**Лабораторные работы по курсу**
**Базы данных**

# Лабораторная работа 7
# «Процедуры, функции и триггеры»

**Москва, 2023**

# Оглавление

# 1. Теоретическая часть

## 1.1. Язык программирования PL/pgSQL

### 1.1.1. Введение в PL/pgSQL.

До текущего раздела мы писали запросы к базе данных на языке SQL – декларативном языке программирования. Однако, в некоторых случаях, при составлении запросов со сложной логикой, было бы удобнее писать запрос в стандартном для нам императивном стиле, когда все команды выполняются последовательно. В качестве решения предлагается использовать процедурный язык PL/pgSQL. Он является наследником языка PL/SQL, предназначенного для работы с СУБД Oracle, который в свою очередь наследует языки Ada и Pascal. Таким образом, структура программы на PL/pgSQL может напомнить структуру программы на языке Pascal.

В общем виде, структуру программы на языке PL/pgSQL можно представить следующим образом:

```
DECLARE
    Объявления переменных
BEGIN
    операторы
END;
```

Первым блоком в программе является объявление переменных, которые будут использоваться в программе. Переменные могут иметь любой тип данных, поддерживаемый PostgreSQL.
Общий синтаксис объявления переменной:

```
имя [ CONSTANT ] тип [ COLLATE имя_правила_сортировки ] [ NOT NULL ] [ { DEFAULT | := | = } выражение ]
```

По умолчанию, переменным присваивается значение NULL. В случае необходимости, переменную возможно проинициализировать значением, использовав оператор «:=» для присваивания.

Далее следует код программы в виде последовательно записанных операторов. Код должен быть заключен в специальные скобки из операторов BEGIN и END.

Для запуска скрипта на PL/pgSQL необходимо перед ним указать ключевое слово *do* и заключить его в одинарные кавычки для того, чтобы он распознавался, как строка. Однако в большинстве случаев такой подход неудобен – все кавычки внутри скрипта придется экранировать и исчезнет встроенная подсветка синтаксиса. Поэтому для удобства, вместо кавычек используется символ $$[1]

Приведем простейший скрипт на языке PL/pgSQL.

```
do
$$
BEGIN
      raise notice 'Hello, MIET';
END
$$;
```

```
NOTICE:  Hello, MIET
```

Команда *raise notice* служит для вывода служебных сообщений на экран.

Усложним скрипт, сохранив значение строки в переменной *our_text*.

[1] В общем виде строковые константы можно представить в виде $tag$<string>$tag$, где tag – необязательное поле. Между подобными скобками может содержаться любое число кавычек, специальных символов которые не нужно экранировать

```
do
$$
DECLARE
	our_text TEXT := 'MIET';
BEGIN
	raise notice 'Hello, %', our_text;
END
$$;
```

Для вывода значения переменной с помощью *raise notice,* необходимо указать символом % место, на которое будет подставлено её значение.

```
NOTICE:  Hello, MIET
```

Внутри скрипта возможно использовать стандартные команды языка SQL. Обращаться к значениям, полученным в результате подобного запроса возможно несколькими способами.

Чтобы сохранить результат выполнения запроса *select* в переменной, возможно использовать ключевое слово *into*.

Общий синтаксис запроса:

```
SELECT []
INTO variable
FROM Table
```

Приведем пример, выводящий общее количество студентов в институте:

```
do
$$
DECLARE
	num_of_students INTEGER;
BEGIN
	SELECT count(*)
	INTO num_of_students
	FROM student;
	raise notice 'Количество студентов: %', num_of_students;
END
$$;
```

Обратите внимание, что подобным образом возможно сохранить только одно значение.

Если мы хотим сохранить в переменной значение целой строки, то возможно использовать тип *record*. Данный тип сохраняет внутри себя всю полученную строку, обращение к полям которой возможно с помощью точки.

В следующем примере происходит поиск фамилии и имени студента с указанным номером студенческого билета.

```
do
$$
DECLARE
	students record;
BEGIN
	SELECT surname, name
	INTO students
	FROM student
	WHERE student_id = 838389;
	raise notice 'Имя: %, Фамилия: %', students.name, students.surname;
END
$$;
```

Модернизируем скрипт, сохранив значение номера студенческого билета в качестве константы. Напомним, что константа – значение, которое инициализируется в

начале выполнения кода программы и не может быть изменено внутри него. Для этого, после имени переменной укажем ключевое слово *constant*.

```
do
$$
DECLARE
      students record;
      studentid constant INTEGER := 838389;
BEGIN
      SELECT surname, name
      INTO students
      FROM student
      WHERE student_id = studentid;
      raise notice 'Имя: %, Фамилия: %', students.name, students.surname;
END
$$;
```

Представим ситуацию, что пользователь неверно ввел значение студенческого билета и студент не был найдет в таблице.

Тогда указанный выше скрипт вернет значения NULL.

```
NOTICE:  Имя: <NULL>, Фамилия: <NULL>
```

Чтобы обойти подобную ошибку, возможно использовать оператор ветвления.

### 1.1.2. Оператор ветвления

В общем виде оператор ветвления можно представить в следующем виде:

```
if condition then
  statements;
else
  alternative-statements;
END if;
```

Как и во многих других языках программирования, ветвь *else* является необязательной.

Для определения результата выполнения запроса воспользуемся переменной found. Она принимает значение true, в случае возвращения значения запросом и false, если было возвращено пустое множество.

```
do
$$
DECLARE
      students record;
      studentid constant INTEGER := 8383890;
BEGIN
      SELECT surname, name
      INTO students
      FROM student
      WHERE student_id = studentid;
      if not found then
      raise notice'The student % could not be found', studentid;
      else
            raise notice 'Имя: %, Фамилия: %', students.name,
students.surname;
  end if;
END
$$;
```

Тогда при вводе некорректного значения будет выведено сообщение о том, что указанный студент не был найден.

```
NOTICE:  The student 8383890 could not be found
```

### 1.1.3. Операторы циклов

В языке PL/pgSQL существует несколько способов задать цикл. Рассмотрим некоторые из них.

Один из самых простых способов задать цикл с условием – цикл с использованием ключевого слова WHILE. В общем виде цикл можно представить следующим образом:

```
WHILE логическое-выражение LOOP
    операторы
END LOOP
```

Приведем простой пример вывода последовательных чисел на экран

```
do $$
declare
   iterator integer := 0;
begin
   while iterator < 5 loop
      raise notice 'I = %', iterator;
        iterator := iterator + 1;
   end loop;
end$$;
```

```
NOTICE:  I = 0
NOTICE:  I = 1
NOTICE:  I = 2
NOTICE:  I = 3
NOTICE:  I = 4
```

Аналогичную задачу возможно решить с помощью цикла с заданным числом итераций FOR. В общем виде цикл представляет собой следующую структуру:

```
FOR имя IN [REVERSE] выражение .. выражение [BY выражение] LOOP
    операторы
END LOOP;
```

Тогда решение можно записать в виде следующего скрипта:

```
do $$
begin
   for iterator in 0..4 loop
      raise notice 'I = %', iterator;
   end loop;
end $$;
```

Для прохода по циклу в обратном порядке возможно использовать ключевое слово REVERSE и указания диапазона от большего числа к меньшему.

```
do $$
begin
   for iterator in REVERSE 4..0 loop
      raise notice 'I = %', iterator;
   end loop;
end $$;
```

```
NOTICE:  I = 4
NOTICE:  I = 3
NOTICE:  I = 2
NOTICE:  I = 1
NOTICE:  I = 0
```

По умолчанию, за каждый шаг цикла значение итератора инкрементируется на 1. Чтобы изменить данный шаг, возможно использовать ключевое слово BY.

```
do $$
begin
   for iterator in 0..4 by 2 loop
        raise notice 'I = %', iterator;
   end loop;
end; $$
```

```
NOTICE:  I = 0
NOTICE:  I = 2
NOTICE:  I = 4
```

В предыдущих примерах мы могли получать только одно значение из таблицы. Используя цикл, возможно проходиться по всем записям таблицы, выводить и обрабатывать их последовательно. Для этого используется следующая конструкция

```
FOR цель IN запрос LOOP
    операторы
END LOOP;
```

В следующем примере выведем всех студентов с именем Александр.

```
do
$$
DECLARE
      students record;
      student_name constant TEXT := 'Александр';
BEGIN
      FOR students IN SELECT surname, name
            FROM student
            WHERE name = student_name
      loop
            raise notice 'Имя: %, Фамилия: %', students.name,
students.surname;
   end loop;
END
$$;
```

```
NOTICE:  Имя: Александр, Фамилия: Гаврилов
NOTICE:  Имя: Александр, Фамилия: Егоров
NOTICE:  Имя: Александр, Фамилия: Молчанов
NOTICE:  Имя: Александр, Фамилия: Михеев
```

Сделаем небольшое замечание. При выполнении запроса SELECT внутри скрипта результирующее значение загружается с сервера, где хранится база данных и передается на клиент, где сохраняется внутри переменной *students* типа *record*. Представьте себе, что результат запроса вернет огромное число строк. Передача подобного объема данных займет множество ресурсов. Также, сохранять их всех внутри данной переменной не существует возможности. Поэтому, создатели языка предусмотрели определенную переменную, называемую **курсором**. С его помощью возможно получать данные от запроса порциями, не переполняя при этом память. В языке PL/pgSQL такая переменная создается автоматически при использовании цикла «*FOR цель IN запрос*». Однако, во многих других процедурных надстройках SQL такой возможности нет, поэтому необходимо создавать курсор вручную. Более подробно об этом возможно прочитать в документации [1].

Для удобства сведем наиболее популярные управляющие структуры в таблицу.

| | |
|---|---|
| Присваивание | `переменная := выражение;` |
| Ветвление | `IF условие THEN`<br>`оператор;`<br>`ELSE`<br>`оператор;`<br>`END IF` |
| Бесконечный цикл | `LOOP`<br>`оператор; ...`<br>`END LOOP;` |
| Цикл while | `WHILE логическое-выражение LOOP`<br>`операторы` |

| | |
|---|---|
| | **END LOOP** |
| Целочисленный цикл | **FOR** ***имя*** **IN [REVERSE]** ***выражение*** **..** ***выражение*** **[BY** ***выражение*****] LOOP**<br>    ***операторы***<br>**END LOOP;** |
| Цикл по результатам запроса | **FOR** ***цель*** **IN** ***запрос*** **LOOP**<br>    ***операторы***<br>**END LOOP;** |

### 1.2. Хранимые подпрограммы

При выполнении запросов к базе данных происходит пересылка данных по сети от приложения клиента к серверу баз данных. В некоторых случаях, данная операция является затратной по времени выполнения. Для того, чтобы выполнять запросы более эффективно, возможно создать отдельную подпрограмму, которая будет выполняться в рамках процессов сервера баз данных. В СУБД PostgreSQL существуют два типа подобных подпрограмм – процедуры и функции. Их код может быть написан как на языке SQL, так и на других языках программирования – C, PL/pgSQL, Python, Tcl, Perl, R, Java, JavaScript и др. В данном лабораторном практикуме будут рассмотрены подпрограммы, написанные на языке SQL.

Функции и процедуры имеют несколько особенностей, отличающих их друг от друга. Приведем их сравнение в виде таблицы.

| **Функция** | **Процедура** |
|---|---|
| **Имеет** возвращаемый тип и возвращает значение | **Не имеет** возвращаемого типа |
| Использование запросов на добавление, изменение и удаление строк **невозможно**. Разрешены только SELECT-запросы | Использование запросов на добавление, изменение и удаление строк **возможно** |
| **Не имеет** выходных аргументов | **Имеет** входные и выходные аргументы |
| Использовать транзакции **запрещено** | **Возможно** использовать операции управлением транзакциями |

Рассмотрим их более подробно.

#### 1.2.1. Процедуры

Процедуры – подпрограммы, которые не могут возвращать значения. Чаще всего процедуры используются для модификации данных в таблице – добавление, изменение или удаление.

Для создания процедуры существует команда CREATE PROCEDURE. Её сокращенный синтаксис приведен ниже.

```
CREATE [ OR REPLACE ] PROCEDURE
    имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ { DEFAULT |
= } выражение_по_умолчанию ] [, ...] ] )
  LANGUAGE имя_языка
  AS $$
  …
  $$;
```

Рассмотрим несколько примеров. Создадим процедуру для добавления студента в базу данных.

```
CREATE PROCEDURE add_Student(Student_ID bigint, datebegin date, dateend date,
F VARCHAR(30), I VARCHAR(30), O VARCHAR(30), groupe VARCHAR(7) ,birthday
date, email VARCHAR(30))
LANGUAGE SQL
AS $$
INSERT INTO student VALUES (Student_ID, F, I, O, groupe, birthday, email);
INSERT INTO student_id VALUES (Student_ID, datebegin, dateend);
$$;
```

После ключевых слов CREATE PROCEDURE следует название процедуры. Далее в скобках указываются входные параметры, в виде «*параметр тип*». В теле процедуры указывается язык, на котором она написана и непосредственно сам её код.

Приведем еще один пример. Аналогично создадим процедуру, позволяющую удалять записи о студенте из базы данных.

```
CREATE PROCEDURE del_Student(Student_ID bigint)
LANGUAGE SQL
AS $$
DELETE FROM student_id WHERE student_id = Student_ID;
DELETE FROM student WHERE student_id = Student_ID;
DELETE FROM field_comprehension WHERE student_id = Student_ID;
$$;
```

### 1.2.2. Функции

Функции – подпрограммы, которые могут возвращать значения. Сокращенный синтаксис функции представлен ниже.

```
CREATE [ OR REPLACE ] FUNCTION
    имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ { DEFAULT |
= } выражение_по_умолчанию ] [, ...] ] )
    [ RETURNS тип_результата
      | RETURNS TABLE ( имя_столбца тип_столбца [, ...] ) ]
  { LANGUAGE имя_языка
  }
```

В зависимости от количества возвращаемых значений функции могут быть скалярными и составными.

Скалярные функции – функции, возвращающие одно значение базового типа, например integer.

Приведем пример скалярной функции, возвращающей значение даты окончания действия студенческого билета по его номеру.

```
CREATE OR REPLACE FUNCTION find_date(Student_ID bigint) RETURNS date
LANGUAGE SQL
AS $$
SELECT expiration_date
FROM student_id
WHERE student_id = Student_ID
$$;
```

Вызов скалярной функции происходит с помощью запроса SELECT:

```
SELECT find_date(841576)
```

```
 find_date
------------
 2025-08-31
```

Составные функции

Функция с PostgreSQL может возвращать набор некоторых значений. Далее к этой функции возможно обращаться, как к таблице и выбирать из нее данные. Для того, чтобы

она возвращало множество значений, необходимо определить **составной тип**. Составной тип очень похож на структуру в языке C. Он состоит из списка имён полей и соответствующих им типов данных. В общем виде, синтаксис можно представить следующим образом:

```
CREATE TYPE название AS (
    переменная       тип,
    переменная       тип,
      ...
);
```

Приведем пример следующей задачи. Необходимо написать функцию, которая выводит информацию о студентах, чей студенческий билет закончил действовать до указанной даты. Для этого создадим составной тип Student_ID_date.

```
CREATE TYPE Student_ID_date AS(
      surname varchar(30),
      name varchar(30),
      id_date date
);
```

Далее создадим составную функцию find_id_date. Её основное отличие в синтаксисе от скалярной заключается в том, что функция возвращает тип Student_ID_date с использованием ключевого слова SETOF.

```
CREATE OR REPLACE FUNCTION find_id_date(id_date date) RETURNS SETOF
Student_ID_date
LANGUAGE SQL
AS $$
SELECT surname, name, expiration_date
FROM student_id
INNER JOIN student ON student.student_id = student_id.student_id
WHERE expiration_date<id_date
$$;
```

Вызов функции будет совпадать с обращением к таблице:

```
SELECT * FROM find_id_date('10/09/2024');
```

```
  surname  |  name   |  id_date
-----------+---------+------------
 Дроздов   | Марк    | 2024-08-31
 Самсонов  | Максим  | 2024-08-31
 Смирнов   | Ярослав | 2024-08-31
 Соловьев  | Сергей  | 2024-08-31
 Селиванов | Дмитрий | 2024-08-31
 Егоров    | Артём   | 2024-08-31
 Грачев    | Андрей  | 2024-08-31
 Маслов    | Михаил  | 2024-08-31
```

### 1.2. Триггеры

Для упрощения работы с базой данных было бы полезно совершать некоторые постоянные автоматические действия при наступлении определенного события. Например, вести лог обо всех изменениях в таблице с оценками студентов. Или автоматически рассчитывать поле, содержащее количество должников при получении студентом двойки. Для подобной работы используются триггеры.

Триггер – подпрограмма, автоматически выполняемого при наступлении заданного события. Созданный триггер будет связан с таблицей и будет выполнять заданную функцию при наступлении определенного события. Триггер может выполняться до указанного события (`BEFORE`), после него (`AFTER`) или вместо (`INSTEAD OF`). В качестве

события может быть операция добавления (INSERT), обновления (UPDATE) или удаления значений (DELETE). [1]
Существуют два типа триггеров – построчные и операторные. Построчные триггеры вызываются один раз для каждой строки, с которой произошло изменение. Операторный триггер в таком случае срабатывает только один раз.

Создание триггера происходит в два этапа – создание триггерной функции и прикрепление её к определенной таблице. Триггерная функция может быть написана только на одном из процедурных языков программирования, поддерживаемых PostgreSQL, например PL/pgSQL.

### 1.2.1. Разработка триггерной функции

В общем виде триггерную функцию можно представить в следующем виде:

```
CREATE FUNCTION trigger_function()
   RETURNS TRIGGER
   LANGUAGE PLPGSQL
AS $$
BEGIN
   -- trigger logic
END;
$$
```

Триггерная функция получает данные об изменениях в базе данных через специальную структуру *TriggerData*, которая содержащую набор переменных. Наиболее часто используемые переменные – NEW и OLD, содержащие строки до и после выполнения события, вызвавшего триггер.

Например, создадим триггерную функцию, сохраняющую информацию об изменениях студентами фамилий. Перед этим создадим таблицу, которая будет хранить общую информацию об изменениях.

```
  CREATE TABLE sys_log (
   id INT GENERATED ALWAYS AS IDENTITY,
   info TEXT NOT NULL
);
```

Создадим функцию:

```
CREATE OR REPLACE FUNCTION log_last_name_changes()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN
      IF NEW.surname <> OLD.surname THEN
            INSERT INTO sys_log(info)
            VALUES(concat(OLD.surname, ' сменил фамилию на ',NEW.surname));
      END IF;
      RETURN NEW;
END;
$$
```

### 1.2.2. Создание триггера

В общем случае синтаксис команды создания триггера - CREATE TRIGGER выглядит следующим образом:

```
CREATE [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER | INSTEAD OF } { событие [
OR ... ] }
    ON имя_таблицы
    [ FROM ссылающаяся_таблица ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( условие ) ]
    EXECUTE PROCEDURE имя_функции ( аргументы )

Здесь допускается событие:

    INSERT
    UPDATE [ OF имя_столбца [, ... ] ]
    DELETE
    TRUNCATE
```

Для прикрепления триггерной функции к таблице создадим непосредственно сам триггер.

```
CREATE OR REPLACE TRIGGER last_name_changes
  BEFORE UPDATE
  ON student
  FOR EACH ROW
  EXECUTE PROCEDURE log_last_name_changes();
```

Теперь, при изменении фамилии любого студента информация от этом будет сохранена в таблице *sys_log*.

## 2. Практическая часть

### 2.1. Задание 1.

Напишите запрос по варианту

| № | Условие запроса |
|---|---|
| 1 | Напишите скрипт, выводящий количество всех оценок 5, 4, 3, 2 в единой таблице |
| 2 | Напишите скрипт, в результате работы которого будет выведено ФИО преподавателя, его ставка и реальная занятость – сумма ЗЕТ всех читаемых им дисциплин |
| 3 | Напишите скрипт, формирующий таблицу со средним баллом по каждой дисциплине у преподавателей Института МПСУ |
| 4 | Напишите скрипт, в который возвращает студентов со счастливым студенческим билетом (сумма первых трех цифр номера билета совпадает с суммой последних трех) |
| 5 | Напишите скрипт, выводящий ФИО студента и его пол. |

### 2.2. Задание 2.

Напишите запрос по варианту

| № | Условие запроса |
|---|---|
| 1 | Создайте процедуру добавления нового преподавателя |
| 2 | Создайте процедуру добавления/изменения ученого звания у преподавателя |
| 3 | Создайте процедуру перемещения студента из одной группы в другую |
| 4 | Создайте процедуру полного изменения студенческого билета |
| 5 | Создайте процедуру обновления условий для преподавателя |

### 2.3. Задание 3.

Напишите запрос по варианту

| № | Условие запроса |
|---|---|
| 1 | Сделайте функцию, которая выводит всю информацию о преподавателе по его id |
| 2 | Сделайте функцию, которая выводит всех студентов по определенной группе |
| 3 | Сделайте функцию, которая выводит ср оценку по всем предметам у студента (по id студента) |
| 4 | Сделайте функцию, которая будет считать зарплату преподавателя, по id преподавателя |
| 5 | Сделайте функцию, которая считает количество студентов, в какой-нибудь группе |

### 2.4. Задание 4.

Напишите запрос по варианту

| № | Условие запроса |
|---|---|
| 1 | Создайте процедуру удаления преподавателя |
| 2 | Создайте процедуру удаления ученого звания |
| 3 | Создайте процедуру замены предмета у преподавателя |
| 4 | Создайте процедуру замены имейла у студента |
| 5 | Создайте процедуру продления студ билетов на 1 год |

### 2.5. Задание 5.

2.5.1. Создайте триггер, автоматически добавляющий студента в таблицу с должниками при получении двойки.

2.5.2. Напишите запрос по варианту

| № | Условие запроса |
|---|---|
| 1 | Создайте триггер, который удаляет всю строчку из базы, если удалить любую его часть. |
| 2 | Создайте триггер, который запрещает добавление студента, возраст которого превышает 100 лет |
| 3 | Создайте триггер, который запрещает добавление студенческого билета, у которого дата выдачи > срок действия билета |
| 4 | Создайте триггер, который запрещает изменение в структурных подразделениях |
| 5 | Создайте триггер, который пишет в служебном письме “Are your sure about that?” при добавлении студента John Cena |

# Список литературы


[1] «Исходный код СУБД postgres,» [В Интернете]. Available: https://github.com/postgres/postgres. [Дата обращения: 30 01 2023].

[2] Документация к PostgreSQL 15.1, 2022.

[3] Е. Рогов, PostgreSQL изнутри, 1-е ред., Москва: ДМК Пресс, 2023, p. 662 .

[4] Б. А. Новиков, Е. А. Горшкова и Н. Г. Графеева, Основы технологии баз данных, 2-е ред., Москва: ДМК пресс, 2020, p. 582.

[5] Е. П. Моргунов, PostgreSQL. Основы языка SQL, 1-е ред., Санкт-Петербург: БХВ-Петербург, 2018, p. 336.