**Лабораторные работы по курсу**

**Базы данных**

**Лабораторная работа 6**

**«Язык программирования PL/pgSQL.**

**Процедуры, функции, триггеры»**

**Москва, 2024**

Оглавление

[1. Теоретическая часть 3](#_Toc165221524)

[1.1. Язык программирования PL/pgSQL 3](#_Toc165221525)

[1.1.1. Введение в PL/pgSQL. 3](#_Toc165221526)

[1.1.2. Оператор ветвления 5](#_Toc165221527)

[1.1.3. Операторы циклов 6](#_Toc165221528)

[1.2. Хранимые подпрограммы 8](#_Toc165221529)

[1.2.1. Процедуры 8](#_Toc165221530)

[1.2.2. Функции 9](#_Toc165221531)

[1.3. Триггеры 11](#_Toc165221532)

[1.3.1. Создание триггерной функции 12](#_Toc165221533)

[1.3.2. Прикрепления триггерной функции к таблице. 13](#_Toc165221534)

[2. Практическая часть 13](#_Toc165221535)

[2.1. Задание 1. 13](#_Toc165221536)

[2.2. Задание 2. 13](#_Toc165221537)

[2.3. Задание 3. 17](#_Toc165221538)

[Контрольные вопросы 17](#_Toc165221539)

[Список использованной литературы 17](#_Toc165221540)

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 нельзя 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 students;

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

stud record;

BEGIN

SELECT last\_name, first\_name

INTO stud

FROM students

WHERE student\_id = 866017;

raise notice 'Имя: %, Фамилия: %', stud.first\_name, stud.last\_name;

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

stud record;

studentid constant INTEGER := 866017;

BEGIN

SELECT last\_name, first\_name

INTO stud

FROM students

WHERE student\_id = studentid;

raise notice 'Имя: %, Фамилия: %', stud.first\_name, stud.last\_name;

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 успешно выполненного запроса и false, если было возвращено пустое множество.

do

$$

DECLARE

stud record;

studentid constant INTEGER := 8660107;

BEGIN

SELECT last\_name, first\_name

INTO stud

FROM students

WHERE student\_id = studentid;

if not found then

raise notice'The student % could not be found', studentid;

else

raise notice 'Имя: %, Фамилия: %', stud.first\_name, stud.last\_name;

end if;

END

$$;

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

NOTICE: The student 8660107 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**

**stud record;**

**student\_name constant TEXT := 'Александр';**

**BEGIN**

**FOR stud IN SELECT last\_name, first\_name**

**FROM students**

**WHERE first\_name = student\_name**

**loop**

**raise notice 'Имя: %, Фамилия: %', stud.first\_name, stud.last\_name;**

**end loop;**

**END**

**$$;**

NOTICE: Имя: Александр, Фамилия: Лобов

NOTICE: Имя: Александр, Фамилия: Погорельцев

NOTICE: Имя: Александр, Фамилия: Вершинин

NOTICE: Имя: Александр, Фамилия: Заметов

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 *stud* 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, обновляющую значение ЗЕТ у определенной дисциплины.

CREATE PROCEDURE update\_zet(param\_field\_id uuid, new\_zet integer)

LANGUAGE SQL

AS

$$

UPDATE fields SET zet = new\_zet

WHERE field\_id = param\_field\_id

$$;

Для того, чтобы вызвать процедуру необходимо использовать ключевое слово CALL.

CALL update\_zet('f81d63d6-ccd0-4cf0-a13a-340c44b852af', 5)

Для изменения процедуры возможно использовать ключевое слово REPLACE вместо CREATE, для удаления – DROP.

DROP PROCEDURE update\_zet;

Аналогично 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 students VALUES (Student\_ID, F, I, O, groupe, birthday, email);**

**INSERT INTO student\_ids VALUES (Student\_ID, datebegin, dateend);**

**$$;**

* + 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 используют OUT 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\_ids

WHERE student\_id = Student\_ID

$$;

Вызов скалярной функции происходит с помощью запроса SELECT:

SELECT find\_date(841576)

find\_date

------------

2027-08-31

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(

last\_name varchar(30),

first\_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 last\_name, first\_name, expiration\_date

FROM student\_ids

INNER JOIN students ON students.student\_id = student\_ids.student\_id

WHERE expiration\_date < id\_date

$$;

Вызов функции будет совпадать с обращением к таблице:

SELECT \* FROM find\_id\_date('10/09/2024');

last\_name | first\_name | id\_date

-----------+------------+------------

Корзухина | Серафима | 2024-08-31

Голубков | Сергей | 2024-08-31

Чарнота | Григорий | 2024-08-31

Хлудов | Роман | 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). Обратите внимание, что INSTEAD OF работает только в представлениях. 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.last\_name <> OLD.last\_name THEN

INSERT INTO sys\_log(info)

VALUES(concat(OLD.last\_name, ' сменил фамилию на ',NEW.last\_name));

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 students

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, 61 |
| 2 | 2, 12, 22, 32, 42, 52, 62 |
| 3 | 3, 13, 23, 33, 43, 53, 63 |
| 4 | 4, 14, 24, 34, 44, 54, 64 |
| 5 | 5, 15, 25, 35, 45, 55, 65 |
| 6 | 6, 16, 26, 36, 46, 56, 66 |
| 7 | 7, 17, 27, 37, 47, 57, 67 |
| 8 | 8, 18, 28, 38, 48, 58, 68 |
| 9 | 9, 19, 29, 39, 49, 59, 69 |
| 10 | 10, 20, 30, 40, 50, 60, 70 |

Сборник запросов к учебной базе данных

*Скрипты на языке PL/pgSQL*

1. Напишите скрипт, выводящий количество всех оценок 5, 4, 3, 2
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, который возвращает фамилию, имя студентов со счастливым студенческим билетом (сумма первых трех цифр номера билета совпадает с суммой последних трех)
5. Напишите скрипт, генерирующий случайным образом вариант контрольной работы для каждого студента определенной группы. Первая цифра варианта – сумма цифр номера группы, остальные две выбираются рандомно.
6. Напишите скрипт, выводящий ФИО студента и его пол.
7. Напишите скрипт, который считает частоту появления каждого имени студента
8. Напишите скрипт, определяющий, сколько дней рождений приходится на каждый из месяцев
9. Напишите скрипт определяющий знак зодиака каждого студента. Выведите Фамилию, имя, дату рождения и знак зодиака.
10. Напишите скрипт, вычисляющий количество студентов, у которых совпадает имя и отчество (Например, Сергей Сергеевич).
11. Напишите скрипт, шифрующий с помощью шифра Цезаря все названия дисциплин.
12. Напишите скрипт, генерирующий случайное местоположение каждого студента на карте (его координаты в градусах).
13. 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 :(»
14. Напишите скрипт, отбирающий случайным образом 10 студентов из списка студентов 3-го курса, имеющих больше 10 двоек для участия в субботнике. Выведите фамилию имя и количество двоек у студента.
15. Найти двух студентов, имеющих максимальное совпадение имени, фамилии и отчества.
16. Напишите скрипт, генерирующий случайное местоположение каждого студента на карте (его координаты в градусах). Определите материк, соответствующий этим координатам. Считайте, попаданием на материк, если координаты лежат в пределах:

Европа – между 1° с.ш. и 77° с.ш. и между 9° з.д. и 67° в.д.

Африка – между 37° с.ш. и 34° ю.ш. и между 13° з.д. и 51° в.д.

Австралия – между 10° ю.ш. и 39° ю.ш. и между 113° в.д. и 153° в.д.

Северная Америка – между 7° с.ш. и 71° с.ш. и между 55° з.д. и 168° з.д.

Южная Америка – между 12° с.ш. и 53° ю.ш. и между 34° з.д. и 81° з.д.

Антарктида– ниже 63° ю.ш.

1. Напишите скрипт, выводящий фамилию, имя, дату рождения и знак зодиака студентов совместимых с вами по знаку зодиака. Совместимость найти в любом гороскопе.
2. Напишите скрипт имитирующий шахматный турнир. Выберите случайным образом 10 студентов. Каждый играет с каждым по одному разу. Результат игры между двумя участниками выбирается рандомно из победы (победивший получает 2 очка, проигравший 0) и ничьи (каждому добавляется по одному баллу). Вывести результат каждой игры (фамилии участников и результат игры) и итоговую таблицу всех участников с суммой полученных баллов в порядке убывания очков.
3. Напишите скрипт имитирующий шахматный турнир. Выберите случайным образом 10 студентов. Каждый играет с каждым по одному разу. Результат игры между двумя участниками выбирается рандомно из победы (победивший получает 2 очка, проигравший 0) и ничьи (каждому добавляется по одному баллу). После проведения первого тура отсеиваются 2 участника набравшие наименьшее количество очков. Проводится 2 тур между оставшимися 8 студентами. И так продолжать до тех пор, пока не останутся два победителя. Вывести результаты каждой игры каждого тура (фамилии участников и результат игры) и также для каждого тура итоговые таблицы всех участников с суммой полученных баллов в порядке убывания очков.
4. Написать скрипт, имитирующий пересдачу экзамена студентами. Для одного (любого) преподавателя выбрать студентов, имеющих долги дисциплинам, которые он ведет. “Пересдача” будет состоять в сравнении суммы цифр id преподавателя и студента (если у студента сумма больше – пересдал, иначе нет). Суммирование проводить до тех пор, пока не останется одна цифра (898899 -> 51 ->6). В начале списка вывести фамилию, имя, id преподавателя. Далее вывести id студентов, информацию о пересдаче и суммы цифр id студента и преподавателя.

*Процедуры на языке SQL*

1. Создайте процедуру перемещения студента из одной группы в другую
2. Создайте процедуру изменения курса в определенной группе. Входные параметры – курс, номер группы.
3. Создайте процедуру увеличения зарплаты преподавателя в зависимости от стажа. Стаж до 10 лет – увеличиваем на **a**, от 11 до 20 – на **2a**, более на **3a** рублей. Где **a** входной параметр процедуры
4. Создайте процедуру, изменяющую преподавателя у данной дисциплины. Входные параметры – id нового преподавателя, название дисциплины).
5. Создайте процедуру изменяющую оценку у студента по определенной дисциплине. Входные параметры – id студента, название дисциплины, новая оценка.
6. Создайте процедуру увеличивающую на один бал оценку студентов именинников по определенной дисциплине. Входной параметр - название дисциплины
7. Создайте процедуру изменения почты у студента.
8. Создайте процедуру продления студенческих билетов у определенной группы на 1 год. Номер группы вводится в качестве параметра.
9. Создайте процедуру продления студенческих билетов у определенной группы на 1 год. Входной параметр - номер группы.
10. Создайте процедуру изменения ставки у преподавателя. Учтите ограничения – 0 < ставка <= 2, содержит не более двух цифр после запятой и последняя цифра либо 0, либо 5. . В случае если входной параметр не удовлетворяет диапазону – ставку не менять, последняя цифра не 0, и не 5 – округлить по правилам. Входной параметр – id преподавателя, ставка.

*Функции на языке SQL*

1. Создайте функцию, которая выводит количество всех студентов по определенной группе
2. Создайте функцию, которая выводит среднюю оценку по всем предметам у студента (по id студента)
3. Создайте функцию, которая определяет разницу в возрасте между двумя студентами
4. Создайте функцию, рассчитывающую среднюю зарплату преподавателей в определенном структурном подразделении.
5. Создайте функцию, рассчитывающую дисперсию оценки у конкретного студента
6. Создайте функцию, рассчитывающую моду оценки у конкретного студента
7. Создайте функцию, возвращающую фамилию случайного студента
8. Создайте функцию, которая будет рассчитывать зарплату преподавателя, по id преподавателя
9. Создайте функцию, которая считает количество преподавателей, в структурном подразделении с номером N. N вводится в качестве параметра.
10. Создайте функцию, вычисляющую средний возраст студентов определенной группы
11. 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*;*)
12. Создайте функцию, вычисляющую средний возраст студентов для каждой группы.
13. Создайте функцию, рассчитывающую среднюю зарплату преподавателей в каждом структурном подразделении
14. Создайте функцию, выводящую всех однофамильцевопределенного студента. Выведите девушек и юношей с аналогичной фамилией, а также их группу.
15. Создайте функцию, рассчитывающую средний стаж преподавателей в каждом структурном подразделении.
16. Создайте функцию, вычисляющую число преподавателей в каждом структурном подразделении.
17. Создайте функцию, выводящую всех студентов именинников в определенном диапазоне. Входные параметры – дата рождения и количество дней диапазона (плюс, минус).
18. Создайте функцию, которая выведет студентов, у которых количество двоек превышает заданное значение.
19. Создайте функцию, выводящую всех преподавателей, преподающих в определенном структурном подразделении.
20. Создайте функцию, выводящую фамилию определенного преподавателя и название всех дисциплин, которые он читает в определенном семестре. Входные данные – id преподавателя, семестр.

*Триггеры*

1. Создайте триггер запрещающий понижать оценки студентов.
2. Создайте триггер, который запрещает добавление или изменение дня рождения студента, возраст которого превышает 100 лет.
3. Создайте триггер, который запрещает добавление и изменение даты выдачи студенческого билета, у которого дата выдачи> даты по которую он действителен.
4. Создайте триггер, который запрещает изменение в структурных подразделениях
5. Создайте триггер, который выводит сообщение “Arе your sure about that?” при добавлении студента John Cena.
6. Создайте триггер, выводящий сообщение об изменении оценок у студентов. Сообщение должно содержать id студента, старую и новую оценку.
7. Создайте триггер, выводящий сообщение об изменении оценок у студентов. Сообщение должно содержать фамилию студента, предмет, старую и новую оценку.
8. Создайте триггер, который не позволяет поставить оценку 2 студенту, чье имя совпадает с вашим.
9. Создайте триггер, запрещающий ставить студенту больше 4 двоек.
10. Создайте триггер, блокирующий изменение и добавление email, у которых в домене отсутствует «.ru».
11. Преподаватель в течение нескольких лет проводил занятия в командировке в другой стране. Каждый год работы зарплата повышалась на 10%. Создайте триггер, который при увеличении стажа на время работы в командировке увеличивает зарплату преподавателя.
12. Создайте триггер, не позволяющий вводить дубли в таблицу преподавателей (совпадение фамилии, имени и отчества с точностью до двух букв в каждом). Используйте функцию *levenshtein().*
13. Создайте триггер, не позволяющий вводить дубли в таблицу студентов (совпадение фамилии, имени и отчества с точностью до двух букв в каждом). В предупреждении выведите ФИО студента с похожими данными. Используйте функцию *levenshtein().*
14. Создайте триггер, сохраняющий информацию о изменении зарплаты преподавателей и дату изменения.
15. Создайте триггер, который после удаления преподавателя сохраняет информацию об этом в таблицу (ФИО и текущую дату).
16. Создать триггер, который при уменьшении заработной платы преподавателя до 0 удаляет запись о нем.
17. Создать триггер, который при выставлении пятой и более двойки удаляет студента из базы. Запись об удалении заносится в информационную таблицу.
18. Создайте триггер, который запрещает ввод латинских букв в фамилии, имени и отчестве и выводит соответствующее предупреждение.
19. Создайте триггер, сохраняющий информацию об изменениях оценок у студентов.
20. Создайте триггер, проверяющий уникальность руководителя подразделения. Учтите, что фамилия, имя, отчество руководителя в поле *head\_of\_the\_unit* могут быть переставлены. Используйте функции *position, regexp\_split\_to\_array.*
    1. Задание 3.

Для добавленной в 4-й лабораторной работе таблицы создайте любой триггер.

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)