

Московский Государственный Технический Университет имени Н.Э. Баумана
Кафедра "Системы обработки информации и управления"

Методические указания
на тему «Работа в среде СУБД PostgreSQL»
по дисциплине «Постреляционные базы данных»

Составил Крутов Т.Ю.

МГТУ им. Н.Э. Баумана, кафедра ИУ5

Москва, 2020 г.

Содержание

Содержание.....	2
Теоретический материал.....	3
<i>Введение.....</i>	3
<i>Основы архитектуры PostgreSQL.....</i>	3
<i>Работа с PostgreSQL на платформе pgAdmin.....</i>	5
<i>Создание таблиц БД.....</i>	7
<i>Секционирование таблиц.....</i>	19
<i>Хранимые функции.....</i>	23
Учебная литература.....	28

Теоретический материал

Введение

PostgreSQL - это объектно-реляционная система управления базами данных (ОРСУБД, ORDBMS), основанная на POSTGRES (руководитель проекта - профессор Массачусетского технологического института Майкл Стоунбрейкер), Version 4.2 — программе, разработанной на факультете компьютерных наук Калифорнийского университета в Беркли. В POSTGRES появилось множество новшеств, которые были реализованы в некоторых коммерческих СУБД гораздо позднее.

PostgreSQL - СУБД с открытым исходным кодом, основой которого был код, написанный в Беркли. Она поддерживает большую часть стандарта SQL и предлагает множество современных функций:

- сложные запросы,
- внешние ключи,
- триггеры,
- изменяемые представления,
- транзакционная целостность,
- многоверсионность.

Основы архитектуры PostgreSQL

PostgreSQL реализован в архитектуре клиент-сервер. Рабочий сеанс PostgreSQL включает следующие взаимодействующие процессы (программы):

- Главный серверный процесс, управляющий файлами баз данных, принимающий подключения клиентских приложений и выполняющий различные запросы клиентов к базам данных. Эта программа сервера БД называется postgres.
- Клиентское приложение пользователя для выполнения операций в базе данных. Клиентские приложения разнообразны: это может быть текстовая утилита, графическое приложение, веб-сервер, использующий базу данных для отображения веб-страниц или специализированный инструмент для обслуживания БД. Некоторые клиентские приложения поставляются в составе дистрибутива PostgreSQL, однако большинство создают сторонние разработчики.

Как и в других типичных клиент-серверных приложениях, клиент и сервер могут располагаться на разных компьютерах, в этом случае они взаимодействуют по сети TCP/IP. Сервер PostgreSQL может обслуживать одновременно несколько подключений клиентов, запуская отдельный процесс для каждого подключения.

Установка PostgreSQL на Windows

С полной документацией по PostgreSQL можно ознакомиться, перейдя по ссылке:
<https://postgrespro.ru/docs/postgresql/>.

Скачайте установочный файл PostgreSQL с официального сайта
<https://www.postgresql.org/download/>.

Установите PostgreSQL (см. рис.1).

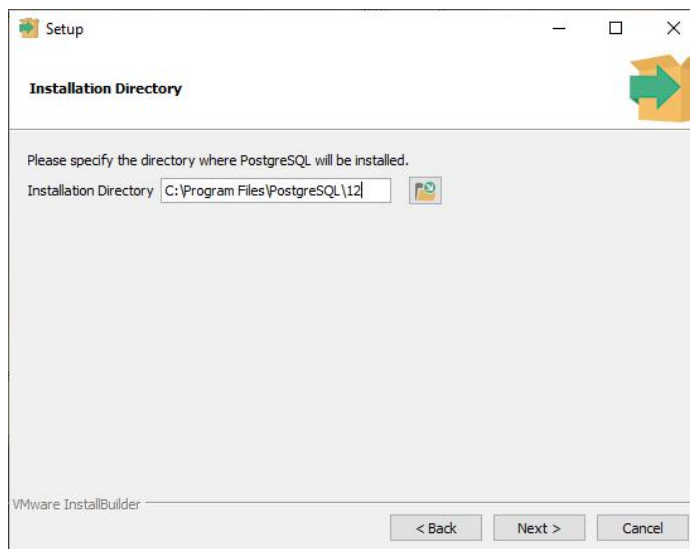


Рис.1. Процесс установки PostgreSQL

Установщик предлагает сразу установить графический интерфейс для администрирования pgAdmin. Установим его (рис.2).

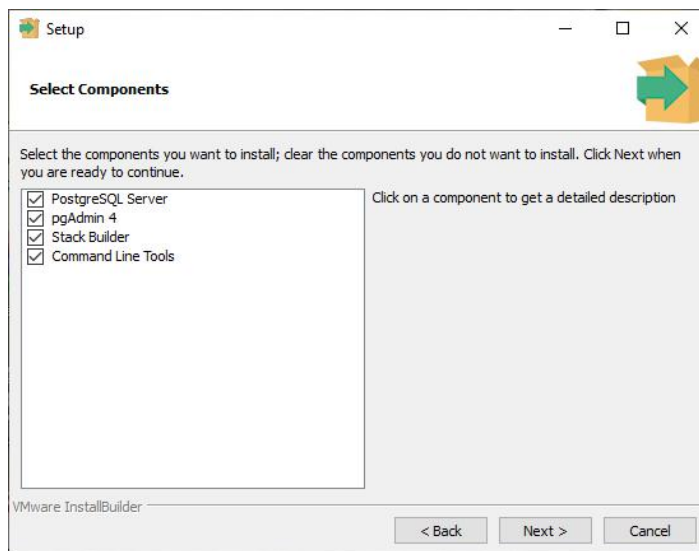


Рис.2. Процесс установки PostgreSQL

Установим пароль, который будет необходимо вводить при каждом входе в pgAdmin. Пароль должен состоять минимум из 4 символов. (см. рис.3).

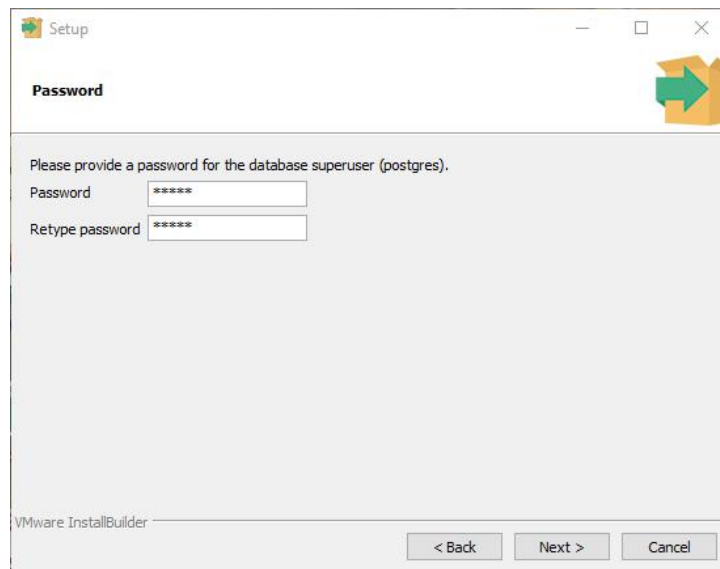


Рис.3. Процесс установки PostgreSQL

Остальные настройки оставим по умолчанию. Запустим pgAdmin, введём пароль, указанный при установке.

Работа с PostgreSQL на платформе pgAdmin

pgAdmin – это открытая платформа администрирования и разработки для PostgreSQL и связанных с ней систем управления базами данных.

Меню pgAdmin включает немало полезных инструментов — подсветку строк, редактор, быстрый поиск. Функционал условно делится на 3 области (при необходимости их расположение можно изменять, см. рис. 4):

- Слева - дерево всех объектов.
- По центру - данные о конкретном объекте.
- Справа - операторы, использовавшиеся для создания этого объекта.

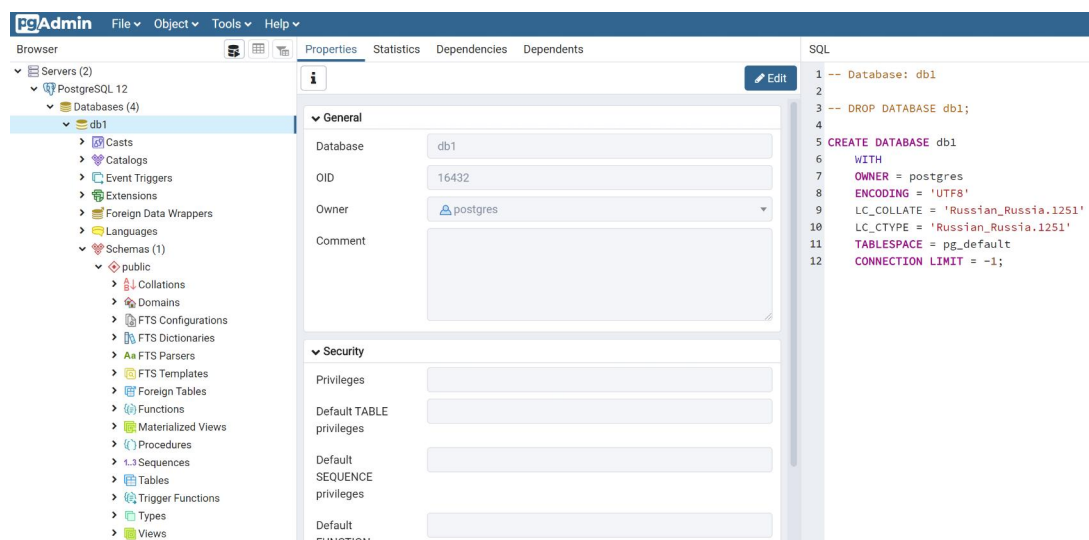


Рис. 4. Интерфейс pgAdmin

Работа с базами данных в pgAdmin осуществляется на сервере (после первого запуска программы необходимо добавить и настроить новый сервер, см. рис. 5) как при помощи инструментов интерфейса, так и посредством написания SQL – сценариев.

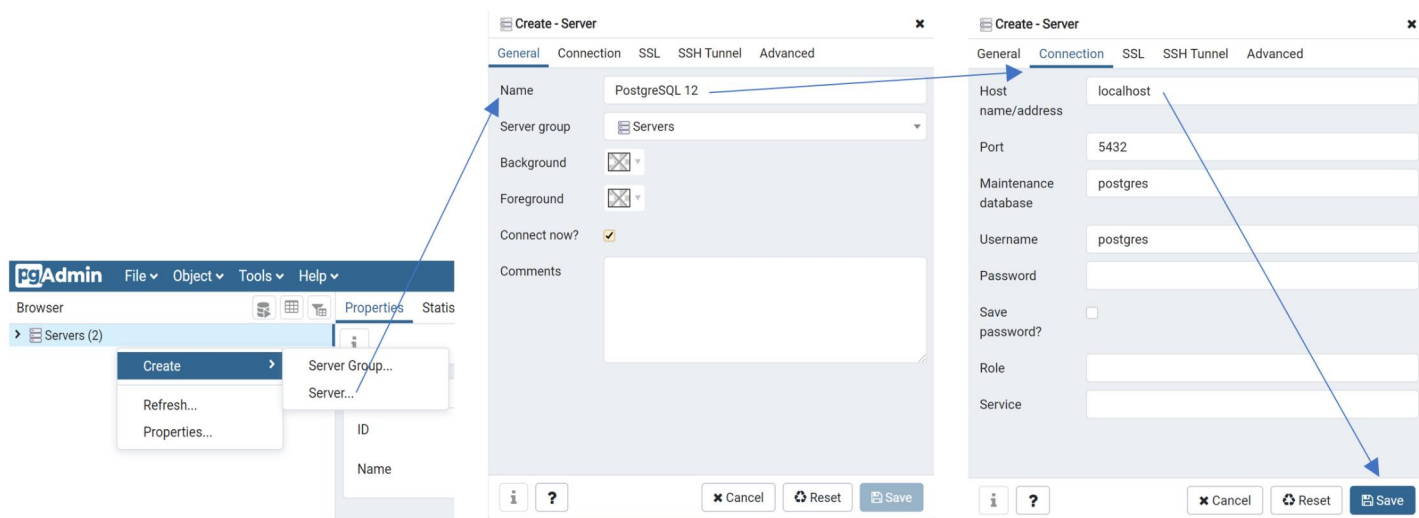


Рис. 5. Создание сервера

После создания базы данных (см. рис. 6) можно приступить к написанию запросов, которые выполняются при помощи Query Tool (см. рис. 7). Запуск запроса производится кнопкой Execute на панели инструментов в верхней части экрана, результат запроса выводится в окна Data Output, Messages, Notifications, расположенные внизу. Содержимым и свойствами баз данных удобно управлять через дерево – Browser, щелкая ПКМ по интересующему объекту. Просматривать содержимое таблиц можно, используя кнопку View Data, находящуюся в окне Browser.

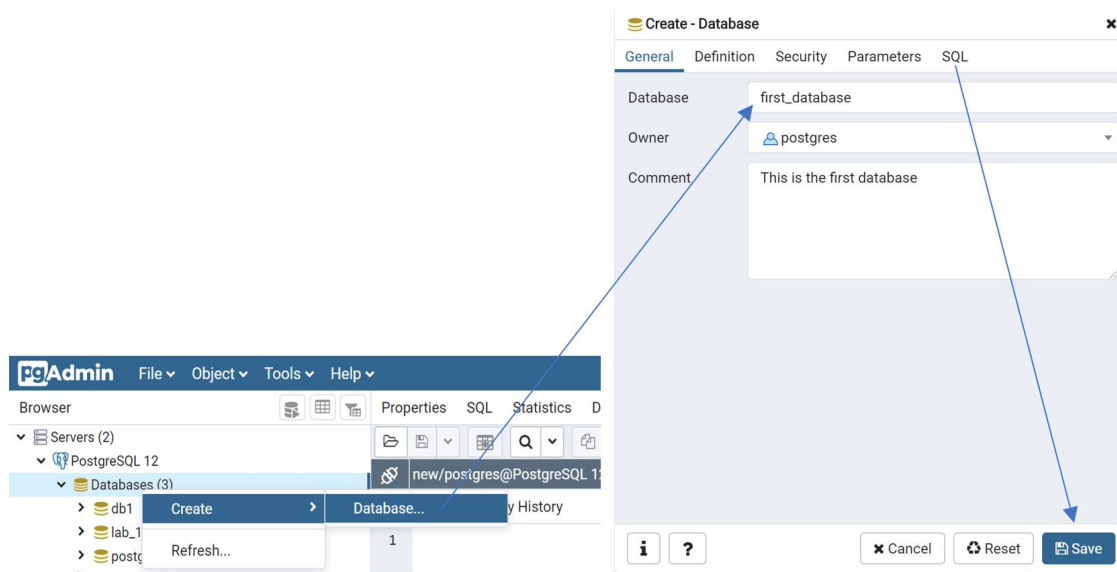


Рис. 6. Создание базы данных

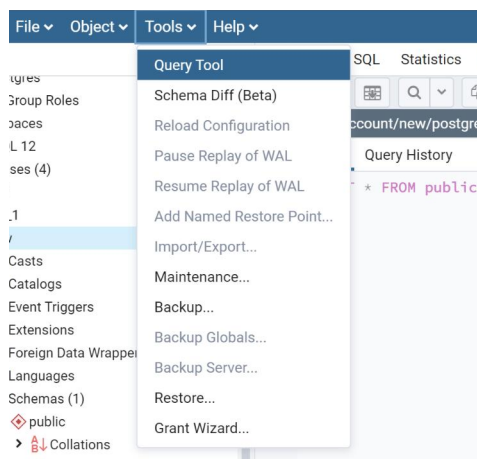


Рис. 7. Открытие Query Editor

Создание таблиц БД

Рассмотрим процедуру создания таблиц на примере базы данных «Системы оцифровки изображений и текста», указав имена таблиц и перечислив все имена столбцов и их типы.

- Таблица «**Usser**» будет содержать информацию о пользователях системы (ID пользователя, фамилию, имя, e-mail, телефон, адрес, дату рождения, даты заказов).
- Таблица «**Account**» будет хранить данные для входа в личный кабинет (ID аккаунта, ID пользователя, логин и пароль).
- В таблицу «**Order**» будут помещены записи о заказах клиентов (ID заказа, ID клиента, количество объектов для оцифровки и тип объектов, подлежащих оцифровке).

创建一个表 ■ Создание таблицы (<https://postgrespro.ru/docs/postgresql/10/tutorial-table>)

выполняется при помощи конструкции:

CREATE TABLE имя_таблицы (имя_поля тип_поля ограничение, ...);

Создадим таблицу с названием «Usser». Добавим в качестве полей первичный ключ с автоинкрементом (userID), три ненулевых строки: имени (FirstName – строка переменной длины не более 40 символов), фамилии (LastName – строка переменной длины не более 40 символов) и электронной почты (Email – строка переменной длины не более 40 символов) клиента. Поле телефона (Telephone) определим как массив (элементы массива – строки переменной длины не более 15 символов). Поле адреса (Addr) является структурой типа address. Дата рождения клиента (BirthDay) хранится в переменной типа DATE. В таких же переменных хранится массив дат создания заказов конкретным клиентом (dat).

Код создания таблицы «Usser» будет выглядеть следующим образом:

```
CREATE TABLE Usser
(
    userID SERIAL PRIMARY KEY,
    FirstName CHARACTER VARYING(40) NOT NULL,
    LastName CHARACTER VARYING(40) NOT NULL,
    Email CHARACTER VARYING(40) NOT NULL,
    Telephone varchar(15) [],
    Addr address,
    BirthDay DATE,
    dat DATE [],
    UNIQUE (Email)
);
```

首要的关键 ▪ **Первичный ключ** (<https://postgrespro.ru/docs/postgrespro/9.5/ddl-constraints#ddl-constraints-primary-keys>) задаётся конструкцией **PRIMARY KEY** после указания

типа поля.

Например, мы указываем, что поле userID в таблице Usser является автоинкрементным (**SERIAL**) первичным ключом:

```
userID SERIAL PRIMARY KEY,
```

外键 ▪ **Внешний ключ** (<https://postgrespro.ru/docs/postgresql/11/tutorial-fk>) при создании таблицы задаётся конструкцией:

```
имя_поля тип_поля,
FOREIGN KEY(имя_поля) REFERENCES имя_таблицы(имя_поля) ограничения,
```

Например, в таблице Orrder поле accountID – внешний ключ, ссылающийся на таблицу Account:

```
FOREIGN KEY (accountID) REFERENCES Account (accountID) ON UPDATE CASCADE ON DELETE SET NULL,
```

По умолчанию внешний ключ не позволяет производить никаких действий по обновлению и удалению записей. Чтобы это настроить, используем ограничения на связь таблиц.

表关系限制 ▪ **Ограничения на связь таблиц** (<https://postgrespro.ru/docs/postgrespro/9.5/ddl-constraints#ddl-constraints-exclusion>) обеспечивают каскадное обновление данных

в случае удаления записей таблиц, на которые ссылаются другие таблицы. Для этого используют конструкцию:

ON UPDATE CASCADE ON DELETE SET NULL

Пример использования можно увидеть в создании таблицы `Order`: значение поля `accountID`, ссылающегося на таблицу `Account`, будет обнулено в случае удаления соответствующей записи в `Account`:

```
FOREIGN KEY (accountID) REFERENCES Account (accountID) ON UPDATE CASCADE ON DELETE SET NULL,
```

唯一性约束 ▪ **Ограничения на уникальность** (<https://postgrespro.ru/docs/postgrespro/9.5/ddl-constraints#ddl-constraints-unique-constraints>) выставляются посредством добавления ключевого слова **UNIQUE** после указания типа поля или в конце запроса с помощью конструкции **UNIQUE**(имя_поля), как, например, добавление ограничения на уникальность поля `email` в таблице `Usrer`:

```
UNIQUE (Email)
```

价值限制 ▪ **Ограничения на значения** (<https://postgrespro.ru/docs/postgresql/12/ddl-constraints#DDL-CONSTRAINTS-CHECK-CONSTRAINTS>) выставляются проверкой условий с помощью конструкции:

```
имя_поля тип_поля CHECK(условие),
```

Так, в таблице `Order` существует ограничение на количество объектов оцифровки - их должно быть не менее 1 и не более 98:

```
Ammount_items INTEGER DEFAULT 1 CHECK (Ammount_items > 0 AND Ammount_items < 99),
```

空约束 ▪ **Ограничение на NULL** (<https://postgrespro.ru/docs/postgrespro/9.5/ddl-constraints#idp7>) обеспечивает защиту полей таблиц от ввода нулевого значения. Записывается в формате:

```
имя_поля тип_поля NOT NULL,
```

Например, в таблице `Usrer` значения полей `FirstName`, `LastName` и `Email` не могут быть нулевыми:

```
FirstName CHARACTER VARYING(40) NOT NULL,  
LastName CHARACTER VARYING(40) NOT NULL,  
Email CHARACTER VARYING(40) NOT NULL,
```

▪ Если пользователь не введёт значение некоторого поля, можно устанавливать это **значение по умолчанию** (<https://postgrespro.com/docs/postgresql/10/ddl-default>) при помощи конструкции:

имя_поля тип_поля **DEFAULT** значение

Например, в таблице Order значение по умолчанию для количества объектов оцифровки (Amount_items) – 1:

```
Amount_items INTEGER DEFAULT 1 CHECK (Amount_items > 0 AND Amount_items < 99),
```

Полный код создания вышеуказанных таблиц приведён на рисунке 8, пояснения по ограничениям полей приведены ниже.

```
1 CREATE TYPE address AS (  
2     town varchar(20),  
3     street varchar (20),  
4     house varchar (20)  
5 );  
6  
7 CREATE TABLE Usser  
8 (  
9     userID SERIAL PRIMARY KEY,  
10    FirstName CHARACTER VARYING(40) NOT NULL,  
11    LastName CHARACTER VARYING(40) NOT NULL,  
12    Email CHARACTER VARYING(40) NOT NULL,  
13    Telephone varchar(15) [],  
14    Addr address,  
15    BirthDay DATE,  
16    dat DATE[],  
17    UNIQUE(Email)  
18 );  
  
1 CREATE TABLE Account (  
2     accountID SERIAL PRIMARY KEY,  
3     userID INTEGER,  
4     Login CHARACTER VARYING (20) CHECK (Login != ''),  
5     Pass CHARACTER VARYING (20) CHECK (Pass != ''),  
6     FOREIGN KEY (userID) REFERENCES Usser (userID) ON UPDATE CASCADE ON DELETE SET NULL,  
7     CONSTRAINT check_login UNIQUE (Login)  
8 );  
9  
10 CREATE TABLE Orrder(  
11     orderID SERIAL PRIMARY KEY,  
12     accountID INTEGER,  
13     FOREIGN KEY (accountID) REFERENCES Account (accountID) ON UPDATE CASCADE ON DELETE SET NULL,  
14     Amount_items INTEGER DEFAULT 1 CHECK (Amount_items > 0 AND Amount_items < 99),  
15     CREATE TYPE OrderType AS ENUM ('Текст', 'Изображение')  
16 );
```

Рис. 8. Создание таблиц

Таблица Usser:

- первичный автоинкрементный ключ – userID;

- ограничение на уникальность задано для поля Email;
- поле пользовательского типа address (тип данных address включает три строки переменной длины не более 20 символов – town, street, house) – Addr (подробнее ознакомиться с типом можно в разделе «Структуры» методических указаний);
- ограничение на нулевые значения заданы для полей FirstName, LastName, Email.

Таблица Account: 账户表 :

- первичный автоинкрементный ключ – accountID;
- ограничение на уникальность задано для поля Login;
- ограничения на принимаемые значения установлено для полей Login и Pass (строки логина и пароля не должны быть пустыми);
- внешний ключ с ограничениями на связь таблиц – userID (ссылается на таблицу Usset).

Таблица Orrder:

- первичный автоинкрементный ключ – orderID;
- поле перечислимого типа, принимающее только одно из двух возможных значений («Текст» или «Изображение») – OrderType (подробнее ознакомиться с типом можно в разделе «ENUM» методических указаний);
- поле со значением по умолчанию ('1') – Ammount_items;
- ограничение на принимаемые значения ('> 0 и < 99') установлено для поля Ammount_items;
- внешний ключ с ограничениями на связь таблиц – accountID (ссылается на таблицу Account).

При создании таблиц удобно применять атрибуты составных типов. Некоторые из них описаны ниже.

■ **Многомерные массивы** (<https://postgrespro.ru/docs/postgresql/12/arrays>)

PostgreSQL позволяет определять столбцы таблицы как многомерные массивы переменной длины. Элементами массивов могут быть любые встроенные или определённые пользователями базовые типы, перечисления, составные типы, типы-диапазоны или домены.

Общая конструкция объявления полей – массивов при создании таблицы выглядит следующим образом:

```
CREATE TABLE имя_таблицы (
    имя_поля_1 тип_поля [],
    имя_поля_2 тип_поля [[]]
```

);

Можно задать точный размер массива, указав в квадратных скобках числа.

Например, следующие команды (рис. 9) добавляют в таблицу Account поле pass_history типа «текстовый двумерный массив» для хранения данных вида «дата установки пароля – пароль пользователя» и внесут данные об устанавливавшихся паролях пользователя с ID=6.

```
40 ALTER TABLE account ADD COLUMN pass_history text [][]
43 UPDATE account
44 SET pass_history = '{{"02.03.2018", "grumP7h"}, {"04.05.2018", "gse166G"}}'
45 WHERE accountid = 6
```

Рис. 9. Добавление двумерного массива

Выведем три пароля, устанавливавшихся пользователем с идентификатором accountId=5:

```
SELECT pass_history[1:3] FROM account WHERE accountId=5
```

	pass_history text[]
1	{{31.12.2019,qwerty2019},{12.01.2020,myPassword},{17.02.2020,#274%#85*47}}

Рис. 10. Обращение к элементам массива и вывод результата

Проверить наличие элемента в массиве можно с помощью при помощи слова **ANY**. Проверим, устанавливал ли кто-то из пользователей пароль «strong_password»:

```
SELECT * FROM account WHERE 'strong_password' = ANY (pass_history);
```





Data Output		Explain	Messages	Notifications			
	accountid [PK] integer		userid integer		pass character varying (20)		pass_history text[]
1		3	1	ptrovvi	strong_password1	{{11.03.2020,password},{19.03.2020,strong_password}}	

Рис. 11. Обращение к элементам массива и вывод результата

Узнать размерность массива можно при помощи словосочетания «array_length», которое возвращает число элементов в указанной размерности массива. Например, определим, сколько раз менял пароль каждый из пользователей:

```
SELECT array_length(pass_history, 1) FROM account;
```

	Data Output	Explai
	array_length integer	
1		[null]
2		[null]
3		2
4		3
5		[null]
6		[null]
7		2
8		[null]
9		[null]

- **ENUM** (<https://postgrespro.ru/docs/postgresql/12/datatype-enum>)

Типы перечислений (enum) определяют статический упорядоченный набор значений, так же, как и типы enum, существующие в ряде языков программирования. В качестве перечисления можно привести дни недели или набор состояний.

Например, в таблице Order с помощью конструкции **CREATE TYPE AS ENUM** был создан перечислимый тип OrderType, ограничивающий выбор типа оцифровываемых объектов типами изображения и текста:

```
CREATE TYPE OrderType AS ENUM ('Текст', 'Изображение')
```

- **Структуры** (<https://postgrespro.ru/docs/postgresql/12/sql-createtype>)

Составной тип представляет структуру табличной строки или записи; по сути это просто список имён полей и соответствующих типов данных. PostgreSQL позволяет использовать составные типы во многом так же, как и простые типы. Например, в определении таблицы можно объявить столбец составного типа.

Так с помощью **CREATE TYPE** address **AS** (...); объявляется структура адреса: город – улица – дом, а затем используется в качестве типа данных поля Addr таблицы User:

```
CREATE TYPE address AS (  
    town varchar(20),  
    street varchar (20),  
    house varchar (20)  
);
```

```
CREATE TABLE User  
(  
    ...  
    Addr address,
```

...
);

Добавим новую запись, содержащую составной тип, выведем все адреса:

```
15 INSERT INTO usser (firstname, lastname, addr, email)
16 VALUES('Игорь', 'Робертс', ROW ('Ярославль', 'Свердлова', '45'),'robertsIg@bmstu.ru')
17 SELECT addr FROM usser
```

Рис.12. Добавление записи, содержащей структуру.

Обратимся к конкретному полю структуры, выведем все города, в которых проживают клиенты:

```
3 SELECT DISTINCT (addr).town FROM usser
```

	town character varying (20)
1	Санкт-Петербург
2	Калуга
3	Новосибирск
4	Тверь
5	Москва

Рис.13. Вывод всех городов клиентов.

Написание скобок необходимо, в противном случае тип addr воспринимается как имя таблицы.

- **Диапазоны** (<https://postgrespro.ru/docs/postgresql/12/rangetypes>)

Диапазонные типы представляют диапазоны значений некоторого типа данных. Любой непустой диапазон имеет две границы: верхнюю и нижнюю, и включает все точки между этими значениями.

PostgreSQL имеет следующие встроенные диапазонные типы:

- int4range — диапазон подтипа integer;
- int8range — диапазон подтипа bigint;
- numrange — диапазон подтипа numeric;
- tsrange — диапазон подтипа timestamp without time zone;
- tstzrange — диапазон подтипа timestamp with time zone;
- daterange — диапазон подтипа date.

Например, на рис. 14 производится добавление поля exec_duration диапазона tsrange в таблицу Order для указания продолжительности выполнения заказа.

```
59 ALTER TABLE order ADD COLUMN exec_duration tsrange
```

Рис. 14. Добавление диапазона

После добавления столбца `exec_duration` укажем продолжительность выполнения заказов и выведем их на экран (рис.15).

```
8 UPDATE orrder SET exec_duration = '["2019-10-20 12:10:00","2019-10-21 13:15:00"]'
9 WHERE orderid = 6
10
11 SELECT * FROM orrder
12 WHERE orderid BETWEEN 6 AND 7;
```







Data Output	Explain	Messages	Notifications	
 orderid [PK] integer 	accountid integer 	ammount_items integer 	ord_type ordertype 	exec_duration tsrange 
1	7	8	1 Изображение	["2019-10-23 13:45:00","2019-10-23 14:15:00"]
2	6	6	1 Изображение	["2019-10-20 12:10:00","2019-10-21 13:15:00"]

Рис.15. Определение продолжительности выполнения заказов

■ Геометрические типы (<https://postgrespro.ru/docs/postgresql/12/datatype-geometric>)

Геометрические типы данных представляют объекты в двумерном пространстве: точка

на плоскости, бесконечная прямая, отрезок, прямоугольник и т. д.

Для выполнения различных геометрических операций, в частности масштабирования, вращения и определения пересечений, PostgreSQL предлагает богатый набор функций и операторов.

Например, на рис. 16 в таблицу `Orrder` добавляется поле `sizes` типа «массив прямоугольников» для указания размеров изображений, подлежащих оцифровке, а затем вносятся данные о размерах изображений пользователя с `ID = 9`.

```
62 ALTER TABLE orrder ADD COLUMN sizes box [] ;
63
64 UPDATE orrder
65 SET sizes = '{(0,0) (1920,1080) ; (1920,1080)(0,0) ; (2560,1440)(0,0)}'
66 WHERE orderid = 9
```

Рис. 16. Добавление геометрического типа

Data Output		Explain	Messages	Notifications		
	orderid [PK] integer	accountid integer	ammount_items integer	ord_type ordertype	exec_duration tsrange	sizes box[]
1	4	5	1	Текст	["2019-08-23 10:45:00","2019-08-23 1...	[null]
2	8	5	3	Изображение	["2019-10-23 10:45:00","2019-10-23 1...	{{(749,749),(657,654);(754,978),(675,123)}}
3	9	5	3	Изображение	["2019-10-20 10:50:00","2019-10-22 1...	{{(1920,1080),(0,0);(1920,1080),(0,0);(2560,1440),(0,0)}}

Рис. 17. Размер изображений в таблице `order`

В PostgreSQL над геометрическими типами имеется возможность производить определённые геометрические операции, среди них: подсчёт площади, диаметра окружности, длины отрезка и т.д. Рассмотрим функцию подсчета площади:

```
11 select area(sizes[1]) as Area from orrder
```

	area	
	double precision	
1		[null]
2		8740
3		2073600

Рис. 17. Подсчёт площади геометрического объекта

- **XML** (<https://postgrespro.ru/docs/postgresql/12/datatype-xml>)

Тип xml предназначен для хранения XML-данных. Его преимущество по сравнению с обычным типом text в том, что он проверяет вводимые значения на допустимость по правилам XML, для работы с ним есть типобезопасные функции.

Для демонстрации работы с XML создадим таблицу с переменной типа XML (см. рис. 18).

```

37 CREATE TABLE text_info (
38     textid SERIAL PRIMARY KEY,
39     information XML
40 );

```

Рис. 18. Создание таблицы text_info

Внесем данные о тексте в формате XML в таблицу и посмотрим результат вывода запроса **SELECT** (рис. 19, 20).


```

41 INSERT INTO text_info (information) VALUES
42 ('
43     <BasicDetails>
44         <title>Sigh No More</title>
45         <type>poems</type>
46         <author>
47             <first_name>William</first_name>
48             <last_name>Shakespeare</last_name>
49         </author>
50     </BasicDetails>
51 ');
52 INSERT INTO text_info (information) VALUES
53 ('
54     <BasicDetails>
55         <title>Fahrenheit 451</title>
56         <type>novel</type>
57         <author>
58             <first_name>Ray</first_name>
59             <last_name>Bradbury</last_name>
60         </author>
61     </BasicDetails>
62 ');
63 SELECT
64     unnest(xpath('//title/text()', information::XML)) AS title
65     ,unnest(xpath('//type/text()', information::XML)) AS type
66     ,unnest(xpath('//first_name/text()', information::XML)) AS first_name
67     ,unnest(xpath('//last_name/text()', information::XML)) AS last_name
68 FROM text_info;

```

Рис. 19. Добавление данных XML и запрос

	title xml	type xml	first_name xml	last_name xml
1	Sigh N...	poems	William	Shakespeare
2	Fahren...	novel	Ray	Bradbury

Рис. 20. Вывод результата запроса

Более простой пример работы XML-запроса можно представить в виде конструкции xpath (рис. 21).

```

5 SELECT
6     (xpath( '//title/text()', information))[1]::text AS name,
7     (xpath( '//last_name/text()', information))[1]::text AS author
8 FROM text_info;

```

	name text	author text
1	Sigh No More	Shakespeare
2	Fahrenheit 451	Bradbury

- **JSON** (<https://postgrespro.ru/docs/postgresql/12/datatype-json>)

Типы JSON предназначены для хранения данных JSON (JavaScript Object Notation, Запись объекта JavaScript).

JSON – текстовый формат обмена и хранения данных, представляющий одну из двух структур:

- набор пар “ключ – значение”;
- упорядоченный набор значений.

Формат JSON поддерживается современными языками программирования, что позволяет унифицировать его в среде обмена данными.

В PostgreSQL имеются два типа для хранения данных JSON: JSON и JSONb.

Тип JSON сохраняет точную копию введенного текста, тогда как данные JSONb сохраняются в разобранном двоичном формате, что несколько замедляет ввод из-за преобразования, но значительно ускоряет обработку, не требуя многократного разбора текста. Кроме того, JSONb поддерживает индексацию, что тоже может быть очень полезно.

Изменим способ хранения информации для текста с XML на JSON. Для загружаемого пользователями текста создадим таблицу text_. Тип поля information определим как JSONb и будем вносить туда информацию о самом тексте. Назначение других полей таблицы будет рассмотрено далее.

```
1 CREATE TABLE text_ (  
2     textid SERIAL PRIMARY KEY,  
3     orderid integer,  
4     information JSONb ,  
5     body text  
6 );
```

Рис. 22. Создание таблицы text_

Для примера запишем несколько книг и статей:

```
INSERT INTO text_ (orderid, information)  
VALUES (3, '{"title": "Sigh No More", "type": "poems", "author": { "first_name": "William", "last_name": "Shakespeare"}}');  
INSERT INTO text_ (orderid, information)  
VALUES (8, '{"title": "Fahrenheit 451", "type": "novel", "author": { "first_name": "Ray", "last_name": "Bradbury"}}');  
INSERT INTO text_ (orderid, information)  
VALUES (11, '{"title": "Famed Roman statue not ancient", "type": "article", "author": { "first_name": "", "last_name": ""}}');  
INSERT INTO text_ (orderid, information)  
VALUES (12, '{"title": "Magic Mushrooms May Permanently Alter Personality", "type": "article", "author": { "first_name": "", "last_name": ""}}');
```

Рис. 23. Добавление книг и статей

Отообразим номер заказа, название текста и фамилию автора путем выполнения следующего запроса:

```

20 SELECT orderid,
21     information->'title' AS title, information->'author'->'last_name' AS author
22 FROM text_

```

	orderid integer	title jsonb	author jsonb
1	11	"Famed Roman statue not anci...	""
2	8	"Fahrenheit 451"	"Bradbury"
3	3	"Sigh No More"	"Shakespeare"
4	12	"Magic Mushrooms May Perm...	""
5	9	"The Worlds Largest Indoor Far...	""

Выведем все тексты, у которых не указан автор:

```

5 SELECT information->'title' AS title
6 FROM text_
7 WHERE information->'author'->'last_name' = ''

```

	title jsonb
1	"Famed Roman statue not ancient"
2	"Magic Mushrooms May Permanently Alter Personality"
3	"The Worlds Largest Indoor Farm Produces 10,000 Heads of Lettuce a Day in Japan"

Рис. 24. Запросы и результат их выполнения

Секционирование таблиц

PostgreSQL поддерживает простое секционирование таблиц. Секционированием таблиц называется разбиение одной большой логической таблицы на несколько маленьких физических секций. Секционирование может принести следующую пользу:

- а) в определённых ситуациях кардинально увеличивает быстродействие;
- б) удаление отдельной секции выполняется гораздо быстрее.

Применяется два типа секционирования: декларативное секционирование и секционирование с использованием наследования. Рассмотрим второй вариант.

Реализация этого метода производится в пять этапов

1. Создание главной таблицы, от которой будут наследоваться дочерние таблицы:

```

1 CREATE TABLE orders(
2     orderID SERIAL ,
3     accountID INTEGER,
4     FOREIGN KEY(accountID) REFERENCES Account (accountID) ON UPDATE CASCADE ON DELETE SET NULL,
5     ammount_items INTEGER DEFAULT 1 CHECK (Ammount_items > 0 AND Ammount_items < 99 ),
6     order_date date NOT NULL,
7     ord_type ordertype
8 ) ;

```

Рис. 25. Создание родительской таблицы

2. Создание нескольких дочерних таблиц. Для примера создадим две таблицы по принципу четный / нечетный номер заказа. Оператор **INHERITS** используется для наследования от родительской таблицы orders:

```

9 CREATE TABLE order_even() INHERITS (orders);
10 CREATE TABLE order_odd() INHERITS (orders);

```

Рис. 26. Создание дочерних таблиц

3. Добавление в таблицы неперекрывающихся ограничений с помощью конструкции:

CHECK (выражение)

Таким образом мы будем уверены, что каждый чётный или нечётный идентификатор записи попадёт в нужную таблицу:

```

10 Create table order_even (
11 CHECK (orderId % 2 = 0)
12 )INHERITS (orders);
13
14 Create table order_odd (
15 CHECK (orderId %2 !=0)
16 )INHERITS (orders);

```

Рис. 27. Добавление ограничений

4. Необходимо создать индекс для каждой дочерней таблицы по ключевому столбцу, а также любые другие индексы по своему усмотрению. Индекс – одно из средств увеличения производительности БД. Используя индексацию, сервер БД может быстрее находить и извлекать необходимые строки. Конструкция создания индекса выглядит так:

CREATE INDEX [имя] **ON** [имя_таблицы] (имя_столбца)

```

20 CREATE INDEX order_even_inx ON order_even(orderID);
21 CREATE INDEX order_odd_inx ON order_odd(orderID);

```

Рис. 28. Создание индексов

5. Для того, чтобы после выполнения запроса **INSERT INTO orders** (значение) данные

оказались в необходимой дочерней таблице, необходимо реализовать подходящую триггерную группу. Создадим ее с помощью триггерной функции, которая будет вызываться при изменении данных в базе данных.

Триггер будет связан с указанной таблицей, представлением или сторонней таблицей и будет выполнять заданную функцию при определённых событиях.

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).

Триггерная функция предотвращает изменения, фактически не влияющие на данные в строке, тогда как обычно изменения выполняются вне зависимости от того, были ли изменены данные.

В общем случае триггер объявляется так:

```
CREATE [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER | INSTEAD OF }  
  {событие [ OR ... ] }  
  ON имя_таблицы  
EXECUTE PROCEDURE имя_функции ( аргументы )
```

Триггер необходимо прописывать в триггерной функции. Общий синтаксис триггерной функции можно представить в виде:

```
CREATE OR REPLACE FUNCTION имя_функции  
RETURNS TRIGGER AS $$  
BEGIN  
    [тело функции]  
END;  
$$  
LANGUAGE plpgsql;
```

```
25 CREATE OR REPLACE FUNCTION order_insert_trigger ()  
26 RETURNS TRIGGER AS $$  
27 BEGIN  
28     IF (NEW.orderID % 2 = 0) THEN  
29         INSERT INTO order_even VALUES (NEW.*);  
30  
31 ELSEIF (NEW.orderID % 2 != 0) THEN  
32     INSERT INTO order_odd VALUES (NEW.*);  
33  
34     ELSE  
35         RAISE EXCEPTION  
36         'Error!!!';  
37     END IF;  
38     RETURN NULL;  
39 END;  
40 $$  
41 LANGUAGE plpgsql;  
42  
43 CREATE TRIGGER order_insert_trigger  
44     BEFORE INSERT ON orders  
45     FOR EACH ROW EXECUTE FUNCTION order_insert_trigger();
```

Рис. 29. Триггеры и триггерные функции

Аналогично напишем триггерную функцию на обновление данных

```

20 CREATE OR REPLACE FUNCTION order_update_trigger ()
21 RETURNS TRIGGER AS $$
22 BEGIN
23     IF (OLD.orderID % 2 = 0) THEN
24         UPDATE order_even
25         SET VALUES = (NEW.*)
26         WHERE orderID = OLD.orderID;
27
28     ELSEIF (OLD.orderID % 2 != 0) THEN
29         UPDATE order_even
30         SET VALUES = (NEW.*)
31         WHERE orderID = OLD.orderID;
32
33     ELSE
34         RAISE EXCEPTION
35         'Error???';
36     END IF;
37     RETURN NULL;
38 END;
39 $$
40 LANGUAGE plpgsql;
41
42 CREATE TRIGGER order_update_trigger
43 BEFORE INSERT ON orders
44 FOR EACH ROW EXECUTE FUNCTION order_update_trigger();

```

Рис. 30. Обновление с помощью триггера

Прокомментируем работу кода: в строке 25 объявим триггерную функцию `order_insert_trigger()`, в строке 28 и 31 создадим два неперекрывающихся условия `NEW.orderID % 2 = 0` и `NEW.orderID % 2 != 0`

Переменная **NEW** содержит новую строку базы данных для команд **INSERT/UPDATE**.

Переменная **OLD** содержит старую строку базы данных для команд **UPDATE/DELETE**.

Таким образом мы проверяем остаток от деления на 2 нового `orderID`, и, если он равен нулю, заносим всю строку (**VALUES (NEW.*)**) в «чётную» таблицу. Аналогично для нечётного идентификатора.

В строке 43 создаётся триггер `order_insert_trigger`. Условием на его срабатывание является добавление записи в таблицу (**BEFORE INSERT ON orders**).

Запись **FOR EACH ROW EXECUTE FUNCTION** `order_insert_trigger ()` означает, что триггерная функция будет срабатывать для каждой добавляемой строки.

После этого вносим данные в таблицы.

```

47 insert into orders (accountid, ammount_items, order_date, ord_type)
48 values ('4', '2', '12.03.2020', 'Изображение');
49
50 insert into orders (accountid, ammount_items, order_date, ord_type)
51 values ('5', '2', '12.03.2020', 'Изображение');

```

Рис. 31. Добавление данных

Теперь для просмотра результата выведем дочерние таблицы и убедимся, что в каждую дочернюю таблицу попал соответствующий результат.






	 orderid integer	 accountid integer	 ammount_items integer	 order_date date	 ord_type ordertype
1	3	5	2	2020-03-12	Изображение

Рис. 32. Таблица order_odd






	 orderid integer	 accountid integer	 ammount_items integer	 order_date date	 ord_type ordertype
1	2	4	2	2020-03-12	Текст

Рис. 33. Таблица order_even

Хранимые функции

Хранимые процедуры – это функции, представляющие собой набор инструкций, которые выполняются как единое целое, хранятся на сервере. Написание хранимых процедур позволяет ограничить доступ к данным и таблицам и тем самым обезопасить их от непреднамеренного изменения данных. <https://postgrespro.ru/docs/postgrespro/12/sql-createprocedure>.

■ Динамический запрос

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. Для исполнения динамических команд предусмотрен оператор EXECUTE:

EXECUTE строка-команды [**INTO** [**STRICT**] цель] [**USING** выражение],

где строка-команды - это выражение, формирующее строку с самим текстом команды, которую нужно выполнить. Цель — это переменная, куда будут помещены результаты команды. Выражения в **USING** формируют значения, которые будут вставлены в команду.

В тексте команды можно использовать значения параметров, ссылки на параметры обозначаются как \$1, \$2 и т. д. Эти символы указывают на значения, находящиеся в предложении **USING**. Символы параметров можно использовать только вместо значений данных. Если же требуется динамически формировать имена таблиц или столбцов, их необходимо вставлять в виде текста. Вместо имени таблиц или столбцов можно

использовать указание формата %I с функцией format() (текст, разделённый символами новой строки, соединяется вместе).

Например, создадим функцию «select_dynamic» с динамическим запросом, которая будет принимать на вход название таблицы (some_table), некоторую дату, название колонки с типом date в этой таблице (some_date) и некоторое поле произвольного типа в этой таблице (some_element), а возвращать будет значение поля some_element, соответствующее дате в поле some_date, равной введенной дате.

```

2 CREATE OR REPLACE FUNCTION select_dynamic(some_table text, some_date date, some_column text, some_element anyelement,
3                                         OUT select_result anyelement)
4 RETURNS anyelement AS
5 $$
6 BEGIN
7     EXECUTE format('SELECT %I FROM %I WHERE %I = $1 ORDER BY %I',
8                   some_element, some_table, some_column, some_element)
9     INTO select_result
10    USING some_date;
11 END;
12 $$
13 LANGUAGE plpgsql;

```

Рис. 55. Создание хранимой функции с динамическим запросом

Такую функцию можно вызвать, например, чтобы найти e-mail пользователя с нужной датой рождения.

```

16 SELECT select_dynamic('usser', '2000-09-04', 'birthday', 'email'::varchar);

```

Data Output	Explain	M
select_dynamic character varying		
1	mtfo@mail.ru	

Рис. 56. Вызов функции с динамическим запросом

■ Циклы и условия

В PostgreSQL применяется несколько видов условных операторов <https://postgrespro.ru/docs/postgrespro/12/plpgsql-control-structures> :

1. IF-THEN

IF логическое-выражение **THEN**

операторы

END IF;

2. IF-THEN-ELSE

IF логическое-выражение **THEN**

операторы

ELSE

операторы

END IF;

3. IF-THEN-ELSEIF

IF логическое-выражение **THEN**

операторы

[**ELSIF** логическое-выражение **THEN** операторы [**ELSIF** логическое-выражение **THEN** операторы ...]]

[**ELSE** операторы]

END IF;

4. Простой CASE

CASE выражение-поиска

WHEN выражение [, выражение [...]] **THEN**

операторы

[**WHEN** выражение [, выражение [...]] **THEN** операторы ...]

[**ELSE** операторы]

END CASE;

5. CASE с перебором условий

CASE

WHEN логическое-выражение **THEN**

операторы

[**WHEN** логическое-выражение **THEN** операторы ...]

[**ELSE** операторы]

END CASE;

Применение некоторых из них уже встречалось выше.

Циклы в PostgreSQL бывают простые, такие как:

1. LOOP

[<<метка>>]

LOOP

операторы

END LOOP [метка];

2. EXIT

EXIT [метка] [**WHEN** логическое-выражение];

3. CONTINUE

CONTINUE [метка] [**WHEN** логическое-выражение];

4. WHILE

[<<метка>>]

WHILE логическое-выражение **LOOP**

операторы

END LOOP [метка];

5. FOR

[<<метка>>]

FOR имя **IN** [**REVERSE**] выражение .. выражение [**BY** выражение] **LOOP**

операторы

END LOOP [метка];

А также бывают циклы по запросам и по элементам массива.

В качестве примера рассмотрим хранимую процедуру, содержащую цикл по элементам массива. Такая функция будет подсчитывать общий размер файлов текста в заказе в зависимости от введенного идентификатора заказа (см. рис. 57).

```
9 CREATE FUNCTION size_of_order (n integer) RETURNS integer AS
10 $$
11 DECLARE
12     size_of_ord INTEGER := 0;
13     counter INTEGER := 0;
14     col integer := 0;
15     arr integer[] ;
16 BEGIN
17     SELECT INTO col (SELECT array_length(text_sizes, 1) FROM orrder where orderid = n);
18     SELECT INTO arr (select text_sizes FROM orrder where orderid = n);
19 LOOP
20     EXIT WHEN counter = col;
21     counter := counter + 1;
22     size_of_ord := size_of_ord + arr[counter];
23 END LOOP;
24 RETURN size_of_ord;
25 END;
26 $$ LANGUAGE plpgsql;
27
28 Select size_of_order(5) as q;
```

Рис. 57. Создание и вызов хранимой процедуры цикла

Поясним работу функции: в строках 11-15 объявим переменные, которые будут использоваться в хранимой процедуре. В строках 17, 18 присваиваем переменным col, arr

результат выполнения запросов. В переменную `col` запишем количество элементов в массиве `text_sizes`. Скопируем массив `text_sizes` в переменную `arr`. Объявим цикл в строке 19, условием выхода из цикла будет полный перебор всех элементов массива. Увеличиваем счетчик `counter` на каждой итерации цикла, добавляем значения размера текущего элемента массива к общему. Возвращаем суммарный размер массива.

▪ Работа с файловой системой

Рассмотрим применение хранимой процедуры для чтения файла, хранящегося в локальной директории. Считываемый файл будем заносить в таблицу. Такой метод подходит для случаев, когда информация представлена в текстовом файле, допустим, CSV. <https://postgrespro.ru/docs/postgrespro/12/sql-copy>.

Для этого сначала создадим таблицу `image` (рис. 58).

```
8 CREATE TABLE image (  
9     imageid SERIAL PRIMARY KEY,  
10    orderid integer ,  
11    imgsize integer  
12 );
```

Рис. 58. Создание таблицы `image`

Подготовив данные в формате CSV, напомним хранимую процедуру, принимающую параметр выполнения операции чтения/записи. Для копирования информации с диска необходимо выполнить функцию:

```
COPY имя_таблицы(атрибуты)  
FROM 'путь' DELIMITER ',' CSV HEADER;
```

Для выгрузки информации из базы данных используется функция:

```
COPY ( SELECT * FROM имя_таблицы )  
TO 'путь' WITH CSV DELIMITER ',' ;
```

Теперь в зависимости от параметра информация будет либо загружаться, либо выгружаться из базы данных (см. рис. 59).

```

15 CREATE OR REPLACE FUNCTION CSV_INSERT (param int) RETURNS integer
16 AS $$
17 BEGIN
18     IF (param = 1) THEN
19         COPY image (orderid, imgsize)
20         FROM 'C:\downloads\2\data.csv' DELIMITER ',' CSV HEADER;
21         RETURN 1;
22     ELSEIF (param = 2) THEN
23         COPY (SELECT * FROM image) To 'C:\downloads\export.csv' With CSV DELIMITER ',';
24         RETURN 1;
25     ELSE
26         RAISE EXCEPTION
27         'ERROR!!!';
28         RETURN 0;
29     END IF ;
30 END;
31 $$
32 LANGUAGE plpgsql;
33
34 SELECT CSV_INSERT(2);

```

Рис. 59. Процедура работы с локальными файлами и её вызов

Учебная литература

1. Документация PostgreSQL [электронный ресурс]. – Режим доступа: <https://postgrespro.ru/docs/postgresql/>, свободный.