

Procedures, Views, Cursors & Triggers in PostgreSQL

Stored Procedures:

Stored procedures in PostgreSQL are a way to encapsulate and store a set of SQL statements and procedural logic that can be executed as a single unit. They are used to perform complex operations, automate repetitive tasks, enforce business logic, and manage data more efficiently. Stored procedures can be written in various procedural languages supported by PostgreSQL, such as PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python.

Key Features of Stored Procedures:

1. **Encapsulation:** Encapsulate complex business logic and SQL statements.
2. **Reusability:** Define once, use many times across different applications and modules.
3. **Performance:** Reduce network traffic by executing multiple SQL statements in a single call.
4. **Security:** Provide an additional layer of security by controlling access to data through the procedure.
5. **Maintainability:** Improve code maintainability by organizing code into manageable blocks.

Creating a Stored Procedure:

Stored procedures are created using the `CREATE PROCEDURE` statement.

Executing a Stored Procedure:

Stored procedures are executed using the `CALL` statement.

Stored Procedures with IN, OUT, and INOUT Parameters:

Stored procedures can accept input parameters (`IN`), return output parameters (`OUT`), or both (`INOUT`).

Example with IN and OUT Parameters:

```
CREATE PROCEDURE get_employee_details(IN employee_id INT, OUT employee_name
TEXT, OUT employee_department TEXT)
LANGUAGE plpgsql
AS $$
BEGIN
SELECT name, department INTO employee_name, employee_department
FROM employees
WHERE id = employee_id;
END;
$$;
```

— Call the procedure and retrieve the output parameters

```
CALL get_employee_details(1, employee_name, employee_department);
```

Using Transactions in Stored Procedures:

Stored procedures can manage transactions using `BEGIN`, `COMMIT`, and `ROLLBACK` statements.


Example with Transaction Management:

Dropping a Stored Procedure:

To remove a stored procedure, use the `DROP PROCEDURE` statement.

Syntax:


sql

 Copy code

```
DROP PROCEDURE procedure_name (parameter_list);
```

Example:

sql

 Copy code

```
DROP PROCEDURE raise_salary(INT, NUMERIC);
```

Stored procedures in PostgreSQL offer a robust way to implement and manage business logic, enhance data integrity, and optimize database operations. By encapsulating logic within the database, stored procedures can lead to more maintainable and secure applications.

Views:

A view in PostgreSQL is a virtual table representing the result of a stored query. Views are created by defining a SQL query that pulls data from one or more tables in the database. Unlike physical tables, views do not store data themselves. Instead, they dynamically present data from the underlying tables whenever they are queried.

Key Features of Views:


1. **Simplicity:** Views can simplify complex queries by encapsulating them into a single object that can be queried as if it were a table.
2. **Security:** Views can restrict access to specific rows and columns, enhancing data security by providing a controlled way of exposing data.
3. **Abstraction:** Views provide a layer of abstraction, allowing users to work with a simplified interface instead of complex joins and aggregations.
4. **Reusability:** Once a view is created, it can be reused in multiple queries, ensuring consistency and saving time.
5. **Data Integrity:** Views ensure that the data is always presented in a consistent format, even if the underlying tables change.

Creating a View:

To create a view, you use the `CREATE VIEW` statement followed by the view name and the SQL query defining the view.

Syntax:


sql

 Copy code

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

sql


 Copy code

```
CREATE VIEW employee_view AS
SELECT id, name, department
FROM employees
WHERE active = true;
```

Querying a View:

You can query a view in the same way you query a table.

sql

 Copy code

```
SELECT * FROM employee_view;
```


Updating Data Through a View:

In some cases, you can also update data through a view, provided the view is simple enough and meets certain criteria. However, more complex views (involving joins, aggregations, etc.) may not support direct updates.

Modifying or Dropping a View:

To modify a view, you can use the `CREATE OR REPLACE VIEW` statement:


sql

 Copy code

```
CREATE OR REPLACE VIEW view_name AS  
SELECT column1, column2, ...  
FROM table_name  
WHERE new_condition;
```

To drop a view, you use the `DROP VIEW` statement:

sql

 Copy code

```
DROP VIEW view_name;
```

Views are a powerful feature in PostgreSQL, enabling you to simplify query management, enhance security, and ensure data consistency.

Cursors:

A cursor in PostgreSQL is a database object that allows you to retrieve a few rows from the result set of a larger query and process them one at a time. Cursors are particularly useful when dealing with large datasets, where fetching all rows at once is not practical due to memory constraints or the need for incremental processing.

Key Features of Cursors:


1. **Efficiency:** Cursors allow you to fetch and process a limited number of rows at a time, reducing memory usage and improving performance for large result sets.
2. **Control:** Cursors provide precise control over the traversal of result sets, including the ability to fetch rows in batches, move forward or backward, and fetch specific rows.
3. **Contextual Use:** Cursors are commonly used in stored procedures and functions where row-by-row processing is required.

Declaring a Cursor:

To use a cursor in PostgreSQL, you first need to declare it. This involves specifying the SQL query that the cursor will execute.

Syntax:


sql

 Copy code

```
DECLARE cursor_name CURSOR FOR query;
```

Example:

sql

 Copy code


```
DECLARE employee_cursor CURSOR FOR  
SELECT id, name, department  
FROM employees  
WHERE active = true;
```

Opening and Fetching from a Cursor:

After declaring a cursor, you open it and fetch rows from it as needed.

Syntax:


sql

 Copy code

```
OPEN cursor_name;  
FETCH [direction] FROM cursor_name;
```

Example:

sql

 Copy code

```
OPEN employee_cursor;  
  
-- Fetch the first row  
FETCH NEXT FROM employee_cursor;  
  
-- Fetch the next 10 rows  
FETCH 10 FROM employee_cursor;
```


Directions in FETCH:


- **NEXT:** Fetches the next row (default behavior).
- **PRIOR:** Fetches the previous row.
- **FIRST:** Fetches the first row.
- **LAST:** Fetches the last row.
- **ABSOLUTE n:** Fetches the nth row from the start (positive) or from the end (negative).
- **RELATIVE n:** Fetches the nth row from the current position (positive forward, negative backward).

Closing a Cursor:

When you are done with a cursor, you should close it to free up resources.

Syntax:


sql

 Copy code

```
CLOSE cursor_name;
```

Example:

sql

 Copy code

```
CLOSE employee_cursor;
```

Using Cursors in Functions:

Cursors are often used in PL/pgSQL functions for row-by-row processing.

```
CREATE OR REPLACE FUNCTION process_employees() RETURNS void AS $$  
DECLARE  
    employee_rec RECORD;  
    employee_cursor CURSOR FOR SELECT id, name, department FROM employees WHERE  
    active = true;  
BEGIN  
    OPEN employee_cursor;  
    LOOP  
        FETCH NEXT FROM employee_cursor INTO employee_rec;  
        EXIT WHEN NOT FOUND;  
        — Process each employee
```

```
RAISE NOTICE 'Employee ID: %, Name: %, Department: %', employee_rec.id,  
employee_rec.name, employee_rec.department;  
END LOOP;  
CLOSE employee_cursor;  
END;  
$$ LANGUAGE plpgsql;
```

In this example, the `process_employees` function uses a cursor to fetch and process active employees one at a time.

Cursors provide a powerful way to handle large datasets and complex row-by-row operations in PostgreSQL, offering flexibility and control in data processing tasks.

Triggers:

Triggers in PostgreSQL are database objects that are automatically executed or fired when a specified event occurs on a particular table or view. Triggers can be used to enforce business rules, validate input data, maintain audit logs, and synchronize tables.

Key Features of Triggers:

1. **Automation:** Automate repetitive tasks by triggering actions based on specific events.
2. **Data Integrity:** Enforce complex integrity constraints that cannot be implemented with standard database constraints.
3. **Audit and Logging:** Track changes to data for auditing and logging purposes.
4. **Complex Business Logic:** Implement complex business rules and validations that are executed automatically.

Types of Triggers:


1. **BEFORE Triggers:** Executed before the triggering event (INSERT, UPDATE, DELETE) occurs. Can be used to modify or validate the data before it is committed.
2. **AFTER Triggers:** Executed after the triggering event occurs. Often used for logging changes or cascading actions to other tables.
3. **INSTEAD OF Triggers:** Used with views to perform the specified actions instead of the triggering event.

Creating a Trigger:

To create a trigger, you need to define a trigger function first. The trigger function contains the logic that will be executed when the trigger fires.

Syntax for Trigger Function:


sql

 Copy code

```
CREATE OR REPLACE FUNCTION function_name()
RETURNS TRIGGER AS $$
BEGIN
    -- logic here
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Example of a Trigger Function:

sql


 Copy code

```
CREATE OR REPLACE FUNCTION update_modified_column()
RETURNS TRIGGER AS $$
BEGIN
    NEW.modified_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Next, you create the trigger and associate it with the trigger function.

Syntax for Creating a Trigger:


sql

 Copy code

```
CREATE TRIGGER trigger_name
BEFORE INSERT OR UPDATE ON table_name
FOR EACH ROW
EXECUTE FUNCTION function_name();
```

Example of Creating a Trigger:

sql

 Copy code

```
CREATE TRIGGER set_modified_timestamp
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION update_modified_column();
```


Using Triggers:

Example 1: Audit Log Trigger

This example logs changes to an `employees` table into an `employees_audit` table.

1. Create Audit Table:

sql

 Copy code

```
CREATE TABLE employees_audit (
  id SERIAL PRIMARY KEY,
  employee_id INT,
  action VARCHAR(10),
  modified_at TIMESTAMP,
  old_data JSONB,
  new_data JSONB
);
```

2. Create Trigger Function:

```
sql Copy code

CREATE OR REPLACE FUNCTION log_employee_changes()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'DELETE' THEN
        INSERT INTO employees_audit (employee_id, action, modified_at, old_data)
        VALUES (OLD.id, 'DELETE', NOW(), row_to_json(OLD)::jsonb);
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO employees_audit (employee_id, action, modified_at, old_data, new_data)
        VALUES (NEW.id, 'UPDATE', NOW(), row_to_json(OLD)::jsonb, row_to_json(NEW)::jsonb);
    ELSIF TG_OP = 'INSERT' THEN
        INSERT INTO employees_audit (employee_id, action, modified_at, new_data)
        VALUES (NEW.id, 'INSERT', NOW(), row_to_json(NEW)::jsonb);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

3. Create Trigger:

```
sql Copy code

CREATE TRIGGER audit_employee_changes
AFTER INSERT OR UPDATE OR DELETE ON employees
FOR EACH ROW
EXECUTE FUNCTION log_employee_changes();
```

Example 2: Preventing Deletion of Certain Records

This example prevents the deletion of employees from the `employees` table if their `is_manager` field is set to `true`.

1. Create Trigger Function:

```
sql Copy code

CREATE OR REPLACE FUNCTION prevent_manager_deletion()
RETURNS TRIGGER AS $$
BEGIN
    IF OLD.is_manager THEN
        RAISE EXCEPTION 'Managers cannot be deleted';
    END IF;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

2. Create Trigger:

```
sql Copy code

CREATE TRIGGER no_manager_deletion
BEFORE DELETE ON employees
FOR EACH ROW
EXECUTE FUNCTION prevent_manager_deletion();
```

Triggers in PostgreSQL provide a powerful mechanism to automate tasks, enforce data integrity, and implement complex business rules directly within the database. By carefully designing and implementing triggers, you can enhance the functionality and reliability of your database applications.