SECTION 2 DBMS LAB

Structure **Page Nos.**

2.0 INTRODUCTION

Information Technology (IT) has influenced organisational performance and competitive standing. The increasing processing power and sophistication of analytical tools and techniques require a strong foundation of structured form of data. Thus, the presence of a commercial, secure, reliable database management system is the major requirement of businesses.

This section is an attempt to familiarise you with commercial database management features, so that you are able to apply some of the important features of DBMS in a commercial situation. This would require a sound background of the concepts of DBMS which can be acquired through the courses MCS-023: Introduction to DBMSs and MCS-043: Advanced DBMSs. You would also have to apply the conceptual background to obtain problem solving skills using a commercial DBMS. The topics that have been included in this section are sub-queries, joins, views, indexes,

embedded languages including cursors, triggers, and some advanced features like object tables, nested tables, user management etc.

Please note that we have given many examples in this lab section in different sessions, highlighting the basic syntax of various statements; however, they are not exhaustive, as commercial DBMSs implement a large number of statements and functions. Therefore, you must refer to the user reference manuals of the commercial DBMS for more details on these topics. Also, we have given the examples for a typical DBMS; however, you may do practicals on any commercial RDBMS supporting such concepts. However, you should find the changes in the statements used here for the DBMS you may be using. Remember, the objectives of this section consisting of ten sessions will be completed only if you *practice* using any commercial DBMS.

2.1 OBJECTIVES

After going through this section and performing all the practical exercises you should be able to:

- write and run SQL queries using join;
- create and run sub-queries, views, indexes and procedures;
- write simple embedded SQL related programs;
- handle errors and manage transactions;
- write simple triggers;
- create advanced tables and objects, and
- perform simple user management.

2.2 SESSION 1: SUBQUERIES AND JOINS

Objectives

At the end of this session, you should be able to:

- define subqueries and joins
- create different types of subqueries
- perform different types of join queries.

The queries given in this session and most of the later sessions are based on the following relations:

teacher (t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
class (class_no, t_no, room_no)
payscale (Min_limit, Max_limit, grade)

2.2.1 Subquery

Subquery means a SELECT statement which retrieves the values from the table and passes these values as argument to the main or calling SELECT statement. In simple words, nested queries are subqueries. Subqueries can be of different types:

1) Single-row subquery

2) Multiple-row subquery

3) Multiple-column subquery

4) Correlated query.

Single-Row Subquery

Single-row subquery is the query that returns only one value or row to the calling SELECT statement. For example:

Query 1: Display the name of the teacher who is oldest among all teachers.
Explanation: To know the name of the teacher who is oldest, you need to first find the minimum birth date and then corresponding to that date display the name of the teacher.

```
SELECT f_name, l_name
FROM teacher
WHERE birthdate = (SELECT MIN(birthdate)
                   FROM teacher);
```

Query 2: Display teacher numbers and names of those teachers who are earning less than 'Jatin'.
Explanation: To find the list of teachers earning less than 'Jatin', you need to find first the salary of 'Jatin'.

```
SELECT t_no, f_name, l_name
FROM teacher
WHERE salary < (SELECT salary
                FROM teacher
                WHERE UPPER(f_name) = 'JATIN');
```

Multiple-Row Subquery

Multiple-row subqueries are the queries that return only more than one value or rows to the calling SELECT statement. For example:

Query 3: Display the list of all teachers who are earning equal to any teacher who have joined before '31-dec-94'
Explanation: First you need to know the salaries of all those who have joined before '31-dec-94' and then any teacher whose salary matches any of these returned salaries. IN operator looks for this existence into the set. You can also use Distinct to avoid duplicate salary tuples.

```
SELECT t_no, f_name, l_name
FROM teacher
WHERE salary IN   (SELECT salary
                   FROM teacher
                   WHERE joindate < '31-dec-94');
```

Query 4: Display the list of all those teachers whose salary is greater than any other teacher with job title 'PRT'.
Explanation: First you need to know the salaries of all those who are 'PRT' and then any teacher whose salary is greater than any of these returned salaries. ANY operator looks for inequality.

```
SELECT t_no, f_name, l_name, salary
FROM teacher
WHERE salary >    ANY (SELECT salary
                       FROM teacher
                       WHERE UPPER (title) = 'PRT');
```

Query 5: Display the list of all those teachers whose salary is greater than all the teachers with job title as 'PRT'.
Explanation: First you need to know the salaries of all those who are 'PRT' and then any teacher whose salary is greater than all of these returned salaries. ALL operator looks for inequality.

```
SELECT t_no, f_name, l_name, salary
FROM teacher
WHERE salary > ALL (SELECT salary
```

```
                    FROM teacher
                    WHERE UPPER(title) = 'PRT');
```

Single-row Subquery		Multiple-row subquery	
Operator	**Description**	**Operator**	**Description**
=	Equal to	IN	Returns true if any of the values in the list match i.e. equality check
>	Greater than		
<	Less than	ALL	Returns true if all the values returned by the subquery match the condition
>=	Greater than equal to		
<=	Less than or equal to	ANY	Returns true if any of the values returned by subquery match the condition.
<>	Not equal to		

Figure 1: Operators for subqueries

Multiple-Column subquery

Multiple-Column subquery is the subquery that returns more than one column to the calling or main SELECT statement. For example:

Query 6: Display the list of all teachers whose job title and salary is same as that of the employee whose first name is 'Jaideep'.
Explanation: Firstly you need to find the job title and salary of 'Jaideep' and then you need to find all other teachers whose job title and salary exactly matches Jaideep's job title and salary.

```
SELECT t_no, f_name,l_name, title, salary
FROM teacher
WHERE (title, salary) = (SELECT title, salary
                         FROM teacher
                         WHERE LOWER(f_name) = 'jaideep');
```

Correlated Subqueries

This is another type of subquery where the subquery is executed for each and every record retrieved by the calling or main SELECT statement. A correlated subquery returns only a Boolean value (true/false). A *true* is returned if the subquery returns one or more records otherwise a *false* is returned. The operator EXISTS can be used for these types of subqueries. For example:

Query 7: Display the records in the format given below for all class teachers:

Jaideep Kumar is a class teacher

Explanation: The main query uses the EXISTS operator to find whether the teacher is a class teacher or not. If the correlated subquery returns a true value, then the record retrieved by main SELECT statement is accepted otherwise it is rejected.

```
SELECT f_name, l_name, 'is a class teacher'
FROM teacher
WHERE exists (SELECT *
              FROM class
              WHERE class.t_no = teacher.t_no)
```

2.2.2 Joins

Joins means retrieving data from more than one table in a single query. There are several types of joins:

1) **Equijoin** – In this type of join, two or more tables are joined over common columns and common values. For example:

   Query 8: Display names of all the teachers who are class teachers.

```
SELECT f_name, l_name
FROM teacher t, class c
WHERE t.t_no = c.t_no;
```

2) **Non-Equijoin** - In this type of join, two or more tables do not have common columns and common values but they are joined indirectly. For example:

Query 9: Display names, salaries and salary grades of all teachers.

```
SELECT f_name, l_name, salary, grade
FROM teacher, payscale
WHERE salary BETWEEN min_limit AND max_limit;
```

3) **Outer join** - In this type of join, two or more tables are joined over common column but the records may or may not have common values. For example:

Query 10: Display names and class numbers of all the teachers. In addition display the classes of those teachers who are class teachers. Thus, the result should include names of teachers who are not class teachers.

```
SELECT f_name, l_name, class_no
FROM teacher t, class c
WHERE t.t_no = c.t_no (+) ;
```

4) **Self join** - In this type of join, the table is joined to itself. For example:

Query 11: Display teacher number and names of all teachers along with the names of their supervisors and number. Please note that the supervisor of a teacher is also a teacher. Therefore, the query requires a self-join.

```
SELECT t.f_name, t.l_name, t.supervisor, s.f_name, s.l_name
FROM teacher t, teacher s
WHERE t.supervisor = s.t_no;
```

5) **Cartesian join** – In this type of join, each record of one table is joined to each record of the other table i.e., the join condition is not given. For example:

Query 12: Show all possible teacher – class values.

```
SELECT f_name, l_name, class_no
FROM teacher t, class c;
```

2.2.3 Exercise 1

1) Please attempt the following problems for the teacher, class and payscale relations given in this section.

   (a) Display the name of the teacher(s) who is (are) the youngest among all the teachers.

   (b) Display details of all the teachers who have the same job title as that of 'Jaideep'

   (c) Display the list of all the teachers who have joined after '10-Jul-95' and whose salary is equal to that of any of the teachers who joined before '10-Jul-95'.

   (d) Use a correlated query to determine the teachers who are not class teachers.

   (e) Identify all those teachers who are in grade 'B'.

   (f) Display the names and numbers of all teachers who are class teachers and are in grade 'C'.

   (g) Display the names of all teachers who are supervisors.

   (h) Display the teacher id and salaries of all those teachers who are in grade 'A' or 'C' and who have at least two L's in their names.

(i) Display details of all those teachers who are class teachers of classes 1 to 5.

(j) Display the names of all teachers along with their dates of birth whose birthday is in the current month.

2) In an Open University a student's data is to be maintained using relations. The university maintains data of all the students, their batches, Regional Centres and study centre details. Each batch of students has one or more representative students. The batches are taught by same or different faculty.

(a) Design and implement the suitable relations for the University. Make and state suitable assumptions.

(b) Write at least 15 queries (they must include at least one query of each type given in this section 2.2) for the database. Implement these queries using SQL. Also explain the purpose of each query.

3) Design a suitable database system for a bank along with 20 possible queries to the database (the queries should be such that their solution involves subqueries or joins or both). Implement the database and the queries in a commercial DBMS using SQL.

2.3 SESSION 2: CREATING VIEWS, INDEXES AND QUERIES ON THEM

Objectives

At the end of this session, you will be able to :

- define database objects such as views and indexes
- create views and use views
- create and maintain indexes.

The views help you to simplify your complex queries on the tables. Also they increase the secrecy of data. To speed up the performance of queries, you should create indexes on one or more columns. The discussion in this session is based on the following relations:

Teacher(t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
Class(class_no, t_no, room_no)
Payscale(Min_limit, Max_limit, grade)

2.3.1 Creating a view

Views are actually virtual tables that are derived from base tables at runtime. A view after creation can be used as any other table but a view does not store data and it derives data only when it is called. The CREATE VIEW command is used to create views on the tables.

Syntax: CREATE VIEW [or replace] [force | noforce] AS
Select statement
[with check option [constraint]]
[with read only]

or replace: Replaces the existing view with the new one.
force: Allows the view to be created even if the tables don't exist.
No force: Allows the view to be created only when the tables exist.
With check option: Allows the DML operations only on the rows accessible to the view.
With read only: Does not allow any DML operation using view. For example:

Example 1: Create a view that displays teacher number, the names of teachers along with salary, job title, age and grade.

```
CREATE VIEW  teacher_details
        AS  SELECT t_no, f_name||' '||l_name as name, title, salary,
              trunc(months_between(sysdate,birthdate)/12,0) age, grade
    FROM teacher , payscale
    WHERE salary BETWEEN min_limit AND max_limit;
```

Let us make a few queries on this view.

Query (a): Display teacher number, their names, age and grade of all PGT teachers.

```
SELECT t_no, name, age, grade
FROM teacher_details
WHERE  UPPER (title) = 'PGT';
```

Query (b): Display details of all the teachers who are more than 40 years old.

```
SELECT t_no, name, salary, title, age
FROM teacher_details
WHERE age > 40;
```

To remove a view, DROP statement is used. For example:

```
DROP VIEW teacher_details;
```

2.3.2 Indexes

Index is the Oracle object that reduces the record-retrieval time. An index contains the Rowid and the values of the target columns. A rowid is a pointer that determines the location of the record. Indexes can be based on one or more table columns. Two types of indexes can be created:

1. Unique indexes - These indexes are created automatically whenever a PRIMARY KEY or UNIQUE constraint is defined.
2. Non-unique indexes - These indexes are created explicitly by the user on non-unique keys.

Syntax:

```
CREATE INDEX index_name
ON  tablename (column1 , column2 ,……);
```

Example 2: Create an index on the relation teacher on the job title for fast access.

```
CREATE INDEX t_title_index
ON teacher(title);
```

To remove an index, drop statement is used. For example,

```
DROP INDEX t_title_index;
```

2.3.3 Exercise 2

1) Please attempt the following problems for the teacher, class and payscale relations given in this section.

   (a) Create a view named 'supervisor_details' that stores the names and numbers of all the supervisors.

   (b) Create a non-unique index on the foreign key column of the 'class' table.

   (c) Modify the view created in (a) and add details like salary, job title, joining date, birth date etc. of all supervisors.

   (d) Using the view created in (c) display details of all supervisors who have worked for more than 15 years.

   (e) Create a view that stores the details of all the teachers who are TGT and earning more than Rs.12000/-.

   (f) Drop the view created in (e).

(g) Create a non-unique index on the names of teachers in the 'teachers' table.

(h) Drop the index created in (b).

(i) Create a view named 'teacher_info' that only allows users to view the teacher id, name, salary and grade of the teachers. It should not allow any user to change/update any information.

(j) Create a view that displays details of all teachers who are in grade 'B' and are more than 40 years old.

2) Design suitable views for the University database system (Exercise 1, question 2). For example, you can create a view for the faculty giving him/her details of his/her students. Create at least 5 suitable queries on each of the views created by you. Also create indexes for the University database system. For example, you can create an index on student name, thus, allowing faster access to student information in the order of student names. You must explain how the indexes created by you would enhance the performance of the database system. Implement the above in a commercial DBMS using SQL.

3) Design suitable views and indexes for the Bank database system (Exercise 1, question 3). Create at least 5 suitable queries on each of the views. Explain how the indexes created by you would enhance the performance of the database system. Implement the above in a commercial DBMS using SQL.

2.4 SESSION 3: PL/SQL – CONTROL LOOPS AND PROCEDURES

Objectives

At the end of this session, you should be able to:

- use procedural constructs like LOOP
- define and use FOR loop, While loop
- define and use procedures.

PL/SQL extends SQL by adding control loops found in procedural languages, resulting in a structural language that is more powerful than SQL. We will discuss the use of PL/SQL in this session.

2.4.1 Control loops

Loops are the constructs that help to repeat the statement groups on certain conditions. There are three kinds of looping constructs:

- LOOP
- WHILE
- FOR

LOOP Structure

It is the basic loop structure. It is used to repeat statement groups without providing a condition to come out of loop. This structure begins with the keyword LOOP and ends with the keyword END LOOP. To come out of the loop, EXIT statement is used to break the most current loop and the control is passed to the next statement after the END LOOP statement. This basic loop structure allows execution of statement group at least once even if the condition is already true when entering the loop. Let us explain this with the help of examples.

Example 1: A loop that will execute 100 times may be:

```
DECLARE
  i NUMBER := 0;
```

```
BEGIN
  LOOP
    i := i + 1;
    IF i = 100 THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

Example 2: Another way of writing example 1

```
DECLARE
  I   NUMBER := 0;
BEGIN
  LOOP
    I := I + 1;
    EXIT WHEN I = 100;
  END LOOP;
END;
```

Example 3: This example shows insertion of values in 'sample' table.

```
DECLARE
   I NUMBER := 1;
BEGIN
   LOOP
      INSERT INTO sample VALUES(I, I);
      I := I * 5;
      EXIT WHEN I >100;
   END LOOP;
END;
```

FOR Loops

This loop structure is an extension of the basic LOOP structure along with an additional control statement tagged at the front of the LOOP keyword. By default the loop increments the control variable starting from the lower index and going up to the higher index value. The REVERSE clause is used to make the loop decrement the control variable from highest to lowest index value. For example:

Example 4: Using for loop for a counter from 1 to 100.

```
DECLARE
   a NUMBER := 1;
BEGIN
   FOR counter IN 1 .. 100
   LOOP
      a := a + (counter * 5);
      EXIT WHEN a > 10000;
   END LOOP;
END;
```

Example 5: In this example, the 'counter' will start from 100 and reach 1.

```
DECLARE
   a NUMBER := 1;
   lower_index := 1;
   upper_index := 100
BEGIN
    FOR counter IN REVERSE lower_index .. upper_index
    LOOP
```

```
        a := a + (counter * 5);
    END LOOP;
END;
```

WHILE Loops

This loop structure is also an extension of basic LOOP and the statements are iterated based upon a condition test. If condition is *true* then the loop is executed otherwise not.

Example 6: A quantity can be issued if the total amount is below a sanctioned amount.

```
DECLARE
  qty NUMBER := 1;
 sanctioned_amt  NUMBER := 1000;
 unit_price NUMBER := 10;
 tot_amt NUMBER := 0;
BEGIN
    WHILE tot_amt < sanctioned_amt
    LOOP
        tot_amt := unit_price * qty;
        qty := qty + 1;
    END LOOP;
END;
```

2.4.2 Procedures

PL/SQL procedure is a named program block i.e., logically grouped set of SQL and PL/SQL statements that perform a specific task. They can be invoked or called by any PL/SQL block. The syntax is:

```
CREATE OR REPLACE PROCEDURE name
(argument {IN, OUT, INOUT} data type, ……..)
IS
    Variables declarations;
BEGIN
    PL/SQL subprogram body
EXCEPTION
    Exception PL/SQL block
END procedure name;
```

Replace: if procedure is already created then replace it.
IN : Indicates that the parameter will accept a value from the user.
OUT : Indicates that the parameter will return a value to the user.
INOUT : Indicates that the parameter will either accept from or return a value to the user.

Example 7: Following procedure searches the name of a teacher in the relation *teacher*

```
CREATE OR REPLACE PROCEDURE search_teacher
        (o_t_no IN NUMBER,
         o_f_name OUT VARCHAR2,
         o_l_name OUT VARCHAR2)
IS
BEGIN
        SELECT f_name, l_name
        FROM teacher
        WHERE t_no = o_t_no;
END search_teacher;

run;  /* it is used to create the procedure */
```

To call this procedure:

```
DECLARE
o_f_name  teacher.f_name%TYPE;
o_l_name teacher.l_name%TYPE;
BEGIN
    search_teacher( 113, o_f_name, o_l_name);
    DBMS_OUTPUT.PUT_LINE('Employee : 113');
    DBMS_OUTPUT.PUT_LINE('Name :  '|| o_f_name || '  ' || o_l_name);
END;
```

Example 8: To demonstrate use of INOUT parameter

```
CREATE PROCEDURE bonus_calc
          (o_t_no IN INTEGER,
           bonus INOUT INTEGER)
IS
          join_date DATE;
BEGIN
          SELECT salary * 0.20, joiningdate INTO bonus, join_date
           FROM teacher
          WHERE t_no = o_t_no;
          IF MONTHS_BETWEEN(SYSDATE, join_date) > 36 THEN
               bonus := bonus + 1000;
          END IF;
END;
```

Important Notes:

1) Unlike the type specifier in a PL/SQL variable declaration, the type specifier in a parameter declaration must be unconstrained i.e.`CHAR` or `VARCHAR2` should be used instead of `CHAR(5)` or `VARCHAR2` (20).

2) The name of the procedure after the END is optional.

3) A constant or a literal argument should not be passed in for an OUT/INOUT parameter.

2.4.3 Exercise 3

1) Please perform the following using the following relations:

Teacher(t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
Class(class_no, t_no, room_no)
Payscale(Min_limit, Max_limit, grade)

(a) Calculate the bonus amount to be given to a teacher depending on the following conditions:

I. if salary > 10000 then bonus is 10% of the salary.
II. if salary is between 10000 and 20000 then bonus is 20% of the salary.
III. if salary is between 20000 and 25000 then bonus is 25% of the salary.
IV. if salary exceeds 25000 then bonus is 30% of the salary.

(b) Using a simple LOOP structure, list the first 10 records of the 'teachers' table.

(c) Create a procedure that selects all teachers who get a salary of Rs.20, 000 and if less than 5 teachers are getting Rs.20, 000 then give an increment of 5%.

(d) Create a procedure that finds whether a teacher given by user exists or not and if not then display "teacher id not exists".
(e) Using FOR loop, display name and id of all those teachers who are more than 58 years old.
(f) Using while loop, display details of all those teachers who are in grade 'A'.
(g) Create a procedure that displays the names of all those teachers whose supervisor is 'Suman'.
(h) Calculate the tax to be paid by all teachers depending on following conditions:

I. if annual salary > 1,00,000 then no tax.
II. if annual salary is between 1,00,001 and 1,50,000 then tax is 20% of the annual salary.
III. if annual salary is between 1,50,001 and 2,50,000 then tax is 30% of the annual salary.
IV. if salary exceeds 2,50,000 then tax is 40% of the annual salary.

(i) Create a procedure that finds the names of all teachers with the job title 'PRT' and if the number of teachers returned is more than 10 than change the job title to 'TGT' for the top 3 'PRT' teachers based on their hiredate.

2) Identify the need of procedures for the University database system; for example, you can create a procedure that awards 2% grace marks for those students who have got 48% marks. Design at least 5 such procedures. Implement these procedures using an embedded SQL.

3) Implement at least five procedures for the Bank Database system using embedded SQL.

2.5 SESSION 4: CURSORS

Objectives

At the end of this session, you should be able to:

- define implicit and explicit cursors.
- perform open, fetch and close operations on the explicit cursors
- create cursor *FOR* loops.

A private work area for an SQL statement is called cursor. The cursors can be implicit or explicit. You can perform several major actions on the explicit cursors. Let us discuss them in more detail.

2.5.1 Cursors

When SQL statement is executed in the PL/SQL block, the Oracle Engine assigns a private work area for that statement. This work area is private to the SQL statement i.e., it stores the statement and the result after execution and is called a *cursor.* Cursors can either be created implicitly or explicitly.

1. Implicit cursors

When an SQL command is executed from within a PL/SQL block, PL/SQL creates an implicit cursor. PL/SQL provides some attributes that allow you to evaluate what happened when the implicit cursor was last used.

2. Explicit cursors

An explicit cursor is defined as a SELECT statement within PL/SQL block. The general syntax is:

```
CURSOR cursorname IS
    SELECT statement;
```

There are a number of actions performed on explicit cursors. They are: -

ACTION	DESCRIPTION	EXAMPLE
DECLARE	It defines the name and structure of the cursor using the SELECT statement.	CURSOR cur_sample IS SELECT * FROM teacher;
OPEN	It executes the query that populates the cursor with rows.	OPEN cur_sample;
FETCH	It loads the row currently referred by the cursor pointer into variables or record and moves the cursor pointer to the next row ready for the next fetch.	FETCH cur_sample INTO teacher_rec **OR** FETCH cur_sample1 INTO cur_t_no, cur_salary;
CLOSE	It releases the data within the cursor and closes it. The cursor can be reopened again.	CLOSE cur_sample;

The SQL cursor attributes that can be used with implicit or explicit cursors are:

ATTRIBUTE	DESCRIPTION
%ROWCOUNT	Returns the number of rows processed by an SQL statement.
%FOUND	Returns TRUE if at least one row was processed.
%NOTFOUND	Returns TRUE if no rows were processed.
%ISOPEN	Return TRUE if cursor is open or FALSE, if cursor has not been opened or has been closed. Only used with explicit cursors.

Example 1: Implicit cursor

```
DECLARE
      total_row_del NUMBER;
BEGIN
      DELETE * FROM teacher WHERE salary < 10000;
      total_row_del := SQL%ROWCOUNT;
END;
```

Example 2: Explicit Cursor

```
DECLARE
     c_t_no     teacher.t_no%TYPE;
     c_f_name  teacher.f_name%TYPE;
     c_l_name  teacher.l_name%TYPE;
     c_salary  teacher.salary%TYPE;
     CURSOR  c1 IS
          SELECT t_no,f_name, l_name, salary
          FROM teacher;
BEGIN
     OPEN c1;
     LOOP
          FETCH c1 INTO c_t_no, c_f_name, c_l_name, c_salary ;
          EXIT WHEN NOT c1%FOUND;
          UPDATE teacher SET salary = salary * 1.10 WHERE salary > 20000;
      END LOOP;
      CLOSE c1;
END;
```

Example 3: Explicit Cursor using record name in FOR loop

```
DECLARE
     CURSOR  c2 IS
          SELECT t_no,f_name, l_name, salary
          FROM teacher ;
```

```
    teacher_rec  c2%ROWTYPE;
BEGIN
    OPEN c2;
    FOR teacher_rec  IN c2
    LOOP
        IF teacher_rec.salary > 20000
            Teacher_rec.title = "SUPERVISOR";
        ENDIF;
    END LOOP;
    CLOSE c2;
END;
```

2.5.2 Exercise 4

1) Please perform the following using the following relations:

Teacher(t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
Class(class_no, t_no, room_no)
Payscale(Min_limit, Max_limit, grade)

(a) Create a host language block to declare a cursor for displaying teacher numbers and their names for all teachers having title 'PGT'.

(b) Create a host language block using a cursor to calculate bonus for teachers as 5% of their salary. Display on screen the teacher details along with the bonus given.

(c) Write a host language block to delete all the rows from the 'teacher' table where the salary is less than Rs.5000.

(d) Write a host language code to insert the supervisor information from 'teacher' table to another table called 'supervisor'. The new table should have only those records where the job title is 'supervisor'.

(e) Write a block in host language that deletes all the rows from 'teacher' table if the teacher was hired for more than 10 years.

(f) Write a block in host language using cursor that displays the names of all teachers who will attain the age of 60 years in the current year.

(g) Write a block in host language using cursors that display teacher details along with the tax to be paid by that teacher. The tax is calculated depending on following conditions:

I. if annual salary < 1,00,000 then no tax.

II. if annual salary is between 1,00,001 and 1,50,000 then tax is 20% of the annual salary.

III. if annual salary is between 1,50,001 and 2,50,000 then tax is 30% of the annual salary.

IV. if salary exceeds 2,50,000 then tax is 40% of the annual salary.

(h) Write a block in host language that displays the details of all those teachers who have reached maximum limit of their grade.

2) Write at least four embedded SQL blocks having cursors for the University database system; for example, you can create a cursor to update the examination marks of a student that are given in a list to the student's database. Implement these procedures using an embedded SQL.

3) Implement at least five embedded SQL blocks having cursors for the Bank Database system.

2.6 SESSION 5: ERROR HANDLING AND TRANSACTION MANAGEMENT

Objectives

At the end of this session you should be able to:

- perform basic error handling
- use commit, rollback and save point specifications.

2.6.1 Error Handling

Exceptions are identifiers that are raised during the execution of a PL/SQL block to terminate its action in case of a problem, error or abnormal condition. A block is always terminated when PL/SQL raises an exception but we can also capture exceptions by defining our own error handlers and perform some final actions before quitting the block.

There are two classes of exceptions. These are:

1) Predefined: It basically refers to Oracle predefined errors which are associated with specific error codes. For example: CURSOR_ALREADY_OPEN, NO_DATA_FOUND.

2) User-defined: These are defined by the user and raised when specifically requested within a block. If an error occurs within a block PL/SQL passes control to the EXCEPTION section of the block. If no EXCEPTION section exists in the block or the EXCEPTION section cannot handle the error then the block is terminated with an unhandled exception. Exceptions execute through nested blocks until an exception handler is found that can handle the error. If no exception handler is found in any block the error is passed out to the host environment. Exceptions occur implicitly when either an Oracle error occurs or you explicitly raise an error using the RAISE statement.

Example 1: Predefined Exception

```
DECLARE
        C_id  teacher.t_no%TYPE;
        C_f_name teacher.f_name%TYPE;
        want_id NUMBER := 110;
BEGIN
         SELECT t_no, f_name INTO c_t_no, c_f_name from teacher
         WHERE t_no = want_id;
          DBMS_OUTPUT.PUTLINE ( "teacher : "|| c_t_no ||'  '||c_f_name)
EXCEPTION
          WHEN INVALID_NUMBER THEN
           DBMS_OUTPUT.PUTLINE(want_id || ' not a valid teacher id');
END;
```

Example 2: User Defined Exception

```
DECLARE
    C_title    teacher.title%TYPE;
    CURSOR  c_teacher  IS
       SELECT  t_no,f_name, l_name, title
       FROM teacher;
```

```
    C_teacher_row   c_teacher%ROWTYPE;
BEGIN
    C_title := 'PGT';
    FOR c_teacher_row IN c_teacher
    LOOP
        EXIT WHEN c_teacher%NOTFOUND;
        DECLARE
            emptytitle EXCEPTION;
        BEGIN
            IF c_teacher_row IS NULL THEN
                RAISE emptytitle;
            ELSE
                DBMS_OUTPUT.PUTLINE (c_teacher_row.f_name,
c_teacher_l_name, c_teacher_title);
            ENDIF;
            EXCEPTION
            WHERE emptytitle THEN
                DBMS_OUTPUT.PUTLINE(c_teacher_row.f_name);
        END;
    END LOOP;
    CLOSE c_teacher;
END;
```

2.6.2 Transaction Management

A transaction is a logical unit of work that is composed of one or more Data Manipulation Language (DML) or Data Definition Language (DDL) or Data Control Language (DCL) statements. A transaction starts when an SQL statement is executed and terminates when any of the following occurs:

- Any DDL or DML statement is executed.
- A COMMIT or ROLLBACK statement is given
- User exits SQL.

The transactions can be committed implicitly or explicitly. All DDL or DCL statements are implicitly committed.

There are three explicit transaction specifications:

1) COMMIT: It ends the current transaction and saves permanently all pending changes since last saved or COMMIT or ROLLBACK. The syntax is:

```
COMMIT;
```

2) SAVEPOINT: It marks and saves the current point of the transaction process. It is useful to perform partial rollbacks. The syntax is:

```
SAVEPOINT  name;
```

3) ROLLBACK: It ends the transaction but discards all pending changes since last commit or save point. The syntax is:

```
ROLLBACK  [ TO SAVEPOINT name];
```

Example:

```
SQL >  UPDATE teacher   SET   title = 'PGT' WHERE salary > 15000;
SQL>   COMMIT
                        /* Changes made by Update statement are saved */
SQL> INSERT INTO class(class_no, t_no, room_no) VALUES (10,127,226);
SQL> ROLLBACK;
```

```
                        /* values inserted into 'class' table are discarded */
SQL> UPDATE teacher   SET   title = 'SUPERVISOR' WHERE salary > 20000;
SQL> SAVEPOINT  update_done;
                        /* savepoint created */
SQL> DELETE FROM teacher;
SQL> ROLLBACK TO SAVEPOINT  update_done;
                        /* just delete statement is discarded */
```

2.6.3 Exercise 5

1) Write an embedded SQL block along with exceptions to select the name of the teacher with a given salary. If more than one row is returned than display more than one row retrieved. If no row is returned then display 'no teacher with this salary'.

2) Create a program that updates a record into the 'teacher' table. Trap and handle the exception if the teacher id is not available in the 'teacher' table.

3) Write a program with exceptions that displays the details of all those teachers who have reached maximum limit of their grade. If no row is retrieved then raise the exception "no teacher reached max limit of grade".

4) Insert at least 5 new rows in the 'teacher' table and then try to rollback last 3 rows inserted to the table. (Here, you are required to use save points).

5) Write a program with exceptions that displays the names of all teachers who will attain the age of 60 years in the current year. If no row is retrieved than display suitable exception.

6) Write a PL/SQL block that displays all the rows from 'teacher' table if the teacher was hired for more than 10 years and still a 'PRT'. If no result than display suitable message.

7) In all the embedded SQL program segments that you have created so far for the University and Bank database system, create suitable error handling features.

8) Experiment with DDL and DML commands along with COMMIT and ROLLBACK for the Teacher, University and Banking databases.

2.7 SESSION 6: TRIGGERS AND FUNCTIONS

Objectives

At the end of this section, you should be able to:

- define triggers and functions
- define and create statement and row triggers
- create functions with or without parameters.

For the following discussion, we will consider the following relations:
Teacher(t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
Class(class_no, t_no, room_no)
Payscale(Min_limit, Max_limit, grade)

2.7.1 Triggers

A trigger is PL/SQL block stored in the database. It is executed by an event that occurs to a database table. Triggers are implicitly invoked by DML commands like INSERT, UPDATE or DELETE. But you cannot use COMMIT, ROLLBACK and SAVEPOINT statements within trigger blocks. Triggers can be called BEFORE or AFTER the events. Triggers can be of two types:

1) STATEMENT type - STATEMENT triggers fire BEFORE or AFTER the execution of the statement that caused the trigger to fire.
2) ROW type - ROW triggers fire BEFORE or AFTER any affected row is processed.

Example 1: A STATEMENT trigger for not allowing work on weekends.

```
CREATE OR REPLACE TRIGGER not_working_hour
    BEFORE DELETE OR INSERT OR UPDATE ON teacher
BEGIN
    IF (TO_CHAR(SYSDATE, 'DY') IN ('SAT', 'SUN'))
    THEN
    DBMS_OUTPUT.PUTLINE ('Sorry! you cannot delete/update/insert on
                                        weekends');
     END IF;
END;
```

Example 2: A ROW trigger for supplying new teacher id and system date as the joining date.

```
CREATE OR REPLACE TRIGGER new_teacher _id
     AFTER INSERT ON  teacher
FOR EACH ROW
DECLARE
     o_t_no  teacher.t_no%TYPE;
    o_joiningdate teacher.joiningdate%TYPE;
BEGIN
    SELECT  t_no_sequence.nextval
        INTO  o_t_no
        FROM  dual;
    :NEW.t_no := o_t_no;
    :NEW.joiningdate := SYSDATE;
END;
```

2.7.2 Functions

Functions are similar to procedures but in function definitions you must explicitly return a value to the calling block via the RETURN statement. A function can have zero, one or more parameters.

Example 3: Find the grade of a teacher (one parameter function).

```
CREATE OR REPLACE FUNCTION get_grade (o_t_no  IN NUMBER)
IS o_grade VARCHAR2(20);
BEGIN
  SELECT grade  INTO o_grade FROM Payscale, teacher
     WHERE t_no = o_t_no  AND salary between min_limit AND max_limit;
  RETURN (o_grade);
END get_grade;
```

This function can be called from PL/SQL block as:

```
o_teacher_grade := get_grade(o_teacher_no);
```

Example 4: Function without parameter

```
CREATE OR REPLACE FUNCTION welcome_note
          RETURN  VARCHAR2
IS
BEGIN
    RETURN  'Good Morning!';
END welcome_note;
```

This function can be called from PL/SQL block as:

```
a : =  welcome_note;
DBMS_OUPUT.PUTLINE (a);
```

To drop a function:

```
DROP FUNCTION <function_name>;
```

2.7.3 Exercise 6

1) Write a trigger that is fired before the DML statement's execution on the TEACHER table. The trigger checks the school timings based on SYSDATE. Beyond the School working hours the trigger raise an exception, which does not allow any work to be happened.
2) Write a trigger that is fired before an UPDATE statement is executed for the teacher table. The trigger should write the name of teacher, user name and system date in an already created table called UPDATE_TABLE.
3) Write a trigger that is fired before any row is inserted in the 'teacher' table.
4) Write a function and pass a job title to it. If the TEACHER table does not contain any row corresponding to that title then return false otherwise true.
5) Write a trigger that verifies the joining date when a new row is inserted in the 'teacher' table. Joining date should be greater or equal to current date.
6) Write a function that gets the teacher id as parameter and returns the class number associated with that teacher. If the teacher is not a class teacher then give suitable message.
7) Write a function and pass a teacher id to it. If the TEACHER table does not contain that id then return false otherwise true.
8) Write a function that takes teacher id as parameter and returns back the name and joining date of the teacher.
9) Write appropriate triggers and functions for the University and Bank database systems.

2.8 SESSION 7: CREATING OBJECT TYPES/TABLES

Objectives

At the end of this session, you should be able to:

- define objects and object type.
- create and use object types.
- create object methods

2.8.1 Objects

An object in ORACLE is a reusable component representing real-world entities. An object can be defined using user-defined data type called an object type that is further used to define object columns in the tables. An object contains a name, attribute and methods. The methods can be used to perform data manipulation on the object tables.

Defining Types

Oracle defines the types similar to the types of SQL.

Syntax:

```
CREATE [ OR REPLACE ] TYPE <name> AS OBJECT
                                    <attribute name> datatype, ..
MEMBER PROCEDURE | FUNCTION <procedure or function spec>,...,
```

```
        [ MAP | ORDER MEMBER FUNCTION <comparison function
        spec>,
              ... ]
        [ PRAGMA RESTRICT_REFERENCES (<what to restrict>,
        restrictions) ]
        );
/
```

Note: The slash at the end processes the type definition in ORACLE.

In the type definition as above:

OR REPLACE – Replace if definition already exists

Name - Oracle identifier
attribute name - A legal PL/SQL identifier for the attribute.

Data type - Any legal Oracle data type *except* LONG, LONG RAW, NCHAR, NCLOB, NVARCHAR2, ROWID, BINARY_INTEGER, BOOLEAN, PLS_INTEGER, RECORD, REF CURSOR, %TYPE, %ROWTYPE, or types that exist only within packages.

Comparison function - Defines a function that allows comparison of object values.

What to restrict - This is either the name of the function or procedure, or the keyword DEFAULT. Using DEFAULT tells Oracle that *all* member functions and procedures in the object type will have the designated restrictions, without having to list each one in its own RESTRICT_REFERENCES pragma.

Restrictions - One or more of the following: RNDS, WNDS, RNPS, and WNPS.

FORCE++ -Tells Oracle that you want to drop a type even if there are other objects with dependencies on it. Even if you use FORCE, you can only drop a type if it has not been implemented in a table; you must first drop the table(s) before dropping the type.

Example 1: Declaring a point type consisting of two numbers:

```
CREATE TYPE pointtype AS OBJECT (
   a NUMBER,
   b NUMBER
);
/
```

This object type can be used like any other type in further declarations of object-types or table-types.

Example 2: Defining a line type by using previously created object type.

```
CREATE TYPE linetype AS OBJECT (
   x  pointtype,
   y  pointtype
);
/
```

These objects can further be used to create a table that is a set of lines with ``line ID's'' a

Example 3: Create object table

```
CREATE TABLE lines (
   line_id  INT,
   line   linetype
);
```

Dropping Types

To remove a type, we use a DROP statement as shown below:

```
DROP TYPE linetype;
```

Note: Before dropping a type, drop all tables and other types that use this type.

Constructing Object Values

Oracle provides built-in constructors for values of a declared type with the name of the type.

Example 4 : Inserting a new row in the Object table 'line' for a line from co-ordinates (0,0) to (4,5).

```
INSERT INTO lines
  VALUES (12, linetype(
                    point type(0.0, 0.0),
                    point type(4.0, 5.0)
                )
              );
```

Declaring and Defining Methods

The object methods can be declared by `MEMBER FUNCTION` or `MEMBER PROCEDURE` in the `CREATE TYPE` statement.

Example 5:

```
CREATE TYPE linetype AS OBJECT (
  x  pointtype,
  y  pointtype,
  MEMBER FUNCTION length(scale IN NUMBER) RETURN NUMBER,
       PRAGMA RESTRICT_REFERENCES (length, WNDS)
);
/
```

Note : The 'pragma' means the `length` function will not modify the database (WNDS means 'write no database state'). It is necessary when you want to use `length` in queries.

Queries on Object tables

The values of the components of an object are accessed with the dot notation. Also, to access fields of an object, you need to give alias of the table name.

Example 6: Find the length of all the lines for scale factor 3.

```
SELECT line_id, line_alias.line.length(3.0)
 FROM lines line_alias;
```

Note: 'line_alias' is table alias.

2.8.2 Exercise 7

1) Assuming that in the teacher relation an attribute of object type called dependent is added. Make dependent a type that may consist of only one dependent. Add few records in the tables and output them using a query on teacher name. Find if you can search on a dependent name.

2) Create at least two object types in both the University and Bank database systems. Use these object types in few relations and enter some data into these relations. Query these databases.

2.9 SESSION 8: NESTED TABLES

Objectives

At the end of this session, you should be able to:

- define and create nested tables
- initialize nested tables.

2.9.1 Nested Table

Nested table is defined as table within another table. Nested tables are single-dimensional arrays of elements that are of same type such as a table of numbers or characters. In nested tables, each row can have multiple rows itself.

Example:

```
CREATE  TYPE new_table  AS   OBJECT
(subject_name  VARCHAR2(25),
credit_hours NUMBER(5));

CREATE TYPE  new_type AS TABLE OF new_table;
```

This will create an object-type table named 'new table'.

Example: To use this nested object table.

```
CREATE TABLE student_credits
(rollno  NUMBER(5),
s_name VARCHAR2(25),
subject_credits  NEW_TYPE)
NESTED TABLE subject_credits STORE AS new_type_table;
```

Example: To insert values in the above created table,

```
INSERT INTO student_credits
VALUES (100, 'suman' , new_table ( new_type ('english' , 30),
                                   new_table ( new_type('hindi', 35)));
```

To query from a nested table Oracle provides a new function THE. Function. THE is used to select data from a field in the nested table for which we must have to flatten the table first.

Example:

```
SELECT  s.credit_hours FROM
   THE (SELECT subjects_credit FROM student_credits
            WHERE s_name = 'suman')  s
WHERE s.subject_name = 'english';
```

The nested tables can also be created as

```
CREATE  TYPE new_table  AS   OBJECT
my_table  new_table;
```

This definition creates un-initialised objects. Generally, for these types of un-initialized object type tables, Oracle generates an error that will cause an exception to be raised. These tables can be initialized by either initializing the whole table at declaration time or by initializing partially at declaration time and then later using EXTEND method to define new or extra elements.

Example 1: Initializing at declaration time.

```
DECLARE
   TYPE new_table IS TABLE OF NUMBER ;
   my_table new_table := new_table (2);
BEGIN
   my_table (1)  := 13;
   my_table (2)  := 15;
```

```
    DBMS_OUTPUT.PUT_LINE('my_table(1) is '||my_table(1));
    DBMS_OUTPUT.PUT_LINE('my_table(2) is '||my_table(2));
 END;
/
```

The table can also be initialized as:

```
my_table new_table := new_table (5,9,7,11,4,10);
```

This will create a new table with 6 elements.

Example 2: Partial initialisation and then using EXTEND

```
 DECLARE
   TYPE new_table IS TABLE OF NUMBER ;
   my_table new_table ;
BEGIN
    my_table := new_table();
   my_table.EXTEND(2);
   my_table(1) := 13;
   DBMS_OUTPUT.PUT_LINE('my_table(1) is '||my_table(1));
   DBMS_OUTPUT.PUT_LINE('my_table(2) is '||my_table(2));
END;
/
```

Note: You do not have a control over where these elements are created because Oracle automatically appends them to the existing table .

Following are the some of the functions that can be used with nested tables.

Functions	Description
COUNT	returns the number of elements in the collection
DELETE	deletes one or more elements in the collection - an optional start and end point specify which elements are to be deleted
EXISTS(n)	determines if the specified element has been created and not deleted, returns TRUE if the element exists, FALSE if not
FIRST	returns the subscript of the first element in the nested table
LAST	returns the subscript of the last element in the nested table
PRIOR	returns the subscript of the previous element in the nested table
NEXT	returns the subscript of the next element in the nested table

2.9.2 Exercise 8

1) Add a nested table in the teacher relation. Do some queries using nested tables?

2) Create at least two nested tables for both the University and Bank database systems. Use these tables and enter some data into these relations. Query these databases.

2.10 SESSION 9: STORING BLOBS

Objectives

At the end of this session, you should be able to:

- define LOBs

- types of LOBs – BLOB, CLOB, NCLOB, BFILE
- use blob data types in tables.

2.10.1 BLOBS (Large objects)

LOBs are data types that are capable of storing large volumes of unstructured data like images, sound clips, large texts etc. An LOB data can hold data ranging from 8 terabytes to 128 terabytes depending on how the database is configured. The use of LOBs enables you to access and manipulate data efficiently in your application. LOB data types allow random access to data. The LOB data types now available are BLOB, CLOB, NCLOB, and BFILE.

1) **CLOB(character large object)** stores large blocks of single-byte character data. Generally, used for large strings or documents that use the database character set exclusively.

2) **BLOB(binary large object)** stores large binary objects inside or outside row. Typically used for multimedia data such as images, audio and video.

3) **NCLOB (National Character Set Large Object)** stores string data in National character set format. Used for large strings or documents in the National character set.

4) **BFILE (External Binary File)** A binary file is stored outside the database in the host operating system file system but it can be accessible from database tables. This type of LOB can be accessed from your application on a **read-only** basis. Generally, BFILEs are used to store static data, such as image data, that does not need to be manipulated in applications.

LOB stands for Locator Object, which indicates that it is a pointer to that place where the actual data is stored. This locator will only occupy some bytes in the row.

Example 1:

```
CREATE TABLE message (
   msg_id  NUMBER(8) NOT NULL PRIMARY KEY,
      email_add       VARCHAR(200),
      name           VARCHAR (200),
      message         CLOB,
      posting_time    DATE,
      sort_key        VARCHAR (600));
```

LOBs are accessed via a LOB or BFILE "locator". The techniques you use when accessing a cell in a LOB column differ depending on the state of the given cell. A cell in a LOB Column can be in one of the following states:

- **NULL**: The table cell is created, but the cell holds no locator or value.
- **Empty:** A LOB instance with a locator exists in the cell, but it has no value. The length of the LOB is zero.
- **Populated:** A LOB instance with a locator and a value exists in the cell.

Use built-in functions EMPTY_CLOB() for CLOBs and NCLOBs, EMPTY_BLOB() for BLOBs, and BFILENAME() method to initialize a BFILE column to point to an OS file.

Example 2: Use of BLOB s

```
DECLARE
 Image10         BLOB;
 image_number    INTEGER := 101;
```

```
BEGIN
  SELECT item_blob INTO image10 FROM lob_table10
     WHERE key_value = image_number;
  DBMS_OUTPUT.PUT_LINE('Image size
is:'||DBMS_LOB.GETLENGTH(image10));
    ------
    ------
    ------
    ------
END;
```

Example 3: Inserting lob values in tables.

```
INSERT INTO message1
VALUES (101, 1, EMPTY_BLOB(),'my Oracle', EMPTY_CLOB(),NULL,
     BFILENAME('dir_object', 'my_image'), NULL);
```

Example 4: Selecting blob values from tables.

```
SELECT COUNT (*) FROM message1 WHERE image_graphic IS NOT NULL;
 SELECT COUNT (*) FROM message1 WHERE image_graphic IS NULL;
```

2.10.2 Exercise 9

1) Identify the use of large object types in the teacher's table. Do some queries using these objects.

2) Create at least two large objects for both the University and Bank database systems. Enter some data into these relations. Query these databases.

2.11 SESSION 10: USER MANAGEMENT AND OBJECT PRIVILEGES

Objectives

At the end of this session, you should be able to:

- define and create user accounts
- define the types of privileges
- give and withdraw the privileges.

2.11.1 User Management

Before the database is used by a number of users, they should be given privileges on their user groups. By default, Oracle has two accounts: SYS and SYSTEM. The SYSTEM account is used to give privileges to the users and to create new user accounts.

The user account can be created by the CREATE USER command.

Example 1: Create a user account 'manager' with password 'pass'.

```
CREATE USER manager
   IDENTIFIED BY pass;
```

These user accounts are of use when the privileges are given to them. There are two types of privileges:

1) System privilege: This kind of privilege allows a user to use DDL Commands like CREATE TABLE, DROP INDEX etc.

2) Object privilege: This kind of privilege allows a user to use DML Commands like UPDATE, INSERT etc.

To give and take back the privileges to user accounts, we use GRANT and REVOKE command respectively.

Example 2: Give CREATE and DROP privileges related to table/view to the user account 'manager'.

```
GRANT  CREATE TABLE, DROP TABLE, CREATE VIEW, DROP VIEW
                                        TO  manager;
```

Example 3: Withdraw DROP privilege related to table/view given to the user account 'manager'.

```
REVOKE DROP TABLE, DROP VIEW FROM manager;
```

2.11.2 Exercise 10

1) Create a user account "class" and give privileges related to table/view creation, deletion, updating and dropping.
2) Create a student account and give permission to this account for only viewing the information on the relation *Class (class_no, t_no, room_no.*
3) Create at least 3 to 4 different types of users for each of the database systems: University and Bank. Design suitable access privileges for the users. Grant these permissions to the users.
4) Consider when you have a large number of students and teachers in the University databases that have different access rights. Is there any mechanism in DBMS that allows defining an account type to some specific role? If yes, then define such types for the University database system.
5) Define different types of users for the Bank database and provide them suitable access rights.

2.12 SUMMARY

This section has tried to broadly cover about ten different aspects of a commercial DBMS. We have covered practical sessions relating to sub-queries and join based queries, creating views, indexes and performing queries on them, embedded SQL with or without the use of cursors, error handling during the embedded SQL operations and using COMMIT and ROLLBACK commands used for transactions. We have also done sessions on triggers and functions, creating types, nested tables, BLOBs and user management.

You should try to solve not only the problems given in these exercises, but also many more problems during your practical sessions. You should consult online help/reference manuals of the commercial DBMS you are using while solving the problems. Practice is the key for achieving the basic objectives of the sessions.

Finally, please keep in mind the fact that the basic objective of this section is NOT to make you an expert DBA, but it is to give you a feel of many different types of things that you may be able to do with a DBMS. You should look forward to applying your theoretical knowledge of MCS-023 and MCS-043 along with the skills obtained during the practical sessions to create solutions with respect to the needs of various business organisations.