МВД (DDL) SQL. Частина 2

Представлення, які модифікуються

Необхідними умовами модифікації і виводу значень через представлення є такі:

1. Представлення *не повинне містити* функцій агрегатування.
2. Представлення *не повинне використовувати* оператори GROUP BY і HAVING у своєму визначенні.
3. Для забезпечення можливості введення значень представлення *повинне включати* всі поля базової таблиці, для яких *визначене* обмеження NOT NULL.

Наприклад. Представлення

```
CREATE VIEW     AveRat    AS
      SELECT     Kod, AVG(Mark)      FROM    Rating
            GROUP BY   Kod;
```

не є обновлюваним.

А в представленні

```
CREATE VIEW     FIO      AS
      SELECT      Kod, SecondName, FirstName, Patronymic     FROM      Student;
```

можна модифікувати ПІБ, але *не можна вводити нові* кортежі, тому що ряд полів мають обмеження на NULL-значення і не мають значень за замовчуванням.

Представлення ж

```
CREATE VIEW     BadRat      AS
      SELECT   Kod, DKod, Mark, MDate    FROM      Rating   WHERE        Mark = 0;
```

є обновлюваним. Тобто за допомогою нього в базову таблицю можна і вводити, й модифікувати значення. Але команда

```
INSERT INTO     BadRat    VALUES(15,5,45, '25.12.2011');
```

буде прийнята як коректна, і додасть кортеж у таблицю Rating. Однак же *побачити* результат цієї операції, ні, тим більше, його *видалити* користувач *не зможе*, тому що нове значення не відповідає умові вибірки. Уникнути такої ситуації можна шляхом додавання у визначення представлення інструкції:

```
…     WITH CHECK OPTION;
```

Ця інструкція працює за принципом „*все або нічого*“ і блокує всі команди відновлення, результати яких не можуть бути відображені в представленні. Відповідно, команда INSERT з прикладу буде *відхилена*.

Збережені процедури (ЗП)

Збереженою процедурою називається **скомпільована програма** довільної довжини, написана мовою *SQL*, що **зберігається в БД** разом з іншими об'єктами. Збережені процедури дозволяють скоротити кількість повідомлень чи транзакцій між клієнтом і сервером.

Існує 2 типи збережених процедур: **процедури вибору** і **виконувані процедури**: процедури *вибору* ***обов'язково*** *повинні повертати значення* до викликаючої програми, а *виконувані* — ***можуть*** і ***не*** *повертати* значення.

Збережені процедури складаються з **заголовка** процедури й **тіла** процедури і створюються за допомогою команди

CREATE PROCEDURE *процедура*

Заголовок процедури містить:

—**ім'я** збереженої процедури, яке повинне бути *унікальним* серед імен процедур і таблиць у БД;

—**необов'язковий** список **вхідних параметрів** і їхніх типів даних, значення яких одержує від викликаючої програми;

—**оператор** RETURNS зі списком **вихідних параметрів** і їхніх типів даних, у тому випадку, якщо процедура повертає значення джерелу виклику.

Тіло процедури складається з:

—необов'язкового списку **локальних змінних** із вказівкою типів даних;

—**блока інструкцій**, вкладеного між операторами BEGIN і END (не у всіх СУБД). Більшість СУБД і стандарт *SQL* підтримують *багаторівневе вкладення* блоків.

Мова **збережених процедур і тригерів** *SQL* містить:

- команди ММД *SQL*: INSERT, UPDATE, DELETE і SELECT;
- оператори і вирази *SQL*;
- розширення *SQL*, що включають інструкції присвоєння, управління потоками, реєстрації подій і обробки помилок:
  - —*змінна = вираз* — операція присвоєння значення виразу локальній змінній, вхідному або вихідному параметру;
  - —INTO *список змінних* — присвоєння змінним або параметрам результату роботи команди SELECT;
  - —SUSPEND — повертає вихідні параметри викликаючій програмі і припиняє збережену процедуру доти, поки викликаюча програма не зажадає наступний кортеж. *Не рекомендується використовувати у виконуваних процедурах*;
  - —FOR *SELECT-вираз* DO *складений_оператор* — *складений_оператор* — інструкція або блок інструкцій, який буде виконаний для *кожного рядка*, повернутого *SELECT-виразом*, де *SELECT-вираз* — звичайна команда SELECT, за винятком того, що наприкінці команди має *обов'язково* бути присутнім оператор INTO;

—IF(умова) THEN складений_оператор [ELSE складений_оператор]

—WHILE(*умова*) DO *складений_оператор* - цикл з передперевіркою;

—EXCEPTION *виняток* — генерує пойменовану виняткову ситуацію *виняток* — визначену користувачем помилку, яка може бути оброблена оператором WHEN;

—WHEN{*помилка*[, *помилка*[, …]] | ANY} DO *складений_оператор* — обробка помилок, вказаних у списку, або будь-яких помилок, якщо вказано ANY. Ця інструкція, якщо є присутньою, вказується *наприкінці* збереженої процедури безпосередньо перед END;

—EXIT — перехід на кінець збереженої процедури (останній END), наприклад, для переривання циклу;

—EXECUTE PROCEDURE *процедура* [*параметр*[, *параметр*[, …]]] [RETURNING_VALUES [*параметр*[, *параметр*[, …]]] — виконує збережену процедуру з ім'ям *процедура*, вхідними параметрами, переліченими після імені, і яка повертає значення в параметрах, перелічених після RETURNING_VALUES.

Вхідними і вихідними параметрами повинні бути змінні, визначені *всередині поточної* процедури. Завдяки цій інструкції можливі *вкладені виклики* процедур і *рекурсії*.

—/* *коментар* */ — для коментарів.

Наприклад, *процедура вибору*, яка повертає інформацію про кожну групу: чисельність, мінімальний, середній і максимальний рейтинг з кожної дисципліни:

```
SET TERM  ;  ^
CREATE PROCEDURE  GroupInfo
    RETURNS  (Spec  CHAR(2), GNum  INT, DName  CHAR(30), NumOfStud INT, MinRat  DEC, AveRat  DEC, MaxRat DEC)
AS
BEGIN
    FOR  SELECT  S.Spec, GNum, DName, COUNT(S.Kod), MIN(Mark), AVG(Mark), MAX(Mark)
            FROM  Student S, Rating R, Discipline D  WHERE  S.Kod = R.Kod  AND  D.DKod=R.Dkod
            GROUP BY  S.Spec, GNum, DName
            INTO  :Spec, :GNum, :DName, NumOfStud, :MinRat, :AveRat, :MaxRat
    DO
        SUSPEND;
END^
SET TERM  ^  ;
```

Виклик збереженої процедури:

```
SELECT  *  FROM  GroupInfo;
```

що дасть такий результат:

Spec	GNum	DName	NumOfStud	MinRat	AveRat	MaxRat
AC	954	Економіка	1	97	97	97
AI	943	Мережеві технології	2	76	81	85
AM	971	Мережеві технології	1	10	10	10
AC	102	Технології проектування БД	1	46	46	46
AC	102	Мережеві технології	1	85	85	85
AM	951	Технології проектування БД	1	0	0	0
AI	943	Технології проектування БД	2	54	72	90
AM	971	Дослідження операцій	1	75	75	75
АП	971	Технології проектування БД	1	86	86	86

Інший приклад використання збереженої процедури, якою можна скористатися в прикладі поділу групи:

```
SELECT  Spec, MAX(GNum)  FROM  GroupInfo  GROUP BY  Spec;
```

У процедуру можна передавати параметри. Наприклад, процедура вибору, яка повертає коди студентів певної групи:

```
SET TERM  ;  ^
CREATE PROCEDURE  StudGroup  (Spec CHAR(2), GNum INT)
    RETURNS  (Kod INT)
AS
BEGIN
    FOR  SELECT  Kod  FROM  Student  WHERE  Spec = :Spec AND GNum = :Gnum
         INTO  :Kod
    DO
         SUSPEND;
END^
SET TERM  ^  ;
```

Скористатися цією збереженою процедурою можна, задавши параметри *в явному вигляді*. Наприклад:

```
SELECT  *  FROM  StudGroup('AC', 941);
```

Не менш ефективним буде виклик цієї збереженої процедури *в підзапиті*. Наприклад:

```
SELECT  S.Spec, GNum, AVG(Mark)  FROM  Student S, Rating R  WHERE  R.Kod  IN
        (SELECT  Kod  FROM  StudGroup  (S.Spec, S.GNum))  GROUP BY  S.Spec, GNum;
```

Виконувані процедури, як видно з назви, повинні виконувати які-небудь дії. Наприклад, збережена процедура, яка видаляє усіх випускників з таблиці Student і їхній рейтинг із таблиці Rating:

```
SET TERM    ;     ^

CREATE PROCEDURE       RemoveDiploma      (GNum INT)
AS
DECLARE VARIABLE       Kod       INT;
BEGIN
        FOR      SELECT       Kod       FROM       Student WHERE   GNum = :Gnum
                 INTO         :Kod
        DO
                 BEGIN
                         DELETE       FROM       Rating    WHERE     Kod = :Kod;
                 END
        DELETE       FROM       Student      WHERE        GNum = :GNum;
END^

SET TERM    ^     ;
```

Так як ця збережена процедура *не повертає значення* і *знищує при цьому кортежі*, то використовувати її в команді SELECT *не можна*. Для виклику таких процедур використовується команда розширення *SQL* — EXECUTE PROCEDURE *процедура*.

```
EXECUTE PROCEDURE        RemoveDiploma      (961);
```

Особливості побудови збережених процедур в СУБД PostgreSQL

Мова програмування PLpg/SQL СУБД PostgreSQL. Структура підпрограм PLpg/SQL

На відміну від збережених процедур стандартного *SQL, PLpg/SQL* підтримує тільки *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)

С***труктура 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.
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. 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* або властивий діалекту *PLpg/SQL*. Наприклад, необхідно оголосити змінну з ім'ям *Tax* так, щоб вона могла зберігати 4-хзначні числові значення, і змінну з ім'ям *Valid*, що може приймати булеве значення TRUE або FALSE:

```
…   Tax     NUMBER(4),
    Valid   BOOLEAN…
```

Також є можливість оголошувати записи і таблиці, використовуючи складні типи даних: RECORD і TABLE.

Присвоєння значень змінним

– оператор присвоювання ':='. Наприклад:

```
Tax := Price * Tax_Rate;
Bonus := Current_Salary * 0.10;
```

– введення в змінну значення з БД за допомогою фрази INTO команди SELECT. 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", 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;…
```

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. Така змінна, оголошена з атрибутом %ROWTYPE, може зберігати цілий рядок даних, отриманий з таблиці.

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* існують ***команди передачі управління***: оператор *умовного переходу* і оператори *циклів*.

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

Для звернення до цієї функції треба виконати такий запит:

```
SELECT    Попередження_if_else();
```

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 командою SELECT побачимо у результаті значення 1, а в таблиці Letters відповідні записи. Якщо в студента з кодом 5 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).

Тригери

Тригери — це ***підпрограми***, написані SQL, які **виконуються тоді, коли відбувається певна *подія*, спрямована до конкретної таблиці**.

Такою подією може бути виконання якоїсь команди відновлення ММД *SQL*: вставки, відновлення або видалення даних. Відповідно **кожен тригер асоціюється з *конкретною* операцією і *конкретною* таблицею**. Така відповідність встановлюється при створенні тригера командою CREATE TRIGGER. Формат цієї команди в різних СУБД трохи відрізняється. Тому спочатку розглянемо побудову тригерів в стандартному *SQL*, а потім — особливості роботи з тригерами з використанням діалекту *PostgreSQL*.

Тригери можуть використовуватися в наступних областях функціонування додатків:

— реалізація контролю за всіма діями користувачів БД;
— забезпечення цілісності змісту БД;
— реалізація складних правил роботи програм;
— забезпечення складних правил безпеки даних;
— автоматичне створення значень у полях таблиць БД.

Як приклад забезпечення цілісності розглянемо задачу видалення кортежів, що відносяться до випускників вузу, із двох таблиць: *Student* і *Rating*. Ця задача розв'язувалась шляхом виклику виконуваної процедури, тому що було необхідно видалити *спочатку* кортежі з *деталізованої* таблиці, а *потім* — з *базової*. Розв'язання цієї задачі:

```
CREATE TRIGGER      DelRating     FOR     Student
ACTIVE BEFORE DELETE
AS   BEGIN
          DELETE      FROM      Rating R WHERE      R.Kod = OLD.Kod;
     END
```

Зверніть увагу, що цей тригер працює з таблицею *Rating*, проте „прив'язаний" до таблиці *Student* і направленої до неї події DELETE.

У результаті розв'язання поставленої задачі зведеться до єдиного (як і при збереженій процедурі) запиту:

```
DELETE   FROM      Student      WHERE       GNum = 941;
```

У команді CREATE TRIGGER додався ще ряд інструкцій, характерних *тільки* для *мови збережених процедур і тригерів*:

Параметр	**Опис**
ACTIVE	необов'язковий оператор, який використовується за замовчуванням (тільки в *InterBase*), призначений *для активації* тригера. За допомогою інструкції ALTER TRIGGER INACTIVE у *InterBase* можна „*відключити*“ тригер, не знищуючи його.
BEFORE	говорить про те, що тригер повинен бути активований *перед* виконанням команди, з яким пов'язаний тригер (INSERT, UPDATE або DELETE). Протилежна їй інструкція — AFTER (активація *після* виконання операції).
OLD	**контекстна змінна**, що застосовується *для посилання на значення атрибута*, яке *було* в кортежі ***перед*** виконанням операції UPDATE або DELETE. Відповідно, контекстна змінна NEW дозволяє послатися на *нове значення стовпця* ***при*** виконанні операції UPDATE або INSERT.

Наприклад:

```
CREATE TRIGGER    AddNewRating    FOR    Student
AFTER INSERT
AS    BEGIN
            INSERT INTO    Rating    (Kod)    VALUES(NEW.Kod);
      END
```

Цей тригер дозволяє при виконанні команди

```
INSERT INTO    Student VALUES( 20, 'Aaa', 'Ббб', NULL, 5, 'OI', 981, NULL, NULL);
```

автоматично додати кортеж у таблицю *Rating* з *таким же* значенням поля *Kod* та нульовим значенням рейтингу (значення за замовчуванням).

У СУБД ***PostgreSQL*** тригер складається із двох частин: *тригерна подія*, *дії тригера*.

Тригерна подія описується у вигляді:

CREATE OR REPLACE TRIGGER *Ім'я_тригера Тип_події Подія*
ON *Ім'я_таблиці Область_застосування*;

Параметр *Тип_події* (BEFORE або AFTER) і параметр *Ім'я_таблиці* (операції модифікації над якою перехоплюються тригером) нічим не відрізняються від використовуваних у стандартному *SQL*, як і параметр *Подія*, який визначає тип *SQL*-операції. Але в цьому діалекті при операції UPDATE можна вказувати ім'я атрибута таблиці, при впливі на який спрацьовує тригер.

Основною особливістю тригерної події *PLpg/SQL* є параметр *Область_застосування*, що може мати наступні значення:

FOR EACH ROW тригер спрацьовує стільки разів, скільки операція модифікації (UPDATE) впливає на записи таблиці (якщо при операції модифікації не був порушений жодний запис, то тригер не спрацює жодного разу);

FOR STATEMENT тригер спрацьовує лише один раз разом із самою операцією модифікації, навіть якщо операція модифікації не торкнулася жодного запису таблиці.

При використанні множинних подій, коли тригер може перехоплювати відразу кілька типів подій (INSERT, UPDATE, DELETE) у підпрограму *PLpg/SQL* необхідно включити системні логічні змінні INSERTING, UPDATING, DELETING, які мають тип даних BOOLEAN (TRUE, FALSE) і повертають TRUE, якщо тригер спрацював при операції INSERT, UPDATE, DELETE відповідно.

Дії тригера — це збережена процедура у вигляді функції (FUNCTION), яка виконується при спрацьовуванні тригера і повертає значення типу TRIGGER. Якщо процедура спроектована з використанням мови *PLpg/SQL*, то, крім стандартних *контекстних змінних* NEW. і OLD., можна також використовувати наступні *системні змінні*:

TG_NAME ім'я тригера, який виконується;
TG_WHEN значення параметра *Тип_події* з тригерної події;
TG_LEVEL значення параметра *Область_застосування* з тригерної події;
TG_OP тип операції модифікації, яка перехоплена тригером;
TG_RELNAME ім'я таблиці, над якою виконується операція модифікації;
TG_NARGS кількість параметрів збереженої процедури, яка входить у дії тригера;
TG_ARGV[] масив значень параметрів збереженої процедури (індекс починається з 0).

Приклад. Нехай у БД існує таблиця *Employer* зі структурою:

```
CREATE TABLE    Employer
    (Emp_Num    INTEGER,        Name  CHAR(40),        Job     CHAR(20),        Tax     NUMERIC(4,2));
```

Необхідно відслідковувати зміни над таблицею *Employer* через збереження їх у таблицю *Audit_Employer*, структура якої може мати вигляд:

```
CREATE TABLE    Audit_Employer
(Emp_Num INTEGER, Name CHAR(40), Job CHAR(20), Tax NUMERIC(4,2), Oper_type CHAR(1), User_Name CHAR(20), Change_time TIMESTAMP);
```

Для цього створимо тригер, який перехоплює всі операції модифікації над таблицею *Employer* і вносить їх у таблицю *Audit_Employer*. Опис тригерної події та дій тригера представлено нижче.

```
CREATE OR REPLACE FUNCTION Audit_Employer_F() RETURNS TRIGGER
    AS $$
        BEGIN
            IF (Tg_op ='INSERT' OR Tg_op ='UPDATE') THEN
                INSERT INTO Audit_Employer VALUES
                    (NEW.Name, NEW.Job, NEW.Tax, SUBSTR(Tg_op,1,1), CURRENT_USER, CURRENT_TIME);
                RETURN NEW;
            ELSE
                INSERT INTO Audit_Employer VALUES
                    (OLD.Name, OLD.Job, OLD.Tax, SUBSTR(Tg_op,1,1), CURRENT_USER, CURRENT_TIME);
                RETURN OLD;
            END IF;
        END;
    $$ LANGUAGE plpgsql;

CREATE TRIGGER Audit_Employer AFTER INSERT OR UPDATE OR DELETE ON Employer
    FOR EACH ROW EXECUTE PROCEDURE Audit_Employer_F();
```

У представленому тригері застосовуються наступні системні функції:

CURRENT_USER — ім’я користувача, який виконує операції;

CURRENT_TIME — поточний час;

SUBSTR(*Рядок*, *Початковий_номер*, *Кількість_символів*) — функція, що повертає з рядка підрядок, який починається з символу з номером (*Початковий_номер* + 1) і має довжину *Кількість_символів*.

Тригери, які описуються конструкціями діалекту *PLpg/SQL* можуть забезпечувати більш ***складні правила цілісності***, ніж це передбачено в стандарті.

Сценарії

Для того, щоб упорядкувати виклик і полегшити створення послідовності **незв'язаних** запитів, наприклад, команд МВД *SQL* для створення таблиць, збережених процедур, тригерів тощо, їх доцільно включати в так звані **сценарії** (*script*). Це файли, що мають стандартне розширення *SQL*, які можна створювати і редагувати в будь-якому текстовому редакторі.

Відзначимо тільки два нюанси:

1. Переважна більшість сценаріїв повинна починатися командою

   CONNECT ім'я/аліас БД USER ім'я_користувача PASSWORD пароль;

   і закінчуватися командами

   COMMIT;

   та

   EXIT;

   Якщо створюються зв'язані таблиці або для таблиць створюються представлення, індекси та ін., то командою COMMIT (зберігання результатів транзакції, тобто виконаних дій) повинна закінчуватися кожна команда CREATE TABLE.

   Якщо в параметрах СУБД встановлено опцію AUTOCOMMIT, то додавати цю команду до сценарію не треба.

   Команда, що виконує дію, зворотну COMMIT, тобто відміняє результати транзакції — ROLLBACK („*відкіт*" виконаних операцій).

2. Друга тонкість зв'язана зі створенням збережених процедур за допомогою сценаріїв (і не тільки). Справа в тому, що *в усіх без винятку СУБД* команди, з яких складається збережена процедура, повинні *обов'язково* закінчуватися символом ';'. Але при цьому більшість СУБД вимагає, щоб цим символом закінчувалися і всі запити в сценарії. Тому, якщо не прийняти відповідних заходів, то ці СУБД будуть намагатися виконувати команди збереженої процедури в порядку її створення. Для того щоб уникнути такої ситуації, необхідно *кожну* або *блок команд CREATE PROCEDURE* випереджати командою

   SET TERM[INATOR] *символ* ;

   де *символ* — наприклад, '^', і закінчувати кожну команду створення збереженої процедури цим **символом-замінником** (подібно оператору END), а після її або в самому кінці зробити *зворотне перепризначення*:

   SET TERM ; *символ*