DATABASE MANAGEMENT SYSTEMS – CSE/IT – II Year / IV Sem. – 22CS403

Manju S,
AP/CoE,
RTC,Coimbatore.

TOPICS

- **DATABASE PROGRAMMING**

Database programming with function calls, stored procedures - views – triggers. Embedded SQL. ODBC or JDBC connectivity with front end tools. Implementation using ODBC/JDBC and SQL/PSM, implementing functions, views, and triggers in PSQL

Stored Procedures

- **Stored procedures** in PostgreSQL are powerful tools that store sets of instructions directly within the database for execution
- Embedding programming in SQL
- They are a powerful feature for achieving complex data manipulation, batch processing, and business logic encapsulation directly within the database, which can significantly improve performance and reduce network traffic
- They enable efficient data manipulation and complex transaction handling.

Why Stored Procedures

- Imagine you have a complex task that needs to be done repeatedly, like updating multiple rows of data, calculating some values, filtering the data, aggregating the values, or performing several database operations together.
- Instead of writing the same code for a task every time you need it, you can create a stored procedure. Easy right?
- Think of stored procedures in PostgreSQL like functions in programming languages, both are used to avoid repetition and using the same piece of code for repeated use.

Stored Procedures : Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name(param1 data_type,param2 data_type,...)
AS $$
DECLARE
    variable1 data_type;
   variable2 data_type;
   -- More variables (if needed)
BEGIN
   -- Procedure body (PSQL statements)
END;
$$
LANGUAGE plpgsql;
```

Created Procedures can be called by,

CALL procedure_name(arguments)

Stored Procedures : Syntax

- **CREATE [OR REPLACE] PROCEDURE:**
  This SQL statement is used to create a procedure or replace an already existing procedure with the same name.
- **procedure_name:**
  The name of the procedure we are creating.
- **Procedure parameters:**
  If the procedure needs to take input values, we can list them inside the parentheses after the procedure name.
- **AS $$ ... $$:**
  The code block between the AS $$ and $$ keywords is the body of the stored procedure where we write the SQL statements and procedural code.

Stored Procedures : Syntax

- **DECLARE:**
  This keyword is used to define variables. Each variable should have a name and type.
- **BEGIN ... END:**
  The BEGIN keyword marks the beginning of the procedure's main code block and the END keyword marks the end of the procedure's main code block.
- **LANGUAGE plpgsql:**
  This line is used to specify the procedural language for the stored procedure. Here, we are using plpgsql, which stands for PL/pgSQL, the procedural language for PostgreSQL.

Stored Procedures : Schema

cid	name	balance
1	a	2000
2	b	2000
3	c	3000
4	d	4000

- Banking Schema : bankk
- Assume the schema and create a procedure for transferring amount from one person to other person.

Stored Procedures : Transfer

```
CREATE  PROCEDURE transfer_amount( sender_id INT, receiver_id INT,amount INT)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE bankk  SET balance = balance – amount  WHERE cid = sender_id;
    UPDATE bankk SET balance = balance + amount WHERE cid = receiver_id;
END;
$$;
```

```
CALL transfer_amount(1, 2, 1000);
```

cid	name	balance
1	a	**1000**
2	b	**3000**
3	c	3000
4	d	4000

Stored Procedures : Conditional Transfer

```
CREATE PROCEDURE transfer_amount(sender_id INT,receiver_id INT,amount INT)
LANGUAGE plpgsql
AS $$
BEGIN
        IF (SELECT balance FROM bankk WHERE cid = sender_id) - amount < 1000 THEN
        RAISE EXCEPTION 'Transfer cancel: sender's balance would drop below min. required balance.';
        END IF;
    UPDATE bankk SET balance = balance – amount WHERE cid = sender_id;
    UPDATE bankk SET balance = balance + amount WHERE cid = receiver_id;
END;
$$;
```

Error : Transfer cancel: sender's balance would drop below min. required balance

```
CALL transfer_amount(1, 2, 1000);
```

RAISE EXCEPTION Vs. RAISE NOTICE

- RAISE statement to generate log messages, show notices, or throw exceptions.
- Throwing an exception is a way to signal an error or an unexpected situation that requires attention.
- When an exception is raised, **it interrupts the normal execution of the procedure** or function, and control is passed to the nearest surrounding exception handler, if one exists.
- If there's no handler for the exception, the execution of the function or procedure is **terminated**, and the error is propagated to the caller
- But RAISE **Notice will not terminate the execution of function.**

Guess the Output ?!

- DO $$
- BEGIN
- RAISE NOTICE 'This is a notice. Execution will continue.';
- RAISE EXCEPTION 'This is an exception. Execution will stop.';
- RAISE NOTICE 'This notice will not be reached due to the exception.';
- END $$;

Why didn’t the last notice raise?!
Because the previous exception stopped the execution!

Output :
This is a notice. Execution will continue
This is an exception. Execution will stop

Stored Procedures : Repetitions : Loop

```
CREATE OR REPLACE PROCEDURE update_grade_with_loop()
LANGUAGE plpgsql
AS $$
DECLARE row record;
BEGIN
    ALTER TABLE bankk ADD COLUMN grade CHAR(1);
    FOR row IN SELECT id, balance FROM bankk LOOP
            IF row.balance > 2000 THEN
                    UPDATE bankk SET grade = 'A' WHERE cid = row.id;
            ELSE
                    UPDATE bankk SET grade = 'B' WHERE cid = row.id;
             END IF;
   END LOOP;
END;
$$;
```

- **Record is a** data type which is a flexible pseudo-type that can hold a row of data that may have any number of fields, and the fields can be of any data type.
- A **record** variable (row) is essentially a row or tuple of data whose structure is not rigidly defined

Call update_grade_with_loop()

Stored Procedures : Repetitions : Loop

- The stated procedure would add a new column called grade and fill in details for each row with either ‘A’ or ‘B’ depending on the balance.
- This repeats for until a table has rows in it.
- Output of the procedure would be

cid	name	balance	grade
1	a	1000	B
2	b	3000	A
3	c	3000	A
4	d	4000	A

To do!

- Write a procedure to read a string say your name and display a message stating “ Welcome #YourName”.
- Write a procedure to read two numbers and to display the sum, difference, product, quotient and modulo of two numbers.
- Write a procedure to read a number and to return cube and square of that number

Welcome Message Procedure !

```
CREATE PROCEDURE welcome_message(your_name TEXT)
LANGUAGE plpgsql
AS $$
BEGIN
    RAISE NOTICE 'Welcome %', your_name;
END;
$$;
```

Output :
NOTICE : Welcome Minion
CALL

```
Call welcome_message(“Minion”);
```

Calculator Procedure !

```
CREATE  PROCEDURE calculate_operations(a NUMERIC, b NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
    RAISE NOTICE 'Sum: %', (a + b);
    RAISE NOTICE 'Difference: %', (a - b);
    RAISE NOTICE 'Product: %', (a * b);
     IF b > 0 THEN
        RAISE NOTICE 'Quotient: %', (a / b);
        RAISE NOTICE 'Modulo: %', (a % b);
    ELSE
        RAISE NOTICE 'Quotient Modulo: undefined (division by zero)';
END IF;
END;
$$;
```

```
Call calculate_operations(12,3)
```

```
psql:commands.sql:16: NOTICE: Sum: 15
psql:commands.sql:16: NOTICE: Difference: 9
psql:commands.sql:16: NOTICE: Product: 36
psql:commands.sql:16: NOTICE: Quotient:
4.000000000000000 psql:commands.sql:16:
NOTICE: Modulo: 0
```

Returning Values : Return Statement

- How can we write a procedure to return values?
- **Procedures do not return a value directly**. They can use **OUT** parameters to output values.
- Parameters in procedure can be of 3 types
  - IN : Default mode, passes value to the procedure, cannot modify or return these values directly
  - OUT : they are used to store and return results, must be a variable, not a constant
  - INOUT : parameters combine the behaviors of both IN and OUT parameters. They allow you to pass values into the procedure and get modified values back

IN and OUT Parameters

CREATE PROCEDURE **calculate_square_cube(number IN NUMERIC, square OUT NUMERIC, cube OUT NUMERIC**)

LANGUAGE plpgsql

AS $$

BEGIN

square = number * number;

cube = number * number * number;

END;

$$;

```
CREATE PROCEDURE
square | cube
--------+------
16 | 64
(1 row)
```

call calculate_square_cube(4,**null,null**)

OUT parameters should be passed some values preferably “NULL” else procedure argumentmismatch error will occur

RATHINAM

IN and OUT Parameters

```
CREATE PROCEDURE  calculate_operations(a NUMERIC, b NUMERIC, OUT sum NUMERIC, OUT difference NUMERIC, OUT product NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
    sum = a + b;
    difference = a - b;
    product = a * b;
 END;
$$;
```

```
call calculate_operations(12,3,null,null,null)
```

```
CREATE PROCEDURE
sum | difference | product
 -----+------------+---------
15 | 9 | 36
(1 row)
```

Stored Procedures Vs. Functions in PSQL

Functions	**Stored Procedure**
The SELECT statements can have function calls.	The SELECT statements can never have procedure calls. Procedures are called by the statement “call”
A function would return the **returning value/control** to the code or calling function. Has return statement.	A procedure, on the other hand, would return the control, but **would not return any value** to the calling function or the code.

Functions and stored procedures

- Both stored procedures and functions are used to encapsulate SQL code for reuse
- Functions are designed to return a single value or a set of rows (table). They can be used in a SELECT statement, and their return type must be specified during creation
- Unlike functions, procedures cannot be called from within a SELECT statement or any SQL expression because they do not directly return a value.
- Procedures can accept input parameters (IN), change and return parameters (INOUT), and produce output parameters (OUT)

Function returning a single value

```
CREATE FUNCTION square(number numeric) RETURNS numeric
AS $$
BEGIN
    RETURN number * number;
END;
$$ LANGUAGE plpgsql;
```

```
Square
--------
25
(1 row)
```

```
select square(5)
```

Function returning a row of values

- CREATE table employee(name VARCHAR(100),id int, salary int);
- insert into employee values('abc',12,3000);
- insert into employee values(‘aaa',10,4000);

Name	id	Salary
abc	12	3000
aaa	10	4000

Function returning a row of values

```
CREATE OR REPLACE FUNCTION calculate_compensation(emp_id INT) RETURNS employee
AS $$
DECLARE result employee;
BEGIN
SELECT name, id, (salary + 1000) INTO result FROM employee WHERE id = emp_id;
RETURN result;
END;
$$ LANGUAGE plpgsql;
```

- Result is of type employee where employee is table created already

```
select calculate_compensation(12)
```

```
CREATE FUNCTION
calculate_compensation
-----------------------
(abc,12,4000)
(1 row)
```

Function to find the factorial of a number

```
CREATE OR REPLACE FUNCTION factorial_iterative(n BIGINT) RETURNS BIGINT AS $$
DECLARE
    result BIGINT := 1;
BEGIN
    IF n = 0 OR n = 1 THEN
        RETURN result;
    ELSE
        FOR i IN 2..n LOOP
            result = result * i;
        END LOOP;
        RETURN result;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
select factorial(5);
```

```
Factorial
 -----------
120 (1 row)
```

Function to find the factorial of a number (Recursive)

```
CREATE OR REPLACE FUNCTION factorial(n BIGINT) RETURNS BIGINT AS $$
BEGIN
    IF n = 0 THEN
        RETURN 1;
    ELSE
        RETURN n * factorial(n - 1);
    END IF;
END;
$$ LANGUAGE plpgsql ;
```

```
select factorial(5);
```

```
Factorial
 -----------
120 (1 row)
```

Views

- Views act as a proxy or virtual table created from the original table.
- Views simplify SQL queries and allow secure access to underlying tables.
- Views can present a **different logical structure of data**, independent of how the data is physically stored in the database.
- Views in DBMS can be visualized as **virtual tables** that are formed by original tables from the database
- The view has 2 two primary purposes:
  - Simplifying complex SQL queries.
  - Restricting users from accessing sensitive data.

Why views? Simplifying complex SQL queries!

- Sometimes SQL queries get very complicated by joins, Group By clauses, and other referential dependencies,
- So those Types of queries can be simplified to proxy data or virtual data which simplifies the queries

Why views? Restricting users !

- Suppose, the user needs only 2 columns of data, so instead of giving him access to the whole table in the database, the Database Administrator can easily create a virtual table of 2 columns that the user needs using the views.
- This will not give full access to the table and the user is only seeing the projection of only 2 columns and it keeps the database secure

Views

- CREATE VIEW view_name AS SELECT column1, column2, ... FROM table_name WHERE condition;
- **AS**: This keyword is used to separate the view name from the query that defines the view
- 3rd purpose of view : Logical data independence
  - If our applications use views, we can freely modify the structure of the base tables.
  - In other words, views enable you to create a layer of abstraction over the underlying tables

Views

Employee

id	name	department	salary
1	John Doe	Engineering	70000
2	Jane Smith	Marketing	55000
3	Mike Brown	Sales	50000
4	Linda White	Engineering	80000

- Create view highempsal as Select * from employee where salary>60000
- Select id,name,salary from highempsal;

id	**name**	**salary**
1	John Doe	70000
4	Linda White	80000

Views

- Insert into employee (5,’ashwin’, ‘Engineering’,56000);
- Insert into employee (6,’suganth’, ‘Engineering’,76000);

id	name	department	salary
1	John Doe	Engineering	70000
2	Jane Smith	Marketing	55000
3	Mike Brown	Sales	50000
4	Linda White	Engineering	80000
5	Ashwin	Engineering	56000
6	Suganth	Engineering	76000

Guess what happens ? Views

- Understand the order
  - Created table with 4 rows initially
  - Created a view as highempsal
  - Then inserted 2 more rows!
- Now lets try to see the output of the view created!
- Select * from highempsal;
- Guess what would be output?!

Views stored as what?! As Definitions only!

- The output would be

id	name	salary
1	John Doe	70000
4	Linda White	80000
6	Suganth	76000

- How? The data inserted right after creating the view is displayed ? Why?
- Because the Views are **stored as just query definitions** instead of being stored as tables/data.
- And hence views are called as **STORED SQL QUERIES**

Can views be used on multiple tables?

- Yes. Views can be created by combining tables as well.
- Assume 2 tables ‘employee’ having employee details and ‘manages’ table which has manager names of respective departments.

id	name	department	salary
1	John Doe	Engineering	70000
2	Jane Smith	Marketing	55000
3	Mike Brown	Sales	50000
4	Linda White	Engineering	80000
5	Ashwin	Engineering	56000
6	Suganth	Engineering	76000

department	Manager
Engineering	Charles Babbage
Marketing	Elon Musk
Sales	Tim Peter

Views from multiple tables

- Create view emp_manager as select e.id, e.name, e.department, e.salary , m.manager from employee e inner join manages m on e.department = m.department;
- Emp_manager combines 2 tables employee and manages.
- Select * from emp_manager;

id	name	department	salary	Manager
1	John Doe	Engineering	70000	Charles Babbage
2	Jane Smith	Marketing	55000	Elon Musk
3	Mike Brown	Sales	50000	Tim Peter
4	Linda White	Engineering	80000	Charles Babbage
5	Ashwin	Engineering	56000	Charles Babbage
6	Suganth	Engineering	76000	Charles Babbage

Deleting Views

- VIEWS in sql allows them to delete a view using the DROP statement.
- DROP VIEW view_name;
- E.g.:
- DROP VIEW highempsal;

To do! Construct views !

- **Assume the following schema for Moie DB**
  - Movies: MovieID, Title, ReleaseYear, DirectorID
  - Directors: DirectorID, FirstName, LastName
  - Genres: MovieID, GenreName
- Create a view that displays each movie's title along with the director's full name.
- Construct a view that shows movies released in the year 2010, including their title and release year.

Views for Movie DB

- CREATE VIEW MoviesWithDirectors AS

SELECT m.Title, d.FirstName, d.LastName FROM Movies m JOIN Directors d ON m.DirectorID = d.DirectorID;

- CREATE VIEW MoviesByYear AS SELECT Title, ReleaseYear FROM Movies WHERE ReleaseYear = 2010;

Have you experienced triggers ?

- Have you ever received a text message from your account holding banks when a transaction is performed with respect to your account? E.g. : credit or debit?
- Have you ever received a mail/ text when your mail account is logged in from other computing machines?
- Have you ever received texts/ alerts when you attempt to change your password of any social media platforms?

Triggers

- Triggers in SQL are special stored programs that are executed automatically in response to certain events or actions performed on a table, such as inserting, updating, or deleting data
- A trigger in SQL is a procedural code that is automatically executed in response to certain events on a specified table.

Why and when triggers?

- Automating data validation before or after changes to ensure data integrity.
- Synchronizing data across tables to maintain consistency across your database.
- Capturing audit trails by logging changes to data, such as inserts, updates, and deletes.
- Implementing complex business logic that requires actions to be taken automatically in response to data modifications.
- Cascading changes to maintain referential integrity beyond simple foreign key constraints.

Triggers in PSQL : Creation : 2 steps

- Creating a trigger in PostgreSQL involves a couple of distinct steps:
  - **defining a trigger function** that contains the logic to be executed, and
  - then creating the **actual trigger** that specifies when (i.e., at what event) and where (i.e., on which table) this function should be executed.

Creation of Triggers : Syntax : Step 1

- Define a function that PostgreSQL will call when the trigger event occurs. This function must return a type of **trigger.**

```
CREATE OR REPLACE FUNCTION your_trigger_function()
RETURNS TRIGGER AS $$
BEGIN
    -- Your SQL statements to execute when the trigger fires
    -- Example: RAISE NOTICE 'Trigger function executed.';
    RETURN NEW; - or OLD; depending on the operation (old has no values)
END;
$$ LANGUAGE plpgsql;
```

Creation of Triggers : Syntax : Step 2

- After defining the function, create the trigger itself, specifying when it fires (e.g., BEFORE or AFTER an INSERT, UPDATE, DELETE, or TRUNCATE operation) and which table it is associated with.

```
CREATE TRIGGER your_trigger_name
-- Specify when the trigger should fire (BEFORE, AFTER, or INSTEAD OF)
-- and the event (INSERT, UPDATE, DELETE, TRUNCATE)
AFTER INSERT ON your_table_name
FOR EACH ROW -- or FOR EACH STATEMENT
EXECUTE FUNCTION your_trigger_function();
```

Old vs New values in triggers !?

- In PostgreSQL, OLD and NEW are special record variables available in PL/pgSQL trigger functions. These records hold the old and new row versions for triggers fired by INSERT, UPDATE, or DELETE operations
  - For an INSERT trigger, OLD contains no values, and NEW contains the new values.
  - For an UPDATE trigger, OLD contains the old values, and NEW contains the new values.
  - For a DELETE trigger, OLD contains the old values, and NEW contains no values

Triggers : Order inventory

- -- Schema for Products Table
- CREATE TABLE Products (ProductID INT PRIMARY KEY, ProductName VARCHAR(255), QuantityInStock INT);

- -- Schema for Orders Table
- CREATE TABLE Orders (OrderID INT PRIMARY KEY, ProductID INT, QuantityOrdered INT, FOREIGN KEY (ProductID) REFERENCES Products(ProductID));

Need for Triggers in this order inventory?!

- Assume a user has placed an order of 3 quantities of some product ‘watch’
- What changes should be driven in the tables?
- When an order is placed, the stock of the product should be reduced by the ‘quantity placed’ in order to maintain our database correct!
- This is where need for triggers arrive!
- i.e. when an order is placed (insert done on order table) a change in products table by reducing the count of that particular item is expected to happen.
- Can that be done manually? Yes but involves a lot of man power to do so.
- Imagine a big billion sale hosted by Flipkart! Manual changes is so tedious to keep track of stock at all tables!
- And hence we create triggers to do things automatically

Triggers : Auto update of inventory : Step 1

1. Define the Trigger Function

This function will be called after a new row is inserted into the **Orders** table.

```
CREATE OR REPLACE FUNCTION update_inventory()
RETURNS TRIGGER
AS $$
BEGIN
    UPDATE Products
    SET QuantityInStock = QuantityInStock - NEW.QuantityOrdered
    WHERE ProductID = NEW.ProductID;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Triggers : Auto Update of inventory : Step 2

2. Now, attach the function to a trigger that fires after an insert operation on the **Orders** table

- CREATE TRIGGER after_order_placed

AFTER INSERT ON Orders

FOR EACH ROW

EXECUTE FUNCTION **update_inventory()**;

Happenings through triggers!

- insert into Products values(1,'watch',100);
- insert into Products values(2,'bicycle',10);
- select * from Products;

productid	productname	quantityinstock
1	watch	100
2	bicycle	10

Auto firing of triggers !

- insert into Orders values(11,1,2);
- insert into Orders values(12,2,6);
- select * from Products;
- select * from Orders;

productid	productname	quantityinstock
1	watch	98
2	bicycle	4

orderid	productid	quantityordered
11	1	2
12	2	6

To do! Organization Schema !

- CREATE TABLE Departments (DepartmentID SERIAL PRIMARY KEY, DepartmentName VARCHAR(255) NOT NULL, EmployeeCount INT DEFAULT 0);
- CREATE TABLE Employees (EmployeeID SERIAL PRIMARY KEY, Name VARCHAR(100) NOT NULL,DepartmentID INT, FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID));
- Create triggers
  - to ensure that whenever a new employee is added to a department, the employee count for that department is incremented by 1,
  - when an employee leaves (is deleted from the **Employees** table), the count is to be decremented by 1

Organization schema! Auto Increment !

- CREATE OR REPLACE FUNCTION increment_employee_count()

```
RETURNS TRIGGER AS $$
BEGIN
    UPDATE Departments
    SET EmployeeCount = EmployeeCount + 1
    WHERE DepartmentID = NEW.DepartmentID;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- CREATE TRIGGER trigger_employee_addition

```
AFTER INSERT ON Employees
FOR EACH ROW
EXECUTE FUNCTION increment_employee_count();
```

Organization schema! Auto Decrement !

- CREATE OR REPLACE FUNCTION decrement_employee_count()

```
RETURNS TRIGGER AS $$
BEGIN
    UPDATE Departments
    SET EmployeeCount = EmployeeCount - 1
    WHERE DepartmentID = OLD.DepartmentID;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

- CREATE TRIGGER trigger_employee_deletion

```
BEFORE DELETE ON Employees
FOR EACH ROW
EXECUTE FUNCTION decrement_employee_count();
```

Embedded SQL

- Embedded SQL refers to the practice of integrating SQL (Structured Query Language) statements directly within the code of a host programming language, such as C, C++, Java, or Python.
- This allows programmers to directly manipulate database resources (such as tables and queries) through their usual programming environment, without having to switch contexts or use separate database query tools.
- Embedded SQL combines the data processing power of SQL with the procedural capabilities of a host language, making it a powerful tool for applications requiring frequent and complex interactions with a database.

Why embedded SQL?

- Embedded SQL is particularly useful in situations where the application heavily interacts with a database and where performance, in terms of query execution and data retrieval, is crucial.
- It is widely used in legacy systems, especially in banking, finance, and other sectors where transaction integrity and speed are critical.

Goal !

- To develop a GUI where the data we enter should be communicated directly to the database!
- Imagine the forms that we fill directly gets stored in databases!
- End to end application !
- Front end and Back end to be integrated!
- What can be done?

Connectivity with front end tools! Python with PSQL!

Python : Tkinter, psycopg

Simple tkinter GUI window!

```
import tkinter as tk
root = tk.Tk() #main window
root.title("Simple Employee Data Entry")
tk.Label(root, text="Name:").pack()
name_entry = tk.Entry(root)
name_entry.pack()
tk.Label(root, text="Employee ID:").pack()
empid_entry = tk.Entry(root)
empid_entry.pack()
submit_button = tk.Button(root, text="Submit")
submit_button.pack()
root.mainloop()
```

- Tk is the alias name use for tkinter in this program
- Root is the name given for the main window and Tk() is the constructor for root window
- Window's title can be fixed using title()
- A label with the text "Name:" is created and added to the main window (root).
- The pack() method is called on the label object to automatically manage the layout. It places the label in the window and adjusts the size according to the content.
- An entry widget (text input field) is created for inputting the name
- A button labeled "Submit" is created. This button is meant to be clicked to perform an action, like submitting the data entered in the entry widgets.
- root.mainloop() : It keeps the window open until the user closes it, constantly listening for events

Psycopg !

- **Psycopg** is the most popular PostgreSQL adapter used in Python.
- It serves as a bridge between Python and PostgreSQL databases, allowing developers to use Python to execute PostgreSQL commands and manage database operations.
- It is designed to perform heavily multi-threaded applications that usually create and destroy lots of cursors and make a large number of simultaneous INSERTS or UPDATES.

Installation guide for psycopg!

- Upgrade your Pip first using the command
  - python -m pip install --upgrade pip
- Install the psycopg adapter for psql using the command,
  - python -m pip install psycopg

Steps to connect !

1. **Install PostgreSQL**
2. **Create a Database and Table:** In pgAdmin or via the command line, create database and a table with SQL as needed
3. **Install psycopg:** Run pip install psycopg in your command line or terminal to install the psycopg library.
4. **Establish a Database Connection and cursor:** Directly in your main script, create a connection to PostgreSQL using psycopg inside the event handler of our GUI.
5. **Direct Database Operations:** Write SQL commands directly in our GUI event handlers (like the button click events) to insert, update, or query data.

Steps to connect !

6. **Directly Modify the GUI:** Set up our GUI elements (like buttons and text fields) in the main part of our script and directly link them to database operations.

7. **Bind GUI Elements and Database Operations:** Use direct calls to psycopg within button commands to handle data insertion when the GUI buttons are clicked.

8. **Run and Test:** Execute script to start the GUI, test the inputs and interactions to ensure data flows correctly into the database.

Python code to connect to PSQL!

```
import tkinter as tk
import psycopg2
from tkinter import messagebox
root = tk.Tk()
root.title("Employee Data Entry")
tk.Label(root, text="Name:").pack()
name_entry = tk.Entry(root)
name_entry.pack()
tk.Label(root, text="Employee ID:").pack()
empid_entry = tk.Entry(root)
empid_entry.pack()
```

Prerequisite :
Create a table employees in the postgres database as postgres user!

Python code to connect to PSQL!

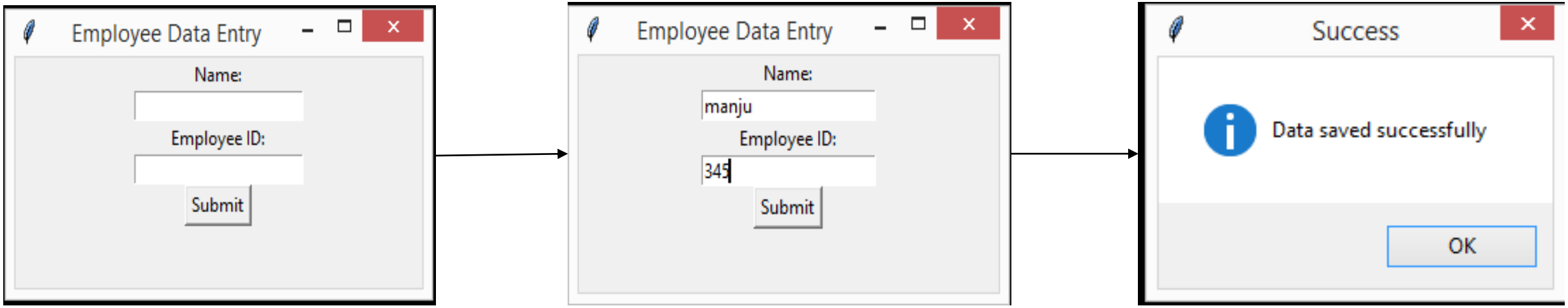
```
def submit():
    name = name_entry.get()
    emp_id = empid_entry.get()
    try:
        conn = psycopg2.connect(dbname="postgres“,user="postgres",password="1234",host="localhost",   port="5432" )
        cur = conn.cursor()
        cur.execute("INSERT INTO employees (name, id) VALUES (%s, %s)", (name, emp_id))
        conn.commit()
        messagebox.showinfo("Success", "Data saved successfully")
    except Exception as e:
        messagebox.showerror("Error", str(e))
    finally:
        if conn:
            cur.close()
            conn.close()
```

- A cursor is a database object that allows us to retrieve and manipulate each row one at a time in the DB.
- A cursor is nothing more than a row pointer

RATHINAM

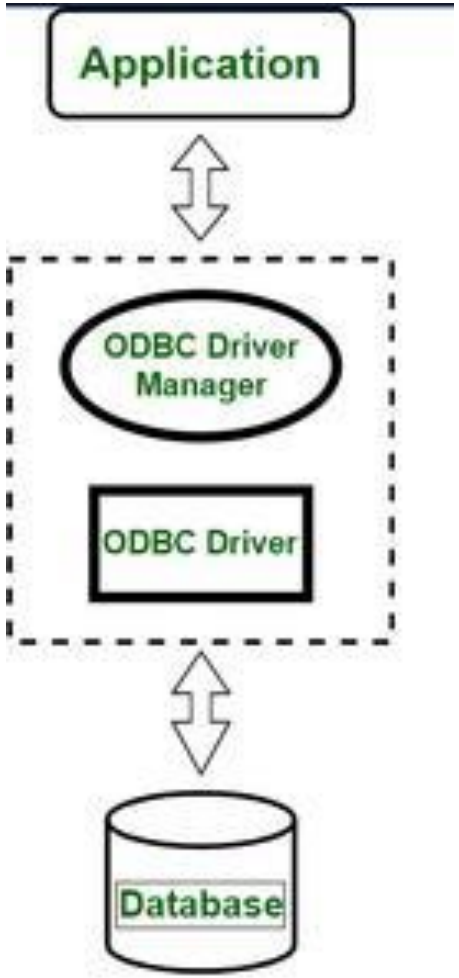
Python code to connect to PSQL!

```
submit_button = tk.Button(root, text="Submit", command=submit)
submit_button.pack()
root.mainloop()
```



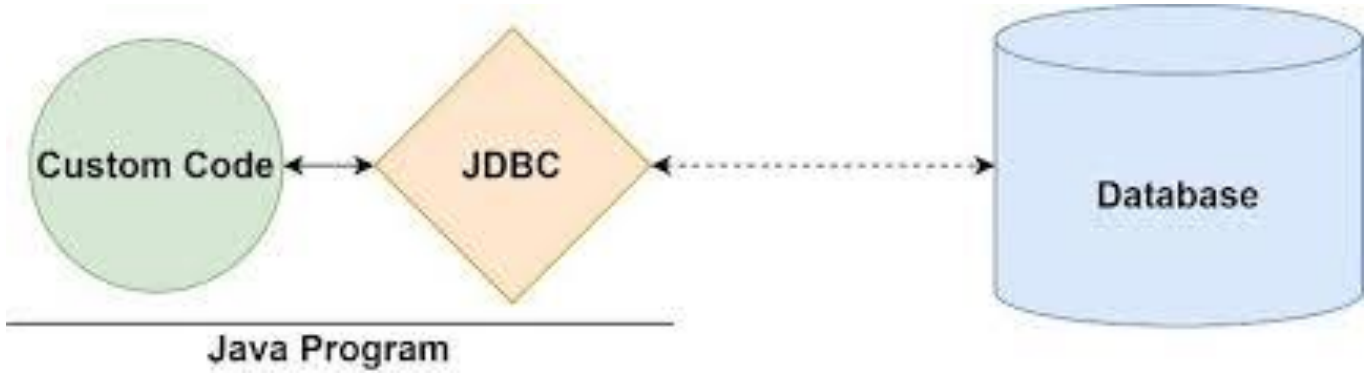
ODBC : Open Database connectivity

- ODBC (Open Database Connectivity) is a standard API (Application Programming Interface) for accessing database management systems (DBMS).
- The goal of ODBC is to make it possible to access any data from any application, regardless of which DBMS is handling the data.
- ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS.
- The purpose of this layer is to translate the application's data queries into commands that the DBMS understands.



JDBC : Java database Connectivity

- JDBC (Java Database Connectivity) is an API for the Java programming language that defines how a client may access a database.
- It is a part of the Java Standard Edition platform from Oracle Corporation.
- This API provides methods to query and update data in a database and is oriented towards relational databases.



JDBC vs. ODBC

Feature	JDBC	ODBC
Language Support	Java only	Language-agnostic (C, C++, Java, etc.)
API Type	Specific to Java platform	Universal, independent of programming language
Usage	Used exclusively in Java environments	Used across various programming environments
Database Communication	Through JDBC drivers specific to databases	Through database drivers via the Driver Manager
Standardization	Managed by Oracle (Java's steward)	Managed by Microsoft, but widely adopted
Compatibility	Works with any database that has a JDBC driver	Works with any database that has an ODBC driver
Deployment	Commonly used in web and enterprise applications	Commonly used in both desktop and server applications

Thank you!