient. In fact, some Oracle practitioners suggest using only VARCHAR2(n) instead of CHAR(n). The minimum storage size for VARCHAR2 is 1 byte; the maximum size is 4000 bytes. Since there is no default size for VARCHAR2, you must specify a size.

*****Begin Note*****
Older versions of Oracle and other SQLs used VARCHAR (without the "2") instead of VARCHAR2. VARCHAR may not be supported in future versions, so we advise you to use VARCHAR2 instead.
*****End Note*****

2.6.4 NCHAR and NVARCHAR2 Datatypes (rarely used)

NCHAR stores fixed length character strings, and NVARCHAR2 stores variable length character strings, both of which are Unicode datatypes corresponding to the national character set. The character set of NCHAR and NVARCHAR2 datatypes are specified at database creation time. The maximum length of an NCHAR column is 2000 bytes, and the maximum size of a NVARCHAR2 column is 4000 bytes.

Unicode is an effort to enable encoding of every character in every known written language. Oracle database users with global applications may need to use Unicode data for non-English characters.

2.6.5 LONG, RAW, LONG RAW, and BOOLEAN Datatypes (rarely used except in very specific circumstances)

The LONG datatype is similar to VARCHAR2 and has a variable length of up to 2 gigabytes. However, there are some restrictions in the access and handling of LONG datatypes:

- Only one LONG column can be defined per table.
- LONG columns may not be used in subqueries, functions, expressions, WHERE clauses, or indexes.

A RAW or LONG RAW datatype is used to store binary data such as graphics characters or digitized pictures. The maximum size for RAW is 2000 bytes while the maximum size for LONG RAW is 2 gigabytes.

In Oracle, there is also a BOOLEAN datatype with values TRUE, FALSE, and NULL, but it is not often used.

*****Begin Note*****
The BOOLEAN datatype is only available when running the procedural language (PL/SQL). We will discuss PL/SQL in Chapter 11.
*****End Note*****

2.6.6 Large Object (LOB) Datatypes (rarely used except in very specific circumstances)

As of Oracle 8, four new large object (LOB) datatypes are supported: BFILE and three LOB datatypes -- BLOB, CLOB, and NCLOB. BFILE is an external LOB datatype that only stores a locator value pointing to the external binary file. BLOB is used for binary large objects, CLOB is used for large character objects, and NCLOB is a CLOB datatype for multi-byte character sets.

Data in the BLOB, CLOB, or NCLOB datatypes is stored in the database, although LOB data does not have to be stored with the rest of the table. Single LOB columns can hold up to 4 gigabytes in length and multiple LOB columns are allowed per table. In addition, Oracle allows you to specify a separate storage area for LOB data, greatly simplifying table sizing and data administration activities for tables containing LOB data. LOB datatypes consume large quantities of memory.

2.6.7 Abstract Datatypes (sometimes used, but uncommon)

In Oracle, you can also define and use abstract datatypes. An abstract datatype defines a *range* of allowable values as well as *operations* that can be performed. An abstract datatype (ADT) defines the operations explicitly (in methods or procedures) and should allow data to be accessed only by those methods. ADTs are created with the CREATE TYPE statement.

In addition to ADTs, the CREATE TYPE command allows you to create more complicated datatypes. For example, Oracle's *collection types* allow you to put a table within a table or allow a varying array in a table. Both of these concepts are non-third normal form (non-3NF) constructions and should be used only with a strong need to violate the 3NF assumption for relational database. These more exotic datatypes also may present performance problems for large databases.

*****Begin Note*****
A complete treatment of CREATE TYPE and abstract datatypes is a more advanced topic we will discuss later in this text. We mention CREATE TYPE here only to make you aware of its existence as an Oracle datatype.
*****End Note*****

2.6.8 The XML Datatype

XMLType has been created by Oracle to handle XML data. XML is a standardized textual coding technique used to exchange data over the internet. *Why does Oracle need an XML datatype?* The answer is some developers create and exchange data as XML. If Oracle did not have an XML datatype, then to handle the XML data in an Oracle database, there would have to be a bridge built to transform the XML data to a common SQL datatype or vice-versa. As with other datatypes, XMLType can be used as a datatype for a column of a table or view. The maximum size of an XMLType attribute is 4 gigabytes.

Using XML within SQL is beyond the scope of this material. Using XML involves data definition documents, style sheets, and other ancillary tools. The conversion of data to and from XML involves using CLOB datatypes in SQL and SQL procedures specifically designed to handle this new and exciting datatype.[1]

2.6.9 The DATE Datatype and Type Conversion Functions (commonly used)

A DATE datatype allows the storage and manipulation of dates and times. The time part of the datatype is often ignored, but it is defined as part of the datatype. There are functions to add, take differences between dates, convert to a four digit year, and so on. DATE datatypes are defined in a manner similar to what we have seen. Here is an example of a table containing a DATE datatype:

```
CREATE TABLE Date_example
    (day_test        DATE,
    amount           NUMBER(6,2),
    name             VARCHAR2(20));
```

Data is entered into the *day_test* attribute, in the character format 'dd-Mon-yy,' which automatically converts the character string to a date format.

Begin Note
The format of the DATE datatype can be changed by the DBA (Data Base Administrator), but dd-Mon-yy is commonly used.
End Note

Some examples of inserts for the **Date_example** table would be:

```
INSERT INTO date_example (day_test) VALUES ('10-oct-23')    /*  valid  */
INSERT INTO date_example (day_test) VALUES ('10-OCT-23')    /*  valid (month not case
     sensitive)  */
INSERT INTO date_example (day_test) VALUES (10-oct-23)      /* invalid (needs quotes) */
INSERT INTO date_example (day_test) VALUES (sysdate)        /* valid (system date) */
INSERT INTO date_example (day_test) VALUES ('10-RWE-23')  /* invalid (bad month)  */
INSERT INTO date_example (day_test) VALUES ('32-OCT-23')   /* invalid (bad day) */
INSERT INTO date_example (day_test) VALUES ('31-OCT-23')   /* valid  */
INSERT INTO date_example (day_test) VALUES ('31-SEP-23')   /* invalid (bad day --
     Oracle keeps up with the correct days per month)  */
```

For other than "standard" dates in the form 'dd-Mon-yy', the TO_DATE function can be used to insert values in other ways. The TO_DATE function has two arguments: TO_DATE (a,b), where "a" is the string you are using to enter the date and "b" is a recognizable Oracle character format.

For example, to insert the date '2-1-23' in the format 'mm-dd-yy' you would type:

```
INSERT INTO Date_example (day_test)
VALUES (TO_DATE ('2-1-23', 'mm-dd-yy'))
```

Likewise, to enter the date '2/1/2023' in the format 'mm/dd/yyyy,' you would type:

```
INSERT INTO Date_example(day_test)
VALUES (TO_DATE ('2/1/2023', 'mm/dd/yyyy'))
```

To convert a DATE datatype to a character datatype, another function, TO_CHAR is used. TO_CHAR is useful for displaying dates in formats other than the standard one, for example, if you type:

```
INSERT INTO Date_example
VALUES ('21-OCT-23',NULL, NULL);
```

And then you type:

```
SELECT          *
FROM Date_example;
```

You will get:

```
DAY_TEST        AMOUNT NAME
--------- ----------    -------------------------
21-OCT-23
```

And if you type:

```
SELECT          TO_CHAR(day_test,'mm/dd/yy') DDATE
FROM            Date_example;
```

This will give:

```
DDATE
-----------
10/21/23
```

And if you type:

```
SELECT          TO_CHAR(day_test,'Month dd, yyyy') DATE1
FROM            Date_example;
```

This will give:

```
DATE1
------------------
October 21, 2023
```

As mentioned previously, the DATE datatype may be used to store times. Consider an example in which we have expanded the input date to include the hour (using a 24-hour clock) and minute. Suppose we create a table like this:

```
CREATE TABLE Date_test2 (dte DATE);
```

We then INSERT some data as follows:

```
INSERT INTO Date_test2
VALUES (TO_DATE('2-11-2023 16:05','mm-dd-yyyy hh24:mi'))
```

A simple SELECT will show only the day, month, and year, as follows:

```
SELECT          dte
FROM            Date_test2;
```

We then get:

```
DTE
---------
11-FEB-23
```

The other stored information can be fully displayed using the TO_CHAR function:

```
SELECT          (TO_CHAR(dte,'dd-Mon-yyyy hh:mi:ss')) dtime
FROM            Date_test2;
```

Will give:

```
DTIME
--------------------
11-Feb-2023 04:05:00
```

We can specify other data like this:

```
SELECT      (TO_CHAR(dte,'dd-Mon-yyyy hh:mi:ss j q w PM cc')) dtime
FROM        Date_test2;
```

We get:

```
DTIME
---------------------------------------------
11-Feb-2023 04:05:00 1 2 PM 21
```

The format part of the TO_CHAR uses these formatting characters:

- The **q** is the quarter of the year (1st quarter).
- The **w** is the week of the month (2nd week).
- The **PM** signifies PM if PM and AM if AM.
- The **cc** specifies the century (21st).

2.6.9.1 Entering Four-Digit Years

There will often be times when we would want to enter and display four digit years. If there is the possibility of confusion when entering dates, then entering 4-digit years is safe and proper. To enter years as four digits, we use TO_DATE and a format to match the year part, for example:

```
INSERT INTO   Date_example (day_test)
VALUES (TO_DATE ('03-21-2023','mm-dd-yyyy'));
```

Now,

```
SELECT      *
FROM        Date_example
WHERE       TO_CHAR(day_test,'yyyy') = '2023'
```

Will give:

```
DAY_TEST      AMOUNT NAME
--------- ----------  --------------------
21-MAR-23
```

```
SELECT      TO_CHAR (day_test,'Month dd, yyyy') date2
FROM        Date_example
WHERE       TO_CHAR(day_test,'yyyy') = 2023;
```

Will give:

```
DATE2
------------------
March 21, 2023
```

There are also two handy functions to deal with dates: MONTHS_BETWEEN and ADD_MONTHS. Following are examples of these functions.

Today's date can be found with a statement like this:

```
SELECT      SYSDATE
FROM        Dual;
```

Dual is a dummy table that always returns one row. The **Dual** table and a query like the above are used for testing functions such as "sysdate" and/or variations such as TO_CHAR(SYSDATE, 'mm-day-yyyy'), as in:

```
SELECT      TO_CHAR(SYSDATE, 'mm-Day-yyyy')
FROM        Dual;
```

Now, to show how the MONTHS_BETWEEN function works, consider this example:

```
SELECT      SYSDATE
FROM        Dual;
```

This will give:

```
SYSDATE
---------------
11-APR-20
```

To find how many months there are from today back to February 2, 2014, you may use:

```
SELECT      MONTHS_BETWEEN(SYSDATE, '02-feb-14')
FROM        Dual;
```

Will give:

```
MONTHS_BETWEEN(SYSDATE,'02-FEB-20')
-----------------------------------
             74.3097708
```

And, for the ADD_MONTHS function, consider this example:

```
SELECT      ADD_MONTHS(SYSDATE, 4) four_months_out
FROM        Dual;
```

would give:

```
FOUR_MONTHS_OUT
-------------------------------
11-AUG-20
```

Finally, to change the default date format, the ALTER SESSION statement may be used:

```
ALTER SESSION SET nls_date_format = 'dd-mon-yyyy';
```

Footnote

[1] For a brief introduction to SQL and XML, see Earp, Richard Walsh and Bagui, Sikha Saha, *Advanced SQL Functions in Oracle 10g,* Wordware Publishing, Inc., 2006, Chapter 9.

Exercises for Chapter 2

As you do the exercises, it is a good idea to copy/paste your query as well as your query result into a word processor (review the SPOOL command in Chapter 1).

2-1. a. Create a table called **Cust** with a customer number field as a fixed-length character string of 3, an address field with a variable character string of up to 20, and a numeric balance of five digits (you choose the *attribute* names).

b. Insert values into the table with INSERT INTO .. VALUES. Use the form of INSERT INTO .. VALUES requiring you to have a value for each attribute; therefore, if you have a customer number, address, and balance, you must insert three values with INSERT INTO .. VALUES.

c. Create at least five rows (rows in the table) with customer numbers 101 to 105 and balances of 200 to 2000.

d. Display the table with a simple SELECT.

2-2. Show a listing of the customers from Exercise 2-1 in balance order (high to low) and use ORDER BY in your SELECT.

2-3. From the **Student-Course** database, use the **Student** table to display each student's *sname*, *class*, and *major* for freshmen or sophomores (*class* <= 2) in descending order of *class* (note: If you haven't created the synonyms of all the tables in the **Student-Course** database as per the Chapter 1 exercises, you will need to use **rearp.Student** instead of **Student** in the query).

2-4. From your **Cust** table, show a listing of only the customer balances in ascending order where *balance* > 400. You can choose some other constant if you want, for example, *balance* <= 600. The results will depend on your data. Make sure you use the attribute names you chose.

2-5. a. Create another table with the same types as **Cust** but without the customer address. Call this table **Cust1**. Use attribute names *cnum* for customer number and *bal* for balance. Load the table with the data you have in the **Cust** table with one less row. Use an INSERT INTO .. SELECT with appropriate attributes and an appropriate WHERE clause.

b. Display the resulting table. If it appears ok, COMMIT your work.

c. Assuming you have COMMITted in step b, delete about half of your rows from **Cust1** ("DELETE FROM **Cust1** WHERE bal < some value" [or bal > some value, etc.]).

d. Show the table after you have deleted the rows.

e. Undelete the rows with ROLLBACK.

f. Display the table with the reinstated rows.

g. Delete one row from the **Cust1** table and SAVEPOINT point1. Display the table.

h. Delete another row from the table and SAVEPOINT point2. Display the table.

i. ROLLBACK to SAVEPOINT point1, display the table, and explain what is happening.

j. Try to ROLLBACK to SAVEPOINT point2, see what happens, and explain it.

2-6. a. Using the **Cust1** table from the Exercise 2-5, COMMIT the table as it exists.

b. ALTER the table by adding a *date_opened* column of type date.

After each of the following, display the table.

c. Set the *date_opened* value in all rows to '01-JAN-23' and COMMIT.

d. Set all balances to zero, display the table, then ROLLBACK the action and display again.

e. Set the *date_opened* value of one of your rows to '21-OCT-23' and display.

f. Change the datatype of the *balance* attribute in **Cust1** to number (8,2). Display the table. Set the *balance* for one row to 888.88 and display the table again.

g. Try changing the datatype of *balance* to NUMBER (3,2). What happens? Why does this happen?

h. Change the values of all dates in the table to the system date using SYSDATE.

i. When you have finished the exercises (but be sure you are finished), "DROP TABLE **Cust1**" to delete the table. Use SELECT * FROM tab to be sure you dropped **Cust1**.

For the next three problems, use the **Student** table from our **Student-Course** database.

2-7. Using the **Student** table, list the *sname*, *major*, and *class* of all students who are ART majors and juniors.

2-8. Using the **Student** table, list the *sname*, *major*, and *class* of all students who are ART majors or juniors.

2-9. From the **Student** table, list the *sname*, *major*, and *class* of all sophomores, juniors and seniors. Use the BETWEEN operator for this.