Лабораторна робота 9. Збережені процедури, користувальницькі функції, тригери в PostgreSQL

Мета: Ознайомити студентів з реалізацією процедур, функцій та тригерів в PostgreSQL.

Завдання:

1. Виконати SQL-оператори для створення та запуску збережених процедур;
2. Виконати SQL-оператори для створення та запуску користувальницьких функцій;
3. Виконати SQL-оператори для створення та запуску тригерів на різні операції маніпулювання даними.

Результат:

Студенти повинні подати SQL-скрипти, що відображають створення та використання збережених процедур, користувальницьких функцій та тригерів на різні операції маніпулювання даними згідно завдання та предметної області, їх опис, а також звіт з результатами тестування.

Теоретичні відомості до виконання лабораторної роботи

SQL, як і будь-яка інша мова програмування, надає функції та збережені процедури. Функції та збережені процедури в SQL, як і в будь-якій іншій мові програмування, забезпечують можливість повторного використання і гнучкість. Функції та збережені процедури являють собою блок коду або запитів, що зберігаються в базі даних, які можна використовувати знову і знову. Замість того, щоб писати одні й самі запити, зручніше згрупувати всі запити та зберегти їх, щоб можна було використовувати їх багато разів. Що стосується гнучкості, то щоразу, коли відбувається зміна логіки запитів, можна передавати новий параметр функцій і збережених процедур. Між функціями і збереженими процедурами в PostgreSQL є кілька відмінностей. Вони показані у таблиці нижче.

Функції	**Збережені процедури**
Функція має тип, що повертається, і повертає значення	Збережена процедура не має типу, що повертається, але має вихідні аргументи
Використання DML (insert, update, delete) запитів усередині функції неможливе. У функціях дозволені лише SELECT-запити	Використання DML-запитів (insert, update, delete) можливе в збереженій процедурі.
Функція не має вихідних аргументів	Збережена процедура має і вхідні, і вихідні аргументи
Виклик збереженої процедури з функції неможливий	Використання або управління транзакціями можливе в збереженій процедурі.
Виклик функції всередині SELECT запитів можливий	Виклик збереженої процедури із SELECT запитів неможливий

Збережені процедури

Синтаксис створення процедур в PostgreSQL:

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } ] [, ...] ] )
  { LANGUAGE lang_name
    | sql_body
  } ...
```

Опис

CREATE PROCEDURE визначає нову процедуру. CREATE OR REPLACE PROCEDURE створить нову процедуру або замінить існуюче визначення. Таким чином неможливо змінити ім’я або типи аргументів процедури. Коли CREATE OR REPLACE PROCEDURE використовується для заміни існуючої процедури, право власності та дозволи процедури не змінюються. Усім іншим властивостям процедури присвоюються значення, вказані або передбачені в команді.

Ім’я нової процедури не має збігатися з жодною існуючою процедурою чи функцією з однаковими типами вхідних аргументів. Проте процедури та функції різних типів аргументів можуть мати спільну назву (це називається перевантаженням *overloading*).

Параметри

name – ім'я процедури для створення.

argmode – режим аргументу: IN, OUT, INOUT. Якщо опущено, за умовчанням є IN.

argname – ім'я аргументу.

argtype – тип(и) даних аргументів процедури, якщо є. Типи аргументів можуть бути базовими, складеними чи доменними, або можуть посилатися на тип стовпця таблиці.

Тип стовпця вказується шляхом запису *table_name.column_name*%TYPE. Використання цієї функції іноді може допомогти зробити процедуру незалежною від змін у визначенні таблиці.

*lang_name – н*азва мови, на якій реалізовано процедуру. Це може бути *sql, c, internal* або назва визначеної користувачем процедурної мови, наприклад, *plpgsql*. Взяття імені в одинарні лапки застаріло та потребує відповідності регістру.

sql_body

Тіло процедурив стилі LANGUAGE SQL. Це має бути блок

```
BEGIN ATOMIC
  statement;
  statement;
  ...
  statement;
END
```

Це схоже на написання тексту тіла процедури як рядкової константи, але є деякі відмінності: ця форма працює лише для LANGUAGE SQL, форма рядкової константи працює для всіх мов. Ця форма аналізується під час визначення процедури, форма рядкової константи аналізується під час виконання.

Часто корисно використовувати доларові котирування $$, щоб написати рядок визначення процедури, а не звичайний синтаксис одиночних лапок. Без $$ будь-які одинарні лапки або зворотні косі риски у визначенні процедури мають бути подвоєні.

Режими параметрів визначають поведінку параметрів. PL/pgSQL підтримує три режими параметрів: IN, OUT і INOUT. Параметр приймає режим IN за замовчуванням, якщо він не вказується явно. Наступна таблиця ілюструє режими параметрів:

IN	OUT	INOUT
За замовчуванням	Явно вказано	Явно вказано
Передає значення в функцію	Повертає значення з функції	Передає значення в функцію та повертає оновлене значення
Параметри діють як константи	Параметри діють як неініціалізовані змінні	Параметри діють як ініціалізовані змінні
Неможливо призначити значення	Потрібно призначити значення	Потрібно призначити значення

Викликати процедуру можна за допомогою оператора *CALL*. Якщо процедура має будь-які вихідні параметри, буде повернено рядок результату, що містить значення цих параметрів.

Синтаксис

CALL *name* ([*argument*] [, ...])

Параметри

name – *і*м'я процедури.

argument – вираз аргументу для виклику процедури. Аргументи можуть включати імена параметрів, використовуючи синтаксис name => value. Це працює так само, як у звичайних викликах функцій. Аргументи повинні бути надані для всіх параметрів процедури, які не мають стандартних значень, включаючи параметри OUT. Однак аргументи, що відповідають параметрам OUT, не оцінюються, тому для них прийнято просто писати NULL.

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.

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***.

Приклади

Приклад 1. *Процедура додає рядок в таблицю жанр, що включає в себе цілочисельне та символьне значення*

```
CREATE OR REPLACE PROCEDURE genre_insert_data (GenreId INTEGER, Name CHARACTER VARYING) -- оголошення процедури, що приймає 2 параметри IN
LANGUAGE SQL -- мова програмування
AS $$
INSERT INTO Genre VALUES (GenreId, Name); -- вставляємо в таблицю Genre значення
$$;
```

Викликаємо процедуру:

```
CALL genre_insert_data(26,'Pop'); -- передаємо два параметра
```

Перевіряємо результат:

```
SELECT * FROM Genre WHERE GenreId = 26;
```

```
 GenreId | Name
---------+------
   26 | Pop
```

Приклад 2. *Відображення повідомлення на екрані*

```
CREATE OR REPLACE PROCEDURE display_message (INOUT msg TEXT)
AS $$
```

```
BEGIN -- початок тіла процедури
RAISE NOTICE 'Procedure Parameter: %', msg ; -- оператор виведення повідомлення на екран
END ; -- кінець
$$
LANGUAGE plpgsql ; -- процедурна мова програмування PL/pgSQL
```

Викликаємо процедуру:

```
CALL display_message('This is my test case');
```

Результат:

```
NOTICE:  Procedure Parameter: This is my test case
     msg
----------------------
 This is my test case
(1 row)
```

Приклад 3. *Наступна процедура в декларативній секції використовує оголошення змінної, що має тип даних, як стовпець GenreId в таблиці Genre, а далі виводить повідомлення, що містить максимальне значення цього стовпця*.

```
CREATE OR REPLACE PROCEDURE genre_id_max()
LANGUAGE plpgsql
AS $$
DECLARE -- секція оголошення змінних
   id Genre.GenreId%TYPE; -- id отримує той тип даних, що і стовпець Genre в табл. Genre
BEGIN
   SELECT MAX(GenreId) INTO id FROM Genre;
   RAISE NOTICE 'Maximum of GenreId is : %', id ;
END;
$$ ;
```

Виклик процедури:

```
CALL genre_id_max();
```

Відображення результату:

```
NOTICE:  Maximum of GenreId is : 26
```

Приклад 4. *Виведення сповіщень, попереджень та INFO-повідомлень: вбудована функція часу now() виводить поточну дату і час з зазначенням часової зони. Оголошується змінна warn цілочисельного типу, яка ініціалізується значенням 10.*

Використовуються оператори виведення повідомлень RAISE *level*, де *level:* INFO, NOTICE, WARNING, які відрізняються рівнями пріоритету, але не переривають на відміну від EXCEPTION поточну транзакцію. Після *level*, можете вказати рядок формату, який має бути простим рядковим літералом, а не виразом. Рядок формату визначає текст повідомлення, який потрібно повідомити. Після рядка формату можуть слідувати необов'язкові вирази аргументів, які потрібно вставити в повідомлення. У рядку формату % замінюється рядковим представленням значення наступного необов'язкового аргументу. Для того, щоб включити

символ «%» в виведений текст, потрібно використовувати дублювання символу – %%. Кількість аргументів має відповідати кількості % заповнювачів у рядку формату, інакше під час компіляції процедури виникне помилка.

```
CREATE OR REPLACE PROCEDURE raise_warning()
AS $$
DECLARE
   warn INT := 10; -- змінна warn цілочисельного типу отримує значення 10
BEGIN
   RAISE NOTICE 'value of warn : % at %: ', warn, now();
   warn := warn + 10;
   RAISE WARNING 'value of warn : % at %: ', warn, now();
   warn := warn + 10;
   RAISE INFO 'value of warn : % at %: ', warn, now();
END;
$$
LANGUAGE plpgsql;
```

Виклик процедури:

```
CALL raise_warning();
```

Результат вивклику:

```
NOTICE:  value of warn : 10 at 2019-12-03 16:25:34.339094+05:30:
WARNING:  value of warn : 20 at 2019-12-03 16:25:34.339094+05:30:
INFO:  value of warn : 30 at 2019-12-03 16:25:34.339094+05:30:
```

Приклад 5. *Створення винятків: оператор EXCEPTION перериває виконання процедури*. Оператор USING HINT у виразі RAISE – надає підказку. Наступна процедура відрізняється тим, що видає помилку, перериваючи виконання процедури, із виведенням повідомлення, прописаного в EXCEPTION та підказки в HINT.

```
CREATE OR REPLACE PROCEDURE genre_id_exception()
LANGUAGE plpgsql
AS $$
DECLARE
   id Genre.GenreId%TYPE;
BEGIN
   SELECT MAX(GenreId) INTO id FROM Genre;
   RAISE EXCEPTION 'Maximum of GenreId is : %', id
   USING HINT = 'Test For Raising exception.';
END;
$$ ;
```

Виклик процедури:

```
CALL genre_id_exception();
```

Результат:

```
ERROR:  Maximum of GenreId is : 26
HINT:  Test For Raising exception.
CONTEXT:  PL/pgSQL function genre_id_exception() line 6 at RAISE
```

Приклад 6. *Обхід значень у таблиці за допомогою циклу FOR*. *Псевдо-тип record* ідентифікує функцію, яка приймає або повертає невизначений тип рядка. Генерується *цикл, що проходить по всій таблиці* Genre за допомогою *змінної genre_rec*, що приймає значення як результат запиту до таблиці, впорядкованої за id. В тілі циклу видається повідомлення NOTICE.

```
CREATE OR REPLACE PROCEDURE genre_traverse()
LANGUAGE plpgsql
AS $$
DECLARE
   genre_rec record; --змінна genre_rec отримує псевдо-тип record
BEGIN
  FOR genre_rec IN
            (SELECT GenreId, Name FROM Genre ORDER BY GenreId) --оголошуємо цикл for що буде проходити по всім рядкам таблиці
  LOOP --початок тіла циклу
            RAISE NOTICE 'Genre Id is : % , Name is : %', genre_rec.GenreId, genre_rec.Name;
  END LOOP; -- кінець тіла циклу
END;
$$;
```

Виклик процедури:

```
CALL genre_traverse();
```

Результат виконання:

```
NOTICE:  Genre Id is : 1 , Name is : test
NOTICE:  Genre Id is : 2 , Name is : Jazz
NOTICE:  Genre Id is : 3 , Name is : Metal
NOTICE:  Genre Id is : 4 , Name is : Alternative & Punk
…
NOTICE:  Genre Id is : 25 , Name is : Opera
NOTICE:  Genre Id is : 26 , Name is : Pop
```

Змінити визначення процедури можна командою ALTER PROCEDURE

Синтаксис

```
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
   RENAME TO new_name
```

Параметри

name – ім'я існуючої процедури, якщо список аргументів не вказано, ім’я має бути унікальним у його схемі.

argname – ім'я аргументу. ALTER PROCEDURE насправді не звертає уваги на імена аргументів, оскільки для визначення ідентичності процедури використовуються лише типи даних аргументів.

new_name – нова назва процедури.

Приклад: перейменувати процедуру insert_data з двома аргументами типу integer на insert_record:

```
ALTER PROCEDURE insert_data(integer, integer)
RENAME TO insert_record;
```

Видаляє визначення однієї чи кількох існуючих процедур команда DROP PROCEDURE. Типи аргументів для процедури(-й) зазвичай повинні бути визначені, оскільки може існувати кілька різних процедур з однаковою назвою та різними списками аргументів.

Синтаксис

DROP PROCEDURE [IF EXISTS] *name* [([[*argmode*] [*argname*] *argtype* [, ...]])] [, ...]
[CASCADE | RESTRICT]

Параметри

IF EXISTS – не видавати помилку, якщо процедура не існує. У цьому випадку видається повідомлення.

CASCADE – автоматично видаляти об'єкти, які залежать від процедури, і, у свою чергу, усі об'єкти, які залежать від цих об'єктів.

RESTRICT – забороняє видаляти процедуру, якщо від неї залежать будь-які об'єкти. Це значення за умовчанням.

Користувальницькі функції

Синтаксис створення користувальницьких функцій в СУБД PostgreSQL наступний:

CREATE [OR REPLACE] FUNCTION
name ([[*argmode*] [*argname*] *argtype* [{ DEFAULT | = } *default_expr*] [, ...]])
[RETURNS *rettype*
| RETURNS TABLE (*column_name column_type* [, ...])]
{ LANGUAGE *lang_name*
| { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
| *sql_body*
} ...

Опис

CREATE FUNCTION визначає нову функцію. CREATE OR REPLACE FUNCTION створить нову функцію або замінить існуюче визначення. Ім'я нової функції не має збігатися з жодною існуючою функцією чи процедурою з однаковими типами вхідних аргументів у тій же схемі. Однак функції та процедури різних типів аргументів можуть мати спільну назву (це називається перевантаженням).

Щоб замінити поточне визначення існуючої функції, використовуйте CREATE OR REPLACE FUNCTION. Але дана команда не дозволяє змінити ім'я чи аргументи функції. При спробі таких дій, фактично буде створено нову, незележну функцію. Крім того, CREATE OR REPLACE FUNCTION не дозволить змінити тип результату наявної функції. Щоб зробити це, потрібно видалити та створити функцію заново.

Якщо видалити функцію, а потім створити її знову, нова функція не буде тією самою сутністю, що й стара; доведеться видалити існуючі правила, представлення, тригери тощо, які посилаються на стару функцію. Потрібно використовувати CREATE OR REPLACE FUNCTION, щоб змінити визначення функції, не порушуючи об'єктів, які посилаються на функцію. Крім того, ALTER FUNCTION можна використовувати для зміни більшості допоміжних властивостей існуючої функції.

Параметри

name – ім'я (необов'язково доповнене схемою) функції, яку потрібно створити.

argmode – режим аргументу: IN, OUT, INOUT. Крім того, аргументи OUT та INOUT не можна використовувати разом із нотацією RETURNS TABLE.

argname – ім'я аргументу. Деякі мови (зокрема SQL і PL/pgSQL) дозволяють використовувати цю назву в тілі функції. Ім'я вихідного аргументу є важливим, оскільки воно

визначає ім’я стовпця в типі рядка результату. (Якщо опустити назву вихідного аргументу, система вибере назву стовпця за замовчуванням.)

argtype – ім'я аргументу.

default_expr – вираз, який буде використовуватися як значення за умовчанням, якщо параметр не вказано.

rettype – тип даних, що повертаються в результат. Тип результуючих даних може бути базовим, складеним або доменним типом або може посилатися на тип стовпця таблиці. Якщо функція не має повертати значення, можна вказати тип void як тип результату.

Якщо є параметри OUT або INOUT, пропозицію RETURNS можна опустити. Якщо він присутній, він має узгоджуватися з типом результату, що передбачається вихідними параметрами: RECORD, якщо є кілька вихідних параметрів, або того самого типу, що й один вихідний параметр.

Тип стовпця вказується шляхом запису *table_name.column_name*%TYPE.

column_name – ім'я вихідного стовпця в синтаксисі RETURNS TABLE. Фактично це інший спосіб оголошення іменованого параметра OUT.

column_type – тип даних вихідного стовпця в синтаксисі RETURNS TABLE.

lang_name – назва мови, на якій реалізовано функцію.

CALLED ON NULL INPUT (за замовчуванням) вказує на те, що функція буде викликана нормально, якщо деякі з її аргументів є NULL-значеннями.

RETURNS NULL ON NULL INPUT або STRICT вказує на те, що функція завжди повертає NULL, коли будь-який з її аргументів має NULL-значення. Така функція не буде викликатись з аргументами NULL, натомість автоматично буде повертатись результат NULL.

sql_body

Тіло функції LANGUAGE SQL. Це може бути або одне зоголошення

RETURN *expression*

Або блок

```
BEGIN ATOMIC
  statement;
  statement;
  ...
  statement;
END
```

Виклик користувальницької функції

Користувальницькі функції повертають задане значення, отже викликаються оператором SELECT.

PostgreSQL надає три способи виклику визначеної користувачем функції:

— Використання позиційного запису.
— Використання іменованого позначення.
— Використання змішаної нотації.

1) Використання позиційного запису

Викликаючи функцію з використанням позиційної нотації, потрібно надати аргументи в тому порядку, в якому параметри визначені в синтаксисі функції. Такий тип виклику зазвичай використовується, якщо функція має мало параметрів. Якщо функція має багато параметрів, краще викликати її, використовуючи іменовану нотацію, оскільки це зробить виклик функції більш очевидним.

2) Використання іменованого запису

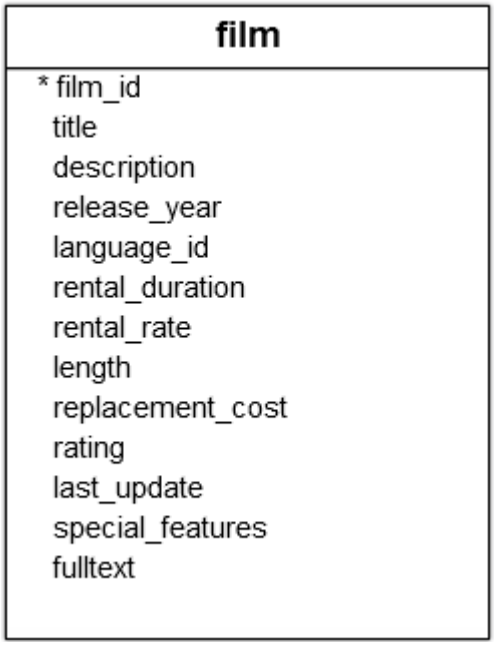
В іменованій нотації використовується знак «=>», щоб розділити ім’я аргументу та його значення. Для зворотної сумісності PostgreSQL підтримує старіший синтаксис на основі знаку «:=».

3) Використання змішаної нотації

Змішана нотація — це комбінація позиційної та іменованої нотацій. Але не можна використовувати іменовані аргументи перед позиційними аргументами. В такому випадку сервер поверне помилку.

Приклади

Будемо використовувати таблицю фільмів з наступною схемою відношення:



Створюємо функцію, яка повертає кількість фільмів, довжина яких між параметрами len_from і len_to:

```
CREATE FUNCTION get_film_count (len_from INT, len_to INT) --назва та аргументи
RETURNS INT --тип значення, що повертає (цілий)
LANGUAGE plpgsql --мова програмування
AS --тіло функції
$$
DECLARE --секція оголошення змінних
  film_count INTEGER; --цілочисельна змінна film_count
BEGIN --блок виконання команд
  SELECT COUNT(*)
        INTO film_count --заносимо кількість у змінну film_count
  FROM film
  WHERE length BETWEEN len_from AND len_to;
  RETURN film_count; --повертаємо змінну
END;
$$;
```

Функція get_film_count має два основні розділи: заголовок та тіло функції.

У розділі заголовка: назва функції — get_film_count, що слідує за ключовими словами функції CREATE; параметри з цілим типом даних, які приймає функція: len_from і len_to; пропозиція RETURNS INT визначає, що функція повертає ціле число; мовою функції є PL/pgSQL, яка вказується мовою plpgsql.

У тілі функції:

Використовується синтаксис строкової константи в лапках, який починається на $$ і закінчується на $$. Між цими $$ можна розмістити блок, що містить оголошення та логіку функції.

У розділі оголошень оголошується змінна film_count, яка зберігає кількість фільмів із таблиці фільмів.

У тілі блоку використовується оператор SELECT INTO, щоб вибрати кількість фільмів, довжина яких знаходиться між len_from і len_to, і призначити її змінній film_count.

У кінці блоку використовується оператор RETURN, щоб повернути film_count.

Приклади виклику функцій різними типами.

1) Використання позиційного запису:

```
select get_film_count(40,90);
```

Результат:

```
get_film_count
---------------
          325
(1 row)
```

У цьому прикладі аргументи get_film_count() дорівнюють 40 і 90, що відповідає параметрам from_len і to_len.

2) Використання іменованого запису

```
select get_film_count(
   len_from => 40,
    len_to => 90
);
```

Результат:

```
get_film_count
---------------
          325
(1 row)
```

Або використовуючи старіший синтаксис:

```
select get_film_count(
   len_from := 40,
   len_to := 90
);
```

3) Використання змішаної нотації

```
select get_film_count(40, len_to => 90);
```

При використанні іменованих аргументів перед позиційними, повернеться помилка:

```
select get_film_count(len_from => 40, 90);
```

Помилка:

```
ERROR:  positional argument cannot follow named argument
LINE 1: select get_film_count(len_from => 40, 90);
```

Наступна функція знаходить фільм за його ідентифікатором і повертає назву фільму. Оскільки режим для параметра p_film_id не вказано, за замовчуванням він приймає режим IN:

```
CREATE OR REPLACE FUNCTION find_film_by_id (p_film_id INT)
RETURNS VARCHAR
LANGUAGE plpgsql
AS $$
DECLARE
  film_title film.title%TYPE;
BEGIN
 -- find film title by id
 SELECT title
  INTO film_title
 FROM film
 WHERE film_id = p_film_id;

 IF NOT FOUND THEN
   RAISE 'Film with id % not found', p_film_id;
 END IF;

 RETURN film_title;

END;$$
```

Викликаємо функцію find_film_by_id(), щоб знайти назву фільму з ідентифікатором 100:

```
SELECT * FROM find_film_by_id(100);
```

Результат:

```
find_film_by_id
------------------
 Academy Dinosaur
(1 row)
```

У наступному прикладі визначено функцію get_film_stat, яка має три параметри типу OUT:

```
CREATE OR REPLACE FUNCTION get_film_stat(
   OUT min_len INT,
   OUT max_len INT,
   OUT avg_len NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN

 SELECT MIN(length), MAX(length), AVG(length)::NUMERIC(5,1)
  INTO min_len, max_len, avg_len
 FROM film;

END;
$$
```

У функції get_film_stat, вибираємо мінімальну, максимальну та середню довжину фільму з таблиці film за допомогою агрегатних функцій MIN(), MAX() та AVG() і призначаємо результати відповідним OUT параметрам.

Наступний оператор викликає функцію get_film_stat:

```
SELECT get_film_stat();
```

Результат:

```
get_film_stat
---------------
 (46,185,115.3)
(1 row)
```

Результатом функції є запис. Щоб розділити вихідні дані на стовпці, скористайтеся таким прикладом виклику:

```
SELECT * FROM get_film_stat();
```

Результат:

```
min_len | max_len | avg_len
---------+---------+---------
      46 |     185 |   115.3
(1 row)
```

Наступна функція обміну приймає два цілі числа та міняє їх значення місцями:

```
CREATE OR REPLACE FUNCTION swap(
   INOUT x INT,
   INOUT y INT
)
LANGUAGE plpgsql
AS $$
BEGIN
   SELECT x, y INTO y, x;
END;
$$;
```

Наступний оператор викликає функцію swap():

```
SELECT * FROM swap(10,20);
```

Результат:

```
x  | y
----+----
 20 | 10
(1 row)
```

Синтаксис створення функції, яка повертає таблицю:

```
CREATE OR REPLACE FUNCTION function_name ( parameter_list)
RETURNS TABLE ( column_list )
LANGUAGE plpgsql
```

```
AS
$$
DECLARE
-- оголошення змінних
BEGIN
-- тіло
END;
$$;
```

Замість того, щоб повертати одне значення, цей синтаксис дозволяє повертати таблицю з указаним списком стовпців: RETURNS TABLE (column_list).

Наступна функція повертає всі фільми, назви яких відповідають певному шаблону, використовуючи оператор ILIKE.

```
CREATE OR REPLACE FUNCTION get_film ( p_pattern VARCHAR)
RETURNS TABLE (
  film_title VARCHAR,
  film_release_year INT
)
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY
        SELECT
              title,
              release_year::INTEGER
        FROM
              film
        WHERE
              title ILIKE p_pattern;
END;
$$;
```

Функція get_film(VARCHAR) приймає один параметр p_pattern, який є шаблоном для порівняння з назвою фільму.

Функція повертає набір запитів на основі оператора SELECT. Потрібно переконатися, що стовпці в наборі результатів відповідають тим, які визначено в таблиці після пропозиції RETURNS TABLE.

Оскільки тип даних стовпця release_year з таблиці film не є цілим числом, потрібно привести його до цілого числа за допомогою оператора приведення типу «::».

Нижче показано приклад виклику функції get_film():

```
SELECT * FROM get_film ('Al%');
```

Результат:

```
film_title    | film_release_year
------------------+-------------------
 Alabama Devil    |            2006
 Aladdin Calendar |            2006
Alone Trip       |           2006
 Alter Victory    |            2006
(10 rows)
```

Приклад обробки рядків в циклі перед додаванням їх до набору результатів роботи функції:

```
CREATE OR REPLACE FUNCTION get_film (
   p_pattern VARCHAR,
   p_year INT
)
RETURNS TABLE (
   film_title VARCHAR,
   film_release_year INT
)
LANGUAGE plpgsql
AS $$
DECLARE
    var_r record;
BEGIN
   FOR var_r IN(
          SELECT title, release_year
          FROM film
          WHERE title ILIKE p_pattern AND   release_year = p_year
       ) LOOP
      film_title := upper(var_r.title) ; --перетворює до верхнього регістру рядок назви фільму
      film_release_year := var_r.release_year;
      RETURN NEXT;
   END LOOP;
END; $$
```

У цьому прикладі створюється функція get_film(VARCHAR,INT), яка приймає два параметри:

1) Шаблон p_pattern використовується для пошуку фільмів.
2) p_year – рік випуску фільмів.

У тілі функції використовується оператор циклу FOR для обробки запиту рядок за рядком. Оператор RETURN NEXT додає рядок до таблиці, що повертається функцією.

Приклад виклику функції get_film():

```
SELECT * FROM get_film ('%er', 2006);
```

Результат:

```
film_title         | film_release_year
---------------------------+------------------
 ACE GOLDFINGER            |            2006
 ALI FOREVER               |           2006
 ALIEN CENTER              |            2006
 AMISTAD MIDSUMMER         |           2006
 ARACHNOPHOBIA ROLLERCOASTER |           2006
 DYING MAKER               |            2006
 BIRDCAGE CASPER           |            2006
...
```

Змінити визначення функції можна команою ALTER FUNCTION.

Синтаксис

ALTER FUNCTION *name* [([[*argmode*] [*argname*] *argtype* [, ...]])]
RENAME TO *new_name*

Параметри

name – ім’я існуючої функції.

argmode – режим аргументу: IN, OUT, INOUT. ALTER FUNCTION насправді не звертає уваги на аргументи OUT. Отже, достатньо перерахувати аргументи IN, INOUT.

argname – ALTER FUNCTION фактично не звертає уваги на імена аргументів, оскільки для визначення ідентичності функції потрібні лише типи даних аргументу.

argtype – тип(и) даних аргументів функції, якщо є.

new_name – нова назва функції.

Приклад перейменування функції sqrt для типу integer на square_root:

ALTER FUNCTION sqrt(integer) RENAME TO square_root;

Видаляє визначення існуючої функції команда DROP FUNCTION. Необхідно вказати типи аргументів функції, оскільки може існувати кілька різних функцій з однаковими назвами та різними списками аргументів.

Синтаксис:

DROP FUNCTION [IF EXISTS] *name* [([[*argmode*] [*argname*] *argtype* [, ...]])] [, ...]
[CASCADE | RESTRICT]

Параметри:

IF EXISTS – не видавати помилку, якщо функція не існує. У цьому випадку видається повідомлення.

CASCADE – автоматично видаляти об’єкти, які залежать від функції (наприклад, оператори чи тригери), і, у свою чергу, усі об’єкти, які залежать від цих об’єктів.

RESTRICT – забороняє видаляти функцію, якщо від неї залежать будь-які об’єкти. Це значення за замовчуванням.

Приклад команди, яка видаляє функцію квадратного кореня:

DROP FUNCTION sqrt(integer);

Видалити кілька функцій однією командою:

DROP FUNCTION sqrt(integer), sqrt(bigint);

Тригери

Тригер створюється командою CREATE TRIGGER.

Синтаксис

CREATE [OR REPLACE] TRIGGER ім’я { BEFORE | AFTER | INSTEAD OF} { подія [OR ...] }
ON ім’я_таблиці
[FOR [EACH] { ROW | STATEMENT }]
[WHEN (умова)]
EXECUTE { FUNCTION | PROCEDURE } ім’я_функції (аргументи)

де подія може бути:

INSERT
UPDATE [OF ім’я_стовпця [, ...]]
DELETE
TRUNCATE

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 оператора, що викликав тригер.

— Оператори, виконання яких викликає 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.

Тригерні функції

В даний час неможливо написати функцію тригера на чистому SQL. Використовуються функції написані на процедурній мові PL/pgSQL.

Створюється задана користувачем функція, оголошена як функція без аргументів і повертає тип TRIGGER, яка буде викликатись при спрацьовуванні тригера.

У синтаксисі CREATE TRIGGER ключові слова FUNCTION і PROCEDURE рівнозначні, але тригерна функція, що вказується, повинна в будь-якому випадку бути функцією, а не процедурою. Ключове слово PROCEDURE тут підтримується з історичних причин і вважається застарілим.

В якості аргументів може бути необов'язковий список аргументів через кому, які будуть передані функції при спрацьовуванні тригера. Як аргументи функції передаються рядкові константи. І хоча в цьому списку можна записати і прості імена або числові константи, вони також будуть перетворені на рядки. Порядок звернення до таких аргументів функції тригера може відрізнятися від звичайних аргументів, тому його слід уточнити в описі мови реалізації цієї функції.

Для видалення тригера використовується команда DROP TRIGGER.

Приклади

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;

Структура звіту до лабораторної роботи

Для кожного з запитів представити:

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

1. Виконати запити на створення процедур:
1.1) яка внесе значення в таблицю та перевірити результат її виконання;
1.2) яка використовує агрегатну функцію для певного розрахунку та виводить обчислюване значення на екран;
1.3) яка виводить в повідомленнях – NOTICE, WARNING, INFO дані на поточний час;
1.4) яка переривається помилкою та виводить підказку;
1.5) яка виводить дані з таблиці в циклі через повідомлення.
2. Виконати запити на створення користувальницьких функцій:
2.1) з вхідними параметрами, яка повертає вихідний параметр. Викликати функцію з використанням позиційного запису;
2.2) з вихідними параметрами (OUT). Викликати функцію з виведенням значень у табличному вигляді та у вигляді масиву;
2.3) яка поміняє місцями значення обраних атрибутів таблиці. Викликати функцію з використанням іменованого запису;
2.4) яка здійснює пошук в таблиці(-ях) значення за заданим шаблоном. Викликати функцію.
3. Виконати запити на створення тригерів та тригерних функцій з використанням контексних змінних(як мінімум NEW та OLD):
3.1) тригер на операцію модифікації даних INSERT;
3.2) тригер на операцію модифікації даних DELETE;
3.3) тригер на операцію модифікації даних UPDATE.

Звіт до лабораторної роботи 9 можна здати онлайн на сайті ДО edu.op.edu.ua до початку вашого заняття.