

ACCA7002 Database Systems

Tutorial 13

Views, Stored Procedure and Triggers in PostgreSQL using pgAdmin 4

Views:

A view is a virtual table in a database that is based on the result of a SELECT query. It does not store the data itself but provides a way to represent the data stored in one or more tables. Views allow users to retrieve and manipulate data as if they were working with a regular table, simplifying complex queries and providing a layer of abstraction.

View Creation Syntax:

The basic syntax for creating views in PostgreSQL using the CREATE VIEW statement is as follows:

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW view_name AS
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition;
```

- OR REPLACE: Optional clause that allows updating an existing view.
- TEMPORARY: Optional clause for creating a temporary view.
- view_name: Name of the view to be created.
- AS SELECT ...: Specifies the SELECT query that defines the view.

Joining Tables in Views:

Views can be used to combine information from multiple tables through joins. By defining a SELECT query in the view creation, you can join tables and present the combined result as a virtual table. This allows users to query and retrieve information from different tables without directly accessing the underlying tables.

Data Manipulation:

Views are read-only by default, meaning users can query data from views but cannot modify the underlying tables through the view. While data manipulation is limited, views provide a way to simplify complex queries by abstracting the underlying table structure. Users can focus on querying the view without dealing with the intricacies of the base tables.

Advantages of Views:

Data Abstraction: Views provide a layer of abstraction, allowing users to interact with a simplified representation of the data.

Security: Views can be used to limit access to specific columns or rows, enhancing data security.

Simplified Query Logic: Users can work with views instead of complex joins and subqueries, making queries more straightforward and readable.

Performance Considerations:

While views offer a convenient abstraction layer, there can be performance considerations, especially when dealing with large datasets. The execution of complex views may involve additional processing overhead. It's essential to consider the performance implications and optimize views if necessary.

Dynamic Views:

Dynamic views refer to views whose underlying data changes as the source tables are updated. As the base tables change, the results presented by the view are automatically updated to reflect the changes. This dynamic nature ensures that users always access the latest information without explicitly refreshing the view.

Security Implications:

Views can be used to restrict access to certain columns or rows in a table. This enhances data security by allowing administrators to control which parts of the data are visible to different users. Views can act as a security layer, presenting a subset of data based on user roles and permissions.

Exercise 1: Basic Views

Objective:

Create a simple view that displays information about books in a library.

Instructions:

Create a table named books with columns: book_id, title, author, and publication_year.
Insert some sample data into the books table.

Create a view named book_information that includes the title, author, and publication_year columns from the books table.

Answer:

```
-- Table creation
CREATE TABLE books (
    book_id SERIAL PRIMARY KEY,
    title VARCHAR(200),
    author VARCHAR(100),
    publication_year INT
);

-- Sample data insertion
INSERT INTO books (title, author, publication_year) VALUES
    ('Book1', 'Author1', 2020),
    ('Book2', 'Author2', 2018),
    ('Book3', 'Author1', 2015);

-- View creation
CREATE OR REPLACE VIEW book_information AS
    SELECT title, author, publication_year
    FROM books;
```

Exercise 2: Joined Views

Objective:

Create a view that joins information from multiple tables.

Instructions:

Create two tables: students and grades. The students table has columns student_id and student_name. The grades table has columns student_id, course, and grade.

Insert some sample data into both tables.

Create a view named student_grades that includes the student_name, course, and grade columns by joining the students and grades tables.

Answer:

```
-- Table creation
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    student_name VARCHAR(100)
);

CREATE TABLE grades (
    student_id INT REFERENCES students(student_id),
    course VARCHAR(50),
    grade INT
);

-- Sample data insertion
INSERT INTO students (student_name) VALUES
    ('Student1'),
    ('Student2');

INSERT INTO grades (student_id, course, grade) VALUES
    (1, 'Math', 85),
    (1, 'Science', 90),
    (2, 'Math', 75);

-- View creation
CREATE OR REPLACE VIEW student_grades AS
    SELECT students.student_name, grades.course, grades.grade
    FROM students
    JOIN grades ON students.student_id = grades.student_id;
```

Stored Procedures:

Definition:

Stored procedures are precompiled SQL statements that can be executed as a single unit. They allow you to encapsulate a series of SQL statements into a reusable routine.

Advantages:

Modularity: Encapsulate complex logic into a single unit.

Reusability: Can be called from multiple parts of an application.

Security: Limit access to data through stored procedures.

Syntax:

Example of a simple stored procedure:

```
CREATE OR REPLACE FUNCTION my_procedure(param1 INT, param2
TEXT) RETURNS INT AS $$
DECLARE
    result INT;
BEGIN
    -- Procedure logic here
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Parameters:

Stored procedures can take input parameters and return values.

Example with parameters:

```
CREATE OR REPLACE FUNCTION calculate_sum(a INT, b INT) RETURNS
INT AS $$
BEGIN
    RETURN a + b;
END;
$$ LANGUAGE plpgsql;
```

Triggers:

Definition:

Triggers are database objects that automatically execute a specified action in response to certain events on a particular table or view.

Types of Triggers:

BEFORE Triggers: Execute before the specified event (e.g., INSERT, UPDATE, DELETE).

AFTER Triggers: Execute after the specified event.

INSTEAD OF Triggers: Execute instead of the triggering event.

Event Types:

INSERT: Triggered when a new row is inserted.

UPDATE: Triggered when a row is updated.

DELETE: Triggered when a row is deleted.

Syntax:

Example of a simple AFTER INSERT trigger:

```
CREATE OR REPLACE FUNCTION my_trigger_function() RETURNS
TRIGGER AS $$
BEGIN
    -- Trigger logic here
    RETURN NEW; -- or NULL for INSTEAD OF triggers
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER my_trigger AFTER INSERT ON my_table
FOR EACH ROW EXECUTE FUNCTION my_trigger_function();
```

Exercise 1: Library Database Setup

Create Tables: Create tables for books, authors, and members with appropriate columns such as book_id, title, author_id, author_name, member_id, member_name, etc. Ensure proper relationships between the tables.

Insert Sample Data: Insert some sample data into the tables to represent books, authors, and library members.

```
-- Sample SQL for creating tables
CREATE TABLE authors (
    author_id SERIAL PRIMARY KEY,
    author_name VARCHAR(100)
);

CREATE TABLE books (
    book_id SERIAL PRIMARY KEY,
    title VARCHAR(200),
    author_id INT REFERENCES authors(author_id),
    status VARCHAR(20) DEFAULT 'Available'
);

CREATE TABLE members (
    member_id SERIAL PRIMARY KEY,
    member_name VARCHAR(100),
    account_balance DECIMAL(10, 2) DEFAULT 0.0
);

-- Sample SQL for inserting data
INSERT INTO authors (author_name) VALUES ('Author1'),
('Author2');
INSERT INTO books (title, author_id) VALUES ('Book1', 1),
('Book2', 2);
INSERT INTO members (member_name) VALUES ('Member1'),
('Member2');
```

Exercise 2: Stored Procedure for Book Checkout

Stored Procedure:

Create a stored procedure named `checkout_book` that takes parameters such as `book_id` and `member_id`.

Inside the procedure, check if the book is available. If available, update the book's status to 'Checked Out' and record the checkout date.

```
-- Sample SQL for stored procedure
CREATE OR REPLACE FUNCTION checkout_book(p_book_id INT,
p_member_id INT) RETURNS VOID AS $$
BEGIN
    UPDATE books
    SET status = 'Checked Out', checkout_date = CURRENT_DATE
    WHERE book_id = p_book_id AND status = 'Available';

    -- Additional logic for member record or transactions
END;
$$ LANGUAGE plpgsql;
```

Exercise 3: Stored Procedure for Book Return

Stored Procedure:

Create a stored procedure named `return_book` that takes parameters such as `book_id` and `member_id`.

Inside the procedure, check if the book is checked out by the specified member. If yes, update the book's status to 'Available' and calculate the late fee if applicable.

```
-- Sample SQL for stored procedure
CREATE OR REPLACE FUNCTION return_book(p_book_id INT,
p_member_id INT) RETURNS VOID AS $$
BEGIN
    UPDATE books
    SET status = 'Available', return_date = CURRENT_DATE
    WHERE book_id = p_book_id AND status = 'Checked Out';

    -- Additional logic for late return penalty or
    transactions
END;
$$ LANGUAGE plpgsql;
```

Exercise 4: Trigger for Late Return Penalty

Trigger: Create a trigger named `late_return_penalty` that activates whenever a book is returned.

If the return date is later than the expected return date, calculate a penalty fee and update the member's account balance.

```
-- Sample SQL for trigger
CREATE OR REPLACE FUNCTION late_return_penalty() RETURNS
TRIGGER AS $$
BEGIN
    IF NEW.return_date > NEW.expected_return_date THEN
        -- Calculate penalty fee and update member's account
balance
        UPDATE members
        SET account_balance = account_balance + penalty_fee
        WHERE member_id = NEW.member_id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER late_return_trigger
AFTER UPDATE ON books
FOR EACH ROW
WHEN (NEW.return_date > NEW.expected_return_date)
EXECUTE FUNCTION late_return_penalty();
```

Exercise 5: Stored Procedure for Author Analytics

Stored Procedure: Create a stored procedure named `author_analytics` that takes an `author_id` as a parameter.

Inside the procedure, retrieve information about the author, including the number of books written, average book rating, etc.

```
-- Sample SQL for stored procedure
CREATE OR REPLACE FUNCTION author_analytics(p_author_id INT)
RETURNS TABLE (
    total_books INT,
    average_rating NUMERIC
) AS $$
BEGIN
    RETURN QUERY (
        SELECT
            COUNT(b.book_id) AS total_books,
            COALESCE(AVG(b.rating), 0) AS average_rating
        FROM
            books b
        WHERE
            b.author_id = p_author_id
    );
END;
$$ LANGUAGE plpgsql;

-----End-----
```