Лабораторная работа № 5

Хранимые процедуры и функции

Использование хранимых процедур и функций позволяет реализовать основное преимущество архитектуры клиент-сервер: обработка данных может быть выполнена на сервере базы данных. Это существенно снижает время обработки и нагрузку на сеть.

- Процедуры используются в тех случаях, когда нужно внести изменения в данные. Они имеют входные и выходные параметры. Используя выходные параметры, можно вернуть в вызывающую среду необходимые данные.
- Задача функций состоит в том, чтобы выполнить обработку данных и вернуть в вызывающую среду результат этой обработки. Результат может представлять собой как скалярное значение, так и таблицу.

Хранимые процедуры

Хранимая процедура - это модуль, предназначенный для выполнения одного или нескольких действий с базой данных. Синтаксис оператора создания хранимой процедуры:

```
CREATE [OR REPLACE] PROCEDURE {имя процедуры}
[({список параметров})]
AS $$
[DECLARE
{объявление локальных переменных}]
BEGIN
{тело процедуры}
END $$ LANGUAGE '{язык программирования}';
```

Для создания подпрограмм можно использовать несколько языков программирования: PL/pgSQL, SQL, PL/Perl, PL/Pyton.

Как правило, процедура имеет один или несколько параметров, но могут быть процедуры без параметров. Тип данных параметров задается без указания размера, например :

```
numeric а не numeric(n,m);
varchar а не varchar(n).
```

Описание параметра содержит следующие элементы: имя; режим передачи – IN, OUT, INOUT; тип, значение по умолчанию.
Созданные процедуры являются объектами базы данных. Для вызова процедур используется оператор CALL

CALL {имя процедуры}({значения параметров});

Процедуру можно вызвать из

- анонимного блока;
- другой процедуры;
- внешнего приложения.

При определении формальных параметров процедуры указываются допустимые способы его использования, которые называют режимами использования. Существует три режима использования формальных параметров: IN, OUT, INOUT.

Параметры в режиме IN используются для получения данных из вызывающей программы, соответствующие им фактические параметры могут представлять собой константы, переменные или выражения. Режим передачи IN, используется по умолчанию и его необязательно указывать.

Параметры в режиме OUT используются для передачи данных из процедуры в вызывающую программу, соответствующие им фактические параметры должны обязательно быть переменными совместимого типа.

Параметры в режиме INOUT можно использовать как для получения данных из вызывающей программы, так и для передачи результатов обработки из процедуры в вызывающую программу.

Фактические параметры соответствующие формальным параметрам с режимом передачи IN OUT должны быть переменными совместимого типа.

В листинге 1 приведен пример создания процедуры **Emp_Add_Sal**, которая предназначена для увеличения зарплаты сотрудника. Номер сотрудника и размер увеличения зарплаты, являются параметрами процедуры. В вызывающую среду передается новое значение заработной платы.

Листинг 1. Создание процедуры Emp_Add_Sal, для изменения зарплаты сотрудника.

```
CREATE OR REPLACE PROCEDURE Emp_Add_Sal(p_id integer, p_add
numeric, p_salary OUT numeric)
AS $$
BEGIN
   UPDATE Employees
   SET salary = salary+p_add
   WHERE employee_id = p_id;
--
   SELECT salary INTO p_salary
   FROM Employees
   WHERE employee_id = p_id;
END $$ LANGUAGE 'plpgsql';
```

Листинг 2. Вызов процедуры Emp_Add_Sal.

```
DO $$
DECLARE
    v_emp_id integer :=110;
    v_add_salary numeric(8,2):=500;
    v_emp_salary numeric(8,2);
BEGIN
    RAISE NOTICE 'Результат:';
    CALL Emp_Add_Sal(v_emp_id,v_add_salary,v_emp_salary);
--
```

```
     RAISE NOTICE 'employee_id = % % %', v_emp_id,
                  'new_salary = ', v_emp_salary;
END $$;
```

```
Результат:
employee_id = 110 new_salary =  9850.00
```

Рассмотрим еще одну задачу. Нужно изменить статус заказа по его номеру, с выполнением правила: нельзя изменять статус заказа, если его текущий статус равен Shipped.

Листинг 3. Создание процедуры Ord_Upd_Status, для изменения статуса заказа по его номеру.

```
CREATE OR REPLACE PROCEDURE Ord_Upd_Status(p_id integer,
p_status varchar)
AS $$
DECLARE
   v_status varchar(20);
BEGIN
   SELECT status INTO v_status
   FROM orders_copy
   WHERE order_id = p_id;
--
  IF v_status= 'Shipped'
  THEN
      RAISE NOTICE ' Статус заказа % %', p_id,
                   ' изменить нельзя';
  ELSE
      UPDATE orders_copy
      SET status = p_status
      WHERE order_id = p_id;
      RAISE NOTICE ' Статус заказа % % %', p_id,
                   'изменен на', p_status;
  END IF;
END $$ LANGUAGE 'plpgsql';
```

Листинг 4. Вызов процедуры Ord_Upd_Status, для изменения статуса заказа по его номеру.

```
DO $$
```

```
DECLARE
    v_order_id integer :=101;
    v_status varchar(20):='Canceled';
BEGIN
    RAISE NOTICE 'Результат:';
    CALL Ord_Upd_Status(v_order_id,v_status);
END $$;
```

```
Результат:
 Статус заказа 101 изменен на Canceled
```

Способы связывания формальных и фактических параметров

Можно использовать два метода связывания фактических параметров с формальными параметрами:

- позиционное связывание;
- связывание по имени.

При использовании позиционного связывания, соответствие между фактическими и формальными параметрами устанавливается по их позиции, в списке параметров.

В листинге 14.7 приведен пример оператора создания процедуры с параметрами, которая предназначена для добавления данных о новом заказе с проверкой следующего правила: новый заказ можно добавлять, если сумма заказов, находящихся в состоянии ожидания ('Pending'), для данного клиента, не превышает его кредитного лимита.

Листинг 5. Создание процедуры с параметрами **Add_Order**, которая предназначена для добавления данных о новом заказе.

```
CREATE OR REPLACE PROCEDURE Add_Order(
       p_odder_id integer, p_customer_id integer,
       p_status varchar DEFAULT'Pending',
       p_salesman_id integer DEFAULT NULL,
       p_order_date date default CURRENT_DATE)
AS $$
DECLARE
```

```
    v_credit_limit numeric(10,2);
    v_orders_sum  numeric(10,2);
BEGIN
   SELECT credit_limit into  v_credit_limit
   FROM Customers
   WHERE customer_id =  p_customer_id;
--
   SELECT SUM(quantity*unit_price) into v_orders_sum
   FROM Orders JOIN Order_Items USING (order_id)
   WHERE customer_id =  p_customer_id
     AND status = 'Pending';
--
   IF v_orders_sum > v_credit_limit
     THEN
       RAISE NOTICE 'Операция отклонена, так как у клиента % %',
                    p_customer_id,' превышен кредитный лимит';
     ELSE
       RAISE NOTICE ' Данные о заказе успешно добавлены';
       INSERT INTO Orders
       VALUES(p_odder_id, p_customer_id, p_status,
             p_salesman_id, p_order_date);
   END IF;
END $$ LANGUAGE plpgsql
```

Листинг 6. Вызов процедуры Add_Order, с использованием значений по умолчанию и связыванием параметров по имени.

```
DO $$
BEGIN
    RAISE NOTICE 'Результат:';
    CALL Add_Order(p_odder_id=>122, p_salesman_id=>165,
                   p_customer_id=>5);
END $$;
```

```
Результат:
 Данные о заказе успешно добавлены
```

Хранимые функции

Хранимая функция – это подпрограмма, который принимает значения одного или нескольких параметров, выполняет обработку данных, и возвращает полученный результат в вызывающую программу. Возможны функции без параметров, но реально, функции без параметров используются довольно редко. Синтаксис оператора для создания хранимой функции выглядит следующим образом:

```
CREATE [OR REPLACE] FUNCTION {имя функции}
[{ список параметров}]
RETURNS {возвращаемый тип данных}
AS $$
[DECLARE
{объявление локальных переменных}]
BEGIN
   {Тело функции
      RETURN {возвращаемый результат}
    }
END; $$ LANGUAGE 'plpgsql';
```

После служебного слова RETURNS, указывается тип данных возвращаемых функцией. Функции могут возвращать данные практически любого типа, как скалярные типы данных, так и данные имеющие сложную структуру: коллекций, курсорные переменные, объектные типы.

В теле функции должен содержаться оператор RETURN, после этого оператора указывается значение, возвращаемое в программу, которая вызвала функцию. Функция может содержать несколько операторов RETURN.

Переедем к рассмотрению конкретных примеров создания и использования функций.

Листинг 7. Создание функции SUM_SAL, которая возвращает общую сумму покупок клиента.

```
CREATE OR REPLACE FUNCTION SUM_SAL(p_cust_id integer)
returns numeric
AS $$
DECLARE
   v_sum_sal  numeric(10,2);
BEGIN
   SELECT  SUM(quantity*unit_price) INTO v_sum_sal
   FROM Orders JOIN Order_Items USING (order_id)
   WHERE customer_id = p_cust_id;
   --
   RETURN  v_sum_sal;
END $$ LANGUAGE plpgsql;
```

В отличие от вызова процедуры, который представляет собой отдельный оператор, вызов функции является частью исполняемого оператора PL/pgSQL. Для вызова функции, в исполняемом операторе нужно указать имя функции и значение ее параметров, вместо переменной, имеющей тип данных возвращаемого функцией.

Листинг 8. Пример использования функции SUM_SAL.

```
DO $$
DECLARE
   v_sum_sal  numeric(10,2);
   v_cust_id integer:=3;
BEGIN
    RAISE NOTICE 'Результат:';
     v_sum_sal:= SUM_SAL(v_cust_id);
     RAISE NOTICE ' Сумма покупок клиента % % %', v_cust_id ,'
равна ',  v_sum_sal ;
END $$
```

```
Результат:
 Сумма покупок клиента 3  равна  819220.00
```

Если тип данных, которые возвращаются функцией, поддерживается SQL, то такие функции можно использовать в операторах SQL. В листинге 9 содержится пример запроса SQL, который использует функцию SUM_SAL

Листинг 9. Вывести номера и суммы продаж клиентов, у которых сумма покупок >1000000.

```
SELECT customer_id, SUM_SAL(customer_id) AS "SUM_SALES"
FROM Customers
WHERE SUM_SAL(customer_id) > 1000000
```

```
customer_id|SUM_SALES |
-----------+----------+
         45|1221050.00|
         48|1512100.00|
         49|1880350.00|
```

Хранимые функции, возвращающие таблицу

Этот вид хранимых функций позволяет осуществлять сложную обработку данных, требующую использование операторов PL/pgSQL, с возможностью обращаться к этим данным в операторах SQL.

Можно также использовать эти функции, для замены представлений. Преимущество табличных функций заключается в том, что они имеют параметры, и могут возвращать разные данные, которые будут зависеть от значений параметров. Заголовок оператора создания функции, возвращающей таблицу, должен быть представлен в следующем виде:

```
CREATE [OR REPLACE] FUNCTION {имя функции}
[{ список параметров}]
RETURNS TABLE {список: имя столбца тип}
```

В теле функции, для каждой строки возвращаемой таблицы, столбцам нужно присвоить значение, и выполнить оператор **RETURN NEXT.**

Изучение этих функций начнем с рассмотрения функции **Tab_Emp,** которая должна возвращать данные о сотрудниках имеющих заданное значение JOB_ID.

Листинг 10. Создание функции **Tab_Emp**, которая возвращает данные о сотрудниках, имеющих заданное значение **job_id**.

```
CREATE OR REPLACE FUNCTION Tab_Emp(p_job_id IN VARCHAR)
RETURNS TABLE
(id_emp integer,
f_name VARCHAR(25),
l_name VARCHAR(25),
dep_id integer,
job VARCHAR(10))
AS $$
DECLARE
   Cur_Emp cursor(p_id varchar) FOR
   SELECT employee_id, first_name, last_name,
          department_id,job_id
   FROM Employees
   WHERE job_id =p_id;
BEGIN
   FOR emp_rec IN Cur_Emp(p_job_id)
   LOOP
      id_emp := emp_rec.employee_id;
      f_name := emp_rec.first_name;
      l_name := emp_rec.last_name;
      dep_id := emp_rec.department_id;
      job    := emp_rec.job_id;
      RETURN NEXT;
   END LOOP;
END; $$ LANGUAGE plpgsql;
```

Создадим еще одну функцию с именем **Tab_Emp**, в которой, в условии выбора, используется столбец **department_id.**

Листинг 11. Создание функции **Tab_Emp**, которая возвращает данные о сотрудниках, имеющих заданное значение **department_id**.

```
CREATE OR REPLACE FUNCTION Tab_Emp(p_dep_id integer)
```

```
RETURNS table
(id_emp integer,
f_name VARCHAR(25),
l_name VARCHAR(25),
dep_id integer,
job VARCHAR(10))
AS $$
DECLARE
   Cur_Emp cursor(p_id integer) FOR
   SELECT employee_id, first_name, last_name,
          department_id,job_id
   FROM Employees
   WHERE department_id =p_id;
BEGIN
   FOR emp_rec IN Cur_Emp(p_dep_id)
   LOOP
      id_emp := emp_rec.employee_id;
      f_name := emp_rec.first_name;
      l_name := emp_rec.last_name;
      dep_id := emp_rec.department_id;
      job    := emp_rec.job_id;
      RETURN next;
   END LOOP;
END; $$ LANGUAGE plpgsql
```

Проверим работу этих функций.

```
SELECT * FROM Tab_Emp('ST_MAN')
```

```
id_emp|f_name |l_name  |dep_id|job   |
------+-------+--------+------+------+
   122|Payam  |Kaufling|    50|ST_MAN|
   123|Shanta |Vollman |    50|ST_MAN|
   124|Kevin  |Mourgos |    50|ST_MAN|
   120|Matthew|Weiss   |    50|ST_MAN|
   121|Adam   |Fripp   |   300|ST_MAN|
```

```
SELECT * FROM Tab_Emp(60)
```

```
id_emp|f_name   |l_name   |dep_id|job    |
------+---------+---------+------+-------+
   104|Bruce    |Ernst    |    60|IT_PROG|
   107|Diana    |Lorentz  |    60|IT_PROG|
   103|Alexander|Hunold   |    60|IT_PROG|
   106|Valli    |Pataballa|    60|IT_PROG|
   105|DAVID    |Austin   |    60|IT_PROG|
```

Задание

Задача 1. Создать хранимую процедуру, которая увеличивает на 1 рейтинг товаров, если их рейтинг меньше 5, и сумма продаж больше p_sum_sal. Входной параметр: p_sum_sal.

Задача 2. Создать хранимую процедуру с параметрами **p_summ_min, p_summ_max**, которая изменяет значение кредитного лимита клиентов в зависимости от суммы оформленных заказов v_summ. По следующему правилу:

- Если v_summ > p_summ_max, то повысить кредитный лимит на 10%.
- Если p_summ_min <= v_summ <= p_summ_max, то оставить текущее значение кредитного лимита.
- Если v_summ < p_summ_min, то уменьшить кредитный лимит на 10%.

Задача 3. Создать хранимую функцию, которая возвращает 1 в том случае, если рассматриваемый сотрудник осуществлял продажи товара с заданным номером, и 0 в противном случае. Входные параметры: номер сотрудника, номер товара.

Задача 4. Создать хранимую функцию, которая возвращает общую сумму продаж заданного товара за заданный промежуток времени. Входные параметры: номер товара, начало периода, конец периода.

Задача 5. Создать табличную функцию, которая возвращает данные о N товарах с максимальными суммами продаж. Значение N задать в виде параметра.