MANIPAL INSTITUTE OF TECHNOLOGY

MANIPAL

(A Constituent Institution of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that Ms./Mr. ……...…… Reg. No.: …..…… Section: …… Roll No.: ……... has satisfactorily completed the lab exercises prescribed for Database Systems Lab [CSE 2241] of Second Year B.Tech.(Computer Science and Engineering) Degree at MIT, Manipal, in the academic year 2024-2025.

Date: …….....

Signature
Faculty in Charge

CONTENTS

Course Objectives

- To understand database creation and explore the database query language
- Learn to develop stored procedures, functions and packages.
- To develop an application software with host language interface.

Course Outcomes

At the end of this course, students will have the

1. Write queries for design and manipulation of database tables
2. Identify the effective use of stored procedures, functions and packages.
3. Design and develop applications

Evaluation plan

- Internal assessment: 60 marks
  - ✓ Continuous evaluation component (for each week): 4 marks.
  - ✓ The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce.
  - ✓ Total marks for the 10 experiments are 40 Marks.
  - ✓ Mini project is for 20 Marks.

- End semester assessment: 40 marks
- Total (Internal assessment + End semester assessment): 100 marks

INSTRUCTIONS TO THE STUDENTS

Pre-Lab Session Instructions

- Students should carry the Lab Manual Book and the required stationery to every lab session.
- Be in time and follow the institution dress code.
- Must Sign in the log register provided.
- Make sure to occupy the allotted seat and answer the attendance.
- Adhere to the rules and maintain the decorum.

In-Lab Session Instructions

- Follow the instructions on the allotted exercises.
- Show the program and results to the lab Teacher on completion of experiments.
- You must have your lab notebook signed by your lab Teacher before you leave lab each day. Any pages not signed on the day the experiment was performed will adversely affect your lab notebook grade.
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
    - Programs should perform input validation (Data type, range error, etc.).
    - Use meaningful names for variables and functions.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.

- The exercises for each week are divided under two sets:
  - Lab exercises - to be completed during lab hours.
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill.
- In case a student misses a lab session, he/she may ensure that the experiment is completed during the extra lab session (of other batch) with the permission of the HOD.
- Questions for lab tests and examination are not necessarily limited to the questions in the manual but may involve some variations and/or combinations of the questions.
- A sample note preparation is given as a model for observation.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

Sample Lab Observation Note Preparation

LAB NO.: **Date:**

Title: STRUCTURED QUERY LANGUAGE

1. Implement the Bank Database and execute the given queries/updates

Bank Database Schema:

- ACCOUNT(ACCOUNT_NUMBER, BRANCH_NAME, BALANCE)
- BRANCH (BRANCH_NAME, BRANCH_CITY, ASSETS)
- CUSTOMER (CUSTOMER_NAME CUSTOMER_STREET, CUSTOMER_CITY)
- LOAN (LOAN_NUMBER, BRANCH_NAME, AMOUNT)
- DEPOSITOR(CUSTOMER_NAME, ACCOUNT_NUMBER)
- BORROWER(CUSTOMER_NAME, LOAN_NUMBER)

- **Creating Tables**

```
CREATE TABLE BRANCH
(BRANCH_NAME VARCHAR (15) PRIMARY KEY,
BRANCH_CITY VARCAHAR (20),
        ASSETS NUMBER (10));

CREATE TABLE ACCOUNT
(ACCOUNT_NUMBER NUMBER (10) PRIMARY KEY,
BRANCH_NAME VARCHAR (15) REFERENCES BRANCH,
        BALANCE NUMBER (8));

CREATE TABLE CUSTOMER
(CUSTOMER_NAME VARCHAR (20) PRIMARY KEY,
       CUSTOMER_STREETVARCHAR (15),
       CUSTOMER_CITY VARCHAR (10));

CREATE TABLE LOAN
```

```
(LOAN_NUMBER NUMBER (10) PRIMARY KEY,

BRANCH_NAME VARCHAR (15)REFERENCES BRANCH,
AMOUNT NUMBER (10))

CREATE TABLE DEPOSITOR
(CUSTOMER_NAME VARCHAR (2) REFERENCES CUSTOMER,
                ACCOUNT_NUMBER NUMBER (10) REFERENCES ACCOUNT,
PRIMARY KEY (CUSTOMER_NAME, ACCOUNT_NUMBER));

CREATE TABLE BORROWER
      (CUSTOMER_NAME VARCHAR(2) REFERENCES CUSTOMER,
       LOAN_NUMBER  NUMBER(10) REFERENCES LOAN,
      PRIMARY KEY(CUSTOMER_NAME,LOAN_NUMBER));
```

Queries/Update on Bank Database

(Questions followed by SQL statements)

Retrieving records from a table:

1. list the information of all account holders (name and account number).
   Select * from depositor.
2. List all branch names and their assets
   SELECT BRANCH_NAME, ASSETS FROM BRANCH;

3. List all accounts of Brooklyn branch
   SELECT * FROM ACCOUNT WHERE BRANCH_NAME= 'BROOKLYN';

4. List all loans with amount > 1000.
   SELECT * FROM LOAN WHERE AMOUNT>1000;

Updating records from a table:

4. Change the assets of Perryridge branch to 340000000.

```
UPDATE BRANCH SET ASSETS=340000000
WHERE BRANCH_NAME='Perryridge';
```

LAB NO.: 1 **Date:**

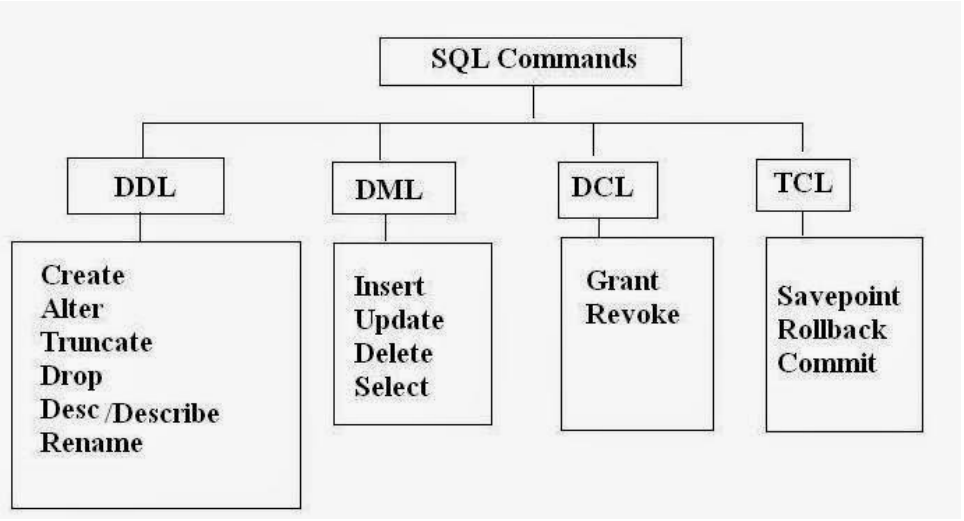
INTRODUCTION TO SQL

Objectives:

In this lab, student will be able to:

- Understand the working of DDL/DML commands.

SQL Statements can be categorized as -



DDL (Data definition language): Are used to define database structure or schema.

DML (Data manipulation language): Are used to change or alter data with the database or schema.

DCL (Data Control Language): Are used to control access or privileges.

TCL (Transaction Control Language): Are used to manage transactions in the database.

Basic Data Types used in SQL:

- CHARACTER [(length)] or CHAR [(length)],
- VARCHAR (length)
- BOOLEAN
- SMALLINT
- INTEGER or INT
- DECIMAL [(p[,s])] or DEC [(p[,s])]
- NUMERIC [(p[,s])]

- REAL
- FLOAT(p)
- DOUBLE PRECISION
- DATE
- TIME
- TIMESTAMP
- CLOB [(length)] or CHARACTER LARGE OBJECT [(length)] or CHAR LARGE OBJECT [(length)]
- BLOB [(length)] or BINARY LARGE OBJECT [(length)]

DDL COMMANDS:

1. CREATION OF TABLE:

SYNTAX:

create table<tablename>(column_name1 datatype(<size>),column_name2 datatype(<size>) ...);

EXAMPLE:

SQL>
create table STUDENT (
reg_no number (5),
stu_name varchar(20),
stu_age number(5),
stu_ dob date,
subject1_marks number (4,2),
subject2_marks number(4,2),
subject3_marks number(4,1));
SQL>insert into STUDENT values (101, 'AAA',16, '03-jul-88',80,90,98);

2. Modifying the structure of tables

a) Add new columns

Syntax:

Alter table <tablename>add (<new col><datatype (size),<newcol>datatype(size));

Ex: Add a new column 'Gender' to student table.

alter table student add(Gender char (5));

3. Dropping a column from a table

Syntax: Alter table <tablename> drop column <col>;

Ex: To drop a column 'Gender' from student table.

Alter table student drop column Gender;

4. Modifying existing columns

Syntax: Alter table <tablename> **modify** (<col><newdatatype>(<newsize>));

Ex: To modify the datatype of stu_age

Alter table student **modify** (stu_age number(3));

5. Renaming the tables

Syntax:

Rename <oldtable> to <new table>;

Ex: Rename student to students;

6. Truncate the table

Syntax: Trunc table <tablename>;

Ex: Trunc table students;

7. Delete the table structure

Syntax: Drop table <tablename>;

Ex: drop table student;

DML commands (ADDITIONAL EXAMPLES):

1. Selecting the information from table(s)

Syntax: Select col1,col2,col3,……, coln from <table_name> where < condition >

Ex:

a) List all the students
   Select * from student;
b) List age of all students with column aliased as 'student_age' rather stu_age
   Select stu_age student_age from student;
c) Find the sum of all three subject marks and name it as tot_marks.
   Select subject1_marks + subject2_marks + subject3_marks tot_marks from student.

2**. Inserting Data into Tables:**

Syntax: Insert into <tablename> (<col1>,<col2>) values (<exp>,<exp>);

Ex: insert into STUDENT(reg_no, stu_name) values (102, 'KRISH');

3. Delete operations

a) Removal of specified row/s

Syntax:Delete from <tablename> where <condition>;

Ex: Delete from STUDENT where reg_no=102;

b) Remove all rows

Syntax: Delete from <tablename>;

Ex: Delete from STUDENT;

4. Updating the contents of a table

a) Updating all rows

Syntax: Update <tablename> set <col>=<exp>, <col>=<exp>;

Ex: Update STUDENT set stu_name='MANAV';

b) Updating selected records

Syntax: Update <tablename> set <col>=<exp>,<col>=<exp>where <condition>;

Ex: Update STUDENT set stu_name='YADAV' where reg_no=101;

LAB EXERCISES:

1. Create a table employee with (emp_no, emp_name, emp_address)
2. Insert five employees information.
3. Display names of all employees.
4. Display all the employees from 'MANIPAL'.
5. Add a column named salary to employee table.
6. Assign the salary for all employees.
7. View the structure of the table employee using describe.
8. Delete all the employees from 'MANGALORE'
9. Rename employee as employee1.
10. Drop the table employee1.

LAB NO.: 2 **Date:**

INTEGRITY CONSTRAINTS IN SQL

Objectives:

In this lab, student will be able to:

- Understand the use of integrity constraints.

Integrity Constraints:

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database. Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQL.

- NOT NULL Constraint: Ensures that a column cannot have NULL value.
- DEFAULT Constraint: Provides a default value for a column when none is specified.
- UNIQUE Constraint: Ensures that all values in a column are different.
- PRIMARY Key: Uniquely identifies each row/record in a database table.
- FOREIGN Key: Uniquely identifies row/record in another database table.
- CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use ALTER TABLE statement to create constraints even after the table is created.

Dropping Constraints:

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the following command:

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

Integrity Constraints Syntax:

- Ensure the uniqueness of the primary key(PRIMARY KEY)
  →column_name data_type primary key
  → Primary key(column_name(s))
- Ensure the uniqueness of the candidate key which is not the primary key
  →column_name data_type unique
- Ensure that child records in related tables have a parent record.
  →foreign key(column_name) references table_name(column_name)
- Delete child records when the parent record is deleted.
  →foreign key(column_name) references table_name(column_name) on delete cascade
- Ensure that columns always contain a value.
  →column_name data_type not null
- Ensure that a column contains a value within a set/specific range.
  →check (column_name in (value1, value2,..))
  →check (predicate)
- Ensure that a default value is placed in a column.
  →column_name data_type default (value)
- Naming Constraints: Constraints can have unique user defined name as given below
  →CONSTRAINT <constraint_name> <constraint_definition>
  Ex.: constraint account_pk primary key(account_number)
- To list constraints defined on the table:
  Select * from user_cons_columns where table_name=’employee’;
- Modifying Constraints:

ALTER TABLE <table_name>
ADD / MODIFY/DROP/DISABLE/ENABLE/VALIDATE/NOVALIDATE
CONSTRAINT <constraint_name>

Built-in Functions:

- **LENGTH (string):** The Oracle/PLSQL LENGTH function returns the length of the specified string.
  Syntax: LENGTH(string1)

- **LOWER (string):** The Oracle/PLSQL LOWER function converts all letters in the specified string to lowercase. If there are characters in the string that are not letters, they are unaffected by this function.
  Syntax: LOWER(string)

- **SUBSTR(string, start, count):**The Oracle/PLSQL SUBSTR functions allows you to extract a substring from a string.
  Syntax: SUBSTR(string, start_position [, length])

- **UPPER (string):**The Oracle/PLSQL UPPER function converts all letters in the specified string to uppercase. If there are characters in the string that are not letters, they are unaffected by this function.

  Syntax: UPPER (string)
- **NVL (column name, substitute value):** The Oracle/PLSQL NVL function lets you substitute a value when a null value is encountered.
  Syntax: NVL (string, replace_with)

- **ROUND (value, precision):** The Oracle/PLSQL ROUND function returns a number rounded to a certain number of decimal places.
  Syntax: ROUND (number [, decimal places])

- **TO_CHAR (date1, format):** The Oracle/PLSQL TO_CHAR function converts a number or date to a string.

Syntax: TO_CHAR(value [, format mask] [, nls_language])

- **LAST_DAY(date):**The Oracle/PLSQL LAST_DAY function returns the last day of the month based on a *date* value.
  Syntax: LAST_DAY(date)
- **MONTHS_BETWEEN (date1, date2):**The Oracle/PLSQL MONTHS_BETWEEN function returns the number of months between *date1* and *date2*.
  Syntax: MONTHS_BETWEEN (date1, date2)
- **NEXT_DAY(date1, 'day'):**The Oracle/PLSQL NEXT_DAY function returns the first weekday that is greater than a *date*.
  Syntax: NEXT_DAY (date, weekday)
- **TO_DATE (string, 'format'):**The Oracle/PLSQL TO_DATE function converts a string to a date.
  Syntax:TO_DATE(string1 [, format_mask] [, nls_language])

  Ex: to_date ('12021998', 'DDMMYYYY')

LAB EXERCISES:

Consider the following schema:
Employee (EmpNo, EmpName, Gender, Salary, Address, DNo)
Department (DeptNo, DeptName, Location)

1. Create Employee table with following constraints:
   - Make EmpNo as Primary key.
   - Do not allow EmpName, Gender, Salary and Address to have null values.
   - Allow Gender to have one of the two values: 'M', 'F'.
2. Create Department table with following:
   - Make DeptNo as Primary key
   - Make DeptName as candidate key
3. Make DNo of Employee as foreign key which refers to DeptNo of Department.
4. Insert few tuples into Employee and Department which satisfies the above constraints.

5. Try to insert few tuples into Employee and Department which violates some of the above constraints.
6. Try to modify/delete a tuple which violates a constraint.

   (**Ex:** Drop a department tuple which has one or more employees)
7. Modify the foreign key constraint of Employee table such that whenever a department tuple is deleted, the employees belonging to that department will also be deleted.

Naming Constraints:

8. Create a named constraint to set the default salary to 10000 and test the constraint by inserting a new record.

University Database Schema:

Student (ID, name,dept-name, tot-cred)
Instructor(ID, name, dept-name, salary)
Course (Course-id, title,dept-name, credits)
Takes (ID, course-id, sec-id, semester, year, grade)
Classroom (building, room-number, capacity)
Department (dept-name, building, budget)
Section (course-id, section-id, semester, year, building, room-number, time-slot-id)
Teaches (id, course-id, section-id, semester, year)
Advisor(s-id, i-id)
Time-slot (time-slot-id, day, start-time, end-time)
Prereq (course-id, prereq-id)

(Use University database for the exercise problems given below)

Retrieving records from a table:

9. List all Students with names and their department names.
10. List all instructors in CSE department.
11. Find the names of courses in CSE department which have 3 credits.
12. For the student with ID 12345 (or any other value), show all course-id and title of all courses registered for by the student.

13. List all the instructors whose salary is in between 40000 and 90000.

Retrieving records from multiple tables:

14. Display the IDs of all instructors who have never taught a course.
15. Find the student names, course names, and the year, for all students those who have attended classes in room-number 303.

Rename and Tuple Variables (Use as in select and from):

16. For all students who have opted courses in 2015, find their names and course id's with the attribute course title replaced by c-name.
17. Find the names of all instructors whose salary is greater than the salary of at least one instructor of CSE department and salary replaced by inst-salary.

String Operations (Use %, _, LIKE):

18. Find the names of all instructors whose department name includes the substring 'ch'.

Built-in Functions:

19. List the student names along with the length of the student names.
20. List the department names and 3 characters from 3rd position of each department name
21. List the instructor names in upper case.
22. Replace NULL with value1(say 0) for a column in any of the table
23. Display the salary and salary/3 rounded to nearest hundred from Instructor.

Add date of birth column (DOB) to Employee Table. Insert appropriate DOB values for different employees and try the exercise problems given below:

24. Display the birth date of all the employees in the following format:
    - 'DD-MON-YYYY'
    - 'DD-MON-YY'
    - 'DD-MM-YY'

25. List the employee names and the year (fully spelled out) in which they are born
    - 'YEAR'
    - 'Year'
    - 'year'

ADDITIONAL EXERCISE:

1. Modify the employee table to check the salary of every employee to be greater than 5000.
2. Find the quarter of year from the given date.
3. Convert seconds to hours, minutes and seconds format.
4. Find the week of the year from the given date.
5. Find the names of all departments with instructor, and remove duplicates.
6. For all instructors who have taught some course, find their names and the course ID of the courses they taught.
7. Find all the instructors with the courses they taught.
8. List all the students with student name, department name, advisor name and the number of courses registered.

LAB NO.: 3 **Date:**

INTERMEDIATE SQL

Objectives:

In this lab, student will be able to:

- Understand the set operations and intermediate level queries.

SET Operations in SQL:

Multiple queries using the set operators **UNION**, **UNION ALL**, **INTERSECT**, and **MINUS**. All set operators have equal precedence. If a SQL statement contains multiple set operators, then Oracle Database evaluates them from the left to right unless parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and must be in the same data type group (such as numeric or character).

The **UNION** operator returns only distinct rows that appear in either result.

```
SELECT product_id FROM order_items
UNION
SELECT product_id FROM inventories;
```

The following statement combines the results with the **INTERSECT** operator, which returns only those rows returned by both queries:

```
SELECT product_id FROM inventories
INTERSECT
SELECT product_id FROM order_items;
```

The following statement combines results with the **MINUS** operator, which returns only unique rows returned by the first query but not by the second:

```
SELECT product_id FROM inventories
MINUS
SELECT product_id FROM order_items;
```

CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

SQL CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

LAB EXERCISE:

Implement the following Queries on UNIVERSITY Database:

Set Operations

UNION (Use union all to retain duplicates):

1. Find courses that ran in Fall 2009 or in Spring 2010

INTERSECT (Use intersect all to retain duplicates):

2. Find courses that ran in Fall 2009 and in spring 2010

MINUS:

3. Find courses that ran in Fall 2009 but not in Spring 2010

Null values

4. Find the name of the course for which none of the students registered.

Nested Subqueries

Set Membership (in / not in):

5. Find courses offered in Fall 2009 and in Spring 2010.
6. Find the total number of students who have taken course taught by the instructor with ID 10101.
7. Find courses offered in Fall 2009 but not in Spring 2010.

8. Find the names of all students whose name is same as the instructor's name.

Set Comparison (>=some/all)

9. Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.
10. Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.
11. Find the departments that have the highest average salary.
12. Find the names of those departments whose budget is lesser than the average salary of all instructors.

Test for Empty Relations (exists/ not exists)

13. Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester.
14. Find all students who have taken all courses offered in the Biology department.

Test for Absence of Duplicate Tuples

15. Find all courses that were offered at most once in 2009.
16. Find all the students who have opted at least two courses offered by CSE department.

Subqueries in the From Clause

17. Find the average instructors salary of those departments where the average salary is greater than 42000

Views

18. Create a view all_courses consisting of course sections offered by Physics department in the Fall 2009, with the building and room number of each section.
19. Select all the courses from all_courses view.
20. Create a view department_total_salary consisting of department name and total salary of that department.

ADDITIONAL EXERCISE:

1. Find the names of all departments with instructor and remove duplicates.
2. For all instructors who have taught some course, find their names and the course ID of the courses they taught.
3. Find all the instructors with the courses they taught.
4. List all the students with student name, department name, advisor name and the number of courses registered.

LAB NO.: 4 **Date:**

COMPLEX QUERIES ON SQL

Objectives:

In this lab, student will be able to:

- Use the concept of grouping and ordering of data.
- Implement Complex Queries
- Understand the concept of COMMIT and ROLLBACK.

GROUP BY:

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples, we specify this in SQL using the group by clause. The attribute or attributes given in the group by clause are used to form group. Tuples with the same value on all attributes in the group by clause are placed in one group.

Syntax:

SELECT columnname, columnname
FROM tablename
GROUP BY columnname;

Example:
SQL> SELECT EMPNO, SUM (SALARY) FROM EMP GROUP BY EMPNO;

JOIN with GROUP BY: This query is used to display a set of fields from two relations by matching a common field in them and group the corresponding records for each and every value of a specified key(s) while displaying.

Syntax: SELECT column_name, aggregate_function(column_name)
FROM relation_1,relation_2
WHERE relation_1.field_x=relation_2.field_y
GROUP BY field_z;

Example:

SQL> SELECT empno,SUM(SALARY) FROM emp,dept WHERE emp.deptno =20 GROUP BY empno;

GROUP BY - HAVING: At times it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those employees where the count of orders is more than 10. This condition does not apply to a single tuple, rather it applies to each group constructed by the GROUP BY clause. HAVING clause is used to handle such cases.

Syntax: SELECT column_name, aggregate_function(column_name)
FROM table_name WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value;

Example:
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM (Orders INNER JOIN Employees ON
Orders.EmployeeID=Employees.EmployeeID)
GROUP BY LastName
HAVING COUNT (Orders.OrderID) > 10;

ORDER BY: This query is used to display a selected set of fields from a relation in an ordered manner base on some field.

Syntax:
SELECT column_name
FROM table_name
ORDER BY column_name;

Example:
SQL> SELECT empno, ename, job
FROM emp
ORDER BY job;

JOIN with ORDER BY: This query is used to display a set of fields from two relations by matching a common field in them in an ordered manner based on some fields.

Syntax:

SELECT column_name1, column_name2
FROM relation_1, relation_2
WHERE relation_1.field_x = relation_2.field_y
ORDER BY field_z;

Example:

SQL> SELECT empno,ename,job,dname
FROM emp,dept
WHERE emp.deptno =dept.dept.no and emp.deptno = 20
ORDER BY job;

WITH Clause: The WITH clause may be processed as an inline view or resolved as a temporary table. The advantage of the latter is that repeated references to the subquery may be more efficient as the data is easily retrieved from the temporary table, rather than being re-queried by each reference.

Syntax:

WITH <temporary table name> AS (
SELECT <attributes>
FROM <table name>
GROUP BY <attribute>)
SELECT <attribute/s> from <tablename/s>

Example:

WITH temporaryTable(averageValue) as
(SELECT avg(Salary) from Employee),
SELECT EmployeeID,Name, Salary from Employee
WHERE Employee.Salary > temporaryTable.averageValue;

COMPLEX QUERIES (Derived relations/Subqueries in the from Clause):
SQL allows a subquery expression to be used in the from clause. The key concept applied here is that any select-from-where expression returns a relation as a result and, therefore, can be inserted into another select-from-where anywhere that a relation can appear.

Example:

select branch-name, avg-balance from (select branch-name, avg (balance) as avg-balance from account group by branch-name) where avg-balance > 1200

COMMIT AND ROLLBACK:

Changes due to insert, update, and delete statements are temporarily stored in the database system. They become permanent only after the COMMIT statement has been issued.

- You can use the SQL ROLLBACK statement to rollback (undo) any changes you made to the database before a COMMIT was issued.
- The SQL COMMIT statement saves any changes made to the database. When a COMMIT has been issued, all the changes since the last COMMIT, or since you logged on as the current user, are saved.

Example:

Implementing the save point

SQL> SAVEPOINT S23;

Save point created.

SQL> select * from employee;

EMPNO	ENAME	JOB	DEPTNO	SAL
1	Mathi	AP	1	10000
2	Arjun	ASP	2	15000
3	Gugan	ASP	1	15000
4	Karthik	Prof	2	30000

SQL> INSERT INTO EMPLOYEE VALUES(5,'Akalya','AP',1,10000);

1 row created.

SQL> select * from employee;

EMPNO	ENAME	JOB	DEPTNO	SAL
1	Mathi	AP	1	10000
2	Arjun	ASP	2	15000
3	Gugan	ASP	1	15000
4	Karthik	Prof	2	30000
5	Akalya	AP	1	10000

Implementing the rollback

SQL> rollback;

SQL> select * from employee;

EMPNO	ENAME	JOB	DEPTNO	SAL
1	Mathi	AP	1	10000
2	Arjun	ASP	2	15000
3	Gugan	ASP	1	15000
4	Karthik	Prof	2	30000

Implementing the commit

SQL> COMMIT;

Commit complete.

LAB EXERCISE:

Implement the following Queries on UNIVERSITY Database.

Group By:

1. Find the number of students in each course.
2. Find those departments where the average number of students are greater than 10.
3. Find the total number of courses in each department.
4. Find the names and average salaries of all departments whose average salary is greater than 42000.
5. Find the enrolment of each section that was offered in Spring 2009.

Ordering the display of Tuples (Use ORDER BY ASC/DESC):

6. List all the courses with prerequisite courses, then display course id in increasing order.
7. Display the details of instructors sorting the salary in decreasing order.

Derived Relations:

8. Find the maximum total salary across the departments.
9. Find the average instructors' salaries of those departments where the average salary is greater than 42000.
10. Find the sections that had the maximum enrolment in Spring 2010
11. Find the names of all instructors who teach all students that belong to 'CSE' department.

12. Find the average salary of those department where the average salary is greater than 50000 and total number of instructors in the department are more than 5.

With Clause:

13. Find all departments with the maximum budget.
14. Find all departments where the total salary is greater than the average of the total salary at all departments.

(Use ROLLBACK (and SAVEPOINT) to undo the effect of any modification on database before COMMIT)

15. Transfer all the students from CSE department to IT department.
16. Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise

ADDITIONAL EXERCISE:

1. Display lowest paid instructor details under each department.
2. Find the sum of the salaries of all instructors of the 'CSE' department, as well as maximum salary, the minimum salary, and the average salary in this department.
3. Retrieve the name of each student who registered for all the subjects offered by 'CSE' department

LAB NO.: 5 **Date:**

ER MODEL AND SQL

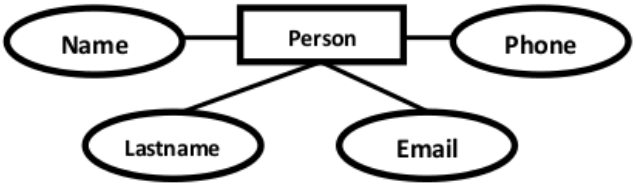
Objectives:

In this lab, student will be able to:

- Convert ER Diagram to relational schema.

Reduction of ER Diagram to Relational Schema: Refer the reference no. 5.

Entities and Simple Attributes:

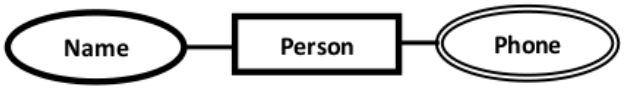


When reducing this ERD into tables we get:

Persons(personid , name, lastname, email)

Multi-Valued Attributes

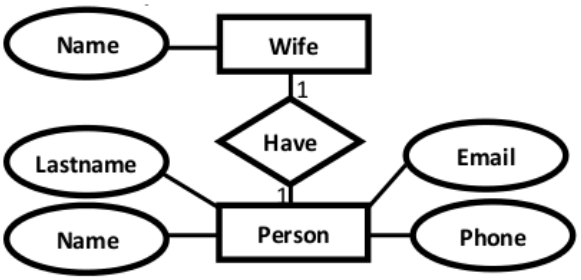
A multi-valued attribute is usually represented with a double-line oval



If you have a multi-valued attribute, take the attribute and turn it into a new entity or table of its own. Then make a 1:N relationship between the new entity and the existing one. In simple words. 1. Create a table for the attribute. 2. Add the primary (id) column of the parent entity as a foreign key within the new table as shown below:

Persons(personid , name, lastname, email)
Phones (phoneid , ***personid***, phone)

1:1 Relationships

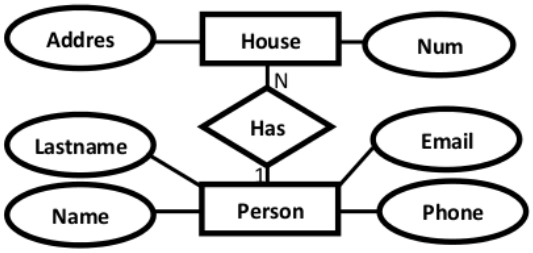


Persons(<u>personid</u> , name, lastname, email , ***wifeid***)
Wife (<u>wifeid</u> , name)

Or

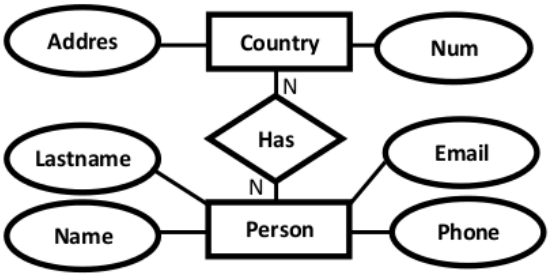
Persons(<u>personid</u> , name, lastname, email)
Wife (<u>wifeid</u> , name , ***personid***)

1:N Relationships



Persons(<u>personid</u> , name, lastname, email)
House (<u>houseid</u> , num , address, ***personid***)

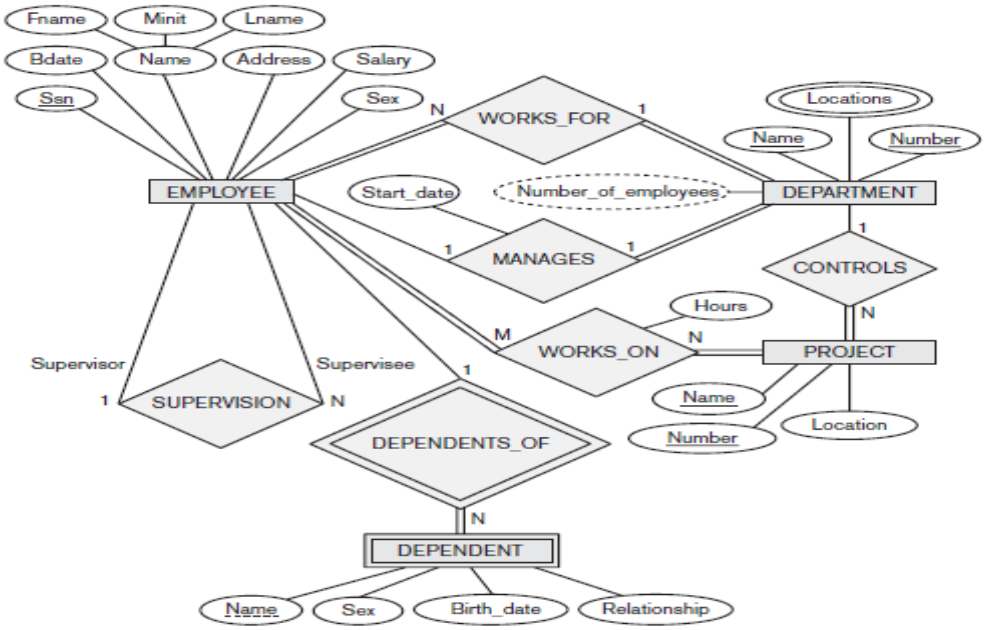
N:N Relationships



Persons(personid, name, lastname, email)
Countries(countryid, name, code)
HasRelat(hasrelatid, personid , countryid)

LAB EXERCISES:

Design the database for the following ER Diagram



Implement the following queries:

1. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'. Retrieve the name and address of all employees who work for the 'Research' department.
2. For every project located in 'Stanford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.
3. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.
4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.
5. Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.
6. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.
7. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.
8. Retrieve the names of employees who have no dependents.
9. List the names of managers who have at least one dependent.
10. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.
11. For each project, retrieve the project number, the project name, and the number of employees who work on that project.
12. For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.
13. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than 40,000.

ADDITIONAL EXERCISE:

1. Find the names of employees who work on all the projects controlled by department number 5.
2. Find the names of all employees who have a higher salary than some instructor in ‘Research’ department.
3. Find the total number of (distinct) employees who have worked on project ‘ProductX’.

LAB NO.: 6 **Date:**

Mini Project (Phase I)

Objectives:
In this lab, student will be able to:

- Identify a domain that uses database to manage data.
- Formulate the synopsis for mini project.
- Perform the requirement gathering and design phases of the project.

Guidelines:
All the students are instructed to form a team of two members. Students need to submit synopsis by the end of sixth week.

Synopsis Format:

- Title, Team Members
- Abstract (with modifications, if any)
- Problem Statement (covering both data and functional requirements)

(Refer 'Project Suggestions' in Chapter 9, Database System Concepts, Korth, Sixth Edition)

LAB EXERCISE:
For the selected problem statement:

1. Design and implement the database.
2. Implement basic and complex queries.

LAB NO: 7 **Date:**

PL/SQL BASICS

Objectives:
In this lab, student will be able to understand and use:

- PL/SQL anonymous block
- PL/SQL Conditional, Iterative and Sequential Control Statements
- Exception Handlers

PL/SQL

PL/SQL is the Oracle procedural extension of SQL. It is a portable, high-performance transaction-processing language.

Though SQL is the natural language of Oracle DB, it has some disadvantages when used as a programming language:

1. SQL does not have any procedural capabilities like condition checking, looping and branching.
2. SQL statements are passed on to the Oracle engine one at a time. This adds to the network traffic in a multi-user environment and decreases data processing speed.
3. SQL has no facility for programmed error handling.

Main Features of PL/SQL

1. PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages.
2. PL/SQL allows users to declare constants and variables, control program flow, define subprograms, and trap runtime errors.
3. Complex problems can be broken into easily understandable subprograms, which can be reused in multiple applications.
4. PL/SQL sends the entire block of SQL to the Oracle engine in one go, reducing network traffic and leading to faster query processing.
5. Variables in PL/SQL blocks can be used to store intermediate result of a query for later processing

PL/SQL Block Structure: The basic unit of a PL/SQL source program is the block, which groups related declarations and statements. The syntax for which is shown in Figure 7.1

```
<< label >> (optional)
DECLARE    -- Declarative part (optional)
  -- Declarations of local types, variables, & subprograms

BEGIN      -- Executable part (required)
  -- Statements (which can use items declared in declarative part)

[EXCEPTION -- Exception-handling part (optional)
  -- Exception handlers for exceptions (errors) raised in executable part]
END;
```

Fig 7.1 PL/SQL Block Structure

COMPONENTS OF A PL/SQL BLOCK

1. DECLARE

An optional declaration part in which variables, constants, cursors, and exceptions are defined and possibly initialised.

2. BEGIN …… END;

A mandatory executable part consists of a set of SQL and PL/SQL statements. Data manipulation, retrieval, looping and branching constructs are specified in this section.

3. EXCEPTION

An optional exception part deals with handling of errors that arise during execution of data manipulation statements in the PL/SQL code block. Errors can arise due to syntax, logic and/or validation rule violation.

Example: A PL/SQL block to display ‘Hello, World’.

```
declare
       message varchar2(20):='Hello, World!';
begin
       dbms_output.put_line(message);
end;
/
```

'SET SERVEROUTPUT ON' command should be issued before executing the PL/SQL block. Alternatively, it can be included in the beginning of every PL/SQL block.

SQL DATA TYPES

The default data types that can be declare in PL/SQL are

1. Number
2. Char
3. Date
4. Boolean

Null values are allowed for Number, Char and Date but not Boolean data type.
E.g.: sname varchar2(30);

%TYPE

PL/SQL can use %Type to declare variables based on column definition in a table. Hence, if a column's attribute changes, the variable's attribute will change as well.

E.g.: sname student.name%TYPE;

%ROWTYPE

The %ROWTYPE attribute lets you declare a record that represents a row in a table or view. For each column in the referenced table or view, the record has a field with the same name and data type. To reference field in the record use *record_name.field_name*. The record fields do not inherit the constraints or default values of the corresponding columns. If the referenced item table or view changes, your declaration is automatically updated. You need not change your code when, for example, columns are added or dropped from the table or view.

E.g., srecord student%ROWTYPE;

Not Null: 'Not null' causes creation of a variable or a constant that cannot be assigned 'null' value. Attempt to assign null value to such a variable or constant will return an internal error.

VARIABLES: In PL/SQL a variable name must begin with a character with maximum length of 30. Space not allowed in variable names. Reserve words cannot be used as variable names unless enclosed within double quotes. Case is insignificant when declaring variables.

Values can be assigned to variables by:

1. Using the assignment operator **:=** (a colon followed by an 'equal to').
2. Selecting or fetching table data values INTO variables.

Constants can be declared with constant keyword.

E.g.: pi constant number := 3.141592654;

DISPLAYING USER MESSAGES ON SCREEN

dbms_output is a package that includes procedures and functions that accumulate information in a buffer so that it can be retrieved later.

put_line is a procedure in dbms_output package used to display information in the buffer. SERVEROUTPUT should be set to ON before calling this procedure.

COMMENTS

Comment can be of two forms, as:

1. The comment line begins with a double hypen (--). The entire line is considered as a comment.
2. The comment line begins with a slash followed by an asterisk (/*) and ends with asterisk followed by slash (*/). All lines within is considered as comments.

CONDITIONAL CONTROL: IF-THEN-ELSIF-ELSE-END IF

Syntax:

```
IF < condition> THEN
       < action >

ELSIF <condition> THEN
       < action >

ELSE
       < action >
END IF;
```

Example: A PL/SQL block to display the grade for given letter grade.

```
DECLARE
       grade CHAR(1);
BEGIN
       grade := '&g';
IF grade = 'A' THEN
       DBMS_OUTPUT.PUT_LINE('Excellent');
ELSIF grade = 'B' THEN
       DBMS_OUTPUT.PUT_LINE('Very Good');
ELSIF grade = 'C' THEN
```

```
       DBMS_OUTPUT.PUT_LINE('Good');
ELSIF grade = 'D' THEN
       DBMS_OUTPUT. PUT_LINE('Fair');
ELSIF grade = 'F' THEN
       DBMS_OUTPUT.PUT_LINE('Poor');
ELSE
       DBMS_OUTPUT.PUT_LINE('No such grade');
END IF;
END;
/
```

ITERATIVE CONTROL: Simple Loop

Syntax:

```
LOOP
       <Sequence of statements>
END LOOP;
```

Once a loop begins to execute, it will go on forever. A conditional statement to control the number of times loop executes should accompany the simple loop construct.

Example: The following PL/SQL block traces the control flow in a simple LOOP construct.

```
DECLARE
       x NUMBER := 0;
BEGIN
       LOOP
              DBMS_OUTPUT.PUT_LINE ('Inside loop: x = ' ||
                                                  TO_CHAR(x));
              x := x + 1;
              IF x > 3 THEN  EXIT;
              END IF;
       END LOOP;
-- After EXIT, control resumes here
DBMS_OUTPUT.PUT_LINE(' After loop: x = ' || TO_CHAR(x));
END;
```

Output:

```
Inside loop: x = 0
Inside loop: x = 1
Inside loop: x = 2
Inside loop: x = 3
After loop: x = 4
```

ITERATIVE CONTROL: While Loop

Syntax:

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

Example: The following PL/SQL block traces the control flow in a while LOOP.

```
DECLARE
       x NUMBER := 0;
BEGIN
       WHILE x < 4
         LOOP
              DBMS_OUTPUT.PUT_LINE ('Inside loop: x = ' ||
                                                    TO_CHAR(x));
              X := x + 1;
         END LOOP
END;
/
```

ITERATIVE CONTROL: For Loop

Syntax:

```
FOR index IN [ REVERSE ] lower_bound..upper_bound LOOP
statements
END LOOP ;
```

With REVERSE, the value of index starts at upper_bound and decreases by one with each iteration of the loop until it reaches lower_bound.

Example: The following PL/SQL block traces the control flow in a FOR LOOP

```
BEGIN

       DBMS_OUTPUT.PUT_LINE ('lower_bound < upper_bound');
       FOR i IN 1..3 LOOP
              DBMS_OUTPUT.PUT_LINE (i);
       END LOOP;
       DBMS_OUTPUT.PUT_LINE ('lower_bound = upper_bound');
       FOR i IN 2..2 LOOP
              DBMS_OUTPUT.PUT_LINE (i);
       END LOOP;

       DBMS_OUTPUT.PUT_LINE ('lower_bound > upper_bound');
       FOR i IN 3..1 LOOP
              DBMS_OUTPUT.PUT_LINE (i);
       END LOOP;
END;
/
```

Output:

```
lower_bound < upper_bound
1
2
3
lower_bound = upper_bound
2
lower_bound > upper_bound
```

SEQUENTIAL CONTROL: GOTO statement

The GOTO statement transfers control to a label unconditionally. The label must be unique in its scope and must precede an executable statement or a PL/SQL block. When run, the GOTO statement transfers control to the labeled statement or block.

Syntax:

```
GOTO Label
The code block is marked using tags
<<Label>>
```

Example: A PL/SQL block to check whether a given number is prime number.

```
DECLARE
       p VARCHAR2(30);
       n PLS_INTEGER := 37;
BEGIN
       FOR j in 2..ROUND(SQRT(n)) LOOP
              IF n MOD j = 0 THEN
                     p := ' is not a prime number';
                     GOTO print_now;
              END IF;
       END LOOP;
       p := ' is a prime number';
<<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

ERROR HANDLING:

In PL/SQL, a warning or error condition is called an *exception*. Exceptions can be internally defined (by the run-time system) or user defined. Examples of internally defined exceptions include *division by zero* and *out of memory* etc.

When an error occurs, an exception is *raised*. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system.

To handle raised exceptions, you write separate routines called *exception handlers*. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

Syntax:

```
EXCEPTION
WHEN ex_name_1 THEN
       < Exception handler_1>
WHEN ex_name_2 OR ex_name_3 THEN
       < Exception handler_2>
WHEN OTHERS THEN
       < Exception handler_3>
END;
```

Some pre-defined exceptions:

Exception	Raised when ...
DUP_VAL_ON_INDEX	The program attempts to store duplicate values in a database column that is constrained by a unique index.
NO_DATA_FOUND	A SELECT INTO statement returns no rows
PROGRAM_ERROR	PL/SQL has an internal problem.
TOO_MANY_ROWS	A SELECT INTO statement returns more than one row.
VALUE_ERROR	An arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	The program attempts to divide a number by zero.

Example: Update all accounts with balance less than 0 to 0 in account(account_number, balance) table, populated with {(1, 100); (2, 3000); (3, 500)}

```
DECLARE
BEGIN
       update account set balance = 0 where balance<0

EXCEPTION

       WHEN NO_DATA_FOUND THEN dbms_output.put_line("No rows
found");
```

```
END;
```

User Defined Exceptions:

User can define exceptions of his own in the declarative part of any PL/SQL block, subprogram, or package. For example, an exception named `insufficient_balance` can be defined to flag overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions *must* be given names and they must be raised explicitly by `RAISE` statements.

Example: Validate the resultant balance before supporting withdrawal of money from the account table such that a min. of 500 is maintained.

```
Declare
       Insufficient_Balance Exception;
       Amount account.balance%Type;
       Temp account.balance%Type;
       ANumber account.account_number%Type;
BEGIN
       ANumber := &Number;
       Amount := &Amount;
       Select balance into Temp from account where
                                    account_number = ANumber;
       Temp := Temp – Amount;
       IF (Temp >=500) THEN
       update account set balance = Temp where account_number =
       ANumber;
       ELSE
       RAISE Insufficient_Balanae;
EXCEPTION
       WHEN Insufficient_Balance THEN
              dbms_output.put_line(“Insufficient Balance”);

WHEN OTHERS THEN
              dbms_output.put_line(“ERROR”);

END;
```

LAB EXERCISE:

NOTE: Use a table StudentTable(RollNo, GPA) and populate the table with {(1, 5.8); (2, 6.5); (3, 3.4); (4,7.8); (5, 9.5)} unless a different DB schema is explicitly specified.

1. Write a PL/SQL block to display the GPA of given student.

Usage of IF –THEN:

2. Write a PL/SQL block to display the letter grade(0-4: F; 4-5: E; 5-6: D; 6-7: C; 7-8: B; 8-9: A; 9-10: A+} of given student.
3. Input the date of issue and date of return for a book. Calculate and display the fine with the appropriate message using a PL/SQL block. The fine is charged as per the table 8.1:

Late period	Fine
7 days	NIL
8 – 15 days	Rs.1/day
16 - 30 days	Rs. 2/ day
After 30 days	Rs. 5.00

Table 8.1

Simple LOOP:

4. Write a PL/SQL block to print the letter grade of all the students(RollNo: 1 - 5).

Usage of WHILE:

5. Alter StudentTable by appending an additional column LetterGrade Varchar2(2). Then write a PL/SQL block to update the table with letter grade of each student.

Usage of FOR:

6. Write a PL/SQL block to find the student with max. GPA without using aggregate function.

Usage of GOTO:

7. Implement lab exercise 4 using GOTO.

Exception Handling:

8. Based on the University database schema, write a PL/SQL block to display the details of the Instructor whose name is supplied by the user. Use exceptions to show appropriate error message for the following cases:
    a. Multiple instructors with the same name
    b. No instructor for the given name

ADDITIONAL EXERCISE

Usage of IF –THEN:

1. Write a PL/SQL block to find out if a year is a leap year.

2. You went to a video store and rented DVD that is due in 3 days from the rental date. Input the rental date, rental month and rental year. Calculate and print the return date, return month, and return year.

Simple LOOP:

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

4. Write a PL/SQL block to print all odd numbers between 1 and 10.

Usage of WHILE:

5. Write a PL/SQL block to reverse a given string.

Usage of FOR:

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

Usage of GOTO:

7. Write a PL/SQL block of code to achieve the following: if the price of Product 'p00001' is less than 4000, then change the price to 4000. The Price change has to be recorded in the old_price_table along with Product_no and the date on which the price was last changed. Tables involved:

Product_master(product_no, sell_price)
Old_price_table(product_no,date_change, Old_price)

Exception:

8. Write a PL/SQL block that asks the user to input first number, second number and an arithmetic operator (+, -, *, /). If the operator is invalid, throw and handle a user-defined exception. If the second number is zero and the operator is /, handle the ZERO_DIVIDE predefined server exception.

LAB NO: 8 **Date:**

CURSORS

Objectives:
In this lab, student will be able to understand and use:

- Implicit and Explicit Cursors
- Cursor For Loops, Parameterized Cursors
- Transactions

CURSORS: A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.

There are two kinds of cursors:

1. **Implicit Cursor**: A cursor that is constructed and managed by PL/SQL
2. **Explicit Cursor**: A cursor that the user constructs and manages.

Implicit Cursor

Oracle engine implicitly opens a cursor on the server to process each SQL statement and manages space, populates data and releases memory itself. Implicit cursor can be used to access information about the status of last insert, update, delete or single row select statements. SQL is the cursor name of the implicit cursor. The attributes of the implicit cursor are listed below.

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND.

%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Example: A PL/SQL block to delete the student records of History Department in University database. Use implicit cursor attributes to check the success of delete operation.

```
DECLARE

dname CONSTANT student.dept_name%TYPE := 'History';

BEGIN

       DELETE FROM student WHERE dept_name = dname;
IF SQL%FOUND THEN
       DBMS_OUTPUT.PUT_LINE ('Delete succeeded for department:
       '|| dname);
ELSE
       DBMS_OUTPUT.PUT_LINE ('No students of department: ' ||
       dname);
END IF;

END;
/
```

Output: If there is no row that matches the delete query in *student* table.

```
No students of department: History
```

Explicit Cursor

Steps involved in using an explicit cursor and manipulating data in its active set are:

1. Declare a cursor mapped to a select statement.
2. Open the Cursor.
3. Fetch data from the cursor one row at a time into memory variables
4. Process the data held in the memory variables as required using a loop

5. Exit from loop after processing is complete
6. Close the cursor

Cursor is defined in the declarative part.

Syntax:

```
CURSOR [ parameter_list ] [ RETURN return_type ]
IS select_statement;
OPEN cursor_name;
Loop
FETCH cursor_name into variable_list;
End Loop

CLOSE cursor_name;
```

No memory is allocated at this point and only intimation is sent to the engine.

Open defines a private area named by the cursor name, executes the query , retrieves the data and creates the Active Data Set.

Fetch moves the data held in the active data set into the memory variable. It is placed in a Loop … End Loop which causes data to be fetched and processed until all rows are processed

Close disables the cursor and the active data set becomes undefined. After the fetch loop is executed the data needs to be closed. Close will release the memory occupied by the cursor.

Example: A PL/SQL block to list the student names of 'Comp. Sci.' department in University database.

```
DECLARE
dname CONSTANT student.dept_name%TYPE := 'Comp. Sci.';
CURSOR C1 is select name from Student where dept_name = dname;
sname student.name%TYPE;
BEGIN
```

```
DBMS_OUTPUT.PUT_LINE( '---------------------------------');
OPEN c1;
       LOOP
         -- Fetches student name into variable
         FETCH c1 INTO  sname;
```

```
          EXIT WHEN c1%NOTFOUND;
          DBMS_OUTPUT.PUT_LINE( RPAD('Name: ' || sname);
        END LOOP;
CLOSE c1;
DBMS_OUTPUT.PUT_LINE( '-----------------------------------');
END;
/
```

CURSOR FOR LOOPS

The cursor FOR LOOP statement lets the user run a SELECT statement and then immediately loop through the rows of the result set. This statement can use either an implicit or explicit cursor (but not a cursor variable).
The cursor FOR LOOP statement implicitly declares its loop index as a %ROWTYPE record variable of the type that its cursor returns. This record is local to the loop and exists only during loop execution. Statements inside the loop can reference the record and its fields.

After declaring the loop index record variable, the FOR LOOP statement opens the specified cursor. With each iteration of the loop, the FOR LOOP statement fetches a row from the result set and stores it in the record. When there are no more rows to fetch, the cursor FOR LOOP statement closes the cursor. The cursor also closes if a statement inside the loop transfers control outside the loop or if PL/SQL raises an exception.

Example: A PL/SQL block to find the department having maximum Budget in University database, without using max() aggregate function.

```
DECLARE
CURSOR C1 is select * from Department;
vBudget Department.Budget%Type :=0;
vdname Department.dept_name%TYPE;
BEGIN
 For dept in C1
 LOOP
 IF dept.Budget > vBudget THEN
 vBudget := dept.Budget;
 vdname := dept.dept_name;
 END IF;
 END LOOP;
 Dbms_output.put_line('Max. Budget: ' || vBudget || '  Dept: '
                      || vdname);
END;
```

WHERE CURRENT OF

It states that the most recent row fetched from the table should be updated or deleted. Inside a cursor loop, WHERE CURRENT OF allows the current row to be directly updated. When the sessions open a cursor with the FOR UPDATE clause, all the rows in the return set will hold row level locks. When SELECT FOR UPDATE is associated with an explicit cursor, the cursor is called a FOR UPDATE cursor. Only a FOR UPDATE cursor can appear in the CURRENT OF clause of an UPDATE or DELETE statement.

Example: A PL/SQL block to increase the Budget of different departments in University database based upon current Budget. Increase the budget by 10%, 15% or 20% for the ranges 'greater than 100000', 'between 70000 and 100000' or 'less than or equal to 70000', respectively.

```
DECLARE
CURSOR C1 is select * from Department for update;
BEGIN
 For dept in C1
 LOOP
 IF dept.Budget <= 70000 THEN
 update Department SET Budget = Budget*1.2 where current of C1;
 ELSIF dept.Budget > 7000 and dept.Budget <= 100000 THEN
 update Department SET Budget = Budget*1.15 where current of C1;
 ELSE
 update Department SET Budget = Budget*1.1 where current of C1;
 END IF;
 END LOOP;
 END;
/
```

PARAMATERIZED CURSORS

PL/SQL allows parametrized cursor so that the cursor can be generic and the data that is retrieved from the table be changed according to need. The user can create an explicit cursor that has formal parameters, and then pass different actual parameters to the cursor each time it is opened. In the cursor query, a formal cursor parameter can be used anywhere that a constant is used. Outside the cursor query, formal cursor parameters cannot be referenced.

Syntax:

```
CURSOR cursor_name( variable_name Datatype) is SELECT statement
```

Example: Based on the University database schema, the following example uses a parametrized cursor on the Instructor table to display the instructors of given department.

```
DECLARE
       cursor c(dname instructor.dept_name%TYPE) is select *
       from Instructor  where dept_name = dname;
BEGIN
       FOR tmp IN c('Comp. Sci.')
       LOOP
              dbms_output.put_line('EMP_No: '||tmp.ID);
              dbms_output.put_line('EMP_Name: '||tmp.name);
              dbms_output.put_line('EMP_Dept: '||tmp.dept_name);
              dbms_output.put_line('EMP_Salary:'||tmp.salary);
       DBMS_OUTPUT.PUT_LINE( '-------------------------------');
       END Loop;
END;
/
```

TRANSACTIONS:

A series of one or more statements that are logically related are termed as a Transaction. A transaction begins with the first executable SQL statement after a commit, rollback or connection made to the oracle engine. A transaction can be closed by using a commit or a rollback statement.

COMMIT ends the current transaction and makes permanent changes made during the transaction

ROLLBACK ends the transaction but undoes any changes made during the transaction

SAVEPOINT marks and saves the current point in the processing of a transaction.

Syntax: Commit, Rollback & Savpoint

```
COMMIT;
SAVEPOINT <Savepointname>
ROLLBACK TO <Savepointname>;

SAVEPOINT: Is optional and is used to rollback a transaction
partially as far as the specified savepoint
Savepointname: Is a savepoint created during the current
transaction
```

ROLLBACK can be fired with or without the SAVEPOINT clause. Rollback operation performed without the SAVEPOINT clause amounts to the following:

1. Ends the transaction
2. Undoes all the changes in the current transaction
3. Erases all savepoints in that transaction
4. Releases the transactional locks

Example:

Consider account (account_number, balance) table, populated with {(1, 200); (2, 3000); (3, 500)}. Withdraw an amount 200 and deposit 1000 for all the accounts. If the sum of all account balance exceeds 5000 then undo the deposit just made.

```
Declare
       Total_bal account.balance%TYPE;
Begin
       Update account set balance=balance-200;
       Savepoint deposit;
       Update account set balance=balance+1000;
       Select sum(balance) into total_bal from account;
       If total_bal > 5000 then
              Rollback to savepoint deposit;
       End If;
       Commit;
End;
```

LAB EXERCISE:

Note: Use University DB schema for the following, unless a different DB schema is explicitly specified

Cursors:

CursorName %ISOPEN / FOUND / NOT FOUND:

1. The HRD manager has decided to raise the salary of all the Instructors in a given department number by 5%. Whenever, any such raise is given to the instructor, a record for the same is maintained in the salary_raise table. It includes the Instuctor Id, the date when the raise was given and the actual raise amount. Write a PL/SQL block to update the salary of each Instructor and insert a record in the salary_raise table.

   salary_raise(Instructor_Id, Raise_date, Raise_amt)

CursorName%ROWCOUNT:

2. Write a PL/SQL block that will display the ID, name, dept_name and tot_cred of the first 10 students with lowest total credit.

Cursor For Loops:

3. Print the Course details and the total number of students registered for each course along with the course details - (Course-id, title, dept-name, credits, instructor_name, building, room-number, time-slot-id, tot_student_no)

4. Find all students who take the course with Course-id: CS101 and if he/ she has less than 30 total credit (tot-cred), deregister the student from that course. (Delete the entry in Takes table)

Where Current of:

5. Alter StudentTable(refer Lab No. 8 Exercise) by resetting column LetterGrade to F. Then write a PL/SQL block to update the table by mapping GPA to the corresponding letter grade for each student.

Parameterized Cursors:

6. Write a PL/SQL block to print the list of Instructors teaching a specified course.

7. Write a PL/SQL block to list the students who have registered for a course taught by his/her advisor.

ADDITIONAL EXERCISE:

Cursors

1. Write a PL/SQL block that will display the name, department and salary of the top 10 highest paid instructors.

Cursor For Loops:

2. Repeat problem 1
   i. Using cursor for loops. ii. Using where current of.

Parameterized Cursors:

3. Write a PL/SQL block that would update the Bal_stock in the item_master(ItemId, Description, Bal_stock) table each time a transaction takes place with an entry in the item_transaction (TransID, ItemId, Quantity) table. The change in the item_master table depends on the itemId. If the item is present in the item_master table then Bal_stock is updated. Otherwise, itemID is inserted into the item_master table with ZERO as Bal_stock and raises an exception.

4. Write a PL/SQL block to find the number of courses offered and number of students of each department. Use a parameterized cursor which takes department name as a parameter and calculates the number of courses offered and number of students of that department.

Transactions: COMMIT / ROLLBACK / SAVEPOINT:

5. Write a PL/SQL block that will insert a new record in Takes (ID, course-id, sec-id, semester, year, grade) table. Check the total number of students registered for the course and if it exceeds 30, then undo the insert made to the Takes table.

LAB NO.: 9 **Date:**

PROCEDURES, FUNCTIONS & PACKAGES

Objectives:

In this lab, student will be able to:

- Understand the different types of subprograms: Procedures and Function.
- Use Procedures and Functions to perform specific tasks.
- Understand the concept of Packages.

Subprograms: A PL/SQL subprogram is a named PL/SQL block that can be invoked repeatedly. If the subprogram has parameters, their values can differ for each invocation. PL/SQL has two types of subprograms:

1. **Procedure:** used to perform an action.
2. **Function:** to compute and return a value.

Uses of Subprograms:

- Modularity
- Easier Application Design
- Maintainability
- Packageability
- Reusability
- Better Performance

Since the stored subprograms run in the database server, a single invocation over the network can start a large job.

Part of a Subprogram:

- Declarative part(optional)
- Executable part(required)
- Exception handling part(optional)

PROCEDURE

A procedure is a subprogram that performs a specific action. A procedure invocation (or call) is a statement. In a procedure a return statement returns the control to the invoker and we cannot specify any expression. The declarative part of a subprogram does not begin with the keyword DECLARE, as the declarative part of an anonymous block does.

Syntax:

CREATE OR REPLACE PROCEDURE procedure_name(parameters list) is

```
BEGIN
//statements
END;
/
```

The procedure can be executed by just calling the procedure name with parameters in other PL/SQL block. e.g.: procedure_name (actual values for the parameters);

Example: Create a procedure to print Hello world and execute the procedure.

```
create or replace procedure print_hello is
        begin
        dbms_output.put_line('Hello World');
        end;
  /
```

In another PL/SQL block procedure is called as shown below:

```
declare
        begin
        print_hello;
        end;
/
```

Output:

Hello World

FUNCTIONS

A function has a same structure as a procedure except:

- A function must include a RETURN clause, which specifies the data type of the value that the function returns (A procedure heading cannot have a RETURN clause)
- In the executable part of a function, every execution path must lead to a RETURN statement. Otherwise, the PL/SQL compiler issues a compile-time warning.

Syntax:

CREATE OR REPLACE FUNCTION *function_name*(*variable_name* datatype)

```
        RETURN datatype
        AS
           //declare section
```

```
BEGIN
…….
RETURN var2 //variable of the return datatype
END;
/
```

Example 1: Create a function to return the sum of two numbers

```
create or replace function sum_number(a number, b number)
 return number as
 tot number;
 begin
    tot := a + b;
return tot;
end;
/
```

In another PL/SQL Block

```
setserveroutput on;
declare
begin
dbms_output.put_line(sum_number(5,4));
end;
/
```

Output: 9

Example 2: A function that, given the name of a department, returns the count of the number of instructors in that department. (Using University database schema)

```
create function dept count(dept name varchar)
returns integer
begin
declare d count integer;
select count(*) into d count
from instructor
where instructor.dept name= dept name
return d count;
end
```

This function can be used in a query that returns names and budgets of all departments with more than 12 instructors:

```
select dept name, budget
from instructor
where dept count(dept name) > 12;
```

PACKAGES: A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents. It is a way of creating generic, encapsulated, re-usable code.

Components of an Oracle Package:

A package has two components:

1. **A specification:** declares memory variables, constants, exceptions, cursors and subprograms that are available in the package as public items.
2. **A body:** defines queries for public cursors and code for public subprograms. It can also declare and define private items that cannot be referenced from outside the package. The body can be changed without changing the specification or the references to the public items.

Features of Packages:

Modularity: Packages encapsulates logically related types, variables, constants, subprograms, cursors, and exceptions in named PL/SQL modules.

Easier Application Design: When designing an application, initially only the interface information is required in the package specifications. The specifications can be coded and compiled without their bodies.

Information Hiding: Packages lets the user share your interface information in the package specification, and hide the implementation details in the package body.

Added Functionality: Package public variables and cursors can persist for the life of a session. They can be shared by all subprograms that run in the environment. They allow maintaining data across transactions without storing it in the database.

Better Performance: The first time when a package subprogram is invoked, Oracle Database loads the whole package into memory. Subsequent invocations of other subprograms in same the package require no disk I/O. Packages prevent cascading dependencies and unnecessary recompiling.

Easier to Grant Roles: The user can grant roles on the package, instead of granting roles on each object in the package.

Package Specification: A package specification contains:

- Name of the package
- Names of the data types of any arguments

This declaration is local to the database and global to the package.

(Examples are based on the HR schema provided in the appendix.)

Example:

```
CREATE PACKAGE emp_bonus AS
PROCEDURE calc_bonus
(date_hiredemployees.hire_date%TYPE);
END emp_bonus;
/
```

Package Body

The body of the package contains the definition of public objects that are declared in the specification. The body can also contain other object declaration that is private to the package.

Example:

```
CREATE OR REPLACE PACKAGE BODY emp_bonus AS
PROCEDURE calc_bonus(date_hiredemployees.hire_date%TYPE) IS
BEGIN
DBMS_OUTPUT.PUT_LINE
('Employees hired on ' || date_hired || 'get bonus.');
END;
END emp_bonus;
/
```

Invoking a Package

Example: Invoking the 'calc_bonus' procedure in 'emp_bonus' package

```
Execute emp_bonus. calc_bonus("14-Jan-2012");
[or]
Call emp_bonus. calc_bonus("14-Jan-2012");
```

PL/SQL Subprogram Parameter Modes: IN, OUT, IN

Example:

```
CREATE OR REPLACE PACKAGE emp_mgmt AS
FUNCTION hire (last_name VARCHAR2, job_id VARCHAR2, manager_id
NUMBER,
salary NUMBER, commission_pct NUMBER, department_id
```

```
NUMBER)
RETURN NUMBER;
FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
RETURN NUMBER;
PROCEDURE remove_emp(employee_id NUMBER);
PROCEDURE remove_dept(department_id NUMBER);
PROCEDURE increase_sal(employee_id NUMBER, salary_incr
NUMBER);
PROCEDURE increase_comm(employee_id NUMBER, comm_incr
NUMBER);
no_comm EXCEPTION;
no_sal EXCEPTION;
END emp_mgmt;
/
```

The specification for the emp_mgmt package declares these public program objects:

- The functions hire and create_dept
- The procedures remove_emp, remove_dept, increase_sal, and increase_comm
- The exceptions no_comm and no_sal

Package Body:

```
CREATE OR REPLACE PACKAGE BODY emp_mgmt  AS
FUNCTION  hire (last_name VARCHAR2, job_id VARCHAR2, manager_id
NUMBER, salary NUMBER, commission_pct NUMBER, department_id
NUMBER)
RETURN NUMBER  AS
//declare section
        BEGIN
        ---------
        RETURN var2 //variable of the return datatype
        END;
/

FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
RETURN NUMBER AS
//declare section
        BEGIN
        …….
        RETURN var2 //variable of the return datatype
        END;
```

```
/
PROCEDURE remove_emp(employee_id NUMBER) is

 BEGIN
……
   END;
 END emp_mgmt;
/
```

PL/SQL Subprogram Parameter Modes: IN, OUT, IN OUT:

PL/SQL Subprogram Parameter Modes

IN	OUT	IN OUT
Default mode	Must be specified.	Must be specified.
Passes a value to the subprogram.	Returns a value to the invoker.	Passes an initial value to the subprogram and returns an updated value to the invoker.
Formal parameter acts like a constant: When the subprogram begins, its value is that of either its actual parameter or default value, and the subprogram cannot change this value.	Formal parameter is initialized to the default value of its type. The default value of the type is `NULL` except for a record type with a non-`NULL` default value When the subprogram begins, the formal parameter has its initial value regardless of the value of its actual parameter. Oracle recommends that the subprogram assign a value to the formal parameter.	Formal parameter acts like an initialized variable: When the subprogram begins, its value is that of its actual parameter. Oracle recommends that the subprogram update its value.
Actual parameter can be a constant, initialized variable, literal, or expression.	If the default value of the formal parameter type is `NULL`, then the actual parameter must be a variable whose data type is not defined as `NOT NULL`.	Actual parameter must be a variable (typically, it is a string buffer or numeric accumulator).
Actual parameter is passed by reference.	By default, actual parameter is passed by value; if you specify `NOCOPY`, it might be passed by reference.	By default, actual parameter is passed by value (in both directions); if you specify `NOCOPY`, it might be passed by reference.

Note: Do not use OUT and IN OUT for function parameters. Ideally, a function takes zero or more parameters and returns a single value. A function with IN OUT parameters returns multiple values and has side effects. Regardless of how an OUT or IN OUT parameter is passed:

- If the subprogram exits successfully, then the value of the actual parameter is the final value assigned to the formal parameter. (The formal parameter is assigned at least one value—the initial value.)
- If the subprogram ends with an exception, then the value of the actual parameter is undefined.
- Formal OUT and IN OUT parameters can be returned in any order. In this example, the final values of x and y are undefined:

```
CREATE OR REPLACE PROCEDURE p (x OUT INTEGER, y OUT INTEGER)
AS
BEGIN
x := 17; y := 93;
END;
/
```

Example 1:
A procedure that, given the name of a department, returns the count of the number of instructors in that department.

```
create procedure dept_count_proc(in dept_name varchar,
out d_count integer)
begin
select count(*) into d_count
from instructor
where instructor.dept_name= dept_count_proc.dept_name
end
```

Procedures can be invoked either from an SQL procedure or from embedded SQL by the call statement:

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

Example 2: The procedure p has two IN parameters, one OUT parameter, and one IN OUT parameter. The OUT and IN OUT parameters are passed by value (the default). The anonymous block invokes p twice, with different actual parameters. Before each invocation, the anonymous block prints the values of the actual parameters. The procedure p prints the initial values of its formal parameters. After each invocation, the anonymous block prints the values of the actual parameters again.

```
CREATE OR REPLACE PROCEDURE p (
a PLS_INTEGER, -- IN by default
b IN PLS_INTEGER,
c OUT PLS_INTEGER,
d IN OUT BINARY_FLOAT
) IS
BEGIN
-- Print values of parameters:
DBMS_OUTPUT.PUT_LINE('Inside procedure p:');
DBMS_OUTPUT.PUT('IN a = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(a), 'NULL'));
DBMS_OUTPUT.PUT('IN b = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(b), 'NULL'));
DBMS_OUTPUT.PUT('OUT c = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(c), 'NULL'));
DBMS_OUTPUT.PUT_LINE('IN OUT d = ' || TO_CHAR(d));
-- Can reference IN parameters a and b,
-- but cannot assign values to them.
c := a+10; -- Assign value to OUT parameter
d := 10/b; -- Assign value to IN OUT parameter
END;
/
DECLARE
aa CONSTANT PLS_INTEGER := 1;
bb PLS_INTEGER := 2;
cc PLS_INTEGER := 3;
dd BINARY_FLOAT := 4;
ee PLS_INTEGER;
ff BINARY_FLOAT := 5;
BEGIN
DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');
DBMS_OUTPUT.PUT('aa = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));
DBMS_OUTPUT.PUT('bb = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));
DBMS_OUTPUT.PUT('cc = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));
DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));
```

```
p (aa, -- constant
bb, -- initialized variable
cc, -- initialized variable
dd -- initialized variable
);
DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');
DBMS_OUTPUT.PUT('aa = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));
DBMS_OUTPUT.PUT('bb = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));
DBMS_OUTPUT.PUT('cc = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));
DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));
DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');
DBMS_OUTPUT.PUT('ee = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));
DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));
p (1, -- literal
(bb+3)*4, -- expression
ee, -- uninitialized variable
ff -- initialized variable
);
DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');
DBMS_OUTPUT.PUT('ee = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));
DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));
END;
/
```

Output:

```
Before invoking procedure p:
aa = 1
bb = 2
cc = 3
dd = 4.0E+000
Inside procedure p:
IN a = 1
IN b = 2
OUT c = NULL
IN OUT d = 4.0E+000
```

After invoking procedure p:
aa = 1
bb = 2
cc = 11
dd = 5.0E+000
Before invoking procedure p:
ee = NULL
ff = 5.0E+000
Inside procedure p:
IN a = 1
IN b = 20
OUT c = NULL
IN OUT d = 5.0E+000
After invoking procedure p:
ee = 11
ff = 5.0E-001

EXERCISE:

Procedures:

1. Write a procedure to display a message "Good Day to You".

2. Based on the University Database Schema in Lab 2, write a procedure which takes the dept_name as input parameter and lists all the instructors associated with the department as well as list all the courses offered by the department. Also, write an anonymous block with the procedure call.
3. Based on the University Database Schema in Lab 2, write a Pl/Sql block of code that lists the most popular course (highest number of students take it) for each of the departments. It should make use of a procedure course_popular which finds the most popular course in the given department.
4. Based on the University Database Schema in Lab 2, write a procedure which takes the dept-name as input parameter and lists all the students associated with the department as well as list all the courses offered by the department. Also, write an anonymous block with the procedure call.

Functions:

5. Write a function to return the Square of a given number and call it from an anonymous block.

6. Based on the University Database Schema in Lab 2, write a Pl/Sql block of code that lists the highest paid Instructor in each of the Department. It should make use of a function department_highest which returns the highest paid Instructor for the given branch.

Packages:

7. Based on the University Database Schema in Lab 2, create a package to include the following:
   a) A named procedure to list the instructor_names of given department
   b) A function which returns the max salary for the given department
   c) Write a PL/SQL block to demonstrate the usage of above package components

Parameter Modes: IN, OUT, IN OUT

8. Write a PL/SQL procedure to return simple and compound interest (OUT parameters) along with the Total Sum (IN OUT) i.e. Sum of Principle and Interest taking as input the principle, rate of interest and number of years (IN parameters). Call this procedure from an anonymous block.

ADDITIONAL EXERCISE:

Procedures:

1. Based on the Banking Schema provided in Appendix, create payment (loan_number, payment_number, payment_date, amount) table for the Bank database.
   (a) Write a procedure to find out loan amount issued and total payment done by a customer.
   (b) Write a procedure to find out total deposit and loan amount for the given branch.
2. Based on the Banking Schema provided in Appendix, write a procedure which takes the branch_name as input parameter and lists the names of all customers belonging to that branch. Also, write an anonymous block with the procedure call.

3. Based on the Banking Schema provided in Appendix, write a procedure to calculate the sales made to a particular Customer. (The customer id in the transaction file, item_transaction, must be selected, the quantity sold must be multiplied by the price which is in the item_master and this value must be accumulated for all records of that customer in the Transaction file).

Functions:

4. Based on the HR database schema in Appendix, write a function to return the net salary given the employee number in the employees table and call the same from an anonymous block.

5. Based on the University Database Schema in Lab 2, write a Pl/Sql block of code that lists the most popular course (highest number of students take it) for each of the departments. It should make use of a function course_popular which returns the most popular course in the given department.

Parameter Modes: IN, OUT, IN OUT

6. Based on the University Database Schema in Lab:2, write PL/SQL procedure that takes as input the department name (IN) and budget (IN OUT) gives a 10% hike and returns the new budget. Call this procedure from a function and print the Department Name and Old budget and New budget. Department (dept-name, building, budget)

7. Based on the University Database Schema in Lab 2, write a Pl/Sql block of code that lists the highest paid Instructor in each of the Department. It should make use of a named procedure department_highest which finds the highest paid Instructor for the given branch.

LAB NO.: 10 **Date:**

TRIGGERS

Objectives:

In this lab, student will be able to:

- Use triggers to enforce business logic in a database.
- Use Row triggers to enforce any constraint on a table.
- Understand the usage of "Instead Of" Triggers.

Triggers

A trigger is a named PL/SQL block that is stored in the database and run in response to an event that occurs in the database. The user can specify the event (insert, update or delete)

- whether the trigger fires before or after the event
- whether the trigger runs for each event or for each row affected by the event

Uses of Triggers

1. Enforce referential integrity constraints when child and parent tables are on different nodes of the distributed database.
2. Prevent DML operations on a table after regular business hours.
3. Modify table data when DML statements are issued against views.
4. Enforce complex business rules that you cannot define with constraints.

Triggers vs Constraints

- A trigger applies on new data only while constraint holds true for the entire data Eg.: a trigger can prevent a DML statement from inserting a NULL but the column couldhave null prior.
- Constraints are easier to write and less error prone.
- But triggers can enforce complex business rules that constraints cannot.

Syntax:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE(AFTER) INSERT OR UPDATE OF list of columns
OR DELETE on tablename
FOR EACH ROW
begin
CASE
WHEN INSERTING THEN
//set of actions
```

```
WHEN UPDATING(column1) THEN
//set of actions
WHEN DELETING THEN
//set of actions
END CASE;
END;
/
```

:NEW AND :OLD

- The trigger is fired when DML operations (INSERT, UPDATE, and DELETE statements) are performed on the table.
- The user can choose what combination of operations should fire the trigger.
- Because the trigger uses BEFORE keyword, it can access the new values before they go into the table, and can change the values if there is an easily-corrected error

by assigning to :NEW.column_name.

- NEW.attribute is accessible on insertion and updation.
- OLD.attribute is accessible on deletion and updation.

Example: A trigger to insert the records into the emp_delete table before deleting the records from the employee table, based on the HR schema provided in the appendix.

```
CREATE OR REPLACE trigger emp_trigger
BEFORE DELETE ON employee
FOR EACH ROW
BEGIN
insert into emp_delete values(:OLD.emp_id,
:OLD.emp_name,
:OLD.emp_sal);
END;
/
```

Output:

```
delete from employee where emp_id=1101;
When you execute the above statement in the terminal the same
record is inserted in the emp_delete table.
```

INSTEAD OF TRIGGERS:

- It is created to perform data manipulation of views which cannot be performed ingeneral.

- We cannot perform deletion or insertion on views if the view contains aggregate functions or joins.
- An INSTEAD OF trigger is the only way to update a view that is not inherently updatable.
- Consider a view which is joined on tables A and B. A delete operation on the view is equivalent to deletion on A separately and B separately.

Syntax:

```
Create or replace trigger trigger_name
INSTEAD OF DELETE on view_name
FOR EACH ROW
BEGIN
//Set of Actions
END;
```

EXERCISE:

Row Triggers

1. Based on the University database Schema in Lab 2, write a row trigger that records along with the time any change made in the Takes (ID, course-id, sec-id, semester, year, grade) table in log_change_Takes (Time_Of_Change, ID, courseid,sec-id, semester, year, grade).

2. Based on the University database schema in Lab: 2, write a row trigger to insert the existing values of the Instructor (ID, name, dept-name, salary) table into a new table Old_ Data_Instructor (ID, name, dept-name, salary) when the salary table is updated.

Database Triggers

3. Based on the University Schema, write a database trigger on Instructor that checks the following:
   - The name of the instructor is a valid name containing only alphabets.
   - The salary of an instructor is not zero and is positive
   - The salary does not exceed the budget of the department to which the instructor belongs.
4. Create a transparent audit system for a table Client_master (client_no, name, address, Bal_due). The system must keep track of the records that are being deleted or updated. The functionality being when a record is deleted or modified the original record details and the date of operation are stored in the auditclient (client_no, name, bal_due, operation, userid, opdate) table, then the delete or update is allowed to go through.

Instead of Triggers

5. Based on the University database Schema in Lab 2, create a view Advisor_Student which is a natural join on Advisor, Student and Instructor tables. Create an INSTEAD OF trigger on Advisor_Student to enable the user to delete the corresponding entries in Advisor table.

Additional Exercise:

Row Triggers

1. Write a row trigger to update the Bal_Stock in item_master when a new transaction is entered in the item_transaction

Database Triggers

2. Based on the Banking Schema provided in Appendix; write a database trigger on Account that checks the following:
   - The account for which transaction is performed is a valid account number
   - The transaction amount is not zero and is positive
   - In case of withdrawal, the amount does not exceed balance for that account number

LAB NO.: 11 **Date:**

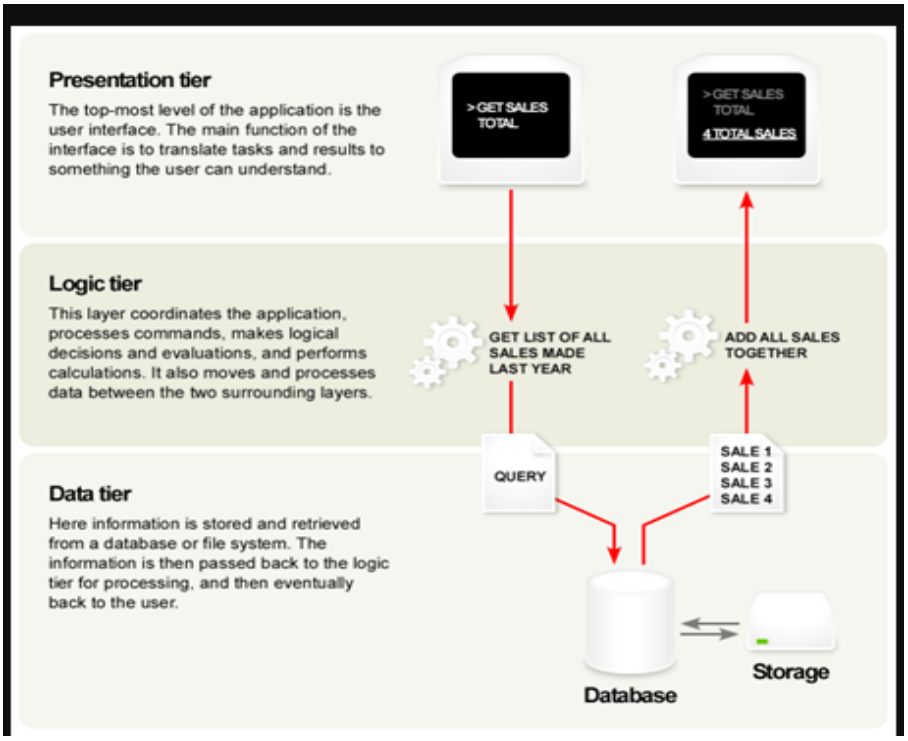
A SIMPLE APPLICATION USING ORACLE DB

Objectives:

- To create a Java based database application using Oracle DB.

Three-tier architecture is an architectural deployment style that describe the separation of functionality into layers with each segment being a tier that can be located on a physically separate computer

Using this architecture, the software is divided into 3 different tiers: ***Presentation tier, Logic tier, and Data tier***. Each tier is developed and maintained as an independent tier.



JDBC

An API that lets the user access virtually any tabular data source from the Java programming language". A tabular data source is virtually any data source, from relational databases to spreadsheets and flat files.

Basic steps to use a database in Java:

1. Establish a connection
2. Create JDBC Statements
3. Execute SQL Statements
4. GET ResultSet
5. Close connections

Example :

The five basic steps are illustrated using an example.

1. Establish a connection

- import java.sql.*;
- Load the vendor specific driver

Class.forName("oracle.jdbc.driver.OracleDriver");Dynamically loads a driver class, for Oracle database

- Make the connection
  Connection con = DriverManager.getConnection("jdbc:oracle:thin:@oracle-prod:1521:OPROD", username, passwd);
- Establishes connection to database by obtaining a Connection object

2. Create JDBC statement(s)

- Create a Statement object for sending SQL statements to the database
  Statement stmt = con.createStatement() ;

3. Executing SQL Statements

- String createLehigh = "Create table Lehigh " + "(SSN Integer not null, Name VARCHAR(32), " + "Marks Integer)";
  stmt.executeUpdate(createLehigh);

- String insertLehigh = "Insert into Lehigh values" +"(123456789,abc,100)";

  stmt.**executeUpdate**(insertLehigh);

4. Get ResultSet

- String queryLehigh = "select * from Lehigh";

  ResultSet rs = Stmt.executeQuery(queryLehigh);

  while (rs.next()) {

  int ssn = rs.getInt("SSN");

  String name = rs.getString("NAME");

  int marks = rs.getInt("MARKS");

  }

5. Close connection

- stmt.close();
- con.close();

Transactions and JDBC

JDBC allows SQL statements to be grouped together into a single transaction. Transaction control is performed by the Connection object, default mode is auto-commit, I.e., each sql Statement is treated as a transaction. We can turn off the auto-commit mode with

con.setAutoCommit(false);

And turn it back on with

con.setAutoCommit(true);

Once auto-commit is off, no SQL statement will be committed until an explicit is invoked con.commit(); At this point all changes done by the SQL statements will be made permanent in the database.

Handling Errors with Exceptions

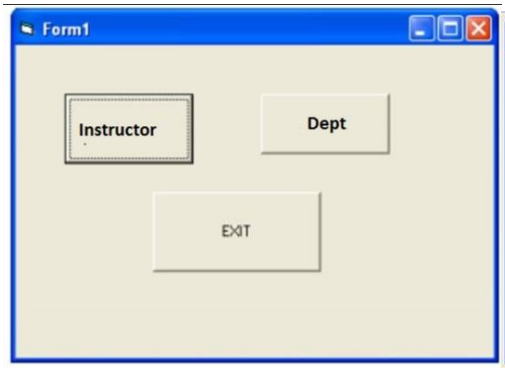
Programs should recover and leave the database in a consistent state. If a statement in the try block throws an exception or warning, it can be caught in one of the corresponding catch statements. E.g., the user could rollback the transaction in a catch { …} block or close database connection and free database related resources in finally {…} block.

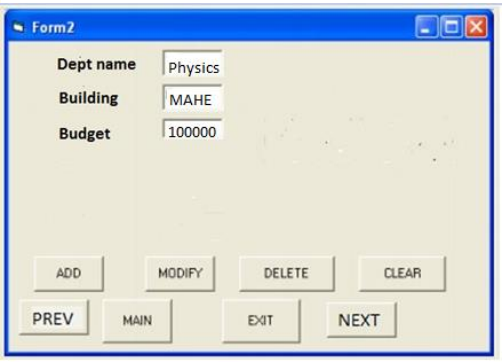
LAB EXERCISES:

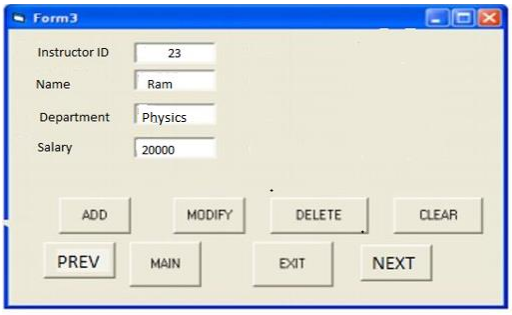
1. Consider the following tables of University database:

Department (dept-name, building, budget)
Instructor(ID, name, dept-name(Foreign key), salary)

Build the application containing the following screens which should cover the functionalities of Insert, Update, Delete, etc. Handle "No Database Connectivity" exception.







LAB NO.: 12 **Mini Project (Phase II)** **Date:**

Objectives:

- To develop an application that uses database to manage data.

Guidelines: Students need to submit the final report and demonstrate the application developed as part of mini project.

Final Report Format:

- Title, Team Members
- Abstract (with modifications, if any)
- Problem Statement (covering both data and functional requirements)
- ER Diagram & Relational Tables along with sample data
- DDL Commands to create table with necessary integrity constraints
- List of SQL Queries
- UI Design (include screenshots)
- PL/SQL Procedures / Functions / Triggers
- Java code for functional design (DB connectivity, PL/SQL Procedure/Function call /data access)
- References

Evaluation Guidelines:

Sl. No.	Topic	Marks
1	Synopsis, Abstract, Problem Statement	2
2	Design: ER Diagram, Normalised Tables	5
3	Demo: Basic Queries, Complex Queries, Procedures, Triggers, DB Connectivity	8
4	UI Design	2
5	Report	3

REFERENCES:

1. Silberschatz, Korth, Sudarshan, "Database System Concepts", McGrawHill, 6th Edition.
2. Ivan Bayross, "SQL, PL/SQL" 2nd/3rd Edition, BPB Publications.
3. www. docs.oracle.com, "PL/SQL Language Reference", Oracle Corp.
4. G. Reese, "Database Programming With JDBC And Java", O'REILLY, Second edition, 2000.
5. *http://www.learndb.com/databases*

APPENDIX:

BANKING SCHEMA

branch = (branch_name, branch_city , assets)

customer = (customer_id , customer_name , customer_street, customer_city)

loan = (loan_number, amount)

account= (account_number, balance)

employee = (employee_id,employee_name, telephone_number, start_date)

dependent_name = (employee_id, dname)

account_branch = (account_number, branch_name)

loan_branch = (loan_number, branch_name)

borrower = (customer_id, loan_number)

depositor = (customer_id, account_number)

cust_banker = (customer_id, employee_id, type)

works_for= (worker_employee_id, manager_employee_id)

payment = (loan_number, payment_number, payment_date, payment_amount)

savings_account= (account_number, interest_rate)

checking_account = (account_number, overdraft_amount)