МІНІСТЕРСТВО ОСВІТИ І НАУКИ
ПРИКАРПАТСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВАСИЛЯ СТЕФАНИКА

PL/pgSQL – SQL ПРОЦЕДУРНА МОВА

НАВЧАЛЬНИЙ ПОСІБНИК

Івано-Франківськ

2015

УДК 681.3.07
Д95
ББК 32.973.26

Рекомендовано до друку Вченою радою факультету математики та інформатики Прикарпатського національного університету імені Василя Стефаника, протокол №3 від 08.12.2015 р.

Рецензенти:

Ткачук В.М. к.т.н., доцент кафедри інформаційних технологій Прикарпатського національного університету імені Василя Стефаника

Козич О.В. зав. лабораторіями ЦІТ, ст. викладач кафедри інформаційних технологій Прикарпатського національного університету імені Василя Стефаника

Дрінь Б.М.

PL/pgSQL: Навч.посібн. [Електронний ресурс] / Б.М. Дрінь – Івано-Франківськ, Прикарпатський національний університет, 2015 . – 60 с.

В даному посібнику дано загальну характеристику і описано стандарт мови PL/pgSQL.

Подано основні теоретичні викладки, які розкривають сутність мови, її функції та призначення. Основну увагу приділено таким можливостям мови як: структура функцій, їх побудова та дія; оператори мови.

Матеріал посібника успішно може бути використаний в навчальному процесі вищих навчальних закладів на спеціальностях які вивчають роботу з базами даних, а також вчителями спеціалізованих загальноосвітніх шкіл, коледжів, ліцеїв, гімназій тощо.

УДК 681.3.07
Д95
ББК 32.973.26

Зміст

1. PL/pgSQL - SQL процедурна мова

1.1. Огляд

PL/PgSQL це процедурна мова для СУБД PostgreSQL, або інакше процедурне розширення мови SQL. Процедурною мовою називається мова програмування, в якій бажаний результат досягається послідовністю тегів.

Метою проектування PL/PgSQL було створення процедурної мови, яка:

- використовується для створення функцій і тригерів;
- дозволяє добавити управляючі структури до мови SQL;
- дозволяє адаптувати програмний код, оскільки функції мови сумісні з усіма платформами, на яких працює СУБД PostgreSQL;
- успадкувала всі типи даних, функцій і операторів мови SQL, що розширює її можливості, а також підвищує гнучкість коду і швидкодію програм;
- дозволяє використовувати декілька команд SQL в програмному блоці PL/pgSQL за допомогою однієї операції, замість звичайної обробки кожної команди;
- може використовуватись для виконання складних обчислень;
- має відносно просту структуру, оскільки кожна логічна послідовність дій організовується у вигляді окремої функції;
- проста у використанні.

До переваг мови можна віднести:

- можливість згрупувати блок обчислень і послідовність запитів в середині сервера БД, що демонструє перевагу використання процедурної мови і простоту використання SQL при значній економії часу на взаємодію клієнт_сервер;
- непотрібність пересилання проміжних результатів на сервер;
- можливість запобіганню багаторазового аналізу одного запиту.

В результаті цього значно підвищується продуктивність у порівнянні з додатками, які не використовують збережувальних функцій.

Регістр символів у ключових словах, ідентифікаторах, іменах функцій тощо не враховується.

PL/pgSQL встановлюється за замовчуванням разом з PostgreSQL.

1.2. Підтримувані аргументи і типи даних

Функції, створені на PL/pgSQL можна використовувати в будь-якому місці, де можна використовувати вбудовані функції. Наприклад, можна створювати складні функції на виконання обчислень, а потім використовувати їх в операторах або в індексних виразах.

Функції можуть приймати в якості аргументів дані будь-якого типу або масиви даних, що підтримуються сервером, і повертати результат у вигляді даних будь-якого з цих типів. Вони можуть також прийняти або повернути будь-який допустимий складений тип (тип ROW – рядок таблиці) з вказаним іменем. Крім того, можна оголосити функцію PL/pgSQL як набір записів.

Функції PL/pgSQL можуть бути оголошені з довільною кількістю аргументів, використовуючи опцію VARIADIC.

PL / функції PGSQL також можуть бути оголошені вхідні та вихідні поліморфні типи: **anyelement**, **anyarray**, **anynonarray**, **anyenum** і **anyrange**– елемент, масив, таблиця, перелічувальний, діапазонний тип. Фактичні типи даних обробляються поліморфною функцією по різному від виклику до виклику.

PL/PGSQL функції також можуть повернути "множину" (або таблицю) будь-якого типу даних, які можуть бути повернуті в якості одного примірника. Така функція генерує RETURN NEXT для кожного потрібного елементу результуючого набору або за допомогою RETURN QUERY для виведення результату обчислення запиту.

Нарешті, функція PL/pgSQL може повернути **void**, якщо результат не потребує подальшої обробки.

1.3. Структура програмного коду PL/pgSQL

PL/pgSQL є блок-структурована мова. Блоком вважається повний текст

визначення функції, який має наступну структуру:

[<label1>]

[DECLARE

декларація (оголошення)змінних **]**

BEGIN

оператори (команди)

END [label1];

Кожна декларація позначається ключовим словом DECLARE і завершується ключовим словом END за яким слідує символ крапка з комою (;).

Всі змінні декларуються у блоці декларації програмного блоку шляхом вказання їх імені та типу.

Після декларації змінних слідує ключове слово BEGIN, що вказує на початок основного програмного блоку. За ним слідують команди, що входять до блоку, кожна із яких завершується символом крапка з комою. Кінець програмного блоку позначається ключовим словом END;.

Більш загальна структура програмного блоку PL/pgSQL може мати вигляд:

CREATE FUNCTION <ідентифікатор>[(<аргументи>)] RETURNS <тип> AS

$$

DECLARE

Декларація змінних [...]

BEGIN

команди: [...]

END;

$$

Тут символи **$$** виступають в ролі розділювачів між заголовком та блоком і можуть бути довільними символами, що подвоюються.

Мітки

Мітки використовується тільки тоді, коли потрібно визначити блок для використання в операторі переходу, або локалізувати імена змінних, оголошених в блоці. Якщо мітка задається після END, вона повинна відповідати імені початку блоку.

Ідентифікатори неявно перетворюються до нижнього регістра за винятком ідентифікаторів в подвійних лапках, аналогічно до звичайних команд SQL.

Коментарі

У програмі, для підвищення її читабельності, можна використовувати коментарі. В PL/pgSQL підтримуються два види коментарів, а сааме: однорядкові і блокові (багаторядкові) коментарі.

Синтаксис коментарів

Коментарі першого типу - однорядкові коментарі - починаються з двох дефісів (--) і діють до кінця рядка.

Приклад використання однорядкових коментарів:

--це буде інтерпретуватись як однорядковий коментар--.

– Коментарі другого типу – блокові коментарі. Блоковий коментар починається з послідовності символів /* та завершується послідовністю */. Вони можуть поширюватися на кілька рядків, при цьому весь текст між початковою і завершальною парою / * і * / вважається коментарем. Кожен новий рядок коментаря починається із символу ‘*’.

Приклад

/*

* Це є

* блок коментарів. */

Хоча блокові коментарі можуть містити вкладені однорядкові коментарі, вкладення блокових коментарів в інші блокові коментарі не допускається.

Команди і вирази

Програмні коди PL/pgSQL, як і в більшості мов програмування, складаються з команд і виразів. Команда виконує деяку дію в коді PL/pgSQL, наприклад, присвоює значення змінної або виконує запит. Послідовність команд в програмному блоці PL/pgSQL визначає порядок виконання дій у цьому блоці. Значна частина команд зазвичай розміщується в основній частині блоку, що знаходиться між ключовими словами BEGIN і END. Деякі команди також можуть бути присутніми в блоці декларацій (після ключового слова DECLARE), але вони тільки декларують і/або ініціюють змінні, які використовуються у программному блоці.

Програмний блок PL/pgSQL може містити необмежену кількість вкладених блоків, які читаються і інтерпретуються по тих же правилам, що і звичайні блоки. В свою чергу, вони можуть містити свої вкладені блоки.

Структура вкладених блоків не відрізняється від структури звичайних блоків, оскільки вони також починаються з ключового слова DECLARE, за яким слідує послідовність команд розміщених в операторних дужках BEGIN … END;.

Основний блок PL/pgSQL функції повинен повертати значення заданого типу, а всі вкладені блоки повинні бути завершені до досягнення ключового слова END.

Вкладені блоки можуть бути використані для логічного групування декількох операторів чи локалізації області дії змінних до невеликої групи операторів і, тим самим, спрощують структуру коду у громіздких функціях PL/pgSQL.

Змінні, вказані в таких блоках маскують (блокують) будь-які однойменні змінні зовнішніх блоків до кінця дії вкладеного блоку; однак можна отримати доступ до зовнішніх змінних, якщо задавати їх імена з допомогою ключового слова **label** зовнішнього блоку.

Наприклад:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
-- AS – розділювач між заголовком і блоком
```

```
<<outerblock>>   -- мітка – початок зовнішнього блоку
DECLARE
   quantity integer := 30;
BEGIN
   RAISE NOTICE  'Quantity here is %', quantity;
-- RAISE NOTICE – виведення повідомлення на екран
-- Prints 30
   quantity := 50;  -- переприсвоєння значення
   --
   -- Create a subblock  {створюємо вкладений блок}
   --
   DECLARE
      quantity integer := 80;
   BEGIN
      RAISE NOTICE 'Quantity here is %', quantity;
 -- Prints 80
      RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
   END;
   RAISE NOTICE 'Quantity here is %', quantity;
 -- Prints 50
   RETURN quantity; --Повернення
END;
$$ LANGUAGE plpgsql;
```

Пояснення. Тут RAISE NOTICE вказує на вивід повідомлення на екран. Символ % вказує, що замість нього буде підставлятись значення quantity=30. Оператор quantity := 50; задає переприсвоєння даній змінній значення рівне 50. Після Create a subblock створюється вкладений блок у якому змінній присвоюється значення =80 (quantity := 80) і виводиться дане значення. Однак

Важливо не плутати використання “початок/кінець” для групування операторів в PL/PgSQL з подібними назвами команд SQL для управління

транзакціями.

1.4. Декларування (оголошення) змінних

Всі змінні, використовувані в блоці повинні бути попередньо визначені в розділі декларації блоку за допомогою ключового словом DECLARE. Виняток становлять лише змінні циклу FOR, яким автоматично присвоюється цілочисельний тип, а також змінні циклу курсору, які автоматично оголошується як змінні типу record (запис). В PL/pgSQL змінні можуть мати будь-який тип даних, що і в SQL. Змінні, задекларовані в блоці, доступні у всіх його вкладених блоках, але не навпаки.

Синтаксис декларації змінної:

<ім'я змінної> [CONSTANT] **<тип даних>** [COLLATE <**collation_name**>] [NOT NULL] [{ DEFAULT | := } **<значення>**];

Фраза DEFAULT, якщо вона присутня, задає початкове значення яке присвоюється змінній при входженні у блок. Якщо дана фраза відсутня, то змінній, за замовчуванням, присвоюється значенням NULL.

Фраза CONSTANT запобігає зміні значення змінної після ініціалізації, і воно буде залишатися постійним протягом дії всього блоку. Опція COLLATE визначає параметри сортування, які будуть використані для даної змінної.

Якщо задано опцію NOT NULL, то присвоєння змінній нульовою значення призводить до помилки під час виконання.

Значення за замовчуванням обчислюється і присвоюється змінній при кожному входженні у блок (не тільки при першому виклику функції). Так, наприклад, якщо змінна типу **timestamp** має функцію **now(),** у якості значення за замовчуванням, то це означає, що змінна завжди буде містити час поточного виклику функції, а не час, коли функція була попередньо скомпільована (створена).

Приклади оголошення змінних.

Приклад №1

user_id integer;

quantity numeric(5);

url varchar;

book_title varchar(10);

myrow tablename%ROWTYPE; -- myrow – рядок таблиці tablename

myfield tablename.columnname%TYPE;

arow RECORD;

Приклад №2

quantity integer default 32;

url varchar := 'http://mysite.com';   --довільний текст

user_id CONSTANT integer := 10;

letter char default ‘a’;

1.5. Оголошення параметрів функції

Функції представляють собою програмний блок, який задається командами SQL для визначення PL/pgSQL функцій в базах даних PostgreSQL.

Перед виконанням функції PL/pgSQL вона повинна отримати конкретні значення у відповідності із кількістю і типом аргументів, які в ній описані. Аргументи роблять функції PL/pgSQL більш універсальними і значно розширюють область їх можливого застосування. задаються після імені функції в круглих дужках і розділяється символом кома.

Кількість і типи аргументів повинні відповідати початковому визначенню функції.

Аргументи, отримані функцією, почергово присвоюються ідентифікаторам, що складаються із знаку долара ($) і порядкового номера. Першому аргументу відповідає ідентифікатор $1, тоді $2 і т.д. Якщо функція має велику кількість аргументів, в позначеннях виду $<номер>, то краще використати псевдоніми, які більш наглядно відображають суть програмного коду.

При бажанні, можуть бути оголошені псевдоніми для $N параметрів. Псевдонім, або цифровий ідентифікатор (задається $N), може бути використаний для звернення до значення параметру.

Є два способи створення псевдоніму.

Перший спосіб, це задати імені параметру в команді CREATE FUNCTION. Наприклад:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
   RETURN subtotal * 0.06;  -- subtotal типу real – псевдонім
END;
$$ LANGUAGE plpgsql;
```

Другий спосіб – явне оголошення псевдоніму за допомогою ключового слова ALIAS

Такий псевдонім являє собою альтернативний ідентифікатор для посилання на аргумент. Перед використанням всі псевдоніми (як і звичайні змінні) повинні бути оголошені в секції оголошень блоку.

Синтаксис декларації:

<ідентифікатор> ALIAS FOR $n;

Цей же приклад, при явному оголошенні псевдоніму, при використанні даного синтаксису виглядає наступним чином:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
   subtotal ALIAS FOR $1;
BEGIN
   RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Ось ще кілька прикладів:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
   v_string ALIAS FOR $1;
   index ALIAS FOR $2;
```

```
BEGIN
   /* виконання деяких обчислень, що використовують
         v_string і index    */
END;
$$ LANGUAGE plpgsql;
```

Наприклад:

```
CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS
$$    -- деяка таблиця
BEGIN
   RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Коли PL/PgSQL функція оголошена з вихідними параметрами, вони теж іменуються $N і теж можуть мати необов'язкові псевдоніми, як і звичайні вхідні параметри. Вихідний параметр є змінна, яка ініціалізується (стартує) як NULL і якій присвоюється певне значення під час виконання функції. Як результат, повертається останнє присвоєне значення. Вихідні параметри є найбільш корисними при поверненні кілька значень.

Наприклад:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
   tax := subtotal * 0.06; --out (повернення) замість return
END;
$$ LANGUAGE plpgsql;
```

Приклад використання декількох вихідних параметрів:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int)
AS $$
BEGIN
   sum := x + y;
```

```
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

Наприклад: Інший спосіб оголошення PL / PgSQL функції – за допомогою RETURNS TABLE.

Нехай дано таблиця sales із полями: itemno – тип integer, quantity –тип integer, price –тип real тощо. Задати функцію на створення (повернення) віртуальної таблиці

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT quantity, quantity * price
            FROM sales
       WHERE itemno = p_itemno; -- порівняння якихось значень
END;
$$ LANGUAGE plpgsql;
```

Для PL / PgSQL функції, яка повертає поліморфний тип (**anyelement**, **anyarray**, **anynonarray**, **anyenum** або **anyrange**), створюється спеціальний параметр $0. Його тип даних відповідає типу, що повертає функція, і який встановлюється на основі фактичних типів вхідних параметрів. Це дозволяє функції отримати доступ до типів даних, що повертаються. Для $0 можна задати псевдонім, оскільки він ініціалізується в NULL і його можна змінювати в середині функції.

Приклад роботи функції з будь-яким типом даних, який підтримується оператором конкатенації (+):

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)      -- … три параметри
RETURNS anyelement AS $$
DECLARE
```

```
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Той же ефект можна отримати, оголосивши один або декілька вихідних параметрів як поліморфний тип. В цьому випадку спеціальна змінна $0 не створюється, оскільки з цією метою використовуються вихідні параметри.

Наприклад:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement, OUT sum anyelement) AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

1.6. Псевдоніми

Основне практичне використання – призначити інші імена для змінних із заданими іменами. При цьому можна оголосити псевдоніми для будь-яких змінних, а не тільки параметрів функцій.

Синтаксис:

<newname> ALIAS FOR <**oldname**;> -- задання двох ідентичних імен для однієї -- змінної

Приклад:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

Оскільки псевдонім створює два різних імена для одного і того ж об'єкту, необмежене його використання, по відношенню до одного об’єкту,

може призвести до неоднозначності. Краще використовувати його тільки з метою одноразового переозначення початкового імені.

1.7. Наслідування (копіювання) типів даних

Змінні PL/pgSQL можуть відноситися до будь-якого із стандартних типів даних SQL (наприклад, integer, char тощо). Для цього використовується синтаксис:

<Variable>%TYPE

Тут %TYPE задає тип змінної.

Використовуючи %TYPE не обов'язково знати тип даних структури на яку здійснюється посилання і, найголовніше, якщо в майбутньому тип даних зміниться (наприклад з цілого на дійсний), то при цьому не потрібно змінювати визначення функції.

Використання даного типу особливо важливе при роботі з поліморфними функціями, оскільки, типи даних, необхідні для внутрішніх змінних, можуть змінюватись при кожному виклику. Відповідні змінні можуть бути створені із використанням даного типу до аргументів і значень функцій, що повертаються.

Крім цього у мові можна декларувати змінні, які будуть, наприклад, використовувати типи існуючої бази даних.

Так, наприклад, для оголошення змінної User1 з таким же типом, що і стовпець user_id в таблиці users потрібно написати:

User1 users.user_id%TYPE;

1.8. Тип ROW

Змінна складеного типу називається рядковою змінною ROW, або змінною рядкового типу (rowtype).

Синтаксис:

<ім'я змінної> <ім'я деякої таблиці>%ROWTYPE;

Значенням такої змінної може бути цілий рядок, отриманий в результаті виконання запиту SELECT або виконання оператора FOR за умови, що набір стовпців цього запиту відповідає заданому типу змінної.

Доступ до окремих значень полів даного типу здійснюється стандартно шляхом вказання імені таблиці і, через крапку, імені поля, наприклад **rowvar.field**.

Змінна типу **ROW** може бути оголошена з таким же типом, що і рядки існуючої таблиці чи представлення за допомогою синтаксису **table_name%ROWTYPE.**

Параметри функції можуть також бути складеного типу (рядками таблиці). В цьому випадку відповідний ідентифікатор $N буде змінною типу **ROW** і, поля таблиці, можуть бути вибрані, наприклад, як **$1.user_id**.

Поля рядкового типу унаслідують розмір і точність типів даних стовпців вихідної таблиці (наприклад таких як *char(n)*, де *n*-точність).

Приклад використання складених типів для таблиць table1 і table2:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
  t2_row table2%ROWTYPE; --оголошуємо змінну t2_row як таблицю
BEGIN
  SELECT * INTO t2_row FROM table2 WHERE ... ;
  RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;
```

1.9. Тип RECORD

Синтаксис:

<name> RECORD;

Змінні типу RECORD (запис) подібні на змінні типу ROW, але вони не мають наперед заданої структури. (A=record end)

Вони отримують структуру рядка, яка їм присвоюється за допомогою оператора SELECT або FOR.

Структура змінної типу запис може мінятися кожного разу при черговому присвоєнні значення. Наслідком цього є те, що поки змінній типу

RECORD не присвоєно значення, вона не має структури, і будь-яка спроба отримати доступ до окремого поля викличе помилку на час виконання.

Слід звернути увагу, що запис не є реальним типом даних, але тільки заповнювачем (тобто, які були поля, ті і заповнюються певними даними). Крім цього потрібно враховувати, що PL/pgSQL функція, яка має тип RECORD для значення, що повертається, це не теж саме, що звичайна змінна типу RECORD, хоча така функція може використовувати змінну даного типу в якості збереження свого результату. В обох випадках структура рядка невідома, на час створення функції, але для функції, що повертає RECORD фактична структура визначається під час аналізу (виконанн) запиту, який посилається на неї.

1.10. Основні оператори мови PL/PgSQL

1.10.1. Оператор присвоєння

Синтаксис:

<variable> := *<expression;>*

Виразом може бути константа, змінна, допустимий вираз чи значення рядка таблиці. Тип виразу повинен співпадати із типом змінної.

Змінна може бути простою змінною, полем змінної ROW або RECORD, чи елементом масиву.

Якщо тип даних результату виразу не збігається з типом даних змінної, або змінна має певний розмір/точність, то результуюче значення буде неявно перетворено інтерпретатором PL/PgSQL у тип змінної (за правилами перетворення типів), використовуючи функцію виводу типу даних результату і функцію приведення типу даних змінної.

Приклади:

```
Declare
Tax real;
D1 date;
My_record record;
```

```
begin
tax := subtotal * 0.06;
my_record.user_id := 20; -- поле user_id у записі my_record
d1 := ‘2015-01-01’;  -- неявно перетворення до типу date.
End;
```

1.10.2. Виконання команди без результату (яка не повертає результат)

В PL/pgSQL функції можна виконати будь-яку команду SQL, яка не повертає рядків таблиці, просто написавши дану команду, наприклад INSERT без фрази RETURNING (повернення результату у вигляді поля чи полів). Їм’я довільної PL/pgSQL змінної у тексті команди розглядається як параметр, після чого поточне значення змінної підставляється в якості значення параметру під час виконання.

Таким чином, при виконанні команди SQL функція PL/PgSQL може зберігати проміжні результати і повторно використовувати принцип дії даної команди. кешувати

Це буває корисним, коли потрібно обчислити значення виразу, або запиту **SELECT**, але в подальшому не використовувати результат їх дії (наприклад, при виклику функції, у якої є побічні ефекти, в результаті чого немає правильного результату). Для цього в PL/PgSQL, використовують оператор PERFORM.

Синтаксис:

PERFORM *query*;

Цей оператор виконує запит і ’відкидає’ результат. Запити задаються аналогічно, як і в операторі SELECT мови SQL, але ключове слово SELECT замінюється на PERFORM.

Для запитів WITH (розширена форма запиту SELECT), після PERFORM потрібно помістити запит в додаткові дужки (в цьому випадку, запит може повернути тільки один рядок). У PL/pgSQL змінні будуть підставлені в запит так само, як і в команду, яка не повертає результат. Крім того, спеціальна змінна

FOUND буде приймати значення істина, якщо запит повертає, по крайней мірі, один рядок, або хибність, якщо не повертає жодного рядка.

1.10.3. Виконання запиту з однорядковим результатом

Результат SQL команди, яка повертає один рядок (можливо з декількох стовпців) може бути присвоєний змінній типу RECORD, змінній рядкового типу (ROW), або списку скалярних змінних. Це робиться шляхом задання команди мови SQL із добавлення у ній фрази INTO.

Синтаксис:

SELECT *expressions* INTO [STRICT] ***target*** FROM …;

INSERT...RETURNING *expressions* INTO [STRICT]***target***;

UPDATE ... RETURNING *expressions* INTO [STRICT] ***target***;

DELETE ... RETURNING *expressions* INTO [STRICT] ***target***;

--наприклад змінна ***target*** після останньої команди DELETE може містити номери останніх видалених рядків

Якщо опцію STRICT не вказано у фразі INTO, то *target* буде містити номер першого рядка результату (що повертається запитом), або NULL, якщо запит не повернув жодного рядка. При цьому «перший вибраний рядок« не буде чітко визначений, якщо не задана фраз ORDER BY. Всі рядки результату після першого рядка відкидаються. Для перевірки, чи був повернутий хоча б один запис, можна використати спеціальну змінну FOUND:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
  RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

Тут SELECT вибирає дані із таблиці і поміщає перший із знайдених рядків у змінну *myrec*. Якщо не було знайдено жодного значення, то виконується дія після THEN.

Якщо результат запиту присвоюється змінній рядкового типу або списку змінних, то вони повинні в точності відповідати за кількістю і типами

даних у стовпцях результату, інакше відбудеться помилка під час виконання. Якщо використовується змінна типу запис, то виконана дія автоматично приводить результату запиту до рядкового типу.

Якщо опція STRICT задана, то запит повинен повертати тільки один рядок або виникне помилка на час виконання із повідомленням: NO_DATA_FOUND (немає рядків) або TOO_MANY_ROWS (більше одного рядка). Можна також використати секцію виключень у блоці для обробки помилок, щоб її ’перехопити’, наприклад:

```
BEGIN
   SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         RAISE EXCEPTION 'employee % not found', myname;
-- співробітник не знайдений
      WHEN TOO_MANY_ROWS THEN
         RAISE EXCEPTION 'employee % not unique', myname;
-- повертає декілька співробітників за заданим критерієм
END;
```

1.10.4. Виконання динамічних команд

Часто потрібно динамічно генерувати (формувати) команди в середині PL/PgSQL функцій, тобто такі команди, у яких при кожному виконанні можуть використовуватись різні таблиці або типи даних.

Динамічні команди це ті, які наперед не задані. Вони складаються із набору рядків і виконуються як команда.

Для виконання динамічних команд, використовують оператор EXECUTE.

Синтаксис:

EXECUTE *command-string* [INTO [STRICT] *target*] [USING *expression* [, ...]];

Тут *command-string* це вираз, що формує рядок типу text із текстом команди, яку

потрібно виконати. USING – підстановка.

Необов'язкова змінна *target*, це та змінна куди буде присвоюватися результат дії команди. Вона може бути змінною типу запис (Record), рядковою змінною або списком скалярних змінних розділених через кому. Необов'язкові вирази у фразі USING формують значення, які будуть вставлені в команду.

У сформованому тексті команди заміна імен змінних PL/pgSQL на їх значення проводитися не буде. Всі необхідні значення змінних повинні бути вставлені в командний рядок при її побудові , або потрібно використовувати відповідні параметри,

Також, немає ніякого плану кешування для команд, виконуваних за допомогою EXECUTE. Замість цього, план створюється кожен раз при виконанні. Таким чином, рядок команди може динамічно створюватися усередині функції для виконання дій з різними таблицями і стовпцями.

Фраза INTO вказує, куди повинні бути присвоєні результати SQL команди, що повертає рядки. Якщо використовується змінна рядкового типу або список змінних, то вони повинні в точності відповідати структурі результату запиту (коли використовується змінна типу запис, вона автоматично приводиться до рядкового типу результату запиту). Якщо повертається декілька рядків, то тільки перший буде прсвоєний змінній (-им) в INTO. Якщо не повертається ні один рядок, то присвоюється значення NULL. За відсутності фрази INTO результати запиту відкидаються .

З опцією STRICT запит повинен повернути рівно один рядок, інакше видається повідомлення про помилку.

Командний рядок може використовувати значення параметрів $1, $2, і т.д.

Приклад:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND
inserted <= $2' INTO c
USING checked_user, checked_date;
```

Тут важливо те, що символи параметрів можна використовувати тільки для

значень даних. Якщо потрібно використовувати динамічно певні імена таблиць або стовпців, необхідно вставити їх в командному рядку як текст. Наприклад, якщо попередній запит необхідно зробити на динамічно вибраній таблиці (динамічно задати ім'я таблиці), то потрібно сформувати текст команди наступним чином:

```
  EXECUTE 'SELECT count(*)
 FROM '|| tabname::regclass
   || ' WHERE inserted_by = $1 AND inserted <= $2'
  INTO c USING checked_user, checked_date;
```

Тут INTO c – помістити результат змінній c

Ще одне обмеження на символи параметрів є те, що вони працюють тільки в SELECT, INSERT, UPDATE і DELETE командах.

1.10.5. Отримання статусу виконання команди

Є декілька способів визначення статусу виконання команди. Перший спосіб полягає у використанні команди GET DIAGNOSTICS, яка має вигляд:

GET [CURRENT] DIAGNOSTICS *variable* = *item* [,...];

Ця команда дозволяє отримати значення індикаторів стану системи. Кожен елемент *item* є ключовим словом ідентифікації значення стану яке буде присвоєно вказаній змінній. Змінна повинна бути відповідного типу даних діагностичної змінної. В PL/PgSQL функціях доступні такі індикатори: ROWCOUNT– кількість рядків, оброблених останньою командою SQL, і RESULTOID – OID останнього рядка, вставлений останньою виконаною командою SQL. При цьому RESULTOID корисний тільки після вставки (через INSERT) записів у таблицю, яка містить ім'я поля (OID – ідентифікатор об'єкту) obgect identifier.

Приклад:

GET DIAGNOSTICS integer_var = ROW_COUNT;

Другий спосіб визначення статусу виконання команди полягає в перевірці значення спеціальної змінної FOUND, що має тип boolean. При виклику

PL/PgSQL функції, змінна FOUND ініціюється в хибність (false). Далі, значення змінної змінюється наступними операторами:

• SELECT INTO присвоює змінній FOUND істина, якщо рядок вибраний, або хибність, якщо рядок (рядки) не вибрані.

• PERFORM, UPDATE, INSERT, DELETE, FETCH, MOVE також присвоюють змінній FOUND істина (true), якщо рядки оброблено (потім вони відкидаються), або хибність, якщо рядки не вибрані.

• MOVE FOR, як і FOREACH, присвоює FOUND істина, якщо була проведена хоча б одна ітерація циклу, в противному випадку в хибність. При цьому значення FOUND буде установлено тільки після виходу із циклу. Поки цикл виконується, оператор циклуа не міняє значень змінної. Однак другі оператори в середині циклу можуть змінювати значення FOUND.

• RETURN QUERY і RETURN QUERY EXECUTE присвоюють FOUND істина, ящо запит поверне хоча б один рядок, або хибність, якщо рядки не вибрані.

Другі оператори PL/pgSQL не змінюють значення FOUND.

FOUND є локальною змінною в кожній PL/pgSQL функції і будь-яка її зиіна, впливає тільки на поточну функцію.

1.10.6. Ігнорування будь-яких дій

Іноді буває корисно виконати команду, яка ігнорує недопустимі дії (наприклад ділення на ноль), чи певні системні помилки пропускаючи їх.

Наприклад можна вказати, що одній із гілок розгалуження (у операторі if/then/else), як результат, свідомо присвоюється значення NULL.

У наступному прикладі вказано фрагмкнт коду ігнорування помилки ділення на ноль:

```
BEGIN
   y := x/0;
EXCEPTION
 -- виключення
   WHEN division_by_zero THEN
```

```
    NULL;
-- division – спеціальна константа
 -- ignore then error
END;
```

2. Керуючі структури

2.1. Повернення результату

Керуючі структури є однією із найбільш корисних і важливих частин PL/pgSQL з допомогою яких можна дуже гнучко і ефективно маніпулювати даними.

Є два оператори, які дозволяють повертати дані з функції: RETURN і RETURN NEXT.

2.2. Оператор RETURN

Синтаксис

RETURN *expression*;

Даний оператор перериває виконання функції і повертає значення виразу (*expression*) у програму, яка її викликала. Ця структура використовується для PL/PgSQL функцій, які не повертають набір рядків.

У функції, що повертає скалярний тип, результуючий вираз автоматично приводиться до типу, що повертається. Однак, щоб повернути результат складеного типу *рядок*, вираз що повертається повинен в точності містити необхідний набір стовпців. При цьому може виникнути потреба у явному приведенні типів у відповідність.

Для функції з вихідними параметрами можна використати RETURN без виразу. При цьому будуть повернуті поточні значення вихідних параметрів .

Для функції, що повертає void (функція без типу) RETURN можна використовувати в будь-якому місці, але без виразу після RETURN.

Значення, що повертає функція, не може залишитися не визначеним. Якщо досягнуто кінець блоку верхнього рівня, а оператор RETURN не був вказаний, то відбувається помилка часу виконання. Це не стосується функцій з вихідними параметрами і функцій, які повертають void. Для них, оператор RETURN виконується автоматично по закінченні блоку верхнього рівня.

Деякі приклади:

```
-- функція, що повертає скалярний тип даних
RETURN 1 + 2;
RETURN scalar_var;
-- повертає те, що було присвоєно зммінній scalar_var

-- функція, що повертає складений  тип даних
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);
-- потрібно виконати приведення типів
```

2.3. Команди: RETURN NEXT і RETURN QUERY

Синтаксис:

RETURN NEXT *expression*;

RETURN QUERY *query*;

RETURN QUERY EXECUTE *command-string* [USING *expression* [, ...]];

Тут RETURN NEXT накопичує результат (рядок за рядком)

Коли PL/PgSQL функція оголошена як результуючий тип SETOF Sometype, то окремі елементи, що повертають значення, формуються командами RETURN NEXT або RETURN QUERY, а потім кінцева команда RETURN без аргументів, яка завершує виконання функції.

RETURN NEXT використовується як із скалярним, так і складеним типом даних. Для складеного типу результат функції повертається у вигляді таблиці.

При цьому RETURN QUERY добавляє результат виконання запиту до результату функції.

RETURN NEXT і RETURN QUERY можна вільно поєднувати в тілі функції, а їх результати будуть об’єднані. Крім цього дані команди не виконують повернення результату виконання функції. Вони просто добавляють рядки у результуючу множину значень, після чого виконання продовжується із наступного оператора функції. Успішне виконання RETURN NEXT і RETURN QUERY формує безліч рядків результату. Для виходу із функції використовується кінцевий RETURN, обов'язково без аргументів (або можна просто дочекатися закінчення виконання функції).

Для функції з вихідними параметрами можна просто використати RETURN NEXT без аргументів. При кожному виконанні RETURN NEXT, поточні значення вихідних параметрів зберігаються для подальшого повернення в якості рядка результату. При цьому, якщо функція з вихідними параметрами повинна повертати безліч значень, то при оголошенні потрібно вказувати RETURNS SETOF. Якщо вихідних параметрів декілька, то використовується RETURNS SETOF record, а якщо тільки один з типом sometype (будь який тип) то треба вказати RETURNS SETOF sometype.

Приклад таблиці і функції, що використовує RETURN NEXT:

```
-- створюється таблиця з полями…
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
-- добавляється два рядки
INSERT INTO foo VALUES (1, 2, 'three'); --1-й рядок
INSERT INTO foo VALUES (4, 5, 'six');  --2-й рядок
-- декларується наступна функція
CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
    DECLARE
        r  foo%rowtype;
```

```
        BEGIN
        FOR  r   IN
             SELECT *  FROM foo WHERE fooid > 0
        LOOP
             -- тут можлива обробка даних
        RETURN NEXT  r;
         /* добавляє поточний рядок запиту до
        результату, що повертається */
        END LOOP;
        RETURN;
    END $BODY$
    LANGUAGE plpgsql;
    -- Виклик даної функції здійснюється командою:
SELECT * FROM get_all_foo();
```

Приклад функції з RETURN QUERY:

```
CREATE FUNCTION get_available_flightid(date)
RETURNS SETOF integer AS
$BODY$
BEGIN
   RETURN QUERY SELECT flightid
                FROM flight
               WHERE flightdate >= $1
                AND flightdate < ($1 + 1);
   /* у випадку відсутності результату  дії команди   RETURN QUERY, додаткова
перевірка викличе повідомлення про помилку */
    IF NOT FOUND THEN
      RAISE EXCEPTION 'No flight at %.', $1;
   END IF;
   RETURN;
```

```
 END
$BODY$
LANGUAGE plpgsql;
-- Приклад виклику даної ф-ції:
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

2.4. Оператори умови (розгалуження)

У мові PL/PgSQL є два оператори IF і CASE які дозволяють виконати альтернативні дії в залежності від виконання, чи не виконання, певної умови чи умов.

Оператор IF…THEN…

Він може бути заданий у таких трьох форматах:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSIF ... THEN ... ELSE...

-- вкладені IF

Синтаксис оператора IF…THEN

IF *boolean-expression* THEN

statements

END IF;

IF-THEN - найпростіша форма IF. Команди між THEN і END будуть виконані, якщо умова істинна, інакше вони будуть пропущені.

Приклад:

```
IF v_user_id <> 0 THEN
   UPDATE users SET email = v_email
WHERE user_id = v_user_id;
END IF;
```

Оператор IF…THEN…ELSE

Синтаксис:

IF *boolean-expression* THEN

statements

ELSE

statements

END IF;

Даний оператор додає у структуру IF…THEN альтернативний набір операторів, які повинні виконуватися, якщо умова не виконується, або змінна приймає значення NULL.

Приклади:

```
IF parentid IS NULL OR parentid = ''
THEN
   RETURN fullname;
ELSE
   RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
   INSERT INTO users_count (count) VALUES (v_count);
   RETURN 't';
ELSE
   RETURN 'f';
END IF;
-- ‘t’ і ‘f’ – стандартні скорочення для true, false.
```

Оператор IF…THEN…ELSIF…

Синтаксис:

IF *boolean-expression* THEN

statements

[ELSIF *boolean-expression* THEN

statements

[ELSIF *boolean-expression* THEN

statements ...]]

[ELSE *statements*]

END IF;

Даний оператор використовується коли є більше ніж дві альтернативи і забезпечує зручний спосіб перевірки декілька варіантів по черзі. Умови аналізуються послідовно доти, поки певна умова не виконається. Якщо так, то виконується пов'язаний з нею оператор чи оператори, після чого управління передається наступному оператору після END IF. Якщо жодна з умов не виконується, то виконується оператор(и) після останнього ELSE якщо він (вони) присутній.

Наприклад:

```
IF number = 0 THEN
   result := 'zero';
ELSIF number > 0 THEN
   result := 'positive';
ELSIF number < 0 THEN
   result := 'negative';
ELSE
      result := 'NULL';
END IF;
```

Приклад вкладених операторів IF…THEN…ELSE

```
      IF demo_row.sex =  'm'
       THEN pretty_sex  := 'man';
      ELSE
           IF demo_row.sex = 'f'
           THEN pretty_sex  :=  'woman';
```

```
        END IF;
    END IF;
```

Оператор CASE

Він може бути заданий у таких двох форматах:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

Простий CASE

Синтаксис:

```
    CASE search-expression
        WHEN expression [, expression [ ... ]]
        THEN statements
      [ WHEN expression   [,   expression   […]]
      THEN statements ...]
      [ ELSE statements ]
    END CASE;
```

Проста форма CASE реалізує умовне виконання на основі порівняння на співпадання операндів search – expression (пошук-вираз) із операндами expression.

Операнд search–expression обчислюється один раз і послідовно порівнюється з кожним виразом expression у фразах WHEN. Якщо збіг знайдено, то виконуються відповідні statements (інструкції) і керування переходить до наступної після END CASE команди. (Усі наступні вирази WHEN не перевіряються.) Якщо збіг не було знайдено, то виконуються ELSE statements, а якщо ELSE відсутнє, то викликається виняток CASE_NOT_FOUND.

Приклад:

```
CASE x
   WHEN 1, 2 THEN
      msg := 'one or two';
   ELSE
      msg := 'other value than one or two';
END CASE;
```

Оператор CASE (з перебором умов)

Синтаксис:

```
CASE
   WHEN boolean-expression THEN
     statements
  [ WHEN boolean-expression THEN
     statements    ... ]
  [ ELSE      statements ]
END CASE;
```

Пошукова форма оператора CASE забезпечує виконання певної дії чи дій за умови істинності значення логічних виразів. Кожен вираз *boolean-expression* у фразі WHEN обчислюється по порядку, до тих пір, поки умова не буде виконана. Після цього виконуються відповідні statements (інструкції) і керування переходить до наступної після END CASE команди. (Усі наступні вирази WHEN не перевіряються). Якщо жодна з умов не буде істинною, то виконуються ELSE statements, а якщо ELSE відсутнє, то викликається виняток CASE_NOT_FOUND.

Дана форма CASE повністю еквівалентна If-Then-ELSIF.

Приклад:

```
CASE
   WHEN x BETWEEN 0 AND 10 THEN
      msg := 'value is between zero and ten';
```

```
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

2.5. Оператори циклу

У L/PgSQL функціях для організації циклів (багаторазового повторення команди, чи серії команд) можна використовувати наступні оператори циклу: LOOP, EXIT, CONTINUE, WHILE, FOR і FOREACH.

Оператор LOOP

Синтаксис:

[<<*label*>>]
LOOP
    statements
END LOOP [*label*];

Даний оператор дозволяє задати безумовний цикл, який повторюється до нескінченності, поки не буде припинений командою виходу EXIT, або через повернення RETURN. Для вкладених циклів можна використати *label* в операторах EXIT і CONTINUE, щоб вказати до якого циклу дані оператори відносяться.

Приклад:

```
LOOP
  -- різні дії
  IF a=0 THEN
    RETURN;
  END IF;
END LOOP:
```

Оператор EXIT

Синтаксис:

EXIT [*label*] [WHEN *boolean-expression*];

Даний оператор використовується для переривання роботи циклу. Якщо мітка *label* не вказана, то виконується самий внутрішній цикл, після чого виконується оператор. що слідує за *фразою* END LOOP. Якщо *label* вказана, то вона повинна відноситись до поточного або зовнішнього циклу, чи бути міткою блоку. При цьому в іменованому циклі (блоці) виконання припиняється, а управління передається на наступний оператор після відповідного END. При наявності слова WHEN, вихід з циклу відбувається тільки тоді, коли логічний вираз *boolean-expression* отримає значення істина. В іншому випадку управління передається оператору, що слідує за EXIT.

Примітки:

- EXIT можна використовувати у всіх типах циклів.
- Якщо EXIT використовується для виходу із блоку циклу, управління переходить до наступного, після останнього у блоці, оператора.

Розглянемо приклади.

Приклад №1

```
LOOP
   IF count > 0 THEN
      EXIT;
 -- вихід із циклу
   END IF;
END LOOP;
```

Приклад №2

```
LOOP
   EXIT WHEN count > 0;
END LOOP;
```

Приклад №3

```
<<ablock>>
BEGIN
    IF stocks > 1000 THEN
    EXIT ablock;
 -- вихід із блоку ablock
   END IF;
   -- обчислення не буде виконано, якщо stocks > 1000
END;
```

Оператор CONTINUE

Синтаксис:

CONTINUE [*label*] [WHEN *boolean-expression*];

Використовується для переходу на наступну ітерацію циклу.

Якщо label не задана, то починається наступна ітерація самого внутрішнього циклу. Тобто всі, що залишились в тілі циклу оператори пропускаються, і управління переходить до управляючого виразу циклу (якщо він є), для визначення чи потрібна ще одна ітерація циклу. *Якщо label* присутня, то вона вказує на мітку циклу, виконання якого буде продовжено.

При наявності фрази WHEN, наступна ітерація циклу почнеться тільки тоді, коли логічний вираз *boolean-expression* прийме значення істина. В противному випадку управління передається оператору, що слідує після CONTINUE..

CONTINUE може бути використаний з усіма типами циклів.

Приклад:

```
LOOP
   EXIT WHEN count > 100;
   CONTINUE WHEN count < 50;
   -- обчислення для count в діапазоні [50 .. 100]
END LOOP;
```

Оператор WHILE

Синтаксис:

[<<label >> **]**

WHILE *boolean-expression* LOOP

statements

END LOOP [*label*];

Даний оператор виконує серію команд до тих пір, поки логічний вираз *boolean-expression* буде приймати значення істина. При цьому вираз перевіряється безпосередньо перед кожним входженням у тіло циклу.

Приклади:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
-- виконання обчислень
END LOOP;

WHILE NOT done LOOP
   -- виконання обчислень
END LOOP;
```

Оператор FOR із цілочисельним виразом

Синтаксис:

[<<*label*>>]

FOR *name* IN [REVERSE] *expression .. expression*

[BY *expression*] LOOP

statements

END LOOP [*label*];

Даний оператор задає цикл, у якому ітерації виконуються по діапазону цілих чисел. Змінна *name* автоматично визначається цілого типу *integer* і її дія поширюється тільки в середині циклу. При цьому, якщо змінна з таким же іменем описана до початку циклу, то в середині циклу вона буде ігноруватись Два вирази,

що задають початкову і кінцеву межі діапазону циклу задаються один раз при вході в цикл. Якщо фраза BY (крок) не вказана, то крок ітерації рівний 1, в іншому випадку використовується значення, вказане у фразі BY, яке також обчислюється один раз при входженні у цикл. Якщо вказана фраза REVERSE, то після кожної ітерації величина кроку віднімається, а не добавляється до попереднього значення.

Приклади:

```
FOR i IN 1..10 LOOP
   -- у циклі змінна і буде приймати значення від  1 до 10
END LOOP;

FOR i IN REVERSE 10..1 LOOP
   -- у циклі змінна і буде приймати значення від  10 до 1 END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
   -- і змінюється від 10 до 2 із кроком (-2)
END LOOP;
```

Якщо початкове значення параметру циклу більше кінцевого (або менше, в разі REVERSE), тіло циклу не виконається жодного разу.

Цикл по результатах запиту

Використовуючи інший варіант циклу FOR, можна організувати цикл по результатах запиту і управління цими даними.

Синтаксис:

```
[ <<label>> ]
FOR target IN query LOOP
   statements
END LOOP [ label ];
```

Тут змінна target може бути змінною типу RECORD, ROW чи списком скалярних змінних розділених комами. При цьому змінній target послідовно присвоюється кожен рядок результату запиту, і тіло циклу виконується для

кожного рядка.

Пприклад:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
   mviews RECORD;
BEGIN
   FOR mviews IN          SELECT * FROM cs_materialized_views
        ORDER BY sort_key LOOP
 /*тут "mviews" містить (приймає) один-за-одним запис із cs_materialized_views */
       EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
       EXECUTE 'INSERT INTO '
          || quote_ident(mviews.mv_name) || ' '
          || mviews.mv_query;
   END LOOP;
   RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Якщо цикл завершується по команді EXIT, то останнє з присвоєних значень рядка доступне і після циклу.

Оператор FOREACH (цикл по елементах масиву)

Синтаксис:

Даний оператор використовується для роботи з масивами, тобто він замість перебору рядків, що повертаються SQL запитом (як у попередньому операторі) здійснює перебір елементів масиву.

Синтаксис:

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
   statements
```

END LOOP [*label*];

Якщо фраза SLICE не вказана, або якщо SLICE= 0, цикл виконується для всіх елементів масиву, отриманих із виразу *expression*. Змінній target послідовно присвоюється кожне значення елементу масиву і тіло циклу виконується для кожного елементу.

Приклад циклу по елементах масиву цілих чисел:

```
CREATE FUNCTION sum(int[]) RETURNS int% AS $$
DECLARE
  s int := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Перехід по елементах масиву здійснюється в тому порядку, в якому вони задані. Як правило змінна target, це звичайна проста змінна, хоча вона може бути і списком змінних, якщо елементи масиву мають складений тип *запис*. В такому випадку змінним присвоюються значення із послідовних стовпців складеного елементу масиву.

Перехоплення (обробка) помилок

За замовчуванням будь-яка помилка у PL/PgSQL функції перериває її виконання, чи транзакцію, яка відноситься до даної функції. Використовуючи в блоці секції фразу EXCEPTION (виключення) можна перехоплювати помилки і обробляти їх.

Синтаксис EXCEPTION є розширенням синтаксису для звичайного BEGIN

блоку і має наступний вигляд:

[<<*label*>>]

[DECLARE

declarations]

BEGIN

statements

EXCEPTION

WHEN *condition* [OR *condition* ...] THEN

handler_statements

[WHEN *condition* [OR *condition* ...] THEN

handler_statements ...]

END;

Якщо при виконанні команд блоку помилка не виникне, блок просто виконує всі команди і управління переходить до наступного оператора після END блоку. Якщо при виконанні *statements* відбудеться помилка, то подальша обробка команд припиняється, і управління переходить до списку виключень у секції після фрази EXCEPTION. У даному списку шукається перше виключення, умова якого відповідає помилці. Якщо виключення знайдено, то виконуються відповідні секції команд — *handler_statements* і управління переходить до наступного оператора після END.

Наприклад (умова виключення для “ділення на 0” - назва помилки або код помилки):

WHEN division_by_zero THEN ...

WHEN SQLSTATE '22012' THEN ...

Якщо при виконанні фрази *handler_statements* виникне нова помилка, то вона може бути перехоплена в даній секції EXCEPTION і, передача здійснюється за межі блоку, для перехоплення в секції зовнішнього блоку.

Якщо помилка так і не була перехоплена, то обробка функції припиняється.

При виконанні команд в секції EXCEPTION, локальні змінні PL/pgSQL функції зберігають ті значення, які були на момент виникнення помилки. При цьому будуть відмінені всі зміни в базі даних, які були виконані в блоці.

Розглянемо наступний фрагмент :

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
   UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
   x := x + 1;
   y := x / 0;
EXCEPTION
   WHEN division_by_zero THEN
      RAISE NOTICE 'caught division_by_zero';
      RETURN x;
END;
```

При присвоєнні значення змінній **y** відбудеться помилка division_by_zero. Вона буде перехоплена в секції EXCEPTION. Оператор RETURN поверне значення x збільшене на одиницю, але зміни зроблені командою UPDATE будуть скасовані. При цьому зміни, виконані командою INSERT, яка передує блоку, не будуть скасовані. В результаті цього база даних буде містити Tom Jones, а не Joe Jones .

3. Курсори

Замість того, щоб виконати цілий запит відразу, можна налаштувати курсор, який інкапсулює запит, а потім отримувати результат запиту по кілька рядків за раз. Однією з причин скористатися цією можливістю є уникнення переповнення пам'яті, коли результат містить велику кількість рядків.

Ще одним варіантом використання курсору є можливість повернення із функції посилання на курсор, що дозволяє отримувати рядки запиту і є ефективним засобом отримати великі набори рядків із функцій.

Оголошення змінних курсору

Доступ до курсору в PL/PGSQL здійснюється через курсорні змінні, які завжди мають спеціальний тип даних REFCURSOR. Тому одним із способів створення курсорної змінної – оголосити її як змінну даного типу. Інший спосіб полягає у використанні синтаксису декларації курсору, який має вигляд:

name [[NO] SCROLL] CURSOR [(*arguments*)] FOR *query*;

Приклади:

```
DECLARE
   curs1 refcursor;
   curs2 CURSOR FOR SELECT * FROM tenk1;
   curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

Тут всі три змінні мають тип даних REFCURSOR, але перша може бути використана з будь-яким запитом, оскільки вона вважається незв'язною, друга зв'язана з повністю сформованим запитом, а остання зв'язана з параметризованим запитом (key буде замінено цілочисельним значенням параметру при відкритті курсору).

Відкриття курсору

Перш ніж отримати рядки із курсору, його потрібно відкрити.

В PL/PgSQL є три види операторів OPEN для відкриття курсору, причому дві із них використовуються для незв'язних курсорних змінних, а третя – для зв'язних.

Примітка: зв'язні курсорні змінні можна використати з циклом FOR без явного відкриття курсору.

Оператор OPEN query

Синтаксис:

OPEN *unbound_cursorvar* [[NO] SCROLL] FOR *query*;

Курсорна змінна відкривається і отримує декларований запит для виконання. Курсор при цьому може і не бути відкритим, а курсорна змінна

повинна бути незв'язаною (тобто просто змінною типу REFCURSOR). Запит повинен бути командою SELECT або будь-якою іншою, яка повертає рядки. Запит обробляється так само, як і інші команди SQL в PL/PgSQL: імена PL/PGSQL змінних замінюються на значення, план запиту кешується для повторного використання. Підстановка значень PL/PGSQL змінних проводиться при відкритті курсору командою OPEN, подальші зміни значень змінних не впливають на роботу курсору.

Приклад:

```
OPEN curs1(mykey integer) FOR SELECT * FROM foo
WHERE key = mykey;
```

Оператор OPEN FOR EXECUTE

Синтаксис:

OPEN *unbound_cursorvar* [[NO] SCROLL] FOR EXECUTE *query_string* [USING *expression* [, ...]];

Запит задається рядковим виразом, так само, як в команді EXECUTE. Це дає гнучкість, оскільки план запиту кожен раз може змінюватися. Це також означає, що заміна змінних виконується не в командному рядку. Як і в EXECUTE, для вставки в динамічну команду значень параметрів використовується USING SCROLL і NO SCROLL, які мають той же зміст, що й для зв’язного курсору

Приклад:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM '
|| quote_ident(tabname)     || ' WHERE col1 = $1' USING keyvalue;
```

Відкриття зв’язного курсору

Здійснюється оператором OPEN *bound_cursorvar.*

Синтаксис:

OPEN *bound_cursorvar* [([*argument_name* :=] *argument_value* [, ...])];

Ця форма OPEN використовується для відкриття курсорною змінної , яка була пов'язана із запитом при оголошенні. Список фактичних значень аргументів повинен бути присутнім тільки в тому випадку, якщо курсор оголошувався з параметрами. Ці значення будуть підставлені в запит.

План запиту для зв’язного курсору завжди вважається кешувальним. В цьому випадку, немає еквівалента EXECUTE. Зверніть увагу, що SCROLL і NO SCROLL не можуть бути вказані в цій формі OPEN.

При передачі значень аргументів можна використовувати позиційну або іменну нотацію. У позиційній нотації, всі аргументи вказуються по порядку. В іменний нотації, ім'я кожного аргументу відділяється від виразу аргументу за допомогою : =.

Приклад:

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

Так як для зв’язного курсору виконується підстановка значень змінних, то існує два способи передачі значення в курсор. Перший – використати явні аргументи в OPEN; другий – неявно, посилаючись на PL/pgSQL змінні в запиті.

У зв'язаному курсорі можна посилатися тільки на ті змінні, які були оголошені до самого курсору. У будь-якому випадку, значення змінної для підстановки в запит визначатиметься на момент виконання OPEN.

Ось ще один спосіб отримати той же результат з curs3, як у прикладі вище

```
DECLARE
        key integer;
        curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
    BEGIN
        key := 42; OPEN
        curs4;
```

Використання курсорів

Після того, як курсор був відкритий, ним можна маніпулювати за допомогою операторів, описаних нижче, причому робота з ним не обов'язково повинна відбуватися в тій же функції, яка відкривала курсор. Із функції можна повернути значення з типом REFCURSOR, що дозволяє продовжити роботу з курсором.

Оператор FETCH

Синтаксис:

FETCH [*direction* { FROM | IN }] *cursor* INTO *target*;

Оператор FETCH витягує(повертає) наступний рядок з курсору у змінну target.

В якості target може бути рядкова змінна, змінна типу запис (*record*), чи розділений комами список простих змінних, як і в SELECT INTO. Якщо наступного рядка немає, змінній target присвоюється NULL. Як і в SELECT INTO перевірити, чи був отриманий запис можна за допомогою змінної FOUND.

Значення *direction*, яке вимагає переміщения назад, приведе до помилки, якщо курсор не був оголошений, або відкритий з опцією SCROLL.

Значення *direction* може бути будь-яким з варіантів: NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*, RELATIVE *count*, FORWARD, або BACKWARD.

Cursor – це змінна з типом refcursor, яка посилається на відкритий курсор.

Приклади:

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

Оператор MOVE

Синтаксис:

MOVE [*direction* { FROM | IN }] *cursor*;

Даний оператор переміщує курсор не повертаючи дані. MOVE працює так само, як команди FETCH, за винятком того, що він тільки переміщує курсор і не повертає при цьому рядок, на який перемістився. Як і в SELECT INTO перевірити, чи було здійснено переміщення можна за допомогою змінної FOUND.

Приклади:

MOVE curs1;

MOVE LAST FROM curs3;

MOVE RELATIVE -2 FROM curs4;

MOVE FORWARD 2 FROM curs4;

Оператори UPDATE/DELETE WHERE CURRENT OF

Синтаксис:

UPDATE *table* SET ... WHERE CURRENT OF *cursor*;

DELETE FROM *table* WHERE CURRENT OF *cursor*;

Коли курсор (вказує) на рядки таблиці, то рядок може бути оновлений або видалений за допомогою даного курсору.

Приклад:

UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;

Оператор CLOSE

Синтаксис:

CLOSE *cursor*;

CLOSE закриває зв’язаний з курсором портал. Це може бути корисним тоді, коли потрібно звільнити ресурси швидше ніж закінчиться транзакція, або

звільнити змінну курсору для повторного відкриття.

Приклад:

```
CLOSE curs1;
```

Повернення курсору із функції

Курсори можна повертати із PL/PgSQL функції. Це корисно тоді, коли потрібно повернути багато рядків і стовпців, особливо з дуже великими наборами результатів. Для цього у функції відкривається курсор і його ім'я повертається тому, хто його викликав. Користувач після цього може вибирати рядки з курсору і закрити його, або він буде автоматично закритий при завершенні транзакції.

Ім'я порталу, використовуване для курсору, може бути зазначено розробником або буде генеруватися автоматично. Щоб вказати ім'я порталу, потрібно присвоїти рядок змінній refcursor перед його відкриттям. Значення рядка у змінній refcursor буде використовуватися командою OPEN як ім'я порталу. Однак, якщо змінна refcursor має значення NULL, то оператор OPEN автоматично генерує ім'я, яке не конфліктує з будь-яким існуючим порталом і привласнює його змінній refcursor .

Примітка:

Зв’язна курсорна змінна перетворюється у рядкове значення, що є іменем самої змінної. Таким чином, ім'я порталу збігається з іменем змінної курсору, крім випадків, коли розробник перевизначив ім'я, присвоївши нове значення перед відкриттям курсору. Незв'язана змінна курсору ініціалізується в NULL і отримає автоматично згенероване унікальне ім'я, якщо не буде її перевизначення.

Наступний приклад показує один із способів передачі імені курсору, який буде виконаний

```
CREATE TABLE test   (col ext);
INSERT INTO test VALUES('123');
CREATE FUNCTION reffunc(refcursor)
RETURNS  refcursor AS   $$
  BEGIN
```

```
    OPEN $1  FOR SELECT col  FROM
    test; RETURN $1;
  END;
$$ LANGUAGE plpgsql;
BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

В наступному прикладі використовується автоматична генерація імен курсору:

```
CREATE FUNCTION reffunc2()   RETURNS  refcursor AS $$
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
 $$ LANGUAGE plpgsql;
   /* для використання курсорів,  потрібно почати транзакцію */
BEGIN;
SELECT reffunc2();

/* Результат виводу
      reffunc2
      <unnamed cursor 1>
      (1 row)
*/
FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

Наступний приклад показує один із способів повернення декількох курсорів із однієї функції:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor
AS
$$
BEGIN
    OPEN $1 FOR SELECT * FROM
    table_1; RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM
    table_2; RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;
/* для використання курсорів,  необхідно почати транзакцію */
BEGIN;
SELECT * FROM myfunc('a',   'b');
FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

Перебір результату курсору (обробка курсору в циклі)

Один із варіантів циклу FOR дозволяє перебирати рядки повернуті курсором:

Синтаксис:

[<<*label*>>]
FOR *recordvar* IN *bound_cursorvar* [([*argument_name* :=] *argument_value* [, ...])]
LOOP
  statements
END LOOP [*label*];

Змінна recordvar автоматично визначається як змінна типу RECORD і її

дія поширюється тільки в середині циклу. Кожен рядок, що повертається курсором послідовно присвоюється даній змінній і виконується тіло циклу для кожного рядка курсору.

Повідомлення про помилки

Для виклику повідомлень і генерації помилок використовується оператор RAISE.

Синтаксис:

RAISE [*level*] '*format*' [, *expression* [, ...]] [USING *option* = *expression* [, ...]];

RAISE [*level*] *condition_name*

[USING *option* = *expression* [, ...]];

RAISE [*level*] SQLSTATE '*sqlstate*'

[USING *option* = *expression* [, ...]];

RAISE [*level*] USING *option* = *expression* [, ...];

RAISE ;

Опція *level* задає рівень важливості помилок. Можливі рівні DEBUG, LOG, INFO, NOTICE, WARNING, і EXCEPTION. За замовчуванням використовується EXCEPTION, який викликає помилку. Решту значень *level* тільки генерують повідомлення з різними рівнями пріоритету.

Після *level* можна задати *format* – простий символьний рядок у якому можна задати повідомлення про помилку,

Усередині рядка формату символ % замінюється на текстове представлення наступного не обов’язкового значення аргументу.

У наступному прикладі символ % буде замінено значенням v_job_id.

RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;

4. Тригерні процедури

4.1. Тригери

Можна дати такі визначення тригерів:

Тригером називається дія, яка пов’язана з подією, що викликає зміни у вмістимому таблиці.

Тригер — це SQL команда чи команди, які автоматично виконуються як додаткова дія при внесенні певних змін у вказану таблицю.

Тригери використовуються для досягнення наступних цілей:

- перевірки коректності введених даних;
- перевірки виконання складних операцій обмеження цілісності, які важко або взагалі неможливо описати стандартними правилами обмеження цілісності встановленої для таблиць;
- генерації повідомлень, що нагадують про виконання деяких дій
- накопичення аудит-інформації за допомогою фіксації даних про здійснені зміни та користувачів, які їх здійснили
- підтримка реплікацій, тобто можливість організації одних і тих же даних на різних серверах.

При умові правильного використання тригери можуть стати потужним механізмом роботи з даними.

Основною перевагою тригерів є їхнє розміщення всередині бази даних та узгоджена активація при кожній зміні її даних. Хоча для тригерів характерні і ряд недоліків, зокрема таких, як:

- складність. Ускладнюється завдання проектування, реалізації та адміністрування баз даних;
- прихована функціональність. Перенесення окремих функції в середину БД приховує їх від користувача. Хоча це спрощує роботу користувача, але позбавляє його контролю за окремими процесами;
- вплив на продуктивність. Перед виконанням кожної команди зміни БД вимушена перевіряти умови необхідності запуску тригера для даної команди.

Тригери можуть бути означені для виконання ПЕРЕД (**before**) чи ПІСЛЯ (**after**) будь-якої операції INSERT, UPDATE, DELETE на кожен рядок таблиці або на цілу команду загалом.

Тригерна функція повинна бути описана перед створенням тригера. Дана функція задається без аргументів і повертає тип **trigger**.

Тригер створюється командою ***CREATE TRIGGER***. Одна й та ж тригерна функція може застосовуватись в різних тригерах.

Синтаксис:

CREATE TRIGGER name { BEFORE | AFTER } { event [OR ...] }

ON table [FOR [EACH] { ROW | STATEMENT }]

EXECUTE PROCEDURE funcname ([arguments])

Опис даної команди:

Name — унікальне ім’я нового тригера

BEFORE чи **AFTER** — задає порядок виконання функції

event — одна з **INSERT, UPDATE**, чи **DELETE** команд

table — ім’я таблиці для якої створюється тригер.

FOR EACH ROW або FOR EACH STATEMENT — задає відповідно рядковий чи командний типи тригера.

funcname — імя функції яка виконується у тригері

arguments — додаткові параметри що передають в тригерну функцію (необовязковий параментр).

PostgreSQL підтримує ***per-row*** тригери (на рядок) і ***per-statement*** тригери (на команду). Рядкові тригери виконуються для кожного рядка таблиці а командні тригери на всю команду не залежно від кількості оброблюваних даних.

Командні тригери завжди повертають NULL. Рядкові тригери повертають, як правило, рядок таблиці.

Рядкові тригери, що спрацьовують перед операціями зміни даних, завжди повертають той рядок таблиці, що передається їм (це NEW - рядок для INSERT і

UPDATE тригерів, OLD - рядок для DELETE тригерів).

Якщо означено більше одного тригера на одну таблицю і на ту ж саму дію, то тригери виконуються в алфавітному порядку своїх імен.

Для видалення тригера існує команда **DROP TRIGGER**

Синтаксис:

DROP TRIGGER [IF EXISTS] ***name*** **ON** ***table*** **[CASCADE | RESTRICT],**

де **name** — ім’я тригера, що знищується, а **table** — ім’я таблиці для якої він заданий.

4.2. Тригери на зміну даних

В PL/pgSQL можна створювати тригерні процедури. Тригерна процедура створюється командою CREATE FUNCTION, при цьому у функції не повинно бути аргументів, а значення, що повертається повинно бути типу trigger. Тут важливо, що функція створюється без аргументів, навіть якщо їй потрібно отримати аргументи, вказані в команді CREATE TRIGGER. Аргументи тригера передаються через спеціальний масив TGARGV, як буде показано нижче.

Коли PL/pgSQL функція спрацьовує як тригер, в блоці верхнього рівня автоматично створюються кілька спеціальних змінних:

– NEW

Тип даних RECORD. Змінна містить новий рядок бази даних для команд INSERT/UPDATE в тригерах рівня рядка. В тригерах рівня оператора і для команди DELETE цій змінній значення не присвоюються .

– OLD

Тип даних RECORD. Змінна містить старий рядок бази даних для команд UPDATE/DELETE в тригерах рівня рядок. В тригерах рівня оператора і для команди INSERT даній змінній значення не присвоюється.

– TG_NAME

Тип даних name. Змінна містить ім'я тригера який спрацював.

– TG_WHEN

Тип даних text. Рядок, що містить BEFORE, AFTER або INSTEAD OF, залежно від визначення тригера.

– TG_LEVEL

Тип даних text. Рядок, що містить ROW або STATEMENT, залежно від визначення тригера.

– TG_OP

Тип даних text. Рядок, що містить INSERT, UPDATE, DELETE або TRUNCATE, залежно від того, для якої операції спрацював тригер.

– TG_RELID

Тип даних OID таблиці, для якої спрацював тригер.

– TG_TABLE_NAME

Тип даних name. Ім'я табліці, для якої спрацював тригер.

– TG_TABLE_SCHEMA

Тип даних name. Ім'я схеми, що містіть таблицю, для якої спрацював тригер .

– TG_NARGS

Тип даних integer. Число аргументів на команді CREATE TRIGGER, які передаються в триггерную процедуру.

– TG_ARGV [ ]

Тип даних масив text. Аргументи від оператора CREATE TRIGGER. Індекс масиву починається з 0. Для неприпустимих значень індексу (< 0 або > = tgnargs) повертається NULL.

Тригерна функція повинна повернути або NULL, або запис/рядок, що відповідає структурі таблиці, для якої спрацював тригер.

Якщо BEFORE тригер рівня рядка повертає NULL, то всі подальші дії з цим рядком припиняються (тобто не спрацьовують наступні тригери, команда INSERT / UPDATE / DELETE для цього рядка не виконується). Якщо повертається не NULL, то подальша обробка триває саме з цим рядком. Повернення запису відмінного від початкового NEW, змінює запис, який буде вставлений або змінений.

Тому, якщо в тригерній функції потрібно виконати деякі дії і не міняти сам рядок, то потрібно повернути змінну NEW (або її еквівалент). Для того, щоб змінити рядок, що зберігається, можна поміняти окремі значення у змінній NEW і потім її повернути. Інший спосіб – створити і повернути повністю нову змінну. У разі рядкового тригера BEFORE для команди DELETE, саме значення, що повертається, не має прямого ефекту, але воно має бути відмінним від NULL, щоб не переривати обробку рядка. Змінна NEW завжди приймає значення NULL в триггерах на видалення – DELETE, а тому повертати її не має сенсу. Традиційно для тригерів DELETE є повернення змінної OLD.

Тригери INSTEAD OF можуть створюватися тільки як тригери рівня рядка і тільки для представлень. Якщо INSTEAD OF тригер повертає NULL, то це означає, що він не виконав жодних змін і подальша обробка цього рядка не вимагається (тобто для відповідної команди INSERT / UPDATE / DELETE не спрацьовують наступні тригера і не збільшується лічильник оброблених рядків). В інших випадках, повинно повертатися значення відмінне від NULL і, це означає, що тригер виконав необхідні дії.

Команди INSERT і UPDATE повинні повертати NEW (яка може бути змінена в тригерній функції) для коректної роботи INSERT RETURNING і UPDATE RETURNING. Значення, що повертається також впливає на значення рядка, який буде передано в наступні тригери. Для команди DELETE значення, що повертається, повинно бути OLD .

Значення, що повертається, для рядкового тригера AFTER і тригерів рівня BEFORE і AFTER завжди ігноруються. Дане значення може бути і NULL. У цих тригерах можна перервати команду, що їх викликала, явно викликавши помилку.

Приклад застосування тригера.

Дано таблицю Students що містить поле прізвище (Surname). Для даної таблиці створити тригер, який буде корегувати можливі помилки вводу користувача, такі як — зайві пропуски на початку та в кінці даних, невірний регістр літер (власні імена завжди пишуться з великої літери).

```
CREATE FUNCTION ToUpper(text) RETURNS text AS $$
	select translate( upper($1),
		'абвгґдеєжзиіїйклмнопрстуфхцчшщюяь',
		'АБВГҐДЕЄЖЗИІЇЙКЛМНОПРСТУФХЦЧШЩЮЯЬ')
$$ LANGUAGE SQL;

CREATE FUNCTION ToLower(text) RETURNS text AS $$
	select translate( lower($1),
		'АБВГҐДЕЄЖЗИІЇЙКЛМНОПРСТУФХЦЧШЩЮЯЬ',
		'абвгґдеєжзиіїйклмнопрстуфхцчшщюяь')
$$ LANGUAGE SQL;

CREATE FUNCTION CapitalizeFirstLetters(text) RETURNS text AS $$
declare
	Str alias for $1;
	Rez text;
	P integer;
begin
	Rez:=trim(both from ToLower(Str));
	Rez:=ToUpper(substr(Rez,1,1))||substr(Rez,2);
	P:=strpos(Rez,'-');
	if P>0 then
	Rez:=overlay(Rez placing ToUpper(substr(Rez,P+1,1)) from P+1 for 1);
	end if;
	P:=strpos(Rez,'(');
	if P>0 then
	Rez:=overlay(Rez placing ToUpper(substr(Rez,P+1,1)) from P+1 for 1);
	end if;
	return Rez;
end;
```

```
$$ LANGUAGE plpgsql;

CREATE FUNCTION updSurname() RETURNS trigger as $$
begin
    NEW.Surname = CapitalizeFirstLetters(NEW.Surname);
    return NEW;
end;
$$ LANGUAGE plpgsql;

CREATE TRIGGER updStudents BEFORE INSERT or UPDATE ON Students
FOR EACH ROW EXECUTE PROCEDURE updSurname;
```

5.3. Тригерна процедура PL/pgSQL

Тригер гарантує, що всякий раз, коли в таблиці добавляється або змінюється запис, у ньому зберігається інформація про поточного користувача і поточну мітку. Також контролюється, що ім’я співробітника указано і розмір зарплати задано більше нуля.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text );

CREATE FUNCTION emp_stamp()
  RETURNS trigger AS $$
BEGIN
    /* Переглянемо,  що вказане ім’я співробітника і зарплата */
    IF NEW.empname IS NULL THEN
        RAISE EXCEPTION  'Не указано имя сотрудника';
    END IF;
```

```
    IF NEW.salary IS NULL THEN
        RAISE EXCEPTION  'Не вказана зарплата для %',
        NEW.empname;
    END IF;
    /* Для чого працювати,  якщо за це щепотрібно  платити? /*
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION
        'У % не повинна бути відємна зарплата',
        NEW.empname;
    END IF;
    -- Запам'ятаємо хто і коли  змінив запис
    NEW.last_date  := current_timestamp;
    NEW.last_user  := current_user;
    RETURN NEW;   ~
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp
    BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Висновки

Мова PL/pgSQL належить до сімейства процедурних мов і є процедурним розширенням мови SQL. Завдяки цьому вона:

– може використовувати всі її типи даних, оператори і функції.

– дозволяє адаптувати програмний код, оскільки функції мови сумісні з усіма платформами, на яких працює СУБД PostgreSQL;

– дозволяє використовувати декілька команд SQL в програмному блоці PL/pgSQL за допомогою однієї операції, замість звичайної обробки кожної команди.

Це підвищує гнучкість використання і швидкодію команд SQL. Структура мови PL/pgSQL відносно проста, оскільки кожен її логічно відокремлений фрагмент коду існує у вигляді функції.

Матеріал посібника успішно може бути використаний в навчальному процесі вищих навчальних закладів на спеціальностях які вивчають роботу з базами даних, а також вчителями спеціалізованих загальноосвітніх шкіл, коледжів, ліцеїв, гімназій тощо.