Chapter 3

QUERY MULTIPLE TABLES

3.0 Objectives
3.1 Introduction
3.2 Joins
3.2.1 Equi-Join.
3.2.2 Non-Equi-Join.
3.2.3 Outer Join versus Inner Join
3.2.4 Joining Table to Itself.
3.3 Procedures and Functions
3.4 Creating a Procedure
3.5 Executing a Procedure
3.6 Deleting a Procedure
3.7 Functions
3.7.1 Aggregate Functions
3.7.2 Date & Time Function
3.7.3 Arithmatic Functions
3.7.4 Character Functions
3.7.5 Conversion Functions
3.7.6 Miscelleneous Functions
3.8 Summary
3.9 Check Your Progress - *Answers*
3.10 Questions for Self – Study
3.11 Suggested Readings

3.0 OBJECTIVES

After reading this chapter you will able to

- explain how to Creating procedure
- explain how to Executing procedure
- explain how to Deleting procedure
- describe Function

3.1 INTRODUCTION

Today you will learn about joins. This information will enable you to gather and manipulate data across several tables. By the end of the day, you will understand and be able to do the following :

- Perform an outer join
- Perform a left join
- Perform a right join
- Perform an equi-join
- Perform a non-equi-join
- Join a table to itself.

3.2 JOINS

One of the most powerful features of SQL is its capability to gather and manipulate data from across several tables. Without this feature you would have to store all the data elements necessary for each application in one table. Without common tables you would need to store the same data in several tables. Imagine having to redesign, rebuild, and repopulate your tables and databases every time your user needed a query with a new piece of information. The JOIN statement of SQL enables you to design smaller, more specific tables that are easier to maintain than larger tables.

Multiple Tables in a Single SELECT Statement

Like Dorothy in The Wizard of Oz, you have had the power to join tables since Day 2, "Introduction to the Query : The SELECT Statement," when you learned about SELECT and FROM. Unlike Dorothy, you do not have to click you heels together three times to perform a join. Use the following two tables, named, cleverly enough, TABLE1 and TABLE2.

INPUT :

```
SELECT *
FROM TABLE1
```

OUTPUT :

```
ROW         REMARKS
=======     =======
row 1       Table 1
row 2       Table 1
row 3       Table 1
row 4       Table 1
row 5       Table 1
row 6       Table 1
```

INPUT :

```
SELECT *
FROM TABLE2
```

OUTPUT :

```
ROW         REMARKS
=========   ========
row 1       table 2
row 2       table 2
row 3       table 2
row 4       table 2
row 5       table 2
row 6       table 2
```

To join these two tables, type this :

INPUT :

```
SELECT *
FROM TABLE1, TABLE2
```

OUTPUT :

ROW	REMARKS	ROW	REMARKS
=========	==========	========	==========
row 1	Table 1	row 1	table 2
row 1	Table 1	row 2	table 2
row 1	Table 1	row 3	table 2
row 1	Table 1	row 4	table 2
row 1	Table 1	row 5	table 2
row 1	Table 1	row 6	table 2
row 2	Table 1	row 1	table 2
row 2	Table 1	row 2	table 2
row 2	Table 1	row 3	table 2
row 2	Table 1	row 4	table 2
row 2	Table 1	row 5	table 2
row 2	Table 1	row 6	table 2
row 3	Table 1	row 1	table 2
row 3	Table 1	row 2	table 2
row 3	Table 1	row 3	table 2
row 3	Table 1	row 4	table 2
row 3	Table 1	row 5	table 2
row 3	Table 1	row 6	table 2
row 4	Table 1	row 1	table 2
row 4	Table 1	row 2	table 2
row 4	Table 1	row 3	table 2
row 4	Table 1	row 4	table 2
row 4	Table 1	row 5	table 2
row 4	Table 1	row 6	table 2
row 5	Table 1	row 1	table 2
row 5	Table 1	row 2	table 2
row 5	Table 1	row 3	table 2
row 5	Table 1	row 4	table 2
row 5	Table 1	row 5	table 2
row 5	Table 1	row 6	table 2
row 6	Table 1	row 1	table 2
row 6	Table 1	row 2	table 2
row 6	Table 1	row 3	table 2
row 6	Table 1	row 4	table 2
row 6	Table 1	row 5	table 2
row 6	Table 1	row 6	table 2

Thirty-six rows! Where did they come from ? And what kind of join is this ?

A close examination of the result of the first join shows that each row from TABLE1 was added to each row from TABLE2. An extract from this join shows what happened :

OUTPUT :

ROW	REMARKS	ROW	REMARKS
=====	==========	=======	========
row 1	Table 1	row 1	table 2
row 1	Table 1	row 2	table 2
row 1	Table 1	row 3	table 2
row 1	Table 1	row 4	table 2
row 1	Table 1	row 5	table 2
row 1	Table 1	row 6	table 2

Notice how each row in TABLE2 was combined with row 1 in TABLE1. Congratulations! You have performed your first join. But what kind of join? An inner join? an outer join? or what? Well, actually this type of join is called a cross-join. A cross-join is not normally as useful as the other joins covered today, but this join does illustrate the basic combining property of all joins : Joins bring tables together.

Suppose you sold parts to bike shops for a living. When you designed your database, you built one big table with all the pertinent columns. Every time you had a new requirement, you added a new column or started a new table with all the old data plus the new data required to create a specific query. Eventually, your database would collapse from its own weight-not a pretty sight. An alternative design, based on a relational model, would have you put all related data into one table. Here's how your customer table would look :

INPUT :

SELECT *

FROM CUSTOMER

OUTPUT :

NAME	ADDRESS	STATE	ZIP	PHONE	REMARKS
========	=========	======	====	=======	========
TRUE WHEEL	55O HUSKER	NE	58702	555-4545	NONE
BIKE SPEC	CPT SHRIVE	LA	45678	555-1234	NONE
LE SHOPPE	HOMETOWN	KS	54678	555-1278	NONE
AAA BIKE	10 OLDTOWN	NE	56784	555-3421	JOHN-MGR
JACKS BIKE	24 EGLIN	FL	34567	555-2314	NONE

Finding the Correct Column

When you joined TABLE1 and TABLE2, you used SELECT *, which returned all the columns in both tables. In joining ORDERS to PART, the SELECT statement is a bit more complicated :

SELECT O.ORDEREDON, O.NAME, O.PARTNUM,
P.PARTNUM, P.DESCRIPTION

SQL is smart enough to know that ORDEREDON and NAME exist only in ORDERS and that DESCRIPTION exists only in PART, but what about PARTNUM, which exists in both? If you have a column that has the same name in two tables, you must use an alias in your SELECT clause to specify which column you want to display. A common technique is to assign a single character to each table, as you did in the FROM clause :

FROM ORDERS O, PART P

You use that character with each column name, as you did in the preceding SELECT clause. The SELECT clause could also be written like this :

SELECT ORDEREDON, NAME, O.PARTNUM, P.PARTNUM, DESCRIPTION

But remember, someday you might have to come back and maintain this query. It does not hurt to make it more readable. Now back to the missing statement.

3.2.1 Equi-Joins

An extract from the PART/ORDERS join provides a clue as to what is missing :

30-JUN-1996 TRUE WHEEL 42 54 PEDALS
30-JUN-1996 BIKE SPEC 54 54 PEDALS
30-MAY-1996 BIKE SPEC 10 54 PEDALS

Notice the PARTNUM fields that are common to both tables. What if you wrote the following ?

INPUT :

SELECT O.ORDEREDON, O.NAME, O.PARTNUM,
P.PARTNUM, P.DESCRIPTION

FROM ORDERS O, PART P
WHERE O.PARTNUM = P.PARTNUM
OUTPUT :

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
===========	=========	========	=======	===========
1-JUN-1996	AAA BIKE	10	10	TANDEM
30-MAY-1996	BIKE SPEC	10	10	TANDEM
2-SEP-1996	TRUE WHEEL	10	10	TANDEM
1-JUN-1996	LE SHOPPE	10	10	TANDEM
30-MAY-1996	BIKE SPEC	23	23	MOUNTAIN BIKE
15-MAY-1996	TRUE WHEEL	23	23	MOUNTAIN BIKE
30-JUN-1996	TRUE WHEEL	42	42	SEATS
1-JUL-1996	AAA BIKE	46	46	TIRES
30-JUN-1996	BIKE SPEC	54	54	PEDALS
1-JUL-1996	AAA BIKE	76	76	ROAD BIKE
17-JAN-1996	BIKE SPEC	76	76	ROAD BIKE
19-MAY-1996	TRUE WHEEL	76	76	ROAD BIKE
11-JUL-1996	JACKS BIKE	76	76	ROAD BIKE
17-JAN-1996	LE SHOPPE	76	76	ROADBIKE

Using the column PARTNUM that exists in both of the preceding tables, you have just combined the information you had stored in the ORDERS table with information from the PART table to show a description of the parts the bike shops have ordered from you. The join that was used is called an equi-join because the goal is to match the values of a column in one table to the corresponding values in the second table.

You can further qualify this query by adding more conditions in the WHERE clause. For example:

INPUT/OUTPUT :
SELECT O.ORDEREDON, O.NAME, O.PARTNUM,
P.PARTNUM, P.DESCRIPTION
FROM ORDERS O, PART P
WHERE O.PARTNUM = P.PARTNUM
AND O.PARTNUM = 76

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
===========	=======	========	==================	
1-JUL-1996	AAA BIKE	76	76	ROAD BIKE
17-JAN-1996	BIKE SPEC	76	76	ROAD BIKE
19-MAY-1996	RUE WHEEL	76	76	ROAD BIKE
11-JUL-1996	JACKS BIKE	76	76	ROAD BIKE
17-JAN-1996	LE SHOPPE	76	76	ROAD BIKE

The number 76 is not very descriptive, and you would not want your sales people to have to memorize a part number. (We have had the misfortune to see many data information systems in the field that require the end user to know some obscure code for something that had a perfectly good name. Please don't write one of those!) Here's another way to write the query :

INPUT/OUTPUT :
SELECT O.ORDEREDON, O.NAME, O.PARTNUM,
P.PARTNUM, P.DESCRIPTION
FROM ORDERS O, PART P

```
WHERE O.PARTNUM = P.PARTNUM
AND P.DESCRIPTION = 'ROAD BIKE'
```

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
1-JUL-1996	AAA BIKE	76	76	ROAD BIKE
17-JAN-1996	BIKE SPEC	76	76	ROAD BIKE
19-MAY-1996	TRUE WHEEL	76	76	ROAD BIKE
11-JUL-1996	JACKS BIKE	76	76	ROAD BIKE
17-JAN-1996	LE SHOPPE	76	76	ROAD BIKE

Along the same line, take a look at two more tables to see how they can be joined. In this example the employee_id column should obviously be unique. You could have employees with the same name, they could work in the same department, and earn the same salary. However, each employee would have his or her own employee_id. To join these two tables, you would use the employee_id column.

EMPLOYEE_TABLE	EMPLOYEE_PAY_TABLE
employee_id	employee_id
last_name	salary
first_name	department
middle_name	supervisor
	marital_status

INPUT :

```
SELECT   E.EMPLOYEE_ID, E.LAST_NAME, EP.SALARY
FROM  EMPLOYEE_TBL E,
      EMPLOYEE_PAY_TBL EP
WHERE    E.EMPLOYEE_ID = EP.EMPLOYEE_ID
 AND     E.LAST_NAME = 'SMITH';
```

OUTPUT :

E.EMPLOYEE_ID	E.LAST_NAME	EP.SALARY
13245	SMITH	35000.00

Back to the original tables. Now you are ready to use all this information about joins to do something really useful: finding out how much money you have made from selling road bikes :

INPUT/OUTPUT :

```
SELECT SUM(O.QUANTITY * P.PRICE) TOTAL
FROM ORDERS O, PART P
WHERE O.PARTNUM = P.PARTNUM
AND P.DESCRIPTION = 'ROAD BIKE'

   TOTAL
===========
  19610.00
```

With this setup, the sales people can keep the ORDERS table updated, the production department can keep the PART table current, and you can find your bottom line without redesigning your database.

Can you join more than one table? For example, to generate information to send out an invoice, you could type this statement:

INPUT/OUTPUT :

```
SELECT C.NAME, C.ADDRESS, (O.QUANTITY * P.PRICE) TOTAL
FROM ORDER O, PART P, CUSTOMER C
WHERE O.PARTNUM = P.PARTNUM
AND O.NAME = C.NAME
```

NAME	ADDRESS	TOTAL
==========	============	=========
TRUE WHEEL	55O HUSKER	1200.00
BIKE SPEC	CPT SHRIVE	2400.00
LE SHOPPE	HOMETOWN	3600.00
AAA BIKE	10 OLDTOWN	1200.00
TRUE WHEEL	55O HUSKER	2102.70
BIKE SPEC	CPT SHRIVE	2803.60
TRUE WHEEL	55O HUSKER	196.00
AAA BIKE	10 OLDTOWN	213.50
BIKE SPEC	CPT SHRIVE	542.50
TRUE WHEEL	55O HUSKER	1590.00
BIKE SPEC	CPT SHRIVE	5830.00
JACKS BIKE	24 EGLIN	7420.00
LE SHOPPE	HOMETOWN	2650.00
AAA BIKE	10 OLDTOWN	2120.00

You could make the output more readable by writing the statement like this :

INPUT/OUTPUT :

```
SELECT C.NAME, C.ADDRESS,
O.QUANTITY * P.PRICE TOTAL
FROM ORDERS O, PART P, CUSTOMER C
WHERE O.PARTNUM = P.PARTNUM
AND O.NAME = C.NAME
ORDER BY C.NAME
```

NAME	ADDRESS	TOTAL
==========	==========	===========
AAA BIKE	10 OLDTOWN	213.50
AAA BIKE	10 OLDTOWN	2120.00
AAA BIKE	10 OLDTOWN	1200.00
BIKE SPEC	CPT SHRIVE	542.50
BIKE SPEC	CPT SHRIVE	2803.60
BIKE SPEC	CPT SHRIVE	5830.00
BIKE SPEC	CPT SHRIVE	2400.00
JACKS BIKE	24 EGLIN	7420.00
LE SHOPPE	HOMETOWN	2650.00
LE SHOPPE	HOMETOWN	3600.00
TRUE WHEEL	55O HUSKER	196.00
TRUE WHEEL	55O HUSKER	2102.70
TRUE WHEEL	55O HUSKER	1590.00
TRUE WHEEL	55O HUSKER	1200.00

You can make the previous query more specific, thus more useful, by adding the DESCRIPTION column as in the following example :

INPUT/OUTPUT :

```
SELECT C.NAME, C.ADDRESS,
O.QUANTITY * P.PRICE TOTAL,
P.DESCRIPTION
FROM ORDERS O, PART P, CUSTOMER C
WHERE O.PARTNUM = P.PARTNUM
AND O.NAME = C.NAME
ORDER BY C.NAME
```

NAME	ADDRESS	TOTAL	DESCRIPTION
==========	==========	=========	==============
AAA BIKE	10 OLDTOWN	213.50	TIRES
AAA BIKE	10 OLDTOWN	2120.00	ROAD BIKE
AAA BIKE	10 OLDTOWN	1200.00	TANDEM
BIKE SPEC	CPT SHRIVE	542.50	PEDALS
BIKE SPEC	CPT SHRIVE	2803.60	MOUNTAIN BIKE
BIKE SPEC	CPT SHRIVE	5830.00	ROAD BIKE
BIKE SPEC	CPT SHRIVE	2400.00	TANDEM
JACKS BIKE	24 EGLIN	7420.00	ROAD BIKE
LE SHOPPE	HOMETOWN	2650.00	ROAD BIKE
LE SHOPPE	HOMETOWN	3600.00	TANDEM
TRUE WHEEL	55O HUSKER	196.00	SEATS
TRUE WHEEL	55O HUSKER	2102.70	MOUNTAIN BIKE
TRUE WHEEL	55O HUSKER	1590.00	ROAD BIKE
TRUE WHEEL	55O HUSKER	1200.00	TANDEM

This information is a result of joining three tables. You can now use this information to create an invoice.

3.2.2 Non-Equi-Joins

Because SQL supports an equi-join, you might assume that SQL also has a non-equi-join. You would be right! Whereas the equi-join uses an = sign in the WHERE statement, the non-equi-join uses everything but an = sign. For example :

INPUT :

```
SELECT O.NAME, O.PARTNUM, P.PARTNUM,
O.QUANTITY * P.PRICE TOTAL
FROM ORDERS O, PART P
WHERE O.PARTNUM > P.PARTNUM
```

OUTPUT :

NAME	PARTNUM	PARTNUM	TOTAL
TRUE WHEEL	76	54	162.75
BIKE SPEC	76	54	596.75
LE SHOPPE	76	54	271.25
AAA BIKE	76	54	217.00
JACKS BIKE	76	54	759.50
TRUE WHEEL	76	42	73.50
BIKE SPEC	54	42	245.00
BIKE SPEC	76	42	269.50
LE SHOPPE	76	42	122.50
AAA BIKE	76	42	98.00
AAA BIKE	46	42	343.00
JACKS BIKE	76	42	343.00
TRUE WHEEL	76	46	45.75
BIKE SPEC	54	46	152.50
BIKE SPEC	76	46	167.75
LE SHOPPE	76	46	76.25
AAA BIKE	76	46	61.00
JACKS BIKE	76	46	213.50
TRUE WHEEL	76	23	1051.35
TRUE WHEEL	42	23	2803.60

...

This listing goes on to describe all the rows in the join WHERE O.PARTNUM > P.PARTNUM. In the context of your bicycle shop, this information does not have much meaning, and in the real world the equi-join is far more common than the non-equi-join. However, you may encounter an application in which a non-equi-join produces the perfect result.

3.2.3 Outer Joins versus Inner Joins

Just as the non-equi-join balances the equi-join, an outer join complements the inner join. An inner join is where the rows of the tables are combined with each other, producing a number of new rows equal to the product of the number of rows in each table. Also, the inner join uses these rows to determine the result of the WHERE clause. An outer join groups the two tables in a slightly different way. Using the PART and ORDERS tables from the previous examples, perform the following inner join:

INPUT :

```
SELECT P.PARTNUM, P.DESCRIPTION, P.PRICE,
O.NAME, O.PARTNUM
FROM PART P
JOIN ORDERS O ON ORDERS.PARTNUM = 54
```

OUTPUT :

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
54	PEDALS	54.25	BIKE SPEC	54
42	SEATS	24.50	BIKE SPEC	54
46	TIRES	15.25	BIKE SPEC	54
23	MOUNTAIN BIKE	350.45	BIKE SPEC	54
76	ROAD BIKE	530.00	BIKE SPEC	54
10	TANDEM	1200.00	BIKE SPEC	54

The result is that all the rows in PART are spliced on to specific rows in ORDERS where the column PARTNUM is 54. Here's a RIGHT OUTER JOIN statement :

INPUT/OUTPUT :

```
SELECT P.PARTNUM, P.DESCRIPTION, P.PRICE,
```

O.NAME, O.PARTNUM
FROM PART P
RIGHT OUTER JOIN ORDERS O ON ORDERS.PARTNUM = 54

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
<null>	<null>	<null>	TRUE WHEEL	23
<null>	<null>	<null>	TRUE WHEEL	76
<null>	<null>	<null>	TRUE WHEEL	10
<null>	<null>	<null>	TRUE WHEEL	42
54	PEDALS	54.25	BIKE SPEC	54
42	SEATS	24.50	BIKE SPEC	54
46	TIRES	15.25	BIKE SPEC	54
23	MOUNTAIN BIKE	350.45	BIKE SPEC	54
76	ROAD BIKE	530.00	BIKE SPEC	54
10	TANDEM	1200.00	BIKE SPEC	54
<null>	<null>	<null>	BIKE SPEC	10
<null>	<null>	<null>	BIKE SPEC	23
<null>	<null>	<null>	BIKE SPEC	76
<null>	<null>	<null>	LESHOPPE	76
<null>	<null>	<null>	LE SHOPPE	10
<null>	<null>	<null>	AAA BIKE	10
<null>	<null>	<null>	AAA BIKE	76
<null>	<null>	<null>	AAA BIKE	46
<null>	<null>	<null>	JACKS BIKE	76

This type of query is new. First you specified a RIGHT OUTER JOIN, which caused SQL to return a full set of the right table, ORDERS, and to place nulls in the fields where ORDERS.PARTNUM <> 54. Following is a LEFT OUTER JOIN statement :

INPUT/OUTPUT :
SELECT P.PARTNUM, P.DESCRIPTION,P.PRICE,
O.NAME, O.PARTNUM
FROM PART P
LEFT OUTER JOIN ORDERS O ON ORDERS.PARTNUM = 54

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
54	PEDALS	54.25	BIKE SPEC	54
42	SEATS	24.50	BIKE SPEC	54
46	TIRES	15.25	BIKE SPEC	54
23	MOUNTAIN BIKE	350.45	BIKE SPEC	54
76	ROAD BIKE	530.00	BIKE SPEC	54
10	TANDEM	1200.00	BIKE SPEC	54

You get the same six rows as the INNER JOIN. Because you specified LEFT (the LEFT table), PART determined the number of rows you would return. Because PART is smaller than ORDERS, SQL saw no need to pad those other fields with blanks.

Don't worry too much about inner and outer joins. Most SQL products determine the optimum JOIN for your query. In fact, if you are placing your query into a stored procedure (or using it inside a program (both stored procedures and Embedded SQL covered on Day 13, "Advanced SQL Topics"), you should not specify a join type even if your SQL implementation provides the proper syntax. If you do specify a join type, the optimizer chooses your way instead of the optimum way.

Some implementations of SQL use the + sign instead of an OUTER JOIN statement. The + simply means "Show me everything even if something is missing". Here's the syntax :

SYNTAX :

```
SQL> select e.name, e.employee_id, ep.salary,
         ep.marital_status
     from e,ployee_tbl e,
        employee_pay_tbl ep
     where e.employee_id = ep.employee_id(+)
      and e.name like '%MITH';
```

This statement is joining the two tables. The + sign on the ep.employee_id column will return all rows even if they are empty.

3.2.4 Joining a Table to Itself

The syntax of this operation is similar to joining two tables. For example, to join table TABLE1 to itself, type this :

INPUT :

SELECT *

FROM TABLE1, TABLE1

OUTPUT :

```
 ROW        REMARKS     ROW      REMARKS
========= ========== ======== ==========
  row 1      Table 1     row 1     Table 1
  row 1      Table 1     row 2     Table 1
  row 1      Table 1     row 3     Table 1
  row 1      Table 1     row 4     Table 1
  row 1      Table 1     row 5     Table 1
  row 1      Table 1     row 6     Table 1
  row 2      Table 1     row 1     Table 1
  row 2      Table 1     row 2     Table 1
  row 2      Table 1     row 3     Table 1
  row 2      Table 1     row 4     Table 1
  row 2      Table 1     row 5     Table 1
  row 2      Table 1     row 6     Table 1
  row 3      Table 1     row 1     Table 1
  row 3      Table 1     row 2     Table 1
  row 3      Table 1     row 3     Table 1
  row 3      Table 1     row 4     Table 1
  row 3      Table 1     row 5     Table 1
  row 3      Table 1     row 6     Table 1
  row 4      Table 1     row 1     Table 1
  row 4      Table 1     row 2     Table 1
...
```

In its complete form, this join produces the same number of combinations as joining two 6-row tables. This type of join could be useful to check the internal consistency of data. What would happen if someone fell asleep in the production department and entered a new part with a PARTNUM that already existed? That would be bad news for everybody: Invoices would be wrong; your application would probably blow up; and in general you would be in for a very bad time. And the cause of all your problems would be the duplicate PARTNUM in the table on the next page :

INPUT/OUTPUT :
SELECT * FROM PART

PARTNUM	DESCRIPTION	PRICE
54	PEDALS	54.25
42	SEATS	24.50
46	TIRES	15.25
23	MOUNTAIN BIKE	350.45
76	ROAD BIKE	530.00
10	TANDEM	1200.00
76	CLIPPLESS SHOE	65.00

<-NOTE SAME #

You saved your company from this bad situation by checking PART before anyone used it :

INPUT/OUTPUT :

```
SELECT F.PARTNUM, F.DESCRIPTION,
S.PARTNUM,S.DESCRIPTION
FROM PART F, PART S
WHERE F.PARTNUM = S.PARTNUM
AND F.DESCRIPTION <> S.DESCRIPTION
```

PARTNUM	DESCRIPTION	PARTNUM	DESCRIPTION
76	ROAD BIKE	76	CLIPPLESS SHOE
76	CLIPPLESS SHOE	76	ROAD BIKE

3.1,3.2 Check Your Progress

Fill in the blanks

1) ………………… can be used for joining of two tables.
2) A………………… must return a value.
3) A Procedure have…………………

3.3 PROCEDURE AND FUNCTIONS

Procedures are simply a named PL/SQL block, that executes certain task. A procedure is completely portable among platforms in which Oracle is executed.

A function is similar to a procedure. The main difference between the function and procedure is that a function returns a value where procedure does not.

3.3.1 Advantages of using Procedures and Functions

1. Improved performance :

- A block is placed on the database it is parsed at the time it is stored. When it is subsequently executed Oracle already has the block compiled and it is therefore much faster.
- Reduce the number of calls to the database and decrease network traffic by bundling commands.

2. Improved maintenance :

- Modify routines online without interfering with other users.
- Modify one routine to affect multiple applications.
- Modify one routine to eliminate duplicate testing.

3. **Improved data security and integrity :**
   - Control indirect access to objects from non privileged users.
   - Ensure that related actions are performed together or not at all, by funnelling actions for related tables through a single path.

3.4 CREATING A PROCEDURE

A procedure is created using CREATE PROCEDURE command.

Syntax :

```
CREATE [OR REPLACE] PROCEDURE  procedure_name
[(argument [in/out/in out] datatype [,argument [in/out/in out] datatype…..])]
{IS / AS}
[variable declaration]
{PL/SQL block};
```

Note : Square brakets [ ] indicate optional part.

CREATE PROCEDURE procedure_name will create a new procedure with the given procedure_name. OR REPLACE is an optional clause. It is used to change the definition of an existing procedure.

If the procedure accept arguments specify argument details as,

Argument_name IN / OUT / IN OUT datatype

Argument_name indicate Variable_name

IN indicate the variable is passed by the calling program to procedure.

OUT indicate that the variable pass value from procedure to calling program.

IN OUT indicate that the variable can pass values both in and out of a procedure.

Datatype specify any PL/SQL datatype.

3.5 EXECUTING A PROCEDURE

To execute the stored procedure simply call it by name in EXECUTE command as

SQL> execute myproc1(7768);

This will execute myproc1 with the value 7768.

The second method of calling the procedure is

Write the following code in an editor.

```
Declare
    C_empno number;
Begin
    Myproc1(&c_empno);
End;
/
```

Execute it as

SQL >/

In this case the value of variable c_empno is accepted from user and then it is pass to myproc1 procedure.

To see the effect of this procedure use command,

SQL>select * from emp

3.6 DELETING A PROCEDURE

To delete a procedure DROP PROCEDURE command is used.

Syntax :

DROP PROCEDURE procedure_name;

For example,

```
DROP PROCEDURE myproc1;
```

3.7 FUNCTIONS

Functions in SQL enable you to perform feats such as determining the sum of a column or converting all the characters of a string to uppercase. By the end of the day, you will understand and be able to use all the following :

(a) Aggregate functions
(b) Date and time functions
(c) Arithmetic functions
(d) Character functions
(e) Conversion functions
(f) Miscellaneous functions.

These functions greatly increase your ability to manipulate the information you retrieved using the basic functions of SQL that were described earlier this week. The first five aggregate functions, COUNT, SUM, AVG, MAX, and MIN, are defined in the ANSI standard. Most implementations of SQL have extensions to these aggregate functions, some of which are covered today. Some implementations may use different names for these functions.

3.7.1 Aggregate Functions

These functions are also referred to as group functions. They return a value based on the values in a column. (After all, you would not ask for the average of a single field.) The examples in this section use the table TEAMSTATS :

INPUT :

```
SQL> SELECT * FROM TEAMSTATS;
```

OUTPUT :

NAME	POS	AB	HITS	WALKS	SINGLES	DOUBLES	TRIPLES	HR	SO
JONES	1B	145	45	34	31	8	1	5	10
DONKNOW	3B	175	65	23	50	10	1	4	15
WORLEY	LF	157	49	15	35	8	3	3	16
DAVID	OF	187	70	24	48	4	0	17	42
HAMHOCKER	3B	50	12	10	10	2	0	0	13
CASEY	DH	1	0	0	0	0	0	0	1

6 rows selected.

- **COUNT**

The function COUNT returns the number of rows that satisfy the condition in the WHERE clause. Say you wanted to know how many ball players were hitting under 350. You would type,

INPUT/OUTPUT :

```
SQL> SELECT COUNT(*)
  2  FROM TEAMSTATS
  3  WHERE HITS/AB < .35;
COUNT(*)
--------
     4
```

- **SUM**

SUM does just that. It returns the sum of all values in a column. To find out how many singles have been hit, type,

INPUT :

```
SQL> SELECT SUM(SINGLES) TOTAL_SINGLES
  2  FROM TEAMSTATS;
```

OUTPUT :

```
TOTAL_SINGLES
----------------------------
        174
```

To get several sums, use

INPUT/OUTPUT :

```
SQL> SELECT SUM(SINGLES) TOTAL_SINGLES, SUM(DOUBLES) TOTAL_DOUBLES,
     SUM(TRIPLES) TOTAL_TRIPLES, SUM(HR) TOTAL_HR
  2  FROM TEAMSTATS;

TOTAL_SINGLES TOTAL_DOUBLES   TOTAL_TRIPLES     TOTAL_HR
-----------------------    --------------------    ------------------------   --------------------
   174                  32                    5                 29
```

To collect similar information on all 300 or better players, type

INPUT/OUTPUT :

```
SQL> SELECT SUM(SINGLES) TOTAL_SINGLES, SUM(DOUBLES) TOTAL_DOUBLES,
     SUM(TRIPLES) TOTAL_TRIPLES, SUM(HR) TOTAL_HR
  2  FROM TEAMSTATS
  3  WHERE HITS/AB >= .300;

TOTAL_SINGLES TOTAL_DOUBLES  TOTAL_TRIPLES   TOTAL_HR
--------------------     --------------------    ---------------------   --------------
     164              30               5            29
```

- **AVG**

The AVG function computes the average of a column. To find the average number of strike outs, use this:

INPUT :

```
SQL> SELECT AVG(SO) AVE_STRIKE_OUTS
  2  FROM TEAMSTATS;
```

OUTPUT :

```
AVE_STRIKE_OUTS
------------------------------
   16.166667
```

The following example illustrates the difference between SUM and AVG :

INPUT/OUTPUT :

```
SQL> SELECT AVG(HITS/AB) TEAM_AVERAGE
  2  FROM TEAMSTATS;
TEAM_AVERAGE
-----------------------
  .26803448
```

- **MAX**

If you want to find the largest value in a column, use MAX. For example, what is the highest number of hits ?

INPUT :

```
SQL> SELECT MAX(HITS)
```

```
  2    FROM TEAMSTATS;
```

OUTPUT :

```
MAX (HITS)
-----------------
70
```

Can you find out who has the most hits?

INPUT/OUTPUT :

```
SQL> SELECT NAME
  2    FROM TEAMSTATS
  3    WHERE HITS = MAX(HITS);
ERROR at line 3 :
ORA-00934: group function is not allowed here.
```

Unfortunately, you can't. The error message is a reminder that this group function (remember that *aggregate functions* are also called *group functions*) does not work in the WHERE clause. Don't despair, Day 7, "Subqueries : The Embedded SELECT Statement" covers the concept of subqueries and explains a way to find who has the MAX hits.

MIN

MIN does the expected thing and works like MAX except it returns the lowest member of a column. To find out the fewest at bats, type

INPUT :

```
SQL> SELECT MIN(AB)
  2    FROM TEAMSTATS;
```

OUTPUT :

```
MIN (AB)
---------
        1
```

The following statement returns the name closest to the beginning of the alphabet :

INPUT/OUTPUT :

```
SQL> SELECT MIN(NAME)
  2  FROM TEAMSTATS;

MIN (NAME)
---------------
CASEY
```

You can combine MIN with MAX to give a range of values. For example :

INPUT/OUTPUT :

```
SQL> SELECT MIN (AB), MAX (AB)
  2    FROM TEAMSTATS;
       MIN(AB)  MAX(AB)
       --------  --------
           1      187
```

This sort of information can be useful when using statistical functions.

- **VARIANCE**

VARIANCE produces the square of the standard deviation, a number vital to many statistical calculations. It works like this :

INPUT :

```
SQL> SELECT VARIANCE(HITS)
  2    FROM TEAMSTATS;
```

OUTPUT :

```
VARIANCE(HITS)
--------------
    802.96667
```

Example for string,

INPUT/OUTPUT :

```
SQL> SELECT VARIANCE(NAME)
  2   FROM TEAMSTATS;
ERROR :
ORA-01722: invalid number
no rows selected,
```

you find that VARIANCE is another function that works exclusively with numbers.

- **STDDEV**

The final group function, STDDEV, finds the standard deviation of a column of numbers, as demonstrated by this example :

INPUT :

```
SQL> SELECT STDDEV(HITS)
  2   FROM TEAMSTATS;
```

OUTPUT :

```
STDDEV(HITS)
---------------------
   28.336666
```

It also returns an error when confronted by a string :

INPUT/OUTPUT :

```
SQL> SELECT STDDEV(NAME)
  2   FROM TEAMSTATS;
ERROR :
ORA-01722: invalid number
no rows selected
```

These aggregate functions can also be used in various combinations :

INPUT/OUTPUT :

```
SQL> SELECT COUNT(AB),
  2   AVG(AB),
  3   MIN(AB),
  4   MAX(AB),
  5   STDDEV(AB),
  6   VARIANCE(AB),
  7   SUM(AB)
  8   FROM TEAMSTATS;

COUNT(AB)  AVG(AB)  MIN(AB)  MAX(AB)  STDDEV(AB)   VARIANCE(AB)  SUM(AB)
---------------- ------------- ----------  ------------- -----------------   --------------------- ----------
       6        119.167      1          187        75.589           5712.97       715
```

The next time you hear a sportscaster use statistics to fill the time between plays, you will know that SQL is at work somewhere behind the scenes.

3.7.2 Date and Time Functions

We live in a civilization governed by times and dates and most major implementations of SQL have functions to cope with these concepts. This section uses the table PROJECT to demonstrate the time and date functions.

INPUT :

SQL> **SELECT * FROM PROJECT;**

OUTPUT :

```
TASK                STARTDATE       ENDDATE
--------------      -------------------  -----------------
KICKOFF MTG         01-APR-95       01-APR-95
TECH SURVEY         02-APR-95       01-MAY-95
USER MTGS           15-MAY-95       30-MAY-95
DESIGN WIDGET       01-JUN-95       30-JUN-95
CODE WIDGET         01-JUL-95       02-SEP-95
TESTING             03-SEP-95       17-JAN-96
6 rows selected.
```

- **NEW_TIME**

If you need to adjust the time according to the time zone you are in, the New_TIME function is for you. Here are the time zones you can use with this function :

Abbreviation	Time Zone
AST or ADT	Atlantic standard or daylight time
BST or BDT	Bering standard or daylight time
CST or CDT	Central standard or daylight time
EST or EDT	Eastern standard or daylight time
GMT	Greenwich mean time
HST or HDT	Alaska-Hawaii standard or daylight time
MST or MDT	Mountain standard or daylight time
NST	Newfoundland standard time
PST or PDT	Pacific standard or daylight time
YST or YDT	Yukon standard or daylight time

You can adjust your time like this :

INPUT :

SQL> **SELECT ENDDATE EDT,**
  2   **NEW_TIME(ENDDATE, 'EDT','PDT')**
  3   **FROM PROJECT;**

OUTPUT :

```
        EDT                  NEW_TIME(ENDDATE
-----------------------------  -------------------------------------
01-APR-95 1200AM       31-MAR-95 0900PM
01-MAY-95 1200AM       30-APR-95 0900PM
30-MAY-95 1200AM       29-MAY-95 0900PM
30-JUN-95 1200AM       29-JUN-95 0900PM
02-SEP-95 1200AM       01-SEP-95 0900PM
17-JAN-96 1200AM       16-JAN-96 0900PM
6 rows selected.
```

Like magic, all the times are in the new time zone and the dates are adjusted.

- **NEXT_DAY**

  NEXT_DAY finds the name of the first day of the week that is equal to or later than another specified date. For example, to send a report on the Friday following the first day of each event, you would type,

INPUT :

```
SQL> SELECT STARTDATE,
  2     NEXT_DAY(STARTDATE, 'FRIDAY')
  3     FROM PROJECT;
```

Which would return,

OUTPUT :

```
STARTDATE      NEXT_DAY(
-------------------  -------------------
01-APR-95      07-APR-95
02-APR-95      07-APR-95
15-MAY-95      19-MAY-95
01-JUN-95      02-JUN-95
01-JUL-95      07-JUL-95
03-SEP-95      08-SEP-95
6 rows selected.
```

The output tells you the date of the first Friday that occurs after your STARTDATE.

- **SYSDATE**

  SYSDATE returns the system time and date :

INPUT :

```
SQL> SELECT DISTINCT SYSDATE
  2     FROM PROJECT;
```

OUTPUT :

```
SYSDATE
----------------
18-JUN-95        1020PM
```

If you wanted to see where you stood today in a certain project, you could type

INPUT/OUTPUT :

```
SQL> SELECT *
  2     FROM PROJECT
  3     WHERE STARTDATE > SYSDATE;
TASK                 STARTDATE      ENDDATE
--------------       --------------------  -----------------
CODE WIDGET          01-JUL-95      02-SEP-95
TESTING              03-SEP-95      17-JAN-96
```

Now you can see what parts of the project start after today.

2.7.3 Arithmetic Functions

Many of the uses you have for the data you retrieve involve mathematics. Most implementations of SQL provide arithmetic functions similar to the functions covered here. The examples in this section use the NUMBERS table :

INPUT :

```
SQL> SELECT *
  2     FROM NUMBERS;
```

OUTPUT :

A	B
3.1415	4
–45	.707
5	9
–57.667	42
15	55
–7.2	5.3

6 rows selected.

- **ABS**

The ABS function returns the absolute value of the number you point to. For example :

INPUT :

SQL> **SELECT ABS(A) ABSOLUTE_VALUE**
 2 **FROM NUMBERS;**

OUTPUT :

ABSOLUTE_VALUE
3.1415
45
5
57.667
15
7.2

6 rows selected.

ABS changes all the negative numbers to positive and leaves positive numbers alone.

- **CEIL and FLOOR**

CEIL returns the smallest integer greater than or equal to its argument. FLOOR does just the reverse, returning the largest integer equal to or less than its argument. For example :

INPUT :

SQL> **SELECT B, CEIL(B) CEILING**
 2 **FROM NUMBERS;**

OUTPUT :

B	CEILING
4	4
.707	1
9	9
42	42
55	55
5.3	6

6 rows selected.

And

INPUT/OUTPUT :

SQL> **SELECT A, FLOOR(A) FLOOR**
 2 **FROM NUMBERS;**

A	FLOOR
3.1415	3
–45	–45
5	5
–57.667	–58
15	15
–7.2	–8

6 rows selected.

- **COS, COSH, SIN, SINH, TAN and TANH**

The COS, SIN, and TAN functions provide support for various trigonometric concepts. They all work on the assumption that n is in radians. The following statement returns some unexpected values if you don't realize COS expects A to be in radians.

INPUT :

SQL> **SELECT A, COS(A)**
2 **FROM NUMBERS;**

OUTPUT :

A	COS(A)
3.1415	–1
–45	.52532199
5	.28366219
–57.667	.437183
15	–.7596879
–7.2	.60835131

- **EXP**

EXP enables you to raise *e* (*e* is a mathematical constant used in various formulas) to a power. Here is how EXP raises *e* by the values in column A :

INPUT :

SQL> **SELECT A, EXP(A)**
2 **FROM NUMBERS;**

OUTPUT :

A	EXP(A)
3.1415	23.138549
–45	2.863E-20
5	148.41316
–57.667	9.027E-26
15	3269017.4
–7.2	.00074659

6 rows selected.

- **LN and LOG**

These two functions center on logarithms. LN returns the natural logarithm of its argument. For example :

INPUT :

```
SQL> SELECT A, LN(A)
  2  FROM NUMBERS;
```

OUTPUT :

ERROR :

ORA–01428: argument '–45' is out of range

Did we neglect to mention that the argument had to be positive? Write

INPUT/OUTPUT :

```
SQL> SELECT A, LN(ABS(A))
  2  FROM NUMBERS;
```

A	LN(ABS(A))
3.1415	1.1447004
–45	3.8066625
5	1.6094379
–57.667	4.0546851
15	2.7080502
–7.2	1.974081

6 rows selected.

- **MOD**

You have encountered MOD before. On Day 3, "Expressions, Conditions, and Operators, you saw that the ANSI standard for the modulo operator % is sometimes implemented as the function MOD. Here is a query that returns a table showing the remainder of A divided by B :

INPUT :

```
SQL> SELECT A, B, MOD(A,B)
  2  FROM NUMBERS;
```

OUTPUT :

A	B	MOD(A,B)
3.1415	4	3.1415
–45	.707	–.459
5	9	5
–57.667	42	–15.667
15	55	15
–7.2	5.3	–1.9

6 rows selected.

- **POWER**

To raise one number to the power of another, use POWER. In this function the first argument is raised to the power of the second :

INPUT :

```
SQL> SELECT A, B, POWER(A, B)
  2  FROM NUMBERS;
```

OUTPUT :

ERROR :

ORA-01428: argument '-45' is out of range

- **SIGN :**

SIGN returns -1 if its argument is less than 0, 0 if its argument is equal to 0 and 1 if its argument is greater than 0, as shown in the following example :

INPUT :

```
SQL> SELECT A, SIGN(A)
  2   FROM NUMBERS;
```

OUTPUT :

```
       A          SIGN(A)
 ------------   --------------
   3.1415            1
      –45           –1
        5            1
  –57.667           –1
       15            1
     –7.2           –1
        0            0
7 rows selected.
```

- **SQRT:**

The function SQRT returns the square root of an argument. Because the square root of a negative number is undefined, you cannot use SQRT on negative numbers.

INPUT/OUTPUT :

```
SQL> SELECT A, SQRT(A)
  2   FROM NUMBERS;
 ERROR :
ORA-01428: argument '-45' is out of range
```

3.7.4 Character Functions

Many implementations of SQL provide functions to manipulate characters and strings of characters. This section covers the most common character functions. The examples in this section use the table CHARACTERS.

INPUT/OUTPUT :

```
SQL> SELECT * FROM CHARACTERS;

LASTNAME           FIRSTNAME              M     CODE
-------------------  ----------------------  ---   ---------
PURVIS             KELLY                  A       32
TAYLOR             CHUCK                  J       67
CHRISTINE          LAURA                  C       65
ADAMS              FESTER                 M       87
COSTALES           ARMANDO                A       77
KONG               MAJOR                  G       52
6 rows selected.
```

- **CHR**

CHR returns the character equivalent of the number it uses as an argument. The character it returns depends on the character set of the database. For this example the database is set to ASCII. The column CODE includes numbers.

INPUT :

```
SQL> SELECT CODE, CHR(CODE)
  2  FROM CHARACTERS;
```

OUTPUT :

```
CODE CH
--------- --
  32
  67 C
  65 A
  87 W
  77 M
  52 4
6 rows selected.
```

The space opposite the 32 shows that 32 is a space in the ASCII character set.

- **CONCAT**

You used the equivalent of this function on Day 3, when you learned about operators. The || symbol splices two strings together, as does CONCAT. It works like this :

INPUT :

```
SQL> SELECT CONCAT(FIRSTNAME, LASTNAME) "FIRST AND LAST NAMES"
  2  FROM CHARACTERS;
```

OUTPUT :

```
FIRST  AND  LAST NAMES
-------------------------------------------
KELLY          PURVIS
CHUCK          TAYLOR
LAURA          CHRISTINE
FESTER         ADAMS
ARMANDO        COSTALES
MAJOR          KONG
6 rows selected.
```

- **INITCAP**

INITCAP capitalizes the first letter of a word and makes all other characters lowercase.

INPUT :

```
SQL> SELECT FIRSTNAME BEFORE, INITCAP(FIRSTNAME) AFTER
  2  FROM CHARACTERS;
```

OUTPUT :

```
BEFORE          AFTER
--------------  ----------
KELLY           Kelly
CHUCK           Chuck
LAURA           Laura
FESTER          Fester
ARMANDO         Armando
MAJOR           Major
6 rows selected.
```

- **LOWER and UPPER**

As you might expect, LOWER changes all the characters to lowercase; UPPER does just the reverse.

The following example starts by doing a little magic with the UPDATE function (you learn more about this next week) to change one of the values to lowercase :

INPUT :

```
SQL> UPDATE CHARACTERS
  2    SET FIRSTNAME = 'kelly'
  3    WHERE FIRSTNAME = 'KELLY';
```

OUTPUT :

```
1 row updated.
```

INPUT :

```
SQL> SELECT FIRSTNAME
  2    FROM CHARACTERS;
```

OUTPUT :

```
FIRSTNAME
--------------------
kelly
CHUCK
LAURA
FESTER
ARMANDO
MAJOR
6 rows selected.
```

Then you write :

INPUT :

```
SQL> SELECT FIRSTNAME, UPPER(FIRSTNAME), LOWER(FIRSTNAME)
  2    FROM CHARACTERS;
```

OUTPUT :

```
FIRSTNAME   UPPER(FIRSTNAME)        LOWER(FIRSTNAME)
---------------  -----------------------------  ---------------
kelly          KELLY                kelly
CHUCK          CHUCK                chuck
LAURA          LAURA                laura
FESTER         FESTER               fester
ARMANDO        ARMANDO              armando
MAJOR          MAJOR                major
```

6 rows selected.

Now you see the desired behavior.

- **LPAD and RPAD**

LPAD and RPAD take a minimum of two and a maximum of three arguments. The first argument is the character string to be operated on. The second is the number of characters to pad it with, and the optional third argument is the character to pad it with. The third argument defaults to a blank, or it can be a single character or a character string. The following statement adds five pad characters, assuming that the field LASTNAME is defined as a 15-character field :

INPUT :

SQL> **SELECT LASTNAME, LPAD(LASTNAME,20,'*')**
  2 **FROM CHARACTERS;**

OUTPUT :

```
LASTNAME        LPAD(LASTNAME,20,'*'
------------------  ------------------------------------
PURVIS          *****PURVIS
TAYLOR          *****TAYLOR
CHRISTINE       *****CHRISTINE
ADAMS           *****ADAMS
COSTALES        *****COSTALES
KONG            *****KONG
```

6 rows selected.

- **LTRIM and RTRIM**

LTRIM and RTRIM take at least one and at most two arguments. The first argument, like LPAD and RPAD, is a character string. The optional second element is either a character or character string or defaults to a blank. If you use a second argument that is not a blank, these trim functions will trim that character the same way they trim the blanks in the following examples.

INPUT :

SQL> **SELECT LASTNAME, RTRIM(LASTNAME)**
  2 **FROM CHARACTERS;**

OUTPUT :

```
LASTNAME       RTRIM(LASTNAME)
---------------   ---------------
PURVIS          PURVIS
TAYLOR          TAYLOR
CHRISTINE       CHRISTINE
ADAMS           ADAMS
COSTALES        COSTALES
KONG            KONG
```

6 rows selected.

You can make sure that the characters have been trimmed with the following statement :

INPUT :

SQL> **SELECT LASTNAME, RPAD(RTRIM(LASTNAME),20,'*')**

```
  2   FROM CHARACTERS;
```

OUTPUT :

```
LASTNAME        RPAD(RTRIM(LASTNAME)
------------------  ----------------------------------------
PURVIS          PURVIS**************
TAYLOR          TAYLOR**************
CHRISTINE       CHRISTINE***********
ADAMS           ADAMS***************
COSTALES        COSTALES************
KONG            KONG****************
6 rows selected.
```

The output proves that trim is working. Now try LTRIM :

INPUT :

```
SQL> SELECT LASTNAME, LTRIM(LASTNAME, 'C')
  2   FROM CHARACTERS;
```

OUTPUT :

```
LASTNAME        LTRIM(LASTNAME,
-------------------   ------------------------------
PURVIS          PURVIS
TAYLOR          TAYLOR
CHRISTINE       HRISTINE
ADAMS           ADAMS
COSTALES        OSTALES
KONG            KONG
6 rows selected.
```

Note the missing Cs in the third and fifth rows.

- **REPLACE**

REPLACE does just that. Of its three arguments, the first is the string to be searched. The second is the search key. The last is the optional replacement string. If the third argument is left out or NULL, each occurrence of the search key on the string to be searched is removed and is not replaced with anything.

INPUT :

```
SQL> SELECT LASTNAME, REPLACE(LASTNAME, 'ST') REPLACEMENT
  2   FROM CHARACTERS;
```

OUTPUT :

```
LASTNAME        REPLACEMENT
------------------    ------------------------
PURVIS          PURVIS
TAYLOR          TAYLOR
CHRISTINE       CHRIINE
ADAMS           ADAMS
COSTALES        COALES
KONG            KONG
6 rows selected.
```

If you have a third argument, it is substituted for each occurrence of the search key in the target string. For example :

INPUT :

```
SQL> SELECT LASTNAME, REPLACE(LASTNAME, 'ST','**') REPLACEMENT
  2     FROM CHARACTERS;
```

OUTPUT :

```
LASTNAME        REPLACEMENT
-------------------    -------------------------
PURVIS          PURVIS
TAYLOR          TAYLOR
CHRISTINE       CHRI**INE
ADAMS           ADAMS
COSTALES        CO**ALES
KONG            KONG
6 rows selected.
```

If the second argument is NULL, the target string is returned with no changes.

INPUT :

```
SQL> SELECT LASTNAME, REPLACE(LASTNAME, NULL) REPLACEMENT
  2     FROM CHARACTERS;
```

OUTPUT :

```
LASTNAME        REPLACEMENT
-------------------    -------------------------
PURVIS          PURVIS
TAYLOR          TAYLOR
CHRISTINE       CHRISTINE
ADAMS           ADAMS
COSTALES        COSTALES
KONG            KONG
6 rows selected.
```

- **SUBSTR**

This three-argument function enables you to take a piece out of a target string. The first argument is the target string. The second argument is the position of the first character to be output. The third argument is the number of characters to show.

INPUT :

```
SQL> SELECT FIRSTNAME, SUBSTR(FIRSTNAME,2,3)
  2     FROM CHARACTERS;
```

OUTPUT :

```
FIRSTNAME      SUB
---------------     ------
kelly          ell
CHUCK          HUC
LAURA          AUR
FESTER         EST
ARMANDO        RMA
MAJOR          AJO
6 rows selected.
```

If you use a negative number as the second argument, the starting point is determined by counting backwards from the end, like this :

INPUT :

SQL> **SELECT FIRSTNAME, SUBSTR(FIRSTNAME, –13, 2)**
 2 **FROM CHARACTERS;**

OUTPUT :

```
FIRSTNAME          SU
--------------------  ------
kelly              ll
CHUCK              UC
LAURA              UR
FESTER             ST
ARMANDO            MA
MAJOR              JO
6 rows selected.
```

Here, is another good use of the SUBSTR function. Suppose you are writing a report and a few columns are more than 50 characters wide. You can use the SUBSTR function to reduce the width of the columns to a more manageable size if you know the nature of the actual data. Consider the following two examples :

INPUT :

SQL> **SELECT NAME, JOB, DEPARTMENT FROM JOB_TBL;**

OUTPUT :

```
NAME_________________________________________________
JOB______________________DEPARTMENT________________
ALVIN SMITH
VICEPRESIDENT            MARKETING
1 Row selected.
```

ANALYSIS :

Notice how the columns wrapped around, which makes reading the results a little too difficult. Now try this select :

INPUT :

SQL> **SELECT SUBSTR(NAME, 1,15) NAME, SUBSTR(JOB,1,15) JOB, DEPARTMENT**
 2 **FROM JOB_TBL;**

OUTPUT :

```
NAME_________________JOB________________DEPARTMENT
ALVIN SMITH     VICEPRESIDENT    MARKETING
```

Much better!

- **TRANSLATE**

The function TRANSLATE takes three arguments : the target string, the FROM string, and the TO string. Elements of the target string that occur in the FROM string are translated to the corresponding element in the TO string.

INPUT :

SQL> **SELECT FIRSTNAME, TRANSLATE(FIRSTNAME**
 2 **'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ**

```
3    'NNNNNNNNNNAAAAAAAAAAAAAAAAAAAAAAAAAAA)
4    FROM CHARACTERS;
```

OUTPUT :

```
FIRSTNAME           TRANSLATE(FIRST
--------------------    -------------------------------
kelly                   kelly
CHUCK                   AAAAA
LAURA                   AAAAA
FESTER                  AAAAAA
ARMANDO                 AAAAAAA
MAJOR                   AAAAA
6 rows selected.
```

Notice that the function is case sensitive.

- **INSTR**

To find out where in a string a particular pattern occurs, use INSTR. Its first argument is the target string. The second argument is the pattern to match. The third and forth are numbers representing where to start looking and which match to report. This example returns a number representing the first occurrence of O starting with the second character :

INPUT :

```
SQL> SELECT LASTNAME, INSTR(LASTNAME, 'O', 2, 1)
  2    FROM CHARACTERS;
```

OUTPUT :

```
LASTNAME        INSTR(LASTNAME,'O',2,1)
---------------    -------------------------------------------
PURVIS                    0
TAYLOR                    5
CHRISTINE                 0
ADAMS                     0
COSTALES                  2
KONG                      2
6 rows selected.
```

- **LENGTH**

LENGTH returns the length of its lone character argument. For example :

INPUT :

```
SQL> SELECT FIRSTNAME, LENGTH(RTRIM(FIRSTNAME))
  2    FROM CHARACTERS;
```

OUTPUT :

```
FIRSTNAME              LENGTH(RTRIM(FIRSTNAME))
---------------        -------------------------------------------------
kelly                        5
CHUCK                        5
LAURA                        5
FESTER                       6
ARMANDO                      7
MAJOR                        5
6 rows selected.
```

3.7.5 Conversion Functions

These three conversion functions provide a handy way of converting one type of data to another. These examples use the table CONVERSIONS.

INPUT :

SQL> **SELECT * FROM CONVERSIONS;**

OUTPUT :

NAME	TESTNUM
40	95
13	23
74	68

The NAME column is a character string 15 characters wide, and TESTNUM is a number.

- **TO_CHAR**

The primary use of TO_CHAR is to convert a number into a character. Different implementations may also use it to convert other data types, like Date, into a character or to include different formatting arguments. The next example illustrates the primary use of TO_CHAR

INPUT :

SQL> **SELECT TESTNUM, TO_CHAR(TESTNUM)**
 2 **FROM CONVERT;**

OUTPUT :

TESTNUM	TO_CHAR(TESTNUM)
95	95
23	23
68	68

- **TO_NUMBER**

TO_NUMBER is the companion function to TO_CHAR, and of course, it converts a string into a number. For example :

INPUT :

SQL> **SELECT NAME, TESTNUM, TESTNUM*TO_NUMBER(NAME)**
 2 **FROM CONVERT;**

OUTPUT :

NAME	TESTNUM	TESTNUM*TO_NUMBER(NAME)
40	95	3800
13	23	299
74	68	5032

3.7.6 Miscellaneous Functions

Here, are three miscellaneous functions you may find useful.

- **GREATEST and LEAST**

These functions find the GREATEST or the LEAST member from a series of expressions. For example :

INPUT : SQL> **SELECT GREATEST('ALPHA', 'BRAVO','FOXTROT', 'DELTA')**
 2 **FROM CONVERT;**

OUTPUT :

```
GREATEST
----------------
FOXTROT
FOXTROT
FOXTROT
```

- **USER**

USER returns the character name of the current user of the database.

INPUT :

SQL> **SELECT USER FROM CONVERT;**

OUTPUT :

```
USER
------------------------------
PERKINS
PERKINS
PERKINS
```

There really is only one of me. Again, the echo occurs because of the number of rows in the table. USER is similar to the date functions explained earlier today. Even though USER is not an actual column in the table, it is selected for each row that is contained in the table.

3.5, 3.6, 3.7 Check Your Progress

Fill in the blanks

1) Procedures can be executes by ………………… command.
2) …………………Keyword is stands for recreating the procedure.
3) ………………… and is used to delete the proceure.

SOLVED EXAMPLES

1. Pass empno as an argument to procedure and modify salary of that emp.

```
    CREATE OR REPLACE PROCEDURE myproc1
    (p_no  IN  number)                              /* argument */
    IS
        v_sal   number(10,2);
    BEGIN
        Select  sal into v_sal
        From emp
        Where empno=p_no;
    If v_sal > 1000 then
        Update emp
        Set sal = v_sal*1.75
        Where empno=p_no;
    Else
        Update emp
        Set sal = 5000
        Where empno=p_no;
    End if;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        Dbms_output.put_line('Emp_no doesn't exists');
END myproc1;
```

2. **Pass a empno as argument to procedure and procedure will pass job to the calling program.**

```
CREATE OR REPLACE PROCEDURE myproc2
(p_no  IN   number, p_job OUT emp.job%TYPE)/* arguments */
IS
      v_job      emp.job%TYPE;
BEGIN
      Select  JOB  into v_job
      From emp
      Where empno=p_no;
      P_job:=v_job;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
     P_job:='NO';
END myproc2;
```

Calling procedure myproc2 using following code

```
Declare
  C_empno   number;
  C_job     emp.job%TYPE;
Begin
   Myproc2(&c_empno,c_job);
   If c_job='NO' then
     Dbms_output.put_line('Emp_no doesn't exists');
       Else
           Dbms_output.put_line('Job of emp. Is ' || c_job);
       End if;
End;
/
```

Execute this code using

SQL> /

3. **Pass salary to procedure and procedure will pass no. of employee(s) having salary equal to given salary in the same variable. (Use IN OUT variable).**

```
CREATE OR REPLACE PROCEDURE myproc3
(p_sal  IN  OUT   emp.sal%TYPE)                                /* arguments */
IS
 v_count   number;
BEGIN
    Select count(*)  into v_count
    From emp
    Where sal=p_sal;
   P_sal:=v_count;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
     P_sal:=0;
END myproc3;
```

Calling procedure myproc3 using following code

```
Declare
 C_sal     emp.sal%TYPE;
Begin
   C_sal:=&c_sal;
   Myproc3(c_sal);
   If c_sal=0 then
     Dbms_output.put_line('No employee is having salary equal to accepted salary');
       Else
```

```
        Dbms_output.put_line('No. of emp. having salary = accepted salary are  ' ||
c_sal);
    End if;
End;
/
```

Execute this code using
SQL/

3.8 SUMMARY

Joins are used to manipulate data from multiple tables. Types of joins are 1) Equi-jions 2) Non-equi –joins Procedures are simply a named PL/SQL block, that executes certain tasks.
Functions increase your ability to manipulate information you retrieved using basic functions of SQL these are as follows
1) Aggregate Functions 2) Date & time Functions 3) Arithmetic Functions 4) Character Functions 5) conversion Functions 6) Miscellaneous functions

3.9 CHECK YOUR PROGRESS-ANSWERS

3.1, ,3.3

1) Joins/Sub-Query
2) Function
3) In, Out, Inout

3.5,3.6,3.7

1) Exec
2) Recreate
3) Drop

3.10 QUESTIONS FOR SELF – STUDY

Q.1 Why cover outer, inner, left, and right joins when I probably won't ever use them ?

Q.2 How many tables can you join on ?

Q.3 Would it be fair to say that when tables are joined, they actually become one table ?

Q.4 How many rows would a two-table join produce if one table had 50,000 rows and the other had 100,000 ?

Q.5 In the WHERE clause, when joining the tables, should you do the join first or the conditions ?

Q.6 In joining tables are you limited to one-column joins, or can you join on more than one column ?

Q.7 In the section on joining tables to themselves, the last example returned two combinations. Rewrite the query so only one entry comes up for each redundant part number.

Q.8 Rewrite the following query to make it more readable and shorter.

INPUT :

```
select orders.orderedon, orders.name, part.partnum,
       part.price, part.description from orders, part
       where orders.partnum = part.partnum and orders.orderedon
       between '1-SEP-96' and '30-SEP-96'
       order by part.partnum;
```

Q.9 From the PART table and the ORDERS table, make up a query that will return the following :

OUTPUT :

ORDEREDON	NAME	PARTNUM	QUANTITY
========	=======	========	==========
2-SEP-96	TRUE WHEEL	10	1

Q.10 What is the advantages of procedures.

Q.11 How to create a procedure ? Explain.

Q.12 Give Syntax of :

(a) Deleting a procedure

(b) Executing a procedure

(c) Creating a procedure.

Q.13 Which function capitalizes the first letter of a character string and makes the rest lowercase ?

Q.14 Which functions are also known by the name group functions ?

Q.15 Will this query work ?

```
SQL> SELECT COUNT(LASTNAME) FROM CHARACTERS;
```

Q.16 How about this one ?

```
SQL> SELECT SUM(LASTNAME) FROM CHARACTERS;
```

Q.17 Assuming that they are separate columns, which function(s) would splice together FIRSTNAME and LASTNAME ?

Q.18 What does the answer 6 mean from the following SELECT ?

INPUT :

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

OUTPUT :

```
COUNT(*)
```

Q.19 Will the following statement work ?

```
SQL> SELECT SUBSTR LASTNAME,1,5 FROM NAME_TBL;
```

Q.20 Using today's TEAMSTATS table, write a query to determine who is batting under .25. (For the baseball-challenged reader, batting average is hits/ab.)

Q.21 Using today's CHARACTERS table, write a query that will return the following:

```
INITIALS ……………………….. CODE
K.A.P.                        32
1 row selected
```

3.11 SUGGESTED READINGS

Teach Yourself SQL in 21 Days **-** By Ryan K. Stephens Ronald R Plew

Using Oracle Application - By Jim Crum

◈◈◈

NOTES

CHAPTER 4

PL / SQL

4.0 Objectives
4.1 Introduction to PL / SQL
4.2 Architecture of PL / SQL
4.3 Fundamentals of PL / SQL
4.3.1 PL / SQL Data type
4.3.2 If Statement
4.4 Loops in PL / SQL
4.4.1 Simple Loop
4.4.2 For Loop
4.4.3 While Loop
4.5 Solved Examples
4.6 Built-in-Functions
4.6.1 Conditional Control
4.6.2 Iterative Control
4.6.3 Sequential Control
4.7 Cursor Management in PL / SQL
4.8 Exception (Error) Handling
4.8.1 Predefined Exception
4.8.2 User Defined Function
4.9 Summary
4.10 Check Your Progress-*Answers*
4.11 Questions for Self – Study
4.12 Suggested Readings

4.0 OBJECTIVES

After reading this chapter you will able to

- Describe Pl/SQL
- State Loops in PL/SQL
- Built in Function
- Describe Cursor Management
- Describe Exception

4.1 INTRODUCTION TO PL/SQL

PL/SQL stands for Procedural Language/SQL. PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL.PL/SQL is not case sensitive. 'C' style comments (/* ……… */) may be used in PL/SQL programs whenever required.

All PL/SQL programs are made up of blocks, each block performs a logical action in the program. A PL/SQL block consists of three parts

1. Declaration section
2. Executable section
3. Exception handling section

Only the executable section is required. The other sections are optional.

A PL/SQL block has the following structure :

```
DECLARE
   /*  Declaration section */
BEGIN
  /* Executable section */
EXCEPTION
 /* Exception handling section */
END;
```

1. **Declaration section :**

This is first section which is start with word Declare. All the identifiers (constants and variables) are declared in this section before they are used in SELECT command.

2. **Executable section :**

This section contain procedural and SQL statements. This is the only section of the block which is required. This section starts with 'Begin' word.

- The only SQL statements allowed in a PL/SQL program are SELECT, INSERT, UPDATE, DELETE and several other data manipulation statements.
- Data definition statements like CREATE, DROP or ALTER are not allowed.
- The executable section also contains constructs such as assignments, branches, loops, procedure calls and trigger which are all discussed in detail in subsequent chapters.

3. **Exception handling section :**

This section is used to handle errors that occurs during execution of PL/SQL program. This section starts with 'exception' word .

The 'End' indicate end of PL/SQL block.

Oracle PL/SQL programs, can be invoke either by typing it in sqlplus or by putting the code in a file and invoking the file. To execute it use '/' on SQL prompt or use '.' and run.

4.2 ARCHITECUTRE OF PL/SQL

The PL/SQL compilation and run-time system is a technology, not an independent product. Think of this technology as an engine that compiles and executes PL/SQL blocks and subprograms. The engine can be installed in an Oracle server or in an application development tool such as Oracle Forms or Oracle Reports. So, PL/SQL can reside in two environments :

1. The Oracle server
2. Oracle tools.

These two environments are independent. PL/SQL is bundled with the Oracle server but might be unavailable in some tools. In either environment, the PL/SQL engine accepts as input any valid PL/SQL block or subprogram. Fig. 3.1 shows the PL/SQL engine processing an anonymous block. The engine executes procedural statements but sends SQL statements to the SQL Statement Executor in the Oracle server.

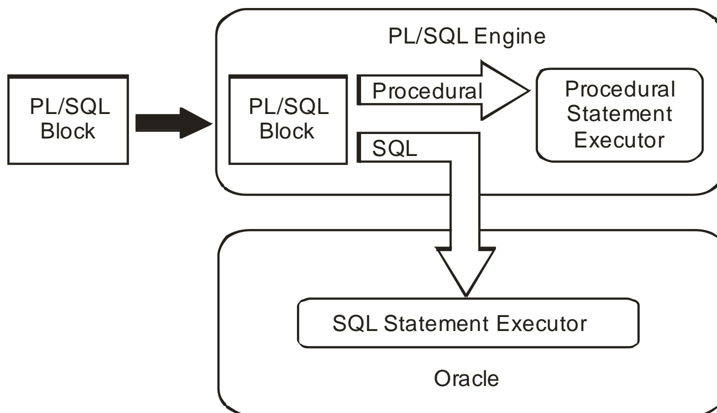

Fig.3.1 : PL/SQL Engine

4.2 FUNDAMENTALS OF PL/SQL

4.3.1 PL/SQL Data Types

PL/SQL and Oracle have their foundations in SQL. Most PL/SQL data types are native to Oracle's data dictionary, there is a very easy integration of PL/SQL code with the Oracle Engine.

The default data types that we can declare in PL/SQL are **number** (for storing numeric data), **char** (for storing character data), date (for storing date and time data) **boolean** (for storing TRUE, FALSE or NULL). **number**, **char** and **date** data types can have NULL values.

Here, we explain two data types,

1. Variable,
2. Constant.

1. Variables and types of declaration in PL/SQL :

The SELECT statement has a special form in PL/SQL in which a single tuple is placed in variables. The information from the database is transferred into *variables* which is used in PL/SQL programs. Every variable has a specific type associated with it.

That type can be :

1. A generic type used in PL/SQL
2. A type same as used by SQL for database columns.

The most commonly used generic type is NUMBER. Variables of type NUMBER can hold either an integer or a real number.

For example :

```
DECLARE
    Salary Number;
```

The most commonly used character string type is VARCHAR2(*n*), where *n* is the maximum length of the string in bytes.

For example :

```
DECLARE
    My_name  VARCHAR2(20);
```

The variable can contain any data type that is valid for SQL and Oracle (such as char, number, long, varchar2, & date) in addition to these types PL/SQL allows following types :

- **Binary integer :** Range is –2,147,483,647 to 2,147,483,647

- **Positive :** Range is 1 to 2,147,483,647.
- **Natural :** Range is 0 to 2,147,483,647.
- **Boolean :** Assigned values either True, False or NULL.
- **%type :** Assign the same type to variable as that of the relation column declared in database.

  If there is any type mismatch, variable assignments and comparisons may not work the way you expect, so instead of hard coding the type of a variable, you should use the %TYPE operator.

For example :

```
DECLARE
        My_name emp.ename%TYPE;
```

gives PL/SQL variable my_name whatever type was declared for the ename column in emp table.

- **%rowtype :** A variable can be declared with %rowtype that is equivalent to a row of a table i.e. record with several fields. The result is a record type in which the fields have the same names and types as the attributes of the relation.

For example :

```
DECLARE
        Emp_rec  emp1%ROWTYPE;
```

This makes variable emp_rec be a record with fields name and salary, assuming that the relation has the schema emp1(name, salary).

The initial value of any variable, regardless of its type, is NULL.

2. Constants :

Declaration of a constant is similar to declaring a variable except that the keyword **constant** must be added to the variable name **and** a value assigned immediately. Thereafter, no further assignments to the constant are possible, while the constant is **within** the constant is **within** the scope of the PL/SQL block.

There are two types :

(i) Raw and

(ii) Rawid

(i) Raw : Raw types are used to store **binary** data. Character variables are automatically converted between character sets by Oracle, if necessary. These are similar to char variables, except that they are not converted between character sets. It is used to store fixed length binary data. The maximum length of a raw variable is 32,767 bytes. However, the maximum length of a database raw column is 255 bytes.

Long raw is similar to long data, except that PL/SQL will not convert between character sets. The maximum length of a long raw variable is 32,760 bytes. The maximum length of a long raw column is 2 GB.

(ii) Rowid : This data types is the same as the database **ROWID** pseudo-column type. It can hold a rowid, which can be considered as a unique key for every row in the database. Rowids are stored internally as a fixed length binary quantity, whose actual fixed length varies depending on the operating system.

Various **DBMS_ROWID** functions are used to extract information about the ROWID pseudo-column. **Extented** and **Restricted** are two rowid formats. **Restricted** is used mostly to be backward compatible with previous versions of Oracle. The **Extended** format takes advantage of new Oracle features.

The **DBMS_ROWID** package has several procedures and functions to interpret the ROWIDs of records. The Table 7.1 shows the **DBMS_ROWID** functions :

Table 4.1 : Functions of DBMS_ROWID

FUNCTION	DESCRIPTION
ROWID_VERIFY	Verifies if the ROWID can be extended; 0 = can be converted to extended format; 1 = cannot be converted to extended format.
ROWID_TYPE	0 = ROWID, 1 = Extended
ROWID_BLOCK_NUMBER	The block number that contains the record; 1 = Extended ROWID
ROWID_OBJECT	The object number of the object that contains the record.
ROWID_RELATIVE_FNO	The relative file number contains the record.
ROWID_ROW_NUMBER	The row number of the record.
ROWID_TO_ABSOLUTE_FNO	The absolute file number; user need to input rowid_val, schema and object; the absolute file number is returned.
ROWID_TO_EXTENDED	Converts the ROWID from Restricted to Extended; user need to input restr_rowid, schema, object; the extended number is returned.
ROWID_TO_RESTRICTED	Converts the ROWID from Extended to Restricted.

ROWID is a pseudo-column that has a unique value associated with each record of the database.

The **DBMS_ROWID** package is created by the,

ORACLE_HOME/RDBMS/ADMIN/DBMSUTIL.SQL script.

This script is automatically run when the Oracle instance is created.

Operator Precedence :

If we combine AND and OR in the same expression, the AND operator takes precedence over the OR operator (which means it's executed first). The comparison operators take precedence over AND. We can override these using parentheses.

PL SQL Expressions

Expressions are a composite of operators and operands . In the case of a mathematical expression ,the operand is the number and operator is the symbol such as + or – that acts on the operand. The expression value is the evaluated total of the operands using the operators.
Operators are divided into categories that describe the way that act upon operands.
-Comparison operators are binary, meaning they work with two operands. Examples of comparison operators are the greater than (>) ,less than(<) and equal(=) signs ,among others.
-Logical operators include AND,OR and NOT
-Arithmetic operators include addition/positive(+),subtraction/negative(-),multiplication(*),and division(/).
-The assignment operator is specific to PL/SQL and is written as colon-equal (:=)
-The lone character operator is a double pipe(||) that joins two strings together, concatenating the operands.
-Other basic SQL operators include IS NULL, IN and BETWEEN.

4.3.2 If statement in PL/SQL

PL/SQL allows decision making using if statement.

An IF statement in PL/SQL looks like :

```
IF <condition> THEN
       <statement_list>
END IF;
```

If condition is true the statements present inside IF will get executed.

If…. Else construct :

```
IF <condition> THEN
      <statement_list>
ELSE
      <statement_list>
END IF;
```

For example :

1. Accept two numbers and print the largest number

```
DECLARE
x  number;
y number;
BEGIN
   x :=&x;
   y :=&y;
   if (x>y) then
      dbms_output.put_line('x is largest than y');
   else
      dbms_output.put_line('y is largest than x');
   end if;
End;
/
SQL> /
Enter value for x : 7
old 5 : x := &x;
new 5 : x :=7;
Enter value for y : 8
old 6 : y :=&y
new 6 : y :=8
Addition is 15
PL/SQL procedure completed
y is largest than x
PL/SQL procedure successfully completed.
```

2. Check whether the salary of 'BLAKE' is grater than 5000 or not.

```
DECLARE
   B_salary emp.sal%type;
BEGIN
   Select sal into B_salary
   From emp
   Where ename='BLAKE';
If (B_salary > 5000) then
   dbms_output.put_line('Blake salary is largest than 5000');
else
   dbms_output.put_line('Blake salary is less than 5000');
end if;
End;
/
SQL> /
Blake salary is less than 5000
PL/SQL procedure successfully completed.
```

If with a Multiway Branch :

```
IF <condition_1> THEN
   ELSEIF <condition_2> THEN
      <statement_list>
```

```
    ELSEIF <condition_n> THEN
        <statement_list>
    ELSE
        <statement_list>
END IF;
```

4.1 - 4.3 Check Your Progress

Fill in the blanks

1) …………………Section is used for declaration of variables.

2) SQL statements are written in ………………… section.

4.4 LOOPS IN PL/SQL

There are three types of loops in PL/SQL :

1. Simple loop
2. For…loop
3. While loop.

4.4.1 Simple Loop

Syntax 1 :

```
LOOP
    <commands> /* A list of statements. */
    if <codition> then
        EXIT;
    End if;
END LOOP;
```

The loop breaks if <condition> is true.

For example :

```
DECLARE
        i NUMBER := 0;
BEGIN
        LOOP
                i := i+1;
                dbms_output.put_line(i);
                If( i>=10) then
                    EXIT;
                End if;
        END LOOP;
END;
/
SQL> /
1
2
3
4
5
6
7
8
9
10
PL/SQL procedure successfully completed.
```

Syntax 2 :

```
LOOP
        <commands> /* A list of statements. */
         EXIT WHEN <condition>;
END LOOP;
```

For example :

```
DECLARE
        i NUMBER := 0;
BEGIN
        LOOP
                i := i+1;
                dbms_output.put_line(i);
                EXIT when i>=10;
        END LOOP;
END;
/
SQL> /
1
2
3
4
5
6
7
8
9
10
PL/SQL procedure successfully completed.
```

The loop breaks when <condition> is true.

4.4.2 For…loop

Syntax :

```
FOR <var> IN[reverse] <start>. .<finish> LOOP
        <commands> /* A list of statements. */
END LOOP;
```

Here, <var> can be any variable; it is local to the for-loop and need not be declared. Also, <start> and <finish> are constants. The value of a variable <var> is automatically incremented by 1.

The commands inside the loops are automatically executed until the final value of variable is reached. Reverse is optional part, when you want to go from maximum value to minimum value in that case reverse is used.

For example :

```
BEGIN
    For i in 1. .10 LOOP
        dbms_output.put_line(i);
    END LOOP;
END;
SQL> /
1
2
3
4
5
6
7
8
9
```

```
10
PL/SQL procedure successfully completed.
```

4.4.3 While loop

Syntax :

```
WHILE <condition> LOOP
        <commands> /* A list of statements. */
END LOOP;
```

This loop executes the commands if the condition is true.

For example :

```
DECLARE
    i NUMBER := 0;
BEGIN
    While i<=10 LOOP
        i := i+1;
        dbms_output.put_line(i);
    END LOOP;
END;
/
SQL> /
1
2
3
4
5
6
7
8
9
10
PL/SQL procedure successfully completed.
```

4.5 SOLVED EXAMPLES

4.5.1 Simple PL/SQL

1. **Accept two numbers and print the largest number.**

```
DECLARE
   n1 number;
   n2 number;
BEGIN
   n1:=&n1;
   n2:=&n2;
   if(n1<n2) then
      dbms_output.put_line(n1 ||' is largest');
   else if (n1<n2) then
         dbms_output.put_line(n2 ||' is largest');
      else
         dbms_output.put_line('Both are equal');
      end if;
   end if;
End;
```

Output :

```
SQL> /
Enter value for n1: 23
old 5: n 1 :=&n 1;
new 5: n 1:=23;
Enter value for n2: 12
old 6: n2:=&n2;
```

```
new 6: n2:= 12;
23 is largest
SQL>/
Enter value for n1: 11
old 5: n 1 :=&n 1;
new 5: n1:=11;
Enter value for n2: 11
old 6: n2:=&n2;
new 6: n2:=11;
Both are equal
```

2. **Accept a number and check whether it is odd or even.**
   If it is even no. print square of it otherwise cube of it.

```
DECLARE
    n1 number;
BEGIN n1:=&n1;
    if(mod(n1,2)=0) then
        dbms_output.put_line(n1 ||' is even no');
        dbms_output.put_line('Square of '|| n1 ||' is '|| n1*n1);
    else
        dbms_output.put_line(n1 at ||' is odd no');
    dbms_output.put_line('Cube of ' | | n 1 | | ' is ' | | n1 * n1 * n1);
    end if;
End;
```

Output :

```
SQL> /
Enter value for n1: 2
old 4: n1:=&n1;
new 4: n1:=2;
2 is even no.
Square of 2 is 4
SQL> /
Enter value for n1: 3
old 4: n1:=&n1;
new 4: n1 :=3;
3 is odd no.
Cube of 3 is 27
```

4.5.2 PL/SQL Block using Table

1. Accept the deptno and print the no. of employees working in that department.

```
DECLARE
    v_deptno emp.deptno%type;
    v_count number;
BEGIN
    v_deptno:=&v_deptno;
    select count(*) into v_count
    from emp
    where deptno=v_deptno;
    dbms_output.put_line('No. of emp working in '|| v_deptno || 'are' || v_count);
End;
/
```

Output :

```
SQL> /
```

```
Enter value for v_deptno: 20
old 5: v_deptno:=&v_deptno;
new 5: v_deptno:=20;
No. of emp working in 20 are 2
```

2. **Accept the deptno and print the department name and location.**

```
DECLARE
     v_deptno dept.deptno%type;
     v_dname dept.dname%type;
     v_loc dept.loc%type;
BEGIN
     v_deptno:=&v_deptno;
     select dname,loc into v_dname,v_loc
     from dept
     where deptno=v_deptno;
     dbms_output.put_line('Department name is '|| v_dname ||' and location is  '||
     v_loc);
End;
/
```

Output :

```
SQL> /
Enter value for v_deptno: 10
old 6: v_deptno:=&v_deptno;
new (6: v_deptno:= 10;
Department name is ACCOUNTING and location is NEW YORK
```

4.6 BUILT-IN-FUNCTIONS

The control statements can be classified into the following categories :

- Conditional Control
- Iterative Control
- Sequential Control

We study here Conditional and Iterative control.

4.6.1 Conditional Control

PL/SQL allows the use of an **IF** statement to control the execution of a block of code. In PL/SQL, the **IF – THEN – ELSIF – ELSE – END IF** construct in code blocks allow specifying certain conditions under which a specific block of code should be executed.

Syntax :

```
IF <Condition> THEN
     <Action>
ELSEIF <Condition> THEN
     <Action>
ELSE
     <Action>
END IF :
```

For example :

Write a PL/SQL code block that will accept an account number from the user, check if the users balance is less than the minimum balance, only then deduct Rs. 100/- from the balance. The process is fired on the ACCT_MSTR table.

```
DECLARE
```

```
/*  Declaration of memory variables and constants to be used in the Execution section
*/
    mCUR_BAL number (11, 2);
    mACCT_NO varchar2(7);
    mFINE number(4) := 100;
    mMIN_BAL constant number(7, 2) := 5000.00;
BEGIN
/*  Accept the Account number from the user */
    mACCT_NO := &mACCT_NO;
/*  Retrieving the current balance from the ACCT_MSTR table where the ACCT_NO
in the table is equal to the mACCT_NO entered by the user */
    SELECT CURBAL INTO mCUR_BAL FROM ACCT_MSTR WHERE
ACCT_NO=mACCT_NO;
/*  Checking if the resultant balance is less than the minimum balance of Rs. 5000. If
the condition is satisfied an amount of Rs. 100 is deducted as a fine from the current
balance of the corresponding ACCT_NO */
    IF mCUR_BAL <mMIN_BAL THEN
        UPDATE ACCT_MSTR SET CURBAL = CURBAL_mFINE
            WHERE ACCT_NO = mACCT_NO;
    END IF;
END;
```

Output :

```
Enter value for mACCT_NO : 'SB9'
Old 11 : mACCT_NO : &mACCT_NO;
new 11 : mACCT_NO : = 'SB9';
```

4.6.2 Iterative Control

Iterative control indicates the ability to repeat or skip sections of a code block. A **loop** marks a sequence of statements that has to be repeated. The keyword **loop** has to be placed before the first statement in the sequence of statements to be repeated, while the keyword **end loop** is placed immediately after the last statement in the sequence. Once a loop begins to execute, it will **go on forever**. Hence, a conditional statement that controls the number of times a loop is executed **always accompanies** loops.

PL/SQL supports the following structures for iterative control :

Simple Loop :

In simple loop, the key word **loop** should be placed before the first statement in the sequence and the keyword **end loop** should be written at the end of the sequence to end the loop.

Syntax :

```
    Loop
        <Sequence of statements>
    End loop :
```

For example :

Create a simple loop such that a message is displayed when a loop exceeds a particular value.

```
DECLARE
    i number := 0;
BEGIN
    LOOP
        i := i + 2;
```

```
        EXIT WHEN I > 10;
    END LOOP;
    dbms_output.put_line(Loop exited as the value of i has reached '|| to_char(i));
END;
```

Output :

Loop exited as the value of i has reached 12

PL/SQL procedure successfully completed.

The WHILE Loop :

Syntax :

```
    WHILE <Condition>
    LOOP
        <Action>
    END LOOP;
```

For example :

Write a PL/SQL code block to calculate the area of a circle for a value of radius varying from 3 to 7. Store the radius and the corresponding values of calculated area in an empty table named **Areas**, consisting of two columns **Radius** and **Area**.

Table Name : Areas

RADIUS	AREA

Create the table AREAS as :

```
    CREATE TABLE AREAS (RADIUS NUMBER (5), AREA NUMBER(14,2));
DECLARE
    /* Declaration of memory variables and constants to be used in the Execution
section */
    pi constant number(4, 2) := 3.14;
    radius number(5);
    area number(14, 2);
BEGIN
/*  Initialize the radius to 3, since calculations are required for radius 3 to 7 */
radius : = 3;
/*  Set a loop so that it fires till the radius value reaches 7 */
    WHILE RADIUS <= 7
    LOOP
    /*  Area calculation for a circle */
        area := pi *power(radius, 2);
    /*  Insert the value for the radius and its corresponding area calculated in the
table */
        INSERT INTO areas VALUES (radius, area);
    /*  Increment the value of the variable radius by 1 */
        radius := radius + 1;
    END LOOP;
END;
```

The above PL/SQL code block initializes a variable **radius** to hold the value of 3. The area calculations are required for the radius between 3 and 7. The value for area is calculated first with radius 3 and the radius and area are inserted into the table **Areas**. Now, the variable holding the value of radius is incremented by 1, i.e. it now holds the value 4. Since the code is held within a loop structure, the code continues to fire till the radius value reaches 7. Each time the value of radius and area is inserted into the areas table.

After the loop is completed the table will now hold the following :

Radius	Area
3	28.26
4	50.24
5	78.5
6	113.04
7	153.86

The FOR Loop

Syntax :

```
FOR variable IN [REVERSE] start..end
LOOP
    <Action>
END LOOP;
```

For example :

Write a PL/SQL block of code for inverting a number 5639 to 9365.

```
DECLARE
/*  Declaration of memory variables and constants to be used in the Execution section
*/
    given_number varchar(5) := ‘5639’;
    str_length number(2);
    inverted_number varchar(5);
BEGIN
/*  strore the length of the given number */
        str_length := length(given_number);
/*  Initialize the loop such that it repeats for the number of times equal to the length of
the given number. Also, since the number is required to be inverted, the loop should
consider the last number first and store it i.e. in reverse order */
FOR cntr IN REVERSE 1..str_length
/*  Variables used as counter in the for loop need not be declared i.e. cntr declaration
is not required */
LOOP
    /* The last digit of the number is obtained using the substr function and stored in a
variable, while retaining the previous digit stored in the variable */
    inverted_number := inverted_number || substr (given_number, cntr, 1);
END LOOP;
/*  Display the initial number, as well as the inverted number, which is stored in the
variable on screen */
    dbms_output.put_line (‘The Given number is “|| given_number);
    dbms_output.put_line (‘The Inverted number is ‘|| inverted_number);
END;
```

Output :

The Given number is 5639