Министерство науки и 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)



**УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ К ЛАБОРАТОРНЫМ РАБОТАМ ПО**

**КУРСУ «БАЗЫ ДАННЫХ»**

**Лабораторная работа №1**

**«Введение в SQL»**

Авторы:

Кудрявцев А.П., [kudryavtsevap@bmstu.ru](mailto:kudryavtsevap@bmstu.ru)

Фомин М.М.

Москва, 2025

# Сокращения

SQL– Structured Query Language («язык структурированных запросов»)

БД – База данных

СУБД – Система управления базами данных

РБД – Реляционная база данных

# 1. Краткая информация о СУБД PostgreSQL

**PostgreSQL** – комплекс программ, относящийся к классу систем управления базами данных. Когда эта система выполняется, мы называем ее сервером PostgreSQL, или экземпляром сервера.

Данные, которыми управляет PostgreSQL, хранятся в базах данных. Один экземпляр PostgreSQL может одновременно управлять несколькими базами, которые вместе называются кластером баз данных.

Чтобы кластер можно было использовать, его необходимо инициализировать (создать). Каталог, в котором размещаются все файлы, относящиеся к кластеру, обычно называют - **PGDATA**, по имени переменной окружения, указывающей на этот каталог.

# 2. Установка программного обеспечения

## 2.1. Установка СУБД PostgreSQL

Для выполнения заданий лабораторной работы предлагается использовать СУБД PostgreSQL 14 или более новые версии, дистрибутивы которых можно скачать по ссылке: <https://www.postgresql.org/download/>

Там же имеются инструкции по установке PostgreSQL на различные ОС.

### 2.1.1. Windows

Запустите установщик. Он предложит установить нужные компоненты для работы с PostgreSQL (рис. 2.1).

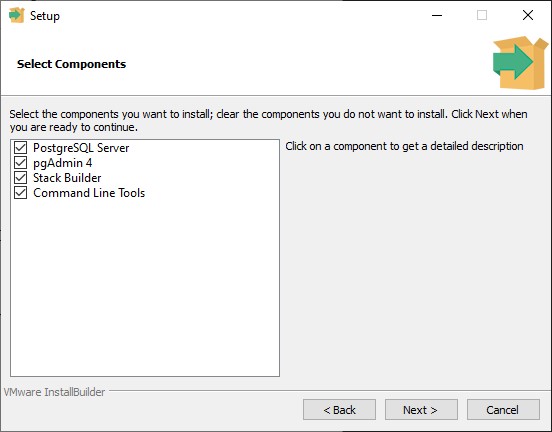


Рисунок 2.1 – Выбор компонентов

Обязательные компоненты - сам PostgreSQL Server и pgAdmin 4, остальное не существенно. Следующим выбором будет каталог, где планируется хранить ваши базы данных (рис. 2.2).

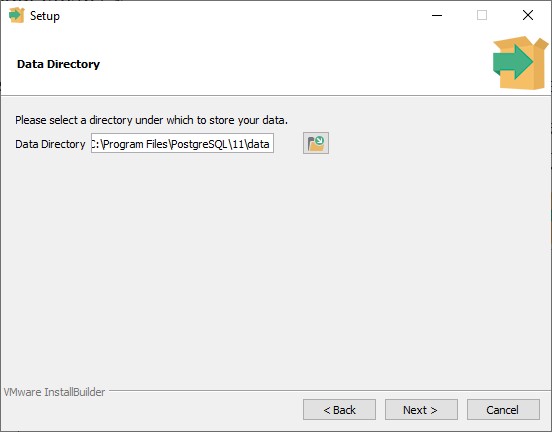


Рисунок 2.2 – Каталог данных

Далее нужно будет установить пароль для вашей базы данных, пароль рекомендуется где-нибудь записать (рис. 2.3).

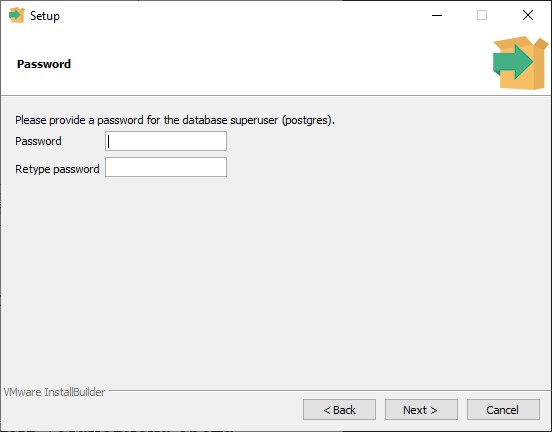


Рисунок 2.3 – Ввод пароля

Так как СУБД является сервером, вам предложат выбрать порт, по которому будет идти соединение с ним, лучше всего оставить значение порта по умолчанию, если конечно в вашей системе он не занят (рис. 2.4).

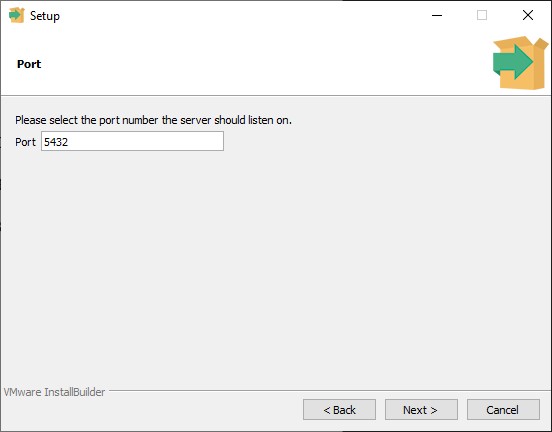


Рисунок 2.4 – Выбор порта

Последним действием можно выбрать кодировку (язык) данных, его так же можно оставить по умолчанию (Default locale – будет опираться на настройки системы).

После установки предлагается установить дополнительные приложения для СУБД. Этот пункт можно пропустить, а в дальнейшем, если понадобится что-то установить, просто наберите в строке поиска Windows название установщика (Application Stack Builder) и дополните вашу среду.

Установка pgAdmin 4

В результате установки СУБД PostgreSQL должна быть установлена среда pgAdmin 4 для подключения к СУБД и выполнения SQL-запросов. Если этого не произошло, то ее необходимо установить отдельно, скачав по ссылке: <https://www.pgadmin.org/download/>.

### 2.1.2. Linux

Ниже излагается установка PostgreSQL 14 на UBUNTU.

Запустите терминал, например, нажав комбинацию клавиш Ctrl+Alt+T.

Войдите с правами пользователя root, выполнив команду:

sudo -i

После нажатия Enter, будет предложено ввести пароль. А далее появится приглашение для ввода команд:

root@...:~#

Проверим, установлена ли утилита **wget:**

which wget

Если установлена, то отобразится путь к утилите:

/usr/bin/wget

В противном случае, установим:

apt update

apt install wget

Создаем конфигурацию файлового репозитория:

sh -c 'echo "deb https://apt.postgresql.org/pub/repos/apt $(lsb\_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'

Импортируем ключ подписи репозитория:

wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -

Обновите списки пакетов:

apt-get update

Устанавливаем 14 версию:

apt-get -y install 'postgresql-14'

(Возможно, потребуется исправить кавычки).

При установке пакета инсталлятор создаст новый PostgreSQL-кластер. Данный кластер представляет из себя коллекцию баз данных, которая управляется одним сервером. Также установщик создаст рабочие директории для PostgreSQL. Данные, необходимые для работы PostgreSQL, будут находиться в папке **/var/lib/postgresql/14/main**, а файлы конфигурации – в папке **/etc/postgresql/14/main**.

После завершения установки вы можете убедиться, что служба PostgreSQL активна. Для этого в командной строке наберите:

systemctl is-active postgresql

Также посмотрите, включена ли служба:

systemctl is-enabled postgresql

И наконец, вы можете увидеть статус службы PostgreSQL:

systemctl status postgresql

● postgresql.service - PostgreSQL RDBMS

Loaded: loaded (/lib/systemd/system/postgresql.service; enabled; vendor preset: enabled)

Active: active (exited) since Tue 2023-12-26 08:51:52 MSK; 1h 24min ago

Process: 1357 ExecStart=/bin/true (code=exited, status=0/SUCCESS)

Main PID: 1357 (code=exited, status=0/SUCCESS)

CPU: 784us

…

В PostgreSQL автоматически создается пользователь ОС без права подключения и одноименный суперпользователь БД без пароля – postgres. И первым шагом начала работы добавим суперпользователя БД с паролем. Подключитесь к системе с помощью учётной записи postgres:

su – postgres

Должно появиться приглашение от имени **postgres**:

postgres@...:~$ exit

Подключившись, выполните команду psql:

psql

Если утилита не запускается, то скорее всего указан порт не 5432 в файле **/etc/postgresql/14/main/postgresql.conf**. Возможны два варианта: первый, для psql указывать порт подключения; второй, это - выяснить, какая программа использует этот порт (скорее всего ранее была неудачная попытка установитьPostgreSQL) и, решившись, измените порт в файле **postgresql.conf.**

Если вы видите приглашение ко вводу команд **postgres=#**, значит вы находитесь в оболочке СУБД PostgreSQL. И значит, можно приступать к добавлению пользователя, созданию базы данных и т.д. Добавим в PostgreSQL пользователя (на своем сервере вы можете использовать свои имена пользователей и баз данных):

CREATE USER имя\_пользователя WITH SUPERUSER LOGIN PASSWORD 'пароль';

Добавив пользователя, завершим работу с PostgreSQL командой:

exit

Все работы будут проводиться на локальном компьютере, это позволит нам избежать ввода пароля при подключении к БД из терминала, изменив файл **/etc/postgresql/14/main/pg\_hba.conf**. Откроем его:

nano /etc/postgresql/14/main/pg\_hba.conf

Найдем строку:

# "local" is for Unix domain socket connections only

local all all peer

И вместо peer ведем **trust**.

# "local" is for Unix domain socket connections only

local all all trust

Сохраним (Ctrl+S) и выйдем (Ctrl+X). Перезапусти PostgreSQL:

systemctl restart postgresql

systemctl status postgresql

Проверим возможность подключения к БД postgres без ввода пароля от имени только что созданного пользователя:

psql postgres -Uимя\_пользователя

Должно появиться приглашение:

postgres=#

И выходим:

exit

И если далее не собираетесь устанавливать pgAdmi4, то отключитесь от root командой:

Logout

Установка pgAdmin 4

pgAdmin 4 является графическим клиентом для работы с сервером, через который удобно удалять, изменять базы данных и управлять ими, а также создавать и выполнять SQL-запросы.

Дистрибутив для установки можно найти на сайте   
https://www.pgadmin.org /download/.

Ниже излагается установка pgAdmin 4 на UBUNTU.

Запустите терминал, например, нажав комбинацию клавиш Ctrl+Alt+T. Войдите с правами пользователя root, выполнив команду:

sudo -i

Проверьте установлена ли утилита curl, запустив команду:

**which curl**

Если утилита установлена, то будет указан к ней путь:

**/snap/bin/curl**

В противном случае, надо curl установить:

**apt update**

**apt install curl**

Установим открытый ключ репозитория (если это не было сделано ранее):

**curl -fsS https://www.pgadmin.org/static/packages\_pgadmin\_org.pub | sudo gpg --dearmor -o /usr/share/keyrings/packages-pgadmin-org.gpg**

Создаем файл конфигурации репозитория:

**sh -c 'echo "deb [signed-by=/usr/share/keyrings/packages- pgadmin-org.gpg] https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/$(lsb\_release -cs) pgadmin4 main" > /etc/apt/sources.list.d/pgadmin4.list && apt update'**

Установим pgAdmin 4 только для рабочего стола:

**apt install pgadmin4-desktop**

## 2.2. Настройка pgAdmin 4

Подключим pgAdmin 4 к PostgreSQL. Запустите pgAdmin 4. На стартовой странице (рис. 2.5) щелкните по **Add New Server**.

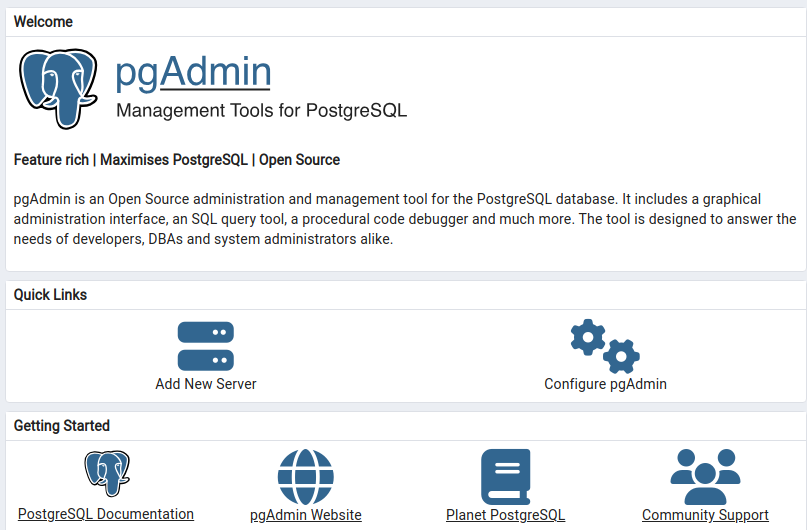


Рисунок 2.5 – Стартовая страница

Откроется диалог для регистрации сервера (рис. 2.6):



Рисунок 2.6 – Диалог регистрации сервера

На первой вкладке в поле **Name** введем название сервера. Далее, щелкнем по вкладке **Connection** (рис. 2.7):

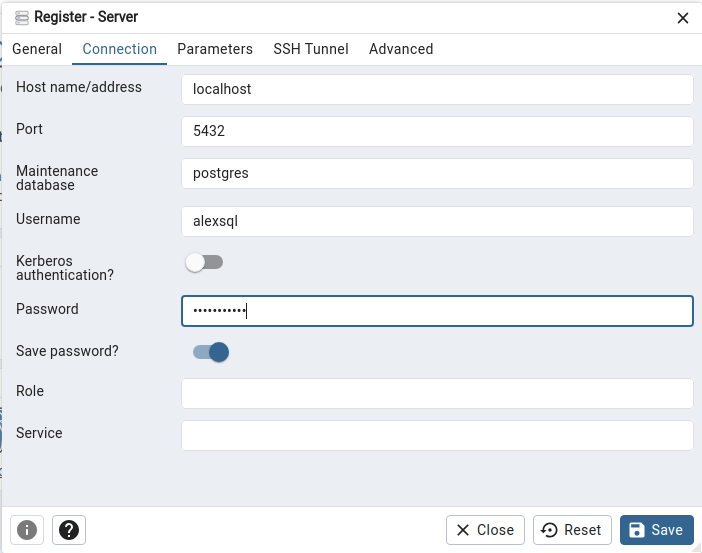


Рисунок 2.7 – Вкладка соединение

Если при установке PostgreSQL был выбран другой порт (и он не был изменен), то в поле **Port** введите его значение. Сохраните, нажав на кнопку **Save**.

## 2.3. Альтернатива PgAdmin 4

Универсальный инструмент для управления СУБД - система DBeaver. Это бесплатный многоплатформенный инструмент для работы разработчиков, администраторов, аналитиков и других специалистов, которым необходимо работать с БД. Поддерживает все популярные базы данных: MySQL, PostgreSQL, SQLite, Oracle, DB2, SQL Server, Sybase, MS Access, и др.

Скачать DBeaver можно на официальном сайте: https://dbeaver.io/

## 2.4. Установка демонстрационной базы данных

Скачайте последнюю версию демонстрационной БД с сайта:

<https://postgrespro.ru/education/demodb>. Там же находится описание БД, с которой необходимо ознакомиться.

Следующим шагом надо распаковать скаченный файл. В итоге в выбранной папке создастся файл **demo-<…>.sql**.

Для установки демонстрационной базы данных нам понадобится утилита psql. Программа psql – это терминальный клиент для работы с PostgreSQL. Она позволяет интерактивно вводить запросы, передавать их в PostgreSQL и видеть результаты. Также запросы могут быть получены из файла или из аргументов командной строки. Кроме того, psql предоставляет пользователю различные возможности, подобные тем, что имеются у командных оболочек, для облегчения написания скриптов и автоматизации широкого спектра задач.

### 2.4.1. Windows

Как только вы откроете консоль, вам сразу же предложат ввести IP сервера, названия базы данных, порт, имя пользователя. В процессе установки PostgreSQL был создан сервер и пустая БД со значениями по умолчанию, они и показаны в квадратных скобках, поэтому можно просто нажать несколько раз на Enter до момента запроса пароля (рис. 2.8) и ввести тот пароль, который вы вводили при установке СУБД. (символы при вводе пароля могут не отображаться).

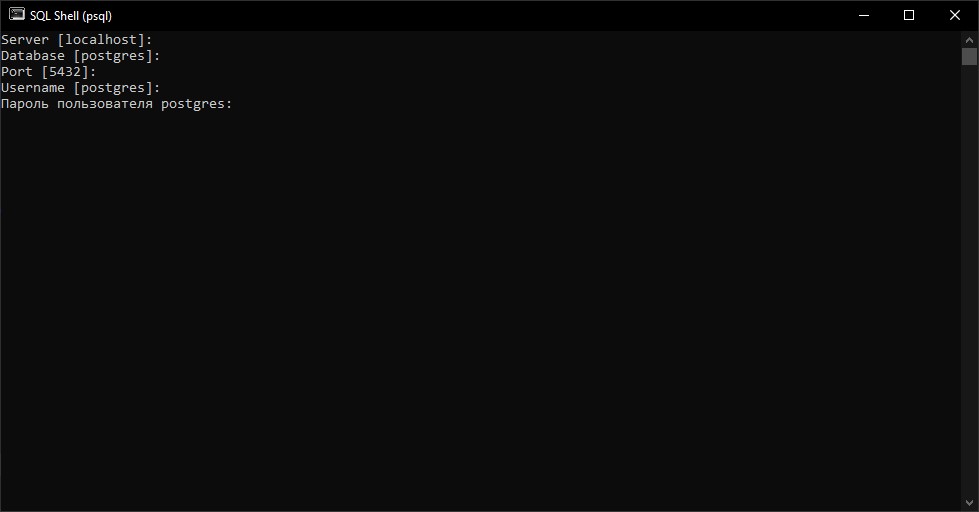


Рисунок 2.8 – Консоль клиент psql

Если у вас появится предупреждение о различных используемых кодировках, как на рисунке 2.9 ниже, то необходимо сменить кодовую страницу, выполнив команду:

postgres=# \! chcp 1251

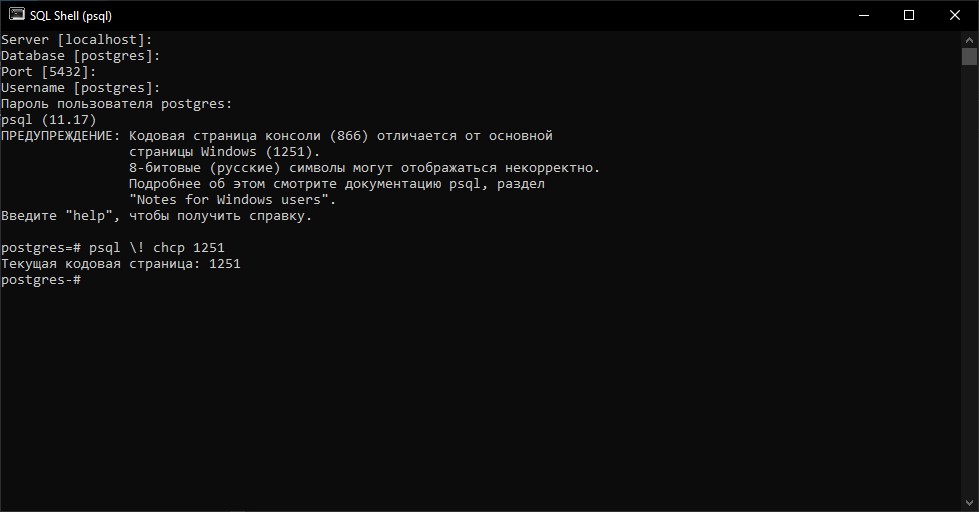


Рисунок 2.9 – Смена кодировки

Развернем БД demo, выполнив команду:

\i 'путь\_к\_файлу\\demo-<…>.sql'

### 2.4.4. Linux

Запустим терминал, подключаемся к БД postgres:

psql postgres – U <имя\_пользователя>

И подключаем demo к нашей БД:

postgres=# \i 'путь\_к\_файлу/demo-<…>.sql'

### 2.4.5. Универсальный

Запустим pgAdmin 4. В левой панели отобразятся ваши БД (рис. 2.10):

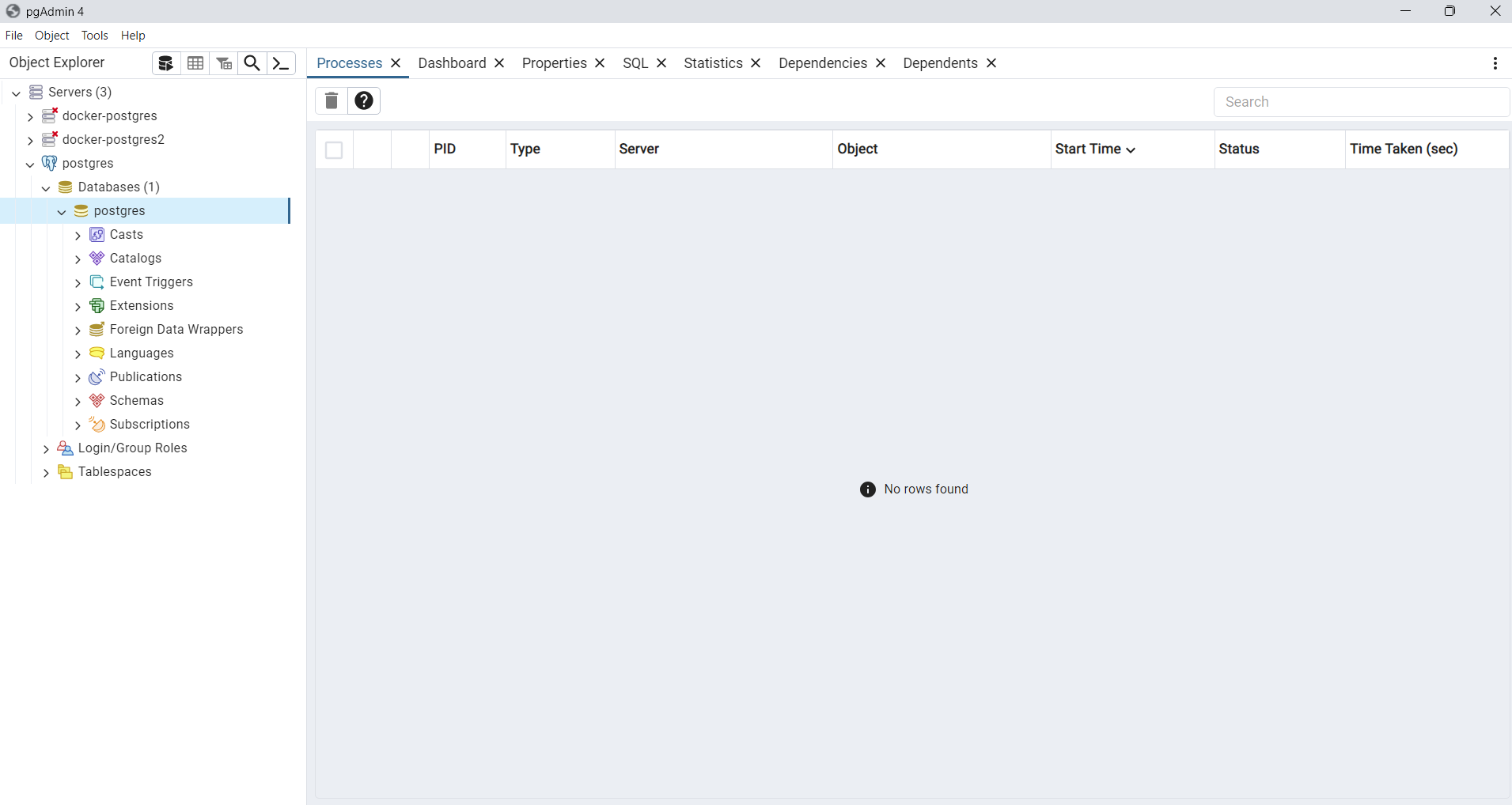


Рисунок 2.10 – Панели pgAdmin 4

Если вы до этого не работали с Postgres, то вам будет доступна БД postgres, установленная при развертывании PostgreSQL.

Выделите БД postgres и щелкните по значку PSQL Tool (рис. 2.11).

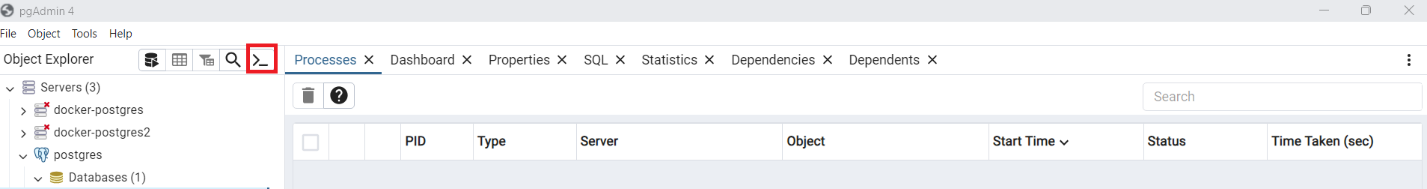


Рисунок 2.11 – Панель инструментов

Если у вас появится предупреждение о различных используемых кодировках, как на рисунке ниже, то необходимо сменить кодовую страницу, выполнив команду (рис. 2.12):

postgres=# \! chcp 1251

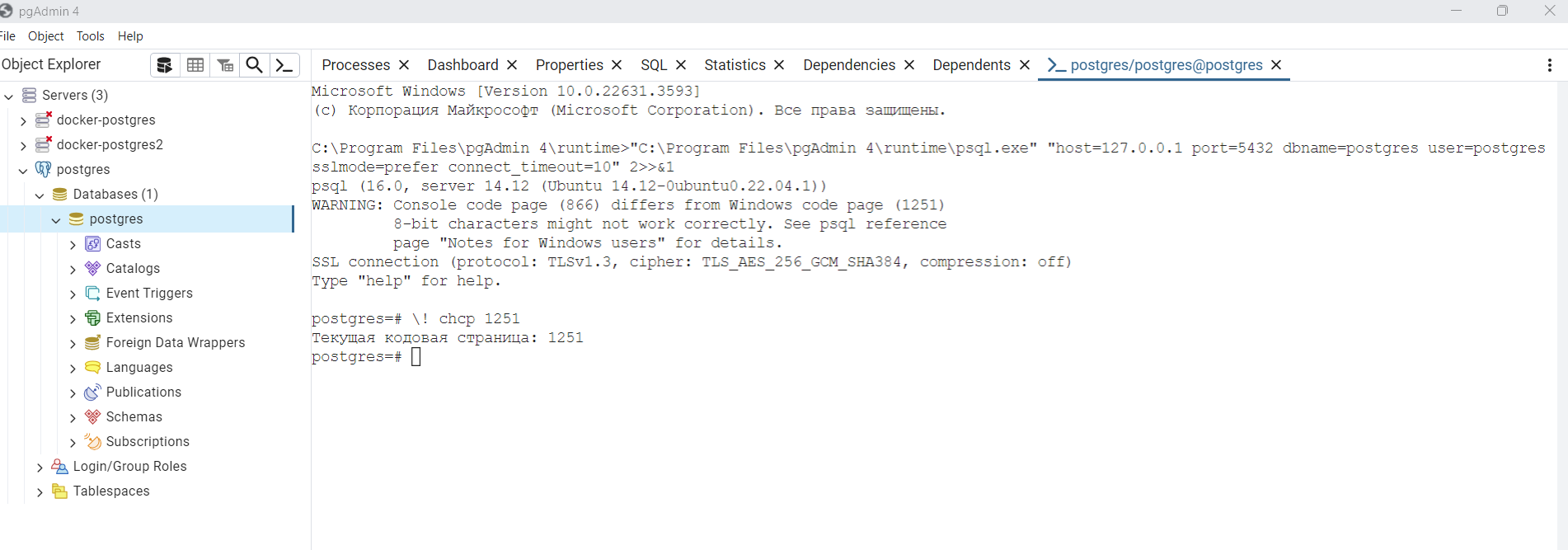


Рисунок 2.12 – Смена кодировки

Развернем БД demo:

postgres=# \i 'C:/путь\_к\_файлу/demo\_<…>.sql'

Щелкните правой кнопкой по Databases и выполните команду меню Refresh, появится БД demo.

# 3. Основы языка SQL. SELECT запросы

## 3.1. Что такое SQL?

***SQL*** (*structured query language* - «язык структурированных запросов») – формальный, непроцедурный, декларативный язык программирования, применяемый для создания, модификации и управления данными в произвольной реляционной базе данных, управляемой соответствующей системой управления базами данных (СУБД).

На рисунке 3.1 показана схема работы пользователя с СУБД.

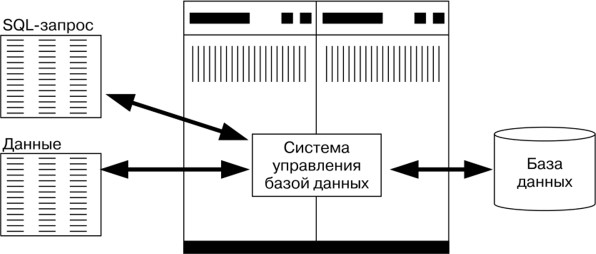


Рисунок 3.1 – Схема работы СУБД

Согласно этой схеме, в вычислительной системе имеется база данных, в которой хранится информация о предметной области. Если вычислительная система относится к сфере бизнеса, то в базе данных могут содержаться сведения о материальных ценностях, выпускаемой продукции, объемах продаж и зарплате. Комплекс программ, которая управляет базой данных, называется системой управления базой данных, или же СУБД.

На самом деле СУБД — это совокупность программных, лингвистических и методических средств общего или специального назначения, обеспечивающих управление созданием и использованием БД.

SQL позволяет пользователю выполнять такие действия как:

* описывать данные (их структуру);
* определять данные в БД и управлять ими;
* читать, создавать и удалять данные, а также другие объекты БД, такие как представления или хранимые процедуры;
* разграничивать доступ пользователей к объектам БД.

## 3.2. Все операторы языка SQL

Одним из основных требований к системам управления базами данных является наличие высокоуровневых средств выполнения запросов. В системах, реализующих реляционную модель данных, в качестве такого средства используется язык SQL. Фактически этот язык содержит полный набор операций, необходимых для выполнения любых действий с базой данных. Стандарты SQL предусматривают подразделение средств языка на несколько категорий. Далее представлены все операторы языка SQL, разбитые на группы в соответствии с современным стандартом:

* Операторы определения данных (*Data Definition Language,* ***DDL***) обеспечивают создание, модификацию и удаление элементов описания структуры базы данных, как логической, так и структуры хранения:
  + CREATE создает объект БД,
  + ALTER изменяет объект,
  + DROP удаляет объект;
* Операторы манипуляции данными (*Data Manipulation Language,* ***DML****)* обеспечивают выполнение поиска, извлечения, добавления, изменения и удаления данных, определенных в описании логической структуры базы данных, но не позволяют изменять эту структуру:
  + SELECT выбирает данные, удовлетворяющие заданным условиям,
  + INSERT добавляет новые данные,
  + UPDATE изменяет существующие данные,
  + DELETE удаляет данные;
* Операторы определения доступа к данным (*Data Control Language,* ***DCL***) обеспечивают разграничение доступа пользователей к данным:
  + GRANT предоставляет пользователю разрешения на определенные операции с объектом БД,
  + REVOKE отзывает ранее выданные разрешения,
* Операторы управления транзакциями (*Transaction Control Language,* ***TCL****)* обеспечивают сохранение БД в согласованном состоянии:
  + START TRANSACTION отмечает начало транзакции,
  + COMMIT сохраняет все изменения текущей транзакции,
  + ROLLBACK откатывает все изменения текущей транзакции,
  + SAVEPOINT делит транзакцию на более мелкие участки.

В расширениях и диалектах языка SQL присутствуют дополнительные средства, обеспечивающие описание особенностей, специфических для конкретной СУБД. Чаще всего это определения конкретных структур хранений и их параметров.

## 3.3. Оператор SELECT

В данной лабораторной работе рассмотрим только один оператор из группы DML — это SELECT.

Оператор SELECT, упрощенный синтаксис которого приведен ниже, извлекает данные, упомянутые в предложении SELECT из таблиц, указанных в предложении FROM в соответствии с правилами, устанавливаемыми в остальных предложениях оператора SELECT.

**SELECT** [{ALL | DISTINCT}] *извлекаемый\_ элемент*

[AS *псевдоним*][, ...]

**FROM** {*таблица* [[AS] *псевдоним*] | *представление*

[[AS] *псевдоним*]}[, ...]

[[*тип\_ соединения*] JOIN *условие\_ соединения*]

[**WHERE** *условие\_ отбора*] [{AND | OR | NOT} *условие\_ отбора* [...]]

[**GROUP BY** *выражение\_ группировки*]

[**HAVING** *условие\_ отбора*]]

[**ORDER BY** {*выражение\_сортировки* [ASC | DESC]}[, ...]]

[**LIMIT** *количество\_строк*]

В таблице 3.1 приведены основные ключевые слова:

**Таблица 3.1. Предложения оператора SELECT**

|  |  |
| --- | --- |
| **Предложение** | **Описание** |
| FROM | Перечисляет используемые в запросе таблицы из базы данных. Все перечисленные таблицы объединяются при помощи «декартова произведения» |
| WHERE | Определяет условие, которое используется для выборки строк. Условие вычисляется для каждой строки таблицы, полученной в предложении FROM, и в итоговую таблицу попадают только строки, для которых это условие - TRUE |
| GROUP BY | Используется для группировки строк |
| HAVING | Определяет условие, которое используется для выборки групп. Работает аналогично предложению WHERE, но не для строк, а для групп. |
| ORDER BY | Определяет выражение, по которому производится сортировка итоговой таблицы |
| LIMIT | Команда LIMIT задает ограничение на количество записей итоговой таблицы |

## 3.4. Типы данных PostgreSQL

Ниже рассмотрены самые необходимые типы данных для выполнения лабораторных работ. Остальные типы данных остаются для самостоятельного изучения.

### 3.4.1. Числовые типы

Числовые типы включают двух-, четырёх- и восьмибайтные целые, четырёх- и восьмибайтные числа с плавающей точкой, а также десятичные числа с задаваемой точностью. Все эти типы перечислены в таблице 3.2.

**Таблица 3.2. Числовые типы**

| Тип данных | Размер | Описание | Диапазон |
| --- | --- | --- | --- |
| smallint | 2 байта | целое в небольшом диапазоне | -32768 ... +32767 |
| integer | 4 байта | типичный выбор для целых чисел | -2147483648 ... +2147483647 |
| bigint | 8 байт | целое в большом диапазоне | -9223372036854775808 ... 9223372036854775807 |
| decimal | переменный | вещественное число с указанной точностью | до 131072 цифр до десятичной точки и до 16383 — после |
| numeric | переменный | вещественное число с указанной точностью | до 131072 цифр до десятичной точки и до 16383 — после |
| real | 4 байта | вещественное число с переменной точностью | точность в пределах 6 десятичных цифр |
| double precision | 8 байт | вещественное число с переменной точностью | точность в пределах 15 десятичных цифр |
| smallserial | 2 байта | небольшое целое с автоувеличением | 1 ... 32767 |
| serial | 4 байта | целое с автоувеличением | 1 ... 2147483647 |
| bigserial | 8 байт | большое целое с автоувеличением | 1 ... 9223372036854775807 |

3.4.1.1. serial

Особо следует обратить внимание на smallserial, serial и bigserial. Они не являются настоящими типами, а являются указанием на то, что для этих столбцов выполняется присвоение следующего уникального целочисленного значения (подобное свойству AUTO\_INCREMENT в некоторых СУБД).

Так как типы smallserial, serial и bigserial реализованы через последовательности, в числовом ряду значений столбца могут образовываться пропуски (или «дыры»), даже если никакие строки не удалялись. Значение, выделенное из последовательности, считается «задействованным», даже если строку с этим значением не удалось вставить в таблицу. Это может произойти, например, при откате транзакции, добавляющей данные.

3.4.1.2. identity

Альтернативой автоинкрементных столбцов smallserial, serial и bigserial являются столбцы, заданные как identity:

CREATE TABLE people (

id bigint **GENERATED ALWAYS AS IDENTITY**,

...,

);

Или

CREATE TABLE people (

id bigint **GENERATED BY DEFAULT IDENTITY**,

...,

);

В современных версиях PostgreSQL использование serial не рекомендуется, предпочтительно использовать IDENTITY.

3.4.1.3. numeric

Тип numeric может хранить числа с большим количеством цифр. Обычно используется тип numeric для хранения чисел, требующих точности, таких как денежные суммы или количества.

Вот синтаксис объявления столбца с numeric типом:

column\_name numeric(precision, scale)

В этом синтаксисе:

* precision – общее количество цифр,
* scale – количество цифр в дробной части.

Тип numeric может содержать значение, содержащее до 131072 цифр до десятичной запятой и 16383 цифр после десятичной запятой. scale типа numeric может быть нулевым, положительным или отрицательным[[1]](#footnote-1).

Ниже объявляется столбец price (цена) с числовым типом, который может хранить итоговые числа с 7 цифрами, 5 цифрами до десятичной точки и 2 цифрами после десятичной точки:

price numeric(7,2)

В следующем примере показано, как объявить столбец числового типа с нулевым scale:

quantity numeric(5, 0)

Это эквивалентно следующему объявлению, в котором явно не указана scale:

quantity numeric(5)

3.4.1.4. decimal и dec

В PostgreSQL типы numeric и decimal являются синонимами, поэтому они взаимозаменяемы:

decimal(p, s)

Если вы предпочитаете более короткое имя, вы можете использовать имя dec, поскольку dec и decimal — это один и тот же тип:

dec(p, s)

Если точность не требуется, не следует использовать этот numeric, поскольку вычисления со numeric значениями обычно выполняются медленнее, чем с целыми числами, числами с плавающей точкой и числами двойной точности.

Рассмотрим пример использования numeric. Если вы сохраняете значение со scale большим, чем объявленный scale в numeric колонке, то PostgreSQL округлит значение до указанного количества дробных цифр. Создадим таблицу products:

CREATE TABLE products

(

id serial PRIMARY KEY,

name varchar(100) NOT NULL,

price numeric(5,2)

);

Вставим некоторые продукты, цены которых превышают точность, заявленный в столбце price:

INSERT INTO products (name, price)

VALUES ('Наушники',500.215),

('Micro SD',500.214);

Следующий запрос возвращает все строки таблицы products (рис. 3.2):

SELECT \* FROM products;

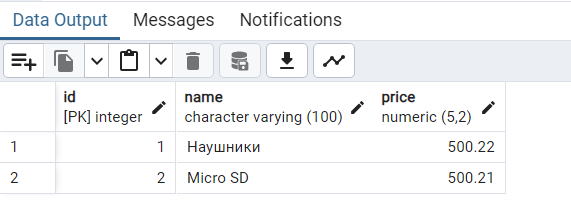


Рисунок 3.2 – Результат

Как видим, произошло округление: 500.215 округлили до 500.22, а 500.214 – до 500.21.

### 3.4.2. Символьные типы

Символьные типы приведены в таблице 3.3:

**Таблица 3.3. Символьные типы**

| Тип данных | Описание |
| --- | --- |
| character varying(*n*), varchar(*n*) | строка ограниченной переменной длины |
| character(*n*), char(*n*) | строка фиксированной длины, дополненная пробелами |
| text | строка неограниченной переменной длины |

Константы символьных типов в SQL-командах заключаются в одинарные кавычки:

SELECT \* FROM bookings.airports

WHERE airport\_name = 'Абакан';

Если кавычек в тексте много, то для лучшей читаемости можно использовать два символа доллара:

SELECT \* FROM bookings.airports

WHERE airport\_name = $$Абакан$$;

Оба char(n) и varchar(n) могут хранить до n символов. Если вы попытаетесь сохранить строку, которая содержит более n символов, PostgreSQL выдаст ошибку. Однако есть одно исключение: если все лишние символы являются пробелами, PostgreSQL обрезает пробелы до максимальной длины (n) и сохраняет обрезанные символы.

char(n) отличается от varchar(n) тем, что char(n) хранит строки фиксированной длины и, если количество символов меньше n, то строка дополняется пробелами, делая длину равную n. Дополненные пробелами значения хранятся в таблице и отображаются в этом виде. Однако эти дополнительные пробелы считаются семантически не важными. При сравнении двух значений типа char конечные пробелы игнорируются, и они будут удалены при преобразовании значений типа char в любые другие строковые типы.

Тип text данных может хранить строку неограниченной длины. Если не указать целое число n для varchar типа данных, он ведет себя как text тип данных. Производительность varchar(без размера n) и text одинакова.

Преимущество указания спецификатора длины для varchar типа данных заключается в том, что PostgreSQL выдаст ошибку, если вы попытаетесь вставить в varchar(n) столбец строку, содержащую более n символов.

В отличие от varchar, оператор character или char без указания длины (n) аналогичен оператору character(1) или char(1).

В отличие от других систем баз данных, в PostgreSQL нет разницы в производительности между тремя типами символов.

В большинстве случаев следует использовать text или varchar, а использовать varchar(n) только тогда, когда вы хотите, чтобы PostgreSQL проверил длину.

Рассмотрим пример, чтобы понять, как работают типы данных, char и varchar, text.

Сначала создайте новую таблицу с именем char\_test:

CREATE TABLE char\_test

(

id serial PRIMARY KEY,

ch char(1),

vc varchar(10),

t text

);

Затем вставим новую строку в таблицу char\_test:

INSERT INTO char\_test (ch, vc, t)

VALUES

(

'Символ',

'Тестируем varchar',

'Сохраняем очень длинную строку в поле text'

);

PostgreSQL выдал ошибку:

ERROR: value too long for type character(1)

SQL state: 22001

Это связано с тем, что тип данных столбца ch — char(1), и мы попытались вставить в этот столбец строку из трех символов.

Давайте исправим это:

INSERT INTO char\_test (ch, vc, t)

VALUES

(

'С',

'Тестируем varchar',

'Сохраняем очень длинную строку в поле text'

);

PostgreSQL выдает другую ошибку:

ERROR: value too long for type character varying(10)

SQL state: 22001

Это связано с тем, что мы попытались вставить строку, содержащую более 10 символов, в столбец vc с указанным varchar(10) типом данных.

Следующий оператор успешно вставляет новую строку в таблицу[[2]](#footnote-2).

INSERT INTO char\_test (ch, vc, t)

VALUES

(

'С',

'varchar',

'Сохраняем очень длинную строку в поле text'

) RETURNING \*;

Вывод (рис. 3.3):

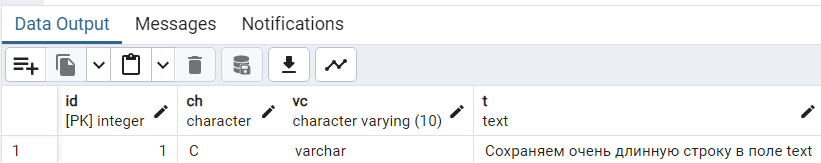


Рисунок 3.3 – Результат

### 3.4.3. Тип «дата/время»

**Таблица 3.4. Типы дата/время**

| Тип данных | Размер | Описание | Наименьшее значение | Наибольшее значение | Точность |
| --- | --- | --- | --- | --- | --- |
| timestamp [(***p***)][without time zone] | 8 байт | дата и время (без часового пояса) | 4713 до н. э. | 294276 н. э. | 1 микросекунда |
| timestamp [ (***p***) ] with time zone | 8 байт | дата и время (с часовым поясом) | 4713 до н. э. | 294276 н. э. | 1 микросекунда |
| date | 4 байта | дата (без времени суток) | 4713 до н. э. | 5874897 н. э. | 1 день |
| time [(***p***)] [ without time zone] | 8 байт | время суток (без даты) | 00:00:00 | 24:00:00 | 1 микросекунда |
| time [ (***p***) ] with time zone | 12 байт | время дня (без даты), с часовым поясом | 00:00:00+1559 | 24:00:00-1559 | 1 микросекунда |
| interval [ ***поля*** ] [ (***p***) ] | 16 байт | временной интервал | -178000000 лет | 178000000 лет | 1 микросекунда |

Типы time, timestamp и interval принимают необязательное значение точности p, определяющее сколько знаков после запятой должно сохраняться в секундах. По умолчанию точность не ограничивается. Допустимые значения p лежат в интервале от 0 до 6.

3.4.3.1. Тип данных date

PostgreSQL предлагает date тип данных, позволяющий хранить даты.

PostgreSQL использует 4 байта для хранения значения даты. Наименьшее и наибольшее значения типа date — 4713 до н.э. и 5874897 н.э. соответственно. При сохранении значения даты PostgreSQL используют формат yyyy-mm-dd, например, 2024-12-31.

**Получить текущую дату**

Чтобы получить текущую дату и время, используйте встроенную функцию now():

SELECT now();

Результат (рис. 3.4):

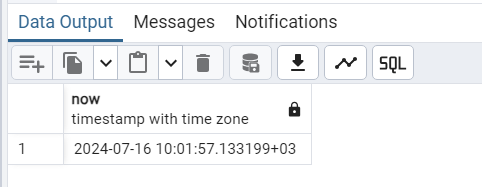


Рисунок 3.4 – Результат

Чтобы получить только часть даты (без части времени), используйте оператор приведения (::) для приведения значения datetime к значению date:

SELECT now()::date;

Результат (рис. 3.5):

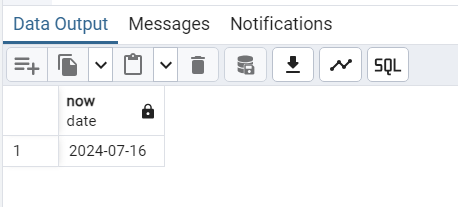


Рисунок 3.5 – Результат

Быстрый способ получить текущую дату — использовать функцию CURRENT\_DATE:

SELECT CURRENT\_DATE;

Результат (рис. 3.6):

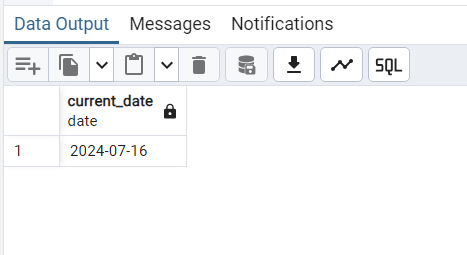


Рисунок 3.6 – Результат

**Извлечь год, месяц и день из даты**

Чтобы получить год, квартал, месяц, неделю и день из значения даты, используйте функцию extract().

В следующем выражении извлекаются год, месяц и день из текущей даты:

SELECT now(),

extract(YEAR FROM now()) AS year,

extract(MONTH FROM now()) AS month,

extract(DAY FROM now()) AS day;

Результат (рис. 3.7):

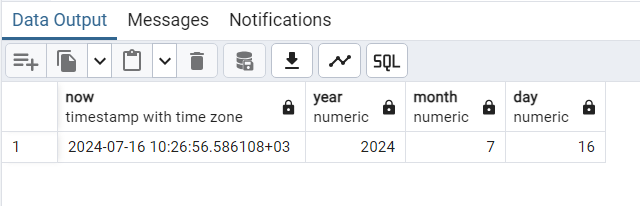


Рисунок 3.7 – Результат

Функции extract позволяет извлечь квартал, декаду, день недели, часы и т.д.

3.4.3.2. Тип данных time

PostgreSQL предоставляет тип данных time, позволяющий хранить время в базе данных.

Вот синтаксис объявления столбца с типом time данных:

column\_name time(precision);

В этом синтаксисе precision указывает точность в десятых долях секунд для значения времени, которая находится в диапазоне от 1 до 6 знаков.

Тип time требует 8 байтов, а его допустимый диапазон — от 00:00:00 до 24:00:00.

Ниже приведены общие форматы значений time:

HH:MI

HH:MI:SS

HHMISS

Чтобы использовать значение времени с precision, можно использовать следующие форматы:

MI:SS.pppppp

HH:MI:SS.pppppp

HHMISS.pppppp

В этом синтаксисе p указывает точность.

На практике часто используется тип данных time для столбцов, которые хранят только время суток, например время события или смены. Например:

Создадим таблицу рабочих смен на некотором предприятии, назвав ее test\_time с помощью оператора CREATE TABLE:

CREATE TABLE time\_test

(

id serial PRIMARY KEY,

shift\_num int NOT NULL,

start\_at time NOT NULL,

end\_at time NOT NULL

);

Вставим несколько строк в таблицу:

INSERT INTO time\_test(shift\_num, start\_at, end\_at)

VALUES(1, '08:00:00', '12:00:00'),

(2, '13:00:00', '17:00:00'),

(3, '18:00:00', '22:00:00');

Запросим данные:

SELECT \* FROM time\_test;

Результат (рис. 3.8):

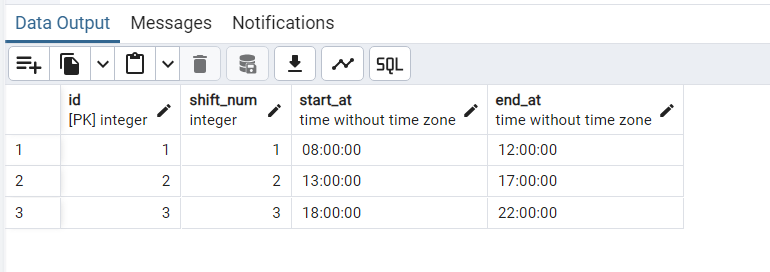


Рисунок 3.8 – Результат

**Время с часовым поясом**

Помимо типа данных time, PostgreSQL предоставляет тип данных время с часовым поясом time with time zone (или timetz), позволяющий хранить и управлять временем суток с учетом часового пояса.

Следующий оператор иллюстрирует, как объявить столбец, тип данных которого время с часовым поясом:

column\_name time with time zone

или

column\_name timetz

Размер хранилища типа данных time with time zone составляет 12 байт, что позволяет хранить значение времени в часовом поясе от 00:00:00+1459 до 24:00:00-1459.

Ниже приведен пример time with time zone:

10:18:38.134767+03

При работе с часовым поясом рекомендуется использовать timestamp вместо типа time with time zone. Это связано с тем, что часовой пояс имеет очень мало смысла, если он не связан и с датой и со временем.

**Получение текущего времени**

Чтобы получить текущее время с учетом часового пояса, используйте функцию CURRENT\_TIME:

SELECT CURRENT\_TIME;

Результат (рис. 3.9):

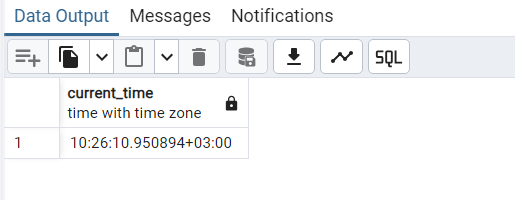


Рисунок 3.9 – Результат

Чтобы получить текущее время с определенной точностью, используйте функцию CURRENT\_TIME(precision):

SELECT CURRENT\_TIME(3);

Результат (рис. 3.10):

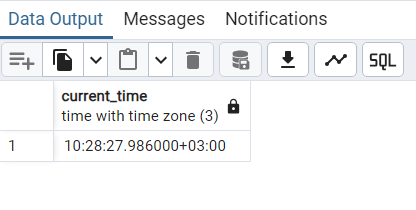


Рисунок 3.10 – Результат

Обратите внимание, что без указания точности функция CURRENT\_TIME возвращает значение времени с максимально возможной точностью.

Чтобы узнать местное время, используйте функцию LOCALTIME:

SELECT LOCALTIME;

Результат (рис. 3.11):

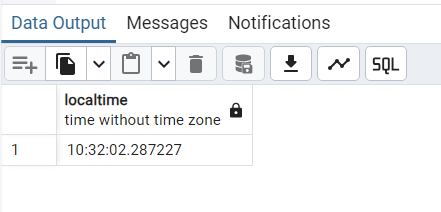


Рисунок 3.11 – Результат

Аналогично, чтобы получить местное время с определенной точностью, используйте функцию LOCALTIME(precision).

**Извлечение часов, минут и секунд из значения времени**

Чтобы извлечь часы, минуты и секунды из значения времени, используйте функцию extract следующим образом:

extract(part FROM time\_value);

part может быть часом, минутой, секундой или миллисекундой. Например:

SELECT

LOCALTIME,

extract (HOUR FROM LOCALTIME) as hour,

extract (MINUTE FROM LOCALTIME) as minute,

extract (SECOND FROM LOCALTIME) as second,

extract (milliseconds FROM LOCALTIME) as milliseconds;

Результат (рис. 3.12):

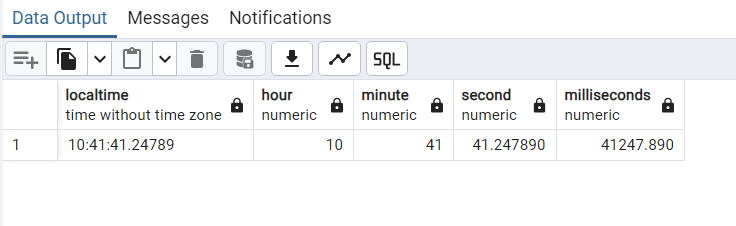


Рисунок 3.12– Результат

3.4.3.3. Тип данных timestamp

PostgreSQL предоставляет вам два типа временных данных для обработки даты и времени:

* timestamp – дата и время без указания часового пояса;
* timestamptz – дата и время с часовым поясом.

Тип timestamp данных позволяет хранить как дату, так и время. Однако он не содержит данных о часовом поясе. Это означает, что при изменении часового пояса сервера базы данных значение timestamp, хранящееся в базе данных, не изменится автоматически.

Тип timestamptz данных — это дата и время с часовым поясом.

Обратите внимание, что timestamp и timestamptz используют 8 байт для хранения значений в базе данных.

Рассмотрим пример использования timestamp и timestamptz.

Создадим таблицу, состоящую из обоих столбцов timestamp и timestamptz.

CREATE TABLE timestamp\_test

(

ts timestamp,

tstz timestamptz

);

Узнаем текущий часовой пояс с помощью команды:

SHOW TIMEZONE;

Результат (рис. 3.13):

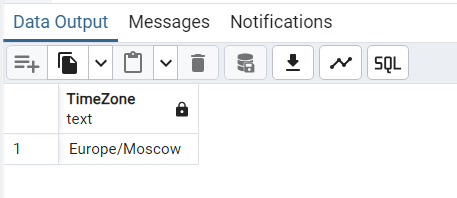


Рисунок 3.13 – Результат

Вставим в таблицу timstamp\_test строку, содержащую дату и время с часовым поясом Свердловской области:

INSERT INTO timestamp\_test (ts, tstz)

VALUES('2024-07-18 11:10:30+05','2024-07-18 11:10:30+05');

После этого запросим данные из столбцов timestamp и timestamptz:

SELECT ts, tstz

FROM timestamp\_test;

Результат (рис. 3.14):

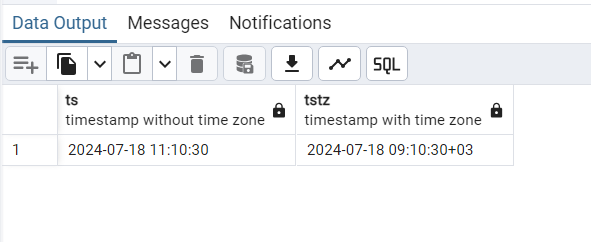


Рисунок 3.14 – Результат

Значение в столбце timestamp не изменяется, тогда как значение в столбце timestamptz корректируется в соответствии с часовым поясом сервера 'Europe/Moscow'.

**Получение текущей даты и времени**

Чтобы получить текущую временную метку, используйте функцию now() следующим образом:

SELECT now();

Результат (рис. 3.15):

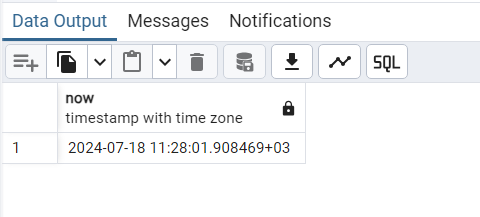


Рисунок 3.15 – Результат

В качестве альтернативы вы можете использовать функцию CURRENT\_TIMESTAMP:

SELECT CURRENT\_TIMESTAMP;

### 3.4.4. Логический тип

Логический тип приведен в таблице 3.5:

**Таблица 3.5. Логический тип**

| Тип данных | Размер | Описание |
| --- | --- | --- |
| boolean | 1 байт | состояние: истина или ложь |

Логические константы могут представляться в SQL-запросах следующими ключевыми словами SQL: TRUE, FALSE и NULL.

Функция ввода данных типа boolean воспринимает следующие строковые представления состояния «true»:

true

yes

on

1

и следующие представления состояния «false»:

false

no

off

0

### 3.4.5. Массивы

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

Чтобы проиллюстрировать использование массивов, мы создадим такую таблицу:

CREATE TABLE pilot

(

name text,

schedule integer[]

);

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

## 3.5. Операции в языке SQL

В таблицах 3.6, 3.7 и 3.8 приведены операции для языка SQL:

**Таблица 3.6. Арифметические операции (здесь a=10 и b=20)**

|  |  |  |
| --- | --- | --- |
| Операция | Описание | Пример |
| + (сложение) | Сложение значений | a + b = 30 |
| — (вычитание) | Вычитание правого операнда из левого | b - a = 10 |
| \* (умножение) | Умножение значений | a \* b = 200 |
| / (деление) | Деление левого операнда на правый | b / a = 2 |
| % (деление по модулю) | Деление левого операнда на правый с остатком (возвращается остаток) | b % a = 0 |

**Таблица 3.7. Операции сравнения (здесь a=10 и b=20)**

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Описание | | Пример |
| = | Определяет равенство значений | | a = b -> false |
| != | Определяет НЕравенство значений | | a != b -> true |
| <> | Определяет НЕравенство значений | | a <> b -> true |
| > | Значение левого операнда больше значения правого операнда? | | a > b -> false |
| < | Значение левого операнда меньше значения правого операнда? | | a < b -> true |
| >= | Значение левого операнда больше или равно значению правого операнда? | | a >= b -> false |
| <= | Значение левого операнда меньше или равно значению правого операнда? | | a <= b -> true |
| !< | Значение левого операнда НЕ меньше значения правого операнда? | | a !< b -> false |
| !> | Значение левого операнда НЕ больше значения правого операнда? | | a !> b -> true |
| **Таблица 3.8. Логические операции** | | | |
| Операция | | Описание | |
| ALL | | TRUE, если булево выражение истинно для всех строк, возвращаемых подзапросом | |
| AND | | TRUE, если оба булевых выражения истинны | |
| ANY | | ANY и его синоним SOME возвращает TRUE, если булево выражение истинно хотя бы для одной строки подзапроса (множества). | |
| BETWEEN | | TRUE, если операнд попадает в заданный интервал | |
| EXISTS | | TRUE, если подзапрос возвращает хотя бы одну строку | |
| IN | | TRUE, если операнд равен хотя бы одному выражению из списка или строке из результата подзапроса | |
| LIKE | | TRUE, если операнд удовлетворяет шаблону | |
| NOT | | Заменяет булево значение на противоположное | |
| OR | | TRUE, если хотя бы одно из двух булевых выражений истинно | |
| IS NULL | | TRUE, если поле не заполнено (содержит NULL) | |

## 3.5. Термины

***База данных*** (БД) ‑ организованная в соответствии с определёнными правилами и поддерживаемая в памяти компьютера совокупность данных, характеризующая актуальное состояние некоторой предметной области и используемая для удовлетворения информационных потребностей пользователей.

***Реляционная база данных*** представляет собой набор взаимосвязанных двухмерных таблиц. Эта модель предложена сотрудником фирмы “IBM” Эдгаром Коддом в 1970 году.

Таблица реляционной БД соответствует одной сущности предметной области и состоит из фиксированного числа полей, собранных в записи, каждая запись соответствует экземпляру сущности.

Большинство современных СУБД поддерживают реляционные БД и, следовательно, являются реляционными.

Все реляционные СУБД поддерживают язык SQL.

***Сущность (****объект****)*** - предмет, человек или событие, о котором собирается и хранится информация в БД.

***Таблица*** - совокупность записей с фиксированным числом полей. Таблица содержит данные о всех экземплярах сущности.

***Запись*** – данные об одном конкретном экземпляре сущности - предмете, человеке или событии. Запись состоит из полей.

Пример. Одна запись таблицы «Сотрудники» составляет сущность «Сотрудник».

***Поле записи (****атрибут, столбец таблицы****)*** - отдельная характеристика (свойство) сущности.

***Домен*** - область значений одного или нескольких атрибутов.

Ниже в таблице 3.9 приведено соответствие терминов в различных областях знаний.

**Таблица 3.9. Терминов**

|  |  |  |
| --- | --- | --- |
| База данных | Реляционная алгебра | ГОСТ (EXCEL) |
| Таблица | Отношение | Таблица |
| Запись | Кортеж | Строка |
| Поле | Атрибут | Столбец |

## 3.6. Примеры оператора SELECT

Рассмотрим несколько примеров использования команды SELECT. В качестве БД для экспериментов будем использовать demo. Запустим pgAdmin, подсоединимся к серверу и выделим БД demo. Все приведенные ниже примеры можно выполнять в терминале PSQL Tool, но мы воспользуемся другим инструментом Query Tool, значок которого расположен слева на панели инструментов (рис. 3.16).

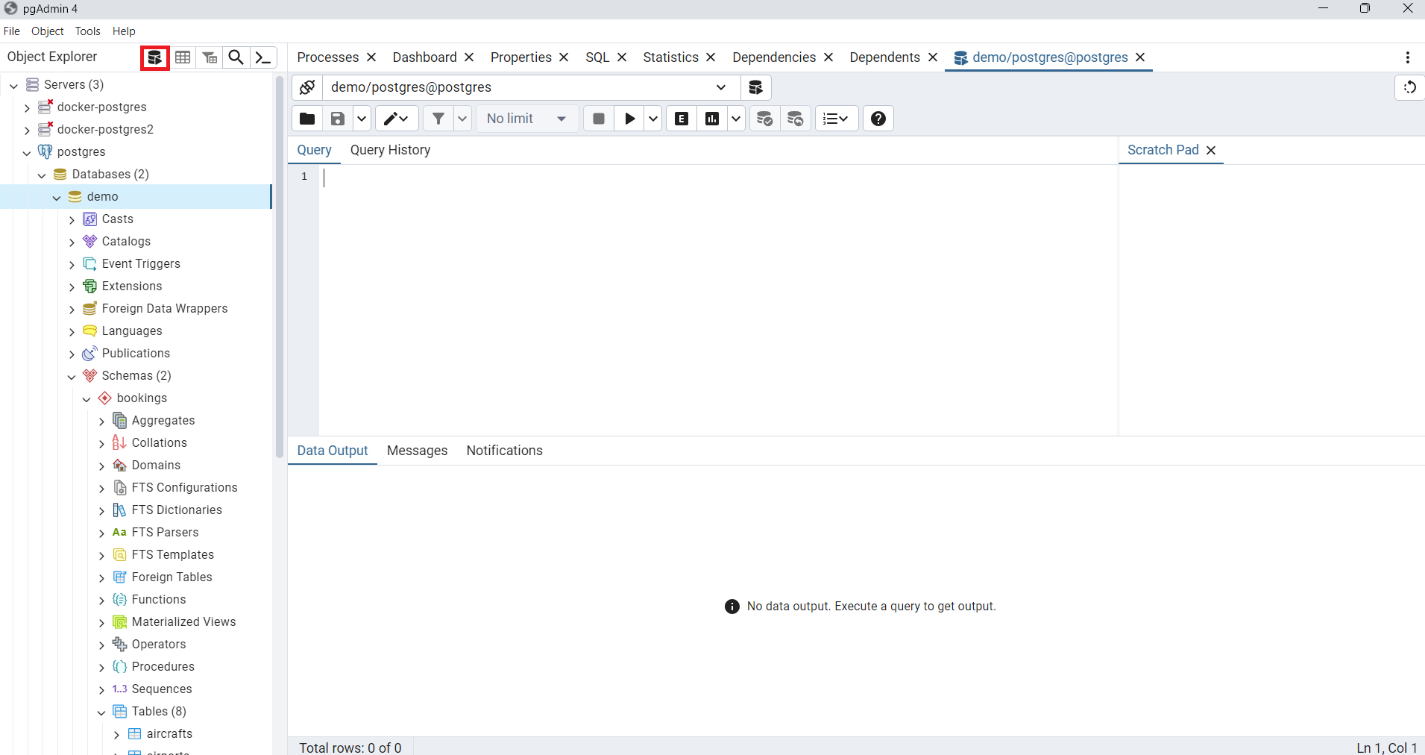


Рисунок 3.16 – Панель инструментов

Перед тем, как перейти к примерам, рассмотрим, что такое схема в БД PostgreSQL. Схемы — это пространства имен для таблиц, предназначенные для логического разделения объектов и предотвращения конфликтов имен. Проще говоря, это некие контейнеры, в которые мы можем помещать таблицы.

Существуют схемы по умолчанию, созданные PostgreSQL. Одна из таких схем называется public. Именно в нее будут попадать все созданные нами таблицы по умолчанию, если мы не укажем какую-то другую схему. Если у нас есть две таблицы с одинаковыми именами, но они находятся в разных схемах, то это нормально, конфликта имен не будет.

В базе demo находится схема bookings (бронирование). Таблицы созданы в этой схеме. Для доступа к любой таблице или другому объекту схемы необходимо указывать схему:

SELECT \* FROM bookings.airports;

Можно задать схему, с которой необходимо начинать поиск таблицы, тем самым сократить написание команд:

SET search\_path = bookings;

SELECT \* FROM airports;

### 3.6.1. Выбор всех полей

Самый простой способ вернуть все поля и просмотреть все содержимое таблицы – использовать SELECT \*:

SELECT \* FROM bookings.airports;

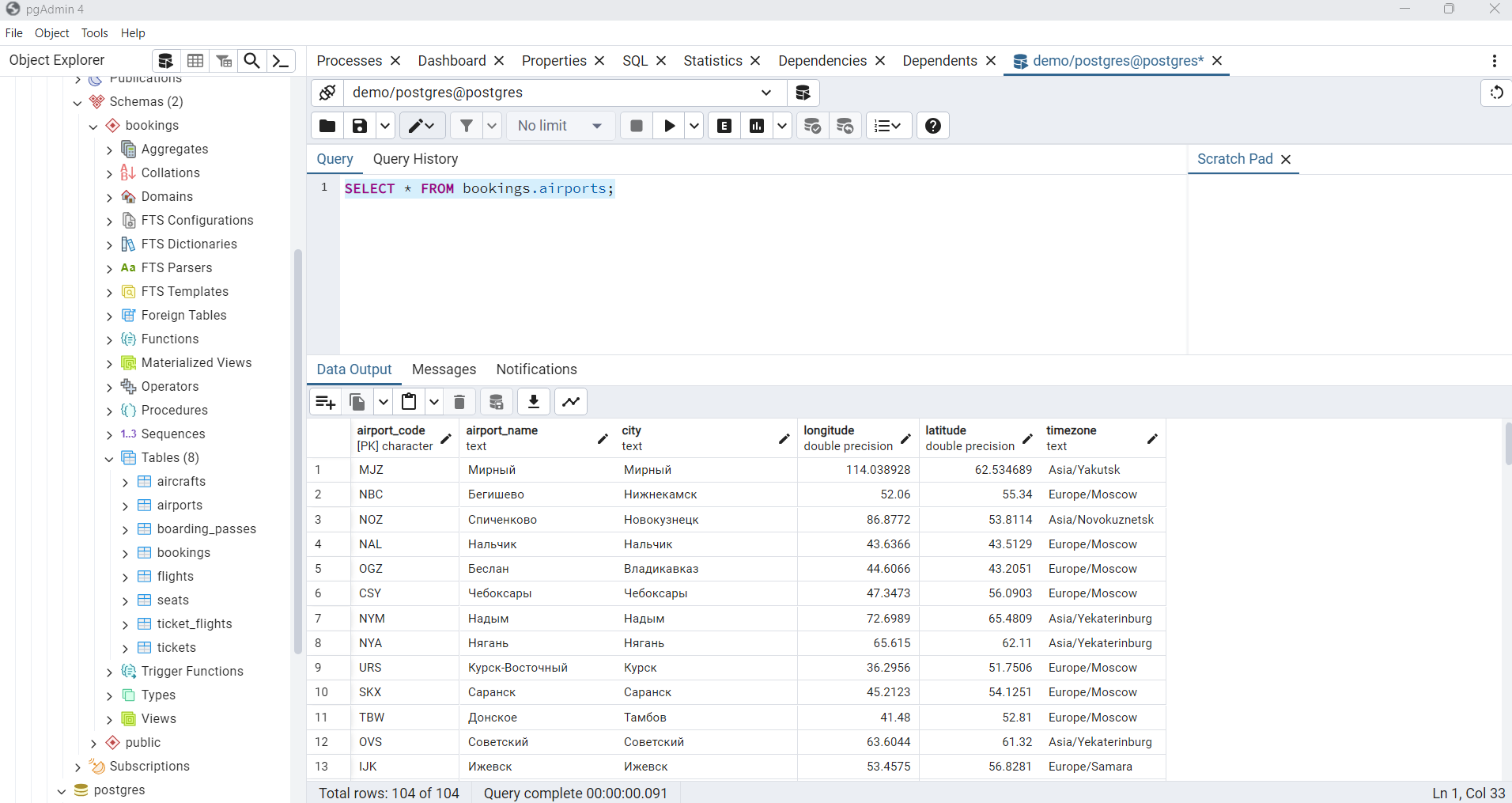


Рисунок 3.17 – Результат запроса

В приведенном выше примере выходные данные показывают все поля, содержащиеся в таблице airports.

### 3.6.2. Выбор отдельных полей из одной таблицы

Оператор SELECT позволяет возвращать отдельные поля из таблицы (рис. 30):

SELECT airport\_name, city, timezone

FROM bookings.airports

ORDER BY airport\_name;

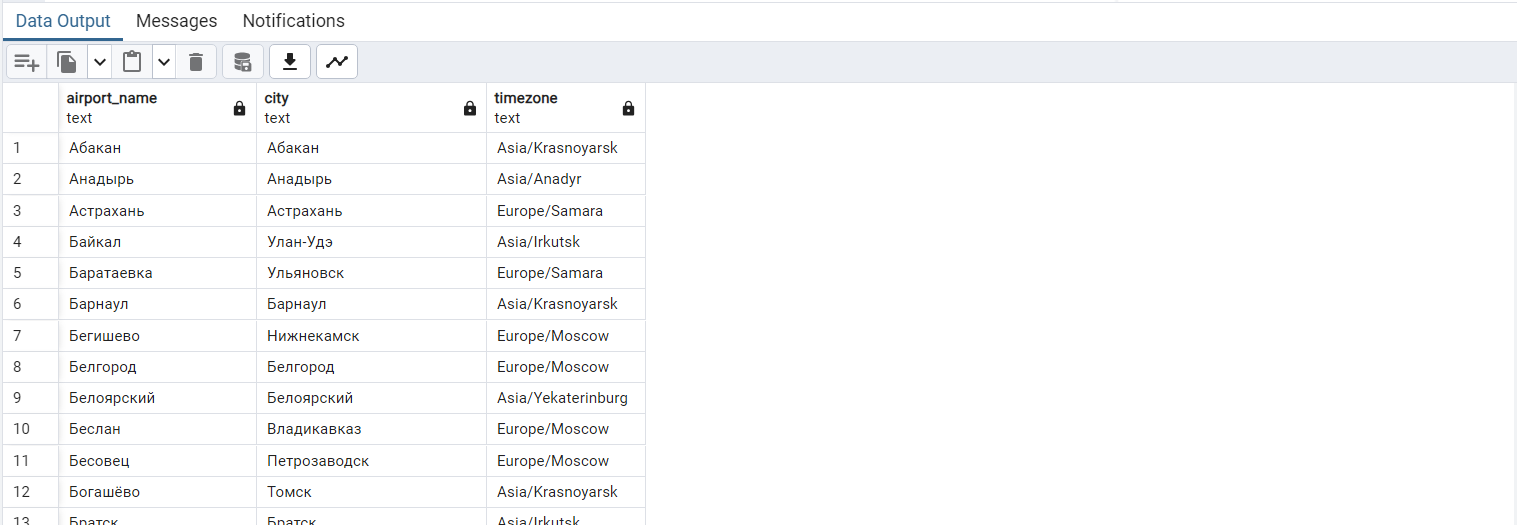


Рисунок 3.18 – Результат запроса

В приведенном выше примере указаны только название аэропорта, города и часовой пояс аэропорта. Вывод упорядочивает результаты по названию аэропорта в порядке возрастания (по умолчанию предполагается ASC).

### 3.6.3. Фильтрация результатов по условию

Ключевое слово WHERE в SQL используется для фильтрации данных, позволяя выбирать только те строки, которые соответствуют определенным условиям. Ниже приведены выражения, которые могут идти после ключевого слова WHERE.

Операторы сравнения.

Выведем перелеты, стоимость (amount) которых больше 200000:

SELECT \*

FROM bookings.ticket\_flights

WHERE 200000 < amount;

Логические операторы

Отобразим перелеты, стоимость которых больше 70000 и не являются бизнес-классом:

SELECT \*

FROM bookings.ticket\_flights

WHERE 74000 < amount AND fare\_conditions != 'Business';

Диапазон

Для определения диапазона значений используется BEETWIN:

SELECT \*

FROM bookings.ticket\_flights

WHERE amount BETWEEN 30000 AND 35000 AND fare\_conditions = 'Business';

Фильтрация по множеству значений

Выберем из таблицы рейсов те, которые выполняются определенными типами самолетов:

SELECT \*

FROM bookings.flights

WHERE aircraft\_code IN ('SU9', 'CN1', 'CR2');

Сравнение по шаблону

Оператор поиска LIKE использует шаблоны, в которых используются символы подчеркивания «\_» и проценты «%»:

* символ подчеркивания «\_» – соответствует любому одному символу,
* символ проценты «%» – соответствует любому количеству символов.

Например, выведем те самолеты, модель которых начинается с «Airbus»:

SELECT \*

FROM bookings.aircrafts

WHERE model LIKE 'Airbus%';

Результат (рис. 3.19):

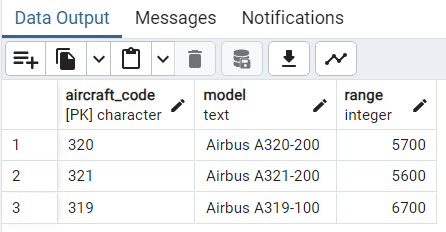


Рисунок 3.19 – Результат запроса

IS NULL, IS NOT NULL

NULL — это специальный маркер, который представляет отсутствующую информацию или несуществующее значение, а не пустую строку или ноль. Таблица в Postgres может содержать, а может и не содержать несколько NULL записей. Оператор IS NULL в Postgres позволяет фильтровать данные и манипулировать ими в зависимости от того, содержит ли данный столбец значение NULL или нет.

Отобразим те рейсы, которые не вылетели:

SELECT \*

FROM bookings.flights

WHERE actual\_departure IS NULL;

Сортировка

Если присутствует предложение ORDER BY, то возвращаемые строки сортируются в указанном порядке. В отсутствие ORDER BY строки возвращаются в том порядке, в каком системе будет проще их выдать. По умолчанию данные сортируются по возрастанию (ASC), однако с помощью оператора DESC можно задать сортировку по убыванию.

Если необходимо отсортировать сразу по нескольким столбцам, то все они перечисляются через запятую после оператора ORDER BY:

SELECT col1, col2, …

FROM table

ORDER BY col1 ASC, col2 DESC;

В общем сортировка определяется следующим образом:

SELECT ***список\_выборки***

FROM ***табличное\_выражение***

ORDER BY ***выражение\_сортировки1*** [ASC | DESC] [NULLS {FIRST | LAST}]

[, ***выражение\_сортировки2*** [ASC | DESC] [NULLS {FIRST | LAST}] ...]

Для определения места значений NULL в отсортированном списке можно использовать указания NULLS FIRST и NULLS LAST, которые помещают значения NULL соответственно до или после значений не NULL. По умолчанию значения NULL считаются больше любых других, то есть подразумевается NULLS FIRST для порядка DESC и NULLS LAST в противном случае.

### 3.6.4. Группировка строк

Строки, полученные командой SELECT, прошедшие фильтр WHERE, можно сгруппировать с помощью предложения GROUP BY, а затем оставить в результате только нужные группы строк, используя предложение HAVING.

SELECT ***список\_выборки***

FROM ...

[WHERE ...]

GROUP BY ***группирующий\_столбец*** [, ***группирующий\_столбец***] ...

Предложение GROUP BY группирует строки таблицы, объединяя их в одну группу при совпадении значений во всех перечисленных столбцах. Порядок, в котором указаны столбцы, не имеет значения. В результате наборы строк с одинаковыми значениями преобразуются в отдельные строки, представляющие все строки группы. Это может быть полезно для устранения избыточности выходных данных и/или для вычисления агрегатных функций, применённых к этим группам.

При группировании столбцы, не включённые в список GROUP BY, можно использовать только в агрегатных выражениях. Например, подсчитаем сколько рейсов выполнено между различными аэропортами:

SELECT departure\_airport, arrival\_airport, count(\*)

FROM bookings.flights

WHERE status = 'Arrived'

GROUP BY departure\_airport, arrival\_airport

ORDER BY departure\_airport, arrival\_airport;

Результат (рис. 3.20):

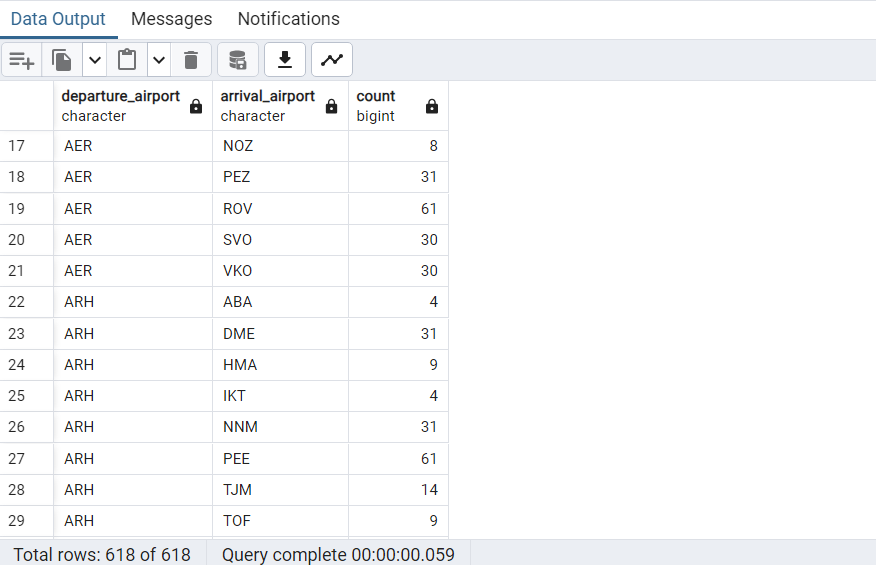


Рисунок 3.20 – Результат запроса

Здесь выполнено группирование по аэропортам вылета и прилета, т.е. строки таблицы flights, имеющие одни те же аэропорты вылета и прилета, объединяются в одну группу. Функция count(\*) подсчитывает количество строк в группе.

Для фильтрации групп следует использовать предложение HAVING. Например, из предыдущего примеры выберем те рейсы, которые отправляются из Шереметьева:

SELECT departure\_airport, arrival\_airport, count(\*)

FROM bookings.flights

WHERE status = 'Arrived'

GROUP BY departure\_airport, arrival\_airport

HAVING departure\_airport = 'SVO'

ORDER BY departure\_airport, arrival\_airport;

Результат (рис. 3.21):

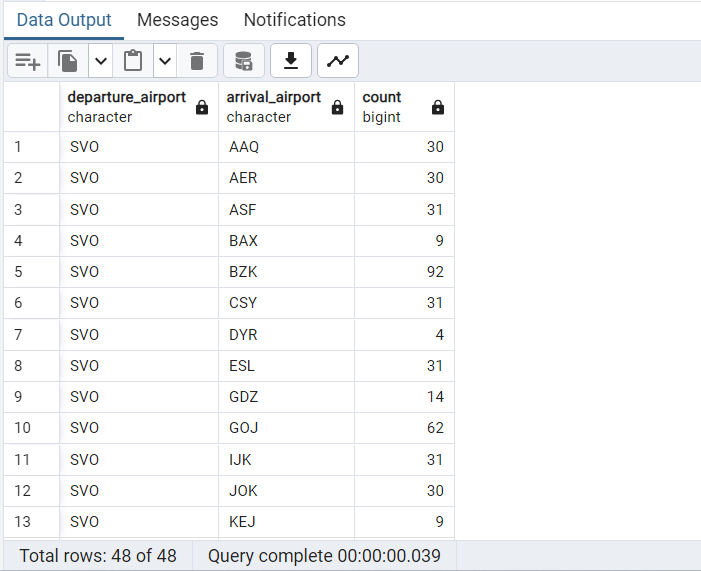


Рисунок 3.21 – Результат запроса

С группировкой используют агрегатные функции, такие как:

* count(\*) - подсчёт кол-ва строк,
* sum(поле) - сумма значений поля,
* max(поле) - максимальное значение поля,
* min(поле) - минимальное значение поля,
* avg(поле) - среднее значение поля.

Пример использования count был приведен выше.

Функция count принимает один из нескольких параметров:

* «\*» – означает то, что функция count возвращает количество всех записей в таблице;
* «поле» – функция count возвращает количество записей с непустыми значениями конкретного поля (NOT NULL);
* «DISTINCT поле» – функция count возвращает количество уникальных записей конкретного поля (только NOT NULL).

# 4. Соединение таблиц JOIN

## 4.1. Общие сведения

Соединение (JOIN) - одна из самых важных операций, выполняемых реляционными системами управления базами данных (РСУБД). РСУБД используют соединения для того, чтобы сопоставить строки одной таблицы строкам другой таблицы. Например, соединения можно использовать для отображения аэропортов вылета и прилета при отображении расписания полетов в БД demo. В таблице flights аэропорты вылета и прилета являются трехсимвольными кодами (внешние ключи), ссылающимися на таблицу аэропортов airports\_data и без отображения естественных названий аэропортов разобраться в расписании будет невозможно. Например, можно сделать так:

SELECT f.scheduled\_departure AS "Время вылета",

f.scheduled\_arrival AS "Время прилета",

ad.airport\_name::jsonb->>'ru' AS "Аэропорт вылета",  
 aa.airport\_name::jsonb->>'ru' AS "Аэропорт прилета"

FROM bookings.flights AS f

JOIN bookings.airports\_data AS ad ON ad.airport\_code = f.departure\_airport

JOIN bookings.airports\_data AS aa ON aa.airport\_code = f.arrival\_airport

ORDER BY f.scheduled\_departure;

Результат (рис. 4.1):

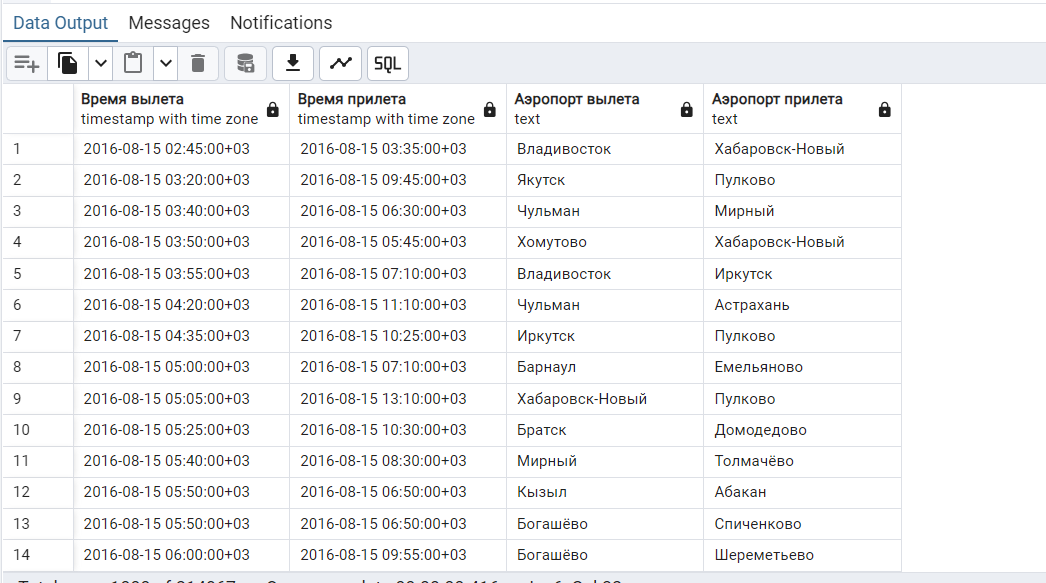


Рисунок 4.1 – Результат запроса

В БД demo название аэропортов хранится на русском и английских языках в формате jsonb, например:

{"en": "Vladivostok International Airport", "ru": "Владивосток"}

Для получения русского названия используется запись:

<поле>::jsonb->>'ru'

Больше мы не будем касаться формата jsonb, т. к. он не входит в текущий курс. Что касается остального в приведенном запросе, то оно будет разобрано ниже.

Оператор JOIN входит в раздел FROM операторов SELECT, UPDATE или DELETE. Ниже мы рассмотрим использование соединения в операторе SELECT.

Упрощенный синтаксис таких соединений следующий:

SELECT список\_столбцов

FROM таблица1

INNER JOIN | LEFT [OUTER] JOIN | RIGHT [OUTER] JOIN | FULL JOIN |  
 CROSS JOIN таблица2

ON таблица1.некий\_столбец=таблица2.некий\_столбец

В SELECT перечисляются необходимые столбцы для вывода в произвольном порядке. Выше приведены следующие типы соединения JOIN:

INNER JOIN – внутреннее соединение,

LEFT [OUTER] JOIN – левое внешнее соединение,

RIGHT [OUTER] JOIN – правое внешнее соединение,

FULL JOIN – полное соединение,

CROSS JOIN – перекрестное соединение.

## 4.2. Псевдонимы

Псевдонимы PostgreSQL — это инструменты, которые позволяют вам присваивать временные имена таблицам или столбцам в ваших запросах. Эти псевдонимы существуют только во время выполнения запроса, что делает ваш SQL-код более читаемым и эффективным.

Псевдонимы столбцов позволяют вам переименовывать столбцы в результирующем наборе вашего запроса, делая вывод более осмысленным:

SELECT f.scheduled\_departure AS "Время вылета",

f.scheduled\_arrival AS "Время прилета",

…

или

SELECT f.scheduled\_departure "Время вылета",

f.scheduled\_arrival "Время прилета",

…

Добавим в приведенный выше пример расчет длительности полета и выполним по ней сортировку:

SELECT …

f.scheduled\_arrival-f.scheduled\_departure "Время полета"

FROM bookings.flights AS f

JOIN bookings.airports\_data AS ad ON ad.airport\_code = f.departure\_airport

JOIN bookings.airports\_data AS aa ON aa.airport\_code = f.arrival\_airport

ORDER BY "Время полета";

Если в качестве псевдонима использовать правила наименования идентификаторов, т.е., например, если использовать буквы и символ подчеркивания (см. документацию), то можно обойтись без кавычек:

SELECT …

f.scheduled\_arrival-f.scheduled\_departure Время\_полета

FROM bookings.flights AS f

JOIN bookings.airports\_data AS ad ON ad.airport\_code = f.departure\_airport

JOIN bookings.airports\_data AS aa ON aa.airport\_code = f.arrival\_airport

ORDER BY Время\_полета;

Псевдонимы таблиц предоставляют способ сократить имена таблиц в ваших запросах, что особенно полезно для таблиц с длинными именами или при соединении нескольких таблиц:

SELECT **f.**scheduled\_departure AS "Время вылета",

f.scheduled\_arrival AS "Время прилета",

**ad**.airport\_name::jsonb->>'ru' AS "Аэропорт вылета",

aa.airport\_name::jsonb->>'ru' AS "Аэропорт прилета",

f.scheduled\_arrival-f.scheduled\_departure AS Время\_полета

FROM bookings.flights AS f

JOIN **bookings.airports\_data AS ad** ON **ad.**airport\_code = f.departure\_airport

JOIN bookings.airports\_data AS aa ON aa.airport\_code = f.arrival\_airport

ORDER BY Время\_полета;

Существует несколько вариантов использования псевдонима таблицы. Некоторые из них перечислены ниже:

* его можно использовать, чтобы сэкономить время нажатия клавиш и сделать ваш запрос более читаемым для таблиц с длинными именами;
* его также можно использовать при запросе данных из нескольких таблиц с одинаковыми именами столбцов;
* его можно использовать для объединения таблицы с самой собой (т.е. самосоединение).

## 4.3. Внутреннее соединение

С помощью ключевого слова INNER JOIN (или JOIN) задается вторая таблица и условие выбора строк из нее. Если условие выполняется, то формируется результирующий набор путем объединения всех строк из обеих таблиц.

Следующий оператор JOIN (по умолчанию INNER JOIN) объединяет первую таблицу flights со второй таблицей airports\_data, сопоставляя значения в столбцах departure\_airport из flights со airport\_code из airports\_data. departure\_airport является внешним ключом, ссылающимся на первичный ключ airport\_code таблицы airports\_data. Использование в соединении таблиц их первичные и вторичные ключи является распространенной практикой, хотя – необязательной. Такое применение целесообразно потому, что поиск подходящей строки во второй таблице по первичному ключу, будет выполняться быстрее, в отличии от сканирования всей таблицы.

SELECT …

FROM bookings.flights AS f

**JOIN** bookings.airports\_data AS ad ON ad.airport\_code = f.departure\_airport

…

Внутреннее соединение проверяет каждую строку в первой таблице flights на равенство значения departure\_airport со значением в столбце airport\_code каждой строки во второй таблице airports\_data. Если эти значения равны, внутреннее соединение создает новую строку, содержащую столбцы из обеих таблиц, и добавляет эту новую строку в результирующий набор. Если строка во второй таблице не найдена, то эта строка из первой таблицы игнорируется.

Рассмотрим пример. Некоторый магазин продает товар, который может быть поставлен покупателю по указанному адресу, либо покупатель сам забирает покупку. Создадим две таблицы products и address:

products

(

id\_product serial,

name text,

id\_address integer

)

address

(

id\_address serial,

address text

)

Таблица products имеет внешний ключ id\_address, ссылающийся на адрес поставки. Если значение поля id\_address «равно» NULL, то покупатель сам забрал товар.

Заполним эти таблицы:

INSERT INTO address (address)

VALUES('Адрес 1'),

('Адрес 2'),

('Адрес 3'),

('Адрес 4'),

('Адрес 5'),

('Адрес 6'),

('Адрес 7'),

('Адрес 8'),

('Адрес 9'),

('Адрес 10');

INSERT INTO products (name, id\_address)

VALUES('товар 1', 3),

('товар 2', NULL),

('товар 3', 5),

('товар 4', NULL),

('товар 5', NULL),

('товар 6', 9),

('товар 7', NULL),

('товар 8', NULL),

('товар 9', NULL),

('товар 10', 1),

('товар 11', NULL),

('товар 12', NULL),

('товар 13', NULL),

('товар 14', 2),

('товар 15', NULL),

('товар 16', NULL),

('товар 17', 3),

('товар 18', NULL),

('товар 19', NULL),

('товар 20', 9);

Распечатаем эти товары и адреса поставки:

SELECT p.\*, a.\*

FROM products AS p

JOIN address AS a ON a.id\_address = p.id\_address;

Результат запроса (рис. 4.2):

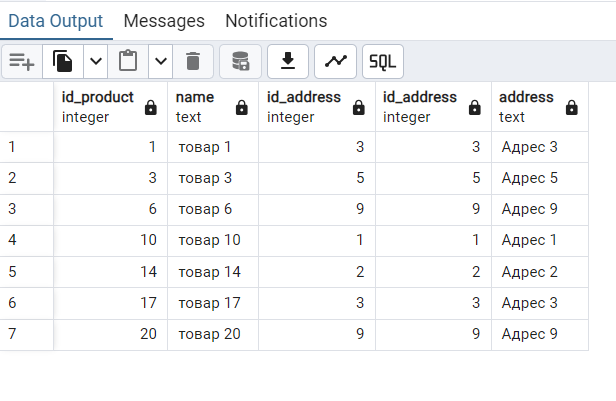


Рисунок 4.2 – Результат запроса

Как видим, выведены только те товары, у которых имеются адреса поставки.

Следующая диаграмма (рис. 4.3) иллюстрирует внутреннее соединение:

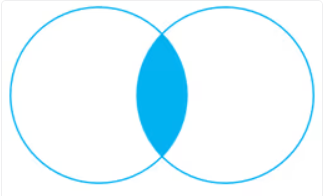


Рисунок 4.3 – INNER JOIN

## 4.4. Внешнее левое соединение

Для задания левого внешнего соединения используется ключевое слово предложение LEFT OUTER JOIN (или LEFT JOIN) для соединения первой таблицы products c таблицей address. В контексте LEFT JOIN первая таблица называется левой таблицей, а вторая таблица называется правой таблицей. Выполним предыдущий запрос, поменяв JOIN на LEFT JOIN:

SELECT p.\*, a.\*

FROM products AS p

LEFT JOIN address AS a ON a.id\_address = p.id\_address

Результат запроса (рис. 4.3):

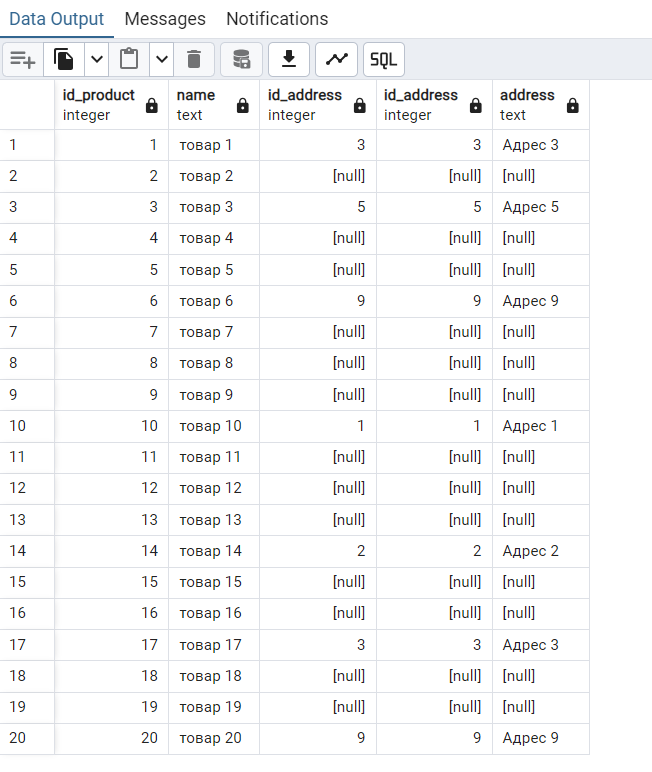


Рисунок 4.3 – Результат запроса

Левое соединение начинает выборку данных из левой таблицы. Оно сравнивает значения в столбце id\_address со значениями в столбце id\_address в таблице address. Если эти значения равны, левое соединение создает новую строку, содержащую столбцы обеих таблиц, и добавляет эту новую строку в набор результатов. В случае, если значения не равны, левое соединение также создает новую строку, содержащую столбцы из обеих таблиц, и добавляет ее в набор результатов. Однако оно заполняет столбцы правой таблицы address значениями null (рис. 4.3).

Следующая диаграмма (рис 4.4) иллюстрирует левое соединение:

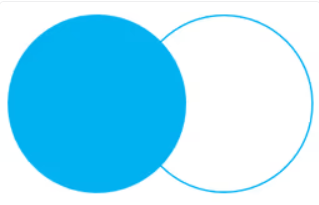


Рисунок 4.4 – LEFT OUTER JOIN

Чтобы выбрать строки из левой таблицы, которые не имеют соответствующих строк в правой таблице, используйте левое соединение с ключевым словом WHERE. Например:

SELECT p.\*, a.\*

FROM products AS p

LEFT JOIN address AS a ON a.id\_address = p.id\_address

WHERE a.id\_address IS NULL;

Результат запроса (рис. 4.5):

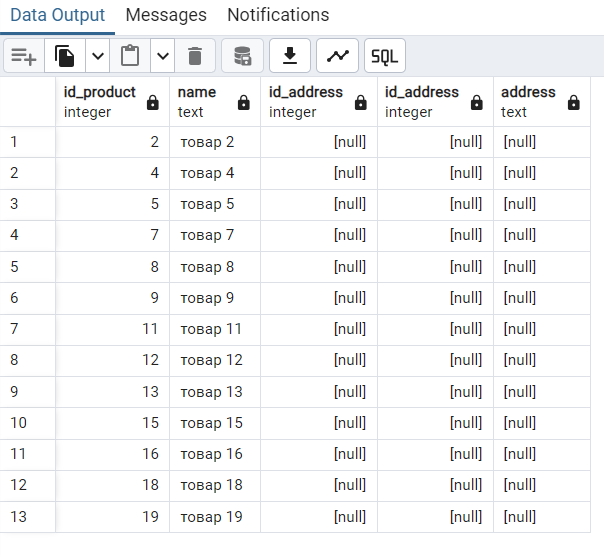


Рисунок 4.5 – Результат запроса

Следующая диаграмма (рис. 4.6) иллюстрирует левое соединение, возвращающее строки из левой таблицы, которым не соответствуют строки из правой таблицы:

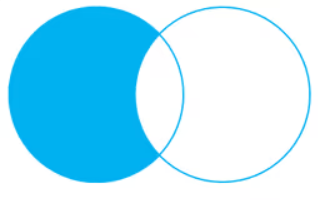


Рисунок 4.6 – LEFT OUTER JOIN

только строки из левой таблицы

## 4.5. Внешнее правое соединение

Правое соединение — это обратная версия левого соединения. Правое соединение начинает выборку данных из правой таблицы. Оно сравнивает значение в столбце id\_address каждой строки в правой таблице address со значением в столбце id\_address каждой строки в таблице products. Если эти значения равны, правое соединение создает новую строку, содержащую столбцы из обеих таблиц. В случае, если эти значения не равны, правое соединение также создает новую строку, содержащую столбцы из обеих таблиц. Однако оно заполняет столбцы в левой таблице значениями NULL.

Следующий оператор использует правое соединение для соединения таблицы products с таблицей address:

SELECT p.\*, a.\*

FROM products AS p

RIGHT JOIN address AS a ON a.id\_address = p.id\_address;

Результат запроса (рис. 4.7):

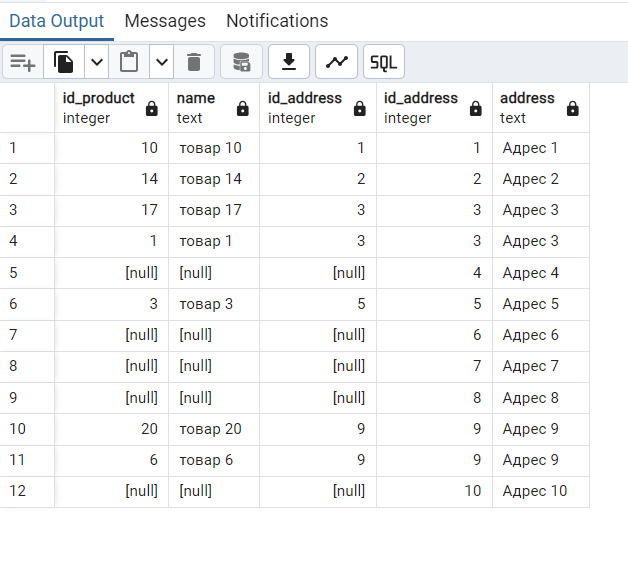


Рисунок 4.7 – Результат запроса

Следующая диаграмма (рис. 4.8) иллюстрирует правое соединение:

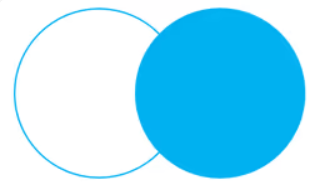


Рисунок 4.8 – RIGHT OUTER JOIN

Аналогично можно получить строки из правой таблицы, которым не соответствуют строки из левой таблицы, добавив WHERE:

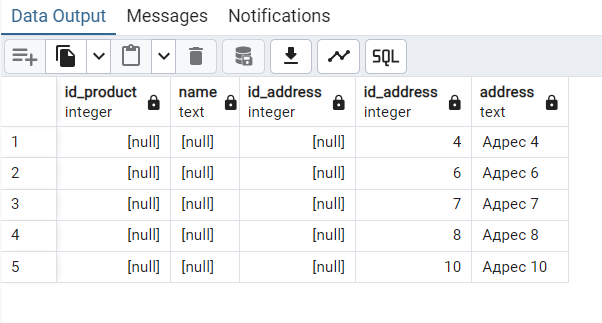
SELECT p.\*, a.\*

FROM products AS p

RIGHT JOIN address AS a ON a.id\_address = p.id\_address

WHERE p.id\_address IS NULL;

Результат запроса (рис. 4.9):



Результат запроса (рис. 4.9):

Следующая диаграмма (рис. 4.10) иллюстрирует правое соединение, которое возвращает строки из правой таблицы, которым не соответствуют строки в левой таблице:

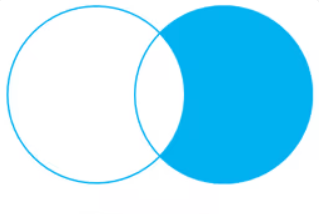


Рисунок 4.10 – RIGHT OUTER JOIN

только строки из правой таблицы

## 4.6. Полное внешнее соединение

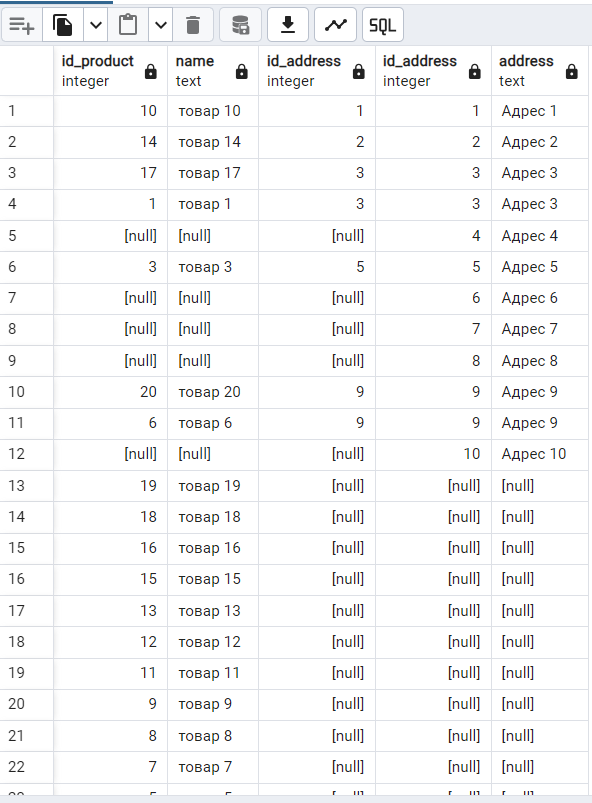
Полное внешнее соединение FULL OUTER JOIN (или FULL JOIN) возвращает результирующий набор, содержащий все строки из левой и правой таблиц, с соответствующими строками с обеих сторон, если они имеются. В случае отсутствия совпадений столбцы таблицы будут заполнены NULL:

SELECT p.\*, a.\*

FROM products AS p

FULL JOIN address AS a ON a.id\_address = p.id\_address;

Результат запроса (рис. 4.11):



Результат запроса (рис. 4.11):

Следующая диаграмма (рис. 4.12) иллюстрирует полное внешнее соединение:

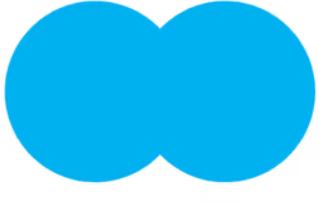


Рисунок 4. 12 – FULL OUTER JOIN

Чтобы вернуть строки в таблице, которым не соответствуют строки в другой таблице, используйте полное соединение с WHERE:

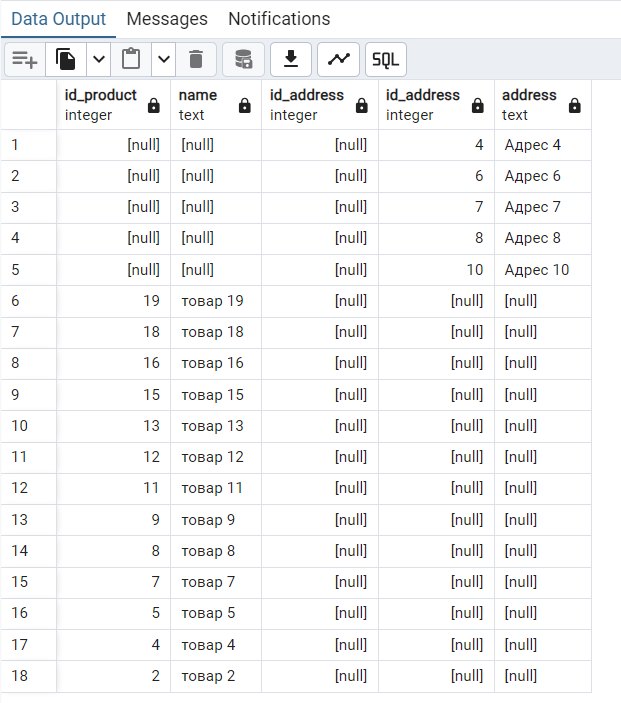
SELECT p.\*, a.\*

FROM products AS p

FULL JOIN address AS a ON a.id\_address = p.id\_address

WHERE p.id\_address IS NULL OR a.id\_address IS NULL;

Результат запроса (рис. 4.13):



Результат запроса (рис. 4.13):

Следующая диаграмма иллюстрирует полное внешнее соединение, возвращающее строки из таблицы, которым не соответствуют строки в другой таблице:

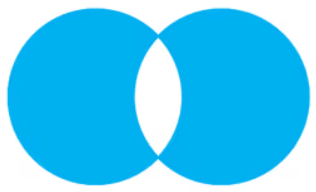


Рисунок 4.14 – FULL OUTER JOIN

только уникальные строки из обеих таблиц

## 4.7. Перекрестное соединение

В PostgreSQL перекрестное соединение позволяет объединить две таблицы, объединив каждую строку из первой таблицы с каждой строкой из второй таблицы, что приводит к полному объединению всех строк. Перекрестное соединение производит декартово произведение строк двух таблиц.

В отличие от других типов соединения, таких как LEFT JOIN или INNER JOIN, в этом CROSS JOIN предложении нет условия соединения.

Предположим, вам нужно выполнить CROSS JOIN из table1 и table2.

Если table1 имеет n строк и table2 имеет m строк, то CROSS JOIN вернет результирующий набор, содержащий (n x m) строк.

Поскольку CROSS JOIN может генерировать большой набор результатов, следует использовать его осторожно, чтобы избежать проблем с производительностью.

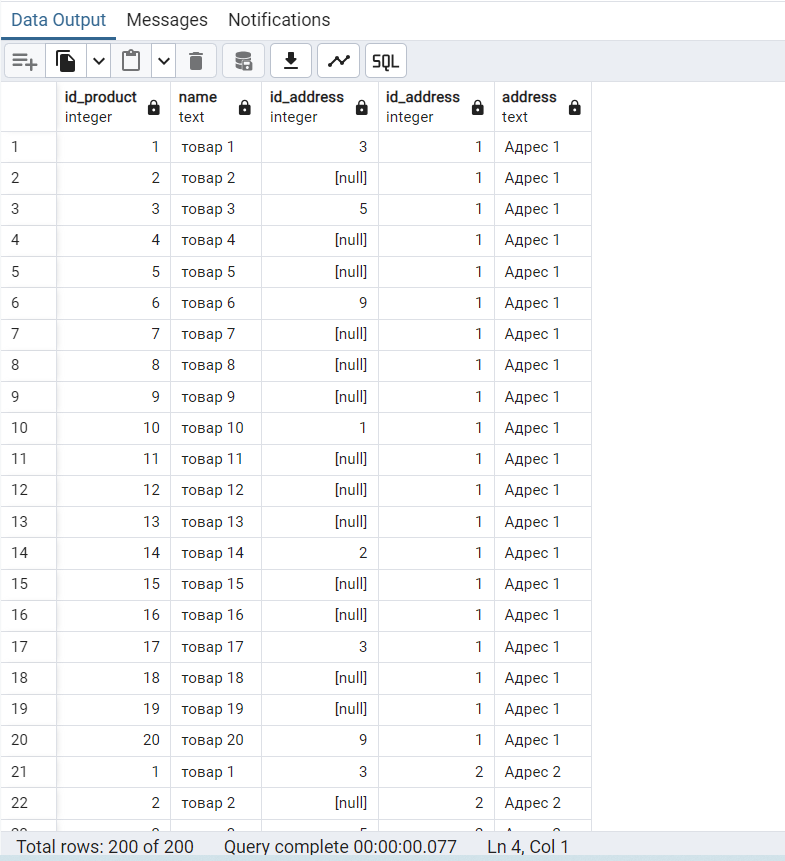
Применим CROSS JOIN к нашим таблицам:

SELECT p.\*, a.\*

FROM products AS p

CROSS JOIN address AS a;

Результат запроса (рис. 4.15):



Результат запроса (рис. 4.15):

Перекрестное соединение эквивалентно приведенному ниже утверждению:

SELECT p.\*, a.\*

FROM products AS p,address AS a;

## 4.8. Естественное соединение

Предложение ON является наиболее общим видом условия соединения: оно принимает логическое значение выражения того же типа, что используется в WHERE предложении. Пара строк из таблица1 и таблица2совпадают, если выражение в ON является истинным.

В Postgres можно использовать сокращение USING, которое позволяет воспользоваться конкретной ситуацией, когда объединение таблиц выполняется на основе равенства столбцов, имеющих одно и тоже имя. Например, приведенные ниже команды эквивалентны:

SELECT p.\*, a.\*

FROM products AS p

JOIN address AS a USING(id\_address);

SELECT p.\*, a.\*

FROM products AS p

JOIN address AS a ON a.id\_address = p.id\_address;

В ключевом слове USING можно указать несколько столбцов, например:

SELECT список\_столбцов

FROM таблица1

[INNER, LEFT, RIGHT] JOIN таблица2 USING(столбец1, столбец2);

Соединение будет выполняться при условии:

SELECT список\_столбцов

FROM таблица1

[INNER, LEFT, RIGHT] JOIN таблица2

ON таблица1.столбец1 = таблица2.столбец1

AND таблица1.столбец2 = таблица2.столбец2;

# 5. Оператор объединения UNION

Оператор UNION позволяет объединить наборы результатов двух или более операторов SELECT в один набор результатов.

Вот основной синтаксис оператора UNION:

SELECT список\_столбцов

FROM таблица1

UNION [ALL]

SELECT список\_столбцов

FROM таблица2;

Оператор UNION удаляет все дубликаты строк из объединенного набора данных. Чтобы сохранить дубликаты строк, используйте UNION ALL.

В этом синтаксисе запросы должны соответствовать следующим правилам:

* список\_столбцов в двух операторах SELECT должны иметь одинаковое количество столбцов,
* типы данных в соответствующих столбцах должны быть совместимыми.

Если столбцы имеют разные имена, то результирующему столбцу присваивается имя столбца из первого запроса.

Операторы UNION и UNION ALL не упорядочивают строки в конечном наборе. Чтобы отсортировать строки в конечном результирующем наборе, укажите ORDER BY после второго запроса:

SELECT \*

FROM products

WHERE id\_address < 7

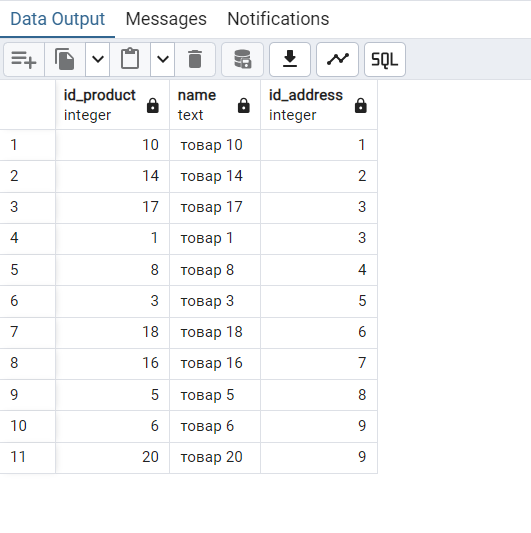
UNION

SELECT \*

FROM products

WHERE 4 < id\_address

ORDER BY id\_address



Результат запроса (рис. 4.16):

Выполним те же запросы с использованием оператора UNION ALL:

SELECT \*

FROM products

WHERE id\_address < 7

UNION ALL

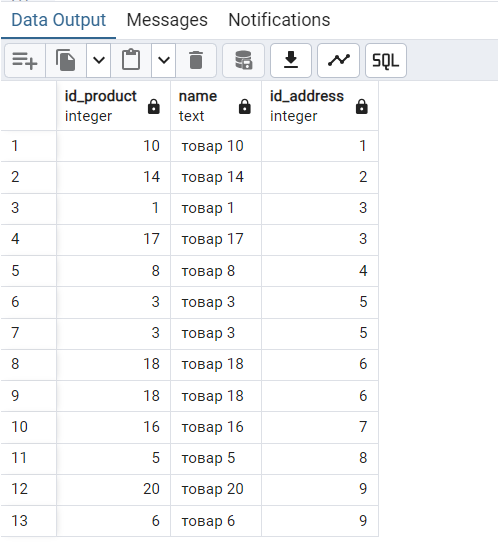
SELECT \*

FROM products

WHERE 4 < id\_address

ORDER BY id\_address;

Результат запроса (рис. 4.17):



Результат запроса (рис. 4.17):

Как можем видеть появились две повторяющиеся строки – 7 и 9.

# 6. Оператор пересечения INTERSECT

Подобно операторам UNION, оператор INTERSECT объединяет результаты двух операторов SELECT в один результирующий набор, содержащий строки, доступные в обоих наборах (пересечение).

Вот основной синтаксис оператора INTERSECT:

SELECT список\_столбцов

FROM таблица1

INTERSECT

SELECT список\_столбцов

FROM таблица2;

Объединяемые запросы должны быть совместимы между собой (см. оператор UNION).

Выполним предыдущий запрос с применением оператора INTERSECT:

SELECT \*

FROM products

WHERE id\_address < 7

INTERSECT

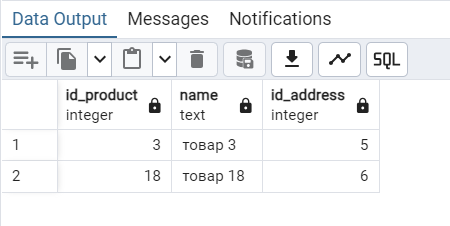
SELECT \*

FROM products

WHERE 4 < id\_address

ORDER BY id\_address;

Результат запроса (рис. 4.18):



Результат запроса (рис. 4.18):

# 7. Оператор вычитания EXCEPT

Оператор EXCEPT возвращает строки из первого запроса, которых нет во втором запросе (вычитание).

Вот основной синтаксис оператора EXCEPT:

SELECT список\_столбцов

FROM таблица1

EXCEPT

SELECT список\_столбцов

FROM таблица2;

Запросы должны быть совместимы между собой (см. оператор UNION).

Выполним предыдущий запрос с применением оператора EXCEPT:

SELECT \*

FROM products

WHERE id\_address < 7

EXCEPT

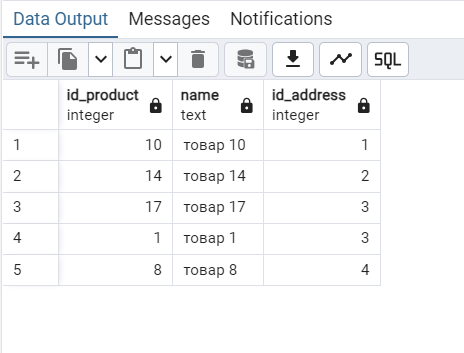
SELECT \*

FROM products

WHERE 4 < id\_address

ORDER BY id\_address;

Результат запроса (рис. 4.19):



Результат запроса (рис. 4.19):

# 8. Темы для самостоятельной проработки

1. Изучить агрегатные функции.
2. Изучить строковые функции и функции даты.

# Контрольные вопросы

1. Знать материал, изложенный в методическом материале.
2. Чем отличается char(n) от varchar(n)?
3. Рассказать про агрегатные функции, предложения GROUP BY и HAVING.
4. Как выбрать только уникальные значения какого-либо поля?
5. Как осуществить сортировку по возрастанию/убыванию по значению какого-либо поля?
6. Как агрегатные функции ведут себя по отношению к неопределённым значениям?
7. Чем отличаются count(\*) и count(поле)?
8. Как подсчитать количество уникальных значений поля?
9. Как можно осуществить проверку на неопределенное значение?
10. Рассказать про предикат LIKE.
11. Как можно выбрать только определенное количество строк?
12. Как округлить вещественное число?
13. Как определить максимальное/минимальное значение, не используя функцию max (min).
14. Какие форматы вывода данных даты и времени.
15. Как извлечь составляющие даты и времени?
16. Для чего нужны тип данных семейства serial и identity?
17. Что такое JOIN, какие существуют типы JOIN?
18. Что такое псевдонимы и когда их целесообразно использовать?
19. Что делают операторы UNION, INTERSECT и EXCEPT?

# 10. Практическая часть

1. Установите СУБД PostgreSQL.
2. Установите pgAdmin 4.
3. Разверните учебную БД demo.
4. В таблице tickets имена пассажиров passenger\_name занесены в виде   
   «имя фамилия» латинскими буквами. Выберите всех пассажиров с именами, состоящими из пяти букв.
5. Используя таблицу tickets, определите, как часто встречаются пассажиры с одинаковыми фамилиями.
6. В таблице flights перечислены все рейсы, которые выполняются между аэропортами. Напишите запрос, который выводит количество направлений для каждого аэропорта (между двумя аэропортами выполняется несколько рейсов, а направление – одно).
7. Объяснить результат запроса.  
   SELECT 'abc '::varchar(5) || '!', 'abc '::char(5) || '!';  
   Результат:  
   "abc !" "abc!"

# 11. Список использованных источников

1. Фомин М.М. Реляционные базы данных. Учебное пособие для бакалавров <https://e-learning.bmstu.ru/iu6/mod/resource/view.php?id=6634>
2. Карпова И.П. Базы данных. Учебное пособие. – Московский государственный институт электроники и математики (Технический университет): учебное пособие / И.П. Карпова; – М., 2009. – 140-141 с, 102 c.
3. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. - СПб.: БХВ-Петербург, 2018. - 65-68 c, 68-72 с.
4. Стоунз Р., Мэтью Н. PostgreSQL. Основы. – Пер. с англ. – СПб: Символ Плюс, 2002. – 640 с., ил. ISBN 5\_93286\_043\_X
5. Болье, А. Изучаем SQL. Генерация, выборка и обработка данных, 3-е изд./ пер. с англ. И.В. Красикова. — Киев.: “Диалектика”,2021. — 402 с.: ил. ISBN 978-1-492-05761-1
6. Синтаксис SQL: <https://postgrespro.ru/docs/postgrespro/14/sql-syntax>
7. SELECT-запросы: <https://postgrespro.ru/docs/postgrespro/14/queries>
8. Функции и операторы: https://postgrespro.ru/docs/postgrespro/14/functions
9. Полное описание синтаксиса встретившихся команд: <https://postgrespro.ru/docs/postgrespro/14/sql-commands>
10. Оператор With и рекурсивные запросы: <https://postgrespro.ru/docs/postgrespro/14/queries-with>
11. Работа в среде pgAdmin: https://www.pgadmin.org/docs/pgadmin4/latest/index.html

1. PostgreSQL 15 и более поздние версии позволяют объявлять scale отрицательным. [↑](#footnote-ref-1)
2. Ключевое слово RETURNING \* возвращает вставленную строку. [↑](#footnote-ref-2)