**Лабораторная работа №3**

Пользовательские функции. Процедурный язык PL/pgSQL.

# Цели лабораторной работы:

* Познакомиться с созданием пользовательских функций и триггеров в PostgreSQL;
* Освоить работу с составными типами данных и массивами;
* Изучить основы работы с процедурным языком PL/pgSQL.

# Задание.

При выполнении работы нужно использовать БД, созданную в лабораторных работах №1 и №2. Нужно выполнить следующие шаги.

1. Для любой таблицы создать функцию save\_имя\_таблицы, которая принимает на вход параметры, соответствующие её столбцам, и, если переданное значение первичного ключа равно null, выполняет запрос insert, иначе – запрос update для соответствующей строки. Функция должна вернуть значение первичного ключа вставленной или изменённой строки.

2. Для любой таблицы, на которую имеются внешние ключи, создать функцию delete\_имя\_таблицы, принимающую на вход значение первичного ключа строки и ничего не возвращающую. Если на удаляемую строку существуют ссылки, то функция должна поднимать ошибку «Невозможно выполнить удаление, так как есть внешние ссылки».

3. Для таблицы, содержащей столбец с числовыми значениям, создать функцию, которая принимает на вход число – минимальное значение – и возвращает setof имя\_таблицы – множество строк, в которых значение числа больше или равно переданному аргументу.

4. Создать составной тип, содержащий не менее 2-3 полей, по крайней мере одно из которых должно быть числовым. Создать функцию, которая принимает массив объектов этого типа и минимальное значение для указанного поля. Функция должна возвращать массив отфильтрованных по переданному значению объектов.

5. Для любой таблицы создать таблицу log\_имя\_таблицы, которая будет содержать лог изменений по любому выбранному столбцу этой таблицы. Для этого нужны столбцы:

* первичный ключ;
* внешний ключ на выбранную таблицу;
* дата изменения строки;
* старое значение столбца;
* новое значение столбца.

Реализовать заполнение таблицы с логом с помощью триггеров на вставку/изменение строк.

6. Реализовать любую функцию на свой выбор, использующую для получения результата динамически формируемый запрос.

Все функции должны быть реализованы на PL/pgSQL. Отчет по лабораторной работе должен содержать код создания перечисленных функций, составного типа из шага 4 и таблицы из шага 5, а также демонстрацию работы созданных функций.

**NOTE:** Если вы уже знакомы с теорией и хотите узнать детали выполнения задания, вы можете сразу перейти к разделу Выполнение задания.

# Написание первой функции.

Функции в PostgreSQL объявляются следующим образом:

**CREATE OR REPLACE FUNCTION** *название\_функции* (*параметр\_n* ***тип параметра\_n***)  
**RETURNS** *тип\_возвращаемого\_значения*  
**AS $$  
 /\* Ваш код \*/  
$$ LANGUAGE** *язык\_кода*;

При вызове данной команды в БД создастся функция, которую можно будет, в последующем, вызывать в запросах.

Параметры должны объявляться по следующему правилу: имя параметра, затем через пробел идет тип параметра. Параметры разделяются запятой.

Функция может возвращать:

* Значения базовых типов;
* Значения составных типов (об этих типах будет рассказано далее);
* Таблицы (для этого нужно вместо RETURNS указать RETURNS TABLE).

Если функция не должна ничего возвращать, то в *тип\_возвращаемого\_значения* указать VOID.

$$ означает скобки, ограничивающие ваш код. Такого вида скобки необходимы, так как команды в программе и запросы в SQL завершаются точкой с запятой. То есть введенная вами точка с запятой между скобками $$ будет однозначно понята интерпретатором как конец команды.

Можно использовать уникальные скобки, указав между долларами текст (Например: $ivt$ … $ivt$).

В скобках пишется код на том языке, который вы указали в *язык\_кода*. Для данной лабораторной работы будут использоваться SQL и PL/PgSQL.

Приведем пример:

**CREATE OR REPLACE FUNCTION** test\_sum(arg1 **NUMERIC**, arg2 **NUMERIC**)  
**RETURNS NUMERIC  
AS $$  
 SELECT** arg1 + arg2;  
**$$ LANGUAGE SQL**;

Данная функция принимает два числовых параметра и возвращает числовое значение их суммы.

**NOTE:** Подробнее о синтаксисе создания функций можно узнать в официальной документации:

<https://postgrespro.ru/docs/postgresql/9.6/sql-createfunction.html>

# Процедурный язык PL/pgSQL.

На практике, для решения определенных задач одного языка SQL бывает недостаточно. Здесь на помощь приходят процедурные языки. Например, в Oracle DB используется язык PL/SQL (Procedural Language). В PostgreSQL используется свой аналог – PL/pgSQL. В некотором роде PL/pgSQL схож с языком Pascal.

**NOTE:** В данном методическом пособии будет рассказано только о базовом синтаксисе PL/pgSQL. Для более глубокого изучения следует обратиться к официальной документации:

[https://postgrespro.ru/docs/postgrespro/9.6/PL/pgSQL](https://postgrespro.ru/docs/postgrespro/9.6/plpgsql)

## Написание «Hello world».

Ниже приведена Hello world программа на процедурном языке:

**CREATE OR REPLACE FUNCTION** *say\_hello* ()  
**RETURNS VOID  
AS $$  
DECLARE** greeting **VARCHAR**;  
**BEGIN** greeting := **'Hello world'**;  
 **RAISE NOTICE '%'**, greeting;  
**END**;  
**$$ LANGUAGE** plpgsql;

Стоит заметить, что появилась секция DECLARE, оно является необходимым только тогда, когда нам нужно объявить переменные, в остальных случаях его использовать необязательно. RAISE позволяет управлять выводом сообщений в консоль, либо же вызов исключений (о них будет рассказано далее.).

На рисунке 1 представлен результат работы вызова функции:



Рисунок 1 – Консольный вывод функции.

## Основные конструкции языка.

### Запросы.

Мы можем писать SQL запросы прямо в коде. Результат запроса можно:

* Возвращать из функции (с помощью RETURN QUERY);
* Записывать его в переменную (с помощью INTO).

Данная функция возвращает в качестве результата запроса таблицу group:

**CREATE OR REPLACE FUNCTION** *foo* ()  
**RETURNS SETOF** subject  
**AS $$  
BEGIN  
 RETURN QUERY** (**SELECT** *\** **FROM** subject);  
**END**;  
**$$ LANGUAGE** plpgsql;

На рисунке 2 представлен результат работы функции:

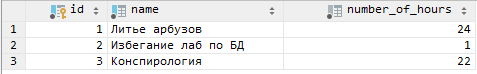


Рисунок 2 – Вывод результата запроса.

А эта функция записывает таблицу в переменную и выводит наименование первой дисциплины на экран:

**CREATE OR REPLACE FUNCTION** *foo* ()  
**RETURNS VOID  
AS $$  
DECLARE** tab subject;  
**BEGIN  
 SELECT** *\** **INTO** tab **FROM** subject;  
  
 **RAISE NOTICE 'Наименование: %'**, tab.**name**;  
**END**;  
**$$ LANGUAGE** plpgsql;

Результат работы представлен на рисунке 3:



Рисунок 3 – Вывод значения переменной.

### Переменные.

В этом разделе необходимо обобщить сказанную ранее информацию о переменных.

Переменные объявляются в секции DECLARE. Базовый синтаксис:

*имя\_переменной* ***тип\_переменной***;

Переменным в коде можно присвоить значение либо через оператор присваивания :=, либо при помощи конструкции запроса INTO (об этом было сказано ранее).

**Условия.**

В Pl/pgSQL поддерживается 2 вида условных операторов IF и CASE.

3 вида IF:

* IF ... THEN ... END IF;

**IF** *логическое\_условие* **THEN** */ \*операторы \*/***END IF**;

* IF ... THEN ... ELSE ... END IF;

**IF** *логическое\_условие* **THEN** */\* операторы \*/*

**ELSE**

*/\* операторы \*/***END IF**;

* IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF.

**IF** *логическое\_условие\_1* **THEN** */\* операторы \*/***ELSIF** *логическое\_условие\_2* **THEN** */\* операторы \*/***ELSE** */\* операторы \*/***END IF**;

2 вида CASE:

* CASE ... WHEN ... THEN ... ELSE ... END CASE;

**CASE** *параметр*  
 **WHEN** *значение* **THEN** */\* операторы \*/*  
 **ELSE**  
 */\* операторы \*/***END CASE**;

* CASE WHEN ... THEN ... ELSE ... END CASE.

**CASE  
 WHEN *логическое\_условие* THEN** */\* операторы \*/* **ELSE** */\* операторы \*/***END CASE**;

### Массивы.

Как и другие языки программирования, PL/pgSQL предоставляет возможность работы с массивами.

Для того чтобы создать массив, необходимо в объявлении переменной к типу данных дописать [].

arr **INT**[];  
matr **INT**[][];

Чтобы присвоить массиву какое-либо значение можно использовать следующую конструкцию:

arr := **ARRAY**[1, 2, 3];

Чтобы обратиться к элементу массива достаточно указать индекс элемента в [].

arr[2] := 10;

Также возможно управлять массивами с помощью специальных функций. Например дополнять, объединять, преобразовывать в строки и т.д.. Функций очень много, все они [описаны в официальной документации](https://postgrespro.ru/docs/postgrespro/9.5/functions-array).

**NOTE:** Подробнее о массивах можно прочитать в официальной документации: <https://postgrespro.ru/docs/postgresql/9.6/arrays>

### Циклы.

В PL/pgSQL бывают простые цикла, а также циклы FOR и WHILE.

Простые (или бесконечные) циклы будут выполняться до тех пор, пока не будет прекращен операторами EXIT или RETURN. Помимо EXIT также можно использовать CONTINUE, если вы захотите пропустить одну итерацию цикла. Например:

**LOOP  
 EXIT WHEN** param < 0;  
**END LOOP**;

Цикл FOR может быть:

* Целочисленным;
* По результатам SQL запроса;
* По элементам массива.

Пример целочисленного цикла:

**FOR** i **IN** 1..10 **LOOP** */\* операторы \*/***END LOOP**;

Пример цикла по результатам запроса. Данный цикл выведет в консоль название всех дисциплин, находящихся в таблице subject:

**FOR** sub **IN** (**SELECT** *\** **FROM** subject) **LOOP  
 raise notice 'Name: %'**, sub.**name**;  
**END LOOP**;

Пример цикла по элементам массива. Данный цикл выведет все значения массива в консоль:

**FOREACH** i **IN ARRAY** arr **LOOP  
 RAISE NOTICE '%'**, i;  
**END LOOP**;

**NOTE:** Все переменные, используемые в циклах должны быть объявлены в поле DECLARE.

Пример цикла WHILE. Данный цикл будет выводить значение переменной, пока та не достигнет значения 10:

**WHILE** i < 10 **LOOP  
 RAISE NOTICE '%'**, i;  
**END LOOP**;

### Исключения.

Любая ошибка при исполнении скрипта, либо пользовательский вызов ошибки прерывает выполнение функции, а также транзакции, относящейся к этой функции.

Использование в блоке нашего кода секции EXCEPTION позволяет перехватывать и обрабатывать исключения. Например:

**CREATE OR REPLACE FUNCTION** *foo* ()  
**RETURNS VOID  
AS $$  
BEGIN** */\* Делаем что-нибудь непоправимое \*/***EXCEPTION  
 WHEN** *код\_ошибки* **THEN  
 RAISE NOTICE 'Я поймал ошибку!'**;  
 **RETURN** 0;  
**END**;  
**$$ LANGUAGE** plpgsql;

В *код\_ошибки* желательно указывать символьный код ошибки, так будет проще понимать, какое исключение перехватывает данный обработчик.

**NOTE:** Таблицу с кодами ошибок можно найти в официальной документации: <https://postgrespro.ru/docs/postgrespro/9.6/errcodes-appendix>

Если вы хотите вызвать свое исключение, то можно использовать оператор RAISE EXCEPTION:

**CREATE OR REPLACE FUNCTION** *foo* ()  
**RETURNS VOID  
AS $$  
BEGIN  
 RAISE EXCEPTION 'Хозяин говорит, что здесь надо ругаться.'**;  
**END**;  
**$$ LANGUAGE** plpgsql;

Результат работы функции на рисунке 4:



Рисунок 4 – Вызов исключения.

### Составные типы данных.

Составные типы данных позволяют создавать свои типы данных, состоящие из именованного списка полей и соответствующих им типов данных.

Синтаксис создания составных типов схож с синтаксисом создания таблиц:

**CREATE TYPE** *имя\_типа* **AS** (  
 *поле\_1* **тип\_поля\_1**,

*поле\_2* **тип\_поля\_2**  
);

Чтобы обратиться к полю составного типа данных, достаточно дописать к названию переменной данного типа точку и имя поля:

subject\_var.id

Возможно обращение к составным типам данных в SQL запросам как к таблицам (создание выборки, обновление данных и т.д.):

**SELECT** *\** **FROM** *имя\_типа*;

**NOTE:** Подробнее о составных типах данных можно узнать в официальной документации: <https://postgrespro.ru/docs/postgrespro/9.5/rowtypes>

### Триггеры и триггерные функции.

Иногда требуется, чтобы до или после изменений в таблице происходили какие-то нужные нам действия. Например, проверка корректности данных, отмена запроса, фиксация факта изменения данных и т.д.. Для этого в Postgres существует механизм триггеров и триггерных функций.

Создание такого обработчика производится в два этапа: создание триггерной функции и создание самого триггера.

Триггерная функция создается как обычная функция, но должна возвращать тип TRIGGER. Эта функция содержит логику, которую задает разработчик. Ниже представлен пример создания триггерной функции:

**CREATE OR REPLACE FUNCTION** *trigger\_func* ()  
**RETURNS TRIGGER** */\* обратите внимание на возвращаемый тип \*/***AS $$  
BEGIN**

*/\* ваш код \*/*

**RETURN NEW;  
END**;  
**$$ LANGUAGE** plpgsql;

Триггерная функция должна вернуть либо строку, которая находится в локальной переменной (чаще всего NEW), либо NULL.

**NOTE:** Если триггер срабатывает до изменений в таблице, а функция возвращает NULL, то изменения в таблице отменяются. Это бывает удобно, когда нам нужно проверять корректность пришедших данных.

При изменении данных в таблице срабатывают триггеры, которые назначены на эту таблицу. Каждый триггер вызывает свою триггерную функцию и передает в нее локальные переменные. Например:

* **NEW.** Переменная содержит новую строку базы данных для команд INSERT/UPDATE в триггерах уровня строки. В триггерах уровня оператора и для команды DELETE этой переменной значение не присваивается.
* **OLD.** Переменная содержит старую строку базы данных для команд UPDATE/DELETE в триггерах уровня строки. В триггерах уровня оператора и для команды INSERT этой переменной значение не присваивается.
* **TG\_OP.** Строка, содержащая INSERT, UPDATE, DELETE или TRUNCATE, в зависимости от того, для какой операции сработал триггер.
* И т. д.

Создать триггер можно следующим образом:

**CREATE TRIGGER** *имя\_триггера*  
***время\_срабатывания операции\_срабатывания*  
ON** *имя\_таблицы*  
**тип\_срабатывания  
EXECUTE PROCEDURE** *название\_триггерной\_функции*();

Поле *время\_срабатывания* определяет до (BEFORE) или после (AFTER) изменений будет срабатывать триггер.

Поле *операции\_срабатывания* определяет операции, при которых срабатывает триггер. Триггер может срабатывать на INSERT, UPDATE, DELETE или TRUNCATE. Если операций несколько, то их необходимо разделять оператором OR.

Поле *имя\_таблицы* определяет, на какую таблицу «подписывается» триггер.

Поле *тип\_срабатывания* определяет, будет ли процедура триггера срабатывать один раз для каждой строки, либо для SQL-оператора. Если не указано ничего, подразумевается FOR EACH STATEMENT (для оператора). Для триггеров ограничений можно указать только FOR EACH ROW.

Поле *название\_триггерной\_процедуры* определяет, какую процедуру будет вызывать триггер. Вызываемая процедура должна возвращать тип TRIGGER.

Приведем пример:

**CREATE TRIGGER** trigger\_func()  
**BEFORE UPDATE OR DELETE  
ON** subject  
**FOR EACH ROW  
EXECUTE PROCEDURE** *trigger\_func*();

Данный триггер срабатывает до операции обновления или удаления строки из таблицы subject. Срабатывание происходит для каждой строки. При срабатывании вызывается функция trigger\_func.

Чтобы удалить триггер, необходимо помимо DROP TRIGGER написать еще имя таблицы, на которую подписан этот триггер:

**DROP TRIGGER** trig **ON** tab;

**NOTE:** Подробнее о триггерах и триггерных функциях можно узнать в официальной документации: [https://postgrespro.ru/docs/postgresql/9.6/PL/pgSQL-trigger](https://postgrespro.ru/docs/postgresql/9.6/plpgsql-trigger). Также в документации можно подробнее узнать о синтаксисе создания триггеров: <https://postgrespro.ru/docs/postgrespro/10/sql-createtrigger>

### Динамически формируемые запросы.

Иногда при решении некоторых задач заранее неизвестно, какой запрос необходимо выполнить. Например, если нужно выполнить одно и то же действие для таблиц с однотипной структурой, вместо написания нескольких функций с почти одинаковым кодом можно написать одну, принимающую на вход имя таблицы.

Для этого в PL/pgSQL поддерживаются динамически формируемые запросы. С помощью команды EXECUTE можно выполнить описанный строкой запрос, передавая ему различные параметры и возвращая результат в указанные переменные.

Конкатенацию строк можно производить с помощью ||. Ниже представлен пример использования EXECUTE:

**EXECUTE 'SELECT '**||*имя\_столбца*||**'::text FROM '**||*имя\_таблицы*||**' WHERE id =  
$1' INTO** retVal **USING** id;

Используя динамические запросы, мы как бы «встраиваем» переменные в строку запроса.

Стоит заметить, оператор USING подставляет значение переменной id вместо «$1».

Результат запроса отправляется в переменную retVal.

**NOTE:** Подробнее динамически формируемых запросах можно узнать в официальной документации: <https://postgrespro.ru/docs/postgresql/9.6/plpgsql-statements.html#plpgsql-statements-executing-dyn>

# Выполнение задания.

## Задание 1. Функция save\_subject.

Данная функция должна принимать на вход:

* ID дисциплины.
* Наименование дисциплины.
* Количество часов.

Если в параметре ID дисциплины указан NULL, то функция должна создать новую запись в таблице, если указан – обновить существующую запись. В случае если мы обновим запись с ID, не существующем на данный момент в таблице, ничего не произойдет.

Функция будет возвращать значение id строки, с которой была произведена работа.

Далее представлен код разработанной функции:

**CREATE OR REPLACE FUNCTION** *save\_subject* (  
 \_id **BIGINT**,   
 \_name **VARCHAR**(50),   
 \_num\_of\_hours **INT**)  
**RETURNS BIGINT  
AS $$  
DECLARE** used\_id **BIGINT**;

**BEGIN  
 IF** \_id **IS NULL THEN  
  
 INSERT INTO** subject (**name**, **number\_of\_hours**)  
 **VALUES** (\_name, \_num\_of\_hours)  
 **RETURNING id** */\* Конструкция позволяет вернуть id нового элемента \*/* **INTO** used\_id; */\* id нового элемента записывается в переменную used\_id \*/* **ELSE  
  
 UPDATE** subject **SET  
 name** = \_name,  
 **number\_of\_hours** = \_num\_of\_hours  
 **WHERE id** = \_id;  
  
 used\_id := \_id; */\* Нам уже известен id, поэтому просто присвоим его \*/* **END IF**;  
  
 **RETURN** used\_id;  
**END**;  
**$$ LANGUAGE** plpgsql;

Стоит обратить внимание на запрос INSERT в нем присутствует ранее не описанное свойство RETURNING, которое позволяет возвращать поля нового элемента.

**NOTE:** В разных СУБД возврат id новой строки реализован по-разному. Например, MySQL не существует свойства RETURNING, там необходимо вызывать функцию **LAST\_INSERT\_ID()**.

Перед именами параметров был добавлен символ нижнего подчеркивания, чтобы избежать неоднозначности при использовании переменных.

Проверим работу функции. Для начала, посмотрим, как выглядит таблица до изменения (Рисунок 5):

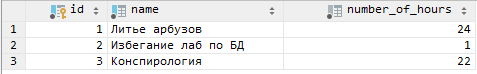


Рисунок 5 – Таблица subject до изменений.

Теперь выполним запрос:

**SELECT** *save\_subject*(**null**, **'Искусство лжи и обмана'**, 10);

На рисунке 6 представлен вывод ID новой строки:



Рисунок 6 – ID новой строки.

Затем выполним запрос:

**SELECT** *save\_subject*(3, **'Двоичная логика на редстоуне'**, 4);

На рисунке 7 представлен вывод ID новой строки:



Рисунок 7 – ID измененной строки.

Проверим таблицу (Рисунок 8). Все верно, создана строка с индексом 4, а содержимое строки с индексом 3 изменено.

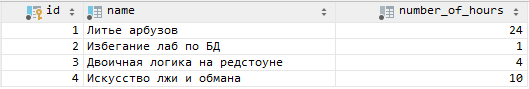


Рисунок 8 – Таблица subject после изменений.

## Задание 2. Функция delete\_subject.

Данная функция будет на вход принимать ID строки, которую нужно будет удалить из таблицы.

Если такого ID нет в таблице, то ничего не произойдет, скрипт отработает без ошибок.

Если в других таблицах строки ссылаются на удаляемую нами строку, мы должны выдать ошибку с текстом: «Невозможно выполнить удаление, так как есть внешние ссылки.».

Саму проверку на наличие внешних ключей производить не стоит. Так как если попытаться удалить строку, на которую присутствует внешняя ссылка, будет вызвано исключение, и работа скрипта завершится. То есть все что нам нужно, это перехватить исключение и вывести необходимое нам сообщение.

Для решения этой задачи воспользуемся перехватом исключений. Сначала находим в [таблице](https://postgrespro.ru/docs/postgrespro/9.6/errcodes-appendix) код исключения. В нашем случае это foreign\_key\_violation. Затем приступаем к разработке функции:

**CREATE OR REPLACE FUNCTION** *delete\_subject* (  
 \_id **BIGINT**)  
**RETURNS VOID  
AS $$  
BEGIN  
 DELETE FROM** subject **WHERE id** = \_id; */\* Просто пытаемся удалить строку \*/*  
  
**EXCEPTION  
 WHEN** foreign\_key\_violation **THEN** */\* Перехват нужного нам исключения \*/* **RAISE EXCEPTION 'Невозможно выполнить удаление, так как есть внешние ссылки.'**;  
**END**;  
**$$ LANGUAGE** plpgsql;

Далее нужно подготовить таблицы к тестам. Таблица до изменений представлена на рисунке 9:

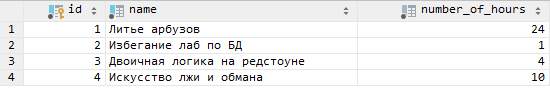


Рисунок 9 – Таблица subject до изменений.

Добавим в таблицу progress строку, в которой поле id\_subject будет указывать, например, на строку с индексом 1.

**INSERT INTO** progress (**id\_student**, **id\_subject**, **rating**) **VALUES** (1, 1, 5);

Затем вызовем функцию с ID строки, на которую нет внешних ссылок. В данном примере это строка с ID 4:

**SELECT** *delete\_subject*(4);

Функция отработала успешно. Теперь вызовем функцию с ID 1:

**SELECT** *delete\_subject*(1);

Функция была завершена с ошибкой, текст которой мы указали ранее. Результат работы функции представлен на рисунке 10:



Рисунок 10 – Результат работы функции delete\_subject(1)

На рисунке 11 представлена функция таблицы после изменений. Можно заметить, что строка с индексом 4 была удалена, а с индексом 1 – нет.

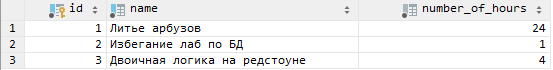


Рисунок 11 – Таблица после изменений.

## Задание 3. Функция фильтрации по числовому значению.

Данная функция будет принимать на вход числовое значение, а затем возвращать таблицу, в которой выбранное нами значение будет больше либо равно заданного нами числового значения.

Реализация функции будет для таблицы subject. Фильтрация будет производиться по полю number\_of\_hours. О том, как возвращать из функции результат запроса было рассказано в разделе "Запросы".

Ниже представлена разработанная функция:

**CREATE OR REPLACE FUNCTION** *filter\_subject\_by\_hours* (  
 min\_val **BIGINT**)  
**RETURNS SETOF** subject  
**AS $$  
BEGIN**

**RETURN QUERY** (**SELECT** *\** **FROM** subject **WHERE number\_of\_hours** >= min\_val);  
**END**;  
**$$ LANGUAGE** plpgsql;

Для проверки работы функции выполним запрос:

**SELECT** *\** **FROM** *filter\_subject\_by\_hours*(4);

Можно заметить, что функция должна вернуть строки о предметах, на которые предусмотрено 4 и более часов.

На рисунке 12 представлена таблица, возвращаемая функцией:



Рисунок 12 – Таблица, возвращаемая функцией filter\_subject\_by\_hours.

## Задание 4. Функция фильтрации массива объектов.

Данная функция будет принимать на вход массив объектов составного типа данных и числовое значение, по которому будет фильтроваться массив данных и возвращаться из функции.

Для начала необходимо создать тип данных. О составных типах данных можно узнать в разделе «Составные типы данных». Составной тип данных будет называться t\_subject и содержать поля, как в таблице subject:

**CREATE TYPE** t\_subject **AS** (  
 id **BIGINT**,  
 **name VARCHAR**(50),  
 number\_of\_hours **INTEGER**);

**NOTE:** В учебных целях рекомендуется создать тип данных с полями, полностью совпадающими с одной из таблиц. Это позволит проверять работу функции уже на готовых данных.

Также для дальнейшей работы необходимо быть знакомым с механизмом работы массивов, а также основными функциями работы с массивами. О массивах можно узнать в разделе «Массивы».

Ниже представлена разработанная функция:

**CREATE OR REPLACE FUNCTION** *filter\_array\_of\_subjects* (  
 arr **t\_subject**[],  
 filter\_var **INTEGER**)  
**RETURNS t\_subject**[]  
**AS $$  
BEGIN  
 RETURN** *ARRAY*( */\* Преобразуем выборку в массив \*/* **SELECT** (id, **name**, number\_of\_hours)::**t\_subject** */\* Создаем таблицу из элементов массива \*/* **FROM** *unnest*(arr)  
 **WHERE** number\_of\_hours >= filter\_var  
 );  
**END**;  
**$$ LANGUAGE** plpgsql;

Давайте разберемся в работе функции. Поиск по массиву можно делать с помощью SQL запросов. Для того, чтобы SQL мог работать с массивом, его нужно преобразовать в таблицу. Таким преобразованием занимается функция **unnest()**. То есть мы в секции FROM представляем массив **arr** как таблицу. Затем в секции SELECT мы берем нужные поля нашей таблицы и преобразуем их к типу **t\_subject**. В секции WHERE мы делаем простое условие для фильтрации выборки.

В итоге мы получили таблицу объектов типа t\_subject, отфильтрованную по значению filter\_var. По заданию нам необходимо, чтобы функция возвращала массив объектов, поэтому преобразуем нашу таблицу обратно в массив с помощью функции array().

Для проверки воспользуемся данным запросом. Он берет данные из таблицы subject, преобразует их в массив и отправляет его в качестве параметра нашей функции:

**SELECT** *filter\_array\_of\_subjects*(  
 *array*(**SELECT** (**id**, **name**, **number\_of\_hours**)::**t\_subject FROM** subject),  
 2  
 );

На рисунках 13 и 14 представлено сравнение таблицы subject и возврата функции filter\_array\_of\_subjects:

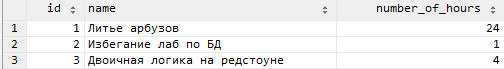


Рисунок 13 – Таблица subject.



Рисунок 14 – Вывод результата функции filter\_array\_of\_subjects.

Как мы видим, функция вернула массив объектов, отсортированных по значению 2.

## Задание 5. Таблица log\_subject.

Данная таблица будет содержать в себе информацию о вставке/изменении таблицы subject:

* Первичный ключ;
* Внешнюю ссылку на строку;
* Дату и время внесенных изменений (будет устанавливаться текущее время с помощью функции NOW);
* Старое значение (если был произведен UPDATE);
* Новое значение.

Ниже представлен скрипт создания таблицы:

**CREATE TABLE** log\_subject (  
 **id BIGSERIAL PRIMARY KEY**,  
 **subject\_id BIGINT REFERENCES** subject(**id**),  
 **change\_datetime TIMESTAMP DEFAULT** *NOW*(),  
 **old\_value INT DEFAULT NULL**,  
 **new\_value INT DEFAULT NULL**);

Для реализации логирования действий над таблицей subject нам необходимо создать триггер и триггерную функцию для него. Узнать о работе триггеров можно в разделе «Триггеры и триггерные функции».

Для начала создадим триггерную функцию. Она будет определять, какой вид запроса был произведен, и, в соответствии с ним, корректировать запрос. Это необходимо потому, что при INSERT нам нужно заполнить поле old\_value значением NULL. Функция всегда будет возвращать переменную NEW.

Ниже представлен скрипт создания функции:

**CREATE OR REPLACE FUNCTION** *trigger\_func*()  
**RETURNS TRIGGER  
AS $$  
DECLARE** old\_val **INT**;  
**BEGIN** */\* Определяем "старое" значение \*/* **IF** (*TG\_OP* = **'UPDATE'**) **THEN** old\_val := **OLD**.**number\_of\_hours**;  
 **ELSIF** (*TG\_OP* = **'INSERT'**) **THEN** old\_val := **NULL**;  
 **end if**;  
  
 */\* Производим запрос \*/* **INSERT INTO** log\_subject  
 (**subject\_id**, **old\_value**, **new\_value**)  
 **VALUES** (**NEW**.**id**, old\_val, **NEW**.**number\_of\_hours**);  
  
 **RETURN NEW**;  
**END**;  
**$$ LANGUAGE** plpgsql;

Можно заметить, что мы определяем, какое значение отправить в поле old\_value с помощью переменной TG\_OP.

Функция возвращает значение новой строки, заключенной в переменную NEW.

Теперь создадим триггер. Он будет срабатывать после изменений в таблице subject, так как нам нужно иметь записи об уже совершенных действиях. Триггер будет реагировать на запросы UPDATE и INSERT:

**CREATE TRIGGER** commit\_subject\_change  
**AFTER UPDATE OR INSERT  
ON** subject  
**FOR EACH ROW  
EXECUTE PROCEDURE** *trigger\_func*();

Чтобы проверить работу триггера, создадим новую строку в таблице subject, а затем изменим ее значение, подставив id новой записи:

**INSERT INTO** subject (**name**, **number\_of\_hours**) **VALUES** (**'М.Л. и Т.А.'**, 1000);  
**UPDATE** subject **SET number\_of\_hours** = 1001 **WHERE id** = 5;

На рисунке 15 можно заметить, что в таблице log\_subject добавилось 2 записи. В первой отсутствует поле old\_value – она соответствует добавлению нового значения. Вторая говорит нам о том, что было изменено существующее значение:



Рисунок 15 – Вывод таблицы log\_subject.

## Задание 6. Создание динамически формируемого запроса.

Вся суть динамически формируемых запросов описана в разделе «Динамически формируемые запросы».

Создадим функцию, которая будет принимать на вход название таблицы, название столбца и id поля, которое будет выведено. На выходе будет возвращаться текстовая строка с содержимым поля.

Ниже представлена разработанная функция:

**CREATE OR REPLACE FUNCTION** *get\_value\_by\_id* (  
 tableName **VARCHAR**,  
 columnName **VARCHAR**,  
 id **BIGINT**)  
**RETURNS** *TEXT***AS $$  
DECLARE** result *TEXT*;  
**BEGIN  
 EXECUTE 'SELECT '**|| columnName ||**' FROM '**|| tableName ||**' WHERE id = $1' USING** id **INTO** result;  
 **RETURN** result;  
**END**;  
**$$ LANGUAGE** plpgsql;

Для теста выполним скрипт:

**SELECT** *get\_value\_by\_id*(**'subject'**, **'name'**, 2);

На рисунке 16 представлен результат работы функции. Мы видим, что функция вернула содержимое столбца «name», строки с id 2, таблицы «subject».



Рисунок 16 – результат работы функции get\_value\_by\_id.