



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

ФАКУЛЬТЕТ Информатика, искусственный интеллект и системы управления

КАФЕДРА Системы обработки информации и управления

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

**Лабораторная работа №1
«Создание объектно-реляционной базы данных
на примере СУБД PostgreSQL»**

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

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

Москва, 2022 г.

ОГЛАВЛЕНИЕ

1. ЗАДАНИЕ.....	3
1.1. Цель работы.....	3
1.2. Средства выполнения.....	3
1.3. Продолжительность работы.....	3
1.4. Пункты задания для выполнения.....	3
1.5. Дополнительное задание.....	4
1.6. Содержание отчета.....	4
2. ТЕОРЕТИКО-ПРАКТИЧЕСКИЙ МАТЕРИАЛ.....	5
2.1. СУБД PostgreSQL.....	5
2.1.1. Установка PostgreSQL на ОС Windows.....	6
2.1.2. Работа с PostgreSQL на платформе pgAdmin.....	7
2.2. Работа с таблицами БД.....	18
2.2.1. Создание таблиц.....	18
2.2.2. Форматирование.....	19
2.2.3. Типы данных.....	19
2.2.4. Удаление таблиц.....	19
2.2.5. Изменение структуры таблицы.....	19
2.2.6. Ограничения.....	20
2.2.7. Основные операторы SQL.....	27
2.3. Сложные типы данных PostgreSQL.....	33
2.3.1. Массивы.....	33
2.3.2. Структурные типы.....	35
2.3.3. Перечисления.....	36
2.3.4. Наследование.....	37
2.4. Пользовательские типы.....	41
2.4.1. Описание пользовательского типа.....	41
2.4.2. Определение созданного типа в PostgreSQL.....	43
2.4.3. Пример создания пользовательского типа.....	46
3. КОНТРОЛЬНЫЕ ВОПРОСЫ.....	55
4. СПИСОК ИСТОЧНИКОВ.....	56

1. ЗАДАНИЕ

Лабораторная работа №1 «Создание объектно-реляционной базы данных на примере СУБД PostgreSQL» по курсу «Постреляционные базы данных».

1.1. Цель работы

- Изучить объектно-реляционную модель данных [1] и возможности языка SQL для работы с ней;
- Освоить построение запросов на языке SQL для описания ограничений целостности, определения и использования полей сложных типов данных и наследования таблиц;
- Получить навыки создания объектно-реляционной базы данных и выполнения запросов к ней на примере СУБД PostgreSQL [2] в среде PgAdmin [3].

1.2. Средства выполнения

- СУБД PostgreSQL,
- Утилита PgAdmin.

1.3. Продолжительность работы

Время выполнения лабораторной работы 4 часа.

1.4. Пункты задания для выполнения

1. В среде PgAdmin (PostgreSQL) создать БД. В БД создать две-три таблицы по теме, выданной преподавателем, связанные как один-ко-многим, содержащие первичные и внешние ключи и ограничения: уникальности, Not NULL, на значения, на значение по умолчанию, автоинкремент. Открыть таблицы на редактирование и заполнить тестовыми данными. Проверить действие ограничений.
2. Добавить в таблицу поле типа массив и продемонстрировать работу с ним.
3. Добавить в таблицу поле типа структуры на основе структурного типа и продемонстрировать работу с ним.

4. Создать собственный перечислимый тип и добавить, как поле в таблицу. Продemonстрировать работу с типом.
5. Создать производную таблицу с дополнительным полем. Продemonстрировать работу с базовыми и производными таблицами (CRUD). Вывести отдельно записи базовой и производной таблиц, удовлетворяющих некоторому условию.

1.5. Дополнительное задание

6. Создать собственный тип данных, определить для него функции сортировки, ввода и вывода. Добавить тип данных как поле в таблицу. Продemonстрировать работу с типом.

1.6. Содержание отчета

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

2. ТЕОРЕТИКО-ПРАКТИЧЕСКИЙ МАТЕРИАЛ

2.1. СУБД PostgreSQL

PostgreSQL — это объектно-реляционная система управления базами данных (ОРСУБД, ORDBMS), основанная на POSTGRES Version 4.2 — программе, разработанной на факультете компьютерных наук Калифорнийского университета в Беркли (руководитель проекта — профессор Массачусетского технологического института Майкл Стоунбрейкер).

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.

СУБД PostgreSQL реализована 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.

2.1.1. Установка PostgreSQL на ОС Windows

С полной документацией по PostgreSQL можно ознакомиться в [2].

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

Установите PostgreSQL (см. Рисунок 1).

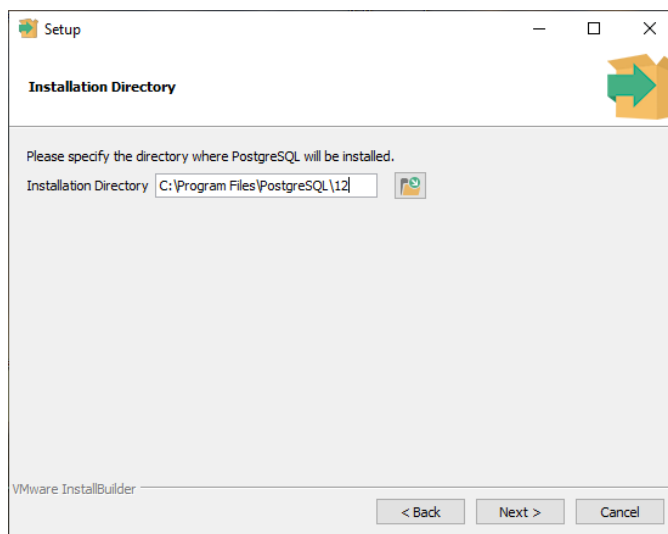


Рисунок 1 – Процесс установки PostgreSQL, шаг 1

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

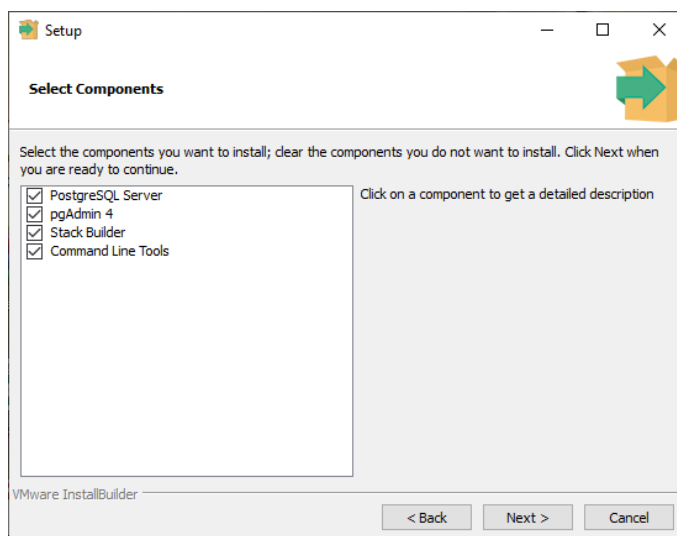


Рисунок 2 – Процесс установки PostgreSQL, шаг 2

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

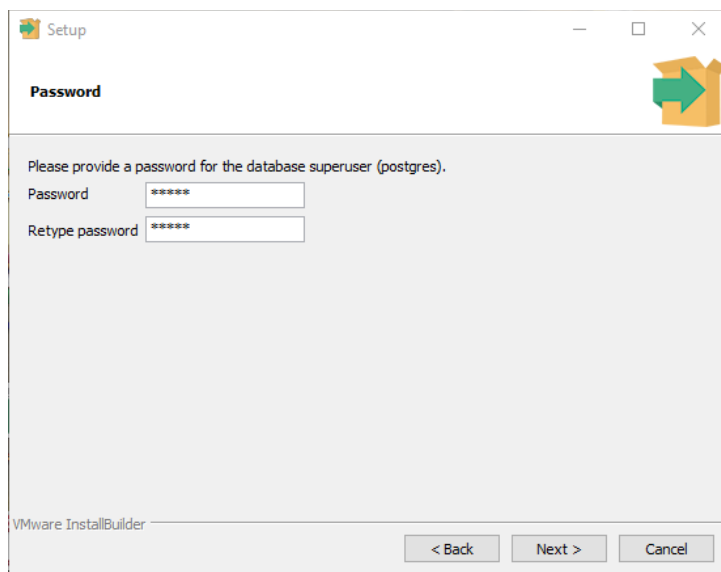


Рисунок 3 – Процесс установки PostgreSQL, ввод пароля

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

2.1.2. 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. Рисунок 4):

- слева – дерево всех объектов,
- по центру – данные о конкретном объекте,
- справа – операторы, использовавшиеся для создания этого объекта.

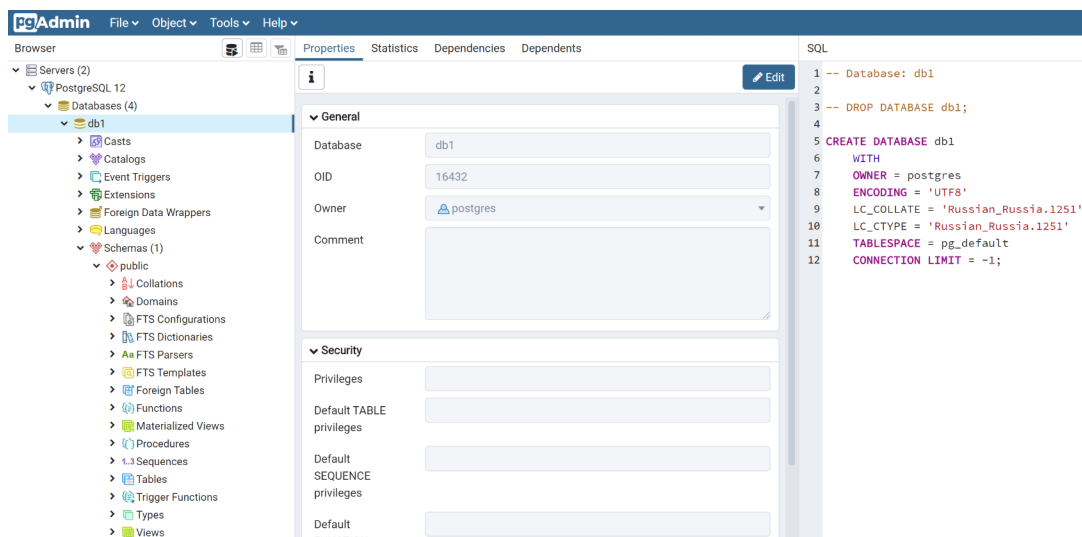


Рисунок 4 – Интерфейс pgAdmin

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

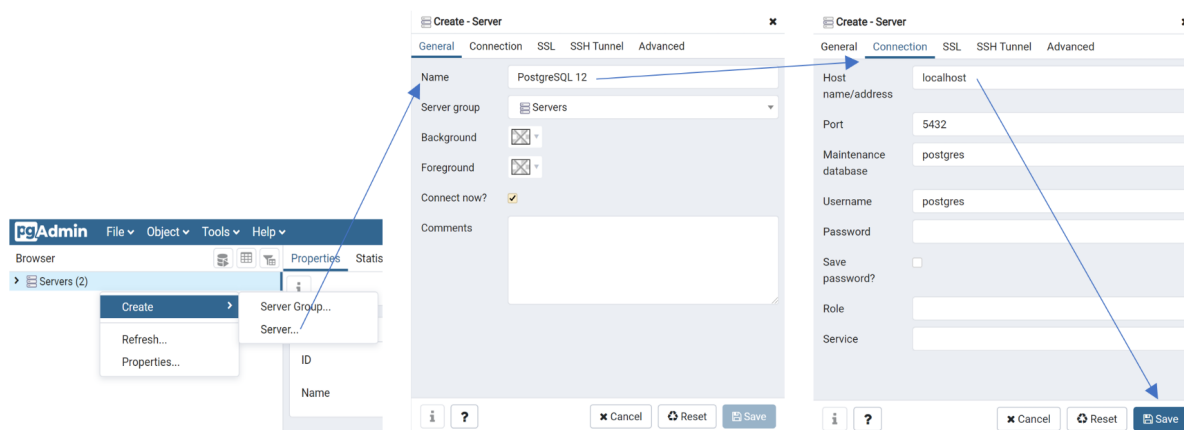


Рисунок 5 – Создание сервера

Для создания базы данных требуется выбрать сервер, кликнуть правой кнопкой мыши на пункт **Databases** (Базы Данных) → **Create** → **Database** (см. Рисунок 6). Введите название базы данных в поле **Database**. Нажать кнопку «Save».

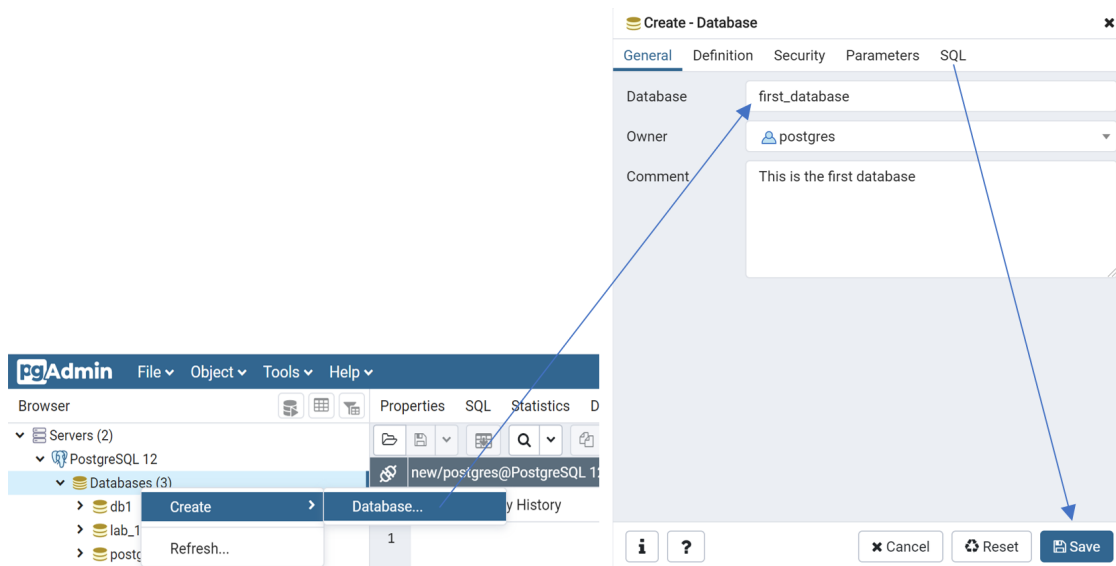


Рисунок 6 - Создание базы данных

Чтобы просмотреть базу данных, необходимо нажать стрелку рядом с Databases, чтобы раскрыть список баз данных (см. Рисунок 7). Аналогично далее можно раскрывать содержимое баз данных.

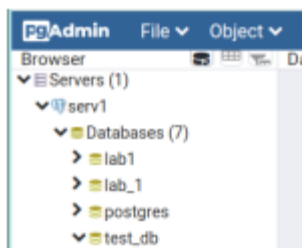


Рисунок 7 - Просмотр списка баз данных

Для создания таблицы необходимо раскрыть **Databases** → Ваша база данных → **Schemas**. Затем нажать правой кнопкой мыши на **Tables** (Таблицы) и выбрать **Create** → **Table** (см. Рисунок 8).

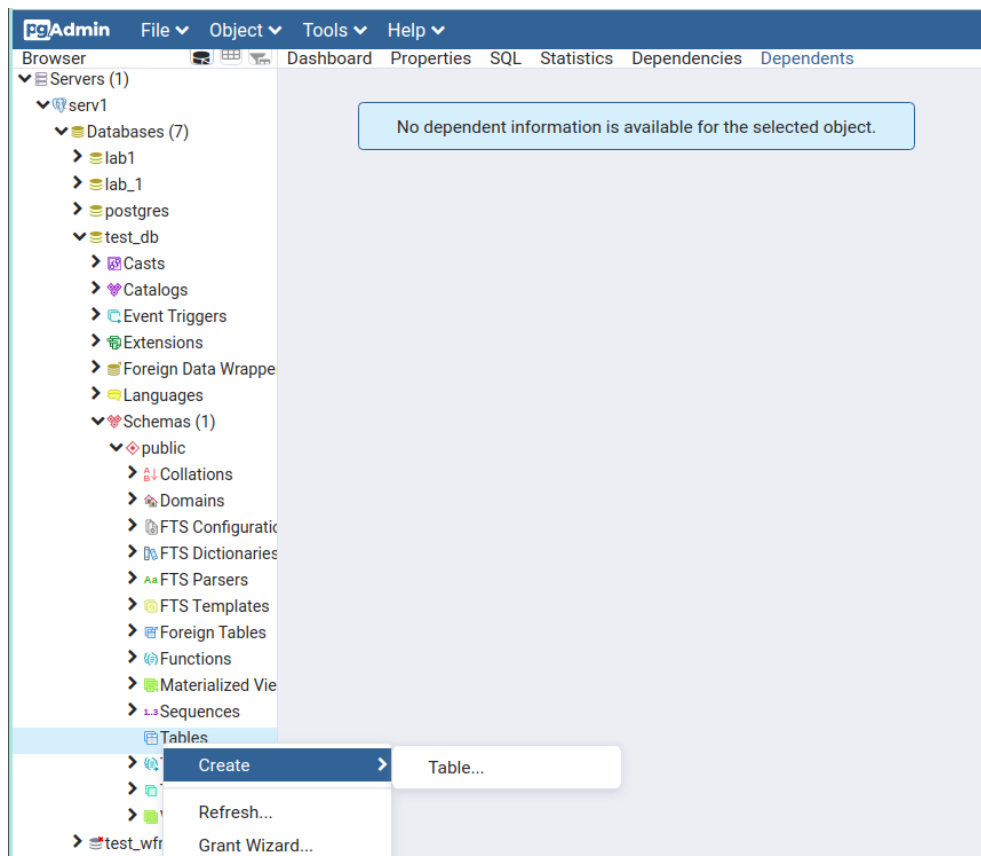


Рисунок 8 - Создание таблицы

В окне создания таблицы ввести название в поле **Name** (см. Рисунок 9).

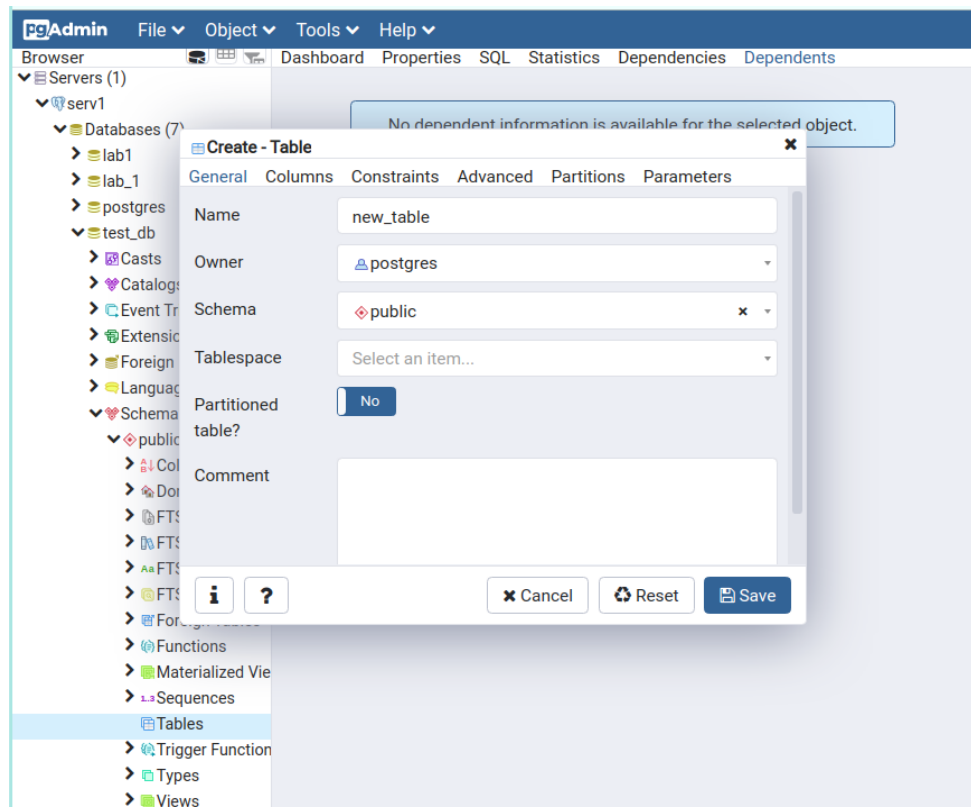


Рисунок 9 - Ввод названия таблицы

Колонки в таблицах можно создать при создании таблицы (см. Рисунок 10). Для добавления колонки 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**».

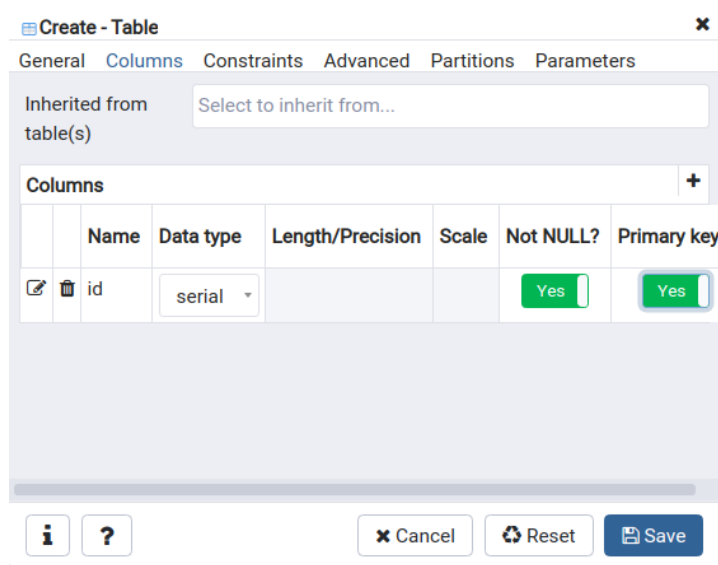


Рисунок 10 - Создание колонок при создании таблицы

Колонки также можно добавить и к уже прежде созданной таблице (см. Рисунок 11). Для этого необходимо раскрыть нужную таблицу, нажать правой кнопкой мыши на **Columns**, выбрать **Create** → **Column**.

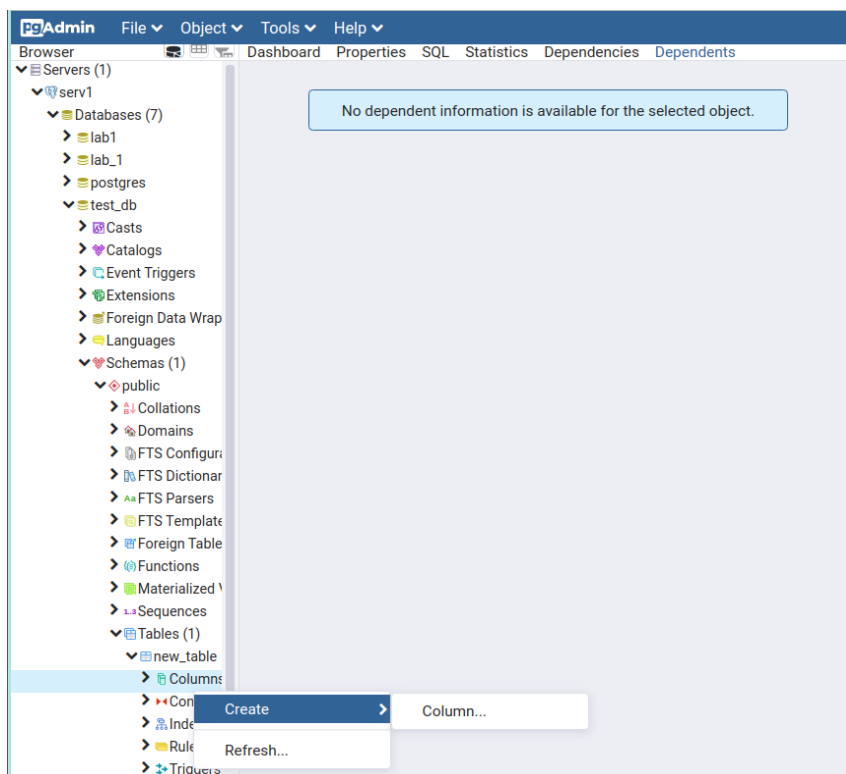
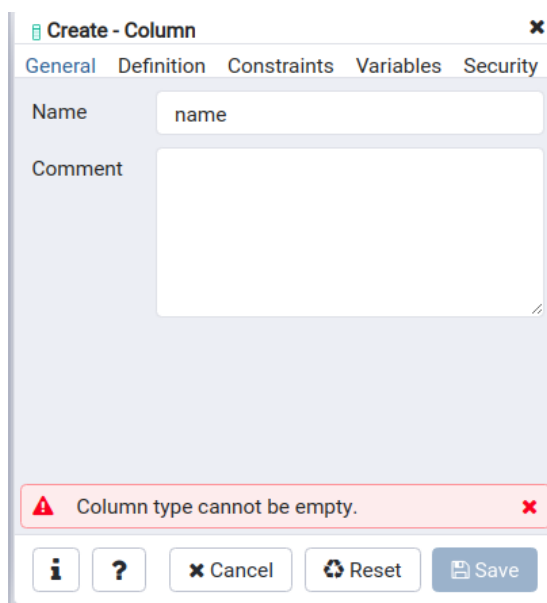


Рисунок 11 - Создание колонок для существующей таблицы

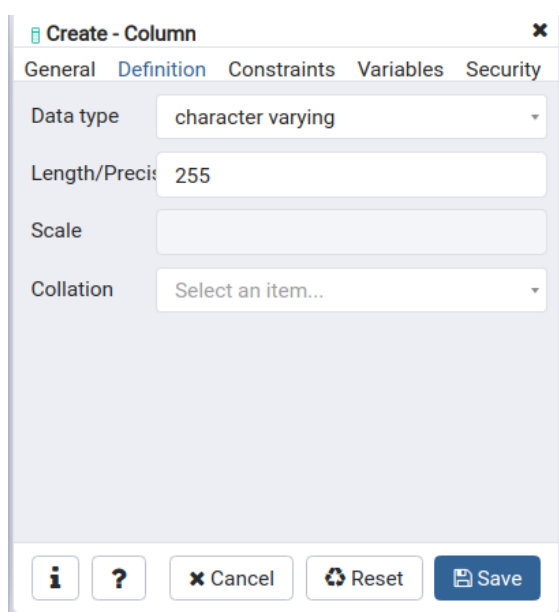
В окне создания колонки ввести название в поле **Name** (см. Рисунок 12).



The screenshot shows the 'Create - Column' dialog box with the 'General' tab selected. The 'Name' field is filled with 'name'. The 'Comment' field is empty. At the bottom, a red error message box displays the text 'Column type cannot be empty.' with a red 'X' icon. Below the error message are buttons for 'i', '?', 'Cancel', 'Reset', and 'Save'.

Рисунок 12 - Ввод названия колонки

Во вкладке **Definition** выбрать тип данных и задать длину поля (см. Рисунок 13).



The screenshot shows the 'Create - Column' dialog box with the 'Definition' tab selected. The 'Data type' dropdown is set to 'character varying'. The 'Length/Precision' field is set to '255'. The 'Scale' field is empty. The 'Collation' dropdown is set to 'Select an item...'. At the bottom are buttons for 'i', '?', 'Cancel', 'Reset', and 'Save'.

Рисунок 13 - Определение типа данных колонки

Во вкладке **Constraints** при необходимости установить ограничения (см. Рисунок 14). Сохранить столбец. Нажать кнопку «**Save**».

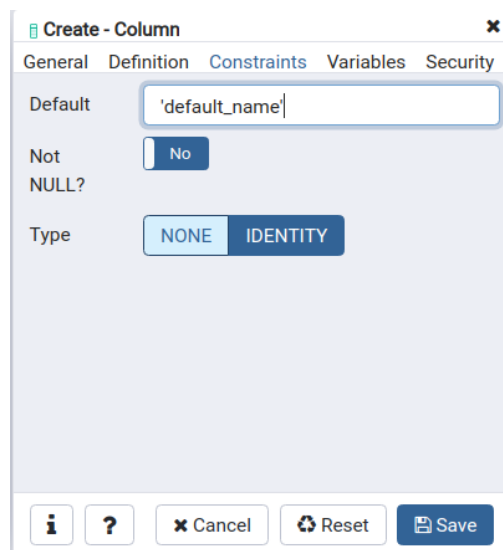


Рисунок 14 - Установка ограничений колонки

Для редактирования данных нажать на необходимую таблицу правой кнопкой мыши и выбрать пункт View/Edit Data и необходимые строки (см. Рисунок 15).

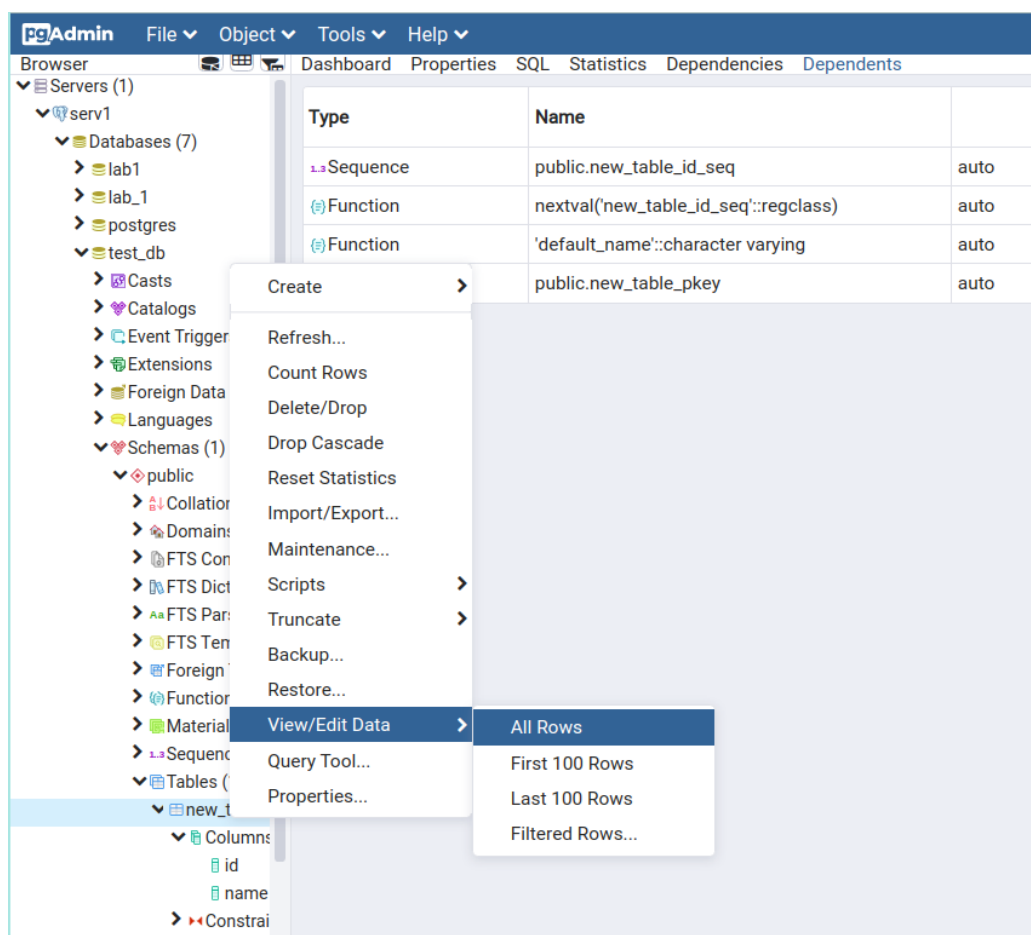


Рисунок 15 – Просмотр или редактирование данных

В окне редактирования данных ввести данные в Data Output и сохранить (см. Рисунок 16), нажав либо F6, либо кнопку, обведенную на рисунке красным прямоугольником.

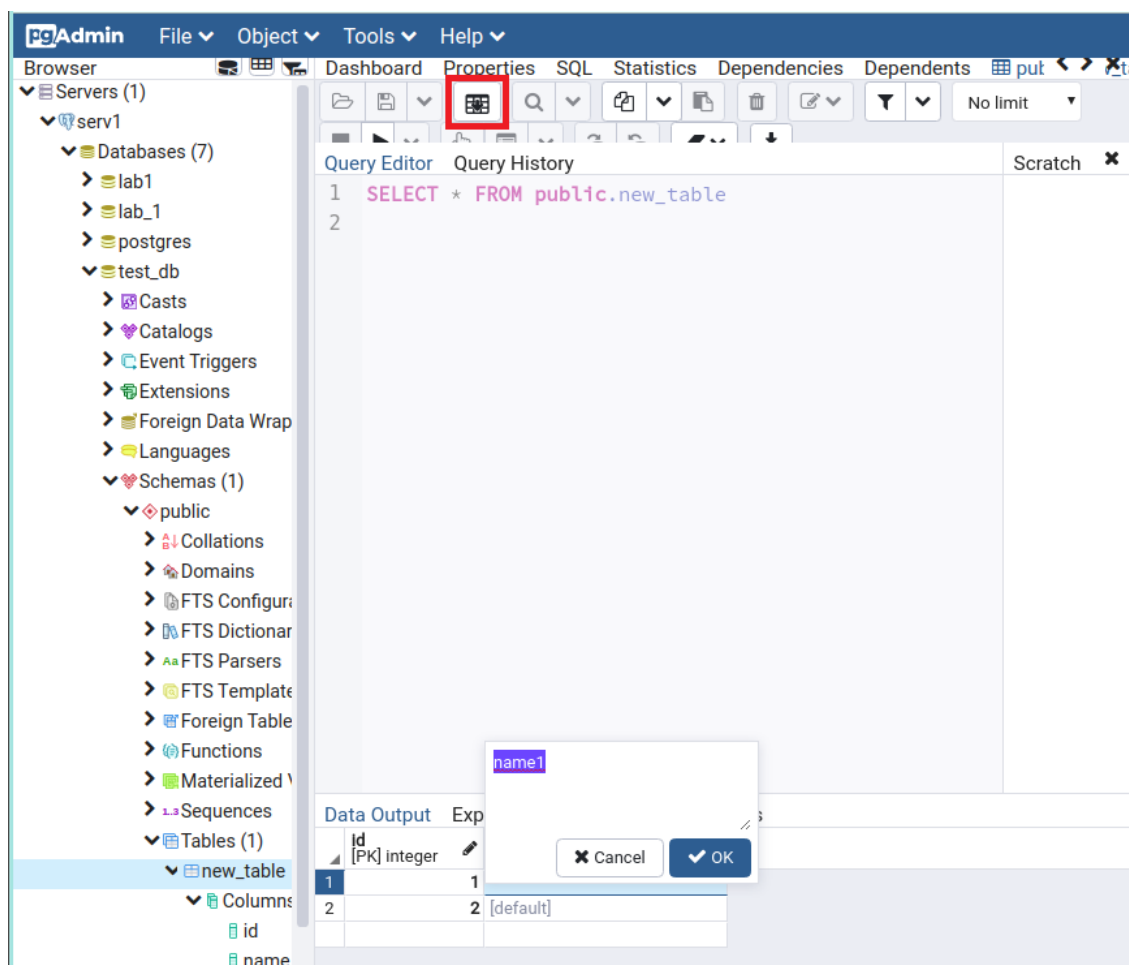


Рисунок 16 - Редактирование данных таблицы

После создания базы данных и таблиц в ней можно приступить к написанию запросов, которые выполняются при помощи **Query Tool** (см. Рисунок 17). 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 правой кнопкой мыши (ПКМ) по интересующему объекту. Просматривать содержимое таблиц можно, используя кнопку **View Data**, находящуюся в окне **Browser**.

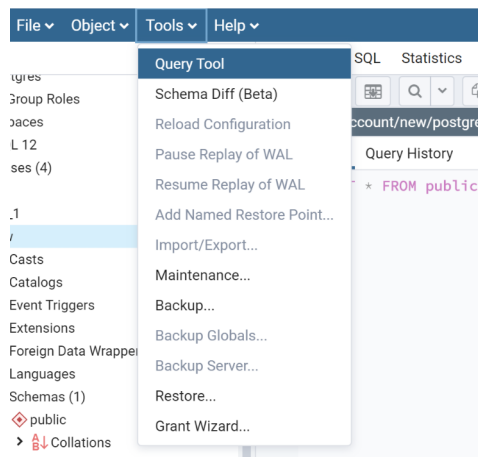


Рисунок 17 – Открытие Query Editor

2.2. Работа с таблицами БД

2.2.1. Создание таблиц

Команда **Create Table** предназначена для описания структуры таблицы. Команда SQL **Create Table** создает пустую таблицу (без строк).

Вы можете создать таблицу с помощью SQL, указав её имя и перечислив все имена столбцов и их типы:

```
CREATE TABLE table_name (
    column1 datatype ограничение(если есть),
    column2 datatype ограничение(если есть),
    column3 datatype ограничение(если есть),
    ....
);
```

Где **column1** и т.д. – название столбца, **datatype** – стандартный или пользовательский тип данных, **ограничение** – накладываемое на столбец ограничение данных.

К примеру:

```
CREATE TABLE weather (
    city varchar(80),
    temp_lo int, -- минимальная температура дня
    temp_hi int, -- максимальная температура дня
    prcp real, -- уровень осадков
    date date
);
```

Весь этот текст можно ввести в PostgreSQL вместе с символами перевода строк. PostgreSQL понимает, что команда продолжается до точки с запятой.

2.2.2. Форматирование

В командах SQL можно свободно использовать пробельные символы (пробелы, табуляции и переводы строк). Это значит, что вы можете ввести команду, выровняв её по-другому или даже уместив в одной строке. Два минуса («-») обозначают начало комментария. Всё, что идёт за ними до конца строки, игнорируется. SQL не чувствителен к регистру в ключевых словах и идентификаторах, за исключением идентификаторов, взятых в кавычки (в данном случае это не так).

2.2.3. Типы данных

Тип `varchar(80)` в примере определяет тип данных, допускающий хранение произвольных символьных строк длиной до 80 символов. `int` — обычный целочисленный тип. `real` — тип для хранения чисел с плавающей точкой одинарной точности. `date` — тип даты.

PostgreSQL поддерживает стандартные типы SQL: `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp` и `interval`, а также другие универсальные типы и богатый набор геометрических типов. Кроме того, PostgreSQL можно расширять, создавая набор собственных типов данных. Как следствие, имена типов не являются ключевыми словами в данной записи, кроме тех случаев, когда это требуется для реализации особых конструкций стандарта SQL.

2.2.4. Удаление таблиц

Если вам больше не нужна какая-либо таблица, или вы хотите пересоздать её по-другому, вы можете удалить её, используя следующую команду:

```
DROP TABLE имя_таблицы;
```

2.2.5. Изменение структуры таблицы

Вы можете изменять структуру таблицы с помощью команды `ALTER TABLE`.

Команда `ALTER TABLE` используется для добавления, удаления или модификации колонки в уже существующей таблице.

Добавление столбца в таблицу:

```
ALTER TABLE t1(pole1 char(10));
```


Изменение типа столбца таблицы. Команда будет успешна, только если все существующие значения в столбце могут быть неявно приведены к новому типу:

```
ALTER TABLE t1 ALTER COLUMN name TYPE INTEGER;
```

Удаление столбца таблицы:

```
ALTER TABLE t1 DROP COLUMN pole1;
```

С помощью команды ALTER TABLE можно изменить имя таблицы без реального переноса физической информации в БД:

```
ALTER TABLE t1 RENAME TO t2;
```

2.2.6. Ограничения

2.2.6.1. Ограничение-проверка

Наиболее общий тип ограничений. В его определении вы можете указать, что значение данного столбца должно удовлетворять логическому выражению (проверке истинности).

В PostgreSQL поддерживаются следующие типы ограничений: на уникальность, на допустимость значения NULL, первичный ключ, внешний ключ, ограничения общего вида.

Общий вид ограничений-проверок:

```
CREATE TABLE tablename (  
    field1 fieldtype,  
    field2 fieldtype,  
    field3 fieldtype CHECK (condition)  
);
```

Например, цену товара можно ограничить положительными значениями так:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0)  
);
```

Вы можете также присвоить ограничению отдельное имя. Это улучшит сообщения об ошибках и позволит вам ссылаться на это ограничение, когда вам понадобится изменить его. Сделать это можно так:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CONSTRAINT positive_price CHECK (price > 0)  
);
```

То есть, чтобы создать именованное ограничение, напишите ключевое слово **CONSTRAINT**, а за ним идентификатор и собственно определение ограничения. (Если вы не определите имя ограничения таким образом, система выберет для него имя за вас.)

Ограничение-проверка может также ссылаться на несколько столбцов. Например, если вы храните обычную цену и цену со скидкой, так вы можете гарантировать, что цена со скидкой будет всегда меньше обычной:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0),  
    discounted_price numeric CHECK (discounted_price > 0),  
    CHECK (price > discounted_price)  
);
```

Вы можете также использовать **CHECK** в качестве табличного ограничения. Это полезно в тех случаях, когда вы хотите включить более одного пол строки в условие. Вы можете указать это со следующим табличным ограничением **CHECK**:

```
CREATE TABLE Salespeople (  
    snum integer NOT NULL UNIQUE,  
    sname char (10) NOT NULL UNIQUE,  
    city char(10),  
    comm decimal,  
    CHECK (comm < .15 OR city = 'Barcelona')  
);
```

2.2.6.2. Ограничения *NOT NULL*

Ограничение `NOT NULL` просто указывает, что столбцу нельзя присваивать значение `NULL`. Пример синтаксиса:

```
CREATE TABLE products (  
    product_no integer NOT NULL,  
    name text NOT NULL,  
    price numeric  
);
```

2.2.6.3. Ограничения уникальности

Ограничения уникальности гарантируют, что данные в определённом столбце или группе столбцов уникальны среди всех строк таблицы. Ограничение записывается так:

```
CREATE TABLE products (  
    product_no integer UNIQUE,  
    name text,  
    price numeric);
```

Чтобы определить ограничение уникальности для группы столбцов, запишите его в виде ограничения таблицы, перечислив имена столбцов через запятую:

```
CREATE TABLE example (  
    a integer,  
    b integer,  
    c integer,  
    UNIQUE (a, c)  
);
```

2.2.6.4. Значение по умолчанию

Ограничение `DEFAULT` используется для предоставления значения по умолчанию для столбца. Значение по умолчанию будет добавлено ко всем новым записям, если другое значение не указано.

Следующий код SQL устанавливает значение `DEFAULT` для столбца `country`, когда создается таблица `users`:

```
CREATE TABLE users (  
    user_id int NOT NULL,  
    name varchar(255) NOT NULL,  
    gender int,  
    country varchar(255) DEFAULT 'Spain'  
);
```

2.2.6.5. Первичные ключи

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов является уникальным идентификатором строк в таблице. Для этого требуется, чтобы значения были одновременно уникальными и отличными от NULL. Таким образом, следующими две таблицы будут принимать одинаковые данные:

```
CREATE TABLE products (  
    product_no integer UNIQUE NOT NULL,  
    name text,  
    price numeric  
);
```

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

Первичные ключи могут включать несколько столбцов; синтаксис похож на запись ограничений уникальности:

```
CREATE TABLE example (  
    a integer,  
    b integer,  
    c integer,  
    PRIMARY KEY (a, c)  
);
```

При добавлении ограничения уникальности будет автоматически создан уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в ограничении., и данные столбцы помечаются как NOT NULL.

Таблица может иметь максимум один первичный ключ. (Ограничений уникальности и ограничений NOT NULL, которые функционально почти равнозначны первичным ключам, может быть сколько угодно, но назначить ограничением первичного ключа можно только одно.) Теория реляционных баз данных требует, чтобы первичный ключ был в каждой таблице.

2.2.6.6. Внешние ключи

Внешние ключи позволяют установить связи между таблицами. Внешний ключ устанавливается для столбцов из зависимой, подчиненной таблицы, и указывает на один из столбцов из главной таблицы. Как правило, внешний ключ указывает на первичный ключ из связанной главной таблицы.

Общий синтаксис установки внешнего ключа на уровне таблицы:

```
[CONSTRAINT имя_ограничения]
FOREIGN KEY (столбец1, столбец2, ... столбецN)
REFERENCES главная_таблица (столбец_главной_таблицы1,
столбец_главной_таблицы2)
[ON DELETE действие]
[ON UPDATE действие]
```

Для создания ограничения внешнего ключа после FOREIGN KEY указывается столбец таблицы, который будет представлять внешний ключ. После ключевого слова REFERENCES указывается имя связанной таблицы, а затем в скобках – имя связанного столбца, на который будет указывать внешний ключ. После выражения REFERENCES идут выражения ON DELETE и ON UPDATE, которые задают действие при удалении и обновлении строки из главной таблицы.

Например, определим две таблицы и свяжем их посредством внешнего ключа:

```
CREATE TABLE Customers
(
    Id INT PRIMARY KEY,
    Age INT,
    FirstName VARCHAR(20) NOT NULL,
```

```

        LastName VARCHAR(20) NOT NULL,
        Phone VARCHAR(20) NOT NULL UNIQUE
    );

CREATE TABLE Orders
(
    Id INT PRIMARY KEY,
    CustomerId INT,
    CreatedAt Date,
    FOREIGN KEY (CustomerId) REFERENCES Customers (Id)
);

```

В данном случае определены таблицы Customers и Orders. Customers является главной и представляет клиента. Orders является зависимой и представляет заказ, сделанный клиентом. Таблица Orders через столбец CustomerId связана с таблицей Customers и ее столбцом Id. То есть столбец CustomerId является внешним ключом, который указывает на столбец Id из таблицы Customers.

С помощью оператора CONSTRAINT можно задать имя для ограничения внешнего ключа:

```

CREATE TABLE Orders
(
    Id INT PRIMARY KEY,
    CustomerId INT,
    CreatedAt Date,
    CONSTRAINT orders_customers_fk
    FOREIGN KEY (CustomerId) REFERENCES Customers (Id)
);

```

Проверим действие созданного ограничения. Добавим в таблицу Customers запись (см. Рисунок 18).

Data Output

Explain

Messages

Notifications

	id [PK] integer	age integer	firstname character varying (20)	lastname character varying (20)	phone character varying (20)
1	1	11	Andrey	Petrov	88005553535

Рисунок 18 – Добавление записи в таблицу Customers

В таблицу `Orders` добавим запись, которая будет ссылаться на несуществующее значение в `Customers`. Это вызовет ошибку (см. Рисунок 19).

Data Output				Explain	Messages	Notifications
	id [PK] integer	customerid integer	createdat date			
1		1	2	11.11.2011		

ERROR: insert or update on table "orders" violates foreign key constraint "orders_customerid_fkey" DETAIL: Key (customerid)=(2) is not present in table "customers".

Рисунок 19 – Ошибка при добавлении записи в `Orders`

Добавим запись в таблицу `Orders` с существующим внешним ключом и попробуем удалить запись из таблицы `Customers`. Это вызовет ошибку, так как на нее ссылается запись из `Orders` (см. Рисунок 20).

ERROR: update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders"
DETAIL: Key (id)=(1) is still referenced from table "orders".

Рисунок 20 – Ошибка при удалении записи из таблицы `Customers`

2.2.6.7. Действия при удалении и обновлении связанных строк

С помощью выражений `ON DELETE` и `ON UPDATE` можно установить действия, которые выполняются соответственно при удалении и изменении связанной строки из главной таблицы. В качестве действия могут использоваться следующие опции:

- **CASCADE**: автоматически удаляет или изменяет строки из зависимой таблицы при удалении или изменении связанных строк в главной таблице.
- **SET NULL**: при удалении или обновлении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение `NULL`. (В этом случае столбец внешнего ключа должен поддерживать установку `NULL`)
- **RESTRICT**: отклоняет удаление или изменение строк в главной таблице при наличии связанных строк в зависимой таблице.
- **NO ACTION**: то же самое, что и **RESTRICT** (Существенное различие между этими двумя вариантами заключается в том, что **NO ACTION** позволяет отложить проверку до более позднего этапа транзакции, а **RESTRICT** — нет.)
- **SET DEFAULT**: при удалении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение по умолчанию, которое задается с помощью соответствующего ограничения.

Каскадное (CASCADE) удаление позволяет при удалении строки из главной таблицы автоматически удалить все связанные строки из зависимой таблицы. Это записывается следующим образом:

```
CREATE TABLE Orders
(
    Id INT PRIMARY KEY,
    CustomerId INT,
    CreatedAt Date,
    FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE
CASCADE
);
```

Подобным образом работает и выражение ON UPDATE CASCADE. При изменении значения первичного ключа автоматически изменится значение связанного с ним внешнего ключа. Однако поскольку первичные ключи изменяются очень редко, да и в принципе не рекомендуется использовать в качестве первичных ключей столбцы с изменяемыми значениями, то на практике выражение ON UPDATE используется редко.

При установке для внешнего ключа опции SET NULL необходимо, чтобы столбец внешнего ключа допускал значение NULL:

```
CREATE TABLE Orders
(
    Id INT PRIMARY KEY,
    CustomerId INT,
    CreatedAt Date,
    FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE SET
NULL
);
```

2.2.7. Основные операторы SQL

2.2.7.1. Добавление данных

Для добавления данных применяется команда INSERT, которая имеет следующий формальный синтаксис:

```
INSERT [INTO] имя_таблицы [(список_столбцов)] VALUES (значение1,
значение2, ... значениеN);
```


Вначале идет выражение `INSERT INTO`, затем в скобках необходимо указать список столбцов через запятую, в которые надо добавлять данные, и в конце после слова `VALUES` в скобках перечисляют добавляемые для столбцов значения.

Например, пусть имеется следующая таблица:

```
CREATE TABLE Products
(
    Id INT IDENTITY PRIMARY KEY,
    ProductName VARCHAR(30) NOT NULL,
    Manufacturer VARCHAR(20) NOT NULL,
    ProductCount INTEGER DEFAULT 0,
    Price INTEGER NOT NULL
);
```

Добавим в нее одну строку с помощью команды `INSERT`:

```
INSERT INTO Products VALUES ('iPhone 7', 'Apple', 5, 52000);
```

Также при вводе значений можно указать конкретно столбцы, в которые будут добавляться значения:

```
INSERT INTO Products (ProductName, Price, Manufacturer)
VALUES ('iPhone 6S', 41000, 'Apple');
```

Для неуказанных столбцов (в данном случае `ProductCount`) будет добавляться значение по умолчанию, если задан атрибут `DEFAULT`, или значение `NULL`. При этом неуказанные столбцы должны допускать значение `NULL` или иметь атрибут `DEFAULT`.

Также мы можем добавить сразу несколько строк:

```
INSERT INTO Products
VALUES
('iPhone 6', 'Apple', 3, 36000),
('Galaxy S8', 'Samsung', 2, 46000),
('Galaxy S8 Plus', 'Samsung', 1, 56000);
```

В данном случае в таблицу будут добавлены три строки.

Предположим, нам необходимо вставить значения из одной таблицы в другую. Это можно сделать следующим образом:

```
INSERT INTO Product_2
SELECT ProductName, Price, Manufacturer
FROM Product
WHERE price > 36000;
```

2.2.7.2. Обновление данных

Для обновления данных применяется команда UPDATE, которая имеет следующий формальный синтаксис:

```
UPDATE <table_name>
SET <col_name1> = <value1>, <col_name2> = <value2>, ...
WHERE <condition>;
```

Пример обновления данных в таблице:

```
Update Product SET Manufacturer = 'Samsung' where ProductName =
'Galaxy';
```

2.2.7.3. Удаление данных

Удаление всех данных из таблицы:

```
DELETE FROM <table_name>;
```

Удаление данных с условием:

```
DELETE FROM Manufacturer WHERE id=1;
```

Удаление таблицы:

```
DROP TABLE <table_name>;
```

2.2.7.4. Выборка данных (оператор SELECT и его модификации)

Оператор **SELECT** используется для получения данных из определённой таблицы:

```
SELECT <col_name1>, <col_name2>, ...
FROM <table_name>;
```

Следующей командой можно вывести все данные из таблицы. Рисунок 21 демонстрирует результат выполнения запроса.

```
SELECT * FROM Product;
```

ProductName character varying	price bigint	Manufacturer character varying
Galaxy S10	50000	Samsung
Iphone 8	60000	Apple
Galaxy S7	30000	Samsung

Рисунок 21 – Результат выполнения запроса *SELECT*

В столбцах таблицы часто могут содержаться повторяющиеся данные. Используйте оператор **SELECT DISTINCT** для получения только неповторяющихся данных.

```
SELECT DISTINCT <col_name1>, <col_name2>, ...  
FROM <table_name>;
```

Например, следующей командой будут выведены все уникальные производители (см. Рисунок 22).

```
SELECT DISTINCT Manufacturer FROM Product;
```

Manufacturer character varying
Samsung
Apple

Рисунок 22 – Результат выполнения оператора *SELECT DISTINCT*

Можно также использовать ключевое слово **WHERE** в операторе **SELECT** для указания условий в запросе:

```
SELECT <col_name1>, <col_name2>, ...  
FROM <table_name>  
WHERE <condition>;
```

В запросе можно задавать следующие условия:

- сравнение текста;

- сравнение численных значений;
- логические операции AND (и), OR (или) и NOT (отрицание).

Примеры:

```
SELECT * FROM Product WHERE ProductName='Galaxy S7';
SELECT * FROM Product WHERE price<40000;
SELECT * FROM Product WHERE ProductName ='Galaxy S7' AND price>20000;
```

Оператор **GROUP BY** часто используется с агрегатными функциями, такими как COUNT, MAX, MIN, SUM и AVG, для группировки выходных значений.

```
SELECT <col_name1>, <col_name2>, ...
FROM <table_name>
GROUP BY <col_name>;
```

В качестве примера выведем количество телефонов каждого производителя. Рисунок 23 демонстрирует результат выполнения запроса.

```
SELECT COUNT(ProductName), Manufacturer
FROM Product
GROUP BY Manufacturer;
```

count bigint	Manufacturer character varying
2	Samsung
1	Apple

Рисунок 23 – Результат выполнения оператора SELECT с GROUP BY

Ключевое слово **HAVING** было добавлено в SQL потому, что WHERE не может быть использовано для работы с агрегатными функциями.

```
SELECT <col_name1>, <col_name2>, ...
FROM <table_name>
GROUP BY <column_name>
HAVING <condition>;
```

Например, выведем список производителей, у которых более одного товара. Рисунок 24 демонстрирует результат выполнения этого запроса.

```
SELECT COUNT(ProductName), Manufacturer
```

```
FROM Product
GROUP BY Manufacturer
HAVING COUNT(ProductName) >1;
```

count	Manufacturer
bigint	character varying
2	Samsung

Рисунок 24 – Результат выполнения оператора *SELECT* с *HAVING*

Важно понимать, как соотносятся агрегатные функции и SQL-предложения *WHERE* и *HAVING*. Основное отличие *WHERE* от *HAVING* заключается в том, что *WHERE* сначала выбирает строки, а затем группирует их и вычисляет агрегатные функции (таким образом, она отбирает строки для вычисления агрегатов), тогда как *HAVING* отбирает строки групп после группировки и вычисления агрегатных функций. Как следствие, предложение *WHERE* не должно содержать агрегатных функций; не имеет смысла использовать агрегатные функции для определения строк для вычисления агрегатных функций. Предложение *HAVING*, напротив, всегда содержит агрегатные функции. (Строго говоря, вы можете написать предложение *HAVING*, не используя агрегаты, но это редко бывает полезно. То же самое условие может работать более эффективно на стадии *WHERE*.)

Ключевое слово **ORDER BY** используется для сортировки результатов запроса по убыванию или возрастанию. *ORDER BY* отсортирует по возрастанию, если не будет указан способ сортировки (*ASC* – «по возрастанию» или *DESC* – «по убыванию»).

```
SELECT <col_name1>, <col_name2>, ...
FROM <table_name>
ORDER BY <col_name1>, <col_name2>, ... ASC|DESC;
```

В качестве примера выведем всю информацию о товарах, отсортировав их по возрастанию цены (см. Рисунок 25).

```
SELECT * FROM Product
ORDER BY price;
```

ProductName character varying 	price bigint 	Manufacturer character varying 
Galaxy S7	30000	Samsung
Galaxy S10	50000	Samsung
Iphone 8	60000	Apple

Рисунок 25 - Результат выполнения оператора *SELECT* с *ORDER BY*

2.3. Сложные типы данных PostgreSQL

PostgreSQL является объектно-реляционной СУБД и поддерживает следующие типы данных: массивы (различной размерности), структуры, перечислимые поля, а также встроенные типы данных (геометрические, json, xml, диапазоны и т.д.)

2.3.1. Массивы

Чтобы проиллюстрировать использование массивов [4], мы создадим такую таблицу:

```
CREATE TABLE sal_emp (
    name text,
    pay_by_quarter integer[],
    schedule text[][]
);
```

Где `pay_by_quarter` – одномерный массив целых чисел,
`schedule` – двумерный массив строк.

Как показано, для объявления типа массива к названию типа элементов добавляются квадратные скобки (`[]`).

Для добавления значений в массив используются фигурные скобки.

```
INSERT INTO sal_emp
VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');
```

Также можно использовать синтаксис конструктора `ARRAY`:

```
INSERT INTO sal_emp
VALUES ('Bill',
```

```
ARRAY[10000, 10000, 10000, 10000],  
ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);
```

С массивом можно производить следующие операции: обращение к элементу по индексу, определение количества элементов массива, преобразование массива в множество значений, добавление и удаление элемента из массива, проверка наличия элемента в массиве.

Добавив данные в таблицу, мы можем перейти к выборкам. Сначала продемонстрируем, как получить один элемент массива. Указанный ниже запрос получает имена сотрудников, зарплата которых изменилась во втором квартале:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <>  
pay_by_quarter[2];
```

Значение массива можно заменить полностью так:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'  
WHERE name = 'Carol';
```

или используя синтаксис ARRAY:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]  
WHERE name = 'Carol';
```

Чтобы найти значение в массиве, необходимо проверить все его элементы. Это можно сделать вручную, если вы знаете размер массива. Например:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR  
pay_by_quarter[2] = 10000 OR  
pay_by_quarter[3] = 10000 OR  
pay_by_quarter[4] = 10000;
```

Однако с большими массивами этот метод становится утомительным, и к тому же он не работает, когда размер массива неизвестен. Показанный выше запрос можно было переписать так:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

Функция **unnest** разворачивает массив в набор строк. К примеру:

```
unnest(ARRAY[1,2])
```

Функция выведет 2 строки:

```
1  
2
```

Функции **array_append** и **array_remove** служат для добавления и удаления элементов массива.

array_append добавляет элемент в конец массива

```
UPDATE sal_emp SET pay_by_quarter =array_append(pay_by_quarter,  
10000) WHERE id=3;
```

array_remove удаляет из массива все элементы, равные заданному значению (массив должен быть одномерным)

```
UPDATE sal_emp SET pay_by_quarter =array_remove(pay_by_quarter,  
10000) WHERE id=3;
```

Проверку наличия набора элементов можно выполнить следующим образом:

```
SELECT * FROM sal_emp WHERE {10000, 20000} <@ pay_by_quarter;
```

Длину массива позволяет узнать функция **array_length**

```
SELECT array_length(pay_by_quarter, 1) AS pay_by_quarter_elements  
FROM sal_emp;
```

2.3.2. Структурные типы

Структурный тип [5] представляет структуру табличной строки или записи. По сути, это просто список имён полей и соответствующих типов данных. Его можно использовать для хранения сложных структур данных, к примеру, ФИО. Ниже приведен пример определения структурного типа:

```
CREATE TYPE inventory_item AS (  
    name          text,  
    supplier_id   integer,  
    price         numeric  
);
```


Определив такие типы, мы можем использовать их в таблицах:

```
CREATE TABLE on_hand (  
    item      inventory_item,  
    count     integer  
);
```

Чтобы обратиться к полю столбца структурного типа, после имени столбца нужно добавить точку и имя поля, подобно тому, как указывается столбец после имени таблицы.

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

Указанные ниже команды иллюстрируют добавление или изменение всего столбца:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));  
UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

Так же можно изменять структурный тип по конкретным полям:

```
UPDATE on_hand SET item.price = 1000 WHERE ...;
```

Вывод структурного типа полностью

```
SELECT item FROM on_hand;
```

2.3.3. Перечисления

PostgreSQL имеет специальный тип данных, который называется **enum** и представляет собой набор констант [6]. Столбец подобного типа может в качестве значения принимать одну из этих констант. Тип перечислений создается с помощью команды **CREATE TYPE**, например так:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

В метках значений регистр имеет значение, т. е. 'happy' и 'HAPPY' — не одно и то же. Также в метках имеют значение пробелы.

Удалять существующие значения из перечисления, а также изменять их порядок, нельзя — для получения нужного результата придётся удалить и заново создать это перечисление.

Невозможно в колонке перечисляемого типа задать значение, которое отсутствует в самом перечислении. Предположим, созданное выше поле `mood` имеет тип `mood`. Тогда указанный ниже запрос выдаст ошибку, так как такого значения нет в перечислении.

```
UPDATE Student"
SET mood='very happy'
WHERE id=2;
```

2.3.4. Наследование

В PostgreSQL таблица может наследоваться от нуля или нескольких других таблиц, а запросы могут выводить все строки родительской таблицы или все строки родительской и всех дочерних таблиц [7].

Формально наследование можно представить так, как показано ниже. Таблица `child` наследует атрибуты от таблицы `base`.

```
CREATE TABLE base (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....
);
```

```
CREATE TABLE child (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....
)INHERITS base;
```

В качестве примера определим таблицу `capitals` как наследника таблицы `cities`:

```
CREATE TABLE cities (
```

```

        name          text,
        population    float,
        altitude      int
    );

    CREATE TABLE capitals (
        state          char(2)
    ) INHERITS (cities);

```

Дочерние таблицы автоматически наследуют от родительской таблицы ограничения-проверки и ограничения NOT NULL (если только для них не задано явно NO INHERIT). Все остальные ограничения (уникальности, первичный ключ и внешние ключи) не наследуются [7]. Для таблиц внутри иерархии наследования, могут быть заданы ограничения целостности check. Все ограничения check для родительской таблицы автоматически наследуются всеми её потомками.

Таблица может наследовать более чем от одной родительской таблицы, и в этом случае она будет содержать общий список колонок из всех родительских таблиц. К этому списку добавляются любые колонки, задаваемые в самой таблице-потомке. Если в нескольких родительских таблицах, встречаются колонки с одинаковым именем или такая колонка есть в родительской таблице и в определении таблицы-потомка, то эти колонки «сливаются» таким образом, что только одна такая колонка будет в таблице-потомке. При слиянии колонки должны иметь одинаковый тип, иначе будет выдано сообщение об ошибке. Колонка после слияния получает копию всех ограничений целостности check от каждой слитой колонки.

Возможности наследования серьёзно ограничены тем, что индексы (включая ограничения уникальности) и ограничения внешних ключей относятся только к отдельным таблицам, но не к их потомкам. Это касается обеих сторон ограничений внешних ключей. Рассмотрим этот вопрос на описанном нами примере.

Если мы объявим `cities.name` с ограничением UNIQUE или PRIMARY KEY, это не помешает добавить в таблицу `capitals` строки с названиями городов, уже существующими в таблице `cities`. И эти дублирующиеся строки по умолчанию будут выводиться в результате запросов к `cities`. На деле таблица `capitals` по умолчанию вообще не будет содержать ограничение уникальности, так что в ней могут оказаться несколько строк с одним названием. Хотя вы можете добавить в `capitals`

соответствующее ограничение, но это не предотвратит дублирование при объединении с `cities`.

Подобным образом, если мы укажем, что `cities.name` ссылается (REFERENCES) на какую-то другую таблицу, это ограничение не будет автоматически распространено на `capitals`. В этом случае решением может стать явное добавление такого же ограничения REFERENCES в таблицу `capitals`.

Если вы сделаете так, чтобы столбец другой таблицы ссылался на `cities(name)`, в этом столбце можно будет указывать только названия городов, но не столиц. В этом случае хорошего решения нет.

Родительскую таблицу нельзя удалить, пока существуют унаследованные от неё. При этом в дочерних таблицах нельзя удалять или модифицировать столбцы или ограничения-проверки, унаследованные от родительских таблиц. Если вы хотите удалить таблицу вместе со всеми её потомками, это легко сделать, добавив в команду удаления родительской таблицы параметр CASCADE.

В качестве примера создадим таблицу `new_course`, наследующуюся от таблицы `Course` и добавляющую поле `partner`.

```
CREATE TABLE public."Course"
(
    "Description" character varying(200) COLLATE
pg_catalog."default",
    "Title" character varying(100) COLLATE pg_catalog."default" NOT
NULL,
    id bigint NOT NULL,
    "Price" bigint NOT NULL DEFAULT 10000,
    "Tip" character varying(100)[] COLLATE pg_catalog."default",
    teacher_contact teacher_contact,
    course_info course_info,
    redirected integer,
    CONSTRAINT "Course_pkey" PRIMARY KEY (id),
    CONSTRAINT "Title" UNIQUE ("Title")
);

CREATE TABLE New_course (
    partner text
) INHERITS (public."Course");
```

Вставим в нее кортеж.

```
INSERT INTO public."new_course" ("Description", "Title", "id",
"Price", "Tip", "teacher_contact", "partner" )
VALUES ('Программирование', 'C++', '1','1000', ARRAY['Web',
'Program'], ROW('test@ya.ru','+79161231212'), 'yandex');
```

Выведем строки из производной таблицы (см. Рисунок 26):

```
SELECT * FROM public."new_course";
```

	Description character varying (200)	Title character varying (100)	id bigint	Price bigint	Tip character varying[] (100)	teacher_contact teacher_contact	partner text	course_info course_info
1	Программирование	Scala	2	10000	{Web,Program}	(test@ya.ru,+791612...	yandex	[null]

Рисунок 26 – Результат вывода из производной таблицы

Если мы выведем строки из базовой таблицы, к ним добавятся строки из дочерней (см. Рисунок 27).

```
SELECT * FROM public."Course";
```

Description character varying (200)	Title character varying (100)	id [PK] bigint	Price bigint	Tip character varying[] (100)	teacher_contact teacher_contact	course_info course_info
test	test title	1	10000	{subd,web,C#}	(del@ya.ru,+7916123...	(1,Title,)
Scala course	Scala	2	10000	[null]	[null]	[null]
C# course	C#	7	10000	{programming,ms}	[null]	[null]
C++ course	C++	3	10000	[null]	[null]	[null]
Программирование	C++	1	1000	{Web,Program}	(test@ya.ru,+791612...	[null]

Рисунок 27 – Результат вывода из базовой таблицы

Чтобы вывести только строки базовой таблицы, необходимо использовать **ONLY**.

Многие операторы (SELECT, UPDATE и DELETE) поддерживают ключевое слово ONLY. Например, выведем только строки из базовой таблицы Course (см. Рисунок 28).

```
SELECT * FROM ONLY public."Course";
```

Description character varying (200)	Title character varying (100)	id [PK] bigint	Price bigint	Tip character varying[] (100)	teacher_contact teacher_contact	course_info course_info
test	Test Title	1	10000	{subd,web,C#}	(del@ya.ru,+7916123...	(1,Title,)
Scala course	Scala	2	10000	[null]	[null]	[null]
C# course	C#	7	10000	{programming,ms}	[null]	[null]
C++ course	C++	3	10000	[null]	[null]	[null]

Рисунок 28 – Результат вывода из базовой таблицы с использованием ONLY

При вставке кортежа в базовую таблицу он не будет добавлен в дочернюю. Рисунок 29 демонстрирует содержимое базовой таблицы, а Рисунок 30 – содержимое дочерней.

```
INSERT INTO public."Course" ("Description", "Title", "id", "Price",
"Tip", "teacher_contact" )
VALUES ('Программирование', 'C++', '11', '1000', ARRAY['Web',
'Program'], ROW('test@ya.ru', '+79161231212'));
SELECT * FROM public."Course";
```

Description character varying (200)	Title character varying (100)	id [PK] bigint	Price bigint	Tip character varying[] (100)	teacher_contact teacher_contact	course_info course_info
Scala course	Scala	7	10000	{programming,ms}	[null]	[null]
C# course	C#	7	10000	{programming,ms}	[null]	[null]
C++ course	C++	3	10000	[null]	[null]	[null]
Программирование	C++	11	1000	{Web,Program}	(test@ya.ru,+791612...	[null]

Рисунок 29 – Содержимое базовой таблицы после вставки кортежа

```
SELECT * FROM public."new_course";
```

Description character varying (200)	Title character varying (100)	id bigint	Price bigint	Tip character varying[] (100)	teacher_contact teacher_contact	partner text	course_info course_info
1 Программирование	Scala	2	10000	{Web,Program}	(test@ya.ru,+791612...	yandex	[null]

Рисунок 30 – Содержимое дочерней таблицы после вставки кортежа в базовую

2.4. Пользовательские типы

PostgreSQL может расширяться и поддерживать новые типы данных.

Пользовательский тип [8] должен всегда иметь функции ввода и вывода. Эти функции определяют, как тип будет выглядеть в строковом виде (при вводе и выводе для пользователя) и как этот тип размещается в памяти. Функция ввода принимает в качестве аргумента строку символов, заканчивающуюся нулём, и возвращает внутреннее представление типа (в памяти). Функция вывода принимает в качестве аргумента внутреннее представление типа и возвращает строку символов, заканчивающуюся нулём. Если мы хотим не просто сохранить тип, но делать с ним нечто большее, мы должны предоставить дополнительные функции, реализующие все операции, которые мы хотели бы иметь для этого типа.

2.4.1. Описание пользовательского типа

Создание нового базового типа требует реализации функций для работы с типом на языке низкого уровня, обычно языке C. Предположим, что нам нужен тип `complex`, представляющий комплексные числа. Естественным образом комплексное число можно представить в памяти в виде следующей структуры на языке C:

```
typedef struct Complex {
```

```

double    x;
double    y;
} Complex;

```

В качестве внешнего строкового представления типа мы выберем строку вида (x,y).

Функции ввода и вывода обычно несложно написать, особенно функцию вывода. Но определяя внешнее строковое представление типа, помните, что в конце концов вам придётся реализовать законченный и надёжный чтения (парсинга) этого представления в своей функции ввода этого типа. Например, функция ввода на C:

```

PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char        *str = PG_GETARG_CSTRING(0);
    double      x, y;
    Complex     *result;
    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));
    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}

```

Функция вывода может быть простой:

```

PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex     *complex = (Complex *) PG_GETARG_POINTER(0);

```

```

char      *result;
result = psprintf("(%g,%g)", complex->x, complex->y);
PG_RETURN_CSTRING(result);
}

```

2.4.2. Определение созданного типа в PostgreSQL

Написав функции ввода/вывода и скомпилировав их в общую библиотеку, мы можем определить тип `complex` в SQL. Сначала мы объявим его как тип-пустышку:

```
CREATE TYPE complex;
```

Это позволит нам ссылаться на этот тип, определяя для него функции ввода/вывода. Теперь мы определим функции ввода/вывода:

```

CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;
CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;
CREATE FUNCTION complex_recv(internal)
    RETURNS complex
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;
CREATE FUNCTION complex_send(complex)
    RETURNS bytea
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;

```

Наконец, мы можем предоставить полное определение типа данных:

```

CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out,
    receive = complex_recv,

```



```
send = complex_send,  
alignment = double  
);
```

Определение типа данных [9] имеет множество опциональных атрибутов, указанных ниже. Обязательными являются функции функция_ввода и функция_вывода, тогда как функция_получения, функция_отправки, функция_модификатора_типа, функция_вывода_модификатора_типа и функция_анализа могут отсутствовать. Обычно эти функции разрабатываются на С или другом низкоуровневом языке.

```
CREATE TYPE имя (  
    INPUT = функция_ввода,  
    OUTPUT = функция_вывода  
    [ , RECEIVE = функция_получения ]  
    [ , SEND = функция_отправки ]  
    [ , TYPMOD_IN = функция_ввода_модификатора_типа ]  
    [ , TYPMOD_OUT = функция_вывода_модификатора_типа ]  
    [ , ANALYZE = функция_анализа ]  
    [ , INTERNALLENGTH = { внутр_длина | VARIABLE } ]  
    [ , PASSEDBYVALUE ]  
    [ , ALIGNMENT = выравнивание ]  
    [ , STORAGE = хранение ]  
    [ , LIKE = тип_образец ]  
    [ , CATEGORY = категория ]  
    [ , PREFERRED = предпочитаемый ]  
    [ , DEFAULT = по_умолчанию ]  
    [ , ELEMENT = элемент ]  
    [ , DELIMITER = разделитель ]  
    [ , COLLATABLE = сортируемый ]  
)
```

Функция **функция_ввода** преобразует внешнее текстовое представление типа во внутреннее, с которым работают операторы и функции, определённые для этого типа. **Функция_вывода** выполняет обратное преобразование. Функцию ввода можно объявить как принимающую один аргумент типа `cstring`. Функция ввода должна возвращать значение нового типа данных. Функция вывода должна принимать один аргумент

нового типа данных, а возвращать она должна `cstring`. Для значений `NULL` функции вывода не вызываются.

Необязательная **функция_получения** преобразует двоичное внешнее представление типа во внутреннее представление.

Функция **функция_отправки** преобразует данные из внутреннего во внешнее двоичное представление. Если эта функция не определена, новый тип не может участвовать в двоичном выводе. Функция отправки должна принимать один аргумент нового типа данных, а возвращать она должна `byte`. Для значений `NULL` функции отправки не вызываются.

Необязательные **функция_ввода_модификатора_типа** и **функция_вывода_модификатора_типа** требуются, только если типы поддерживают модификаторы, или, другими словами, дополнительные ограничения, связываемые с объявлением типа, например `char(5)`.

Функция_ввода_модификатора_типа получает объявленные модификаторы в виде строки `cstring`. Она должна проверить значения на допустимость (и вызвать ошибку, если они неверны), а затем выдать неотрицательное значение `integer`, которое будет сохранено в столбце `typmod`. Если для типа не определена **функция_ввода_модификатора_типа**, модификаторы типа приниматься не будут.

Функция_вывода_модификатора_типа преобразует внутреннее целочисленное значение `typmod` обратно, в форму, понятную пользователю. Она должна вернуть значение `cstring`, которое именно в этом виде будет добавлено к имени типа.

Необязательная **функция_анализа** выполняет сбор специфической для этого типа статистики в столбцах с таким типом данных.

INTERNALLENGTH – внутренняя длина. Если базовый тип данных имеет фиксированную длину, в **INTERNALLENGTH** указывается эта длина в виде положительного числа, а если длина переменная, в **INTERNALLENGTH** задаётся значение **VARIABLE**.

Необязательный флаг **PASSEDBYVALUE** указывает, что значения этого типа данных передаются по значению, а не по ссылке.

Параметр **выравнивание** определяет, как требуется выравнивать данные этого типа. Допускается выравнивание по границам 1, 2, 4 или 8 байт.

Параметр **хранение** позволяет выбрать стратегию хранения для типов данных переменной длины.

Параметр **тип_образец** позволяет задать основные свойства представления типа другим способом: скопировать их из существующего типа.

Параметры **категория** и **предпочитаемый** позволяют определять, какое неявное приведение будет применяться в неоднозначных ситуациях.

Параметр **разделитель** позволяет задать разделитель, который будет вставляться между значениями во внешнем представлении массива с элементами этого типа. По умолчанию разделителем является запятая (,).

Если необязательный логический параметр **сортируемый** равен true, определения столбцов и выражения с этим типом могут включать указания о порядке сортировки, в предложении COLLATE.

Более подробно описания параметров Вы можете прочитать в официальной документации [9].

Когда определяется новый базовый тип, PostgreSQL автоматически обеспечивает поддержку массивов с элементами такого типа. Тип массива обычно получает имя по имени базового типа с добавленным спереди символом подчёркивания (_).

2.4.3. Пример создания пользовательского типа

Пусть имеется следующая таблица, описывающая студента.

```
CREATE TABLE student
(
  first_name varchar(200),
  id integer NOT NULL,
  PRIMARY KEY (id),
);
```

2.4.3.1. Описание типа

Добавим в таблицу student пользовательский тип оценки (marks):

```
'(a,b,c)::marks
```

Где a – количество пятёрок за текущую сессию, b – четверок, c – троек. На ввод и на вывод поступают сразу все значения, сортировка по среднему арифметическому.

Функция ввода (файл marks_in.c) будет выглядеть следующим образом:

```
#include <postgres.h>
#include <fmgr.h>
```

```

#include <libpq/pqformat.h>
PG_MODULE_MAGIC;
typedef struct
{
    int a, b, c;
} marks;
PG_FUNCTION_INFO_V1(marks_in);
PG_FUNCTION_INFO_V1(marks_out);
Datum (marks_in)(PG_FUNCTION_ARGS)
{
    char *s = PG_GETARG_CSTRING(0);
    marks *v = (marks*)palloc(sizeof(marks));
    if (sscanf(s, "(%d,%d,%d)", &(v->a), &(v->b), &(v->c)) != 3)
    {
        ereport(ERROR, (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
errmsg("Invalid input syntax for marks: \"%s\"", s)));
    }
    PG_RETURN_POINTER(v);
}

```

Функция ввода считывает значения и записывает их во внутреннюю переменную типа marks.

Функция вывода (файл marks_out.c):

```

#include <postgres.h>
#include <fmgr.h>
#include <libpq/pqformat.h>
PG_MODULE_MAGIC;
typedef struct
{
    int a, b, c;
} marks;
PG_FUNCTION_INFO_V1(marks_out);
Datum (marks_out)(PG_FUNCTION_ARGS)
{
    marks *v = (marks*)PG_GETARG_POINTER(0);
    char *s = (char*)palloc(100);

```

```
snprintf(s, 100, "(%d,%d,%d)", v->a, v->b, v->c);
PG_RETURN_CSTRING(s);
}
```

Функция вывода считывает значения из переменной *v* типа *marks* и выводит их.

2.4.3.2. Компиляция кода

Для **компиляции кода в общую библиотеку** необходимо выполнить описанные ниже действия.

В Linux для создания кода PIC компилятору передаётся флаг `-fPIC`. Для создания общей библиотеки компилятору передаётся флаг `-shared`. Полный пример будет выглядеть так:

```
cc -fPIC -c marks_out.c
cc -shared -o marks_out.so marks_out.o
```

Полученную общую библиотеку можно будет затем загрузить в PostgreSQL. Когда команде `CREATE FUNCTION` передаётся имя файла, это должно быть имя файла общей библиотеки, а не промежуточного объектного файла. Заметьте, что принятое в системе расширение файлов библиотек (как правило, `.so` или `.sl`) в команде `CREATE FUNCTION` можно опустить, и обычно именно так и следует делать для лучшей портируемости.

Для нахождения общего объектного файла по имени, заданному в команде `CREATE FUNCTION`, применяется следующий алгоритм:

- Если имя задаётся абсолютным путём, загружается заданный файл.
- Если имя начинается со строки `$libdir`, эта часть пути заменяется путём к каталогу библиотек PostgreSQL, который определяется во время сборки.
- Если в имени не указывается каталог, поиск файла производится по пути, заданному конфигурационной переменной [dynamic_library_path](#).
- В противном случае (файл не был найден в пути поиска, или в его имени указывается не абсолютный путь к каталогу), загрузчик попытается принять имя как есть, что, скорее всего, не увенчается успехом. (Полагаться на текущий рабочий каталог ненадёжно.)
- Если эта последовательность не даёт положительный результат, к данному имени добавляется принятое на данной платформе расширение файлов

библиотек (часто .so) и последовательность повторяется снова. Если и это не приводит к успеху, происходит сбой загрузки.

Для поиска общих библиотек рекомендуется задавать либо путь относительно \$libdir, либо путь динамических библиотек. Это упрощает обновление версии при перемещении новой инсталляции в другое место. Какой именно каталог подразумевается под \$libdir, можно узнать с помощью команды pg_config --pkglibdir.

Обычно, по умолчанию pkglibdir = /usr/local/pgsql/lib.

2.4.3.3. Определение описанного пользовательского типа в PostgreSQL

Создание типа в PostgreSQL приведено ниже.

```
CREATE TYPE marks;  
CREATE OR REPLACE FUNCTION marks_in ( s cstring )  
RETURNS marks AS  
'marks_in', 'marks_in'  
LANGUAGE C IMMUTABLE STRICT;  
CREATE OR REPLACE FUNCTION marks_out ( v marks )  
RETURNS cstring AS  
'marks_out', 'marks_out'  
LANGUAGE C IMMUTABLE STRICT;  
CREATE TYPE marks  
(  
internallength = 24,  
input = marks_in,  
output = marks_out,  
);
```

Добавление строк в таблицу выполняется следующим образом. Рисунок 31 демонстрирует результат выполнения оператора.

```
INSERT INTO student VALUES (1, 'Петр', '(1,2,3)')
```

	id [PK] integer	first_name character varying (50)	marks marks
1	2	Василий	(6,0,0)
2	1	Петр	(1,2,3)
3	3	Сергей	(0,0,3)
4	4	Степан	(0,4,0)
5	5	Иннокентий	(2,0,0)

Рисунок 31 – Результат добавления строки с пользовательским типом

2.4.3.4. Описание функций сравнения

Для нашего типа необходимо также определить **функции сравнения**, чтобы иметь возможность пользоваться операторами сравнения, такими как =, <, > и т.д.

Опишем функцию сравнения двух объектов между собой.

```
#include <postgres.h>
#include <fmgr.h>
#include <libpq/pqformat.h>
PG_MODULE_MAGIC;
typedef struct
{
    int a, b, c;
} marks;
PG_FUNCTION_INFO_V1(marks_equal);
PG_FUNCTION_INFO_V1(marks_greater);
PG_FUNCTION_INFO_V1(marks_lesser);
PG_FUNCTION_INFO_V1(marks_greater_eq);
PG_FUNCTION_INFO_V1(marks_lesser_eq);
PG_FUNCTION_INFO_V1(marks_cmp);
```

Определим функцию равенства, которая подсчитывает среднее значение для каждого объекта типа marks и возвращает true, если они равны и false, если нет.

```
Datum marks_equal(PG_FUNCTION_ARGS)
{
    marks *v0 = (marks*)PG_GETARG_POINTER(0);
    marks *v1 = (marks*)PG_GETARG_POINTER(1);
```

```

    bool equal = true;
    double v0_mean = (5*v0->a + 4*v0->b + 3*v0->c)/(v0->a + v0->b + v0->c);
    double v1_mean = (5*v1->a + 4*v1->b + 3*v1->c)/(v1->a + v1->b + v1->c);
    equal &= v0_mean == v1_mean;
    PG_RETURN_BOOL(equal);
}

```

Определим оператор > и остальные аналогичные функции равенства.

```

Datum marks_greater(PG_FUNCTION_ARGS)
{
    marks *v0 = (marks*)PG_GETARG_POINTER(0);
    marks *v1 = (marks*)PG_GETARG_POINTER(1);
    bool equal = true;
    double v0_mean = (5*v0->a + 4*v0->b + 3*v0->c)/(v0->a + v0->b + v0->c);
    double v1_mean = (5*v1->a + 4*v1->b + 3*v1->c)/(v1->a + v1->b + v1->c);
    equal &= v0_mean > v1_mean;
    PG_RETURN_BOOL(equal);
}

Datum marks_lesser(PG_FUNCTION_ARGS)
{
    marks *v0 = (marks*)PG_GETARG_POINTER(0);
    marks *v1 = (marks*)PG_GETARG_POINTER(1);
    bool equal = true;
    double v0_mean = (5*v0->a + 4*v0->b + 3*v0->c)/(v0->a + v0->b + v0->c);
    double v1_mean = (5*v1->a + 4*v1->b + 3*v1->c)/(v1->a + v1->b + v1->c);
    equal &= v0_mean < v1_mean;
    PG_RETURN_BOOL(equal);
}

```



```

Datum marks_greater_eq(PG_FUNCTION_ARGS)
{
    marks *v0 = (marks*)PG_GETARG_POINTER(0);
    marks *v1 = (marks*)PG_GETARG_POINTER(1);
    bool equal = true;
    double v0_mean = (5*v0->a + 4*v0->b + 3*v0->c)/(v0->a + v0->b + v0->c);
    double v1_mean = (5*v1->a + 4*v1->b + 3*v1->c)/(v1->a + v1->b + v1->c);
    equal &= v0_mean >= v1_mean;
    PG_RETURN_BOOL(equal);
}

Datum marks_lesser_eq(PG_FUNCTION_ARGS)
{
    marks *v0 = (marks*)PG_GETARG_POINTER(0);
    marks *v1 = (marks*)PG_GETARG_POINTER(1);
    bool equal = true;
    double v0_mean = (5*v0->a + 4*v0->b + 3*v0->c)/(v0->a + v0->b + v0->c);
    double v1_mean = (5*v1->a + 4*v1->b + 3*v1->c)/(v1->a + v1->b + v1->c);
    equal &= v0_mean <= v1_mean;
    PG_RETURN_BOOL(equal);
}

static int
marks_cmp_internal(marks *v0, marks *v1)
{
    double v0_mean = (5*v0->a + 4*v0->b + 3*v0->c)/(v0->a + v0->b + v0->c);
    double v1_mean = (5*v1->a + 4*v1->b + 3*v1->c)/(v1->a + v1->b + v1->c);
    if (v0_mean < v1_mean)
        return -1;
    if (v0_mean > v1_mean)
        return 1;
    return 0;
}

```

```
}
```

Определим функцию сравнения, которая возвращает 1 если первый объект больше второго, 0 если они равны и -1 если первый объект меньше второго. Данная функция используется при сортировке объектов.

```
Datum marks_cmp(PG_FUNCTION_ARGS)
{
    marks *v0 = (marks*) PG_GETARG_POINTER(0);
    marks *v1 = (marks*) PG_GETARG_POINTER(1);
    PG_RETURN_INT32(marks_cmp_internal(v0, v1));
}
```

2.4.3.5. Задание функции сортировки для типа умолчанию

После написания функций сравнения их необходимо скомпилировать в общую библиотеку и подключить к Postgres (как это было описано выше). Затем мы подключаем функции из библиотеки к нашему типу.

```
CREATE OR REPLACE FUNCTION marks_equal ( v0 marks, v1 marks )
RETURNS boolean AS
'marks_operations', 'marks_equal'
LANGUAGE C IMMUTABLE STRICT;
CREATE OR REPLACE FUNCTION marks_greater ( v0 marks, v1 marks )
RETURNS boolean AS
'marks_operations', 'marks_greater'
LANGUAGE C IMMUTABLE STRICT;
CREATE OR REPLACE FUNCTION marks_lesser ( v0 marks, v1 marks )
RETURNS boolean AS
'marks_operations', 'marks_lesser'
LANGUAGE C IMMUTABLE STRICT;
CREATE OR REPLACE FUNCTION marks_greater_eq ( v0 marks, v1 marks )
RETURNS boolean AS
'marks_operations', 'marks_greater_eq'
LANGUAGE C IMMUTABLE STRICT;
CREATE OR REPLACE FUNCTION marks_lesser_eq ( v0 marks, v1 marks )
RETURNS boolean AS
'marks_operations', 'marks_lesser_eq'
```

```

LANGUAGE C IMMUTABLE STRICT;
CREATE FUNCTION marks_cmp(marks, marks)
RETURNS integer
AS 'marks_operations'
LANGUAGE C IMMUTABLE STRICT;
CREATE OPERATOR CLASS marks_ops
DEFAULT FOR TYPE marks USING btree AS
OPERATOR 1 < ,
OPERATOR 2 <= ,
OPERATOR 3 = ,
OPERATOR 4 >= ,
OPERATOR 5 > ,
FUNCTION 1 marks_cmp(marks, marks);

```

Продemonстрируем работу функции сортировки (см. Рисунок 32).

```

SELECT * FROM student
ORDER BY marks DESC;

```

	Id integer	first_name character varying (50)	marks marks
1	2	Василий	(6,0,0)
2	5	Иннокентий	(2,0,0)
3	4	Степан	(0,4,0)
4	1	Петр	(1,2,3)
5	3	Сергей	(0,0,3)

Рисунок 32 – Результат выполнения функции сортировки пользовательского типа

3. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как создать таблицы, их связи и ограничения используя мастера и DDL-запросы?
2. Что такое ограничения целостности, каких типов они бывают и как их задавать?
3. Как определять и использовать в запросах массивы? Перечислите основные функции для работы с массивами.
4. Как создавать и использовать составные типы? Как обратиться к полю составного типа и его элементам из запроса?
5. Как создавать и использовать перечислимые типы? Какие ограничения для них использования?
6. Что такое наследование таблиц? Как создать и обратиться из запроса к базовым и производным таблицам?
7. Что такое пользовательский тип данных? Как определить его поля? Какие функции нужны при создании пользовательского типа и как их определить?

4. СПИСОК ИСТОЧНИКОВ

1. Виноградов В.И., Виноградова М.В. Постреляционные модели данных и языки запросов: Учебное пособие. — М.: Изд-во МГТУ им. Н.Э. Баумана, 2017. — 100с. - ISBN 978-5-7038-4283-6.
2. PostgreSQL 14.2 Documentation. — Текст. Изображение: электронные // PostgreSQL : [сайт]. — URL: <https://www.postgresql.org/docs/14/index.html> (дата обращения: 25.04.2022)
3. pgAdmin 4 6.5 documentation. — Текст. Изображение: электронные // pgAdmin - PostgreSQL Tools : [сайт]. — URL: <https://www.pgadmin.org/docs/pgadmin4/6.5/index.html> (дата обращения: 25.04.2022)
4. PostgreSQL : Документация: 14: 8.15. Массивы. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/arrays> (дата обращения: 25.04.2022)
5. PostgreSQL : Документация: 14: 8.16. Составные типы. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/rowtypes> (дата обращения: 25.04.2022)
6. PostgreSQL : Документация: 14: 8.7. Типы перечислений. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/datatype-enum> (дата обращения: 25.04.2022)
7. PostgreSQL: Документация: 14: 5.10. Наследование. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/ddl-inherit> (дата обращения: 25.04.2022)
8. PostgreSQL: Документация: 14: 38.13. Пользовательские типы. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL: <https://postgrespro.ru/docs/postgresql/14/xtypes> (дата обращения: 25.04.2022)
9. PostgreSQL: Документация: 14: CREATE TYPE. — Текст. Изображение : электронные // Компания Postgres Professional : [сайт]. — URL:

<https://postgrespro.ru/docs/postgresql/14/sql-createtype>

(дата обращения:

25.04.2022)