ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Лабораторные работы по курсу «Базы данных» выполняются в среде PostgreSQL. Задачей курса "Базы данных" является освоение технологий работы с базами данных.

В пособии рассмотрены лабораторные работы в среде PostgreSQL. В работах студенты осваивают различные технологии работы с объектно-реляционными базами данных на примере СУБД PostgreSQL**.** Работы выполняются в интерактивном режиме и знакомят студентов с основными операциями по работе с базами данных (создание базы данных и таблиц, занесение данных, выполнение простейших операций над данными, формирование запросов на языке SQL).

ВВЕДЕНИЕ

PostgreSQL является объектно-реляционной СУБД. Это даёт ей некоторые преимущества над другими SQL базами данных с открытым исходным кодом.

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, желающее выполнять операции в базе данных. Клиентские приложения могут быть разнообразными: это может быть графическое приложение, текстовая утилита, веб-сервер, использующий базу данных для отображения веб-страниц, или специализированный инструмент для обслуживания БД. Некоторые клиентские приложения поставляются в составе дистрибутива PostgreSQL, но большинство создают сторонние разработчики.

Как и в других клиент-серверных приложениях, клиент и сервер могут располагаться на разных компьютерах. В этом случае они взаимодействуют по сети TCP/IP. Файлы, доступные на клиентском компьютере, могут быть недоступны (или доступны только под другим именем) на компьютере-сервере.

Сервер PostgreSQL может обслуживать одновременно несколько подключений клиентов. Для этого он запускает («порождает») отдельный процесс для каждого подключения. Можно сказать, что клиент и серверный процесс общаются, не затрагивая главный процесс postgres. Таким образом, главный серверный процесс всегда работает и ожидает подключения клиентов, принимая которые, он организует взаимодействие клиента и отдельного серверного процесса. (Конечно всё это происходит незаметно для пользователя, а эта схема рассматривается здесь только для понимания.) [1].

1. УСТАНОВКА PostgreSQL

Установка под Windows.

Загрузить нужную версию можно с этой страницы https://www.enterprisedb.com/downloads/postgres-postgresql-downloads.

Дальше запускаете установщик и появляется первая форма (рис.1.1) для начала запуска процесса установки.

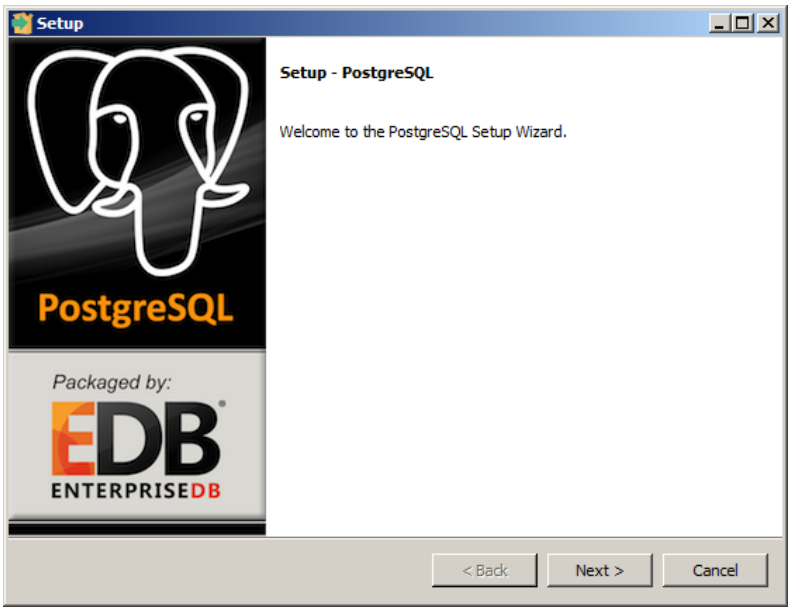

Рис.1.1 Первая форма установщика PostgreSQL

На втором экране (форма на рис. 1.1) вам надо выбрать директорию для установки. Не рекомендуется устанавливать в каталог “Program Files” по умолчанию, т.к. на Windows серверных платформ это бывает чревато, можно поставить директорию “C:\PostgreSQL\”. На домашних системах скорее всего проблем не будет. Далее выбираем все по умолчанию (рис. 1.2).

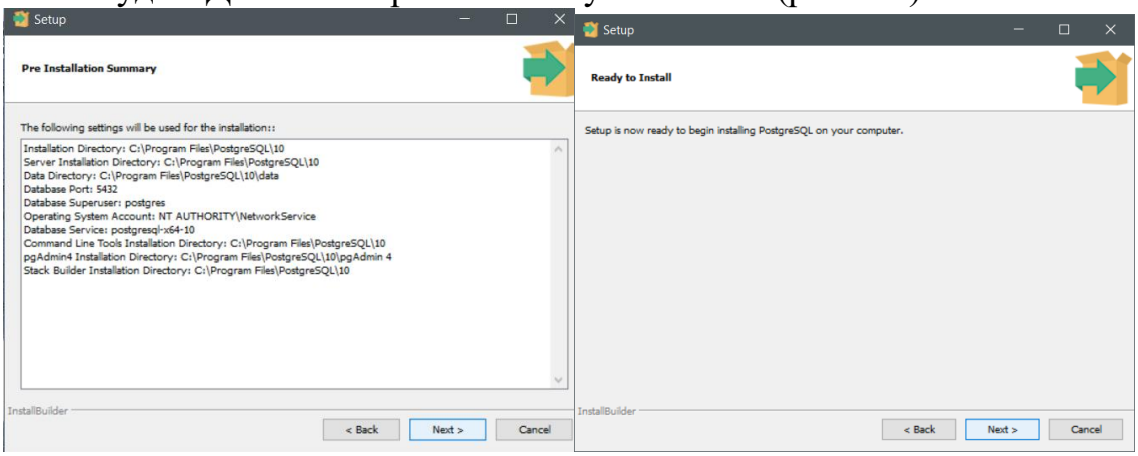

Рис.1.2 Последующие формы при запуске установщика PostgreSQL

По окончании установки можно не ставить дополнительную утилиту Stack Builder — “галочку” снять и нажать “Finish”.

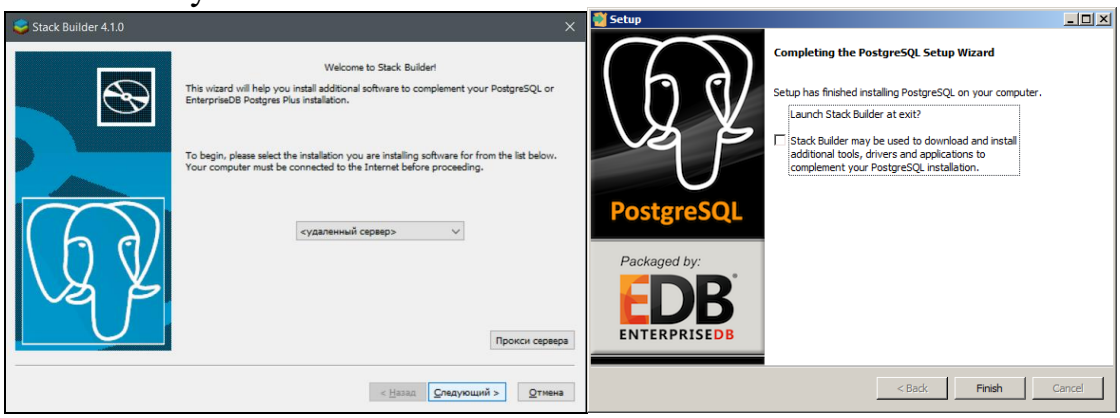

Рис.1.3 Завершающая форма установки PostgreSQL

Установка закончилась (рис. 1.3). Теперь в списке сервисов Windows можно увидеть PostgreSQL (рис. 1.4).

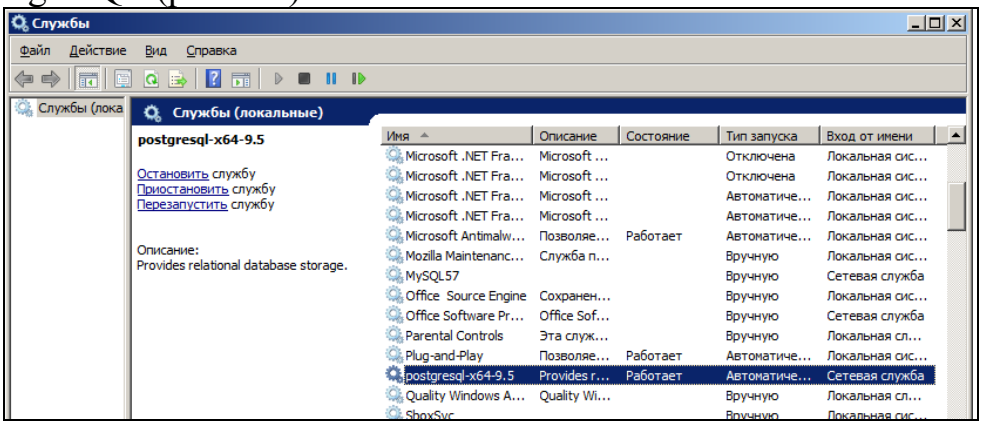

Рис.1.4 PostgreSQL в списке запущенных сервисов в оболочке Windows

В комплекте PostgreSQL устанавливается удобная программа для управления СУБД — **pgAdmin**. Ее можно найти в стартовом меню Windows. После запуска с левой стороны экрана доступно дерево: кликнем по серверу, по базам данных, таблицам, и т.д. (рис. 1.5).

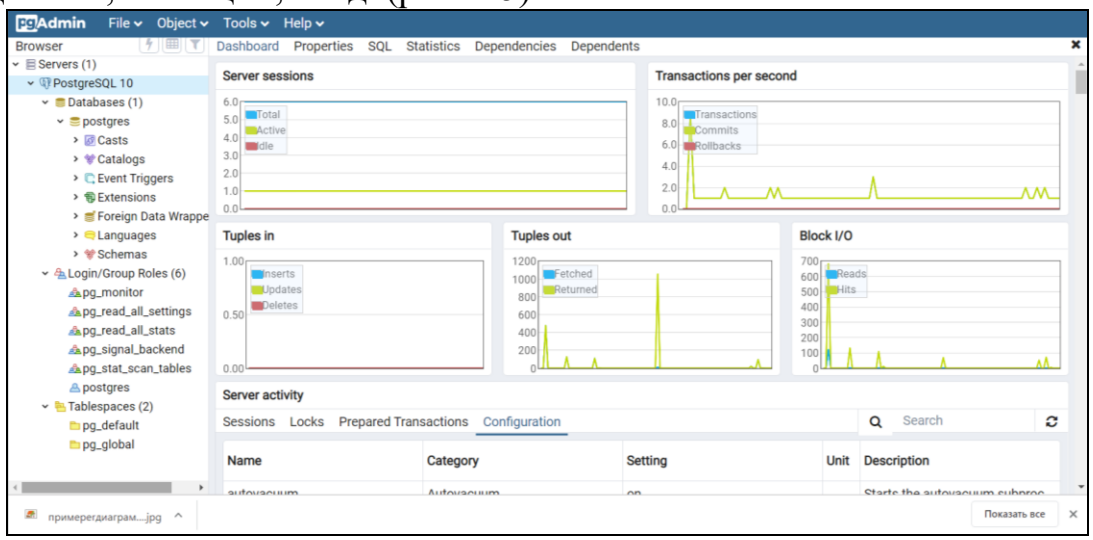

Рис.1.5 Интерфейс программы pgAdmin

2. НАЧАЛО РАБОТЫ

При развертывании сервера приложение pgAdmin развертывается за веб сервером или с интерфейсом WSGI. Если вы устанавливаете pgAdmin в режиме сервера, вам будет предложено указать имя роли и пароль pgAdmin при первом подключении к pgAdmin. Первая роль, зарегистрированная в pgAdmin, будет администратором. Эта роль может использовать диалоговое окно pgAdmin ***User Management*** для создания и управления дополнительными учетными записями пользователей pgAdmin. Когда пользователь аутентифицируется с помощью pgAdmin, древовидный элемент управления pgAdmin отображает определения сервера, связанные с этой ролью входа. Свойства сервера можно менять в окне свойств (рис. 2.1).

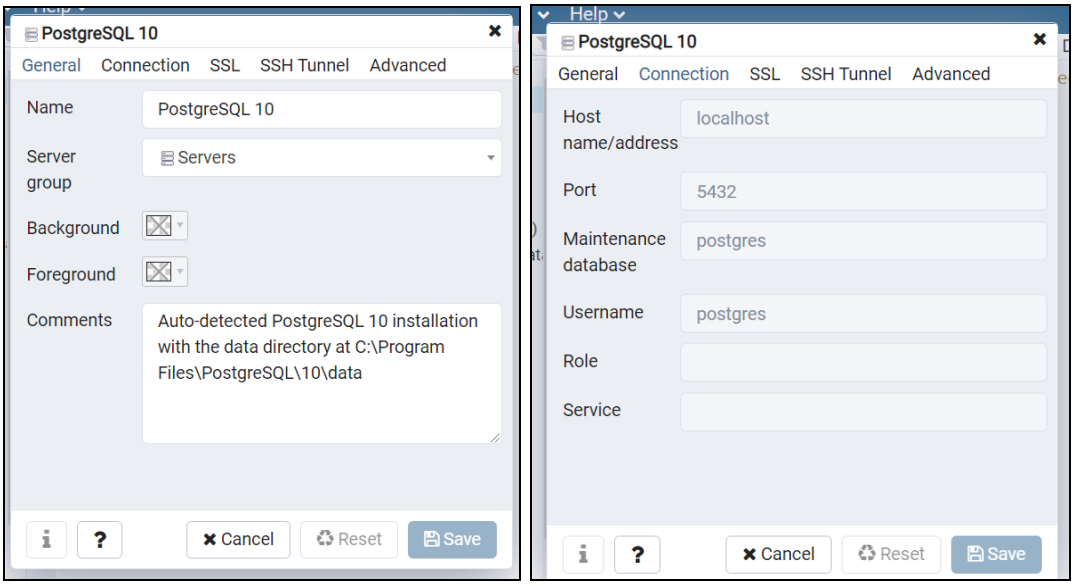

Рис.2.1 Окно свойства установленного сервера postgreSQL

Дерево управления обеспечивает обзор управляемых серверов и объектов, которые находятся на каждом сервере. Щелчок правой кнопкой мыши на узле в древовидном элементе управления дает доступ к контекстно-зависимым меню, которые обеспечивают быстрый доступ к задачам управления для выбранного объекта.

Браузер с вкладками обеспечивает быстрый доступ к статистической информации о каждом объекте в древовидном элементе управления, а также к инструментам и утилитам pgAdmin (таким как инструмент запросов и отладчик). pgAdmin открывает дополнительные вкладки функций каждый раз, когда вы получаете доступ к расширенным функциям, предлагаемым инструментами pgAdmin. Вы можете открывать, закрывать и переупорядочивать вкладки функций по мере необходимости.

Для настройки содержимого и поведения экрана pgAdmin используйте диалоговое окно «***Параметры***». Чтобы открыть диалог настроек, выберите «***Настройки***» в меню «***Файл***».

В нижнем левом углу каждого диалога есть кнопки ***Справки.*** Можно получить доступ к дополнительной справке Postgres, перейдя в меню «***Справка***» и выбрав имя ресурса, который нужно открыть.

3. СОЗДАНИЕ И УДАЛЕНИЕ БД

PostgreSQL позволяет создавать сколько угодно баз данных. Имена баз данных должны начинаться с буквы и быть не длиннее 63 символов. Завершается ввод команды точкой с запятой. Команда для создания БД:

CREATE DATABASE "ИмяБД";

Если имя базы данных не указать, она будет выбрана по имени пользователя. Можно пользоваться графическим интерфейсом и выполнять действия через контекстное меню.

Пример 1. Создадим базу данных с именем ***BD1*** (рис. 3.1).

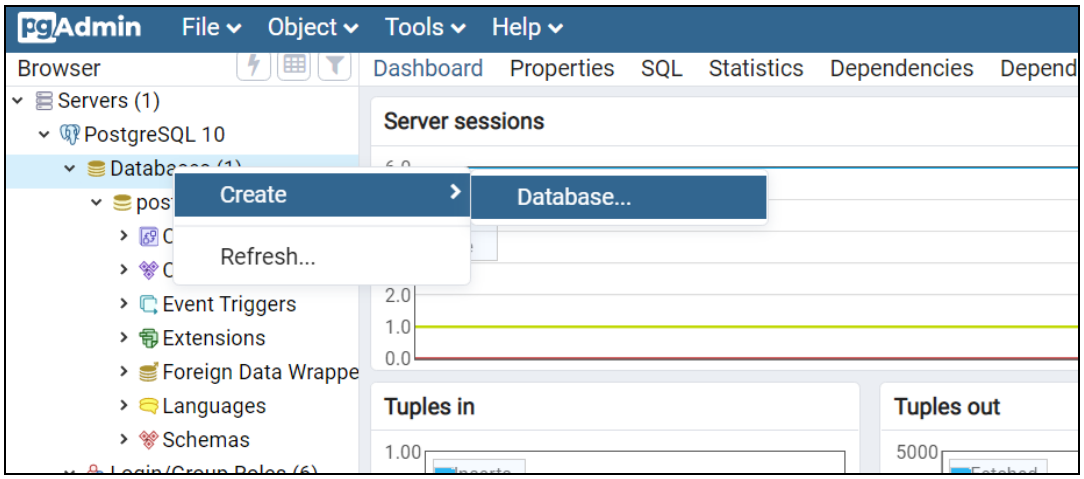

Рис.3.1 - Создание базы данных с помощью интерфейса pgAdmin

Зададим свойства базы данных (рис. 3.2).

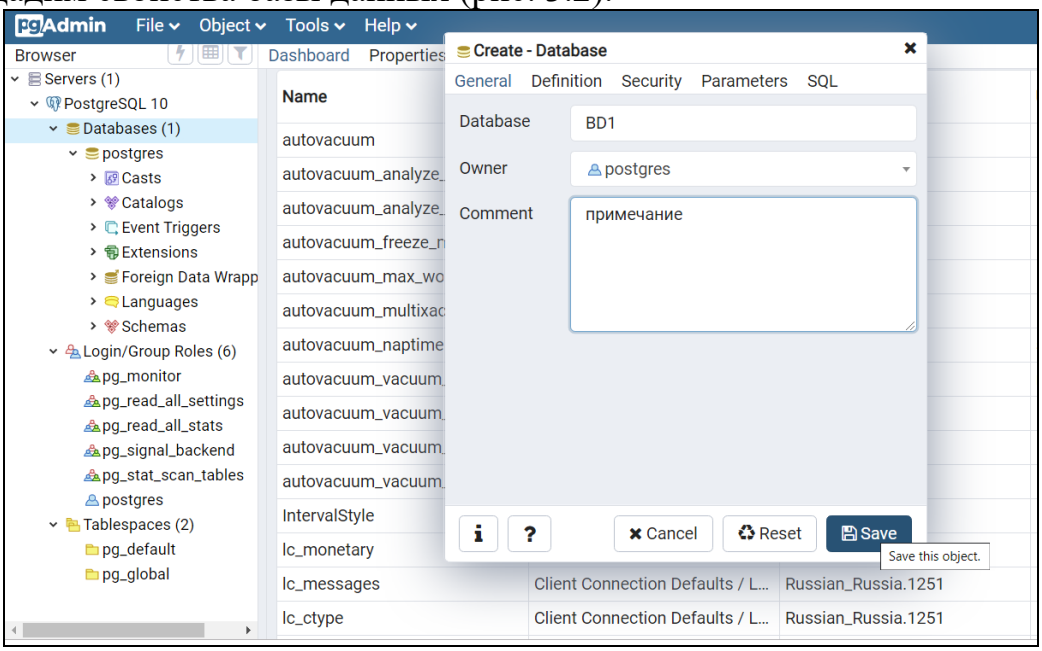

Рис.3.2 - Редактирование свойств для новой базы данных

На вкладке SQL можно посмотреть код операции (рис.3.3).

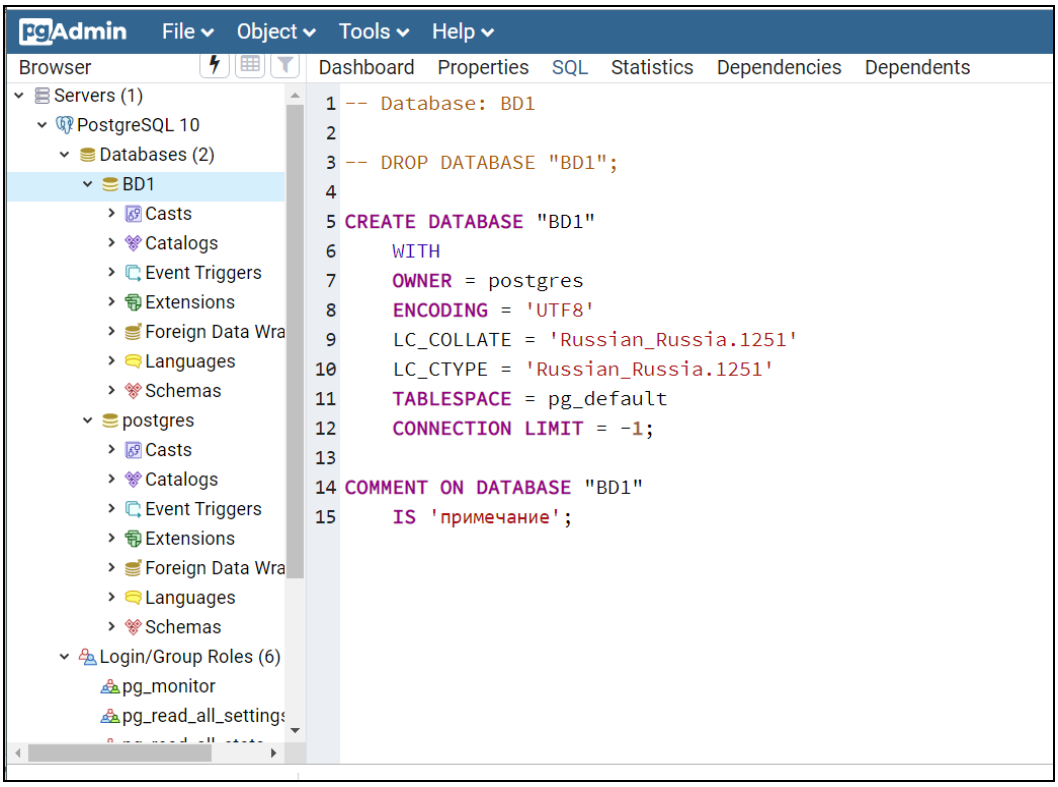

Рис.3.3 Вкладка SQL с кодом создания новой базы данных

4. ПОДКЛЮЧЕНИЕ К БАЗЕ ДАННЫХ

После создания базы данных, к ней можно обратиться:

1.Запустив терминальную программу PostgreSQL под названием ***psql***, в которой можно интерактивно вводить, редактировать и выполнять команды SQL. Появится приглашение, которое показывает, что psql ждёт ваших команд и вы можете вводить SQL-запросы в рабочей среде psql (рис. 4.1).

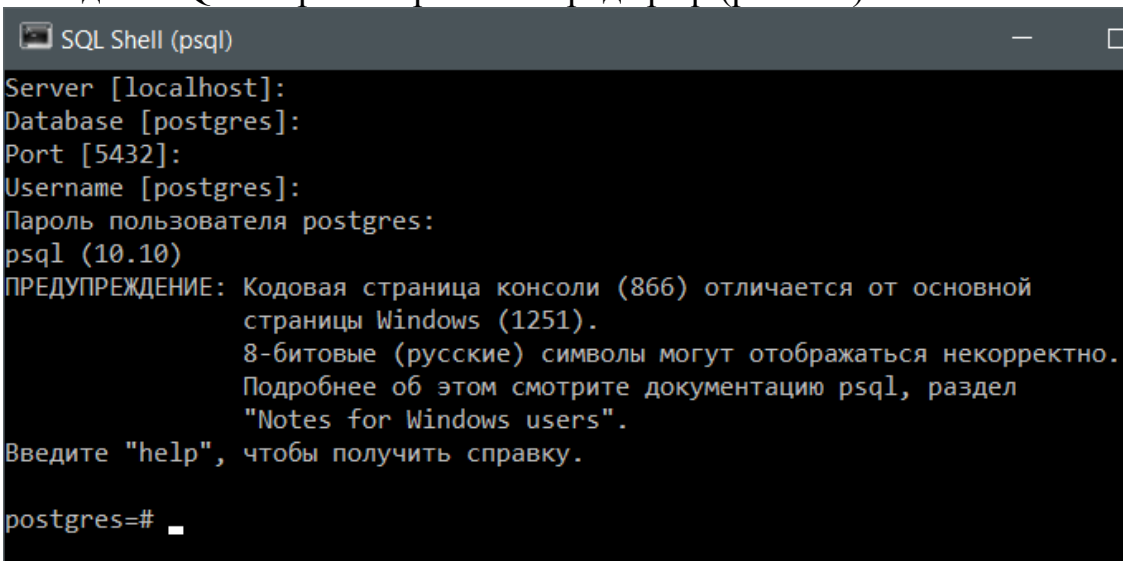

Рис.4.1 Консольная программа psql.exe

Введите ***Help,*** чтобы получить подсказку (рис. 4.2).

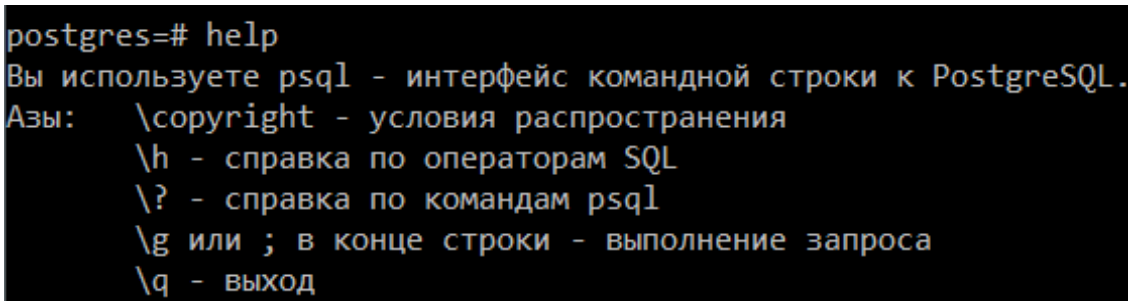

Рис.4.2 Запуск команды help в коммандной строке psql

В программе psql есть внутренние команды, которые не являются SQL-операторами. Они начинаются с обратной косой черты, «\». Для получения справки по различным SQL-командам PostgreSQL, введите: ***\h*** (рис. 4.3).

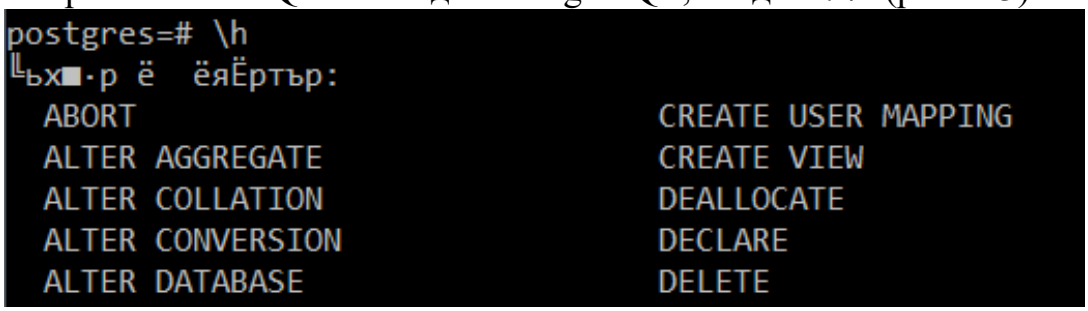

Рис.4.3 Листинг команд postgreSQL

Чтобы узнать версию PosrgreSQL введите команду :
SELECT version(); (рис. 4.4).

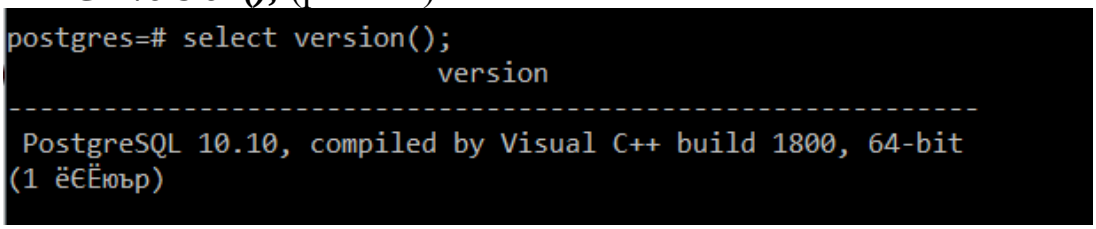

Рис.4.4 Результат выполнения запроса для получения версии СУБД

Для просмотра списка баз данных введите команду \L (рис.4.5).

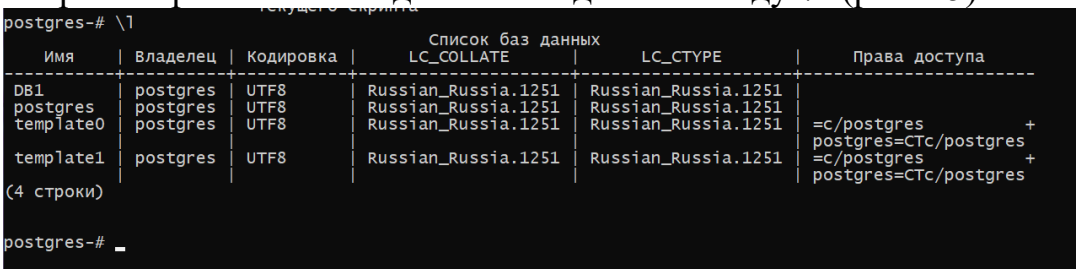

Рис.4.5 Результат выполнения команды \L

Для выхода из psql введите: ***\q.***

Psql завершит свою работу, а вы вернётесь в командную оболочку операционной системы.

Чтобы узнать о внутренних командах, введите **\?** в приглашении командной строки psql.

2. Используя графические инструменты, например, pgAdmin или офисный пакет с поддержкой ODBC или JDBC, позволяющий создавать и управлять базой данных.

3. Написав собственное приложение, используя один из множества доступных языковых интерфейсов.

5. СОЗДАНИЕ И УДАЛЕНИЕ ТАБЛИЦ

Для создания таблиц используется команда:

CREATE TABLE ИмяТабл (ИмяСтолбца ТипСтолбца, …).

PostgreSQL поддерживает стандартные типы данных 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.

В командах SQL можно свободно использовать пробельные символы (пробелы, табуляции и переводы строк). 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.

Пример 1: Создадим таблицу **Clients** (Клиенты), в которой будут храниться фамилия, имя, дата рождения и номер клиента (первичный ключ) (рис. 5.1).

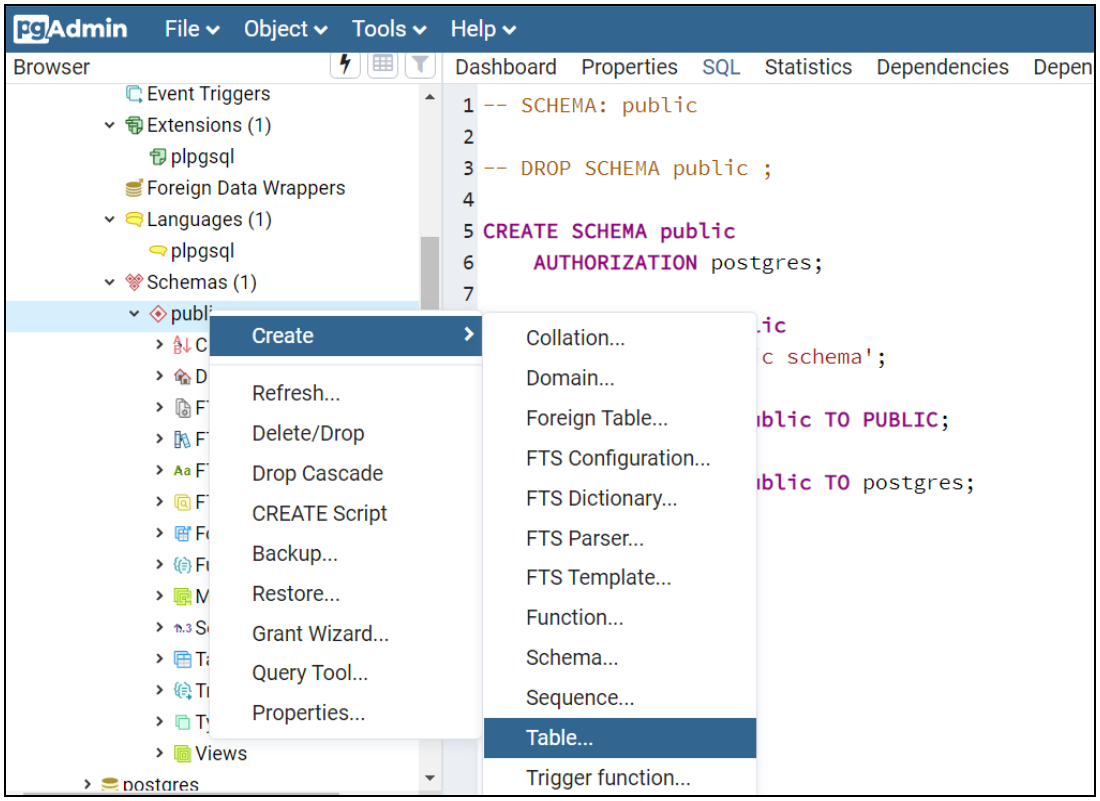

Рис.5.1 Создание новой таблицы с помощью интерфейса pgAdmin

Зададим свойства таблицы (рис. 5.2).

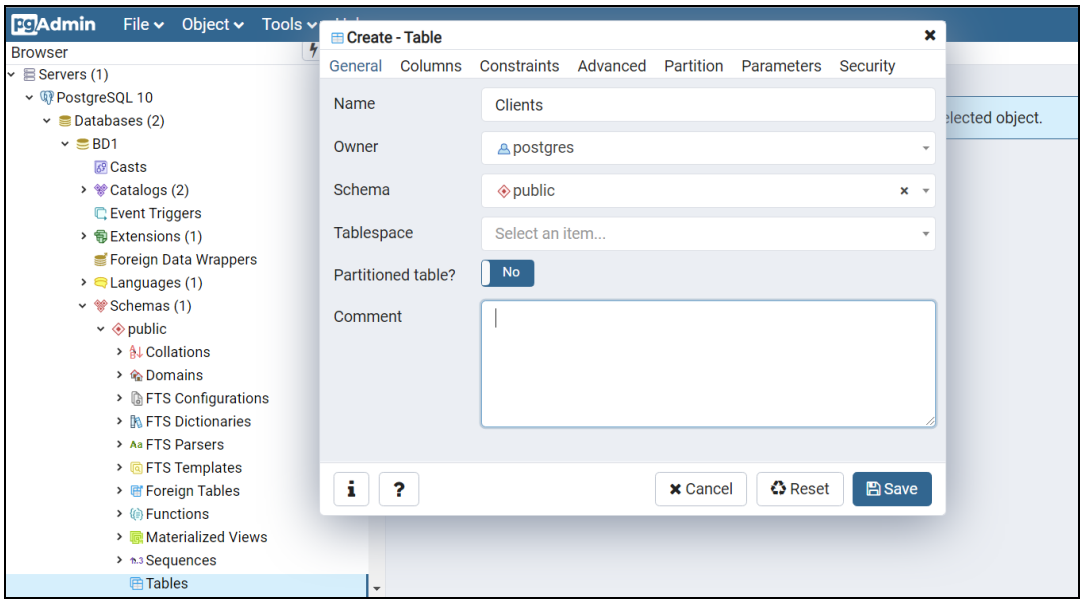

Рис.5.2 -Редактирование параметров, атрибутов и свойств новой таблицы

Во вкладке SQL можно посмотреть код команды создания таблицы Clients (рис. 5.3).

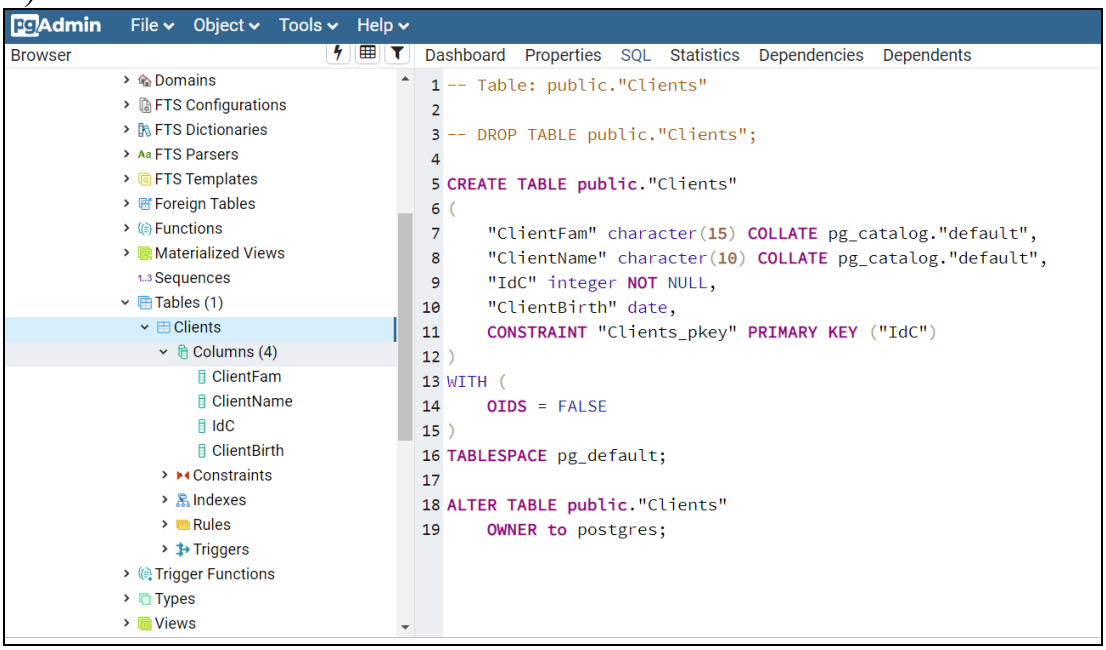

Рис.5.3 Вкладка SQL с кодом создания новой таблицы

Создадим еще три таблицы: **Product** (Товар**), Firms (**Фирмы**)** и **Order** (Заказы).

Таблица **Product** (рис. 5.4).

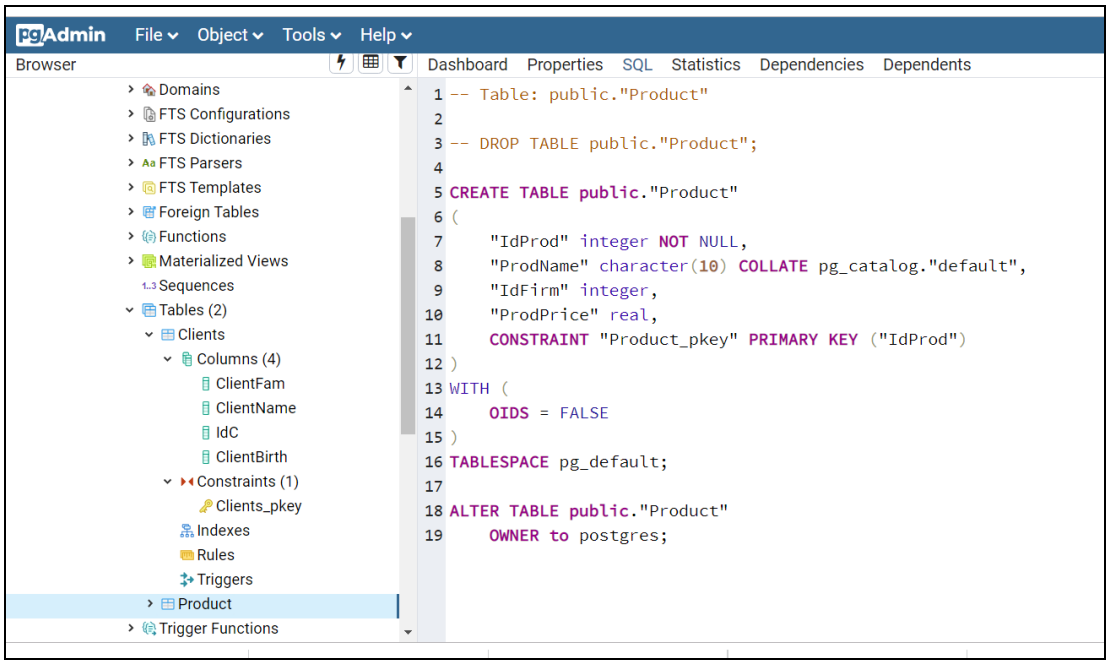

Рис. 5.4 Код для создания таблицы **Product**

Таблица **Firms** (рис. 5.5).

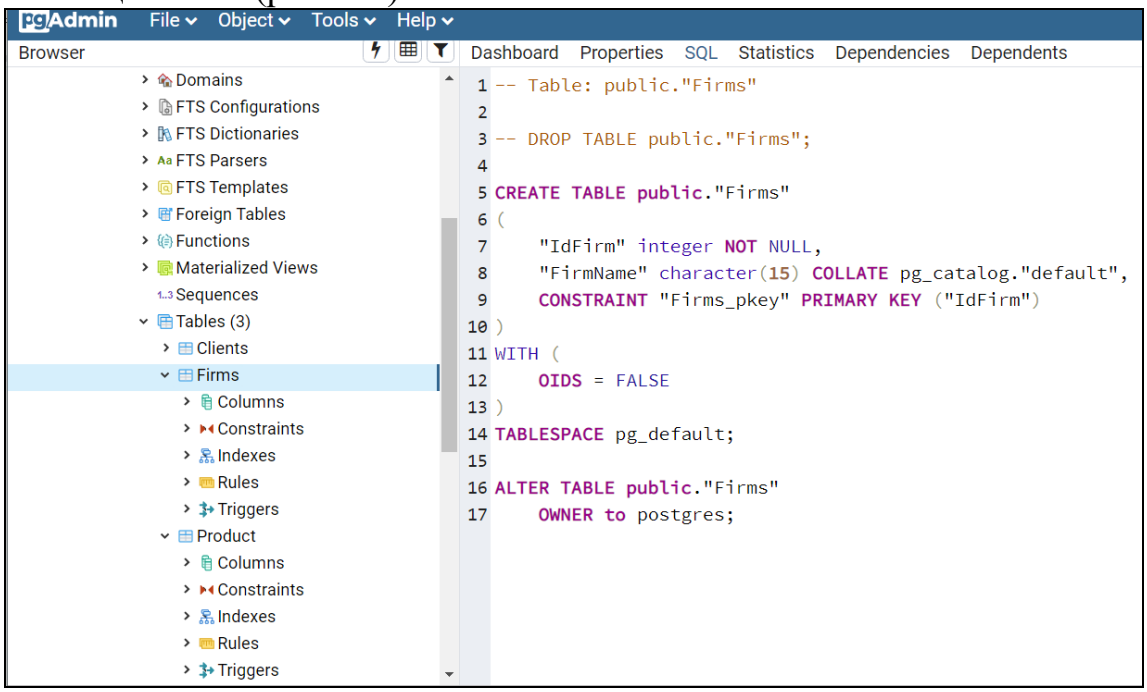

Рис. 5.5 Код для создания таблицы **Firms**

Таблица **Order** (рис. 5.6).

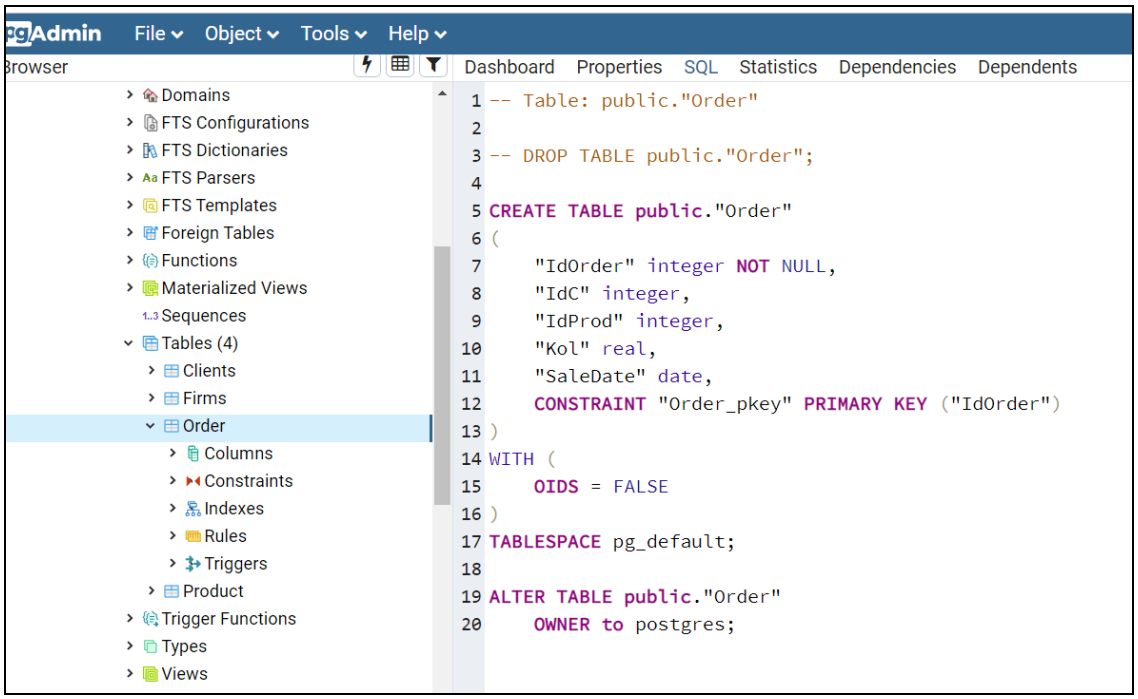

Рис. 5.6 Код для создания таблицы **Order**

Для удаления таблицы используется команда:
DROP Table ИмяТаблицы.

Первичный ключ (Primary Key)

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов является уникальным идентификатором строк в таблице. Для этого требуется, чтобы значения были уникальными и отличными от NULL. Таблица может иметь исключительно один первичный ключ. Формат определения первичного ключа может быть разным. Например, следующие две записи при создании таблицы эквивалентны:

…IdProd integer UNIQUE NOT NULL
или
IdProd integer PRIMARY KEY

Если первичный ключ составной (включает несколько столбцов), можно записать так:

CREATE TABLE Tab1 (
a integer,
b integer,
c integer,
PRIMARY KEY (a, c));

При добавлении первичного ключа автоматически создаётся уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в первичном ключе, и данные столбцы помечаются как NOT NULL.

Добавление внешнего ключа.

Ограничение внешнего ключа указывает, что значения столбца (или группы столбцов) должны соответствовать значениям в некоторой строке другой таблицы. Это называется *ссылочной целостностью* двух связанных таблиц. Например, в таблице Order в заказах должны быть только существующие продукты (которые есть в таблице Product).

Поэтому в таблице Order определим ограничение внешнего ключа, ссылающееся на таблицу Product:

```
CREATE TABLE public."Order" (
   "IdOrder" integer PRIMARY KEY,
   "IdC" integer,
   "IdProd" integer REFERENCES product (IdProd),
   "Kol" real,    ……..)
```

С таким ограничением будет невозможно создать заказ со значением IdProd, отсутствующим в таблице Product. В этой схеме таблицу Order называют ***подчинённой*** таблицей, а **Product** — ***главной*.** Соответственно, столбцы называют так же подчинённым и главным (или ссылающимся и целевым).

Предыдущую команду можно сократить, если опустить список столбцов (т.е. внешний ключ будет неявно связан с первичным ключом главной таблицы).

```
CREATE TABLE public."Order" (
   "IdOrder" integer PRIMARY KEY,
   "IdC" integer,
   "IdProd" integer REFERENCES product,
   "Kol" real,    ……..)
```

Внешний ключ также может ссылаться на группу столбцов. В этом случае его можно записать в виде обычного ограничения таблицы. Например:

```
CREATE TABLE Tab1 (
   a integer PRIMARY KEY
   b integer,
   c integer,
   FOREIGN KEY (b, c) REFERENCES Tab2 (b1, c1) )
```

Число и типы столбцов в ограничении должны соответствовать числу и типам целевых столбцов. Ограничению внешнего ключа можно назначить имя стандартным способом.

Таблица может содержать несколько ограничений внешнего ключа. Это бывает полезно для связи таблиц в отношении многие-ко-многим. Например, есть таблицы продуктов и заказов, но нужно, чтобы один заказ мог содержать несколько продуктов (это невозможно в предыдущей схеме). Для этого можно использовать такую схему (пример 2):

CREATE TABLE Product (
IdProduct integer PRIMARY KEY,
ProdName char(10),
…….);

CREATE TABLE Order (
IdOrder integer PRIMARY KEY,
OrderName text,
...);

CREATE TABLE OrdeItems (
IdProduct integer REFERENCES Product,
IdOrder integer REFERENCES Order,
quantity integer,
PRIMARY KEY (IdProduct, IdOrder)
);

В последней таблице первичный ключ покрывает внешние ключи.

Как быть, если после создания заказов с определённым продуктом мы захотим его удалить? Могут быть такие варианты поведения:

• запретить удаление продукта;
• удалить также связанные заказы;
• другое.

Для примера выше (пример 2), при попытке удаления продукта, на который ссылаются заказы (через таблицу *OrdeItems*), запретим эту операцию. Если попытаются удалить заказ, то удалится и его содержимое. В таблице OrderItems это будет выглядеть так:

CREATE TABLE OrdeItems (
IdProduct integer REFERENCES Product **ON DELETE RESTRICT***,*
IdOrder integer REFERENCES Order, **ON DELETE CASCADE,**
quantity integer,
PRIMARY KEY (IdProduct, IdOrder)
);

RESTRICT предотвращает удаление связанной строки.

NO ACTION означает, что, если зависимые строки продолжают существовать при проверке ограничения, возникает ошибка (это поведение по

умолчанию). (Главным отличием этих двух вариантов является то, что NO ACTION позволяет отложить проверку в процессе транзакции, а RESTRICT — нет.)

CASCADE указывает, что при удалении связанных строк зависимые от них будут так же автоматически удалены.

Есть ещё варианты: SET NULL и SET DEFAULT. При удалении связанных строк они назначают зависимым столбцам в подчинённой таблице значения NULL или значения по умолчанию, соответственно.

Аналогично указанию ON DELETE существует ON UPDATE, которое срабатывает при изменении заданного столбца. При этом возможные действия те же, а CASCADE в данном случае означает, что изменённые значения связанных столбцов будут скопированы в зависимые строки.

Для примера 1 добавляем внешний ключ в таблицу Product для связи с таблицей Firms (рис. 5.7).

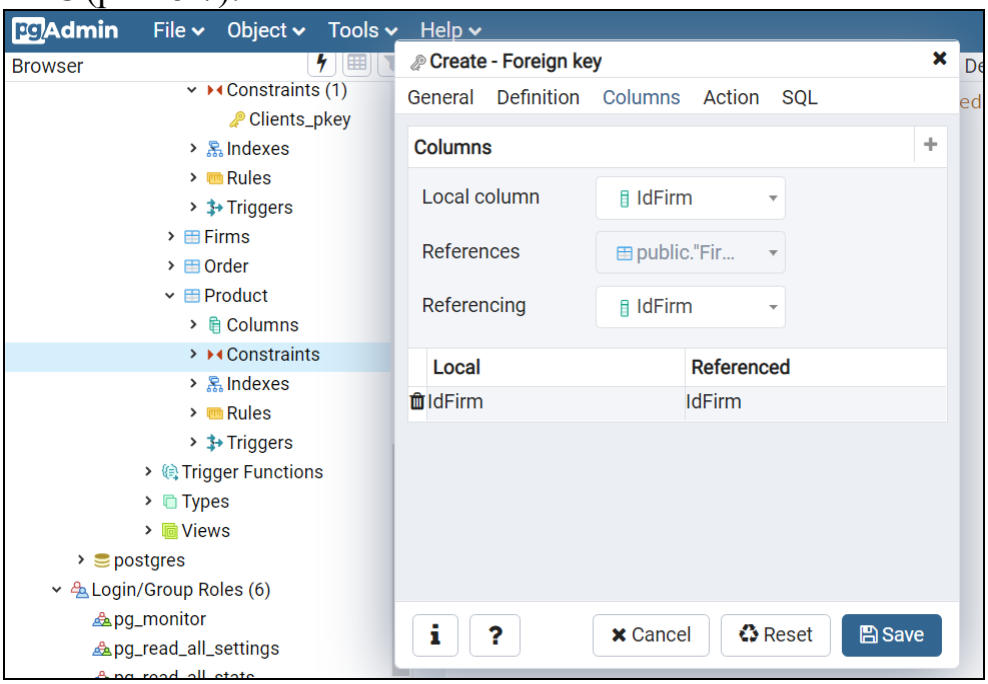

Рис. 5.7 Создание внешнего ключа с помощью pgAdmin

На вкладке SQL можно посмотреть код (рис. 5.8).

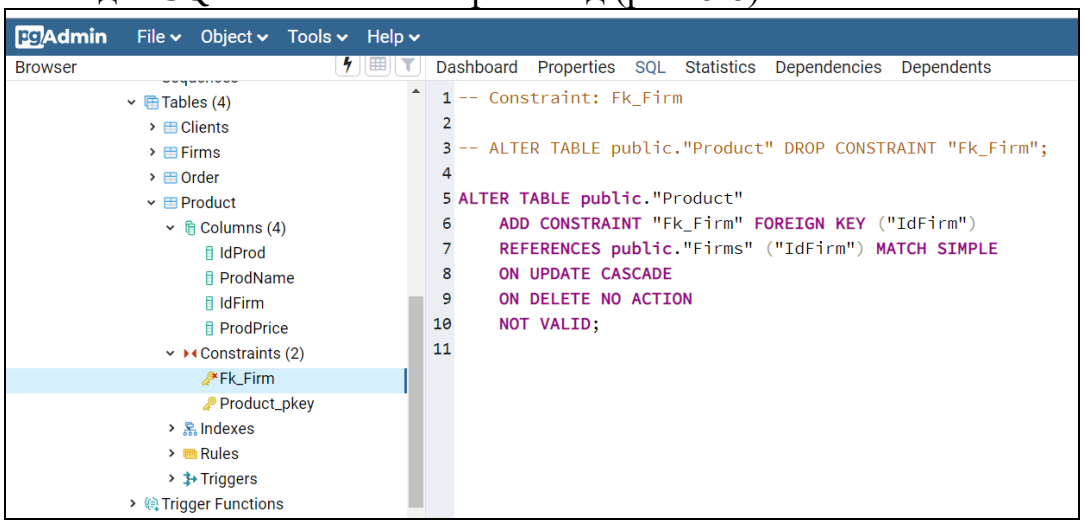

Рис.5.8 Вкладка SQL с кодом добавления внешнего ключа для ранее созданной таблицы.

Для связи с таблицей Clients добавим внешний ключ в таблицу Order.

ALTER TABLE public."Order"
ADD CONSTRAINT "IdC" FOREIGN KEY ("IdC")
REFERENCES public."Clients" ("IdC") MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO AC*TION*
NOT VALID;

Для связи с таблицей Product добавим внешний ключ в таблицу Order.

ALTER TABLE public."Order"
ADD CONSTRAINT "Fk_Prod" FOREIGN KEY ("IdProd")
REFERENCES public."Product" ("IdProd") MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO ACTION
NOT VALID;

Значения по умолчанию

Столбцу можно назначить значение по умолчанию. Если при добавлении новой строки каким-то её столбцам не присваиваются значения, эти столбцы принимают значения по умолчанию. Если значение по умолчанию не объявлено явно, им считается значение NULL. В определении таблицы значения по умолчанию указываются после типа данных столбца.

Например:

CREATE TABLE Product (
IdProduct integer,
ProdName char(10),
ProdPrice numeric ***DEFAULT 9.99***
);

Значение по умолчанию может быть выражением, которое в этом случае вычисляется в момент присваивания значения по умолчанию (а *не* когда создаётся таблица). Например, столбцу timestamp в качестве значения по умолчания часто присваивается CURRENT_TIMESTAMP, чтобы в момент добавления строки в нём оказалось текущее время.

Ещё один полезный пример — генерация «последовательных номеров» для всех строк. В PostgreSQL это можно сделать так:

CREATE TABLE products (
product_no integer ***DEFAULT nextval('products_product_no_seq')****,*
...
);

здесь функция nextval() выбирает очередное значение из *последовательности.* Это употребление настолько распространено, что для него есть специальная короткая запись:

CREATE TABLE products (

*product_no **SERIAL**,*
...
);

Ограничения

Типы данных сами по себе ограничивают множество данных, которые можно сохранить в таблице. Этого часто недостаточно. Например, столбец, содержащий цену продукта, должен, принимать только положительные значения. Или дата рождения клиентов должна относиться к определенному диапазону значений и т.д.

Для управления данными в таблицах используются ограничения для столбцов и таблиц. Если пользователь попытается сохранить в столбце значение, нарушающее ограничения, возникнет ошибка. Ограничения будут действовать, даже если это значение по умолчанию.

Наиболее общий тип ограничений — **проверка**. В его определении указывается, что значение данного столбца должно удовлетворять логическому выражению (проверке истинности). Как и значение по умолчанию, ограничение определяется после типа данных. Например, ограничим цену товара положительными значениями так:

CREATE TABLE Product (
IdProd integer,
...
*price numeric **CHECK (price > 0)***
);

Можно создать именованное ограничение. Это позволит ссылаться на это ограничение, когда понадобится его изменить. Для этого используется ключевое слово CONSTRAINT, за ним идентификатор и определение ограничения. Например:

CREATE TABLE Product (
IdProd integer,
...
*price numeric **CONSTRAINT positive_price** CHECK (price > 0)*
);

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

CREATE TABLE Product (
IdProd integer,
...
price numeric CHECK (price > 0),
discounted_price numeric CHECK (discounted_price > 0),

CHECK (price > discounted_price)
);

Для третьего ограничения используется новый синтаксис. Оно не связано с определённым столбцом, а представлено отдельным элементом в списке. Определения столбцов и такие определения ограничений можно переставлять в произвольном порядке.

Для столбца можно определить больше одного ограничения. Для этого их нужно указать одно за другим:

CREATE TABLE Product (
IdProd integer,
…
price numeric NOT NULL CHECK (price > 0)
);

Примечание.

PostgreSQL не поддерживает ограничения CHECK, которые обращаются к данным, не относящимся к новой или изменённой строке. Хотя ограничение CHECK, нарушающее это правило, может работать в простых случаях, в общем случае нельзя гарантировать, что база данных не придёт в состояние, когда условие ограничения окажется ложным (вследствие последующих изменений других участвующих в его вычислении строк). В результате восстановление выгруженных данных может оказаться невозможным. Во время восстановления возможен сбой, даже если полное состояние базы данных согласуется с условием ограничения, по причине того, что строки загружаются не в том порядке, в котором это условие будет соблюдаться. Поэтому для определения ограничений, затрагивающих другие строки и другие таблицы, используйте ограничения UNIQUE, EXCLUDE или FOREIGN KEY, если это возможно.

Если не нужна постоянно поддерживаемая гарантия целостности, а достаточно разовой проверки добавляемой строки по отношению к другим строкам, можно реализовать эту проверку в собственном триггере. Этот подход исключает вышеописанные проблемы при восстановлении, так как в выгрузке pg_dump триггеры воссоздаются после перезагрузки данных, и поэтому эта проверка не будет действовать в процессе восстановления.

Схемы

Кластер баз данных PostgreSQL содержит один или несколько именованных экземпляров баз.

На уровне кластера создаются роли и некоторые другие объекты. При этом в рамках одного подключения к серверу можно обращаться к данным только одной базы — той, что была выбрана при установлении соединения.

База данных содержит одну или несколько именованных ***схем,*** которые в свою очередь содержат таблицы. Схемы также содержат именованные объекты других видов, включая типы данных, функции и операторы. Одно и то же имя

объекта можно свободно использовать в разных схемах. В отличие от баз данных, схемы не ограничивают доступ к данным: пользователи могут обращаться к объектам в любой схеме текущей базы данных, если им назначены соответствующие права.

Для чего стоит применять схемы:

• Чтобы одну базу данных могли использовать несколько пользователей, независимо друг от друга.

• Чтобы объединить объекты базы данных в логические группы для облегчения управления ими.

• Чтобы в одной базе сосуществовали разные приложения, и при этом не возникало конфликтов имён.

Схемы в некоторым смысле подобны каталогам в операционной системе, но они не могут быть вложенными.

Создадим схему:

CREATE SCHEMA myschema;

Чтобы создать объекты в схеме или обратиться к ним, нужно указать *полное имя*, состоящее из имён схемы и объекта, разделённых точкой: ***схема.таблица***

Таким образом, создать таблицу в новой схеме можно так:

```
CREATE TABLE myschema.mytable (
...
);
```

Для удаления пустой схемы (не содержащую объектов), выполните:

DROP SCHEMA myschema;

Удалить схему со всеми содержащимися в ней объектами можно так:

DROP SCHEMA myschema CASCADE;

Иногда нужно создать схему, владельцем которой будет другой пользователь (это один из способов ограничения пользователей пространствами имён). Сделать это можно так:

CREATE SCHEMA имя_схемы AUTHORIZATION имя_пользователя;

Если при создании таблицы (и других объектов) не указано имя схемы, она автоматически помещается в схему «public». Эта схема содержится во всех создаваемых базах данных. Таким образом, команда:

CREATE TABLE products (...);

эквивалентна:

CREATE TABLE public.products (...);

6. ИЗМЕНЕНИЕ ТАБЛИЦ

PostgreSQL предоставляет набор команд для модификации таблиц.

Можно выполнять следующие действия:

• Добавлять столбцы и ограничения

• Удалять столбцы и ограничения
• Изменять значения по умолчанию
• Изменять типы столбцов
• Переименовывать столбцы и таблицы

Можно изменить таблицу с помощью команды ***ALTER TABLE*** (рис. 6.1).

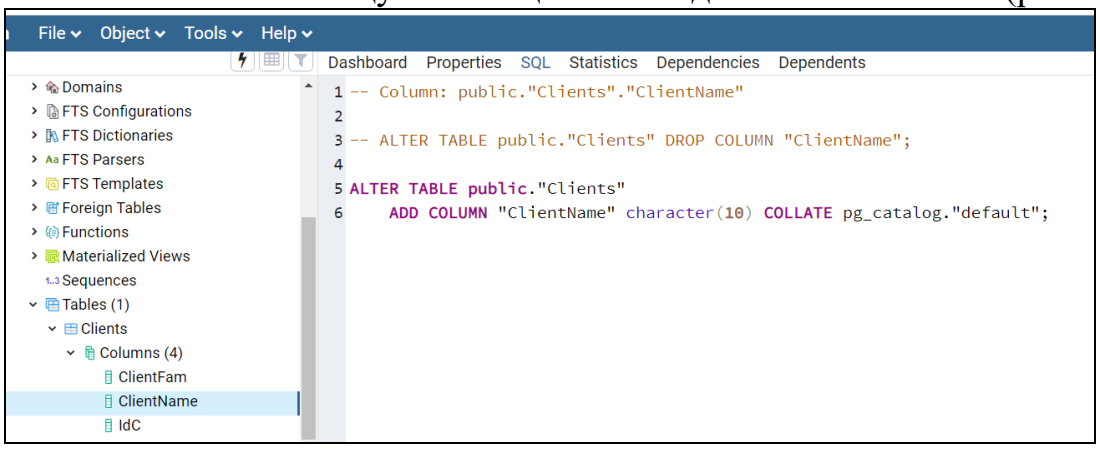

Рис.6.1. Вкладка SQL с кодом модификации (добавления столбца) таблицы

Добавить столбец можно так:

ALTER TABLE products ADD COLUMN description text;

Новый столбец заполняется заданным для него значением по умолчанию (или значением NULL, если вы не добавите указание DEFAULT). При этом можно сразу определить ограничения столбца, используя обычный синтаксис:

ALTER TABLE products ADD COLUMN description text CHECK (description <> '');

Удалить столбец можно так:

ALTER TABLE products DROP COLUMN description;

Данные, которые были в этом столбце, исчезают. Вместе со столбцом удаляются и включающие его ограничения таблицы. Однако, если на столбец ссылается ограничение внешнего ключа другой таблицы, PostgreSQL не удалит это ограничение неявно. Разрешить удаление всех зависящих от этого столбца объектов можно, добавив указание CASCADE:

ALTER TABLE products DROP COLUMN description CASCADE;

Для добавления ограничения используется синтаксис ограничения таблицы. Например:

ALTER TABLE products ADD CHECK (name <> '');

ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);

ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;

Чтобы добавить ограничение NOT NULL, которое нельзя записать в виде ограничения таблицы, используйте такой синтаксис:

ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;

Ограничение проходит проверку автоматически и будет добавлено, только если ему удовлетворяют данные таблицы. Для удаления ограничения вы должны знать его имя. Если вы не присваивали ему имя, это неявно сделала система, и вы должны выяснить его. Здесь может быть полезна команда psql:

\d имя_таблицы (или другие программы, показывающие подробную информацию о таблицах). Зная имя, вы можете использовать команду:

ALTER TABLE products DROP CONSTRAINT some_name;

Как и при удалении столбца, если вы хотите удалить ограничение с зависимыми объектами, добавьте указание CASCADE. Примером такой зависимости может быть ограничение внешнего ключа, связанное со столбцами ограничения первичного ключа.

Назначить столбцу новое значение по умолчанию можно так:

ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;

Это никак не влияет на существующие строки таблицы, а просто задаёт значение по умолчанию для последующих команд INSERT.

Чтобы удалить значение по умолчанию, выполните:

ALTER TABLE products ALTER COLUMN price DROP DEFAULT;

Для преобразования столбца в другой тип данных, используйте команду:

ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);

Она будет успешна, только если все существующие значения в столбце могут быть неявно приведены к новому типу. Если требуется более сложное преобразование, вы можете добавить указание USING, определяющее, как получить новые значения из старых.

Обычно лучше удалить все ограничения столбца, перед тем как менять его тип, а затем воссоздать модифицированные должным образом ограничения.

Чтобы переименовать столбец, выполните:

ALTER TABLE products RENAME COLUMN product_1 TO product_2;

Таблицу можно переименовать с помощью следующей команды:

ALTER TABLE Product RENAME TO NewProduct;

7. ДОБАВЛЕНИЕ, ОБНОВЛЕНИЕ И УДАЛЕНИЕ СТРОК ТАБЛИЦЫ

Для добавления строк в таблицу используется оператор INSERT:

INSERT into table_name (column1, column2, ...) values (value_col1, value_col2...);

С помощью приложения pgAdmin можно сгенерировать SQL-скрипты, относящихся к конкретным объектам в базе данных. Пример генерации SQL-скрипта для добавления новых строк в таблицу (рис. 7.1).

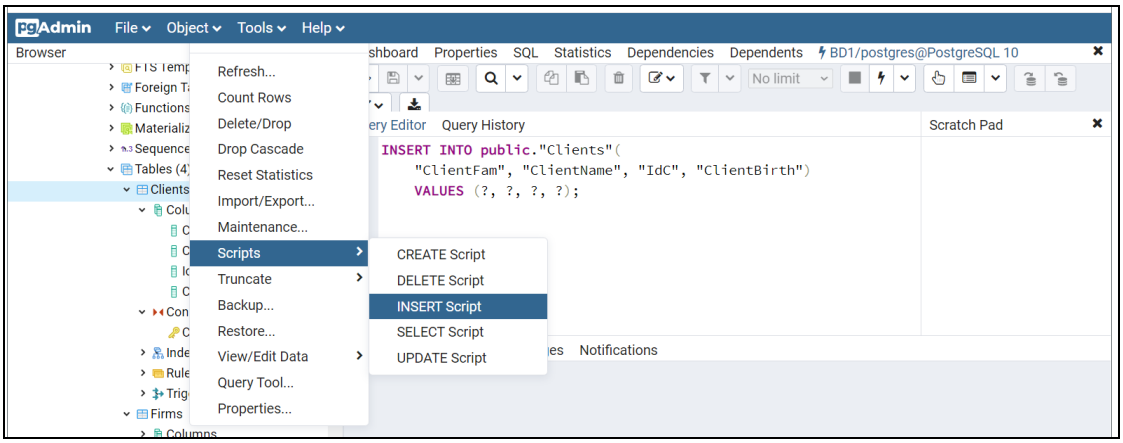

Рис.7.1 Автоматическая генерация SQL-скриптов с помощью pgAdmin

Можно перечислить столбцы в другом порядке, если хотите опустить некоторые из них.

INSERT into table_name (columnk, column1, ..., columnz) values (value_colk, value_col1, ..., value_colz);

Для обновления строк в таблице используется оператор UPDATE:

UPDATE table_name set columnk = value_colk where column_id = value_colid;

Можно написать запрос с выводом результата обновленных столбцов:

UPDATE table_name set columnk = value_colk where column_id = value_colid;

Returning columnk

В рамках одного оператора UPDATE можно обновить несколько столбцов.

Для удаления строк из таблицы используется оператор DELETE:

Удаление всех строк, у которых значение columnk <> value_colk:

DELETE FROM table_name WHERE columnk <> value_colk;

Очистка всех записей из таблицы :

DELETE FROM table_name;

Удаление записей с выводом удалённых строк:

DELETE FROM table_name WHERE columnk = value_colk RETURNING *;

Удаление из tasks строки, на которой в текущий момент располагается курсор c_tasks:

DELETE FROM table_name WHERE CURRENT OF cursor;

В postgreSQL есть оператор очистки TRUNCATE.

Опустошение таблиц table1_name и table2_name:

TRUNCATE table1_name, table2_name;

Та же операция и сброс всех связанных генераторов последовательностей:

TRUNCATE table1_name, table2_name RESTART IDENTITY;

Опустошение таблицы table_name и каскадная обработка всех таблиц, ссылающихся на table1_name по ограничениям внешнего ключа:

TRUNCATE table_name CASCADE;

8. ВЫПОЛНЕНИЕ ЗАПРОСА ВЫБОРКИ

Для получения данные из таблицы, нужно выполнить *запрос*. Для этого предназначен SQL-оператор ***SELECT***. Он состоит из нескольких частей: выборки (в которой перечисляются столбцы, которые должны быть получены), списка таблиц (в нём перечисляются таблицы, из которых будут получены данные) и необязательного условия (определяющего ограничения). В списке выборки можно писать не только ссылки на столбцы, но и выражения.

Запрос можно дополнить «условием», добавив предложение WHERE, ограничивающее множество возвращаемых строк. В предложении WHERE указывается логическое выражение (проверка истинности). В результате оказываются только те строки, для которых это выражение истинно. В этом выражении могут использоваться обычные логические операторы (AND, OR и NOT).

Для примера 1 выведем фамилии и имена клиентов, чьи фамилии начинаются на «А»:

SELECT ClientFam, ClientName FROM CLIENTS WHERE ClientFam LIKE 'A%'

Для создания запроса выборки можно воспользоваться интерфейсом pgAdmin. В строке меню выбрать Tool-QueryToo.

Агрегатные функции

Как большинство других серверов реляционных баз данных, PostgreSQL поддерживает ***агрегатные функции.*** Агрегатные функции выполняют вычисление на наборе значений и возвращают одиночное значение. Аргументами агрегатных функций могут быть как столбцы таблиц, так и результаты выражений над ними. Есть агрегатные функции, вычисляющие: ***COUNT*** (количество), ***SUM*** (сумму), ***AVG*** (среднее), ***MAX***(максимум) и ***MIN*** (минимум) для набора строк.

Предложения GROUP BY, HAVING, ORDER BY, UNION

Предложение GROUP BY позволяет группировать аналогичные данные и используется для определения групп выходных строк, к которым могут применяться агрегатные функции (**COUNT**, **MIN**, **MAX, AVG** и **SUM**). Все выходные строки запроса разбиваются на группы, характеризуемые одинаковыми комбинациями значений в столбцах группировки. После чего к каждой группе применяются агрегатные функции. Если при наличии предложения **GROUP BY**, в предложении **SELECT** отсутствуют агрегатные функции, то запрос просто вернет по одной строке из каждой группы. Предложение **HAVING** применяется после группировки. Оно необходимо для проверки значений, которые получены с помощью агрегатной функции.

Например, выведем общее количество проданного товара для каждого наименования для примера 1 (для простоты оставим IdProd, а не наименование).

SELECT IdProd, sum(Kol) AS total FROM Order GROUP BY IdProd;

Если нужно оставить только те товары, для которых количество проданных больше, например, 100, то запрос будет выглядеть так:

SELECT IdProd, sum(Kol) AS total FROM Order GROUP BY IdProd HAVING sum(Kol) >100;

Для сортировки полученных данных используется **ORDER BY**. Это предложение требует имя столбца, на основе которого будут сортироваться данные.

Следующие два запроса демонстрируют равнозначные способы сортировки результатов по содержимому второго столбца (*columnk*):

SELECT * FROM table_name ORDER BY columnk;

SELECT * FROM table_name ORDER BY 2;

Для объединения таблиц используется ключевое слово **UNION,** которое позволяет объединить результирующие наборы данных двух запросов в один набор данных.

Следующий пример показывает объединение таблиц table1_name и table2_name, ограниченное именами, начинающимися с буквы W в каждой таблице. Интерес представляют только неповторяющиеся строки, поэтому ключевое слово ALL опущено.

Пример:

```
SELECT table1_name.name
   FROM table1_name
   WHERE table1_name.name LIKE 'W%'
UNION
SELECT table_name2.name
   FROM table2_name
   WHERE table2_name.name LIKE 'W%';
```

Команда TABLE

Команда ***TABLE table_name***

равнозначна ***SELECT * FROM table_name.***

Команду TABLE можно применять в качестве команды верхнего уровня или как более краткую запись внутри сложных запросов. С командой TABLE могут использоваться только предложения WITH, UNION, INTERSECT, EXCEPT, ORDER BY, LIMIT, OFFSET, FETCH и предложения блокировки FOR. Предложение WHERE и какие-либо формы агрегирования не поддерживаются.

9. СОЕДИНЕНИЕ ТАБЛИЦ

Запросы могут обращаться сразу к нескольким таблицам или обращаться к той же таблице так, что одновременно будут обрабатываться разные наборы её строк. Запрос, обращающийся к разным наборам строк одной или нескольких таблиц, называется *соединением* (**JOIN)**.

Одним способом соединения таблиц является использование оператора JOIN или INNER JOIN. Он представляет так называемое внутренее соединение. Его формальный синтаксис:

```
SELECT столбцы
FROM таблица1
   [INNER] JOIN таблица2
   ON условие1
   [[INNER] JOIN таблица3
   ON условие2]
```

После оператора JOIN идет название второй таблицы, данные которой надо добавить в выборку. Перед JOIN можно указывать необязательный оператор INNER. Его наличие или отсутствие ни на что не влияет. Далее после ключевого слова ON указывается условие соединения. Это условие устанавливает, как две таблицы будут сравниваться. Как правило, для соединения применяется первичный ключ главной таблицы и внешний ключ зависимой таблицы.

Используя JOIN, выберем все заказы и добавим к ним информацию о товарах (база данных из примера 1):

```
SELECT Order.SaleDate, Order.Kol, Product.ProdName
FROM Order
JOIN Product ON Product.IdProd = Order.IdProd;
```

Поскольку таблицы могут содержать столбцы с одинаковыми названиями, то при указании столбцов для выборки указывается их полное имя вместе с именем таблицы, например, "Orders.IdProd".

С помощью псевдонимов, определяемых через оператор AS, можно сократить код:

```
SELECT O.SaleDate, O.Kol, Ps.ProdName
FROM Order AS O
JOIN Product AS P
ON P.IdProd = O.IdProd;
```

Подобным образом мы можем присоединять и другие таблицы. Например, добавим к заказу информацию о покупателе из таблицы Clients:

```
SELECT Order.SaleDate, Clients.ClientFam, Product.ProdName
FROM Order
JOIN Product ON Product.IdProd = Order.IdProd
JOIN Clients ON Clients.IdC=Order.IdC;
```

Благодаря соединению таблиц мы можем использовать их столбцы для фильтрации выборки или ее сортировки:

```
SELECT Order.SaleDate, Clients.ClientFam, Product.ProdName
FROM Order
JOIN Product ON Product.IdProd = Order.IdProd
JOIN Clients ON Clients.IdC=Order.IdC
WHERE Product.ProdPrice > 45000
```

ORDER BY Clients.ClientFam;

Условия после ключевого слова ON могут быть более сложными по составу. Например, выберем все заказы на товары, производителем которых является Apple.

```
SELECT Order.SaleDate, Clients.ClientFam, Product.ProdName
FROM Order
JOIN Product ON Product.IdProd = Order.IdProd
AND Product.Name='Apple'
JOIN Clients ON Clients.IdC=Order.IdC
ORDER BY Clients.ClientFam;
```

10. ТРАНЗАКЦИИ

Транзакции — это фундаментальное понятие во всех СУБД. Суть транзакции в том, что она объединяет последовательность действий в одну операцию "всё или ничего". Промежуточные состояния внутри последовательности не видны другим транзакциям, и если что-то помешает успешно завершить транзакцию, ни один из результатов этих действий не сохранится в базе данных.

Изменения, производимые открытой транзакцией, невидимы для других транзакций, пока она не будет завершена, а затем они становятся видны все сразу.

В PostgreSQL транзакция определяется набором SQL-команд, окружённым командами ***BEGIN и COMMIT***. Таким образом банковская транзакция по снятию денег Анной могла бы выглядеть так:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Ann';
-- ...
COMMIT;
```

Если в процессе выполнения транзакции мы решим, что не хотим фиксировать её изменения (например, потому что оказалось, что баланс Анны стал отрицательным), мы можем выполнить команду ***ROLLBACK*** вместо COMMIT, и все наши изменения будут отменены.

11. ПРЕДСТАВЛЕНИЯ

Помимо таблиц можно создавать и использовать объекты других типов. Одним из этих объектов является Представление.

Представление (View) является объектом, составляющим логическую структуру любой базы данных. Представление для конечных пользователей выглядит как таблица, однако при этом не содержит данных, а лишь представляет их**.** Физически же представляемые данные расположены в различных таблицах

базы данных. Представления чаще всего используются полностью аналогично таблицам.

Представление реализуется в виде сохраненного запроса**,** на основе которого и производится выборка из различных таблиц базы данных. Этот запрос запрашивает данные из некоторых, но не обязательно всех, столбцов, выборка которых осуществляется из одной или нескольких таблиц. При выборке данных могут применяться или не применяться (в зависимости от определения представления) критерии, которым должны соответствовать данные, содержащиеся в представлении. Представления позволяют вам скрыть внутреннее устройство ваших таблиц, которые могут меняться по мере развития приложения, за надёжными интерфейсами

Синтаксическая структура оператора создания представления выглядит следующим образом:

CREATE VIEW [<schema name>].<view name> [(<column name list>)] [WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW METADATA]]

AS

<SELECT statement> WITH CHECK OPTION

Параметр **WITH ENCRYPTION** определяет шифрование кода запроса и гарантирует, что пользователи не смогут просмотреть и использовать его.

В представлении столбец (столбцы), содержащийся в конструкции WHERE, не обязательно должен быть включен в список выборки. В запросе выборки может быть указана команда SELECT любой сложности, но при этом запрещается использовать раздела ORDER BY. В дальнейшем сортировку можно применить при выборке данных из созданного представления

Например, создадим представление для вывода информации о заказах и товарах для примера 1:

CREATE VIEW viewOrder AS

SELECT Order.SaleDate, Order.Kol, Product.ProdName

FROM Order

JOIN Product ON Product.IdProd = Order.IdProd;

Вызовем на выполнение это представление, упорядочив вывод по дате заказа:

SELECT Order.SaleDate, Product.ProdName FROM viewOrder ORDER BY Order.SaleDate;

12. ВСТРОЕННЫЕ ФУНКЦИИ

Вызов функции записывается просто как имя функции (возможно, дополненное именем схемы) и список аргументов в скобках:

Имя_функции **([*выражение* [*, выражение* ...]])**

Пользователи могут определять свои функции и операторы. Кроме того, PostgreSQL предоставляет огромное количество функций и операторов для встроенных типов данных.

Для множества типов PostgreSQL определены математические и битовые операторы. Битовые операторы работают только с целочисленными типами данных и с битовыми строками bit и bit varying.

В PostgreSQL реализованы математические функции. За исключением случаев, где это указано явно, любая форма функции возвращает результат того же типа, что и аргумент. Определены функции для генерации случайных чисел, тригонометрические функции.

Есть функции и операторы для работы с текстовыми строками. Под строками в данном контексте подразумеваются значения типов character, character varying и text. Если не отмечено обратное, все эти функции работают со всеми этими типами, хотя с типом character следует учитывать возможные эффекты автоматического дополнения строк пробелами.

Например, выполнив конкатенацию строк: ***'string'*** || ***'string',*** получим **stringstring.**

Функция format выдаёт текст, отформатированный в соответствии со строкой формата, подобно функции sprintf в C:

FORMAT(formatstr text [, formatarg "any" [, ...]])

Есть функции и операторы для работы с данными типа byte, для работы с битовыми строками, то есть с данными типов bit и bit varying.

PostgreSQL предлагает три разных способа поиска текста по шаблону: традиционный оператор LIKE языка SQL, более современный SIMILAR TO (добавленный в SQL:1999) и регулярные выражения в стиле POSIX. Помимо простых операторов, отвечающих на вопрос «соответствует ли строка этому шаблону?», в PostgreSQL есть функции для извлечения или замены соответствующих подстрок и для разделения строки по заданному шаблону. Если этих встроенных возможностей оказывается недостаточно, вы можете написать собственные функции на языке Perl или Tcl.

Функции форматирования в PostgreSQL предоставляют богатый набор инструментов для преобразования самых разных типов данных (дата/время, целое, числа с плавающей и фиксированной точкой) в форматированные строки и обратно. Все эти функции следует одному соглашению: в первом аргументе передаётся значение, которое нужно отформатировать, а во втором — шаблон, определяющий формат ввода или вывода.

Например, преобразуем время в текст:

To_char(current_timestamp,'HH12:MI:SS')

Преобразуем строку во время***:***

To_timestamp('05 Dec 2000','DD Mon YYYY')

Есть большое количество функций для работы с датой и временем. Например, для получения текущей даты и времени (на момент начала транзакции)используется функция ***now()***.

EXTRACT(field FROM source) Функция extract получает из значений даты/времени поля, такие как год или час. Здесь *источник (source)* — значение типа timestamp, time или interval. (Выражения типа date приводятся к типу timestamp, так что допускается и этот тип.) Указанное *поле(field)* представляет собой идентификатор, по которому из источника выбирается заданное поле. Функция extract возвращает значения типа double precision.

Например:

SELECT EXTRACT(DAY FROM TIMESTAMP '2020-02-15 20:38:40');

*Результат:*15

SELECT EXTRACT(MONTH FROM TIMESTAMP '2020-02-16 20:38:40');

*Результат:*2

SELECT EXTRACT(YEAR FROM TIMESTAMP '2020-02-16 20:38:40');

Результат:2020

PostgreSQL предоставляет набор функций, результат которых зависит от текущей даты и времени. Так как эти функции возвращают время начала текущей транзакции, во время транзакции эти значения не меняются.

SELECT CURRENT_TIME;

Результат: 14:39:53.662522-05

SELECT CURRENT_DATE;

Результат: 2020-09-03

Большой набор встроенных функций и операторов разработан для геометрических типов point, box, lseg, line, path, polygon и circle.

PostgreSQL позволяет вызывать функции с именованными параметрами, используя запись с *позиционной* или *именной* передачей аргументов. Именная передача особенно полезна для функций со множеством параметров, так как она делает связь параметров и аргументов более явной и надёжной. В позиционной записи значения аргументов функции указываются в том же порядке, в каком они описаны в определении функции. При именной передаче аргументы сопоставляются с параметрами функции по именам и указывать их можно в любом порядке. При записи любым способом параметры, для которых в определении функции заданы значения по умолчанию, можно вовсе не указывать.

PostgreSQL также поддерживает *смешанную* передачу, когда параметры передаются и по именам, и по позиции. В этом случае позиционные параметры должны идти перед параметрами, передаваемыми по именам.

Функцию, принимающую один аргумент составного типа, можно также вызывать, используя синтаксис выбора поля, и наоборот, выбор поля можно записать в функциональном стиле. То есть записи col(table) и table.col равносильны и взаимозаменяемы. Это поведение не оговорено стандартом SQL, но реализовано в PostgreSQL, так как это позволяет использовать функции для эмуляции «вычисляемых полей».

Подробнее с встроенными функциями можно ознакомиться в документации на PostgreSQL.

ЗАДАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

Лабораторная работа 1

Цель работы

Запустить PostgreSQL. Создать с ее помощью базу данных, набор таблиц в ней, заполнить таблицы данными для последующей работы, провести модификацию таблиц.

Содержание работы.

1. Для работы с базой данных вначале необходимо запустить сервер PostgreSQL.

Вывести список имеющихся баз данных с помощью команд **\l и \l+ (аналог SHOW DATABASES)**.

2. Изучить набор команд языка SQL, связанный с созданием базы данных, созданием, модификацией структуры таблиц и их удалением, вставкой, модификацией и удалением записей таблиц:

CREATE DATABASE,
SELECT current_database (),
CREATE TABLE,
\C database_name (для подключения к БД),
\DT
select * from pg_catalog.pg_tables **(SHOW TABLES),**
ALTER TABLE,
DROP TABLE,
DROP DATABASE,
INSERT INTO,
UPDATE,
DELETE.

Вывести и изучить содержимое каталога pg_database с помощью команды: select * from pg_database;

Изучить состав, правила и порядок использования ключевых фраз оператора SELECT:

SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY

Порядок следования фраз в команде **SELECT** должен соответствовать приведенной выше последовательности.

Изучить агрегатные функции.

3. Создать базу данных для отдела реализации книг.

При вводе значений **Название книг и Фамилии авторов – через дефис добавлять свои инициалы и номер в журнале (**Например, есть Иванов Петр Петрович, номер 12 **– «Путешествие Гулливера-ИПП12».**

4. Создать таблицу Books (Книги) (названия книг у разных авторов могут совпадать**).** Создать первичный ключ.

	Название книги	Id автора	Кол. страниц	Дата издания	Примечание
IdBook	**BookName**	**IdAvtor**	**KolStr**	**BookDate**	
1	Стихи	1	60	09.01.2001	

5. Модифицировать таблицу: удалить столбец «Примечание», переименовать столбец kolStr на quantity с изменением типа, добавить столбец под именем rcd с текущем значением даты-времени по умолчанию.

6. Заполнить таблицу данными (8-10 строк). У книги обязательно должен быть IdAvtor.

7. Подготовить и выполнить запросы по выборке информации из таблицы базы данных для решения нижеприведенных задач:

7.1. Вывести все книги какого-то одного автора

Название	IdAvtor

7.2. Все книги, изданные после 2000г.

Название	ГодИздания

7.3. Все книги, изданные в январе 1990г.

Название	Год Издания	Месяц Издания	День издания

7.4. Количество книг у автора с IdAvtor=1.

Количество	IdAvtor

7.5. Общее количество страниц у всех книг (SUM).

7.6. Среднее количество страниц на книгу (AVG).

7.7. Сколько всего авторов (количество уникальных авторов).

7.8. Название книги и IdAvtor с максимальным числом страниц.

7.9. Вывести текущую дату и текущее время отдельными столбцами.

7.10. Вывести список созданных баз данных на сервере. Выбрать 1 базу данных и вывести список таблиц в этой базе.

.

Лабораторная работа 2.

Цель работы

Используя данные базы данных из первой лабораторной работы, создать новую таблицу, используя внешний ключ, подготовить и реализовать серию запросов, связанных с выборкой информации и модификацией данных таблиц.

Содержание работы

1. Создать таблицу Avtor (авторы). Таблица Books должна быть привязана с таблицей Avtor с помощью внешнего ключа. Заполнить (8-10 записей).

Id автора (автоинкремент)	Фамилия автора	Имя автора	Дата рождения автора
IdAvtor	**AvtorFam**	**AvtorName**	**BirthDate**
1	Пушкин	Александр	06.06.1799

2. Изучить соединение таблиц JOIN.

3. Подготовить и выполнить запросы по выборке информации из таблицы базы данных для решения нижеприведенных задач:

3.1. Вывести названия книг, фамилию, имя автора, год издания (join).

Название	Фамилия_Имя_автора	ГодИздания

3.2. Книги с авторами, которые родились до 1960г и после 1980г (5 способами: > <, between, except, UNION , OR).

Название	Фамилия_Имя_автора	ДатаРождения

3.3. Количество книг у каждого автора (group by)

ФамилияИмя	КоличествоКниг

3.4. Количество книг, выпущенное к каждом году

ГодИздания	Количество

3.5. Удалить 1 запись из таблицы Avtor с минимальным значением id с выводом удаленных строк. Удалить 1 запись из таблицы Books с максимальным значением id_avtor. Целостность нарушена, исправить.

3.6. Вывести список книг по убыванию.

3.7. Вывести список авторов, у которых нет книг (2 разными методами: составной запрос и JOIN)

3.8. Вывести список авторов, у которых есть буква "А" в имени.

3.9. Написать запрос SELECT в котором мы заменяем имя первого столбца для таблицы Avtor.

3.10. Удалить первые 2 записи из таблицы books у которых самые молодые авторы.

Лабораторная работа 3.

Цель работы

Используя данные базы данных из предыдущих лабораторных работ, создать новую таблицу, используя внешний ключ, подготовить и реализовать серию запросов, связанных с выборкой информации и модификацией данных таблиц.

Содержание работы

1. Создать таблицу Orders (Поставки книг в магазины), связать ее с остальными таблицами. Заполнить.

В конкретную дату: одно наименование книги может быть поставлено в один магазин только один раз, одно наименование может быть поставлено в разные магазины по разным ценам.

НомерПоставки	Id книги	Магазин /выбор из списка/ Магазин1, Магазин2, Магазин3	Количество	Дата поставки	Цена за шт.
IdOrder	**IdBook**	**Shop**	**KolBook**	**DatePost**	**Price**
0001	1	Магазин1	100	10.09.2020	200

2. Подготовить и выполнить запросы по выборке информации из таблицы базы данных для решения нижеприведенных задач:

2.1. Вывести количество наименований (IdBook) книг, поступивших в каждый магазин (номенклатура)

Магазин	КоличествоНаименований

2.2. Для каждого наименования книги вывести количество магазинов, в которые она поступала.

IdBook	НазваниеКниги	КоличествоМагазинов

2.3. Для каждой книги вывести общее количество книг, переданных в магазины и сумму.

IdBook	НазваниеКниги	Количество Книг	Сумма (руб)

2.4. Написать запрос для выборки последних 3 книг (по дате поставки), с записью результата во временную таблицу. Написать выборку (SELECT * from <имя временной таблицы>). Удалить временную таблицу.

2.5. Написать запрос, в котором выводим список заказов (orders), у которых цена за шт.> 300 за исключением заказов, у которых общая сумма> 1000 применив оператор EXCEPT.

2.6. Добавить внешний ключ в таблице Orders, который будет ссылаться на таблицу книг (books).

2.7. Что происходит при удалении записей из таблицы Books? Что нужно сделать, чтобы при удалении данных из таблицы Books автоматически удалялись записи из таблицы Orders?

2.8. Удалите запись из таблицы orders. СУБД выдает ошибку? Почему?

2.9. Создать представление View, которое содержит номер заказа, общее количество книг и общая стоимость.

2.10. Создать представление View, которое выводит фамилии клиентов, в чьем имени и фамилии есть буква «А», ЛИБО тех, чья дата рождения позже 1 января 1995 года/

2.11. Задание на использование with.

Лабораторная работа 4.

Цель работы

Используя данные базы данных из предыдущих лабораторных работ, подготовить и реализовать серию запросов, связанных с выборкой информации и модификацией данных таблиц.

Содержание работы

В данной работе используется база данных, созданная в предыдущих лабораторных работах.

1.Создать представление для вывода записей из одной таблицы. Представление должно содержать один дополнительный столбец, в котором будет текущая дата на момент обращения к нему.

2. Написать скалярную функцию, которая переводит цену в УЕ. Входными параметрами этой функции будут: цена и курс валюты на текущий день.

3. Написать табличную функцию, которая выводит список книг, у которых цена меньше 100. Написать запрос к этой функции.

4. Создать новую таблицу books_new, идентичной таблице books по структуре. Заполнить таблицу 2 записями (1 из записей должна совпадать из списка записей в таблице books). Использовать конструкцию INSERT+SELECT.

5.Написать команду MERGE, при которой новая запись из таблицы books_new пойдет на вставку в таблицу books, а идентичная запись пойдет на удаление.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Документация к PostgreSQL 10.13, The PostgreSQL Global Development Group, Перевод на русский язык, 2015-2019 гг.:,Компания «Постгрес Профессиональный»: *https://postgrespro.ru/docs/*

2. Дейт К. Дж. Введение в системы баз данных. – М.: Вильямс, 2008

3. Regina, Obe PostgreSQL – Up and Running / Regina Obe. - Москва: СИНТЕГ, 2012. - 166 с.

4. Ригс, Саймон Администрирование PostgreSQL 9. Книга рецептов / Саймон Ригс , Ханну Кросинг. - М.: ДМК Пресс, 2015. - 364 с.

5. Стоунз PostgreSQL. Основы / Стоунз, Мэттью Ричард; , Нейл. - М.: СПб: Символ-Плюс, 2002. - 640 с.

6. Уорсли, Дж. PostgreSQL. Для профессионалов (+ CD) / Дж. Уорсли, Дж. Дрейк. - М.: СПб: Питер, 2002. - 496 с.