



**Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ Информатика, искусственный интеллект и системы управления

КАФЕДРА Системы обработки информации и управления

**Методические указания к лабораторным работам
по курсу «Постреляционные базы данных»**

**Лабораторная работа №2
«Программирование объектно-реляционной базы данных
на примере СУБД PostgreSQL»**

Виноградова М.В., Крутов Т.Ю., Соболева Е.Д., Волков А.С.

Под редакцией к.т.н. доц. Виноградовой М.В.

Москва, 2023 г.

ОГЛАВЛЕНИЕ

1. ЗАДАНИЕ.....	3
1.1. Цель работы.....	3
1.2. Средства выполнения.....	3
1.3. Продолжительность работы.....	3
1.4. Пункты задания для выполнения.....	3
1.5. Содержание отчета	4
2. ТЕОРЕТИКО-ПРАКТИЧЕСКИЙ МАТЕРИАЛ	5
2.1. Работа с СУБД PostgreSQL через pgAdmin	5
2.2. SQL/PSM.....	5
2.2.1. Объявление переменных.....	5
2.2.2. Присвоение значений переменным	6
2.2.3. Условные операторы	9
2.2.4. Циклы – LOOP	11
2.3. Программирование функций с применением SQL/PSM	12
2.3.1. Создание функций.....	13
2.3.2. Вызов определяемой пользователем функции	14
2.3.3. Аргументы SQL-функций.....	14
2.3.4. Скалярные функции	15
2.3.5. Табличные функции	17
2.3.6. Выполнение динамически формируемых команд.....	20
2.3.7. Функция format для форматирования строк	23
2.3.8. Хранимые процедуры	24
2.3.9. Вызов исключений	26
2.3.10. Перехват и обработка ошибок.....	28
2.3.11. Ограничение числа выводимых записей.....	30
2.3.12. Отображение измененных строк при выполнении запроса	31
2.3.13. Рекурсивный запрос	32
2.3.14. Динамические запросы	35
2.3.15. Ранжирующие запросы	39
2.3.16. Функция-агрегат	43
2.3.17. DML-триггеры	45
2.3.18. DDL-триггеры	48
3. КОНТРОЛЬНЫЕ ВОПРОСЫ	50
4. СПИСОК ИСТОЧНИКОВ.....	51

1. ЗАДАНИЕ

Лабораторная работа №2 «Программирование объектно-реляционной базы данных на примере СУБД PostgreSQL» по курсу «Постреляционные базы данных».

1.1. Цель работы

- Изучить постреляционные возможности языка SQL [1].
- Освоить языки и технологии SQL\PSM на примере PostgreSQL [2].
- Получить навыки программирования на стороне сервера.

1.2. Средства выполнения

- СУБД PostgreSQL,
- Утилита PgAdmin.

1.3. Продолжительность работы

Время выполнения лабораторной работы 4 часа.

1.4. Пункты задания для выполнения

1. Через PgAdmin [3] соединиться с PostgreSQL [2] и создать базу данных. В БД создать две-три связанные таблицы по теме, выданной преподавателем. Открыть таблицы на редактирование и заполнить тестовыми данными.

2. Базовое задание:

3. Создать скалярную функцию. Вызвать функцию из окна запроса.
4. Создать табличную функцию (inline). Вызвать функцию из окна запроса.
5. Создать табличную функцию (multi-statement). Продемонстрировать наполнение результирующего множества записей. Вызвать функцию из окна запроса.
6. Создать хранимую процедуру, содержащую запросы, вызов и перехват исключений. Вызвать процедуру из окна запроса. Проверить перехват и создание исключений.
7. Продемонстрировать в функциях и процедурах работу условных операторов и выполнение динамического запроса.

8. Расширенное задание:

9. Продемонстрировать выполнение рекурсивного запроса.
10. Продемонстрировать выполнение запроса на получение первых 3-х записей из результата (limit). Продемонстрировать выполнение запроса на добавление/изменение данных с отображением измененных строк (returning).
11. Продемонстрировать выполнение функций row_number(), Rank(), dense_rank(), ntile(4).
12. Создать функцию, работающую с курсором. Вызвать функцию из окна запроса.
13. Продемонстрировать в функциях и процедурах обращение к встроенным и системным функциям (строковые, работа с диском, работа со сложными типами и т.д.)

14. Дополнительное задание:

15. Создать функцию-агрегат. Проверить работоспособность функции из окна запроса
16. Создать представление на основе двух таблиц. Просмотреть результат запроса к представлению и попробовать внести изменения. Создать DML триггер на представление, который реализует изменение записи. Проверить работу триггера. Продемонстрировать работу с представлением.
17. Создать DDL триггер на удаление таблиц (запрет по условию).

1.5. Содержание отчета

- Титульный лист;
- Цель работы;
- Задание;
- Тексты SQL сценариев (создание объектов БД), запросов, команд и процедур в соответствии с пунктами задания.
- Результаты выполнения запросов (скриншоты);
- Вывод;
- Список используемой литературы.

2. ТЕОРЕТИКО-ПРАКТИЧЕСКИЙ МАТЕРИАЛ

2.1. Работа с СУБД PostgreSQL через pgAdmin

Порядок работы с СУБД PostgreSQL [2] через утилиту pgAdmin [3] был подробно описан в Лабораторной работе №1 «Создание объектно-реляционной базы данных на примере СУБД PostgreSQL» по курсу «Постреляционные базы данных».

2.2. SQL/PSM

SQL/PSM (Persistent Stored Modules) — стандарт, разработанный Американским национальным институтом стандартов (ANSI) в качестве расширения SQL. Стандарт главным образом определяет расширение с процедурным языком: разрешает объявление переменных, управление логикой исполнения, циклы и условия

PSM позволяет нам хранить процедуры как элементы схемы базы данных. PSM-смесь обычных операторов (if, while и т. д.) и SQL. Это позволяет нам делать то, что мы не можем сделать только в SQL.

Перечень **возможностей стандарта PSM** выглядит следующим образом:

- объявление и использование переменных,
- возврат значений,
- условия,
- циклы,
- работа с курсором,
- перехват, обработка и вызов исключений,
- вызов системных функций,
- DML и DDL команды языка SQL,
- динамические запросы.

В PostgreSQL можно программировать на нескольких языках, в том числе на PL/pgSQL [4].

2.2.1. Объявление переменных

Формально переменная объявляется [5] так, как показано ниже:

```
DECLARE <имя переменной> [DEFAULT <значение>] <тип>
```

Все переменные, используемые в блоке, должны быть определены в секции объявления. За исключением переменной-счётчика цикла FOR, которая объявляется автоматически.

Переменные могут иметь любой тип данных SQL, такой как integer, varchar, char и др. Особенностью этих переменных является то, что они могут хранить NULL-значения.

Пример объявления переменной:

```
DECLARE
    user_id integer;
    quantity numeric(5);
    url varchar;
    myrow tablename%ROWTYPE;
    myfield tablename.columnname%TYPE;
    arow RECORD;
```

Общий синтаксис объявления переменной:

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

2.2.2. Присвоение значений переменным

Присвоение значения переменной записывается в виде:

```
переменная := выражение;
-- ИЛИ
переменная = выражение;
```

Как описывалось ранее, выражение в таком операторе вычисляется с помощью SQL-команды SELECT, посылаемой в основную машину базы данных. Выражение должно получить одно значение (возможно, значение строки, если переменная строкового типа или типа record). Целевая переменная может быть простой переменной (возможно, дополненной именем блока), полем в переменной строкового типа или записи; или элементом массива, который является простой переменной или полем. Для присвоения можно использовать знак равенства (=) вместо совместимого с PL/SQL :=.

2.2.2.1. Целочисленная переменная

Ниже приведен пример объявления переменной типа `integer` и присвоения ей значения внутри функции. Раздел 2.3 данного документа подробно описывает работу с функциями.

```
CREATE OR REPLACE FUNCTION set_int_var(new_int integer)
  RETURNS integer AS
  $$
    DECLARE
      var_int integer;
    BEGIN
      var_int := new_int/2;
      RETURN var_int;
    END
  $$
LANGUAGE plpgsql;
```

Вызовем функцию следующим образом:

```
SELECT * FROM set_int_var(30);
```

Рисунок 1 демонстрирует результат выполнения запроса.



Data Output		Expla
	set_int_var	
	integer	
1		15

Рисунок 1 - Результат выполнения запроса над целочисленной переменной

2.2.2.2. Значение Null в целочисленной переменной

Ниже приведена функция, в которой объявляется переменная типа `integer`, и ей присваивается значение `Null`:


```
CREATE OR REPLACE FUNCTION set_null_var(new_int integer)
  RETURNS integer AS
  $$
    DECLARE
      var_int integer;
    BEGIN
      var_int := new_int/2;
      var_int := Null;
      RETURN var_int;
    END
  $$
```

```
$$  
LANGUAGE plpgsql;
```

Выполним запрос, вызвав функцию:

```
SELECT * FROM set_null_var(30);
```

Рисунок 2 демонстрирует результат выполнения запроса.



set_null_var
[null]

Рисунок 2 - Присвоение значения Null переменной

2.2.2.3. Переменная типа rowtype

В коде, представленном ниже, объявляется переменная типа rowtype, и ей присваивается значение строки таблицы Employee по значению поля id.

```
CREATE OR REPLACE FUNCTION get_row_var(emp_id integer) RETURNS text  
AS  
$$  
DECLARE  
    var_row public."Employee"%rowtype; -- Объявление переменной  
BEGIN  
    SELECT * INTO var_row FROM public."Employee" WHERE id = emp_id;  
    -- Присвоение переменной значения строки таблицы  
    IF NOT FOUND THEN  
        RAISE EXCEPTION 'Сотрудник с id % не найден', emp_id;  
    END IF;  
    RETURN var_row.last_name || ' ' || row.first_name || ' ' ||  
row.patronymic;  
END  
$$  
LANGUAGE plpgsql;
```

SELECT * INTO var_row помещает результат выполнения запроса в переменную var_row.

Конструкция IF NOT FOUND – это обработка ситуации, когда запрос не дал результата, RAISE EXCEPTION – это вызов исключения с сообщением об ошибке.

Фрагмент кода RETURN var_row.last_name || ' ' || row.first_name || ' ' || row.patronymic возвращает строку, содержащую значения полей last_name, first_name и patronymic из структуры var_row через пробел. Операторы || отвечают за конкатенацию строк.

Вызовем созданную функцию для вывода ФИО сотрудника с id=2:

```
SELECT * FROM get_row_var(2);
```

Рисунок 3 демонстрирует результат выполнения запроса.

Data Output	Explain	Messages	Notifications
<div>get_row_var text</div>			
1	Кузнецов Максим Максимович		

Рисунок 3 - Вывод ФИО сотрудника с использованием переменной типа rowtype

Попытка вывода ФИО сотрудника с несуществующим id выдаст ошибку.

Рисунок 4 демонстрирует текст ошибки.

```
SELECT * FROM get_row_var(200);
```

Data Output	Explain	Messages	Notifications
ERROR: ОШИБКА: Сотрудник с id 200 не найден КОНТЕКСТ: функция PL/pgSQL get_row_var(integer), строка 7, оператор RAISE			
SQL state: P0001 Context: функция PL/pgSQL get_row_var(integer), строка 7, оператор RAISE			

Рисунок 4 - Ошибка при вызове функции с несуществующим id сотрудника

2.2.3. Условные операторы

Формально синтаксис описания условного оператора [6] можно представить следующим образом:

```
IF <условие> THEN <операторы> [ELSIF <условие> THEN <операторы>]  
[ELSE <операторы>] END IF
```

В качестве примера объявим функцию, которая разделит заданное в параметре число пополам и сравнит результат с 10:

```
CREATE OR REPLACE FUNCTION compare_half_to_10(new_int integer)
RETURNS text AS
$$
DECLARE
    var_int integer;
    var_str text;
BEGIN
    var_int := new_int/2;
    IF var_int > 10
        THEN var_str := 'Больше 10';
    ELSEIF var_int = 10
        THEN var_str := 'Равно 10';
    ELSE var_str := 'Меньше 10';
    END IF;
    RETURN var_str;
END
$$
LANGUAGE plpgsql;
```

Вызовем эту функцию со значением 15. (см. Рисунок 5 с результатом выполнения запроса)

```
SELECT * FROM compare_half_to_10(15);
```

Data Output		Explain	Mes
compare_half_to_10			🔒
text			
1	Меньше 10		

Рисунок 5 - Вызов функции с условным оператором со значением 15

Теперь вызовем эту функцию со значением 20. (см. Рисунок 6 с результатом выполнения запроса)

```
SELECT * FROM compare_half_to_10(20);
```

compare_half_to_10		🔒
text		
1	Равно 10	

Рисунок 6 - Вызов функции с условным оператором со значением 20

Наконец, вызовем эту функцию со значением 30. (см. Рисунок 7 с результатом выполнения запроса):

```
SELECT * FROM compare_half_to_10(30);
```

Data Output		Explain	mes
	compare_half_to_10		
	text		
1	Больше 10		

Рисунок 7 - Вызов функции с условным оператором со значением 30

2.2.4. Циклы – LOOP

Формально синтаксис описания цикла [6] можно представить следующим образом:

```
[<<метка>>]
LOOP
    операторы
END LOOP [ метка ];
```

Ключевое слово LOOP организует безусловный цикл, который повторяется до бесконечности, пока не будет прекращён операторами EXIT или RETURN. Для вложенных циклов можно использовать метку в операторах EXIT и CONTINUE, чтобы указать, к какому циклу эти операторы относятся.

В качестве примера опишем функцию, которая при помощи цикла будет добавлять 1 к переданному значению до тех пор, пока оно не станет равным 0. Когда значение достигнет 100 (или изначально оно ≥ 100), то осуществится выход из цикла. Функция возвращает количество итераций по циклу.

```
CREATE OR REPLACE FUNCTION loop_to_100(new_int integer)
RETURNS integer AS
$$
DECLARE
    steps integer := 0;
BEGIN
    LOOP
        IF new_int >= 100 THEN
            EXIT; -- выход из цикла
        END IF;
        -- здесь производятся вычисления
    END LOOP;
END;
```

```

        new_int := new_int + 1;
        steps := steps + 1;
    END LOOP;
RETURN steps;
END
$$
LANGUAGE plpgsql;

```

Вызовем функцию для значения 15. (см. Рисунок 8 с результатом выполнения запроса)

```
SELECT * FROM loop_to_100(15);
```

Data Output		Explain
	loop_to_100 integer	🔒
1	85	

Рисунок 8 - Вызов функции с циклом со значением 15

Теперь повторим вызов функции со значением 120. (см. Рисунок 9 с результатом выполнения запроса)

```
SELECT * FROM loop_to_100(120);
```

Data Output		Explain
	loop_to_100 integer	🔒
1	0	

Рисунок 9 - Вызов функции с циклом со значением 120

2.3. Программирование функций с применением SQL/PSM

В языках программирования обычно имеется два типа подпрограмм:

- хранимые процедуры;
- определяемые пользователем функции (UDF).

Хранимые процедуры состоят из нескольких инструкций и имеют от нуля до нескольких входных параметров, но обычно не возвращают никаких параметров. В отличие от хранимых процедур, функции всегда возвращают одно значение.

Функции, определенные пользователем (UDF) — подпрограммы, которые принимают параметры, выполняют действие, например, сложные вычисления, и

возвращают результат этого действия в виде значения. Возвращаемое значение может быть либо единичным скалярным значением, либо результирующим набором (таблицей) [7].

2.3.1. Создание функций

Определяемые пользователем функции [8] создаются посредством инструкции CREATE FUNCTION, которая имеет следующий синтаксис:

```
CREATE FUNCTION [schema_name.]function_name

    [( {@param } type [= default]) {,...}

    RETURNS {scalar_type | [@variable] TABLE}

    [WITH {ENCRYPTION | SCHEMABINDING}

    [AS] {block | RETURN (select_statement)}
```

где:

- schema_name - имя схемы
- function_name - имя функции
- @param - входной параметр функции типа type
- default - значение параметра по умолчанию
- RETURNS - определяет тип значения, возвращаемого функцией:
 - для скалярной функции (раздел 0 данного документа) - стандартный тип данных, кроме timestamp
 - для табличной функции (раздел 2.3.5 данного документа) — тип TABLE.
- block определяет блок BEGIN/END, содержащий реализацию функции. Внутри блока BEGIN/END разрешаются операторы SQL, циклы, условия, вычисления переменных, перехваты и вызов исключений.
- RETURN с аргументом (возвращаемым значением) - последняя инструкция блока.

2.3.2. Вызов определяемой пользователем функции

Определенную пользователем функцию можно вызывать с помощью инструкций SELECT, INSERT, UPDATE или DELETE.

Вызов функции осуществляется, указывая ее имя с парой круглых скобок в конце, в которых можно задать один или несколько аргументов.

Аргументы — это значения или выражения, которые передаются входным параметрам, определяемым сразу же после имени функции.

При вызове функции, когда для ее параметров не определены значения по умолчанию, для всех этих параметров необходимо предоставить аргументы в том же самом порядке, в каком эти параметры определены в инструкции CREATE FUNCTION.

2.3.3. Аргументы SQL-функций

К аргументам SQL-функции можно обращаться в теле функции по именам или номерам. Ниже приведены примеры обоих вариантов [9].

Чтобы использовать имя, объявите аргумент функции как именованный, а затем просто пишите это имя в теле функции. Если имя аргумента совпадает с именем какого-либо столбца в текущей SQL-команде внутри функции, имя столбца будет иметь приоритет. Чтобы всё же перекрыть имя столбца, дополните имя аргумента именем самой функции, то есть запишите его в виде *имя_функции.имя_аргумента*. (Если и это имя будет конфликтовать с полным именем столбца, снова выиграет имя столбца. Неоднозначности в этом случае вы можете избежать, выбрав другой псевдоним для таблицы в SQL-команде.)

Старый подход с нумерацией позволяет обращаться к аргументам, применяя запись *\$n*: *\$1* обозначает первый аргумент, *\$2* — второй и т. д. Это будет работать и в том случае, если данному аргументу назначено имя.

Если аргумент имеет составной тип, то для обращения к его атрибутам можно использовать запись с точкой, например: *аргумент.поле* или *\$1.поле*. И опять же, при этом может потребоваться дополнить имя аргумента именем функции, чтобы сделать имя аргумента однозначным.

Аргументы SQL-функции могут использоваться только как значения данных, но не как идентификаторы. Например, это приемлемо:

```
INSERT INTO mytable VALUES ($1);
```

а это не будет работать:

```
INSERT INTO $1 VALUES (42);
```

2.3.4. Скалярные функции

Функция является **скалярной**, если предложение RETURNS определяет один из скалярных типов данных и возвращает в качестве ответа единственное значение при каждом вызове функции.

Рисунок 10 демонстрирует обобщенную структуру скалярной функции [7].



Рисунок 10 - Структура скалярной функции

В качестве примера опишем функцию, которая получает значения x и y , складывает их и возвращает результат:

```
CREATE FUNCTION add_xy(x integer, y integer)  
    RETURNS integer AS $$  
        SELECT x + y;  
    $$ LANGUAGE SQL;
```

Вызовем функцию для сложения цифр 1 и 2:

```
SELECT add_xy(1, 2);
```

Рисунок 11 демонстрирует результат выполнения запроса.

Data Output	Exp
add_xy integer	
1	3

Рисунок 11 - Результат выполнения функции сложения

Создадим другую функцию `max_work_hours`, которая будет возвращать максимальное значение рабочих часов сотрудника в определенном отделе:

```
CREATE OR REPLACE FUNCTION "max_work_hours"(dep_id integer)
  RETURNS integer AS
  'SELECT max(work_hours_per_week)
  FROM public."Employee"
  WHERE department_id=dep_id' LANGUAGE SQL volatile;
```

Вызовем функцию для отдела с `id=1`:

```
SELECT "max_work_hours"(1);
```

Рисунок 12 демонстрирует результат выполнения запроса.

Data Output	Explain	Mes
max_work_hours integer		
1		39

Рисунок 12 - Результат выполнения функции определения макс. значения раб. часов

Создадим аналогичную функцию, но уже с использованием объявляемой целочисленной переменной, которая будет хранить среднее кол-во рабочих часов у сотрудников отдела с заданным `id`.

```
CREATE OR REPLACE FUNCTION get_avg_work_hours(dep_id integer)
  RETURNS integer AS
  $$
  DECLARE
    avg_hours integer;          -- объявление переменной
  BEGIN
    -- присвоение результата запроса переменной
    SELECT avg(work_hours_per_week) INTO avg_hours
    FROM public."Employee" WHERE department_id=dep_id;

    -- возврат переменной
```



```

RETURN avg_hours;

END

$$

LANGUAGE plpgsql;

```

Вызовем данную функцию для получения среднего кол-ва рабочих часов для отдела с id=1. Рисунок 13 демонстрирует результат выполнения запроса.

```
SELECT * FROM get_avg_work_hours(1);
```

Data Output	Explain	Mess:
get_avg_work_hours integer		
1		24

Рисунок 13 - Результат вычисления среднего кол-ва рабочих часов

Теперь повторим вызов данной функции, используем возвращаемое значение в качестве одного из параметров условия WHERE в запросе. Выведем список сотрудников отдела с id=1, у которых кол-во рабочих часов в неделю больше среднего значения по отделу. Результат выполнения запроса представляет собой строки таблицы, соответствующие подходящим по условию сотрудникам (см. Рисунок 14).

```
Select * FROM public."Employee" WHERE department_id=1 and
work_hours_per_week>get_avg_work_hours(1);
```

Data Output		Explain	Messages	Notifications										
id	[PK] integer	last_name	character varying (30)	first_name	character varying (30)	patronymic	character varying (30)	birth_date	date	work_hours_per_week	integer	department_id	integer	c
1		1	Степанова	Светлана		Степановна		[null]		33		33		1 [n]
2		8	Субботин	Георгий		Иванович		[null]				33		1 [n]
3		15	Петров	Петр		Петрович		[null]				39		1 [n]

Рисунок 14 - Сотрудники с кол-вом рабочих часов выше среднего

2.3.5. Табличные функции

Табличная функция возвращает набор строк. За возвращаемое значение отвечает ключевое слово RETURNS. Если тело функции является одним SQL-запросом, то такая функция называется встроенной («inline»). Запрос встроенной функции может рассматриваться как обычный подзапрос с параметром. Инструкция SELECT встроенной функции возвращает результирующий набор в виде переменной с типом данных TABLE.

Многоинструкционная («multi-statement») табличная функция содержит имя, определяющее внутреннюю переменную с типом данных TABLE. Этот тип данных

указывается ключевым словом TABLE, которое следует за именем переменной. В эту переменную вставляются выбранные строки, и она служит возвращаемым значением функции. Такой тип функции состоит из нескольких команд и запросов, вызывается и выполняется как программа.

Рисунок 15 демонстрирует обобщенную структуру табличной функции [7].



Рисунок 15 - Структура табличной функции

Простейший пример синтаксиса записи табличной функции (inline) представлен ниже:

```
CREATE FUNCTION sum_n_product_with_tab (x int)
    RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
    $$ LANGUAGE SQL;
```

Для функции типа «multi-statement» переменная, указанная как возвращаемый тип, содержит набор записей, которые возвращаются в качестве результата функции. Код функции должен обеспечить заполнение этой переменной, например, командой «insert».

Для возврата множество записей также используют конструкции вида:

```
RETURN NEXT выражение; -- добавить в результат строку
RETURN QUERY запрос;   -- выполнить запрос и вернуть его результат
```

```
RETURN QUERY EXECUTE строка-команды; -- выполнить динамический запрос
и вернуть его результат
```

В данном случае оператор RETURN пополняет множество результирующих записей, но не завершает выполнение функции. Далее рассмотрены примеры.

Табличная функция указывается в части «FROM» запроса.

В качестве более реалистичного примера создадим функцию, которая выводит ФИО всех сотрудников отдела с id = dep_id.

```
CREATE OR REPLACE FUNCTION public.employee_fio(dep_id integer)
    RETURNS TABLE(id integer, last_name varchar,
                    first_name varchar, patronymic varchar) AS
    'SELECT id, last_name, first_name, patronymic
      FROM public."Employee"
     WHERE department_id=dep_id'
    LANGUAGE SQL
    ROWS 100;

ALTER FUNCTION public.employee_fio(integer)
    OWNER TO postgres;
```

Вызов функции производится, например, для отдела с id=1, производится привычным образом:

```
SELECT * FROM employee_fio(1);
```

Функцию также можно вызвать в подзапросе. Например, выполним запрос, который выведет всю информацию о сотрудниках, которые относятся к отделу с id=1 и у которых фамилия - Павлов.

```
SELECT * FROM public."Employee" WHERE id IN (SELECT id FROM
employee_fio(1) WHERE last_name='Павлов');
```

Ниже приведен пример синтаксиса функции с циклом:

```
CREATE OR REPLACE FUNCTION get_all_foo()
    RETURNS SETOF foo AS

$BODY$

DECLARE r foo%rowtype;
```

```

BEGIN
FOR r IN SELECT * FROM foo WHERE fooid > 0 LOOP
    -- здесь возможна обработка данных
    RETURN NEXT r; -- возвращается текущая строка запроса
END LOOP;
RETURN;
END;
$BODY$ LANGUAGE plpgsql;

SELECT * FROM get_all_foo();

```

Приведенный ниже пример функции описывает вывод всех рейсов для введенной даты. Если таких рейсов нет, то вызывается исключение.

```

CREATE FUNCTION get_available_flightid(date)
    RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid FROM flight
        WHERE flightdate >= $1 AND flightdate < ($1 + 1);
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Нет рейсов на дату: %.', $1;
    END IF;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

-- Возвращает доступные рейсы либо вызывает исключение, если их нет.
SELECT * FROM get_available_flightid(CURRENT_DATE);

```

2.3.6. Выполнение динамически формируемых команд (запросов)

Часто требуется динамически формировать команды внутри функций на PL/pgSQL, то есть такие команды, в которых при каждом выполнении могут использоваться разные таблицы или типы данных [10]. Обычно PL/pgSQL кеширует

планы выполнения, но в случае с динамическими командами это не будет работать. Для исполнения динамических команд предусмотрен оператор **EXECUTE**:

```
EXECUTE строка-команды [ INTO [STRICT] цель ] [ USING выражение [, ... ] ];
```

где строка-команды — это выражение, формирующее строку (типа `text`) с текстом команды, которую нужно выполнить. Необязательная цель — это переменная-запись, переменная-кортеж или разделённый запятыми список простых переменных и полей записи/кортежа, куда будут помещены результаты команды. Необязательные выражения в `USING` формируют значения, которые будут вставлены в команду.

В сформированном тексте команды замена имён переменных PL/pgSQL на их значения проводиться не будет. Все необходимые значения переменных должны быть вставлены в командную строку при её построении, либо нужно использовать параметры, как описано ниже.

Также, нет никакого плана кеширования для команд, выполняемых с помощью `EXECUTE`. Вместо этого план создаётся каждый раз при выполнении. Таким образом, строка команды может динамически создаваться внутри функции для выполнения действий с различными таблицами и столбцами.

Предложение `INTO` указывает, куда должны быть помещены результаты SQL-команды, возвращающей строки. Если используется переменная строкового типа или список переменных, то они должны в точности соответствовать структуре результата команды; если используется переменная типа `record`, она автоматически приводится к строковому типу результата запроса. Если возвращается несколько строк, то только первая будет присвоена переменной(ым) в `INTO`. Если не возвращается ни одной строки, то присваивается `NULL`. Без предложения `INTO` результаты команды отбрасываются.

С указанием `STRICT` команда должна вернуть ровно одну строку, иначе выдаётся сообщение об ошибке.

В тексте команды можно использовать значения параметров, **ссылки на параметры** обозначаются как `$1`, `$2` и т. д. [9]. Эти символы указывают на значения, находящиеся в предложении `USING`. Такой метод зачастую предпочтительнее, чем вставка значений в команду в виде текста: он позволяет исключить во время исполнения дополнительные расходы на преобразования значений в текст и обратно, и

не открывает возможности для SQL-инъекций, не требуя применять экранирование или кавычки для спецсимволов. Пример:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND
inserted <= $2'
      INTO c
      USING checked_user, checked_date;
```

Обратите внимание, что символы параметров можно использовать только вместо значений данных. Если же требуется динамически формировать имена таблиц или столбцов, их необходимо вставлять в виде текста. Например, если в предыдущем запросе необходимо динамически задавать имя таблицы, можно сделать следующее:

```
EXECUTE 'SELECT count(*) FROM '
      || quote_ident(tabname)
      || ' WHERE inserted_by = $1 AND inserted <= $2'
      INTO c
      USING checked_user, checked_date;
```

Более аккуратным решением будет использование указания %I с функцией format(), позволяющее вставить имя таблицы или столбца, которое будет автоматически заключено в кавычки:

```
EXECUTE format('SELECT count(*) FROM %I '
      'WHERE inserted_by = $1 AND inserted <= $2', tabname)
      INTO c
      USING checked_user, checked_date;
```

Этот пример задействует SQL-правило, согласно которому строковые значения, разделённые символом новой строки, соединяются вместе.

Ещё одно ограничение состоит в том, что символы параметров могут использоваться только в оптимизируемых SQL-командах (SELECT, INSERT, UPDATE и DELETE). В операторы других типов (обычно называемые служебными) значения нужно вставлять в текстовом виде, даже если это просто значения данных.

Команда EXECUTE с неизменяемым текстом и параметрами USING (как в первом примере выше), функционально эквивалентна команде, записанной напрямую в PL/pgSQL, в которой переменные PL/pgSQL автоматически заменяются значениями. Важное отличие в том, что EXECUTE при каждом исполнении заново строит план

команды с учётом текущих значений параметров, тогда как PL/pgSQL строит общий план выполнения и кеширует его при повторном использовании. В тех случаях, когда наилучший план выполнения сильно зависит от значений параметров, может быть полезно использовать EXECUTE для гарантии того, что не будет выбран общий план.

2.3.7. Функция **format** для форматирования строк

Функция **format** выдаёт текст, отформатированный в соответствии со строкой формата, подобно функции `sprintf` в C [11].

```
format(formatstr text [, formatarg "any" [, ...] ])
```

`formatstr` — строка, определяющая, как будет форматироваться результат. Обычный текст в строке формата непосредственно копируется в результат, за исключением спецификаторов формата. Спецификаторы формата представляют собой местозаполнители, определяющие, как должны форматироваться и выводиться в результате аргументы функции. Каждый аргумент `formatarg` преобразуется в текст по правилам вывода своего типа данных, а затем форматируется и вставляется в результирующую строку согласно спецификаторам формата.

Спецификаторы формата предваряются символом `%` и имеют форму

```
%[position][flags][width]type
```

Здесь:

- *position*
 - Строка вида `n$`, где `n` — индекс выводимого аргумента. Индекс, равный 1, выбирает первый аргумент после `formatstr`. Если позиция опускается, по умолчанию используется следующий аргумент по порядку.
- *flags*
 - Дополнительные параметры, управляющие форматированием данного спецификатора. В настоящее время поддерживается только знак минус (`-`), который выравнивает результата спецификатора по левому краю. Он работает, только если также определена ширина.
- *width*
 - Задаёт минимальное число символов, которое будет занимать результат данного спецификатора. Выводимое значение

выравнивается по правой или левой стороне (в зависимости от flags) с дополнением необходимым числом пробелов.

- *type*
 - Тип спецификатора определяет преобразование соответствующего выводимого значения. Поддерживаются следующие типы:
 - **s** форматирует значение аргумента как простую строку. Значение NULL представляется пустой строкой.
 - **I** обрабатывает значение аргумента как SQL-идентификатор, при необходимости заключая его в кавычки. Значение NULL для такого преобразования считается ошибочным.
 - **L** заключает значение аргумента в апострофы, как строку SQL. Значение NULL выводится буквально, как NULL, без кавычек.

В дополнение к спецификаторам, описанным выше, можно использовать спецпоследовательность %, которая просто выведет символ %.

Несколько примеров простых преобразований формата:

```
SELECT format('Hello %s', 'World');
```

Результат: Hello World

```
SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
```

Результат: Testing one, two, three, %

```
SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\Reilly');
```

Результат: INSERT INTO "Foo bar" VALUES('O'Reilly')

```
SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
```

Результат: INSERT INTO locations VALUES('C:\Program Files')

2.3.8. Хранимые процедуры

Хранимые процедуры — это предварительно откомпилированные процедуры, программы, написанные на SQL/PSM и находящиеся в базе.

Особенности [7]:

- Выполняются быстрее обычного SQL/PSM, т. к. хранятся в откомпилированном виде.

- Могут помочь изолировать пользователей от базовых табличных структур, скрыть особенности реализации какой-либо возможности.
- Вызываются клиентской программой, другой хранимой процедурой или триггером.
- Объединяет запросы и процедурную логику (операторы присваивания, логического ветвления и т.п.) и хранится в БД.

Рисунок 16 демонстрирует обобщенную структуру хранимой процедуры [7].

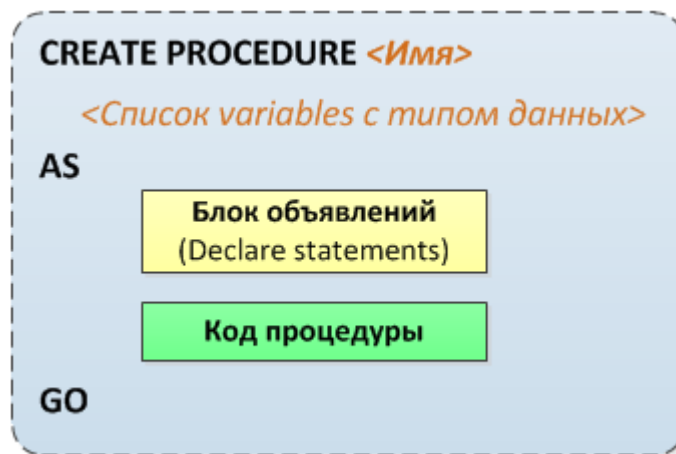


Рисунок 16 - Структура хранимой процедуры

В качестве примера опишем процедуру [12], которая создаст новую таблицу с полями id, title, description и названием, переданным в аргументы.

```

CREATE OR REPLACE PROCEDURE public.create_new_table(tablename
varchar(100))
AS $$
BEGIN
EXECUTE format('
                CREATE TABLE IF NOT EXISTS %I(
                    id integer PRIMARY KEY,
                    title varchar(100),
                    description text )', tablename);

END
$$ LANGUAGE 'plpgsql';
  
```

Вызовем процедуру:

```
CALL create_new_table('New Table');
```

В списке таблиц видно, что действительно была создана новая таблица (см. Рисунок 17).

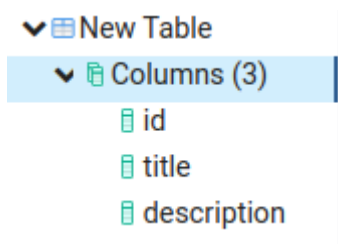


Рисунок 17 - Обновленный список таблиц

Также в качестве доказательства существования таблицы можем выполнить запрос, выводящий все строки таблицы. Рисунок 18 демонстрирует, что запрос успешно выполнен и вывел пустую строку с соответствующими столбцами.

```
SELECT * FROM public."New Table";
```

Data Output				Explain	Messages	Notifications
	id [PK] integer		title character varying (100)		description text	

Рисунок 18 - Вывод содержимого новой таблицы

2.3.9. Вызов исключений

Команда **RAISE** предназначена для вывода сообщений и вызова ошибок [13].

Обобщенный синтаксис команды можно представить следующим образом:

```
RAISE [ уровень ] 'формат' [, выражение [, ... ] ] [ USING параметр = значение [, ... ] ] ;
```

Поле **уровень** задаёт уровень важности ошибки. Возможные значения: **DEBUG**, **LOG**, **INFO**, **NOTICE**, **WARNING** и **EXCEPTION**.

По умолчанию используется **EXCEPTION**. **EXCEPTION** вызывает ошибку (что обычно прерывает текущую транзакцию), остальные значения уровня только генерируют сообщения с различными уровнями приоритета.

При помощи **USING** можно добавить дополнительную информацию к отчёту об ошибке. Возможные ключевые слова для параметра следующие:

- **MESSAGE** – устанавливает текст сообщения об ошибке.
- **DETAIL** – предоставляет детальное сообщение об ошибке.
- **HINT** – предоставляет подсказку по вызванной ошибке.

- **ERRCODE** – устанавливает код ошибки (SQLSTATE). Код ошибки задаётся либо по имени или напрямую, пятисимвольный код SQLSTATE.
- **COLUMN**
- **CONSTRAINT**
- **DATATYPE**
- **TABLE**
- **SCHEMA** – предоставляет имя соответствующего объекта, связанного с ошибкой.

В качестве примера создадим функцию, которая проверяет, существует ли сотрудник с заданной фамилией, перехватывает случай, если не существует и вызывает исключение.

```
CREATE OR REPLACE FUNCTION public.check_lastname(lastname
varchar(100))
  RETURNS text
  AS $$
  DECLARE
    emp_id "Employee".id%TYPE;
  BEGIN
    -- поиск сотрудников с заданной фамилией
    SELECT "id" INTO emp_id FROM public."Employee"
      WHERE "last_name" = lastname;
    -- перехват исключения. Проверка, что сотрудник найден, то есть
    emp_id не пустой, в противном случае вызов исключения
    IF emp_id IS NULL THEN
      RAISE EXCEPTION USING errcode='E0001',
        message='Сотрудник с фамилией ' || $1 || ' не
существует';
    END IF;
    RETURN 'Существует';
  END
  $$ LANGUAGE 'plpgsql';
```

Вызовем функцию для фамилии, которая присутствует в таблице сотрудников. Рисунок 19 демонстрирует, что функция успешно выполнится и выведет результат.

```
SELECT * FROM check_lastname('Сергеев');
```

Data Output	Explain	M
check_lastname		
text		
1	Существует	

Рисунок 19 - Успешная проверка существования фамилии

Теперь вызовем функцию для фамилии, которой нет ни у одного сотрудника. Рисунок 20 демонстрирует, что функция будет выполнена с ошибкой, и будет выведено соответствующее сообщение.

```
SELECT * FROM check_lastname('Зубкова');
```

Data Output	Explain	Messages	Notifications
ERROR: ОШИБКА: Сотрудник с фамилией Зубкова не существует			
КОНТЕКСТ: функция PL/pgSQL check_lastname(character varying), строка 8, оператор RAISE			
SQL state: E0001			
Context: функция PL/pgSQL check_lastname(character varying), строка 8, оператор RAISE			

Рисунок 20 - Ошибка при проверке фамилии

2.3.10. Перехват и обработка ошибок

По умолчанию любая возникающая ошибка прерывает выполнение функции на PL/pgSQL и транзакцию, в которой она выполняется. Использование в блоке секции EXCEPTION позволяет перехватывать и обрабатывать ошибки [6]. Синтаксис секции EXCEPTION расширяет синтаксис обычного блока:

```
[ <<метка>> ]
[ DECLARE
    объявления ]
BEGIN
    операторы
EXCEPTION
    WHEN условие [ OR условие ... ] THEN
        операторы_обработчика
    [ WHEN условие [ OR условие ... ] THEN
        операторы_обработчика
    ... ]
END;
```

Если ошибок не было, то выполняются все операторы блока и управление переходит к следующему оператору после END. Но если при выполнении оператора происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции EXCEPTION. В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено, то выполняются соответствующие операторы_обработчика и управление переходит к следующему оператору после END. Если исключение не найдено, то ошибка передаётся наружу, как будто секции EXCEPTION не было. При этом ошибку можно перехватить в секции EXCEPTION внешнего блока. Если ошибка так и не была перехвачена, то обработка функции прекращается.

В качестве условия может задаваться одно из имён, соответствующих кодам ошибок PostgreSQL [14]. Если задаётся имя категории, ему соответствуют все ошибки в данной категории. Специальному имени условия OTHERS («другие») соответствуют все типы ошибок, кроме QUERY_CANCELED и ASSERT_FAILURE. (И эти два типа ошибок можно перехватить по имени, но часто это неразумно.) Имена условий воспринимаются без учёта регистра. Условие ошибки также можно задать кодом SQLSTATE; например, эти два варианта равнозначны:

```
WHEN division_by_zero THEN ...  
WHEN SQLSTATE '22012' THEN ...
```

Если при выполнении операторов_обработчика возникнет новая ошибка, то она не может быть перехвачена в этой секции EXCEPTION. Ошибка передаётся наружу и её можно перехватить в секции EXCEPTION внешнего блока.

При выполнении команд в секции EXCEPTION локальные переменные функции на PL/pgSQL сохраняют те значения, которые были на момент возникновения ошибки. Однако все изменения в базе данных, выполненные в блоке, будут отменены. В качестве примера рассмотрим следующий фрагмент:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');  
BEGIN  
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';  
    x := x + 1;  
    y := x / 0;  
EXCEPTION  
    WHEN division_by_zero THEN
```

```
        RAISE NOTICE 'перехватили ошибку division_by_zero';
        RETURN x;

END;
```

При присваивании значения переменной `y` произойдёт ошибка `division_by_zero`. Она будет перехвачена в секции `EXCEPTION`. Оператор `RETURN` вернёт значение `x`, увеличенное на единицу, но изменения, сделанные командой `UPDATE`, будут отменены. Изменения, выполненные командой `INSERT`, которая предшествует блоку, не будут отменены. В результате, база данных будет содержать Tom Jones, а не Joe Jones.

Аналогично можно привести пример обработки любой ошибки с использованием ключевого слова `OTHERS`:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe'
        WHERE lastname = 'Jones';
    EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'перехватили ошибку';
    -- действия;
END;
```

2.3.11. Ограничение числа выводимых записей

Ключевое слово **LIMIT** ограничивает число выводимых записей [15].

В качестве примера выведем первые 10 записей с использованием табличной функции `employee_fio`, которая выводит сотрудников отдела с `id=1`. Рисунок 21 демонстрирует результат выполнения запроса.

```
SELECT * FROM employee_fio(1) LIMIT 10;
```






Data Output		Explain	Messages	Notifications	
	Id integer	 last_name character varying	 first_name character varying	 patronymic character varying	
1	1	Степанова	Светлана	Степановна	
2	2	Кузнецов	Максим	Максимович	
3	5	Сергеев	Сергей	Сергеевич	
4	6	Кузнецова	Анна	Олеговна	
5	8	Субботин	Георгий	Иванович	
6	11	Павлов	Павел	Павлович	
7	13	Захарьян	Захар	Захарович	
8	14	Андреев	Андрей	Андреевич	
9	15	Петров	Петр	Петрович	
10	6	Кузнецова	Анна	Олеговна	

Рисунок 21 - Ограничение вывода на 10 записей

2.3.12. Отображение измененных строк при выполнении запроса

Ключевое слово **RETURNING** позволяет осуществлять запросы на добавление/изменение (INSERT/UPDATE) данных с отображением измененных строк [16]. Список RETURNING имеет тот же синтаксис, что и список результатов SELECT.

В качестве примера выполним запрос, который выполнит вставку нового сотрудника и вернет вставленные значения. Рисунок 22 демонстрирует результат выполнения запроса.

```
INSERT INTO public."Employee"(id, last_name, first_name, patronymic,
work_hours_per_week)
VALUES (1, 'Иванова', 'Елена', 'Дмитриевна', 20)
RETURNING *;
```

Query Editor

Query History

```

1 INSERT INTO public."Employee" (id, last_name, first_name, patronymic,
2                                work_hours_per_week)
3 VALUES (1, 'Иванова', 'Елена', 'Дмитриевна', 20)
4 RETURNING *;
5

```

Data Output

Explain

Messages

Notifications

	id [PK] integer	last_name character varying (30)	first_name character varying (30)	patronymic character varying (30)	birth_date date	work_hours_per_week integer	di in
1	1	Иванова	Елена	Дмитриевна	[null]	20	

Рисунок 22 - Вывод добавленных строк таблицы

Приведем пример запроса, который изменяет значения в существующих строках. Данный запрос выполнит изменение поля количество рабочих часов и после выполнения изменения строки вернет ее (см. Рисунок 23).

```
UPDATE public."Employee"
SET work_hours_per_week=35
WHERE last_name = 'Иванова' AND id=1
RETURNING *;
```

```
1 UPDATE public."Employee"
2 SET work_hours_per_week=35
3 WHERE last_name = 'Иванова' AND id=1
4 RETURNING *;
5
```

	Data Output	Explain	Messages	Notifications
	id [PK] integer	last_name character varying (30)	first_name character varying (30)	patronymic character varying (30)
1	1	Иванова	Елена	Дмитриевна
				birth_date date
				work_hours_per_week integer
				35

Рисунок 23 - Вывод измененной в запросе строки

Параметры ключевого слова RETURNING можно модифицировать таким образом, чтобы выводились только конкретные столбцы (см. Рисунок 24).

```
1 UPDATE public."Employee"
2 SET work_hours_per_week=35
3 WHERE last_name = 'Иванова' AND id=1
4 RETURNING last_name, work_hours_per_week;
5
```

	Data Output	Explain	Messages	Notifications
	last_name character varying (30)		work_hours_per_week integer	
1	Иванова		35	

Рисунок 24 - Вывод определенных столбцов в измененной строке

2.3.13. Рекурсивный запрос

Рекурсивный запрос необходим для вывода данных на основе предыдущих строк в выборке. Реализуется он с помощью оператора WITH [17].

Общая схема рекурсивного запроса:

```
WITH RECURSIVE t AS (
```



```

        нерекурсивная часть (база)          (1)
    UNION ALL
    рекурсивная часть (индукционные шаги)    (2)
    )
    SELECT * FROM t;                          (3)

```

Внутри рекурсивный запрос (то, что записано в скобках после ключевого слова AS) можно разделить на две части, которые объединены ключевым словом UNION. Первая часть (базис) — это запрос для поиска элемента, с которого следует начать рекурсивный запрос. Вторая часть (индукция) — то, что выполняется в каждой итерации.

Сначала выполняется база рекурсии, и ее результат помещается в промежуточное отношение t. Затем многократно выполняется индукционная часть, каждая итерация пополняет содержимое отношения t. Итерации выполняются до тех пор, пока меняется содержимое отношения t. После завершения всех шагов индукции будет выполнен запрос к построенному отношению: SELECT * FROM t.

База индукции, то есть нерекурсивная часть, отделяется от индукционного шага с помощью UNION (количество и типы столбцов в обоих запросах должны совпадать). Также необходимо обеспечить возможность остановки рекурсивного запроса.

Рекурсивный запрос останавливается, когда результат текущей итерации не изменяет отношение t. Чтобы избежать бесконечной рекурсии в индукционной части нельзя использовать то, что может изменить полученный ранее результат, например, операторы разности, исключения, группировки, удаления дубликатов и т.д.

Ограничения на содержимое запроса к построенному отношению t не накладываются.

В качестве примера пусть имеется таблица Department (Отдел) с полем «является подразделом» - subdep_of. Добавим соответствующее поле:

```
ALTER TABLE public."Department" ADD subdep_of integer;
```

Заполним таблицу и выведем содержимое таблицы (см. Рисунок 25).

шаге. В данном примере видим, что главным отделом является отдел 5, его подотделом является 2 отдел, а у 2-го подотделом является отдел 4.

2.3.14. Динамические запросы

Динамические запросы — это запросы, текст которых формируется во время выполнения приложения и заранее не компилируется.

Под динамическими запросами понимаются запросы SQL, текст которых формируется и затем выполняется внутри PL/pgSQL-блока, например, в хранимых функциях или в анонимных блоках на этом процедурном языке [18].

Особенности использования:

- дополнительная гибкость в приложении,
- оптимизация отдельных запросов.

Цена:

- не используются подготовленные операторы,
- возрастает риск внедрения SQL-кода,
- возрастает сложность сопровождения.

Синтаксис динамического запроса:

```
EXECUTE строка-команды
      [ INTO [STRICT] цель]
      [ USING выражение[, ...]];
```

Особенности:

- цель может быть переменной составного типа или списком скалярных переменных,
- STRICT — гарантия одной строки,
- USING — подстановка значений параметров,
- проверка результата — GET DIAGNOSTICS, FOUND.

Пример:

```
DECLARE TableName VarChar(200) -- задание переменной
SET TableName = "Products" -- установка переменной значения названия
таблицы
EXECUTE ('SELECT * FROM' + TableName) -- выполнение запроса, который
выберет все записи из таблицы (для сервера это просто строка)
```

Т.к. сервер воспринимает наш код как строку, то туда можно ставить все, что угодно. А EXECUTE пытается сделать из этой строки работающий запрос и выполнить его.

2.3.14.1. Пример динамического запроса с использованием процедуры

Создадим процедуру, которая будет создавать отдельную таблицу для указанного в параметре спорта. Таблица будет называться так же, как и название спорта. В эту таблицу будут сохранены строки из таблицы Sport_Section, удовлетворяющие условию.

```
CREATE OR REPLACE PROCEDURE public.copy_sport(sport_title
varchar(100))  --(1)
AS $$
DECLARE
sport_id "Sport".id%TYPE;                                --(2)
BEGIN
    SELECT "id" INTO sport_id FROM public."Sport"          --(3)
    WHERE "title" = sport_title;
    IF sport_id IS NULL THEN                                --(4)
        RAISE EXCEPTION USING errcode='E0001',            --(5)
        hint='Измените название вида спорта(title) или добавьте
        сперва спорт',                                     --(6)
        message='Вид спорта с title=' || $1 || ' не найден!'; --
(7)
    ELSE
        EXECUTE format('    CREATE TABLE IF NOT EXISTS %I( --(8)
            id integer PRIMARY KEY,
            title varchar(100),
            description text
        )', sport_title);
        EXECUTE format('DELETE FROM %I', sport_title); --(9)
        EXECUTE format('INSERT INTO %I                                --(10)
            SELECT "id", "title", "description"
            FROM public."Sport_Section"
            WHERE "sport" = %L', $1, sport_id);
    END IF;
END
```

```
$$ LANGUAGE 'plpgsql';
```

Ниже дано описание всех шагов данной процедуры, отмеченных комментарием-цифрой:

- 1 — создание процедуры копирования вида спорта с параметром «название спорта»;
- 2 — объявление переменной `sport_id` типа как у колонки `"Sport".id`, `%TYPE` предоставляет переменной тип как у переменной или колонки таблицы, которая указана перед %;
- 3 — запрос, который присваивает переменной `sport_id` значение `id` записи таблицы `Sport` с названием, переданным в процедуру через параметр `sport_title`.
- 4 — если такой записи не оказалось;
- 5 — вызывается исключение с кодом `E0001` (пользовательский код);
- 6 — предоставление подсказки по вызванной ошибке;
- 7 — установление текста сообщения об ошибке. `$1` означает первый параметр, переданный в процедуру;
- 8 — динамический запрос, создающий таблицу с названием вида спорта, если она не существует. `%I` означает, что будет подставлен параметр, указанный после описания запроса через запятую. В данном случае — `sport_title`;
- 9 — динамический запрос, удаляющий все записи из таблицы с названием `sport_title`;
- 10 — динамический запрос, вставляющий в таблицу с названием, указанным в первом параметре (`$1`) процедуры (или иначе `sport_title`), записи из таблицы `Sport_Section` с `id`, совпадающим с `sport_id`, полученным на 3 шаге.

Вызовем процедуру для существующей таблицы `Sport_Section` и выведем все строки новой таблицы (см. Рисунок 27). В результате вызова данной процедуры с динамическими запросами будет создана таблица «Атлетика», куда скопируются записи с видом спорта «Атлетика» из таблицы `Sport_Section`.

```
CALL copy_sport('Атлетика');  
select * from public."Атлетика";
```

	Data Output	Explain	Messages	Notifications
	id [PK] integer		title character varying (100)	description text
1		1	Легкая атлетика	dbjvhfdhjvz
2		2	Тяжелая атлетика	bdsjhcgjdhcs

Рисунок 27 - Содержимое таблицы «Атлетика»

Теперь попробуем вызвать процедуру с несуществующим названием. В результате будет выведена ошибка о несуществующем спорте (см. Рисунок).

```
CALL copy_sport('Not_exists_sport');
```

```
ERROR: ОШИБКА: Вид спорта с title=Not_exists_sport не найден!
HINT: Измените название вида спорта(title) или добавьте сперва спорт
CONTEXT: функция PL/pgSQL copy_sport(character varying), строка 7, оператор RAISE

SQL state: E0001
```

Рисунок 28 - Ошибка о несуществующем виде спорта

2.3.14.2. Пример создания динамического запроса при помощи функции

Приведенная ниже функция скопирует первые n записей из таблицы Employee в новую таблицу. Аргументами функции являются название новой таблицы и кол-во строк, которые необходимо скопировать.

```
CREATE OR REPLACE FUNCTION copy_emp(tablename text, rows_count
integer)
RETURNS void AS
$BODY$
BEGIN
    -- динамический запрос, создающий таблицу с названием,
переданным в аргументе tablename, если такая не существует.
    EXECUTE format('CREATE TABLE IF NOT EXISTS %I (
        "id" serial NOT NULL,
        last_name text,
        first_name text,
        patronymic text,
        CONSTRAINT "%I_pkey" PRIMARY KEY ("id"))',
        tablename, tablename);
```

```

-- динамический запрос, удаляющий все записи из этой таблицы
EXECUTE format('delete from %I', tablename);

-- динамический запрос, вставляющий в новую таблицу первые
rows_count строк, полученных из таблицы Employee
EXECUTE format('insert into %I ("id", last_name,
                    first_name, patronymic) select "id", last_name,
                    first_name, patronymic from public."Employee"
                    limit $1', tablename) USING rows_count;

END

$BODY$ LANGUAGE plpgsql;

```

В результате вызова функции будет создана новая таблица, которая появится в списке таблиц (см. Рисунок 29). Также можно вывести содержимое данной таблицы (см. Рисунок 30).

```
SELECT copy_emp('emp_copy', 3);
```

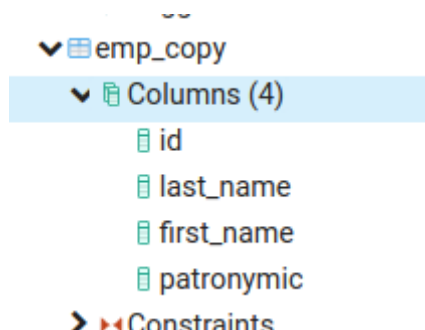


Рисунок 29 - Таблица *emp_copy* в списке таблиц

	Data Output	Explain	Messages	Notifications
	id [PK] integer	last_name text	first_name text	patronymic text
1		1 Степанова	Светлана	Степановна
2		2 Кузнецов	Максим	Максимович
3		3 Игнатов	Игнат	Игнатович

Рисунок 30 - Содержимое таблицы *emp_copy*

2.3.15. Ранжирующие запросы

Ранжирующая (или оконная) функция выполняет вычисления для набора строк, попавшим в один раздел с текущей строкой [19]. В отличие от обычной агрегатной

функции, при использовании оконной функции несколько строк не группируются в одну, а продолжают существовать отдельно.

Синтаксис:

НазваниеФункции () OVER ([PARTITION BY столбцы группировки] ORDER BY столбец сортировки)

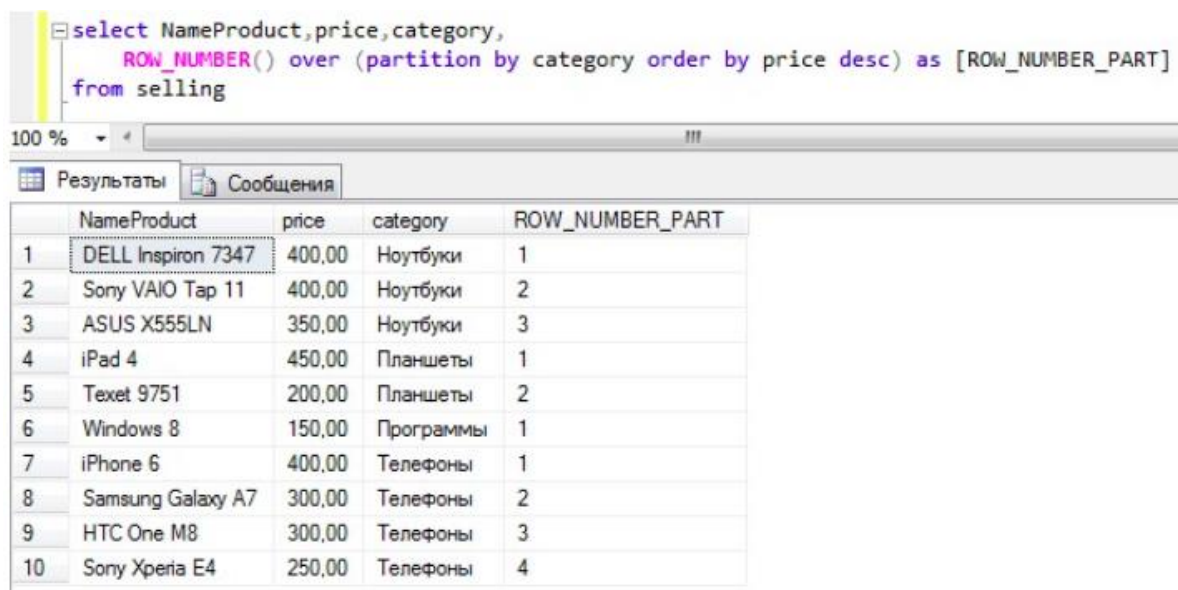
Ранжирующие функции используют:

- OVER - определяет, как именно нужно разделить строки запроса для обработки оконной функцией.
- PARTITION BY - указывает, что строки нужно разделить по группам или разделам.

2.3.15.1. Функция ROW_NUMBER

ROW_NUMBER – функция нумерации, возвращает просто номер строки.

Например, для таблицы Прайс-листа (с полями Название, Цена, Категория) запрос сгруппирует товары по категории и выведет порядковый номер товара в данной категории (см. Рисунок 31).



```
select NameProduct, price, category,
       ROW_NUMBER() over (partition by category order by price desc) as [ROW_NUMBER_PART]
from selling
```

	NameProduct	price	category	ROW_NUMBER_PART
1	DELL Inspiron 7347	400,00	Ноутбуки	1
2	Sony VAIO Tap 11	400,00	Ноутбуки	2
3	ASUS X555LN	350,00	Ноутбуки	3
4	iPad 4	450,00	Планшеты	1
5	Texet 9751	200,00	Планшеты	2
6	Windows 8	150,00	Программы	1
7	iPhone 6	400,00	Телефоны	1
8	Samsung Galaxy A7	300,00	Телефоны	2
9	HTC One M8	300,00	Телефоны	3
10	Sony Xperia E4	250,00	Телефоны	4

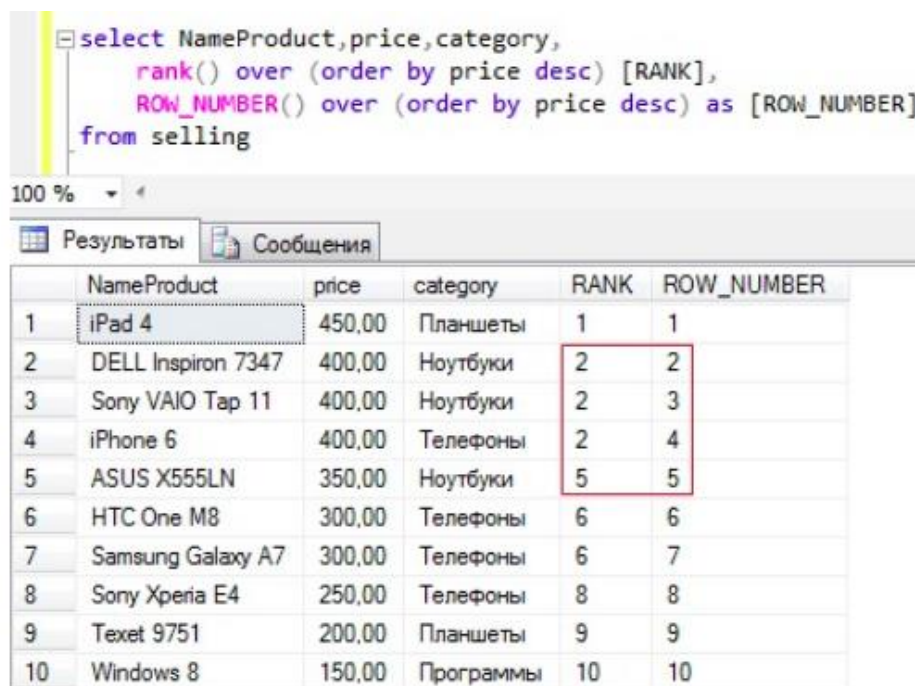
Рисунок 31 - Применение функции ROW_NUMBER

2.3.15.2. Функция RANK

RANK – возвращает ранг каждой строки.

В отличие от ROW_NUMBER() в случае нахождения одинаковых значений функция возвращает одинаковый ранг с пропуском следующего.

Например, данный запрос (см. Рисунок 32) выводит ранг и номер записи в таблице Прайс-листа. ROW_NUMBER сортирует по цене и выводит просто номер строки. RANK возвращает ранг 1-го товара с данной ценой для всех товаров с такой же ценой с пропуском следующего.



```

select NameProduct, price, category,
       rank() over (order by price desc) [RANK],
       ROW_NUMBER() over (order by price desc) as [ROW_NUMBER]
from selling

```

	NameProduct	price	category	RANK	ROW_NUMBER
1	iPad 4	450,00	Планшеты	1	1
2	DELL Inspiron 7347	400,00	Ноутбуки	2	2
3	Sony VAIO Tap 11	400,00	Ноутбуки	2	3
4	iPhone 6	400,00	Телефоны	2	4
5	ASUS X555LN	350,00	Ноутбуки	5	5
6	HTC One M8	300,00	Телефоны	6	6
7	Samsung Galaxy A7	300,00	Телефоны	6	7
8	Sony Xperia E4	250,00	Телефоны	8	8
9	Texet 9751	200,00	Планшеты	9	9
10	Windows 8	150,00	Программы	10	10

Рисунок 32 - Применение функции RANK

2.3.15.3. Функция DENSE_RANK

DENSE_RANK — возвращает ранг каждой строки, но, в отличие от RANK, в случае нахождения одинаковых значений возвращает ранг без пропуска следующего.

Например, данный запрос выводит ранги и номер записи в таблице Прайс-листа (см. Рисунок 33). ROW_NUMBER сортирует по цене и выводит просто номер строки. RANK возвращает ранг 1-го товара с данной ценой для всех товаров с такой же ценой с пропуском следующего. DENSE_RANK возвращает ранг 1-го товара с данной ценой для всех товаров с такой же ценой без пропуска следующего.

```

select NameProduct, price, category,
       rank() over (order by price desc) [RANK],
       DENSE_RANK () over (order by price desc) [DENSE_RANK],
       ROW_NUMBER() over (order by price desc) as [ROW_NUMBER]
from selling

```

100 %

Результаты Сообщения

	NameProduct	price	category	RANK	DENSE_RANK	ROW_NUMBER
1	iPad 4	450,00	Планшеты	1	1	1
2	DELL Inspiron 7347	400,00	Ноутбуки	2	2	2
3	Sony VAIO Tap 11	400,00	Ноутбуки	2	2	3
4	iPhone 6	400,00	Телефоны	2	2	4
5	ASUS X555LN	350,00	Ноутбуки	5	3	5
6	HTC One M8	300,00	Телефоны	6	4	6
7	Samsung Galaxy A7	300,00	Телефоны	6	4	7
8	Sony Xperia E4	250,00	Телефоны	8	5	8
9	Texet 9751	200,00	Планшеты	9	6	9
10	Windows 8	150,00	Программы	10	7	10

Рисунок 33 - Применение функции DENSE RANK

2.3.15.4. Функция NTILE

NTILE — делит результирующий набор на заданное количество групп по определенному столбцу.

Пример. Данный запрос (см. Рисунок 34) делит записи в таблице Прайс-листа на 3 группы сортируя по цене.

```

select NameProduct, price, category,
       NTILE(3) over (order by price desc) [NTILE]
from selling

```

110 %

Результаты Сообщения

	NameProduct	price	category	NTILE
1	iPad 4	450,00	Планшеты	1
2	DELL Inspiron 7347	400,00	Ноутбуки	1
3	Sony VAIO Tap 11	400,00	Ноутбуки	1
4	iPhone 6	400,00	Телефоны	1
5	ASUS X555LN	350,00	Ноутбуки	2
6	HTC One M8	300,00	Телефоны	2
7	Samsung Galaxy A7	300,00	Телефоны	2
8	Sony Xperia E4	250,00	Телефоны	3
9	Texet 9751	200,00	Планшеты	3
10	Windows 8	150,00	Программы	3

Рисунок 34 - Применение функции NTILE

2.3.16. Функция-агрегат

Агрегатная функция вызывается для каждой строки таблицы по очереди и в конечном итоге обрабатывает их все [20]. Между вызовами ей требуется сохранять внутреннее состояние, определяющее контекст ее выполнения. В конце работы она должна вернуть итоговое значение.

Чтобы определить агрегатную функцию, необходимо выбрать:

1. **Тип данных** для значения состояния;
2. **Начальное значение** состояния;
3. **Функцию перехода** состояния - принимает предыдущее значение состояния и входное агрегируемое значение для текущей строки и возвращает новое значение состояния;
4. **Функцию завершения** — для случая, если ожидаемый результат агрегатной функции отличается от данных, которые сохраняются в изменяющемся значении состояния. Принимает конечное значение состояния и возвращает то, что она хочет вернуть в виде результата агрегирования.

Синтаксис:

```

CREATE AGGREGATE имя ( [ режим_аргумента ] [ имя_аргумента ]
тип_данных_аргумента [ , ... ] ) (
    SFUNC = функция_состояния,

```

```

        STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , INITCOND = начальное_условие ]
)

```

В приведенном ниже примере создается функция-агрегат, которая прибавляет 10 к максимальному значению (типа `int`).

Перед созданием самой агрегатной функции необходимо создать 2 функции:

1 — определяет большее число из двух.

2 — добавляет 10.

```

CREATE FUNCTION greater_int (int, int)
RETURNS int LANGUAGE SQL
AS $$
    SELECT
        CASE WHEN $1 < $2 THEN $2 ELSE $1
        END
    $$;

```

```

CREATE FUNCTION int_plus_10 (int)
RETURNS int LANGUAGE SQL
AS $$
    SELECT $1+ 10;
    $$;

```

Далее создаем агрегат.

SFUNC — функция состояния, которая вызывается для всех строк. В нашем случае это функция сравнения 2 значений.

FINALFUNC будет запущена только один раз — в конце. Здесь это прибавление 10 к максимальному значению.

STYPE — тип данных состояния – `integer`.

INITCOND — начальное состояние зададим в 0.

```

CREATE AGGREGATE incremented_max (int) (
    SFUNC = greater_int,
    FINALFUNC = int_plus_10,

```

```

        STYPE = integer,
        INITCOND = 0
    );

```

Вызов функции. Результатом подзапроса (см. Рисунок 35) будет столбец *v* со значениями 3, 10, 12, 5.

```

SELECT incremented_max(v)
FROM (
    SELECT 3 AS v
    UNION SELECT 10
    UNION SELECT 12
    UNION SELECT 5) ALIAS

```

Data Output		Explain	Mes:
	incremented_max		
	integer		
1			22

Рисунок 35 - Результат выполнения функции-агрегата

2.3.17. DML-триггеры

Триггеры PL/SQL уровня команд DML (или просто триггеры DML) активизируются после вставки, обновления или удаления строк конкретной таблиц.

Триггер можно настроить так, чтобы он срабатывал до операции со строкой (до проверки ограничений и попытки выполнить INSERT, UPDATE или DELETE) или после её завершения (после проверки ограничений и выполнения INSERT, UPDATE или DELETE), либо вместо операции (при добавлении, изменении и удалении строк в представлении). Если триггер срабатывает до или вместо события, он может пропустить операцию с текущей строкой, либо изменить добавляемую строку (только для операций INSERT и UPDATE). Если триггер срабатывает после события, он «видит» все изменения, включая результат действия других триггеров.

Синтаксис [21]:

```

CREATE [OR REPLACE] TRIGGER имя_триггера
    {BEFORE | AFTER}
    {INSERT | DELETE | UPDATE | UPDATE OF список_столбцов } ON
имя_таблицы
    [FOR EACH ROW]

```

```
[WHEN (...)]  
[DECLARE ... ]  
BEGIN  
    ...исполняемые команды...  
[EXCEPTION ... ]  
END [имя_триггера];
```

Условно триггеры можно разделить на следующие виды [22]:

- Триггер BEFORE. Вызывается до внесения каких-либо изменений (например, BEFORE INSERT).
- Триггер AFTER. Выполняется для отдельной команды SQL, которая может обрабатывать одну или более записей базы данных (например, AFTER UPDATE).
- Триггер уровня команды [21]. Выполняется для команды SQL в целом (которая может обрабатывать одну или несколько строк базы данных).
- Триггер уровня записи. Выполняется для отдельной записи, обрабатываемой командой SQL. Если, предположим, таблица books содержит 1000 строк, то следующая команда UPDATE модифицирует все эти строки.

```
UPDATE books SET title = UPPER (title);
```

И если для таблицы определен триггер уровня записи, он будет выполнен 1000 раз.

Триггер с пометкой FOR EACH ROW вызывается один раз для каждой строки, изменяемой в процессе операции. Например, операция DELETE, удаляющая 10 строк, приведёт к срабатыванию всех триггеров ON DELETE в целевом отношении 10 раз подряд, по одному разу для каждой удаляемой строки.

В определении триггера можно указать логическое условие WHEN, которое будет проверяться, чтобы посмотреть, нужно ли запускать триггер. В триггерах уровня строки в условии WHEN можно проверять старые и/или новые значения столбцов строки. (В триггерах уровня оператора также можно использовать условие WHEN, хотя в этом случае это не так полезно.) В триггерах BEFORE условие WHEN вычисляется непосредственно перед тем, как триггерная функция будет выполнена, поэтому использование WHEN существенно не отличается от выполнения той же проверки в самом начале триггерной функции. Однако в триггерах AFTER условие WHEN

вычисляется сразу после обновления строки и от этого зависит, будет ли поставлено в очередь событие запуска триггера в конце оператора или нет. Поэтому, когда условие WHEN в триггере AFTER не возвращает истину, не требуется ни постановка события в очередь, ни повторная выборка этой строки в конце оператора. Это может существенно ускорить работу операторов, изменяющих большое количество строк, с триггером, который должен сработать только для нескольких. [23].

В качестве примера создадим триггер, который после добавления записи в таблицу employee будет создавать запись в таблице логов с указанием времени события вставки записи в таблицу employee.

Зададим функцию, которая вставит запись в таблицу логов.

В данной функции используется специальная переменная NEW, которая создается, когда срабатывает триггер. Переменная содержит новую строку базы данных для команд INSERT/UPDATE в триггерах уровня строки.

```
CREATE OR REPLACE FUNCTION rec_insert()
  RETURNS trigger AS
  $$
  BEGIN
      INSERT INTO emp_log(emp_id, salary, edittime)
      VALUES(NEW.employee_id, NEW.salary, current_date);
      RETURN NEW;
  END;
  $$
  LANGUAGE 'plpgsql';
```

Опишем сам триггер, который после вставки строки в таблицу employee вызовет вышеописанную функцию rec_insert.

```
CREATE TRIGGER ins_same_rec
  AFTER INSERT
  ON employee
  FOR EACH ROW
  EXECUTE PROCEDURE rec_insert();
```

Рисунок 36 демонстрирует данные таблицы employee.

employee_id	first_name	last_name	job_id	salary	commission_pct
100	Steven	King	AD_PRES	24000.00	0.00
101	Neena	Kochhar	AD_VP	17000.00	0.00
102	Lex	De Haan	AD_VP	17000.00	0.00
103	Alexander	Hunold	IT_PROG	9000.00	0.00
104	Bruce	Ernst	IT_PROG	6000.00	0.00
105	David	Austin	IT_PROG	4800.00	0.00
106	Valli	Pataballa	IT_PROG	4800.00	0.00
107	Diana	Lorentz	IT_PROG	4200.00	0.00
108	Nancy	Greenberg	FI_MGR	12000.00	0.00
109	Daniel	Faviet	FI_ACCOUNT	9000.00	0.00
110	John	Chen	FI_ACCOUNT	8200.00	0.00
111	Ismael	Sciarra	FI_ACCOUNT	7700.00	0.00
112	Jose Manuel	Urman	FI_ACCOUNT	7800.00	0.00
236	RABI	CHANDRA	AD_VP	15000.00	0.50

Рисунок 36 - Данные таблицы employee

Рисунок 37 демонстрирует данные таблицы логов.

```
postgres=# SELECT * FROM emp_log;
```

emp_id	salary	edittime
100	24000	2011-01-15
101	17000	2010-01-12
102	17000	2010-09-22
103	9000	2011-06-21
104	6000	2012-07-05
105	4800	2011-06-02
236	15000	2014-09-15

Рисунок 37 - Данные таблицы логов

2.3.18. DDL-триггеры

Триггеры DDL активируются в ответ на различные события языка DDL. Эти события в основном соответствуют инструкциям SQL/PSM, которые начинаются с ключевых слов CREATE, ALTER, DROP, GRANT, DENY, REVOKE или UPDATE STATISTICS. Системные хранимые процедуры, выполняющие операции, подобные операциям DDL, также могут запускать триггеры DDL.

Триггеры DDL срабатывают в ответ на событие SQL/PSM, обработанное текущей базой данных или текущим сервером. Область триггера зависит от события. Например, триггер DDL, созданный для срабатывания на событие CREATE TABLE, может срабатывать каждый раз, когда в базе данных или в экземпляре сервера возникает событие CREATE_TABLE. Триггер DDL, созданный для запуска в ответ на событие

CREATE_LOGIN, может выполнять это только при возникновении события CREATE_LOGIN в экземпляре сервера.

Используйте триггеры DDL, если хотите сделать следующее.

- Предотвращать внесение определенных изменений в схему базы данных.
- Настроить выполнение в базе данных некоторых действий в ответ на изменения в схеме базы данных.
- Записывать изменения или события схемы базы данных.

Синтаксис [24]:

```
CREATE EVENT TRIGGER name
ON event
[ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
EXECUTE PROCEDURE function_name()
```

В следующем примере триггер DDL safety будет срабатывать каждый раз, когда в базе данных будет выполняться инструкция DROP_TABLE или происходить событие ALTER_TABLE. Триггер будет выводить предупреждение и откатывать транзакцию, аннулируя все изменения.

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
    PRINT 'You must disable Trigger "safety" to drop or alter tables!'
    ROLLBACK;
```

3. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каковы возможности программирования на стороне сервера с применением SQL\PSM?
2. Хранимые процедуры и функции (скалярные, in-line). Их синтаксис и способы вызова. Передача параметров. Параметры по умолчанию.
3. Конструкции языка SQL-PSM: условие, перехват исключений, создание исключений, присваивание переменных.
4. Использование курсора.
5. Синтаксис и процесс выполнения рекурсивных запросов.
6. Синтаксис и назначение ранжирующих функций.
7. Постреляционные возможности языка SQL.
8. Как создать, собрать, присоединить и вызвать функцию агрегат? Каковы ее методы?
9. Что такое DML триггер, как и когда он запускается и как его создать? Как из триггера определить изменяемые данные?
10. Что такое динамические запросы? Как их выполнять?
11. Что такое обновляемое представление? Каковы его свойства? Как необновляемое представление сделать обновляемым?
12. Что такое DDL триггер, как и когда он запускается и как его создать? Как из триггера определить возникшее событие и его параметры?

4. СПИСОК ИСТОЧНИКОВ

1. Виноградов В.И., Виноградова М.В. Постреляционные модели данных и языки запросов: Учебное пособие. — М.: Изд-во МГТУ им. Н.Э. Баумана, 2017. — 100с. - ISBN 978-5-7038-4283-6.
2. PostgreSQL 14.2 Documentation. — Текст. Изображение: электронные // PostgreSQL : [сайт]. — URL: <https://www.postgresql.org/docs/14/index.html> (дата обращения: 26.04.2022)
3. pgAdmin 4 6.5 documentation. — Текст. Изображение: электронные // pgAdmin - PostgreSQL Tools : [сайт]. — URL: <https://www.pgadmin.org/docs/pgadmin4/6.5/index.html> (дата обращения: 26.04.2022)
4. PostgreSQL: Документация: 14: Глава 43. PL/pgSQL — процедурный язык SQL. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/plpgsql> (дата обращения: 26.04.2022)
5. PostgreSQL: Документация: 14: 43.3. Объявления. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/plpgsql-declarations> (дата обращения: 26.04.2022)
6. PostgreSQL: Документация: 14: 43.6. Управляющие структуры. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/plpgsql-control-structures> (дата обращения: 26.04.2022)
7. Основы T-SQL и примеры — функции (UDF), триггеры, процедуры, курсоры, циклы. — Текст. Изображение : электронные // Авторский сайт ИТ-консультанта : [сайт]. — URL: <https://ivan-shamaev.ru/t-sql-fundamentals-and-examples/> (дата обращения: 26.04.2022)
8. PostgreSQL: Документация: 14: CREATE FUNCTION. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/sql-createfunction> (дата обращения: 26.04.2022)

9. PostgreSQL: Документация: 14: 38.5. Функции на языке запросов (SQL). – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/xfunc-sql> (дата обращения: 26.04.2022)
10. PostgreSQL: Документация: 14: 43.5.4. Выполнение динамически формируемых команд. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/plpgsql-statements#PLPGSQL-STATEMENTS-EXECUTING-DYN> (дата обращения: 26.04.2022)
11. PostgreSQL: Документация: 14: 9.4. Строковые функции и операторы. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/functions-string> (дата обращения: 26.04.2022)
12. PostgreSQL: Документация: 14: CREATE PROCEDURE. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/sql-createprocedure> (дата обращения: 26.04.2022)
13. PostgreSQL: Документация: 14: 43.9. Сообщения и ошибки. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/plpgsql-errors-and-messages> (дата обращения: 26.04.2022)
14. PostgreSQL: Документация: 14: Приложение А. Коды ошибок PostgreSQL. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/errcodes-appendix> (дата обращения: 26.04.2022)
15. PostgreSQL: Документация: 14: 7.6. LIMIT и OFFSET. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/queries-limit> (дата обращения: 26.04.2022)
16. PostgreSQL: Документация: 14: 6.4. Возврат данных из изменённых строк. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт].

- URL: <https://postgrespro.ru/docs/postgresql/14/dml-returning> (дата обращения: 26.04.2022)
17. PostgreSQL: Документация: 14: 7.8. Запросы WITH (Общие табличные выражения). – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/queries-with> (дата обращения: 26.04.2022)
18. PostgreSQL: Документация: 14: 36.5. Динамический SQL. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/ecpg-dynamic> (дата обращения: 26.04.2022)
19. PostgreSQL: Документация: 14: 9.22. Оконные функции. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/functions-window> (дата обращения: 26.04.2022)
20. PostgreSQL: Документация: 14: 9.21. Агрегатные функции. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/functions-aggregate> (дата обращения: 26.04.2022)
21. PostgreSQL: Документация: 14: CREATE TRIGGER. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/sql-createtrigger> (дата обращения: 26.04.2022)
22. Триггеры PL/SQL уровня команд DML на примерах. – Текст. Изображение : электронные // Patches IT Community: [сайт]. – URL: <https://oracle-patches.com/db/sql/триггеры-pl-sql-уровня-команд-dml-на-примерах> (дата обращения: 26.04.2022)
23. PostgreSQL: Документация: 14: 39.1. Обзор механизма работы триггеров. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL: <https://postgrespro.ru/docs/postgresql/14/trigger-definition> (дата обращения: 26.04.2022)
24. PostgreSQL: Документация: 14: CREATE EVENT TRIGGER. – Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. – URL:

<https://postgrespro.ru/docs/postgresql/14/sql-createeventtrigger> (дата обращения:
26.04.2022)