Lecture 11

Stored Procedure

PostgreSQL Procedure

- A **stored procedure** is a set of SQL statements that can be stored and executed in a database management system (DBMS).
- It is a way to encapsulate a sequence of database operations, allowing for code reuse, better performance, and improved security.
- Stored procedures are typically written in SQL, but they can also contain procedural logic (such as loops, conditionals, and error handling), depending on the DBMS.

Advantage 1

- Reduce the number of round trips between application and database servers.

- All SQL statements are wrapped inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of sending multiple SQL statements and wait for the result between each call.

Advantage 2

- Increase application performance because the user-defined functions are pre-compiled and stored in the PostgreSQL database server.

Advantage 2

- Be able to reuse in many applications. Once you develop a function, you can reuse it in any applications.

Disadvantages

- Slow in software development because it requires specialized skills that many developers do not possess.
- Make it difficult to manage versions and hard to debug.
- May not be portable to other database management systems e.g., MySQL or Microsoft SQL Server.

Stored Procedures

- By default, PostgreSQL supports three procedural languages: SQL, PL/pgSQL, and C.
- You can also load other procedural languages e.g., Perl, Python, JS, Ruby, Java and TCL into PostgreSQL using extensions.

Precompiled:

Once a stored procedure is created, it is precompiled and stored in the database. The DBMS doesn't need to parse and compile the SQL each time it's executed.

Encapsulation:

Stored procedures allow you to group related SQL statements into a single logical unit.

Key Characteristics

Parameterization:

- Stored procedures often accept input parameters, and they may also return output parameters or result sets.

Security:

- You can control access to sensitive data by providing users with permissions

Error Handling

PL/pgSQL Overview

- PL/pgSQL (Procedural Language/PostgreSQL) is a procedural extension for PostgreSQL that allows users to write functions, procedures, and triggers with control-flow constructs like loops, conditionals, and error handling. It enhances standard SQL by adding procedural logic for more

- **Key Features of PL/pgSQL**
  - **Control Structures:** Supports IF, CASE, LOOP, WHILE, and FOR.
  - **Error Handling:** Provides EXCEPTION blocks to manage runtime errors.
  - **Dynamic SQL:** Allows execution of dynamically constructed queries.
  - **Variables and Parameters:** Enables declaration and manipulation of variables.
  - **Integration:** Works

PL/pgSQL Basics

```
DO $$
BEGIN
    RAISE NOTICE 'Hello from PL/pgSQL!';
END
$$;
```

Declaring PL/pgSQL Code Blocks

```
DECLARE
    user_name TEXT := 'Aruzhan';
    user_age INT := 25;
```

Declaring Variables

Control Structures

```
IF user_age > 18 THEN
    RAISE NOTICE 'User is an adult.';
ELSE
    RAISE NOTICE 'User is a minor.';
END IF;
```

Error Handling with EXCEPTION Blocks

```
BEGIN
    -- Code that may raise an error
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Division by zero error occurred!';
    WHEN OTHERS THEN
        RAISE NOTICE 'An unknown error occurred: %', SQLERRM;
END;
```

Basic Syntax

- **LANGUAGE plpgsql:** Specifies the procedural language for the procedure.
- **$$:** Delimiters for the procedure body.

```
CREATE OR REPLACE PROCEDURE procedure_name(parameter_list)
LANGUAGE plpgsql
AS $$ BEGIN

-- SQL statements

END;
$$;
```

Basic Syntax

In declare you will describe all variables

```
CREATE OR REPLACE PROCEDURE procedure_name(parameter_list)
LANGUAGE plpgsql
AS $$

DECLARE
-- variables

BEGIN

-- SQL statements - all logic

END;
$$;
```

Parameters

Parameters in stored procedures allow passing data in and out of the procedure, enabling dynamic behavior. PostgreSQL supports three types of parameters:

- **IN**: Used to pass values *into* the procedure.
- **OUT**: Used to return values *from* the procedure.
- **INOUT**: Used to pass a value in and return a value out.

Parameters

```
CREATE PROCEDURE procedure_name(
    param1 TYPE,            -- Default is IN
    param2 TYPE IN,         -- Explicit IN
    param3 TYPE OUT,        -- OUT parameter
    param4 TYPE INOUT       -- INOUT parameter
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Procedure logic here
END;
$$;
```

IN Parameters Example

```
14  CREATE PROCEDURE greet_user(name TEXT)
15  LANGUAGE plpgsql
16  AS $$
17  BEGIN
18      RAISE NOTICE 'Hello, %!', name;
19  END;
20  $$;
21
22  -- Call the procedure:
23  CALL greet_user('Aruzhan');
```

Data Output | Messages | Notifications

```
ЗАМЕЧАНИЕ:  Hello, Aruzhan!
CALL
```

OUT Parameters Example

```
25  -- Create the procedure
26  CREATE PROCEDURE calculate_square(num INT, OUT square INT)
27  LANGUAGE plpgsql
28  AS $$
29  BEGIN
30      square := num * num;
31  END;
32  $$;
33
34  -- Call the procedure and capture the output
35  DO $$
36  DECLARE
37      result INT; -- Declare a variable to store the OUT parameter
38  BEGIN
39      CALL calculate_square(4, result);
40      RAISE NOTICE 'Square: %', result; -- Output the result
41  END;
42  $$;
43
```

Data Output | Messages | Notifications

```
ЗАМЕЧАНИЕ:  Square: 16
DO
```

INOUT Parameters Example

```
44  -- Create the procedure
45  CREATE PROCEDURE double_number(INOUT num INT)
46  LANGUAGE plpgsql
47  AS $$
48  BEGIN
49      num := num * 2;
50  END;
51  $$;
52
53  -- Call the procedure:
54  DO $$
55  DECLARE
56      my_number INT := 5;
57  BEGIN
58      CALL double_number(my_number);
59      RAISE NOTICE 'Doubled Number: %', my_number;
60  END;
61  $$;
```

Data Output | Messages | Notifications

```
ЗАМЕЧАНИЕ:  Doubled Number: 10
DO
```

Functions

```
CREATE FUNCTION function_name (parameters)
RETURNS return_type
LANGUAGE language AS $$
  -- Function body
$$;
```

Syntax

```
CREATE FUNCTION calculate_area(radius NUMERIC)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN 3.14159 * radius * radius;
END;
$$;

SELECT calculate_area(5);
```

Data Output | Messages | Notifications

	calculate_area numeric
1	78.53975

Basic Example

```
CREATE FUNCTION get_full_name(first_name TEXT, last_name TEXT)
RETURNS TEXT
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN first_name || ' ' || last_name;
END;
$$;

SELECT get_full_name('Aruzhan', 'Tugambayeva');
```

Data Output | Messages | Notifications

	get_full_name text
1	Aruzhan Tugambayeva

Function with Multiple Parameters

Stored Procedures vs. Functions

Feature	Stored Procedures	Functions
Introduction	Introduced in PostgreSQL 11	Available since early PostgreSQL versions
Return Value	Do not return values by default	Must return a value (scalar, record, or table)
Transaction Control	Supports COMMIT, ROLLBACK, and savepoints	Cannot manage transactions
Syntax	CREATE PROCEDURE	CREATE FUNCTION
Call Statement	Invoked with CALL	Used within SQL queries or expressions
Purpose	Encapsulates workflows, automation, and complex logic	Handles reusable calculations or data processing
Dynamic SQL	Fully supports execution with EXECUTE	Limited, no transaction control
Typical Use Cases	ETL processes, data migrations, and transaction workflows	Data retrieval, calculations, and reporting

Stored Procedures vs. Functions

Key differences:

- Can return no value
- Support transaction control

Scenario	Use Stored Procedures	Use Functions
Need transaction control	Yes (supports COMMIT, ROLLBACK, savepoints)	No
Returning values	Not directly; uses OUT parameters	Yes
SQL Expression Integration	Cannot be embedded in SQL queries	Fully embeddable in SQL

Simple Example

employee_id [PK] integer	first_name character varying (50)	last_name character varying (50)	performance_rating integer	salary numeric (10,2)
1	John	Doe	4	55000.00
2	Jane	Smith	5	75000.00
3	Mike	Johnson	3	46000.00
4	Emily	Davis	4	52000.00
5	Daniel	Garcia	5	85000.00
6	Sophia	Martinez	2	40000.00
7	David	Rodriguez	3	48000.00
	livia	Wilson	5	78000.00
	ames	Anderson	4	56000.00
	harlotte	Thomas	3	47000.00
	ucas	Taylor	2	42000.00
	melia	Moore	4	53000.00
	lason	Jackson	5	90000.00
	than	White	4	57000.00
	va	Harris	3	49000.00

```sql
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    performance_rating INT CHECK (performance_rating >= 1 AND performance_rating <= 5),
    salary DECIMAL(10, 2)
);

INSERT INTO employees (first_name, last_name, performance_rating, salary)
VALUES
    ('John', 'Doe', 4, 55000.00),
    ('Jane', 'Smith', 5, 75000.00),
    ('Mike', 'Johnson', 3, 46000.00),
    ('Emily', 'Davis', 4, 52000.00),
    ('Daniel', 'Garcia', 5, 85000.00),
    ('Sophia', 'Martinez', 2, 40000.00),
    ('David', 'Rodriguez', 3, 48000.00),
    ('Olivia', 'Wilson', 5, 78000.00),
    ('James', 'Anderson', 4, 56000.00),
    ('Charlotte', 'Thomas', 3, 47000.00),
    ('Lucas', 'Taylor', 2, 42000.00),
    ('Amelia', 'Moore', 4, 53000.00),
    ('Mason', 'Jackson', 5, 90000.00),
    ('Ethan', 'White', 4, 57000.00),
    ('Ava', 'Harris', 3, 49000.00);
```

Simple Example

```sql
CREATE OR REPLACE PROCEDURE AdjustEmployeeSalaries()
LANGUAGE plpgsql
AS $$
BEGIN
    -- Update salaries based on performance rating
    UPDATE employees
    SET salary = CASE
        WHEN performance_rating = 5 THEN salary * 1.15   -- 15% increase for rating 5
        WHEN performance_rating = 4 THEN salary * 1.10   -- 10% increase for rating 4
        WHEN performance_rating = 3 THEN salary * 1.05   -- 5% increase for rating 3
        ELSE salary                                      -- No change for rating 2 or 1
    END;

    RAISE NOTICE 'Employee salaries have been updated based on performance rating';
END;
$$;
```

```sql
CALL AdjustEmployeeSalaries();

SELECT * FROM employees;
```

employee_id [PK] integer	first_name character varying (50)	last_name character varying (50)	performance_rating integer	salary numeric (10,2)
1	John	Doe	4	60500.00
2	Jane	Smith	5	86250.00
3	Mike	Johnson	3	48300.00
4	Emily	Davis	4	57200.00
5	Daniel	Garcia	5	97750.00
6	Sophia	Martinez	2	40000.00
7	David	Rodriguez	3	50400.00
8	Olivia	Wilson	5	89700.00
9	James	Anderson	4	61600.00
10	Charlotte	Thomas	3	49350.00
11	Lucas	Taylor	2	42000.00
12	Amelia	Moore	4	58300.00
13	Mason	Jackson	5	103500.00
14	Ethan	White	4	62700.00
15	Ava	Harris	3	51450.00

Stored Procedure with Multiple SQL Statements

```sql
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_name VARCHAR(100),
    order_date DATE,
    order_amount DECIMAL(10, 2),
    order_status VARCHAR(50)
);

INSERT INTO orders (customer_name, order_date, order_amount, ord
VALUES
    ('John Doe', '2024-10-01', 150.00, 'Pending'),
    ('Jane Smith', '2024-09-25', 200.00, 'Shipped'),
    ('Alice Brown', '2024-09-15', 120.00, 'Delivered'),
    ('Bob White', '2024-10-10', 75.00, 'Pending'),
    ('Charlie Green', '2024-09-30', 180.00, 'Pending'),
    ('David Black', '2024-10-02', 250.00, 'Delivered'),
    ('Emily Davis', '2024-09-18', 90.00, 'Shipped'),
    ('George King', '2024-10-05', 300.00, 'Pending'),
    ('Helen Clark', '2024-09-28', 130.00, 'Shipped'),
    ('Ivy Hall', '2024-09-20', 110.00, 'Pending');
```

order_id [PK] integer	customer_name character varying (100)	order_date date	order_amount numeric (10,2)	order_status character varying (50)
1	John Doe	2024-10-01	150.00	Pending
2	Jane Smith	2024-09-25	200.00	Shipped
3	Alice Brown	2024-09-15	120.00	Delivered
4	Bob White	2024-10-10	75.00	Pending
5	Charlie Green	2024-09-30	180.00	Pending
6	David Black	2024-10-02	250.00	Delivered
7	Emily Davis	2024-09-18	90.00	Shipped
8	George King	2024-10-05	300.00	Pending
9	Helen Clark	2024-09-28	130.00	Shipped
10	Ivy Hall	2024-09-20	110.00	Pending

Stored Procedure with Multiple SQL Statements

```sql
CREATE OR REPLACE PROCEDURE ManageCustomerOrders(
    p_customer_name VARCHAR,     -- Customer name
    p_delete_before_date DATE    -- Delete orders older than this date
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Retrieve all orders for the specified customer
    RAISE NOTICE 'Retrieving all orders for customer: %', p_customer_name;

    SELECT *
    FROM orders
    WHERE customer_name = p_customer_name;

    -- Update the status of all orders for the customer to 'Shipped'
    RAISE NOTICE 'Updating order status for customer: %', p_customer_name;

    UPDATE orders
    SET order_status = 'Shipped'
    WHERE customer_name = p_customer_name AND order_status = 'Pending';

    -- Delete orders older than the specified date
    RAISE NOTICE 'Deleting orders older than: %', p_delete_before_date;

    DELETE FROM orders
    WHERE order_date < p_delete_before_date;

    -- Inform user of the successful operation
    RAISE NOTICE 'Order status updated and old orders deleted for customer: %', p_customer_name;

END;
$$;

CALL ManageCustomerOrders('John Doe', '2024-09-30');

SELECT * FROM orders;
```

	order_id [PK] integer	customer_name character varying (100)	order_date date	order_amount numeric (10,2)	order_status character varying (50)
1	1	John Doe	2024-10-01	150.00	Pending
2	2	Jane Smith	2024-09-25	200.00	Shipped
3	3	Alice Brown	2024-09-15	120.00	Delivered
4	4	Bob White	2024-10-10	75.00	Pending
5	5	Charlie Green	2024-09-30	180.00	Pending
6	6	David Black	2024-10-02	250.00	Delivered
7	7	Emily Davis	2024-09-18	90.00	Shipped
8	8	George King	2024-10-05	300.00	Pending
9	9	Helen Clark	2024-09-28	130.00	Shipped
10	10	Ivy Hall	2024-09-20	110.00	Pending

Example with transaction

```sql
CREATE TABLE bank_accounts (
    account_id SERIAL PRIMARY KEY,
    account_holder_name VARCHAR(100),
    balance DECIMAL(10, 2) CHECK (balance >= 0)
);

INSERT INTO bank_accounts (account_holder_name, balance)
VALUES
    ('John Doe', 5000.00),
    ('Jane Smith', 3000.00),
    ('Mike Johnson', 2000.00),
    ('Emily Davis', 1500.00),
    ('Daniel Garcia', 10000.00);
```

	account_id [PK] integer	account_holder_name character varying (100)	balance numeric (10,2)
1	1	John Doe	5000.00
2	2	Jane Smith	3000.00
3	3	Mike Johnson	2000.00
4	4	Emily Davis	1500.00
5	5	Daniel Garcia	10000.00

Example with transaction

```sql
CREATE OR REPLACE PROCEDURE TransferFunds(
    p_from_account INT,  -- Source account ID
    p_to_account INT,    -- Destination account ID
    p_amount DECIMAL     -- Amount to transfer
)
LANGUAGE plpgsql
AS $$
DECLARE
    from_balance DECIMAL;
    to_balance DECIMAL;
BEGIN
    -- Retrieve balance of the source account
    SELECT balance INTO from_balance
    FROM bank_accounts
    WHERE account_id = p_from_account;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Source account % not found', p_from_account;
    END IF;

    -- Check if there are sufficient funds in the source account
    IF from_balance < p_amount THEN
        RAISE EXCEPTION 'Insufficient funds in account %', p_from_account;
    END IF;

    -- Retrieve balance of the destination account
    SELECT balance INTO to_balance
    FROM bank_accounts
    WHERE account_id = p_to_account;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Destination account % not found', p_to_account;
    END IF;

    -- Perform the transfer (deduct from source, add to destination)
    UPDATE bank_accounts
    SET balance = balance - p_amount
    WHERE account_id = p_from_account;

    UPDATE bank_accounts
    SET balance = balance + p_amount
    WHERE account_id = p_to_account;

    -- Inform user of the successful transfer
    RAISE NOTICE 'Transferred % from account % to account %', p_amount, p_from_account, p_to_account;

EXCEPTION
    WHEN OTHERS THEN
        -- Rollback the transaction in case of any error
        RAISE NOTICE 'Transaction failed. No changes were made.';
        -- Re-throw the exception to ensure the calling application knows about the error
        RAISE;
END;
$$;

CALL TransferFunds(1, 2, 500.00);
```

	account_id [PK] integer	account_holder_name character varying (100)	balance numeric (10,2)
1	3	Mike Johnson	2000.00
2	4	Emily Davis	1500.00
3	5	Daniel Garcia	10000.00
4	1	John Doe	4500.00
5	2	Jane Smith	3500.00

Dynamic SQL in Stored Procedures

```sql
-- Create the procedure
CREATE PROCEDURE safe_dynamic_query(query TEXT)
LANGUAGE plpgsql
AS $$
BEGIN
    EXECUTE query;
EXCEPTION
    WHEN OTHERS THEN
        RAISE NOTICE 'Error executing query: %', SQLERRM;
END;
$$;

-- Call the procedure
CALL safe_dynamic_query('SELECT * FROM non_existing_table');
```

Data Output | Messages | Notifications

```
ЗАМЕЧАНИЕ:  Error executing query: отношение "non_existing_table" не существует
CALL
```

```sql
-- Create the procedure
CREATE PROCEDURE create_dynamic_table(table_name TEXT)
LANGUAGE plpgsql
AS $$
BEGIN
    EXECUTE format('CREATE TABLE %I
    (id SERIAL PRIMARY KEY, name TEXT)', table_name);
END;
$$;

-- Call the procedure
CALL create_dynamic_table('new_employees');
```

Error Handling in Stored Procedures

```
 98  CREATE OR REPLACE PROCEDURE handle_division(a NUMERIC, b NUMERIC, OUT result NUMERIC)
 99  LANGUAGE plpgsql
100  AS $$
101  BEGIN
102      BEGIN
103          result := a / b;
104      EXCEPTION
105          WHEN division_by_zero THEN
106              RAISE NOTICE 'Division by zero occurred. Setting result to NULL.';
107              result := NULL;
108      END;
109  END;
110  $$;
111
112
113  -- Call the procedure:
114  DO $$
115  DECLARE
116      result NUMERIC; -- Declare a variable to hold the OUT parameter
117  BEGIN
118      CALL handle_division(10, 0, result);
119      RAISE NOTICE 'Result: %', result;
120  END;
121  $$;
```

Data Output | Messages | Notifications

```
ЗАМЕЧАНИЕ:  Division by zero occurred. Setting result to NULL.
ЗАМЕЧАНИЕ:  Result: <NULL>
DO
```

Real-World Use Cases for Stored Procedures

1. Data Transformation

- Transform and preprocess data for analytics or reporting.
- Example: Aggregating sales data by region and time period.

2. Business Logic Encapsulation

- Encapsulate business rules within the database to enforce consistency.
- Example: Automating salary adjustments based on performance ratings.

```
CREATE PROCEDURE aggregate_sales_by_region(region TEXT, OUT total_sales NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT SUM(sales) INTO total_sales
    FROM sales_data
    WHERE region = region;
END;
$$;
```

```
CREATE PROCEDURE adjust_salary(employee_id INT, rating TEXT)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE employees
    SET salary = salary * CASE
                            WHEN rating = 'A' THEN 1.10
                            WHEN rating = 'B' THEN 1.05
                            ELSE 1.00
                          END
    WHERE id = employee_id;
END;
$$;
```