

Лабораторная работа «Постреляционное расширение языка SQL на примере PostgreSQL»
по дисциплине «Постреляционные базы данных»

Цель работы:

- 1.** Изучить постреляционные возможности языка SQL.
- 2.** Освоить языки и технологии SQL\PSM.
- 3.** Получить навыки программирования на стороне сервера.

Средства выполнения:

1. СУБД PostgreSQL
2. PgAdmin

Время выполнения:

Время выполнения лабораторной работы 4 часа.

Оглавление

Цель работы:	1
Средства выполнения:	1
Время выполнения:.....	1
Время выполнения лабораторной работы 4 часа.	1
Пункты задания для выполнения:	3
Задание 1. Базовая часть (удовлетворительно)	3
Задание 2. Расширенная часть (хорошо)	3
Дополнительное задание (отлично):	4
Теоретическая часть:	4
Подключение PostgreSQL с использованием pgAdmin.	4
Создание базы данных в среде PgAdmin.	6
SQL/PSM. Конструкции языка SQL-PSM.	11
Объявление переменных	11
Присвоение значений переменным:	11
Условные операторы.	13
Циклы — LOOP.	14
Программирование функций с применением SQL/PSM.	15
Создание и выполнение определяемых пользователем функций.	16
Скалярные функции.	17
Возвращающие табличное значение функции	18
Хранимые процедуры.....	20
Структура хранимой процедуры.	20
Перехват и вызов исключений.	21
Извлечение части записей и результатов запросов на изменение данных	22
LIMIT	22
RETURNING	23
Рекурсивный запрос.	24
Динамические запросы.	26
Ранжирующие запросы.	29
ROW_NUMBER	30
RANK	30
DENSE_RANK	31
NTILE	31
Функция — агрегат	32
DML — триггеры.	33
DDL триггер.	35
Вопросы для самопроверки:	36
В отчет:	36
Литература:	36

Пункты задания для выполнения:

Задание 1. Базовая часть (удовлетворительно)

1.1. Создание и заполнение таблицы

Через PgAdmin соединиться с PostgreSQL и создать базу данных. В БД создать таблицы по теме своей курсовой работе (или использовать таблицы, созданные ранее), например,

"Clinic"(Клиника). Содержит поля:

num - номер – целое - ключ,

city - город - строковое(varchar), not null

addr - адрес - строковое(varchar).

"MedPath"(Направления на лечение), которая содержит свойства:

num – номер направления — целое PK,

msource - название клиники, давшей направление— строковое (not null) FK,

mdest - название клиники, куда направляют— строковое (not null) FK,

diagnoz – диагноз,

fio – ФИО пациента,

age – возраст пациента (≥ 18),

dt - дата,

ame - цель (консультация, операция, обследование и т.д.),

prevnum – номер предыдущего направления — целое FK.

Открыть таблицы на редактирование и заполнить тестовыми данными.

1.2. Программирование функций с применением SQL\PSM

1) Создать **скалярную функцию**, например, **minAge**(клиника), которая возвращает минимальный возраст пациентов указанной клиники. Вызвать функцию из окна запроса.

2) Создать **табличную функцию**, например, **patients**(клиника), которая возвращает ФИО, возраст и диагнозы, пациентов в указанной клинике. Вызвать функцию из окна запроса.

3) Создать **хранимую процедуру** (с перехватом и вызовом исключений, проверки условия и выполнением запросов к таблицам),

например, **Send1medCons** (клиника, фио, диагноз, возраст = 18), которая проверяет наличие клиники. Если ее не существует, то вызывает исключение. Иначе создает направление в ГКБ-1 с текущей датой на консультацию. Перехват исключений на вставку.

Вызвать процедуру из окна запроса. Проверить перехват и создание исключений.

Задание 2. Расширенная часть (хорошо)

2.1. Извлечение части записей и результатов запросов на изменение данных

Продemonстрировать выполнение запроса на получение первых 3-х записей из результата (**limit**).

Продemonстрировать выполнение запроса на добавление/изменение данных с отображением измененных строк (**returning**).

2.2. Выполнение рекурсивных запросов

Продemonстрировать выполнение рекурсивного запроса, например, перечень клиник, где были пациенты ГКБ 67.

2.3. Создание динамических запросов

Создать хранимую процедуру с динамическим запросом, например,

copy_patients (клиника, диапазон), которая выполняет:

- ▲ Если таблица с именем клиники (из входного параметра) не существует, то создать ее с полями (fio,age,diagnos,ame);
 - ▲ Очистить таблицу и добавить в нее пациентов клиники для указанного диапазона.
- Вызвать процедуру из окна запроса.

Дополнительное задание (отлично):

3.1. Выполнение ранжирующих запросов

Продемонстрировать выполнение функций **row_number()**, **Rank()**, **dense_rank()**, **ntile(4)**.

3.2. Программирование функции - агрегата

Создать функцию-агрегат, например, **MaxMin()**, которая применяется к целым числам и возвращает соотношение максимальных и минимальных значений. Проверить работоспособность функции из окна запроса

3.3. Создание DDL триггера

Создать DDL триггер (например, на удаление таблиц, начинающихся с SYS), который пишет в журнал сведения о событии и делает откат транзакции. Проверить работу триггера.

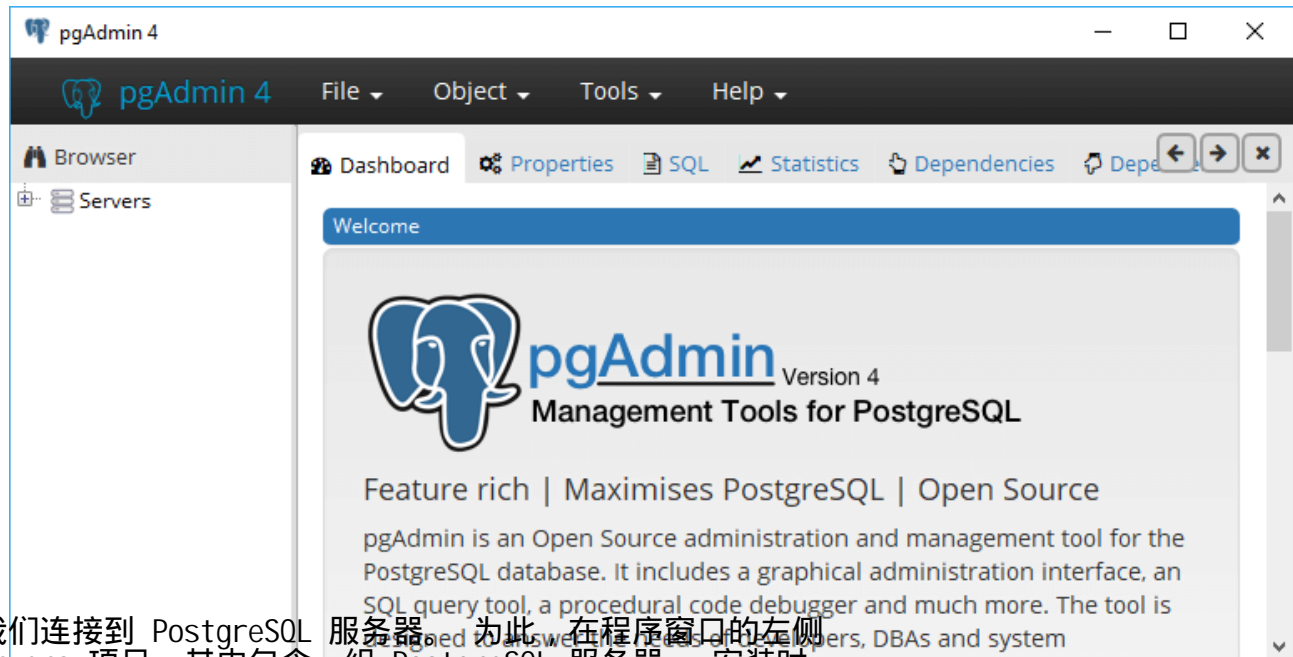
Теоретическая часть:

Подключение PostgreSQL с использованием pgAdmin.

Установка PostgreSQL: <https://www.postgresql.org/download/>

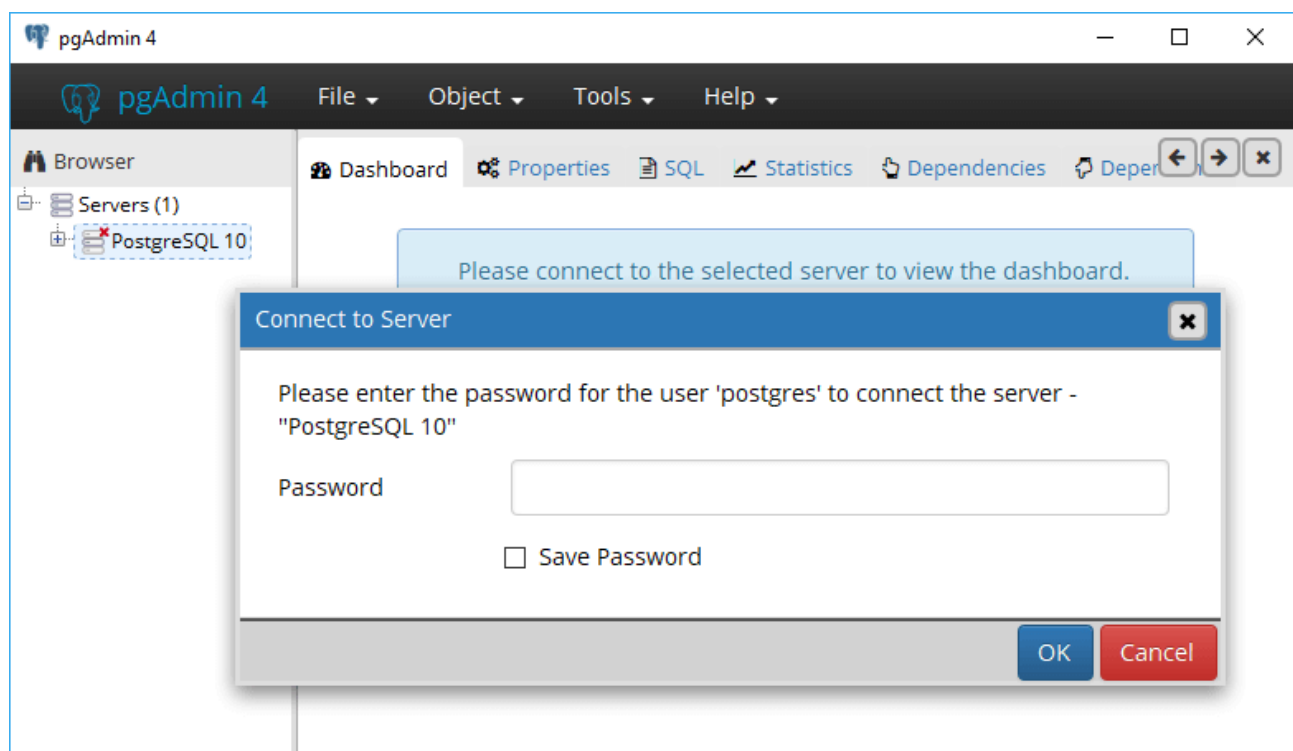
Установка pgAdmin: <https://www.pgadmin.org/download/>

После запуска нам откроется следующая программа:



现在让我们连接到 PostgreSQL 服务器。为此，在程序窗口的左侧展开 Servers 项目，其中包含一组 PostgreSQL 服务器。安装时最新版本安装服务器，默认名称为

Теперь подключимся к серверу PostgreSQL. Для этого в левой части окна программы раскроем пункт Servers, который содержит набор серверов PostgreSQL. При установке последней версии устанавливается сервер, который по умолчанию имеет название



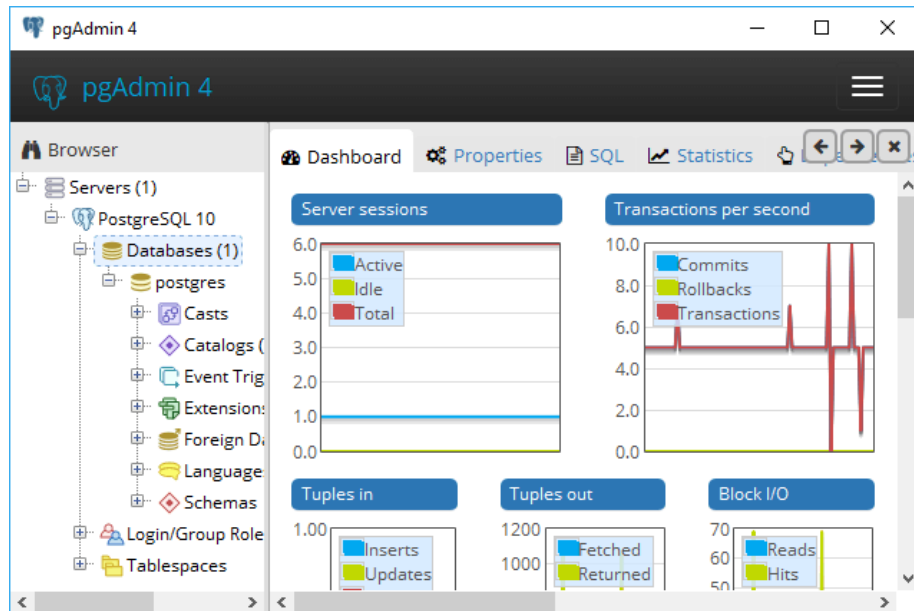
PostgreSQL 10. Нажмем на этот пункт, и нам отобразится окно для ввода пароля:

PostgreSQL 10. 点击此项，我们会看到一个输入密码的窗口：

Здесь необходимо ввести пароль для суперпользователя postgres, который был задан при установке PostgreSQL.

После успешного логина нам откроется содержимое сервера:

这里需要输入 postgres 超级用户的密码，这个密码是在安装 PostgreSQL 登录成功后，服务器的内容会向我们开放

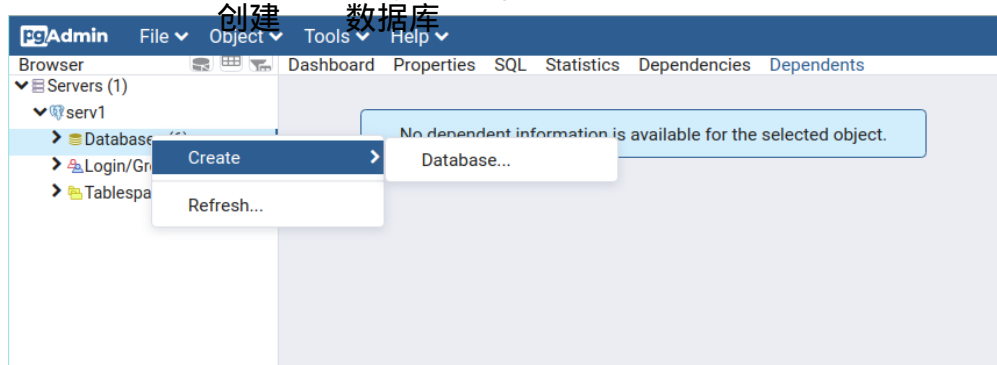


当您登录时，您将拥有一个名为“Postgres”的空默认数据库。

При входе в систему, вы будете иметь пустую БД по умолчанию с названием "Postgres".

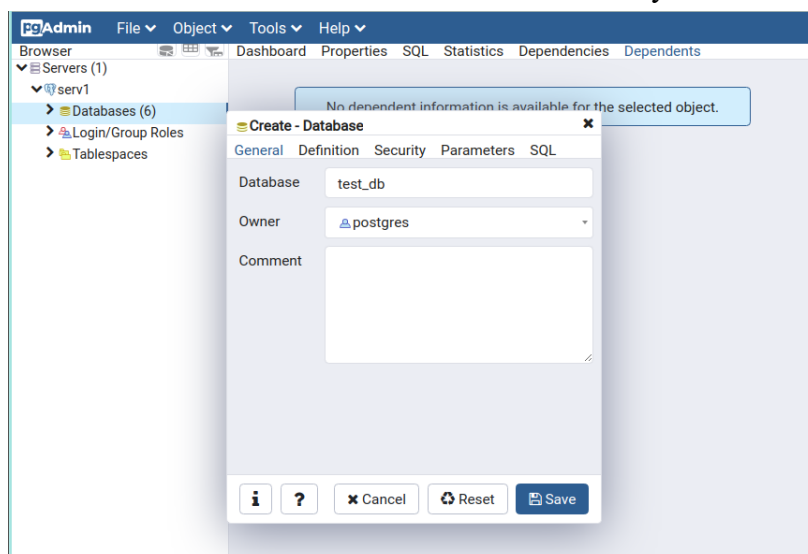
Создание базы данных в среде PgAdmin. 在 PgAdmin 环境中创建数据库。

Выберите сервер и кликните правой кнопкой мыши на пункт Databases (Базы Данных)
→ Create → Database 选择一个服务器并右键单击数据库



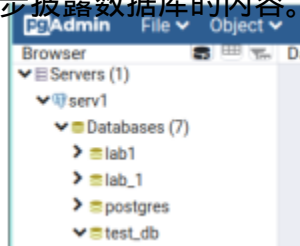
在数据库字段中输入数据库的名称。 按“保存”按钮。

Введите название базы данных в поле Database. Нажать кнопку «Save».



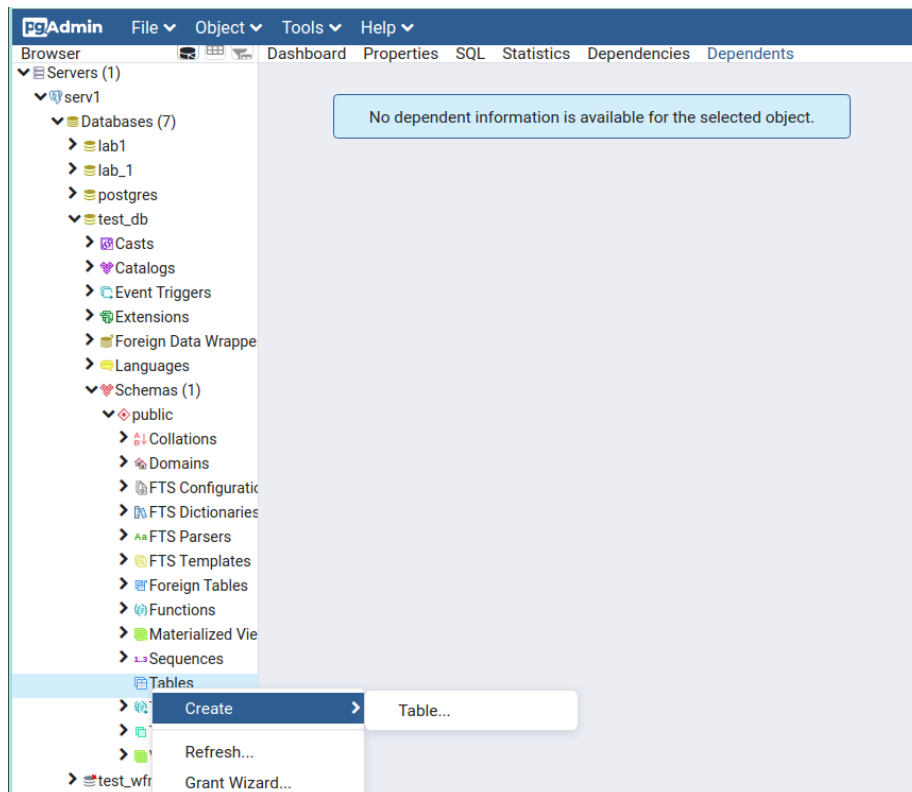
Чтобы просмотреть базу данных, необходимо нажать стрелку рядом с Databases, чтобы
要查看数据库，您必须单击数据库旁边的箭头以

раскрыть список баз данных. Аналогично далее можно раскрывать содержимое баз данных. 展开数据库列表。 同样，您可以进一步披露数据库的内容。

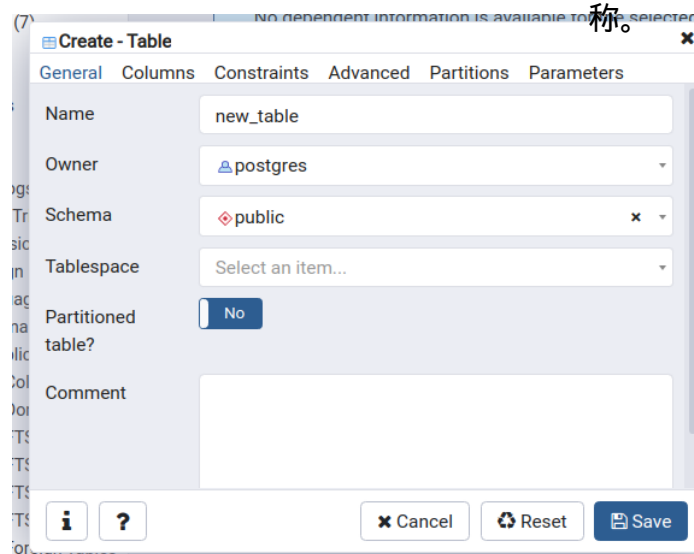


要创建表，请打开数据库 您的数据库 模式。
右键单击表并选择创建 表。

Для создания таблицы необходимо раскрыть Databases → Ваша база данных → Schemas.
Нажать правой кнопкой мыши на Tables (Таблицы) и выбрать Create → Table.



В окне создания таблицы ввести название в поле Name. 在表创建窗口中，在名称字段中输入名称。



Колонки в таблицах можно создать при создании таблицы. Для добавления колонки
创建表时可以创建表中的列。 添加列

надо перейти во вкладку Columns, нажать на знак + и ввести данные о колонке. Также при необходимости можно выбрать таблицу, от которой наследуется данная таблицы. Сохранить таблицу. Нажать кнопку «Save».

您需要转到“列”选项卡，单击+号并输入有关该列的数据。也在
如有必要，您可以选择继承此表的表。保存
桌子。按“保存”按钮。

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key
id	serial			Yes	Yes

第二种添加列的方法：

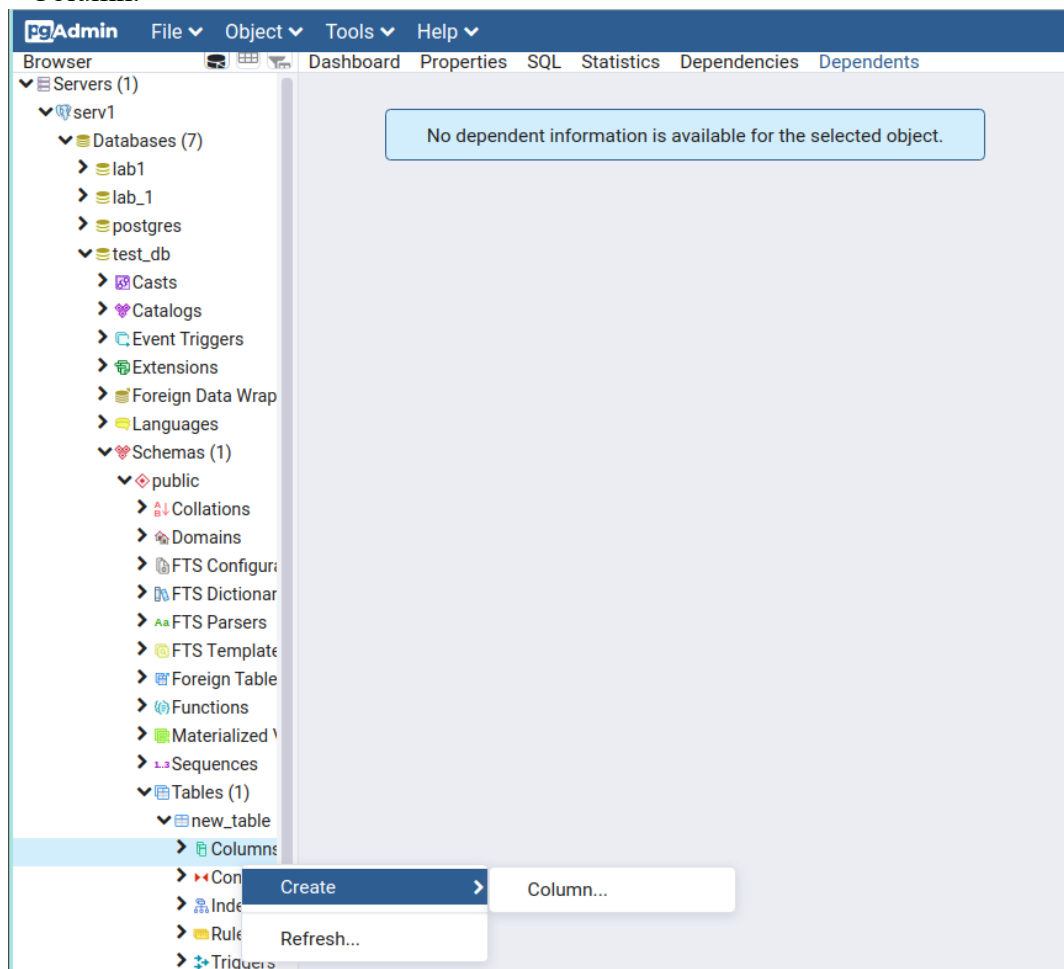
展开所需的表，右键单击列，选择

创建 列

Второй способ добавления колонок:

Раскрыть необходимую таблицу, нажать правой кнопкой мыши на Columns, выбрать

Create → Column.



В окне создания колонки ввести название в поле Name.
在列创建窗口中，在名称字段中输入名称。

Create - Column

General Definition Constraints Variables Security

Name: name

Comment:

Column type cannot be empty.

Buttons: i, ?, Cancel, Reset, Save

Во вкладке Definition выбрать тип данных и задать длину поля.

在定义选项卡中，选择数据类型并设置字段长度

Create - Column

General Definition Constraints Variables Security

Data type: character varying

Length/Precision: 255

Scale:

Collation: Select an item...

Buttons: i, ?, Cancel, Reset, Save

Во вкладке Constraints при необходимости установить ограничения. Сохранить столбец. Нажать кнопку «Save».

Create - Column

General Definition Constraints Variables Security

Default: 'default_name'

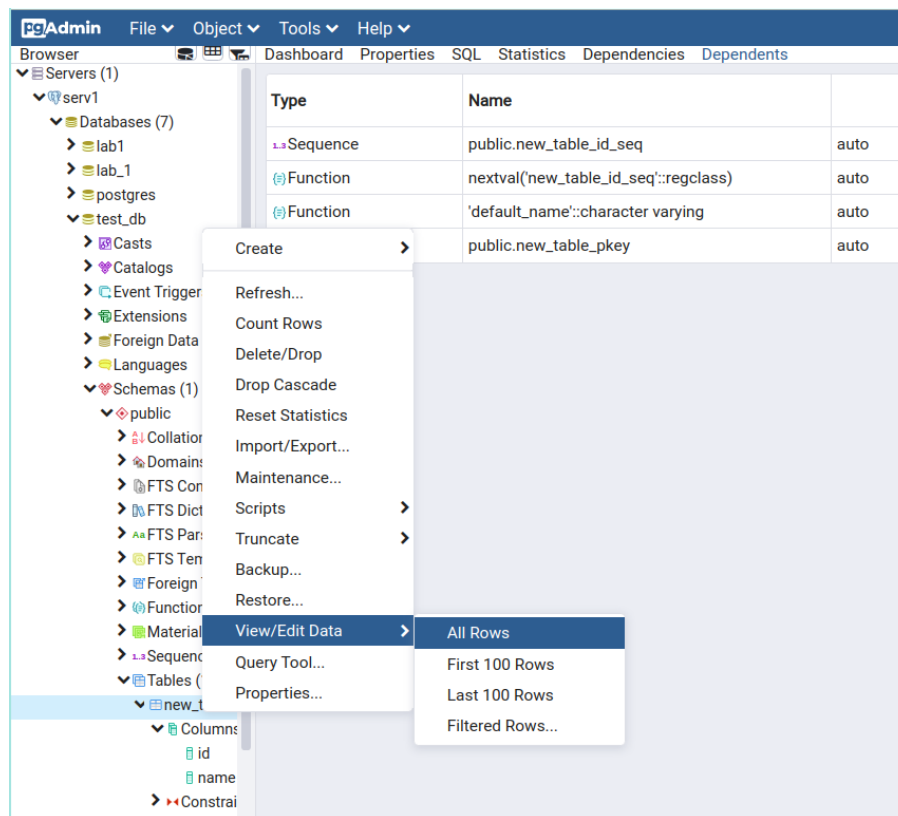
Not NULL?: No

Type: NONE IDENTITY

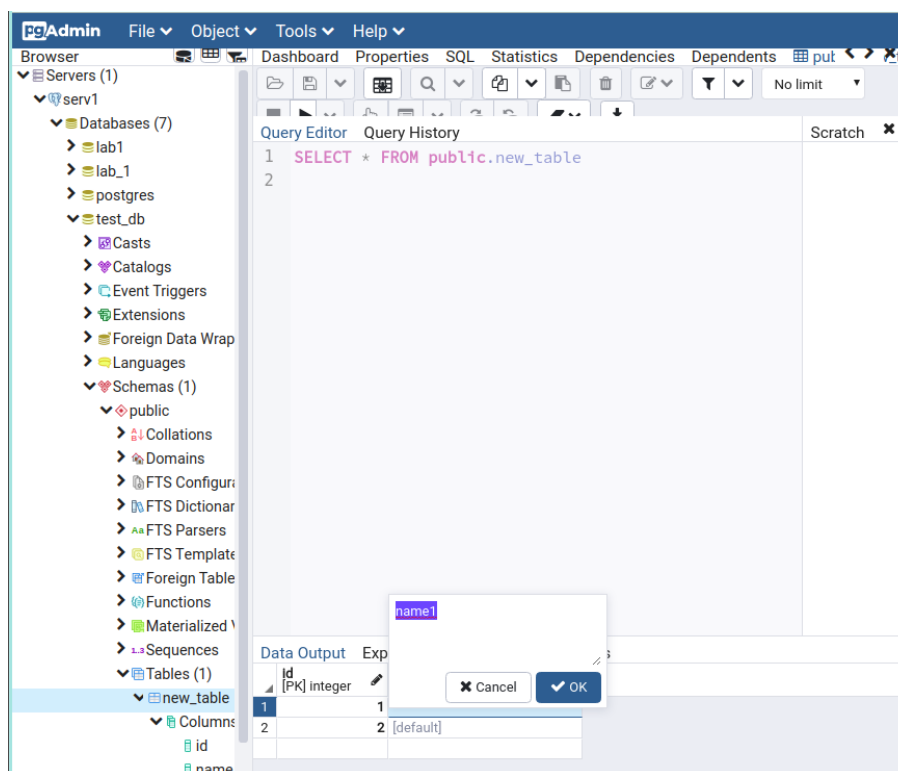
Buttons: i, ?, Cancel, Reset, Save

Для редактирования данных нажать на необходимую таблицу правой кнопкой мыши и
要编辑数据，请右键单击所需的表并

выбрать пункт View/Edit Data и необходимые строки. 选择查看/编辑数据和所需的行



В окне редактирования данных ввести данные в Data Output и сохранить, нажав либо F6, либо кнопку:
在数据编辑窗口中，将数据输入数据输出并按 F6 或按钮保存：



SQL\PSM. SQL-PSM 语言的构造。SQL/PSM (Persistent Stored Modules) 是由美国开发的标准美国国家标准协会 (ANSI) 作为 SQL 的扩展。标准主要用过程语言定义一个扩展：解析声明变量、执行逻辑控制、循环和条件 PSM 允许我们将过程存储为数据库模式的元素。PSM 混合常规语句 (if、while 等) 和 SQL。它允许我们做我们不能做的事情仅在 SQL 中执行。

SQL\PSM. Конструкции языка 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.

Объявление переменных 声明变量

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. В

SQL Server имя локальной переменной предваряется символом @, а глобальной — @@.

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

Блок中使用的所有变量都必须在声明部分中定义。（在后面 FOR 循环的计数器变量除外，它是自动声明的）。

变量可以是任何 SQL 数据类型，例如 integer、varchar、char 等。

这些变量的特点是它们可以存储 NULL 值。在 SQL Server 局部变量名以@ 字符开头，全局变量名以@@ 开头。

Общий синтаксис объявления переменной: 通用变量声明语法

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

更多关于声明变量

Подробнее про объявление переменных: <https://postgrespro.ru/docs/postgresql/9.6/plpgsql-declarations#plpgsql-declaration-records>

Подробнее про типы данных: <https://postgrespro.ru/docs/postgrespro/10/datatype>

有关数据类型的更多信息

Присвоение значений переменным: 给变量赋值：

Присвоение значения переменной записывается в виде: 给变量赋值可以写成：

переменная { := | = } выражение; 变量 { := | = } 表达式；

Как описывалось ранее, выражение в

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 **SELECT** **发送到主数据库引擎。** 表达式应该得到单个值（如果变量的类型为字符串或类型，则可能是字符串值记录）。目标变量可以是一个简单的变量（可能填充块名称），字符串类型或记录变量中的字段；或数组元素这是一个简单的变量或字段。您可以使用分配

знак равенства (=) вместо совместимого с PL/SQL :=. 等号 (=) 而不是 PL/SQL 兼容的 :=。

Подробнее: <https://postgrespro.ru/docs/postgrespro/10/plpgsql-declarations>

Примеры:

1. Объявление переменной типа integer и присвоение значения. 声明一个整数类型的变量并赋值。

```
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;
```

Вызов функции. 函数调用

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

Data Output	Expla
set_int_var integer	
1	15

2. Объявление переменной типа integer и присвоение значения Null. 声明一个整数类型的变量并分配一个 Null 值。

```
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;
```

Вызов функции.

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

Data Output	Explai
set_null_var integer	
1	[null]

3. Объявление переменной типа rowtype (1) и присвоение ей значения строки таблицы Employee по id (2).

声明一个 rowtype(1) 类型的变量并为其分配表行的值
员工 ID (2)。

```

CREATE OR REPLACE FUNCTION get_row_var(emp_id integer) RETURNS text AS
$$
DECLARE
    var_row public."Employee"%rowtype;
(1)
BEGIN
    SELECT * INTO var_row FROM public."Employee" WHERE id = emp_id;
(2)
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Сотрудник с id % не найден', emp_id;
    END IF;
    RETURN var_row.last_name || ' ' || row.first_name || ' ' || row.patronymic;
END
$$
LANGUAGE plpgsql;

```

Вывод ФИО сотрудника с id=2.

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

Data Output	Explain	Messages	Notifications
get_row_var text			
1	Кузнецов Максим Максимович		

Попытка вывода ФИО сотрудника с несуществующим id выдаст ошибку. 尝试显示不存在 id 的员工的全名将产生错误

```
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			

Условные операторы. 条件语句。

Синтаксис: 句法： 运营商

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

Подробнее про условные операторы <https://postgrespro.ru/docs/postgresql/9.6/plpgsql-control-structures#plpgsql-conditionals>

Пример:

Функция, которая разделит заданное в параметре число пополам и сравнит результат с 10.
 一个函数，它将参数中给出的数字分成两半，并将结果与?? 10 进行比较

```

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;

SELECT * FROM compare_half_to_10(15);

```

Data Output	Explain	Message
1	Меньше 10	

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

1	Равно 10	

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

Data Output	Explain	Message
1	Больше 10	

Циклы — LOOP. цикл - цикл.

Синтаксис:

```
[<<метка>>]
```

```
LOOP
```

```
операторы
```

```
END LOOP [метка];
```

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.

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.

Подробнее: <https://postgrespro.ru/docs/postgrespro/10/plpgsql-control-structures#PLPGSQL-CONTROL-STRUCTURES-LOOPS>

Пример: Функция, 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). Возвращает количество итераций по циклу.

示例：使用循环将传递的值加 1 的函数直到它变为 0。当值达到 100（或最初它 >= 100，则循环将退出）。返回循环的迭代次数

```

CREATE OR REPLACE FUNCTION loop_to_100(new_int integer)
RETURNS integer AS
$$
DECLARE
    steps integer := 0;
BEGIN
    LOOP
        IF new_int >= 100 THEN
            EXIT; -- выход из цикла
        END IF;
        -- здесь производятся вычисления 计算在这里进行
        new_int := new_int + 1;
        steps := steps + 1;
    END LOOP;
    RETURN steps;
END
$$
LANGUAGE plpgsql;

```

Вызовем функцию для 15.

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

Data Output		Explain
	loop_to_100 integer	
1	85	

Зададим начальное значение больше 100.

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

Data Output		Explain
	loop_to_100 integer	
1	0	

Программирование функций с применением SQL/PSM.

函数编程使用 SQL/PSM。

编程语言通常有两种类型的子程序：

В языках программирования обычно имеется два типа подпрограмм:

- хранимые процедуры; 存储过程；
- определяемые пользователем функции (UDF). 用户定义函数 (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

存储过程由多个语句组成，并且从零到多个输入参数，但通常不返回任何参数。在存储过程中，函数总是返回相同的值。

用户定义函数 (UDF) 是例程，取参数，执行一个动作，比如复杂的计算，然后返回

результат этого действия в виде значения. Возвращаемое значение может быть либо единичным скалярным значением, либо результирующим набором (таблицей).
此操作的结果作为值。返回值可以是单个标量值或结果集（表）。

Создание и выполнение определяемых пользователем функций. 创建和执行用户定义的职能。

Определяемые пользователем функции создаются посредством инструкции **CREATE FUNCTION**, которая имеет следующий синтаксис
用户定义的函数是使用 CREATE 语句创建的
FUNCTION 具有以下语法

CREATE FUNCTION [schema_name.]function_name

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

RETURNS {scalar_type | [@variable] TABLE}

[WITH {ENCRYPTION | SCHEMABINDING}]

[AS] {block | RETURN (select_statement)}

块定义了一个包含函数实现的 BEGIN/END 块。街区内 BEGIN/END 只允许 SET 语句；虽然和如果；声明，选择，插入、更新和删除。

где:

schema_name - имя схемы 模式名称 RETURN 带参数（返回值） - 块的最后一条指令

function_name — имя функции 函数名

@param - входной параметр функции типа *type* 类型类型函数的输入参数

default - значение параметра по умолчанию 参数默认值

RETURNS - определяет тип значения, возвращаемого функцией: 定义函数返回值的类型：

для скалярной функции - стандартный тип данных, кроме timestamp

для табличной функции — тип TABLE. 对于标量函数 - 标准数据类型，时间戳除外
对于表函数，键入 TABLE

block определяет блок BEGIN/END, содержащий реализацию функции. Внутри блока BEGIN/END разрешаются только инструкции SET; WHILE и IF; DECLARE, SELECT, INSERT, UPDATE и DELETE.

RETURN с аргументом(возвращаемым значением) - последняя инструкция блока.

Вызов определяемой пользователем функции. 调用用户定义的函数。

Определенную пользователем функцию можно вызывать с помощью инструкций SELECT, INSERT, UPDATE или DELETE.
可以调用用户定义的函数，使用 SELECT、INSERT、UPDATE 或 DELETE 语句。

Вызов функции осуществляется, указывая ее имя с парой круглых скобок в конце, в которых можно задать один или несколько аргументов.
通过在 end，可以给出一个或多个参数。

Аргументы - это значения или выражения, которые передаются входным параметрам, определяемым сразу же после имени функции.
参数是传递给输入的值或表达式，紧跟在函数名之后定义的参数。

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.

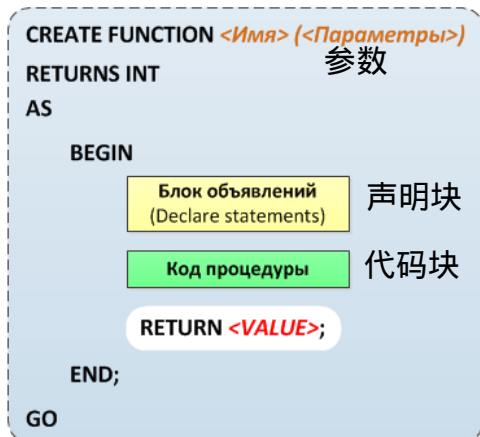
调用函数时，当其参数没有定义的值时默认情况下，所有这些选项都必须在与这些参数在指令中定义的顺序相同创建函数。

Скалярные функции. 标量函数。

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

如果 RETURNS 子句定义了标量之一，则函数是标量数据类型并在每次调用时作为响应返回单个值职能。

Структура скалярной функции



标量函数的结构

Пример скалярной функции 1. 标量函数示例 1。

一个函数，它接受 x 和 y 值，将它们相加并返回结果。 Функция, 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;
  
```

Вызов функции:

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

Data Output	Expr
add_xy integer	
1	3

Пример скалярной функции 2.

max_work_hours 函数将返回工作时间的最大值。

Функция max_work_hours вернет максимальное значение рабочих часов.

```

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;
  
```

Вызов функции для отдела с id=1.

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

Data Output	Explain	Mes
max_work_hours integer		
1		39

Пример скалярной функции 3.

一个类似的功能，但使用一个变量并将值存储到其中。

Аналогичная функция, но с использованием переменной и сохранением в нее значений.

Функция, которая определит среднее кол-во рабочих часов в неделю у сотрудников отдела с заданным id, присвоит значение переменной и вернет его. 确定部门员工每周平均工作时间的函数
给定 id，将值分配给变量并返回它。

```
CREATE OR REPLACE FUNCTION get_avg_work_hours(dep_id integer)
RETURNS integer AS
$$
DECLARE
    avg_hours integer;          -- объявление переменной 变量声明
BEGIN
    -- присвоение результата запроса переменной 将查询结果分配给变量
    SELECT avg(work_hours_per_week) INTO avg_hours
    FROM public."Employee" WHERE department_id=dep_id;
    -- возврат переменной 返回变量
    RETURN avg_hours;
END
$$
LANGUAGE plpgsql;
```

Вариант вызова 1.

Вызов функции для отдела с id=1. 为 id=1 的部门调用函数。

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

Data Output	Explain	Mess:
get_avg_work_hours integer		
1		24

让我们在一个查询中调用该函数，该函数接收一个 id=1 的部门员工列表，他们有多

Вариант вызова 2. 每周工作时间高于部门平均水平

Вызовем функцию в запросе, получающем список сотрудников отдела с id=1, у которых кол-во рабочих часов в неделю больше среднего значения по отделу.

```
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	
1		1	Степанова	Светлана		Степановна		[null]		33		1		1
2		8	Субботин	Георгий		Иванович		[null]		33		1		1
3		15	Петров	Петр		Петрович		[null]		39		1		1

Возвращающие табличное значение функции 表值函数

Функция является возвращающей табличное значение, если предложение RETURNS
如果 RETURNS 子句，则函数是表值的

返回一组行。如果在 RETURNS 子句中指定了 TABLE 关键字，这样的函数是内置的。SELECT 语句内联函数将结果集作为数据类型为 TABLE 的变量返回。

возвращает набор строк. Если в предложении RETURNS ключевое слово TABLE указывается без списка столбцов, такая функция является встроенной. Инструкция SELECT встраиваемой функции возвращает результирующий набор в виде переменной с типом данных TABLE.

Многоинструкционная возвращающая табличное значение 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.

Структура табличной функции



多指令表值函数包含一个名称，定义一个数据类型为 TABLE 的内部变量。这种数据类型由变量名后面的 TABLE 关键字指示。在这个选定的行被插入到变量中并作为返回值职能。

Пример табличной функции 1.

```

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;

```

. 返回 id = dep_id 的所有部门员工的

Пример табличной функции 2. Вернет 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 табличной функции. 调用表函数。

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

Вызов функции в подзапросе. 在子查询中调用函数

Данный запрос выведет всю информацию о сотрудниках, которые относятся к отделу с id=1 и у которых фамилия- Павлов.

此查询将显示属于 id=1 的部门的员工的所有信息，并且他的姓是巴甫洛夫。

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

Подробнее о функциях: <https://postgrespro.ru/docs/postgresql/9.4/xfunc-sql>

<https://postgrespro.ru/docs/postgrespro/9.5/sql-createfunction>

Хранимые процедуры 存储过程 存储过程是预编译过程 用 T-SQL 编写并位于数据库中的程序。

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

Особенности: 特点：

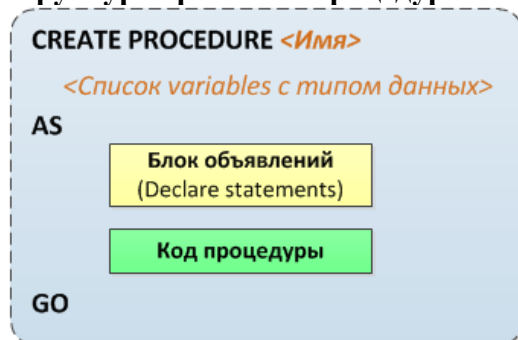
它们比常规 T-SQL 更快，因为它们以编译形式存储

- Выполняются быстрее обычного T-SQL, 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.

可以帮助将用户与底层表结构隔离，隐藏执行任何可能的功能。

存储过程的结构

Структура хранимой процедуры.



？由客户端程序、另一个存储过程或触发器调用。？结合查询和程序逻辑（赋值、逻辑分支等）并存储在数据库中

Подробнее: <https://postgrespro.ru/docs/postgresql/11/sql-createprocedure>

Пример. 将创建一个带有 id、title、description 和 title 字段的新表的过程，传入参数。

Процедура, 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);
```

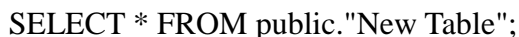
```
END
```

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

Вызов процедуры. 过程调用。

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

Создалась таблица. 表已创建。



捕获和抛出异常。

RAISE 命令用于显示消息和引发错误。
句法：

Синтаксис:

- **通知、警告和例外。** Уровень важности ошибки. Возможные значения: DEBUG, LOG, INFO, NOTICE, WARNING и EXCEPTION.

NOTICE, WARNING и EXCEPTION. EXCEPTION 引发错误 (其中通常中止当前事务), 仅其他级别值生成具有不同优先级的消息 (перевывает текущую транзакцию), остальные значения уровня только

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

使用 USING，您可以在报告中添加关于 MESSAGE 参数的可能关键字是：MESSAGE 安装文本消息的错误。
错误。 参数的可能关键字是：MESSAGE 安装文本消息的错误。

- ? MESSAGE - 设置错误消息的文本。
? DETAIL - 提供详细的错误消息。
? HINT - 为引发的错误提供提示。
? ERRCODE - 设置错误代码 (SQLSTATE)。错误代码设置为按名称或直接使用五个字符的 SQLSTATE 代码。
? CONSTRAINT
? 列
? 约束
? 数据类型
? 表
? SCHEMA - 提供与错误关联的相应对象的名称

Подробнее: <https://postgrespro.ru/docs/postgresql/9.6/plpgsql-errors-and-messages>

Коды ошибок: <https://postgrespro.ru/docs/postgresql/9.6/errcodes-appendix>

检查具有给定姓氏的员工是否存在的函数

Пример: 如果它不存在，则捕获该案例并引发异常。

Функция, 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';

```

为员工表中存在的姓氏调用函数

Вызов функции для фамилии, которая присутствует в таблице сотрудников.
 SELECT * FROM check_lastname('Сергеев');

Data Output	Explain	...
check_lastname	text	
1	Существует	

为没有员工的姓氏调用函数

Вызов функции для фамилии, которой нет ни у одного сотрудника.
 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			

Извлечение части записей и результатов запросов на изменение данных

提取部分记录和查询结果
数据变化

LIMIT

Выводит ограниченное число записей.

Пример получения первых 10 записей с использованием табличной функции employee_fio, которая выводит сотрудников отдела с id=1.

限制

显示有限数量的记录。

使用 employee_fio 表函数获取前 10 条记录的示例，
显示 id=1 的部门的员工。

	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

SELECT * FROM employee_fio(1) LIMIT 10;

Подробнее: <https://postgrespro.ru/docs/postgresql/9.6/queries-limit>

RETURNING 返回 RETURNING 允许您查询添加/更改（插入/更新）数据并显示更改的行。列表 RETURNING 与 SELECT 结果列表具有相同的语法。

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

Подробнее: <https://postgrespro.ru/docs/postgresql/9.5/dml-returning>

Примеры выполнения запроса. 请求执行示例。此查询将插入一个新员工并返回插入的值。 Данный запрос выполнит вставку нового сотрудника и вернет вставленные значения.

```
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	

Данный запрос выполнит изменение поля количество рабочих часов и после выполнения изменения строки вернет ее.

```
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

```

	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
1	1	Иванова	Елена	Дмитриевна	[null]	35

Аналогичный запрос, но с проекцией необходимых столбцов.

```

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

```

	last_name character varying (30)	work_hours_per_week integer
1	Иванова	35

Рекурсивный запрос. 递归查询。

需要递归查询来显示基于先前行的数据样本。它是使用 WITH 语句实现的。

Рекурсивный запрос необходим, для вывода данных на основе предыдущих строк в выборке. Реализуется он с помощью оператора **WITH**.

Общая схема рекурсивного запроса: 递归查询的一般方案：

WITH RECURSIVE t AS (

нерекурсивная часть (база) (1) 非递归部分 (基础)

UNION ALL

рекурсивная часть (индукционные шаги) (2) 递归部分 (归纳步骤)

) 在递归查询中 (as 关键字后括号中的内容)

SELECT * FROM t; (3) 可以分为两部分, 由 union 关键字联合起来。第一部分 - 这是一个查找从其开始递归查询的元素的查询。第二部分 - 在每次迭代中执行什么。

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

В рекурсивном запросе FROM t не выполняет весь запрос снова, а работает так: в первый раз берет то, что лежит в стартовой части рекурсии, а в следующие итерации берет результаты предыдущей итерации.

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.

当当前归纳步骤的结果不是时递归查询停止

包含行。那是空的。Рекурсивный запрос останавливается, когда результат текущего индукционного шага не содержит строк. То есть пуст.

归纳的基础, 即非递归部分, 与归纳步骤分开 UNION (两个查询中的列数和类型必须匹配)。还有必要确保停止递归查询的可能性。

Подробнее про рекурсию: <https://postgrespro.ru/docs/postgrespro/9.5/queries-with>

<https://smyt.ru/blog/kak-ya-teoriyu-6-ti-rukopozhatij-proveryal-chast-1/>

Пример выполнения рекурсивного запроса. 执行递归查询的示例。

Добавим в таблицу Отдел поле „Является подотделом“ - subdep_of и выведем последовательно отделы, подотделом которых является выбранный (с id=4),

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

Выведем id отделов и чьим подотделом является каждый отдел.

将字段 “Is a subdepartment” - subdep_of 添加到 Department 表并输出
依次选择子部门的部门 (id=4),
ALTER TABLE public."Department"
添加 subdep_of 整数;
让我们输出部门的id和每个部门的子部门。

获取部门 4 的所有父部门。

在这个例子中，首先在查询的非递归部分，会得到一个表记录 id = 4 的部门，即归纳基础。该条目被传递到递归部分，其中执行归纳步骤，每个步骤计算哪个部门的细分是上一步得到的部门。在这个例子中，我们看到主要部门是部门 5，其细分是部门 2，第二个细分是部门 4。

```
12 SELECT id, subdep_of from public."Department"
```

Data Output Explain Messages Notifications

	id [PK] integer	subdep_of integer	
1		3	4
2		1	3
3		4	2
4		2	5
5		5	[null]

获取部门 4 的所有父部门。

在这个例子中，首先在查询的非递归部分，会得到一个表记录 id = 4 的部门，即归纳基础。该条目被传递到递归部分，其中执行归纳步骤，每个步骤计算哪个部门的细分是上一步得到的部门。在这个例子中，我们看到主要部门是部门 5，其细分是部门 2，第二个细分是部门 4。

Получим все родительские отделы для 4 отдела.

В данном примере сначала в нерекурсивной части запроса будет получена запись таблицы отделов с id = 4, то есть база индукции. Эта запись передается в рекурсивную часть, где выполняются индукционные шаги, каждый из которых вычисляет, подотделом какого отдела является отдел, полученный на предыдущем шаге. В данном примере видим, что главным отделом является отдел 5, его подотделом является 2 отдел, а у 2-го подотделом является отдел 4.

Query Editor Query History

```
1 WITH RECURSIVE SubDepart AS
2 (
3 SELECT id, name, subdep_of
4 FROM public."Department"
5 WHERE id=4
6 UNION
7 SELECT d.id, d.name, d.subdep_of
8 FROM public."Department" d, SubDepart sdep
9 WHERE sdep.subdep_of = d.id)
10 SELECT * FROM SubDepart;
11
12
```

Data Output Explain Messages Notifications

	id integer	name character varying (255)	subdep_of integer	
1	4	Бухгалтерия	2	
2	2	dep2	5	
3	5	Кредитные процессы	[null]	

динамические запросы. 动态查询。

Динамические запросы.

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.

Синтаксически в поле FROM нельзя подставлять переменную. Чтобы избежать ошибки, надо этот запрос сделать динамическим.

Причины использования:

1. дополнительная гибкость в приложении
2. оптимизация отдельных запросов

Цена:

1. не используются подготовленные операторы
2. возрастает риск внедрения SQL-кода
3. 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 **设置变量**

SET @TableName = "Products" — установка переменной значения **设置变量值**
названия таблицы **表名**

EXECUTE ('SELECT * FROM' + @TableName) — выполнение запроса, который **执行一个请求**
выберет все записи из таблицы (для **将从表中选择所有记录 (对于**
сервера это просто строка) **服务器只是一个字符串)**

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.

Подробнее: <https://postgrespro.ru/docs/postgrespro/9.5/ecpg-dynamic>

<https://postgrespro.ru/docs/postgresql/9.6/plpgsql-statements.html#plpgsql-statements-executing-dyn>

<https://docplayer.ru/79851552-Pl-pgsql-dinamicheskie-komandy.html>

动态查询是 SQL 查询，其文本形成并
然后在 PL/pgSQL 块中执行，例如在存储函数
或
这种程序语言中的匿名块。
从语法上讲，您不能替换 FROM 字段中的变量
。为避免出错，
有必要使这个请求动态化。
使用理由：
1. 应用程序的额外灵活性
2. 个别查询的优化
价钱：
1. 不使用准备好的语句
2. SQL注入风险增加
3. 维护的复杂性增加

Использование параметров в командах: 在命令中使用参数：

В тексте команды можно использовать значения параметров, ссылки на параметры обозначаются как \$1, \$2 и т. д. Такой метод зачастую предпочтительнее, чем вставка значений в команду в виде текста.

В качестве более аккуратного решения, вместо имени таблиц или столбцов можно использовать указание формата %I с функцией format() (текст, разделённый символами новой строки, соединяется вместе):

Подробнее: <https://postgrespro.ru/docs/postgresql/9.6/plpgsql-statements>

使用过程的动态查询示例**Пример динамического запроса с использованием процедуры.**

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

Вызовем процедуру для существующего вида

```
CALL copy_sport('Атлетика'); (11)
select * from public."Атлетика";
```

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

Попробуем теперь вызвать процедуру с несуществующим названием.

```
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
```

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 а 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 1 параметре(\$1) процедуры (или иначе sport_title) , записи из таблицы Sport_Section с id, совпадающим с sport_id, полученным на 3 шаге.

11 — вызов функции для вида спорта Атлетика.

В результате вызова данной процедуры с динамическими запросами будет создана таблица Атлетика, куда скопируются записи с видом спорта Атлетика из таблицы Sport_Section.

使用函数创建动态查询的示例。

Пример создания динамического запроса при помощи функции.

Данная функция копирует 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;
```

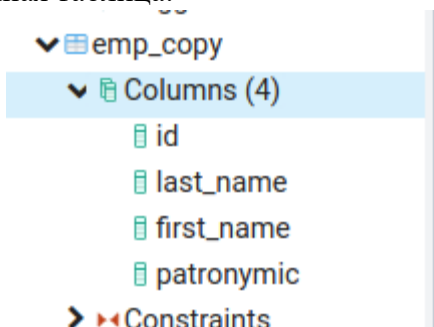
```
END
```

```
$BODY$ LANGUAGE plpgsql;
```

Вызов функции.

```
SELECT cory_emp('emp_cory', 3);
```

Новая созданная таблица:



Data Output				
	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

Ранжирующие запросы. 排名查询。

Ранжирующая(или оконная) функция выполняет вычисления для набора строк, попавшим в один раздел с текущей строкой. В отличие от обычной агрегатной функции, при использовании оконной функции несколько строк не группируются в одну, а продолжают существовать отдельно.

Подробнее: <https://postgrespro.ru/docs/postgrespro/9.5/functions-window>

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.

ROW_NUMBER

ROW_NUMBER – функция нумерации, возвращает просто номер строки.

Пример.

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

```
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

RANK

RANK – возвращает ранг каждой строки.

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

Пример. Данный запрос выводит ранг и номер записи в таблице Прайс-листа.

ROW_NUMBER сортирует по цене и выводит просто номер строки.

```
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

RANK возвращает ранг 1-го товара с данной ценой для всех товаров с такой же ценой с пропуском следующего.

DENSE_RANK

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

Пример. Данный запрос выводит ранги 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.

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
```

	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

NTILE

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

Пример. 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
```

	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

Функция — агрегат

Агрегатная функция вызывается для каждой строки таблицы по очереди и в конечном итоге обрабатывает их все. 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 ]  
)
```

Подробнее: <https://postgrespro.ru/docs/postgresql/9.6/sql-createaggregate>

Пример:

В данном 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 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
```

Результат:

Data Output Explain Mes:		
	incremented_max integer	🔒
1		22

DML — триггеры.

Триггеры PL/SQL уровня команд DML (или просто триггеры DML)

активизируются после вставки, обновления или удаления строк конкретной таблиц.

- 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. 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.

Подробнее о триггерах: <https://postgrespro.ru/docs/postgresql/9.6/sql-createtrigger>

Синтаксис:

```
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];
```

Пример.

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 выше описанную функцию rec_insert.

```
CREATE TRIGGER ins_same_rec
  AFTER INSERT
  ON employee
  FOR EACH ROW
```

```
EXECUTE PROCEDURE rec_insert();
```

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

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

DDL триггер.

Триггеры DDL активируются в ответ на различные события языка DDL. Эти события в основном соответствуют инструкциям Transact-SQL, 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 Transact-SQL, 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.

Подробнее о триггерах: <https://postgrespro.ru/docs/postgresql/9.6/sql-createtrigger>
Синтаксис.

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

Пример.

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;
```

Вопросы для самопроверки:

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. Синтаксис и процесс выполнения рекурсивных запросов.
5. Синтаксис и назначение ранжирующих функций.
6. Постреляционные возможности языка SQL.
7. Как создать, собрать, присоединить и вызвать функцию агрегат? Каковы ее методы?
8. Что такое DML триггер, как и когда он запускается и как его создать? Как из триггера определить изменяемые данные?
9. Что такое динамические запросы? Как их выполнять?
10. Что такое DDL триггер, как и когда он запускается и как его создать? Как из триггера определить возникшее событие и его параметры?

В отчет:

1. Тексты SQL сценариев (создание объектов БД), запросов, команд и процедур.
2. Результаты выполнения запросов (скриншоты).

Литература:

1. Документация PostgreSQL – postgrespro.ru – Документация – PostgreSQL 9.4 – Серверное программирование. <https://postgrespro.ru/docs/>
2. Методические рекомендации.
3. PL/pgSQL Динамические команды <https://docplayer.ru/79851552-Pl-pgsql-dinamicheskie-komandy.html>
4. Downloads - <https://www.postgresql.org/download/>
5. PgAdmin - <https://www.pgadmin.org/download/>