Лабораторна робота 6. Маніпулювання даними. Тригери.

Розглянемо СУБД, які виконують не тільки дії, явно зазначені додатком, але також реагують на події, що виникають в самій БД. Такі системи називаються активними базами даних.

Формально поведінка таких систем визначається в термінах правил, що містять три компоненти: 1) подія; 2) додаткові умови; 3) опис дій, які повинні виконуватися при зазначеній події, якщо умови виконуються.

Активність бази даних реалізується за допомогою апарату тригерів. Тригером називають функцію, зазвичай написану на процедурній мові, яка викликається системою автоматично при спрацьовуванні пов'язаних з нею правил. У специфікації тригера можуть бути визначені додаткові умови. В системі PostgreSQL розрізняють тригери, що спрацьовують при модифікації даних (INSERT, UPDATE, DELETE і TRUNCATE), і тригери подій, що спрацьовують при виконанні операторів мови опису даних, до числа яких відносяться ALTER, CREATE, DROP, GRANT, REVOKE.

Процедурний код, реалізований в тригері, може істотно доповнити або змінити семантику стандартних операторів SQL. Наприклад, тригери можна використовувати для перевірки умов цілісності, які неможливо описати стандартними засобами мови SQL, або для реєстрації змін, які виконуються додатком, в іншій таблиці. Важливо підкреслити, що дії, передбачені в тригері, будуть виконуватися завжди, коли виникає специфікована ситуація, і виконуються в рамках тієї ж транзакції. Додаток не має ніякої можливості обійти або скасувати дію тригера.

Дії, що виконуються тригером, в системі PostgreSQL задаються функцією, яка повинна бути визначена в базі даних до створення тригера. Зазвичай функція тригера не має явно описаних параметрів, тому що інформація про контекст виклику функції може бути отримана іншим способом, і повертає значення типу trigger. Такі функції можуть бути написані на будь-якій процедурній мові програмування, доступній для використання в PostgreSQL, при цьому спосіб доступу до контексту, в якому викликаний тригер, залежить від мови програмування. У функціях, написаних на мові PL/pgSQL, для цього доступні зумовлені змінні.

Розглянемо, як в PostgreSQL визначаються тригери, що викликаються при модифікації даних. Для визначення тригера модифікації використовується оператор CREATE TRIGGER, в якому вказується наступна інформація:

— Рівень тригера. Тригери можуть бути визначені на рівні операторів SQL (FOR EACH STATEMENT) або на рівні рядків (FOR EACH ROW). На рівні рядків тригер викликається для кожного рядка таблиці, який оновлюється оператором SQL. На рівні оператора тригер виконується один раз при виконанні збудливого оператора.

— Оператори, виконання яких збуджує тригер (INSERT, UPDATE, DELETE, TRUNCATE).

— Об'єкт бази даних, при модифікації якого запускається тригер (таблиця або представлення).

— Відносний час виконання тригера (BEFORE, AFTER, INSTEAD OF). Тригери BEFORE спрацьовують безпосередньо до, а тригери AFTER - відразу після збудливого оператора. Тригери INSTEAD OF використовуються тільки для представлень і дозволяють визначити, які дії повинні виконуватися замість операцій модифікації даних.

— Додаткові умови, що обмежують запуск тригера (WHEN).

— Функція тригера, що виконує необхідні дії.

— Можливо, додаткові параметри функції тригера.

Зауваження: одна і та ж функція тригера може використовуватися для визначення різних тригерів.

В PL/pgSQL для тригерів рівня рядків визначені змінні OLD і NEW, що містять відповідно старе і нове значення рядка. При цьому для оператора INSERT не існує старого

значення, а для DELETE - нового. Тригери BEFORE можуть змінювати значення атрибутів у змінній NEW. Для того щоб виконання оператора, який порушив тригер, було нормально продовжено, функція тригера повинна повернути непорожнє (певне) значення. В тригерах, визначених для операторів INSERT і UPDATE, це значення буде використовуватися в якості нового значення оновлюваного кортежу, тому функція повинна повернути вихідне або змінене значення змінної NEW.

Усередині функції тригера можна виконувати будь-які оператори SQL, допустимі в функціях. Це може привести до каскадному запуску іншого або того ж самого тригера, в тому числі може викликати нескінченну рекурсію, відповідальність за запобігання якої покладено на програміста.

Структура підпрограм PLpg/SQL

На відміну від збережених процедур стандартного *SQL PLpg/SQL* містить два типи підпрограм — ***процедури*** (аналогічні стандартним) і *користувальницькі* ***функції***. Розглянемо *користувальницькі* ***функції***.

Для створення такої *функції* використовується наступний синтаксис:

```
CREATE OR REPLACE FUNCTION Ім'я_функції
    (Ім'я_параметра Вид_передачі_значення Тип_параметра, …)
    RETURN Тип_значення_яке_повертається
    AS $$
        PLpg/SQL програма, яка повинна завершуватися оператором
        RETURN Значення_яке_повертається;
    $$ LANGUAGE plpgsql;
```

Можливі два види передачі значення в параметри функції:

IN — значення параметра передається у функцію (передача за значенням)

OUT — значення параметра може бути повернуте з функції (передача по посиланню)

Розглянемо можливості мови *PLpg/SQL* для програмування збережених процедур.

Аналогічно стандартному *SQL* ***структура збереженої процедури***, яка створюється *PLpg/SQL*, містить три блоки: декларативний блок, блок, який виконується, блок обробки винятків — умов, які викликають помилки або попередження про помилки:

```
DECLARE
    // Оголошення
    BEGIN
        // Виконання команд обробки даних
    EXCEPTION
        // Обробка виняткових ситуацій
    END;
```

Блок, який виконується, є обов'язковим, а два інших блоки можуть бути відсутніми.

! Проте важливою відмінністю діалекту *PLpg/SQL* від стандарту є те, що **всі** його функції належать тільки до категорії процедур вибору, а тому **завжди** повинні повертати значення. Внаслідок цього для виклику функцій в *PLpg/SQL* використовується тільки команда SELECT.

Змінні в PLpg/SQL

Змінна в *PLpg/SQL* може мати будь-який тип даних, властивий стандартному *SQL*, такий як NUMBER, CHAR, DATE, або властивий діалекту *PLpg/SQL*, такий як BOOLEAN. Наприклад, необхідно оголосити змінну з ім'ям *Part_No* так, щоб вона могла зберігати 4-хзначні числові значення, і змінну з ім'ям *In_Stock*, що може приймати булеве значення TRUE або FALSE. Оголошуються змінні цього прикладу так:

```
…   Tax     NUMBER(4),
    Bonus   DEC(4, 2),
    Valid   BOOLEAN…
```

Крім того, в *PLpg/SQL* є можливість оголошувати записи і таблиці, використовуючи складні типи даних цього діалекту: RECORD і TABLE.

Присвоєння значень змінним у цьому розширенні нічим не відрізняється від стандарту *SQL*.

По-перше, це оператор присвоювання ‘:=‘. Наприклад:

Tax := Price * Tax_Rate;

Bonus := Current_Salary * 0.10;

Valid := FALSE.

По-друге, це введення в змінну значення з БД за допомогою фрази INTO команди SELECT або FETCH. Наприклад: обчислити 10% премії при виплаті зарплати співробітника:

SELECT Salary * 0.10 INTO Bonus FROM Lecturer;

Після цього значення змінної *Bonus* можна використовувати в інших обчисленнях, або внести його в таблицю БД.

Атрибути

Змінні в *PLpg/SQL* можуть бути так званими „атрибутами“, тобто властивостями, які дозволяють посилатися на тип даних, який має один зі стовпців таблиць БД, не повторюючи його оголошення. Синтаксис оголошення змінної як атрибута наступний:

Атрибут%TYPE

Наприклад, таблиця *Books* містить стовпець із ім’ям *Title*. Щоб дати змінній *My_Title* той же тип даних, що й у стовпця *Title*, не знаючи точного визначення цього стовпця в БД, досить указати наступну інструкцію:

…My_Title Books...Title%TYPE;…

Таке оголошення змінної має дві переваги:

— немає необхідності знати точний тип даних стовпця *Title*;

— якщо визначення стовпця *Title* у БД зміниться, наприклад, збільшиться його довжина, тип даних змінної *My_Title* зміниться відповідно під час виконання.

Також можна використовувати інший вид атрибута із синтаксисом:

Атрибут%ROWTYPE

В PLpg/SQL для групування даних використовуються записи. Запис складається з декількох стовпців, у яких можуть зберігатися значення даних. Атрибут %ROWTYPE позначає тип запису, що представляє рядок у таблиці. Такий запис, тобто змінна, оголошена з атрибутом %ROWTYPE, може зберігати цілий рядок даних, отриманий з таблиці або через курсор (який буде розглянутий у наступному параграфі).

Стовпці в рядку таблиці та відповідні стовпці в запису мають однакові імена і типи даних. У наступному прикладі оголошується запис з ім’ям *Dept_Rec*:

…Dept_Rec Dept%ROWTYPE;…

Для звернення до значень стовпців запису використовуються уточнені посилання, як показує наступна інструкція:

My_DeptNo := Dept_Rec.DeptNo;

Управляючі структури

В *PLpg/SQL*, як і в інших мовах програмування, існують ***команди передачі управління***, до яких належать оператор *умовного переходу* і оператори *циклів*.

В *PLpg/SQL* вони звуться ***управляючими структурами***: умовного управління (IF-THEN-ELSE) і ітеративного управління FOR-LOOP, WHILE-LOOP.

В якості **прикладу** умовного управління, розглянемо програму, яка підраховує кількість академічних заборгованостей студента, код якого дорівнює 5, та у відповідності з припустимою кількістю генерує для студента попереджуючі повідомлення. Оформимо цю програму у вигляді функції *PLpg/SQL*:

```
CREATE FUNCTION     Попередження_if_else()
    RETURNS    INTEGER
    AS $$
    DECLARE
        min_positive_mark           CONSTANT     NUMERIC(2) := 60;
        max_negotive_mark_count     CONSTANT     NUMERIC(1) := 2;
        negotive_mark_count         NUMERIC(1);
        StudentKod                  INTEGER;
    BEGIN
        StudentKod := 5;
        SELECT   COUNT(R1.Kod)   INTO    negotive_mark_count
            FROM      Rating R1
            WHERE    Kod = StudentKod AND Mark < min_positive_mark;
        IF     negotive_mark_count > max_negotive_mark_count  THEN
            INSERT INTO    Letters (kod, content)
                VALUES(StudentKod,'Ваша кількість заборгованостей =' ||
                        negotive_mark_count ||
                        ', що перевищує припустиму кількість!');
        ELSE
            INSERT INTO    Letters (kod, content)
                VALUES(StudentKod,
                        'Вам необхідно ліквідувати заборгованості у кількості '
                    ||
                        negotive_mark_count);
        END IF;
        RETURN 1;
      END;$$
    LANGUAGE 'plpgsql';
```

Для звернення до цієї функції треба виконати такий запит:

```
SELECT    Попередження_if_else();
```

Як було вже відзначено, в *PostgreSQL* всі функції повинні що-небудь повертати. Через те, що від цієї функції не очікується жодного результата, створюємо його штучно – повертаємо значення 1 як символ того, що функція успішно виконалася. Тобто при виклику за допомогою команди SELECT побачимо у результаті значення 1, а в таблиці Letters відповідні записи. Якщо в студента з кодом 5 взагалі немає оцінок або всі оцінки позитивні, додасться запис з кількістю заборгованостей, рівною 0.

Для циклу FOR-LOOP задається інтервал цілих чисел, а дії усередині циклу виконуються один раз для кожного цілого в цьому інтервалі. **Наприклад**: наступна програма визначає кількість академічних заборгованостей студентів з номерами від 1 до 7 та у відповідності з припустимою кількістю генерує для кожного студента повідомлення:

```
CREATE FUNCTION     Попередження_for_loop()
    RETURNS    INTEGER
    AS $$
    DECLARE
        min_positive_mark           CONSTANT     NUMERIC(2) := 60;
        max_negotive_mark_count     CONSTANT     NUMERIC(1) := 2;
        negotive_mark_count         NUMERIC(1);
        StudentKod                  Student.Kod%TYPE;
        Name                        Student.FirstName%TYPE;
    BEGIN
        FOR     StudentKod    IN 1..7   LOOP
          SELECT    FirstName, COUNT(R.Kod)
```

```
            INTO    Name, negotive_mark_count
            FROM    Student S, Rating R
            WHERE   S.Kod = StudentKod AND
                    Mark < min_positive_mark AND S.Kod = R.Kod
            GROUP BY FirstName;
        IF  negotive_mark_count > max_negotive_mark_count   THEN
            INSERT INTO  Letters (kod, content)
                VALUES(StudentKod, Name ||
                        '! Ваша кількість заборгованостей =' ||
                        negotive_mark_count ||
                        ', що перевищує припустиму кількість!');
        ELSE
            INSERT INTO  Letters (kod, content)
                VALUES(StudentKod, Name ||
                        '! Вам необхідно ліквідувати заборгованості у кількості
                    '||
                        negotive_mark_count);
        END IF;
    END LOOP;
    RETURN 1;
END;$$
LANGUAGE 'plpgsql';
```

При виклику цієї функції за допомогою наступної інструкції:

```
SELECT  Попередження_for_loop();
```

побачимо в її результаті 1, а в таблиці *Letters* — відповідні записи. Якщо у студента взагалі немає оцінок або всі вони позитивні, запис до таблиці *Letters* додасться, але без повідомлення (стовпець *Content* буде порожній).

Обробка помилок

В *PLpg/SQL* можна обробляти тільки внутрішні певні умови помилок, які називаються винятками. Коли виникає помилка, обробляється виняток. Це значить, що нормальне виконання припиняється, і управління передається в область обробки винятків або блоку підпрограми *PLpg/SQL*. Для обробки винятків створюються спеціальні програми — ***оброблювачі винятків***.

Приклад рішення завдання до лабораторної роботи 6.

Запишіть *SQL*-запити для маніпулювання даними з таблиць, що створені у лабораторній роботі 1.

П.1. Створіть тригер, який при додаванні нового рейсу внесе дані до зв’язаних(-ої) таблиць(-і).

Розв’язання.

```
CREATE FUNCTION voyage_class()
    RETURNS trigger
AS $$ BEGIN
    INSERT INTO class (voyage) VALUES (new.id_voyage);
    RETURN new;
END;$$
LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER voyage_class
        AFTER INSERT ON voyage
        FOR EACH ROW
        EXECUTE PROCEDURE voyage_class();
```

П.2. Створіть тригер, для видалення співробітника, якого звільнили

Розв'язання.

```
CREATE FUNCTION del_employee()
RETURNS trigger
AS $$ BEGIN
        UPDATE ticket SET employee=NULL  WHERE employee=old.id_employee;
        RETURN old;
END;$$
LANGUAGE 'plpgsql';

CREATE TRIGGER del_employee
        BEFORE DELETE ON employee
        FOR EACH ROW
        EXECUTE PROCEDURE del_employee();
```

П.3. Створіть функцію, яка буде при видачі квитка вносити дані в таблиці *Passenger* та *Ticket*, при цьому перевіряти чи існує співробітник, який видає квиток, та клас салону, на якій видають квиток, при помилці повинні бути відображені відповідні повідомлення.

Розв'язання.

```
CREATE OR REPLACE FUNCTION ticket_create -- назва функції
(full_name_pas CHAR (30), sex CHAR (1), passport CHAR, name_class CHAR (30),
                place INT, full_name_emp CHAR(30), operation CHAR(7))
        RETURNS INTEGER -- функція повертає тип даних integer
        AS $$
    -- Початок тіла функції
    -- Секція об'яви змінних.
        DECLARE
        id_passenger passenger.id_passenger%TYPE; -- тип даних посилається на тип
                даних поля id_passenger таблиці passenger
        id_ticket ticket.id_ticket%TYPE;
        t_class class.id_class%TYPE;
        t_employee employee.id_employee%TYPE;
    -- Секція тіла функції.
        BEGIN
            -- Отримання нового значення коду пасажира з генератора "s_passenger"
            id_passenger := NEXTVAL('s_passenger');
                -- Додавання запису до таблиці
            INSERT INTO passenger VALUES
                    (id_passenger,full_name_pas,sex,passport);
                -- Отримання нового значення коду квитка з генератора "s_ticket"
            id_ticket := NEXTVAL('s_ticket');
                -- Перевірка на правильність значення вхідного параметру класу
                -- з таблиці "class" через отримання відповіді на запит.
                -- Результат запиту відправляється в змінну t_class (код класу).
```

```
    SELECT id_class INTO t_class FROM class WHERE name = name_class;
       -- Якщо відповідь пуста, то значення класу некоректне.
    IF NOT FOUND THEN
       -- Виклик обробника помилки та виведення на екран повідомлення.
       -- В рядку з повідомленням параметр % вказує на
       -- значення змінної після коми.
          RAISE EXCEPTION 'Помилка: Класу % не існує', name_class;
    END IF;

    SELECT id_employee INTO t_employee FROM employee WHERE
          full_name = full_name_emp;
       -- Якщо відповідь пуста, то значення співробітника некоректне.
    IF NOT FOUND THEN
       -- Виклик обробника помилки та виведення на екран повідомлення.
       -- В рядку з повідомленням параметр % вказує на
       -- значення змінної після коми.
          RAISE EXCEPTION'Помилка: Співробітника % не існує',
                full_name_emp;
    END IF;

  -- Додавання запису до таблиці
       INSERT INTO ticketVALUES (id_ticket, id_passenger, t_class, place,
          t_employee, operation);
       -- Виведення на екран повідомлення про успішну видачу квитка
       RAISE NOTICE 'Квиток видано';
       -- Повернення з функції значення коду квитка
       RETURN id_ticket;
  END;
-- Завершення тіла функції
   $$ LANGUAGE plpgsql; -- назва модуля обробки мовних конструкцій тіла
          функції
```

Приклад виклику функції видачі квитка:

```
SELECT Ticket_Create ('Іванов','м','КМ 897654','бізнес',12,'Арсірій О.О.','покупка');
```

І, нарешті, перевірка результату:

```
SELECT * FROM Passenger;
```

Завдання до лабораторної роботи 6

Запишіть *SQL*-запити для маніпулювання даними з таблиць, що створені у лабораторній роботі 1. В роботі обов’язково відобразити:

1) тригер на операцію модифікації даних INSERT;
2) тригер на операцію модифікації даних DELETE;
3) тригер на операцію модифікації даних UPDATE;
4) користувальницьку функцію (збережену процедуру), яка викликається оператором select.

Структура звіту до лабораторної роботи

Для кожного з запитів:

1) постановка задачі, що вирішується;
2) SQL-код рішення;
3) скриншот отриманого результату.