Q-1. Design a database and Create required tables for e.g. Bank, College database.

Ans) the Syntax of Creating database is:

Create database database_name;

College database Schema:

1. Student Table:
   * Student_id (Primary key)
   * first_name
   * Last_name
   * date_of_birth
   * address
   * email
   * Phone_number.

2. Course Table:
   * Course_id (Primary key)
   * Course_name
   * department
   * Credits

3. Enrollments Table:
   * enrollment_id (Primary key)
   * Student_id (foreign key referencing students)
   * Course_id (foreign key referencing Courses)
   * enrollment_date.

4. Grades Table:
   * Grades_id (Primary key)
   * enrollment_id (foreign key referencing Enrollments)
   * Grade (e.g., A, B, C)
   * exam_date

Notes:
* The database, table have a primary key that uniquely identifies each record.
* foreign keys establish relationships between tables.
* Relationships:
  * In the College database, a Student Can enroll in multiple Courses, Creating a one to many relationship between students and Enrollments.
  * Enrollments is a junction table Connecting Students and Courses in a many-to-many relationship.

Q=2. Apply the Constraints like primary key, foreign key, NOT NULL to the tables.

Ans. For Using previous database of College.

1. Students Table!

```
Create Table Students (
    Student_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    Last_name VARCHAR(50) NOT NULL,
    date_of_birth DATE NOT_NULL,
    address VARCHAR (100),
    email VARCHAR(100) NOT NULL,
    phone_number VARCHAR (20) NOT NULL
);
--- INSERT SOME VALUES.
INSERT INTO Students VALUES (101, 'Ram', 'Thakur', '2003', 'mathura',
                             'King@gmail.com', 7505441090');
INSERT INTO students Values (102, 'Sahil', 'Ahmad', '2003', 'Aligarh'
                             'Sahil@gmail.com', '12345678');
INSERT INTO students Values (103, 'Vidya', 'Tiwari', '2002', 'bihar',
                             'Vidya@gmail.com', '23456789');
--- Perform some operations
SELECT * FROM Students;
```

Bank. Database Schema:

1. Customer Table:
   * Customer_id (Primary key)
   * first_name
   * last_name
   * date_of_birth
   * address
   * email.
   * Phone_number.

2. Account table:
   * account_~~table~~ number (Primary key)
   * Customer_id (Foreign key referencing Customers)
   * account_type (e.g. saving, checking)
   * balance
   * date_opened.

3. Transactions Table:
   * transaction_id (Primary key)
   * account_number (Foreign key referencing accounts)
   * transaction_type (e.g. deposit, withdrawal, transfer)
   * account
   * transaction_date.

1. Customer table:

```
CREATE TABLE Customers (
    Customer_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    date_of_birth DATE NOT NULL,
    address VARCHAR(100),
    email VARCHAR(100) NOT NULL,
    Phone_number VARCHAR(20) NOT NULL
);
-- INSERT Value --
INSERT INTO Customers values(101, 'Ram', 'thakur', '09/02/2003',
                             'mathura', 'Ram@.com', '7505441090');
INSERT INTO Customers values(102, 'Vidya', 'Tiwari', '2002', 'bihar',
                             'Vidya@.com', '1234567');
INSERT INTO Customers values('103', 'Sahil', 'Ahmad', '2003', 'Aligarh',
                             'Sahil@.com', '56789');
SELECT * FROM Customers;
```

1. Account Table:

```
CREATE TABLE Accounts (
    account_number INT PRIMARY KEY,
    Customer_id INT,
    account_type VARCHAR(20) NOT NULL,
    balance DECIMAL (12,2),
    date_opened DATE,
    FOREIGN KEY (Customer_id) REFERENCES Customers(customer_id)
);
```

```
-- INSERT VALUE
INSERT INTO Accounts Values (1111, '101', 'Saving', '9999.0', '6/2018');
INSERT INTO Accounts Values (2222, '102', 'Checking', '4444', '3/2009');
INSERT INTO Accounts values (3333, '103', 'Saving', '5555', '5/2005');

SELECT * FROM Accounts;
```

1. Transactions Table:

```
CREATE TABLE Transactions (
    transaction_id INT PRIMARY KEY,
    account_number INT,
    transaction_type VARCHAR(20) NOT NULL
    amount DECIMAL (12,2) NOT NULL,
    transaction_date DATE,
    FOREIGN KEY (account_number) REFERENCES Accounts (account_number)
);
```

```
-- INSERT values
Insert INTO Transactions Values (201, '1111', 'deposit', '5000', '15-jan');
INSERT Into Transactions Values (202, '2222', 'withdrawal', '2000', '14-jan');
insert into Transactions Values (203, '3333', 'transfer', '1000', '14-jan');

--- fetch INFO
SELECT * FROM Transaction;
```

Q-3. Write a SQL Statement for implementing ALTER, UPDATE and DELETE.

Ans: 1. ALTER Statement:

Example: Add a new Column to the Customers table in the bank database.

```
ALTER TABLE Customers
ADD COLUMN Country VARCHAR(50);
```

2. UPDATE Statement:

Example: Update the email of a specific Customer in the Customers table.

```
UPDATE Customers
SET email = 'newemail@example.com'
WHERE Customer_id = 101;
```

3. DELETE Statement:

Example: Delete a specific transaction from the Transactions table.

```
DELETE FROM Transactions
WHERE transaction_id = 201;
```

These examples are very basic, and you should tailor them to fit the specific requirements of your database. Always be cautious when using UPDATE and DELETE statements, especially without a WHERE clause, as they can modify or delete multiple records.

Q=4. Write the queries to implement the joins.

Ans:- 1. INNER JOIN:

Example: Retrieve information about Customers and their Accounts.

```
SELECT Customers.Customer_id, Customers.first_name,
    Customers.Last_name, Accounts.account_number,
    Accounts.account_type, Accounts.balance.
FROM Customers
INNER JOIN Accounts ON Customers.Customer_id = Accounts.
                                    Customer_id;
```

2. LEFT JOIN (or LEFT OUTER JOIN):

Example: Retrieve all students and their enrollments, including those who have not enrolled in any Course.

```
SELECT Students.student_id, Students.first_name, Students.last_name,
    Enrollments.Course_id, Enrollments.enrollment_date
FROM Students
LEFT JOIN Enrollments ON Students.student_id = Enrollments.student_id
```

3. RIGHT JOIN (or ~~LEFT~~ RIGHT OUTER JOIN):

Example: Retrieve all Courses and their enrollments, including Courses with no enrollments.

```
SELECT Courses.course_id, Courses.course_name, Enrollments.
    Student_id, Enrollments.enrollment_date
FROM Courses
RIGHT JOIN Enrollments ON Courses.Course_id = Enrollments.
                                    Course_id;
```

4. FULL JOIN (or FULL OUTER JOIN) - Note:-

Not all database systems support FULL JOIN.

Example:- Retrieve all Customers and their transactions, including those without any transactions.

```
SELECT Customers.customer_id, Customers.first_name, Customers.last_name,
    Transactions.transaction_id, Transactions.transaction_type,
        Transactions.amount
FROM Customers
FULL JOIN Transactions ON Customers.customer_id = Transactions.
                                                    Customer_id;
```

5. SELF JOIN:

Example: Find pairs of Customers who have the same email address.

```
SELECT A.customer_id AS customer_id, A.first_name AS Customer1_
    first_name, A.last_name AS Customer1_Last_name,
  B.customer_id AS Customer2_id, B.first_name AS
  Customer2_first_name, B.last_name AS Customer2_Last_name
FROM Customers A
JOIN Customers B ON A.email = B.email AND
    A.customer_id < B.customer_id;
```

You may need to adapt them based on the specific Columns and relationships in your databases. Always ensure that you join tables on appropriate Column to maintain data integrity.

Q=5: Write the queries for implementing the following functions: MAX(), MIN(), AVG(), COUNT()

Ans: SQL queries that implement the MAX(), MIN(), AVG(), and COUNT() aggregate functions.

I'll use the bank and College databases for illustration.

1. MAX(): Example: Find the maximum balance in the Accounts table.

```
SELECT MAX(Balance) AS max_balance
FROM Accounts;
```

2. MIN(): Example: Find the minimum balance in the Accounts table.

```
SELECT MIN(balance) AS min_balance
FROM Accounts;
```

3. AVG(): Ex: Calculate the average balance in the Accounts table.

```
SELECT AVG(balance) AS avg_balance
FROM Accounts;
```

4. COUNT(): Ex: Count the number of transactions in the Transactions table.

```
SELECT COUNT(*) AS transaction_count
FROM Transactions;
```

Example: Count the number of students in the Students table.

```
SELECT COUNT(*) AS student_count
FROM Students;
```

Q=6. write the queries to implement the concept of integrity Constraints

Ans: Integrity Constraints In a database help maintain the accuracy and Consistency of the data. Common integrity Constraints include PRIMARY KEY, FOREIGN KEY, UNIQUE. NOT NULL, CHECK, and DEFAULT.

Example of SQL queries that implement these integrity Constraints.

1. PRIMARY KEY and FOREIGN KEY: Example:

Create tables with PRIMARY KEY and FOREIGN KEY Constraints.

```
-- Create Customers table with PRIMARY KEY Constraints
CREATE TABLE Customers (
    Customer_id INT PRIMARY KEY,
    First_name VARCHAR(50) NOT NULL,
    Last-name VARCHAR(50) NOT NULL,
    --Other Columns----

    );
-- Create Accounts table with FOREIGN KEY Constraint.
CREATE TABLE Accounts(
    account_number
    Customer_id
    account_type
    balance
    date_opened
    FOREIGN KEY (Customer_id) REFERENCES Customers(customer_id)
    );
```

2. UNIQUE Constraint:

Example: Ensure that email addresses in the Customers table are Unique.

```
-- add UNIQUE Constraints to the email Column.
ALTER TABLE Customers
ADD CONSTRAINT Unique_email UNIQUE (email);
```

3. NOT NULL Constraints:

Example: Ensure that the date_of_birth Column in the students table Cannot be NULL.

```
-- Add NOT NULL Constraint to the date_of_birth Column
ALTER TABLE Students
MODIFY COLUMN date_of_birth DATE NOT NULL;
```

4. CHECK Constraint:

Example: Ensure that the balance in the Accounts table is always positive.

```
-- Add CHECK Constraint to the balance Column.
ALTER TABLE Accounts
ADD CONSTRAINT positive_balance CHECK (balance >= 0);
```

5. DEFAULT Constraint:

Example: Set a default Value for the department Column in the Courses table.

```
-- Add DEFAULT Constraint to the department Column
ALTER TABLE Courses
MODIFY COLUMN department VARCHAR(50) DEFAULT
    'Not Specified';
```

The exact Syntax may very depending on the database DBMS you are using (e.g, MySQL, PostgreSQL, SQL Server).

Q=7. Perform the following operations for demonstrating the insertion, updation and deletion using the referential integrity Constraints.

Ans → In this example, we have two tables: 'Customers' and 'Accounts' with a foreign key relationship between them.

1. Insertion with Referential Integrity:

Example: Insert a new Customer and account, ensuring that the foreign key relationship is maintained.

```
-- Insert a new Customer
INSERT INTO Customers(customer_id, first_name, last_name, date_of_birth,
                      address, email, phone_number)
VALUES (104, 'Love', 'Lesh', '04/05/2002', 'bulandshahar' 'love@.com',
                                                '444-123-');
-- Insert a new account for the Customer
INSERT INTO Accounts( account_number, Customer_id, account_type,
                                balance, date_opened)
VALUES ( 4444, '104', 'Savings', 1000.00, '2024-01-15');
```

Here the 'Customer_id' in the 'Accounts' table must reference an existing 'customer_id' in the 'Customers' table to maintain referential integrity.

2. Updating with Referential Integrity:

Example: Update the 'customer_id' of an existing account while maintaining the referential integrity.

```
-- Update the Customer_id of an existing account.
UPDATE Accounts
SET Customer_id = 104
WHERE Account_number = 3333;
```

Ensure that the new 'Customer_id' (102 in this case) exists in the 'Customers' table to maintain referential integrity.

3. Deletion with Referential Integrity:

Example: Delete a Customer and their associated accounts, maintaining referential integrity.

```
-- Delete a Customer and their associated accounts.
DELETE FROM Customers
WHERE Customer_id = 101;
```

This deletion will only succeed if there are no foreign key references to the Customer being deleted. If there are associated accounts, you may need to either delete the related accounts first or handle the deletion cascade using the appropriate settings in your database system.

The specific syntax and handling of constraints may vary depending on your database management system.

Q=8 ⇒ Creation of Views, Synonyms, Sequence

Ans ⇒ 1. Creation of a View:

A View is a virtual table based on the result of a SELECT query. it allows you to present a subset of the data or perform. Complex operations without modifying the Underlying tables

Example: Create a View to show Customers with their account balances.

```
CREATE VIEW CustomerBalances AS
SELECT Customers.Customer_id, Customers.first_name,
    customers.Last_name, Accounts.balance
FROM Customers
JOIN Accounts ON Customers.customer_id = Accounts.customer_id;
```

you Can then query the View like a regular table:

```
SELECT * FROM CustomerBalances;
```

2. Creation of a Synonym:

A synonym is an alternative name for a table, View, Sequence, or other database objects. It provides a Convenient alias for object names, simplifying query writing.

Example: Create a Synonym for the Customers table.

```
CREATE SYNONYM Cust FOR Customers;
```

Now you Can use 'cust' instead of 'Customers' in queries.

```
SELECT * FROM Cust;
```

3. Creation of a Sequence:

A Sequence is a database object that generates unique numbers. It is often used for creating primary key values.

Example: Create a Sequence to generate unique Customer IDs.

```
CREATE SEQUENCE customer_id_seq
START WITH 1000
INCREMENT BY 1
NO CYCLE;
```

you can then use the sequence to generate unique IDs during an INSERT operation:

```
INSERT INTO Customers( Customer_id , first_name, last_name)
VALUES ( customer_id_seq.NEXTVAL, 'RAM', 'Thakur');
```

These examples demonstrate the basic syntax for creating views, synonyms, and sequences. The specific syntax may very depending on the database management system you are using (e.g., MySQL, PostgreSQL, SQL Server, Oracle).

Q=9. Creation of indexes, save point.?

Ans=> Through the Creation of indexes and the usage of Savepoints in a database. I'll use the Bank database for these examples.

1. Creation of Indexes:

Indexes are database objects that improve the speed of data retrieval operations on database tables.

Example: Create an index on the 'email' column in the 'Customers' table

```
CREATE INDEX idx_customers_email
ON Customers (email);
```

This index will improve the performance of queries that involve searching or filtering based on the 'email' column.

2. Usage of Savepoints:

Savepoint allow you to set points within a transaction to which you can later roll back. This can be useful for creating more granular rollback points within a transaction.

Example: Use a Savepoint within a transaction.

```
-- Start a transaction
BEGIN;
-- Insert a new Customer.
INSERT INTO Customers (Customer_id, first_name, last_name, email)
VALUES (101, 'Ram', 'Thakur', 'Ram@.com');
-- Set a Savepoint
SAVEPOINT after_insert;
```

-- Attempt to insert an account for the Customer (this may fail due to a unique constraints violation)

~~INSERT INTO Customers (customer_id, first_name, last_name, email~~
~~VALUES (101, 'Sahid', 'Ahmad', 'sahid@.com');~~

```
INSERT INTO Accounts (account_number, customer_id, account_type,
                      balance, date_opened)
VALUES (1111, 101, 'Savings', 500.00 '2024-01-20');
```

-- If an error occurs, roll back to the Savepoint

```
ROLLBACK TO after_insert;
```

--- Continue with the transaction

-- (you can Commit or rollback the entire transaction as needed)

-- Commit the transaction

```
COMMIT;
```

In this example, the Savepoint 'after_insert' is set after the successful insertion of a Customer. If the subsequent insertion of an account fails, the transaction can be rolled back to the savepoint, allowing you to handle errors more gracefully.

Q=10. Study of PL/SQL block, PL/SQL block to satisfy some Conditions by accepting input from the User.

Ans⇒ PL/SQL (Procedural language / structured Query Language) is a programming language designed specifically for managing and manipulating data in Oracle database. PL/SQL blocks are units of code that can contain declarations, executable statements, and exception handlers. Here's an example of a simple PL/SQL block that accepts input from the user and performs actions based on certain conditions:

```
-- PL/SQL block to satisfy some Conditions by accepting input from
      the User

DECLARE
    user_input VARCHAR2(50);
BEGIN
   -- Accept input from the user
   DBMS_OUTPUT.PUT_LINE('Enter Something: ');
   ACCEPT user_input CHAR(50) FORMAT 'A50' DEFAULT 'Default Value';
   -- Check Conditions and perform actions.
   IF user_input IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('you did't enter anything.');
   ELSIF LENGTH(user_input) > 10 THEN
      DBMS_OUTPUT.PUT_LINE('your input is longer than 10 characters.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('your input is acceptable.');
   END IF;
END;
/
```

IN this example:

1. DECLARE' is used to declare variables.
2. BEGIN' marks the beginning of the executable section of the block.

3. 'DBMS_OUTPUT.PUT_LINE' is used to display messages on the Screen.

4. 'Accept' is used to take input from the user. Note that the 'ACCEPT' statement is commonly used in SQL*PLUS, a Command-line tool for Oracle Database.

The PL/SQL block prompts the user to enter something, then checks the conditions and displays messages accordingly. You can run this block in an Oracle database environment that supports PL/SQL.

Please note that the usage of 'ACCEPT' for user input is specific to SQL*Plus.

Q=11. PL/SQL block that handles all type of exceptions.

Ans: In PL/SQL, you can use exception handling to manage errors that might occur during the execution of a block of code. Here's an example of a PL/SQL block that handles various type of exceptions:

```
-- PL/SQL block that handles all types of exceptions.

DECLARE
    user_input NUMBER;
BEGIN
    --Accept input from the User
    DBMS_OUTPUT.PUT_LINE("Enter a Number:");
    ACCEPT user_input NUMBER FORMAT '99999999999999999' DEFAULT 0;

    -- Exception handling
    BEGIN
        -- Perform some operations that might raise exceptions.
        IF user_input = 0 THEN
            -- Divide by zero exception
            DBMS_OUTPUT.PUT_LINE('Cannot divide by zero.');
            user_input := -- Avoid division by zero
        END IF;
        -- Perform a division operation
    --DBMS_OUTPUT.PUT_LINE('Caught a divide by zero exception.');
    DBMS_OUTPUT.PUT_LINE ('Result:' || (100/user_input));
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE ("Caught a divide by zero exception.");
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('An error occurred:' || SQLERRM);
END;

END;
/
```

In this example:

1. 'DECLARE' is used to declare variables.
2. 'BEGIN' marks the beginning of the executable section of the block.
3. 'DBMS_OUTPUT.PUT_LINE' is used to display messages on the screen.
4. 'ACCEPT' is used to take a numeric input from the user.
5. The Block Contains a nested 'BEGIN...END' section where some operations are performed that might raise exceptions.
6. The 'EXCEPTION' block catches specific exceptions ('ZERO_DEVIDE' in this case) and handles them.

The 'WHEN OTHERS' clause is used as a catch-all for any other exceptions that might occur.

Keep in mind that the 'WHEN OTHERS' clause should be used with caution, as it catches all exceptions, including those you might not have anticipated. It is a good practice to handle specific exceptions whenever possible.

Q=12. PL/SQL Procedures.

Ans⇒ In PL/SQL, a procedure is a subprogram that performs a specific action. Procedures are similar to anonymous blocks but are stored in the database for reuse. They can have parameters, local variables, and perform operations on the database.

Here's a basic example of a PL/SQL procedure:

```
CREATE OR REPLACE PROCEDURE my_procedure (
    P_Parameter1 IN VARCHAR2,
    P_Parameter2 OUT NUMBER
    ) AS
BEGIN
    -- Procedure body.
    DBMS_OUTPUT.PUT_LINE ('Parameter 1: ' || P_Parameter1);
    -- Perform some operation
    P_Parameter2 := 42;
    DBMS_OUTPUT.PUT_LINE ('Procedure executed successfully.');
END my_procedure;
/ my_procedure;
```

in this example:

* The procedure is named 'my_procedure'.
* It takes two parameters: "P_Parameter1' (input parameter) and P_Parameter2' (output parameter).
* The 'DBMS_OUTPUT.PUT_LINE' statements are used to display message.

You can then call the procedure from an anonymous block or another PL/SQL block:

```
DECLARE
    V_input VARCHAR(50) := 'Hello, world!';
    V_output NUMBER;
BEGIN
    -- Call the procedure.
    my_procedure (V_input, V_output);
    -- Display the output parameter
    DBMS_OUTPUT.PUT_LINE ('Output Parameter: ' || V_output);
END;
/
```

in this example, the 'my_procedure' is called with an input parameter 'v_input' and an output parameter 'v_output'. The output parameter is then displayed.

Key points about PL/SQL procedures:

1. Parameters: Procedure can have input, output, and input-output parameters. these parameters allow you to pass values into the procedure and receive values back.

2. DECLARE... BEGIN__END: The 'DECLARE' section is used to declare local variables and parameters, while the 'BEGIN__END' section contains the actual code of the procedure.

3. CREATE OR REPLACE PROCEDURE: this statement is used to create or update a procedure. The 'CREATE OR REPLACE' syntax allows you to modify an existing procedure without dropping it.

4. OUT Parameter: in this example, 'p_parameter2' is an OUT parameter, meaning it's used to return a value from the procedure

5. DBMS_OUTPUT.PUT_LINE: this is a debugging statement that outputs messages to the console. In a real-world scenario, you might use logging or other methods for handling outputs.

Q=13. PL/SQL triggers and functions.

Ans: PL/SQL triggers and functions are two distinct types of database objects in Oracle that serve different purposes.

PL/SQL Triggers: A trigger is a named program unit that is stored in the database and executed automatically when a particular event occurs. Triggers are often used to enforce business rules, data integrity, or to automate tasks.

Here's a Basic example of a PL/SQL trigger:

```
CREATE OR REPLACE TRIGGER my_trigger
BEFORE INSERT ON my_table
FOR EACH ROW
BEGIN
    -- Trigger body
    :NEW.created_date := SYSDATE;
END;
/
```

In this Example:

* The trigger is named 'my_trigger'.
* It is set to execute 'BEFORE INSERT' on the table 'my_table' for each row.
* The trigger automatically sets the 'created_date' column to the current date and time using the ':NEW' pseudo-record.

Triggers can also be defined for other events such as 'AFTER UPDATE', 'BEFORE DELETE', etc.

PL/SQL FUNCTIONS: A function is a named PL/SQL block that returns a value. Functions are designed to perform a specific task, and they can be used in SQL statements and other PL/SQL blocks.

Here's a simple example of a PL/SQL function:

```
CREATE OR REPLACE FUNCTION calculate_area (
    p_radius NUMBER
) RETURN NUMBER IS
    v_area NUMBER;
BEGIN
    -- Function body
    v_area := 3.14 * p_radius * p_radius;
    RETURN v_area;
END;
```

In this example

* The function is named 'Calculate_area'.
* It takes a Parameter 'p_radius' and returns the Calculated area.
* The 'RETURN' statement is used to return the result.

you can use the function in a SQL statement or another PL/SQL block like this:

```
DECLARE
    radius NUMBER := 5;
    area NUMBER;
BEGIN
    -- Call the function
    area := calculate_area (radius);
    DBMS_OUTPUT.PUT_LINE ('Area: ' || area);
END;
```

function are commonly used to encapsulate logic that returns a single value. They are frequently employed in SQL queries to perform calculations or retrieve data based on certain criteria.

Q=14. write a program to demonstrate %type and %rowtype attributes

Ans:- Create a Simple PL/SQL program to demonstrate the use of %TYPE and %ROWTYPE attributes in Oracle.

Using '%TYPE':

```
DECLARE
  -- Declare variables with %TYPE attribute
  v_employee_id employees.employee_id%TYPE;
  v_employee_name employees.employee_name%TYPE;
BEGIN
  -- Assign values to variables
  v_employee_id := 101;
  v_employee_name := 'Ram thakur';
  -- Display values
DBMS_OUTPUT.PUT_LINE('Employee ID:' || v_employee_id);
DBMS_OUTPUT.PUT_LINE('Employee Name:' || v_employee_name);
END;
```

in this '%TYPE' is used to declare variables 'v_employee_id' and 'v_employee_name' with the same data types as the respective column in the 'employees' table.

Using %ROWTYPE :

```
DECLARE
  -- Declare a record variable with %ROWTYPE attribute
  v_employee_record employees%ROWTYPE;
BEGIN
  -- Assign values to the record
  v_employee_record.employee_id := 102;
  v_employee_record.employee_name := 'Ram thakur';
  v_employee_record.salary := 50000;
  -- Display values
```

```
    DBMS_OUTPUT.PUT_LINE ('Employee ID: ' || v_employee_record.
                                    employee_id);
    DBMS_OUTPUT.PUT_LINE ('Employee Name: ' || v_employee_
                    record.employee_name);
    DBMS_OUTPUT.PUT_LINE ('Salary: ' || v_employee_record.
                                    Salary);
END;
/
```

In this example, '%ROWTYPE' is used to declare a record variable 'v_employee_record' with the same structure as the 'employees' table. This allows you to work with entire rows of data from the table.

Q=15. Write a PL/SQL block for greatest of three numbers Using IF and ELSEIF

Ans → A Simple PL/SQL Block that finds the greatest of three numbers Using 'IF' and 'ELSEIF' statements.

```
DECLARE
    num1 NUMBER := 25;
    num2 NUMBER := 45;
    num3 NUMBER := 30;
    greatest NUMBER;
BEGIN
    -- Find the greatest number
    IF num1 >= num2 AND num1 >= num3 THEN
        greatest := num1;
    ELSIF num2 >= num1 AND num2 >= num3 THEN
        greatest := num2;
    ELSE
        greatest := num3;
    END IF;
    -- Display the result
    DBMS_OUTPUT.PUT_LINE ('The greatest number is: ' || greatest);
END;
/
```

IN this example

* Three Variables ('num1', 'num2', and 'num3') are initialized with numeric values.
* The 'IF', 'ELSIF', and 'ELSE' statements are used to Compare the ~~result~~ values of the three numbers.
* The greatest number is assigned to the Variable 'greatest'.
* The result is displayed using the 'DBMS_OUTPUT.PUT_LINE' Statement.