**Лабораторные работы по курсу**

**Базы данных**

**Лабораторная работа 6**

**«Процедуры, функции, триггеры и представления»**

**Москва, 2024**

Оглавление

[1. Теоретическая часть 3](#_Toc156254177)

[1.1. Язык программирования PL/pgSQL 3](#_Toc156254178)

[1.1.1. Введение в PL/pgSQL. 3](#_Toc156254179)

[1.1.2. Оператор ветвления 5](#_Toc156254180)

[1.1.3. Операторы циклов 5](#_Toc156254181)

[1.2. Хранимые подпрограммы 8](#_Toc156254182)

[1.2.1. Процедуры 8](#_Toc156254183)

[1.2.2. Функции 9](#_Toc156254184)

[1.3. Триггеры 10](#_Toc156254185)

[1.3.1. Разработка триггерной функции 11](#_Toc156254186)

[1.3.2. Создание триггера 11](#_Toc156254187)

[1.4. Представления 12](#_Toc156254188)

[2. Практическая часть 13](#_Toc156254189)

[2.1. Задание 1. 13](#_Toc156254190)

[2.2. Задание 2. 13](#_Toc156254191)

[2.3. Задание 3. 13](#_Toc156254192)

[2.4. Задание 4. 13](#_Toc156254193)

[2.5. Задание 5. 14](#_Toc156254194)

[Контрольные вопросы 14](#_Toc156254195)

[Список использованной литературы 14](#_Toc156254196)

1. Теоретическая часть
   1. Язык программирования PL/pgSQL
      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 наc императивном стиле, когда все команды выполняются последовательно. В качестве решения предлагается использовать процедурный язык для СУБД PostgreSQL - 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* [ NOT NULL ] [ { DEFAULT | := | = } *выражение* ]

Параметр DEFAULT задает начальное значение переменной. По умолчанию, переменным присваивается значение NULL. В случае необходимости, переменную возможно проинициализировать значением, использовав оператор «:=» для присваивания. Параметр CONSTANT указывает, что переменную нельзя изменять, а NOT NULL присваивать ей значение NULL.

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 скрипт 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]](#footnote-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*.

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. 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. 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

WHILE выполняет *операторы* до тех пор, пока истинно *логическое-выражение.* Проверка истинности осуществляется перед каждым входом в цикл.

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. Он называется цикл по результатам запроса.

Для его задания используется следующая конструкция

FOR *цель* IN *запрос* LOOP

*операторы*

END LOOP;

Переменной ***цель*** последовательно присваиваются строки из результата ***запроса*** и для каждой выполняется цикл.

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  оператор;  ELSE  оператор;  END IF |
| Бесконечный цикл | LOOP  оператор; ...  END LOOP; |
| Цикл while | WHILE *логическое-выражение* LOOP  *операторы*  END LOOP |
| Целочисленный цикл | FOR *имя* IN [REVERSE] *выражение* .. *выражение* [BY *выражение*] LOOP  *операторы*  END LOOP; |
| Цикл по результатам запроса | FOR *цель* IN *запрос* LOOP  *операторы*  END LOOP; |

* 1. Хранимые подпрограммы

При выполнении запросов к базе данных происходит пересылка данных по сети от приложения клиента к серверу баз данных и обратно. 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 – С, 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** |
| Использовать транзакции **запрещено** | **Возможно** использовать операции управлением транзакциями |

Рассмотрим их более подробно.

* + 1. Процедуры

Процедуры – подпрограммы, которые не могут возвращать значения. В отличии от обычных запросов процедуры представляют собой объекты, уже готовые к выполнению и не требующие интерпретации. 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 $$

…

$$;

}

После имени процедуры в скобках указывают аргументы. Режим\_аргумента: IN (входной), OUT (выходной), INOUT (входной и выходной) или VARIADIC (переменный). По умолчанию подразумевается IN. Далее следует имя, тип аргумента и, если параметр не задан явно, выражение, используемое для вычисления значения по умолчанию. В теле процедуры указывается язык, на котором она написана. Например, это может быть SQL, C. Команда CREATE OR REPLACE PROCEDURE создает новую или заменяет уже существующую процедуру.

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);

$$;

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. Функции

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

CREATE [ OR REPLACE ] FUNCTION

*имя* ( [ [ *режим\_аргумента* ] [ *имя\_аргумента* ] *режим\_аргумента* [ { DEFAULT | = } *выражение\_по\_умолчанию* ] [, ...] ] )

[ RETURNS *тип\_результата*

| RETURNS TABLE ( *имя\_столбца* *тип\_столбца* [, ...] ) ]

{ LANGUAGE *имя\_языка*

}

Параметры команды схожи с рассмотренными ранее параметрами CREATE PROCEDURE. Команда СREATE OR REPLACE FUNCTION применяется для замены уже существующей функции, однако надо учитывать, что невозможно изменить имя или аргументы функции. При попытке произвести такие изменения возникнет новая, независимая функция. Удобство же использования команды замены состоит в том, что при изменении функции сохраняются все ссылающиеся на нее объекты, тогда как при удалении функции нужно будет удалить вместе с ней и все представления, триггеры, а при задании новой функции воссоздать их заново.

Параметр RETURNS *тип\_результата* задает возвращаемый тип данных*.* Если функция не возвращает значение, то используется слово *void*. Если в качестве *режима\_аргумента* используютOUT или INOUT, параметр RETURNS можно не указывать. Если же он присутствует, то должен согласовываться с *режимом\_аргумента* выходных аргументов.

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 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

Функция с аргументами **составных типов** 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 С. 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. Триггеры

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

В общем случае синтаксис команды создания триггера выглядит следующим образом:

CREATE [ CONSTRAINT ] TRIGGER *имя* { BEFORE | AFTER | INSTEAD OF } { *событие* [ OR ... ] }

ON *имя\_таблицы*

[ FOR [ EACH ] { ROW | STATEMENT } ]

[ WHEN ( *условие* ) ]

EXECUTE PROCEDURE *имя\_функции* ( *аргументы* )

Триггер – подпрограмма, автоматически выполняющаяся при наступлении заданного события. Созданный триггер будет связан с таблицей (*имя\_таблицы)* и будет выполнять заданную функцию (*имя функции* (*аргументы*)) при наступлении определенного события. Допустимые *события:*

INSERT

UPDATE [ OF *имя\_столбца* [, ... ] ]

DELETE

TRUNCATE

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). [2]

Существуют два типа триггеров – построчные и операторные. Построчные триггеры (параметр FOR EACH ROW) вызываются один раз для каждой строки, с которой произошло изменение. Операторный триггер (FOR EACH STATEMENT) срабатывает только один раз, при запуске оператора, независимо от количества строк (даже если таких строк нет).

С помощью параметра CONSTRAINT команда создаёт триггер ограничения. Это обычный триггер, время срабатывания которого можно изменить командой [SET CONSTRAINTS](https://postgrespro.ru/docs/postgrespro/10/sql-set-constraints). [2]

Логическое выражение WHEN ( *условие* ) определяет при каких условиях выполняется функция триггера. В триггерах FOR EACH ROW условие WHEN может ссылаться на значения столбца в старой и/или новой строке, в виде OLD.*имя\_столбца* и NEW.*имя\_столбца*, соответственно. Триггеры INSTEAD OF не поддерживают условия WHEN.

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. Создание 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*, 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

);

С помощью параметра GENERATED ALWAYS AS IDENTITY задается столбец идентификации, последовательность целых чисел, автоматически увеличивающаяся на единицу при добавлении записи.

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. Прикрепления триггерной функции к таблице.

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*.

1. Практическая часть
   1. Задание 1.

Напишите скрипт на языке *PL/pgSQL,* вычисляющий среднюю оценку студента.Аналогичный запрос напишите на языке SQL. Сравните время выполнения работы в обоих случаях. Для расчета времени выполнения скрипта, запустите его в терминале psql, перед этим запустив таймер с помощью команды *\timing*. Для того, чтобы отключить таймер после окончания работы, выполните команду *\timing off.*

* 1. Задание 2.

Напишите SQL запросы к учебной базе данных в соответствии с вариантом. Вариант к практической части выбирается по формуле: V = (N % 10) +1, где N – номер в списке группы, % - остаток от деления.

|  |  |
| --- | --- |
| № варианта | № запросов |
| 1 | 1, 11, 21, 31, 41, 51 |
| 2 | 2, 12, 22, 32, 42, 52 |
| 3 | 3, 13, 23, 33, 43, 53 |
| 4 | 4, 14, 24, 34, 44, 54 |
| 5 | 5, 15, 25, 35, 45, 55 |
| 6 | 6, 16, 26, 36, 46, 56 |
| 7 | 7, 17, 27, 37, 47, 57 |
| 8 | 8, 18, 28, 38, 48, 58 |
| 9 | 9, 19, 29, 39, 49, 59 |
| 10 | 10, 20, 30, 40, 50, 60 |

* 1. Задание 3.

Создайте на основе любых запросов из предыдущих лабораторных работ составьте 2–3 представление. Объясните свой выбор.

Сборник запросов к учебной базе данных

*Скрипты на языке PL/pgSQL*

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, в который возвращает преподавателей со счастливым номером трудового договора (сумма первых трех цифр номера билета совпадает с суммой последних трех)
6. Напишите скрипт, генерирующий случайным образом вариант контрольной работы для каждого студента определенной группы.
7. Напишите скрипт, выводящий ФИО студента и его пол.
8. Напишите скрипт, который считает частоту появления каждого имени студента
9. Напишите скрипт, определяющий, сколько дней рождений приходится на каждый из месяцев
10. Напишите скрипт, шифрующий с помощью шифра Цезаря все названия дисциплин
11. Напишите скрипт, генерирующий случайное местоположение каждого студента на карте (его координаты).
12. Напишите скрипт, который случайным образом разбивает студентов на пары для вальса (мальчик/девочка). Мальчикам, которым не хватило пары в поле партнерши поставить значение «Teddy bear :(»
13. Напишите скрипт, отбирающий случайным образом 10 студентов из списка худших студентов 3-го курса для участия в субботнике
14. Напишите скрипт, вычисляющий количество студентов, у которых совпадает имя и отчество (Например, Сергей Сергеевич)
15. Найти двух студентов, имеющих максимальное совпадение имени, фамилии и отчества

*Процедуры на языке SQL*

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
6. Создайте процедуру удаления преподавателя
7. Создайте процедуру удаления ученого звания
8. Создайте процедуру замены предмета у преподавателя
9. Создайте процедуру изменения почты у студента.
10. Создайте процедуру изменения ЗЕТ у дисциплины
11. Создайте процедуру продления студенческих билетов у определенной группы на 1 год. Номер группы вводится в качестве параметра.
12. Создайте процедуру, выводящую список студентов именинников в ближайшие N дней. N вводится в качестве параметра.
13. Создайте процедуру, выводящую всех преподавателей, преподающих в определенном структурном подразделении на определенной форме обучения.
14. Создайте процедуру, которая случайным образом изменяет оценку на «отл.» у случайного студента по случайной дисциплине.
15. Создайте процедуру, вычисляющую всех преподавателей совместителей, преподающих одновременно на нескольких структурных подразделениях.

*Функции на языке SQL*

1. Создайте функцию, которая выводит зарплату преподавателя по его *id*
2. Создайте функцию, которая выводит количество всех студентов по определенной группе
3. Создайте функцию, которая выводит среднюю оценку по всем предметам у студента (по id студента)
4. Создайте функцию, которая определяет разницу в возрасте между двумя студентами
5. Создайте функцию, рассчитывающую среднюю зарплату преподавателей в определенном структурном подразделении.
6. Создайте функцию, рассчитывающую дисперсию оценки у конкретного студента
7. Создайте функцию, рассчитывающую медиану оценки у конкретного студента
8. Создайте функцию, рассчитывающую медиану оценки у конкретного студента
9. Создайте функцию, возвращающую фамилию случайного студента
10. Создайте функцию, которая будет рассчитывать зарплату преподавателя, по id преподавателя
11. Создайте функцию, которая считает количество преподавателей, в структурном подразделении с номером N. N вводится в качестве параметра.
12. Создайте функцию, возвращающую самое близкую по звучанию фамилию к введенной в качестве параметра. (См. *soundex.* Для использования данной функции необходимо подключить дополнение fuzzystrmatch с помощью команды CREATE EXTENSION fuzzystrmatch*;*)
13. Создайте функцию, вычисляющую средний возраст студента определенной группы
14. Создайте функцию, рассчитывающую средний стаж преподавателей в определенном структурном подразделении.
15. Создайте функцию, вычисляющую число преподавателей на определенном типе подразделения

*Триггеры*

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 “Arе your sure about that?” при добавлении студента John Cena
6. Создайте триггер, выводящий в лог сообщение об изменении оценок у студентов
7. Создайте триггер, который каждые 10 минут пишет в лог сообщение о том, что прошло 10 минут
8. Создайте триггер, проверяющий уникальность руководителя подразделения
9. Создайте триггер, который не позволяет поставить оценку 2 студенту, чье имя совпадает с вашим.
10. Создайте триггер, который после удаления преподавателя выводит сообщение «Преподаватель X Y написал заявление об увольнении по собственному желанию»
11. Создайте триггер, запрещающий ставить студенту больше 4 двоек.
12. Создайте триггер, проверяющий email на вхождение в домен «.ru» - остальные блокируются
13. Создайте триггер, удаляющий студента, получившего 5ю двойку с выводом сообщения в лог «Студент X Y был отчислен за неуспеваемость»
14. Создайте триггер, не позволяющий вводить дубли в таблицу преподаватель (полное совпадение всех строк, кроме номера преподавателя, отличающегося на 1 цифру)
15. Создайте триггер, не позволяющий вводить дубли студентов (полное совпадение всех полей, кроме студенческого билета и ФИО могутотличаться друг от друга на один символ)

Контрольные вопросы

1. В чем отличие между процедурой и функцией?
2. Что такое курсоры?
3. Для каких целей используют триггеры?
4. Возможно ли написать триггерную функцию на языке SQL?
5. Для чего нужны представления?

Список использованной литературы

|  |  |
| --- | --- |
| [1] | «Postgrespro - 40.7. Курсоры,» [В Интернете]. Available: https://postgrespro.ru/docs/postgrespro/10/plpgsql-cursors. |
| [2] | Документация к PostgreSQL 15.1, 2022. |
| [3] | «Исходный код СУБД postgres,» [В Интернете]. Available: https://github.com/postgres/postgres. [Дата обращения: 30 01 2023]. |
| [4] | Е. Рогов, PostgreSQL изнутри, 1-е ред., Москва: ДМК Пресс, 2023, p. 662 . |
| [5] | Б. А. Новиков, Е. А. Горшкова и Н. Г. Графеева, Основы технологии баз данных, 2-е ред., Москва: ДМК пресс, 2020, p. 582. |
| [6] | Е. П. Моргунов, PostgreSQL. Основы языка SQL, 1-е ред., Санкт-Петербург: БХВ-Петербург, 2018, p. 336. |

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, [↑](#footnote-ref-1)