МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»  
(ФГБОУ ВО «ВГТУ», ВГТУ)

Факультет информационных технологий и компьютерной безопасности

Кафедра компьютерных интеллектуальных технологий проектирования

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №9

По дисциплине: «Системы хранения и обработки данных»

Тема: «Формирование запросов к базе данных»

Выполнил работу студент группы змИИВТ-241: Авсянкина М.А.

подпись, дата

Принял: Короленко В.В.

подпись, дата

Воронеж 2024

Цель работы: изучить синтаксис и основные команды для формирования запросов к базе данных, освоить процесс формирования SQL-запросов.

Основные задачи:

• Разработать структуру базы данных в СУБД Postgres.

• Наполнить базу данных данными.

• Написать ряд запросов к базе данных для отработки навыка формирования SQL-запросов.

• Научиться ставить задачу по формированию выборки необходимых данных из базы данных и решать её с помощью SQL-запросов.

Ход работы:

**Основные команды SQL**

**1. SELECT**

Команда SELECT используется для выборки данных из одной или нескольких таблиц базы данных. Синтаксис этой команды, следующий:

SELECT [DISTINCT | ALL] column\_list

FROM table\_name

[WHERE condition]

[GROUP BY expression]

[HAVING condition]

[ORDER BY expression [ASC|DESC]]

[LIMIT count]

[OFFSET start];

**2. INSERT**

Команда INSERT используется для вставки новых данных в таблицу. Синтаксис выглядит так:

INSERT INTO table\_name (column1, column2, ...)

VALUES (value1, value2, ...);

**3. UPDATE**

Команда UPDATE используется для обновления существующих данных в таблице. Синтаксис будет выглядеть следующим образом:

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

[WHERE condition];

**4. DELETE**

Команда DELETE удаляет данные из таблицы. Синтаксис:

DELETE FROM table\_name

[WHERE condition];

**Создание и управление таблицами**

1. **CREATE TABLE**

Создает новую таблицу в базе данных. Синтаксис создания таблицы:

CREATE TABLE table\_name (

column1 datatype,

column2 datatype,

...

);

1. **ALTER TABLE**

Изменяет структуру существующей таблицы. Пример написания синтаксиса:

ALTER TABLE table\_name

ADD column\_name datatype;

1. **DROP TABLE**

Удаляет таблицу из базы данных. Синтаксис удаления таблицы:

DROP TABLE table\_name;

**Индексы и ограничения**

**PRIMARY KEY**

Уникальный идентификатор записи в таблице. Может быть создан при создании таблицы или добавлен позже. Синтаксис уникального индефикатора:

CREATE TABLE customers (

customer\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(100),

last\_name VARCHAR(100)

);

**FOREIGN KEY**

Ссылочная целостность между двумя таблицами. Указывает, что значение в одном столбце должно существовать в другой таблице. Синтаксис**:**

CREATE TABLE orders (

order\_id SERIAL PRIMARY KEY,

customer\_id INT REFERENCES customers(customer\_id)

);

**REFERENCES**

Используется для создания внешнего ключа (foreign key) в таблице. Внешний ключ устанавливает связь между двумя таблицами, обеспечивая ссылочную целостность данных. Синтаксис создания таблицы с внешним ключом:

CREATE TABLE orders (

order\_id SERIAL PRIMARY KEY,

customer\_id INT REFERENCES customers(customer\_id)

);

**UNIQUE –** Обеспечивает уникальность значений в указанном столбце.

**Пример:**

CREATE TABLE emails (

email\_id SERIAL PRIMARY KEY,

email\_address VARCHAR(200) UNIQUE

);

**CHECK –** Проверяет условие перед добавлением или обновлением данных.

**Пример:**

CREATE TABLE products (

product\_id SERIAL PRIMARY KEY,

product\_price NUMERIC CHECK (product\_price >= 0)

);

**Подзапросы**

Подзапросы позволяют использовать результаты одного запроса в другом запросе.

**Пример:**

SELECT \*

FROM users

WHERE id IN (SELECT user\_id FROM orders WHERE total\_amount > 500);

**Объединение таблиц**

**INNER JOIN –** Возвращает строки, которые имеют совпадающие значения в обеих таблицах.

**Пример:**

SELECT users.name, orders.order\_date

FROM users

INNER JOIN orders ON users.id = orders.user\_id;

**LEFT JOIN**

Возвращает все строки из левой таблицы и соответствующие строки из правой таблицы. Если соответствия нет, то вместо значений правой таблицы будут NULL.

**Пример:**

SELECT users.name, orders.order\_date

FROM users

LEFT JOIN orders ON users.id = orders.user\_id;

**RIGHT JOIN –** Аналогично LEFT JOIN, но возвращает все строки из правой таблицы.

**Агрегатные функции**

**COUNT() –** Подсчитывает количество строк.

**Пример:**

SELECT COUNT(\*)

FROM users;

**SUM() –** Суммирует значения в столбце.

**AVG() –** Вычисляет среднее арифметическое значение.

**MAX(), MIN() –** Находит максимальное и минимальное значения соответственно.

**Пример:**

SELECT MAX(salary)

FROM employees;

**Способы оптимизации запросов**

Оптимизация запросов – это процесс улучшения производительности SQL-запросов путем уменьшения времени их выполнения и повышения эффективности использования ресурсов базы данных. Вот несколько ключевых методов оптимизации:

**Использование индексов**:

Индексы ускоряют доступ к данным за счет упорядочивания данных и создания структур, облегчающих поиск. Для часто используемых полей следует создавать индексы, особенно для тех, которые участвуют в условиях WHERE, JOIN и ORDER BY.

CREATE INDEX idx\_users\_age ON users(age);

**Выбор правильного типа соединения (JOIN)**:

Правильный выбор типа соединения (INNER JOIN, LEFT JOIN, RIGHT JOIN, CROSS JOIN) влияет на производительность. Используйте наиболее подходящий тип соединения в зависимости от ваших требований.

**Разбиение сложных запросов на подзапросы**:

Иногда сложные запросы можно разбить на несколько простых подзапросов, что облегчает их выполнение и оптимизацию.

WITH recent\_orders AS (

SELECT user\_id, order\_date

FROM orders

WHERE order\_date > CURRENT\_DATE - INTERVAL '7 days'

)

SELECT u.name, ro.order\_date

FROM users u

INNER JOIN recent\_orders ro ON u.id = ro.user\_id;

**Минимизация использования подзапросов в списке выбора (SELECT)**:

Подзапросы в списке выбора могут замедлять выполнение запроса. Постарайтесь заменить их на более эффективные конструкции, такие как соединение таблиц.

-- Не эффективная конструкция

SELECT u.id, (SELECT COUNT(\*) FROM orders o WHERE o.user\_id = u.id) AS order\_count

FROM users u;

-- Эффективная конструкция

SELECT u.id, COUNT(o.user\_id) AS order\_count

FROM users u

LEFT JOIN orders o ON u.id = o.user\_id

GROUP BY u.id;

**Оптимизация условий WHERE**:

Старайтесь минимизировать количество условий в секции WHERE, используя операторы EXISTS, IN и другие оптимальные методы фильтрации.

-- Не достаточно хороший вариант

SELECT \*

FROM users

WHERE age > 18 AND age < 30;

-- Оптимальный вариант

SELECT \*

FROM users

WHERE age BETWEEN 18 AND 30;

**Использование временных таблиц**:

Временные таблицы могут использоваться для хранения промежуточных результатов сложных запросов, что уменьшает нагрузку на основную базу данных.

**План запроса**

План запроса – это последовательность шагов, которую база данных планирует выполнить для получения результата запроса. Понимание плана запроса помогает выявить узкие места и оптимизировать запросы.

Для просмотра плана запроса в PostgreSQL используется команда EXPLAIN:

EXPLAIN ANALYZE

SELECT u.name, o.order\_date

FROM users u

INNER JOIN orders o ON u.id = o.user\_id;

Эта команда покажет вам, какие операции выполняются, сколько времени они занимают, используются ли индексы и многое другое. Анализируя план запроса, вы можете определить, где происходят задержки, и внести необходимые изменения.

**Оконные функции**

Оконные функции – это специальные функции, которые применяются к каждой строке набора данных, основываясь на определенной группе строк (оконной рамке). Они полезны для вычислений, связанных с агрегированием, ранжированием и другими операциями над группами строк.

Вот несколько примеров оконных функций:

**ROW\_NUMBER()**:

Присваивает каждой строке уникальный номер в пределах окна.

SELECT u.id, u.name, ROW\_NUMBER() OVER (PARTITION BY u.department ORDER BY u.salary DESC) AS rank\_in\_department

FROM users u;

**RANK()**:

Аналогична ROW\_NUMBER(), но присваивает одинаковые номера строкам с одинаковыми значениями.

SELECT u.id, u.name, RANK() OVER (PARTITION BY u.department ORDER BY u.salary DESC) AS rank\_in\_department

FROM users u;

**DENSE\_RANK()**:

Похожа на RANK(), но не оставляет пропусков в нумерации.

SELECT u.id, u.name, DENSE\_RANK() OVER (PARTITION BY u.department ORDER BY u.salary DESC) AS dense\_rank\_in\_department

FROM users u;

**LEAD() и LAG()**:

Возвращают значения из следующей или предыдущей строки относительно текущей.

SELECT u.id, u.name, LEAD(u.salary) OVER (ORDER BY u.salary) AS next\_salary

FROM users u;

Эти функции очень удобны для анализа данных и построения отчетов, позволяя выполнять сложные вычисления прямо в запросах.

Оптимизация запросов, понимание планов запросов и работа с оконными функциями являются важными навыками для эффективного взаимодействия с базами данных. Освоив эти аспекты, мы можем существенно улучшить производительность своих приложений и упростить обработку больших объемов данных.

Сформируем SQL-запросы для создания таблицы в СУБД Postgres в соответствии со следующей структурой:

1. Customers — информация о клиентах:

* CustomerID (int, PK) — уникальный идентификатор клиента.
* FirstName (varchar) — имя клиента.
* LastName (varchar) — фамилия клиента.
* Email (varchar) — адрес электронной почты клиента.

1. Orders — информация о заказах:

* OrderID (int, PK) — уникальный идентификатор заказа.
* CustomerID (int) — идентификатор клиента, совершившего заказ.
* OrderDate (datetime) — дата и время создания заказа.
* TotalAmount (decimal) — общая стоимость заказа.

1. OrderDetails — информация о деталях заказов:

* OrderDetailID (int, PK) — уникальный идентификатор детали заказа.
* OrderID (int) — идентификатор заказа.
* ProductID (int) — идентификатор продукта в заказе.
* Quantity (int) — количество продуктов в заказе.
* UnitPrice (decimal) — цена за единицу продукта.

И создадим также ещё одну таблицу «product», в которой будет храниться информация о названии продукта, его категории, краткой информации и стоимости.

Вид итоговых SQL-запросов, для создания таблиц:

CREATE TABLE Customers (

    CustomerID INT PRIMARY KEY,

    FirstName VARCHAR(100),

    LastName VARCHAR(100),

    Email VARCHAR(255)

);

CREATE TABLE Product (

    ProductID INT PRIMARY KEY,

    Name VARCHAR(100),

    Category VARCHAR(50),

    Description TEXT,

    UnitPrice DECIMAL(10, 2)

);

CREATE TABLE Orders (

    OrderID INT PRIMARY KEY,

    CustomerID INT,

    OrderDate TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

    TotalAmount DECIMAL(10, 3)

);

CREATE TABLE Order\_Details (

    Order\_Detail\_ID INT PRIMARY KEY,

    OrderID INT,

    ProductID INT,

    Quantity INT,

    UnitPrice DECIMAL(10, 3),

    CONSTRAINT fk\_OrderID FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),

  CONSTRAINT fk\_ProductID FOREIGN KEY (ProductID) REFERENCES Product(ProductID)

);

ALTER TABLE Orders ADD FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID);

Ниже на рисунке 1, представлена диаграмма, которая у нас получилась при открытии созданных SQL-запросов в программе. Как видно из нее, вся структура связаны, имеет внешние ключи и уникальные индефикаторы.

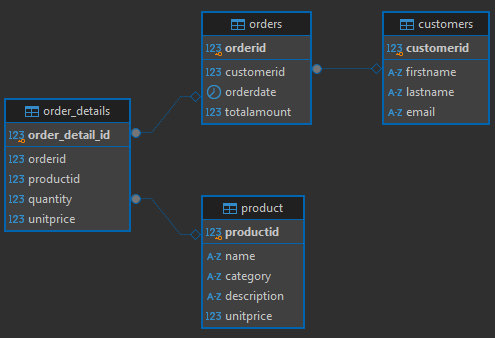


Рисунок 1 – Диаграмма, сформированная на SQL-запросах

Далее, продолжим наполнять нашу базу данных данными:

INSERT INTO Customers (CustomerID, FirstName, LastName, Email) VALUES

(1, 'Иван', 'Иванов', 'ivanov@example.com'),

(2, 'Петр', 'Петров', 'petrov@example.com'),

(3, 'Сергей', 'Сергеев', 'sergeev@example.com'),

(4, 'Андрей', 'Андреев', 'andreev@example.com'),

(5, 'Алексей', 'Александров', 'alexandrov@example.com'),

(6, 'Дмитрий', 'Дмитриев', 'dmitriev@example.com'),

(7, 'Елена', 'Еленина', 'elenina@example.com'),

(8, 'Марина', 'Маринова', 'marinova@example.com'),

(9, 'Ольга', 'Ольгина', 'olgina@example.com'),

(10, 'Виктор', 'Викторов', 'viktorov@example.com'),

(11, 'Ксения', 'Ксенина', 'ksenina@example.com'),

(12, 'Михаил', 'Михайлов', 'mihailov@example.com'),

(13, 'Николай', 'Николаев', 'nikolaev@example.com'),

(14, 'Владимир', 'Владимиров', 'vladimirov@example.com'),

(15, 'Евгений', 'Евгеньевич', 'evgenieevich@example.com'),

(16, 'Антон', 'Антонович', 'antonovich@example.com'),

(17, 'Валерий', 'Валерьевич', 'valerievich@example.com'),

(18, 'Василий', 'Сидоров', 'vasily@example.com'),

(19, 'Анна', 'Анненкова', 'annenkova@example.com'),

(20, 'Федор', 'Федоров', 'fedorov@example.com');

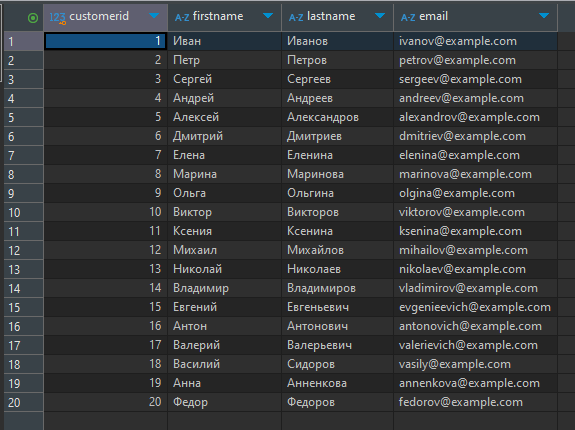


Рисунок 2 – Отображение внесенных данных в таблицу Customers

INSERT INTO Product (ProductID, Name, Category, Description, UnitPrice) VALUES

(1, 'Ноутбук', 'Электроника', 'Высокопроизводительный ноутбук для профессионалов', 1200.00),

(2, 'Смартфон', 'Электроника', 'Последний смартфон с расширенными функциями', 800.00),

(3, 'Наушники', 'Аксессуары', 'Беспроводные наушники с шумоподавлением', 150.00),

(4, 'Внешний жесткий диск', 'Хранилище', 'Внешний жесткий диск на 1ТБ для хранения данных', 70.00),

(5, 'Мышь', 'Периферия', 'Эргономичная беспроводная мышь', 30.00),

(6, 'Клавиатура', 'Периферия', 'Механическая клавиатура для любителей набора текста', 100.00),

(7, 'Монитор', 'Дисплеи', '24-дюймовый монитор Full HD', 200.00),

(8, 'Принтер', 'Офисное оборудование', 'Многофункциональный принтер/сканер/копир', 150.00),

(9, 'Планшет', 'Электроника', '7-дюймовый планшет для использования в пути', 100.00),

(10, 'Камера', 'Фотография', 'Цифровая камера с высокочувствительным сенсором', 500.00),

(11, 'Роутер', 'Сеть', 'Беспроводной роутер для домашнего использования', 90.00),

(12, 'Кабель HDMI', 'Аксессуары', 'Кабель HDMI для подключения устройств', 20.00),

(13, 'Зарядное устройство', 'Аксессуары', 'Универсальное зарядное устройство для смартфонов', 50.00),

(14, 'Чехол для ноутбука', 'Аксессуары', 'Защитный чехол для ноутбука', 40.00),

(15, 'USB-флешка', 'Хранилище', 'USB-флешка на 64 ГБ', 30.00),

(16, 'Bluetooth-колонка', 'Аудио', 'Портативная Bluetooth-колонка', 100.00),

(17, 'Игровая консоль', 'Развлечения', 'Современная игровая консоль', 500.00),

(18, 'Гарнитура', 'Аудио', 'Игровая гарнитура с микрофоном', 150.00),

(19, 'Экшн-камера', 'Фотография', 'Водонепроницаемая экшн-камера', 300.00),

(20, 'МФУ', 'Офисное оборудование', 'Многофункциональное устройство (принтер/сканер/копир)', 180.00);

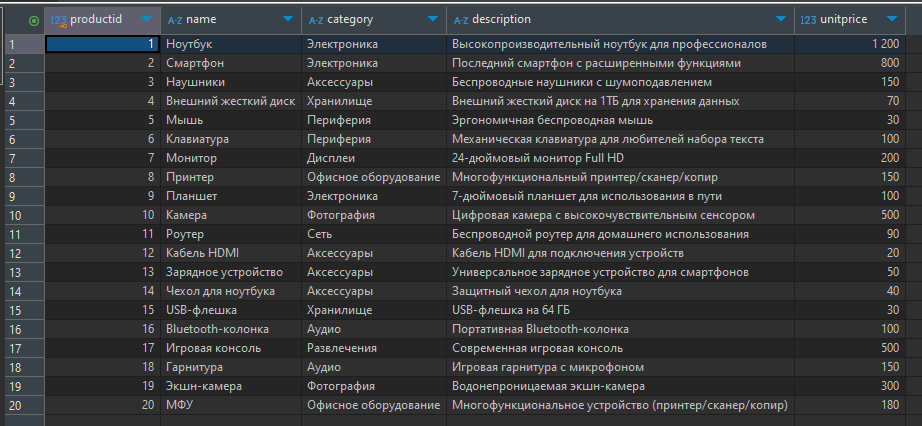


Рисунок 3 – Отображение внесенных данных в таблицу Product

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES

(1, 1, '2023-01-05 15:30:00', 1200.500),

(2, 2, '2023-02-07 09:45:00', 800.250),

(3, 3, '2023-03-11 14:55:00', 1500.750),

(4, 4, '2023-04-13 08:35:00', 200.000),

(5, 5, '2023-05-17 16:22:00', 1100.600),

(6, 6, '2023-06-19 21:48:00', 700.900),

(7, 7, '2023-07-23 12:33:00', 1300.800),

(8, 8, '2023-08-27 10:41:00', 2500.950),

(9, 9, '2023-09-29 17:59:00', 200.000),

(10, 10, '2023-10-31 13:28:00', 500.000),

(11, 11, '2023-11-15 10:34:00', 1450.650),

(12, 12, '2023-12-20 11:47:00', 980.350),

(13, 13, '2024-01-25 12:58:00', 1750.850),

(14, 14, '2024-02-29 14:09:00', 300.000),

(15, 15, '2024-03-31 15:20:00', 1250.700),

(16, 16, '2024-04-12 16:31:00', 850.900),

(17, 17, '2024-05-23 17:42:00', 1550.800),

(18, 18, '2024-06-27 18:53:00', 2750.950),

(19, 19, '2024-07-30 19:59:00', 400.000),

(20, 20, '2024-08-31 20:32:00', 600.000),

(21, 1, '2023-01-05 15:30:00', 2400.500),

(22, 2, '2023-02-07 09:45:00', 1900.250),

(23, 3, '2023-03-11 14:55:00', 2600.750),

(24, 4, '2023-04-13 08:35:00', 3200.000),

(25, 5, '2023-05-17 16:22:00', 4100.600),

(26, 6, '2023-06-19 21:48:00', 3700.900),

(27, 7, '2023-07-23 12:33:00', 4300.800),

(28, 8, '2023-08-27 10:41:00', 5200.950),

(29, 9, '2023-09-29 17:59:00', 4200.000),

(30, 10, '2023-10-31 13:28:00', 5600.000);

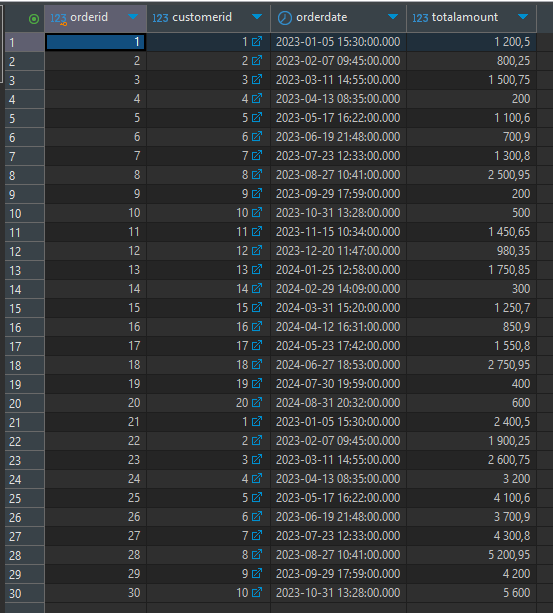


Рисунок 4 – Отображение внесенных данных в таблицу Orders

INSERT INTO Order\_Details (Order\_Detail\_ID, OrderID, ProductID, Quantity, UnitPrice) VALUES

(1, 1, 1, 2, 1200.50),

(2, 1, 2, 1, 800.25),

(3, 1, 3, 3, 150.00),

(4, 1, 4, 2, 30.00),

(5, 1, 5, 1, 100.00),

(6, 2, 1, 1, 70.00),

(7, 2, 2, 1, 100.00),

(8, 2, 3, 1, 100.00),

(9, 2, 4, 4, 200.00),

(10, 2, 5, 1, 150.00),

(11, 3, 1, 2, 1100.60),

(12, 3, 2, 1, 500.00),

(13, 4, 1, 3, 100.00),

(14, 4, 2, 1, 100.00),

(15, 4, 3, 1, 500.00),

(16, 4, 4, 1, 100.00),

(17, 4, 5, 1, 2500.95),

(18, 5, 1, 1, 800.00),

(19, 5, 2, 1, 1200.50),

(20, 6, 1, 1, 800.00),

(21, 6, 2, 1, 1200.50),

(22, 7, 1, 3, 1300.80),

(23, 7, 2, 1, 70.00),

(24, 7, 3, 1, 100.00),

(25, 7, 4, 3, 30.00),

(26, 7, 5, 1, 100.00),

(27, 8, 1, 2, 200.00),

(28, 8, 2, 1, 150.00),

(29, 8, 3, 1, 200.00),

(30, 8, 4, 1, 150.00),

(31, 8, 5, 1, 500.00),

(32, 9, 1, 1, 450.00),

(33, 9, 2, 1, 550.00),

(34, 9, 3, 1, 600.00),

(35, 9, 4, 1, 700.00),

(36, 9, 5, 1, 800.00),

(37, 10, 1, 1, 900.00),

(38, 10, 2, 1, 1000.00),

(39, 10, 3, 1, 1100.00),

(40, 10, 4, 1, 1200.00),

(41, 10, 5, 1, 1300.00),

(42, 11, 6, 1, 1400.00),

(43, 11, 7, 1, 1500.00),

(44, 11, 8, 1, 1600.00),

(45, 11, 9, 1, 1700.00),

(46, 11, 10, 1, 1800.00),

(47, 12, 11, 1, 1900.00),

(48, 12, 12, 1, 2000.00),

(49, 12, 13, 1, 2100.00),

(50, 12, 14, 1, 2200.00);

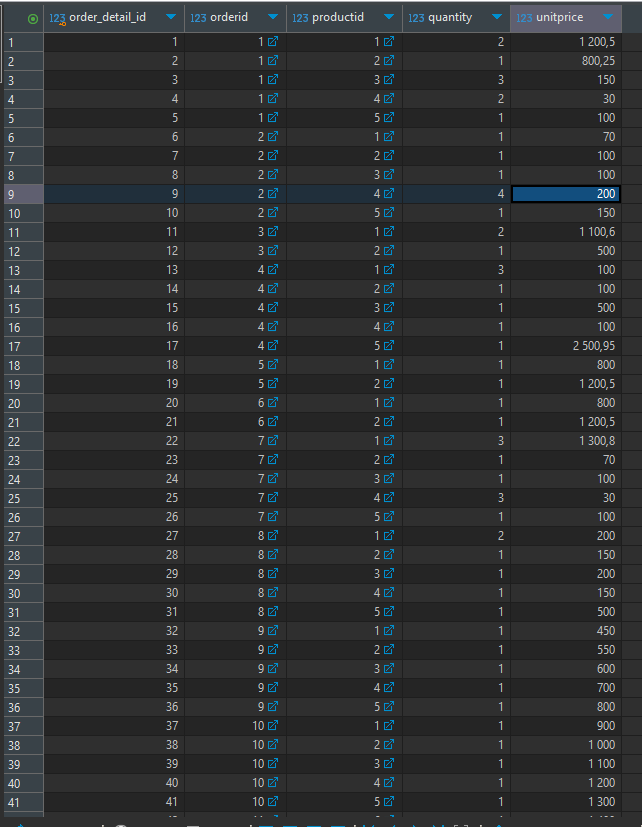


Рисунок 5 – Отображение внесенных данных в таблицу Order\_Details

После того, как закончили наполнять данными нашу базу данных, начнем формировать SQL-запросы, для удобства подсчетов и вывода данных на экран в DBeaver.

Первым сформируем запрос, который будет возвращаться список клиентов и суммарную стоимость заказов каждого клиента:

SELECT c.FirstName, c.LastName, SUM(o.TotalAmount) AS TotalSum

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.FirstName, c.LastName;

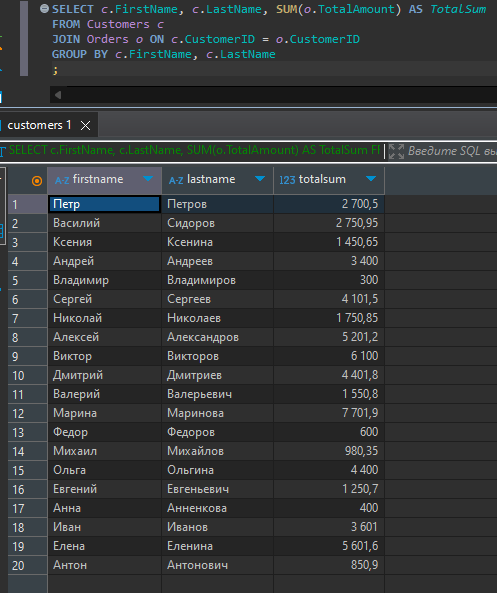


Рисунок 6 – Список клиентов и суммарная стоимость их заказов

Как видим, на рисунке 6, запрос был, верно, нами составлен, так как вывелись столбцы Имя, Фамилия, а также выполнился расчет суммы всех заказов данного клиента.

Далее отсортируем полученный список по убыванию суммарной стоимости заказов клиента:

SELECT c.FirstName, c.LastName, SUM(o.TotalAmount) AS TotalSum

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.FirstName, c.LastName

ORDER BY TotalSum DESC;

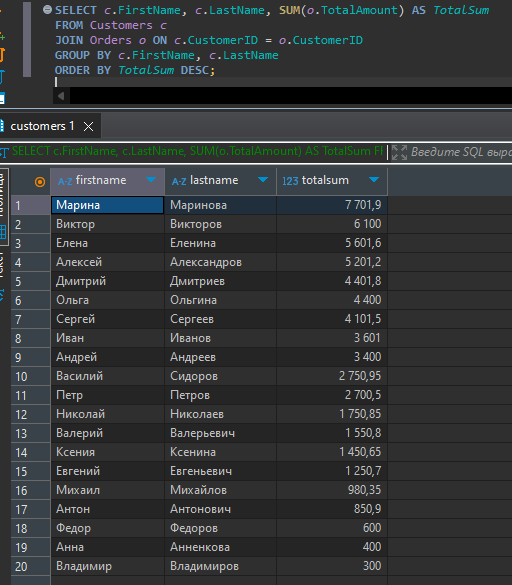


Рисунок 7 – Сортировка по убыванию стоимости заказов клиента

Далее в нашу таблицу добавим столбец со средней суммарной стоимостью заказов:

SELECT FirstName, LastName, TotalSum, AvgTotalSum

FROM (

    SELECT c.FirstName, c.LastName, SUM(o.TotalAmount) AS TotalSum

    FROM Customers c

    JOIN Orders o ON c.CustomerID = o.CustomerID

    GROUP BY c.FirstName, c.LastName

) AS subquery

CROSS JOIN (

    SELECT AVG(TotalSum) AS AvgTotalSum

    FROM (

        SELECT SUM(o.TotalAmount) AS TotalSum

        FROM Customers c

        JOIN Orders o ON c.CustomerID = o.CustomerID

        GROUP BY c.FirstName, c.LastName

    ) AS inner\_subquery

) AS avg\_subquery

ORDER BY TotalSum DESC;

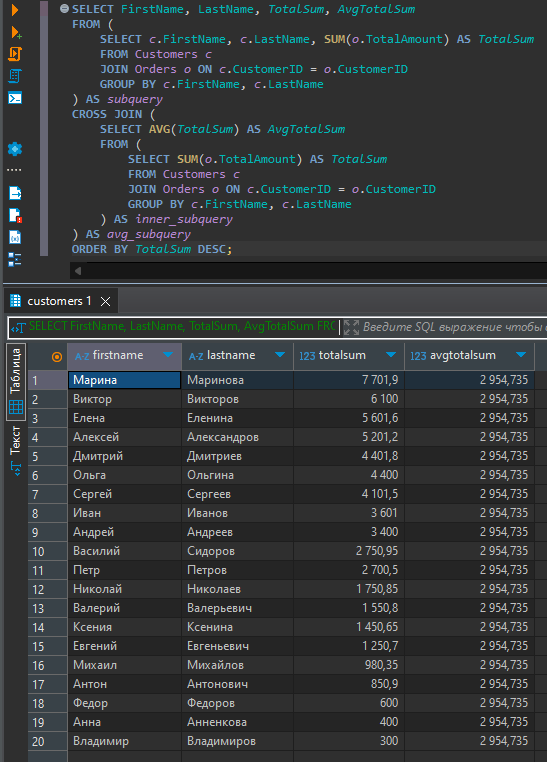


Рисунок 8 – Отображение таблицы с новым столбцом

После чего, создадим SQL-запрос, который будет выводить клиента с наибольшей суммарной стоимостью заказов (имя, фамилия, стоимость):

WITH RankedClients AS (

    SELECT c.FirstName, c.LastName, SUM(o.TotalAmount) AS TotalSum,

           RANK() OVER (ORDER BY SUM(o.TotalAmount) DESC) AS rank

    FROM Customers c

    JOIN Orders o ON c.CustomerID = o.CustomerID

    GROUP BY c.FirstName, c.LastName

)

SELECT \*

FROM RankedClients

WHERE rank = 1;

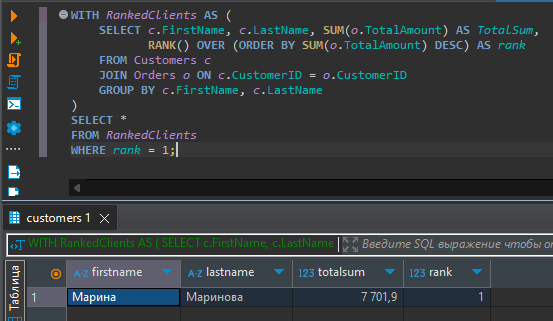


Рисунок 9 – Отображение клиента, с наибольшей суммарной стоимостью заказов

Так же, для удобства, для каждого клиента, с наибольшей суммарной стоимостью заказов выведем список его заказов (номер заказа и стоимость) в порядке возрастания стоимости заказа:

WITH TopCustomers AS (

    SELECT

        c.CustomerID,

        c.FirstName,

        c.LastName,

        SUM(o.TotalAmount) AS TotalOrderValue,

        RANK() OVER (ORDER BY SUM(o.TotalAmount) DESC) AS Rank

    FROM

        Customers c

    JOIN

        Orders o ON c.CustomerID = o.CustomerID

    GROUP BY

        c.CustomerID, c.FirstName, c.LastName

)

SELECT

    tc.FirstName,

    tc.LastName,

    o.OrderID,

    o.TotalAmount

FROM

    TopCustomers tc

JOIN

    Orders o ON tc.CustomerID = o.CustomerID

WHERE

    tc.Rank = 1

ORDER BY

    o.TotalAmount ASC;

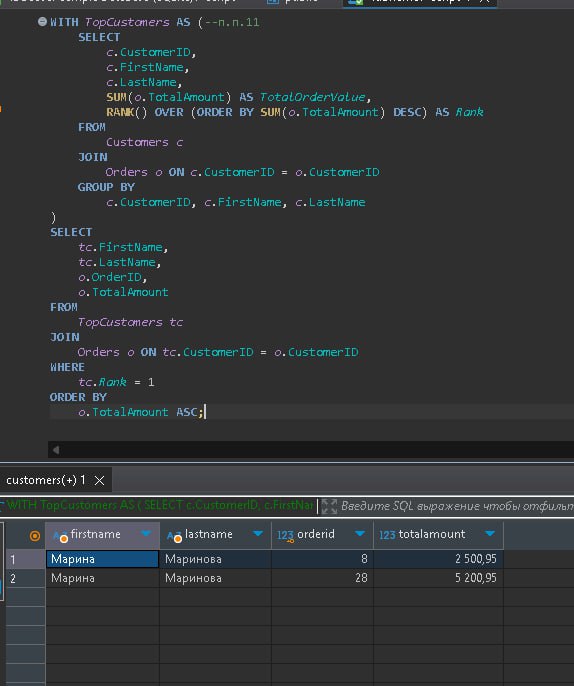


Рисунок 10 – Отображение клиента с наибольшей суммарной стоимостью заказов, по возрастанию стоимости заказа

Далее выведем только тех клиентов, у которых суммарная стоимость заказов превышает среднюю суммарную стоимость заказов клиентов:

WITH ClientTotals AS (

SELECT c.FirstName, c.LastName, SUM(o.TotalAmount) AS TotalSum

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.FirstName, c.LastName

),

AvgTotal AS (

SELECT AVG(TotalSum) AS AvgTotalSum

FROM ClientTotals

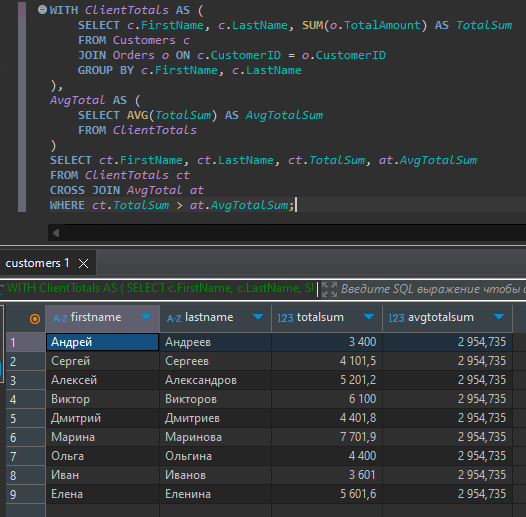
)

SELECT ct.FirstName, ct.LastName, ct.TotalSum, at.AvgTotalSum

FROM ClientTotals ct

CROSS JOIN AvgTotal at

WHERE ct. TotalSum > at.AvgTotalSum;

  
Рисунок 11 – Отображение клиентов, у которых суммарная стоимость заказов превышает среднюю суммарную стоимость заказов клиентов

Таким образом, мы на своих ошибках, с помощью тренировочной задачи, научились составлять запросы, для разных выводов данных, нашей медицинской организации.

Давайте применим, полученные знания, на нашем индивидуальном задании. Начнем, с формирования SQL-запросов, для формирования таблиц:

CREATE TABLE врачи (

    id\_врача SERIAL PRIMARY KEY,

    фамилия VARCHAR(50) NOT NULL,

    имя VARCHAR(50),

    отчество VARCHAR(50),

    специальность VARCHAR(100) NOT NULL,

    стоимость\_приема DECIMAL(10, 2) NOT NULL,

    процент\_отчислений DECIMAL(5, 2) NOT NULL CHECK (процент\_отчислений BETWEEN 0 AND 100)

);

CREATE TABLE пациенты (

    id\_пациента SERIAL PRIMARY KEY,

    фамилия VARCHAR(50) NOT NULL,

    имя VARCHAR(50),

    отчество VARCHAR(50),

    дата\_рождения DATE,

    адрес TEXT

);

CREATE TABLE приемы (

    id\_приема SERIAL PRIMARY KEY,

    id\_врача INT REFERENCES врачи(id\_врача),

    id\_пациента INT REFERENCES пациенты(id\_пациента),

    дата\_приема DATE NOT NULL,

    стоимость\_приема DECIMAL(10, 2) NOT NULL

);

CREATE TABLE платежи (

    id\_платежа SERIAL PRIMARY KEY,

    id\_приема INT REFERENCES приемы(id\_приема),

    сумма DECIMAL(10, 2) NOT NULL,

    дата\_оплаты TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE зарплаты\_врачей (

    id\_начисления SERIAL PRIMARY KEY,

    id\_врача INT REFERENCES врачи(id\_врача),

    id\_приема INT REFERENCES приемы(id\_приема),

    брутто\_зарплата DECIMAL(10, 2),

    нетто\_зарплата DECIMAL(10, 2)

);

CREATE OR REPLACE FUNCTION calculate\_salary() RETURNS TRIGGER AS $$

BEGIN

    NEW.брутто\_зарплата := (SELECT стоимость\_приема FROM приемы WHERE id\_приема = NEW.id\_приема) \* (SELECT процент\_отчислений FROM врачи WHERE id\_врача = NEW.id\_врача) / 100;

    NEW.нетто\_зарплата := NEW.брутто\_зарплата - NEW.брутто\_зарплата \* 0.13;

    RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trg\_calculate\_salary

BEFORE INSERT ON зарплаты\_врачей

FOR EACH ROW

EXECUTE FUNCTION calculate\_salary();

На рисунке 12, представлена диаграмма, которая у нас получилась, после запуска данных SQL-запросов.

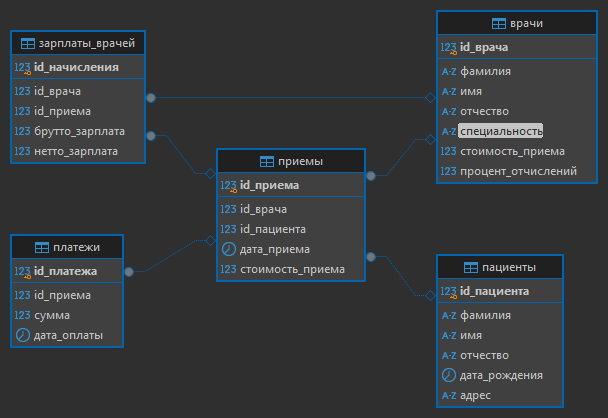


Рисунок 12 – Диаграмма физического отображения, созданной базы данных

Начнем наполнять нашу базу данных, данными.

Начнем с таблицы «Врачи»:

INSERT INTO врачи (id\_врача, фамилия, имя, отчество, специальность, стоимость\_приема, процент\_отчислений)

VALUES

(1, 'Иванов', 'Иван', 'Иванович', 'Терапевт', 1500.00, 15.00),

(2, 'Петров', 'Петр', 'Петрович', 'Хирург', 2000.00, 25.00),

(3, 'Сидоров', 'Сергей', 'Сергеевич', 'Кардиолог', 2500.00, 30.00),

(4, 'Кузнецова', 'Анна', 'Владимировна', 'Невролог', 1800.00, 18.00),

(5, 'Орлова', 'Ольга', 'Николаевна', 'Офтальмолог', 1700.00, 17.00),

(6, 'Васильев', 'Алексей', 'Викторович', 'Ортопед', 1900.00, 19.00),

(7, 'Федорова', 'Елена', 'Павловна', 'Эндокринолог', 2100.00, 21.00),

(8,  'Александров', 'Андрей', 'Андреевич', 'Уролог', 2200.00, 22.00),

(9, 'Захарова', 'Марина', 'Геннадьевна', 'Дерматолог', 1600.00, 16.00),

(10, 'Комаров', 'Дмитрий', 'Валерьевич', 'Психиатр', 2300.00, 23.00),

(11, 'Никитин', 'Николай', 'Александрович', 'Онколог', 2400.00, 24.00),

(12, 'Попова', 'Светлана', 'Борисовна', 'Педиатр', 1400.00, 14.00),

(13, 'Романова', 'Наталья', 'Вячеславовна', 'Гастроэнтеролог', 1950.00, 19.50),

(14, 'Семенов', 'Максим', 'Игоревич', 'Отоларинголог', 1850.00, 18.50),

(15, 'Тимофеева', 'Инна', 'Константиновна', 'Рентгенолог', 1750.00, 17.50),

(16, 'Устинова', 'Дарья', 'Артемовна', 'Гинеколог', 2050.00, 20.50),

(17, 'Филатова', 'Татьяна', 'Михайловна', 'Аллерголог', 1650.00, 16.50),

(18, 'Чернова', 'Екатерина', 'Георгиевна', 'Реабилитолог', 1550.00, 15.50),

(19, 'Шишкин', 'Александр', 'Витальевич', 'Физиотерапевт', 1450.00, 14.50),

(20, 'Яковлев', 'Владимир', 'Евгеньевич', 'Диетолог', 1350.00, 13.50);

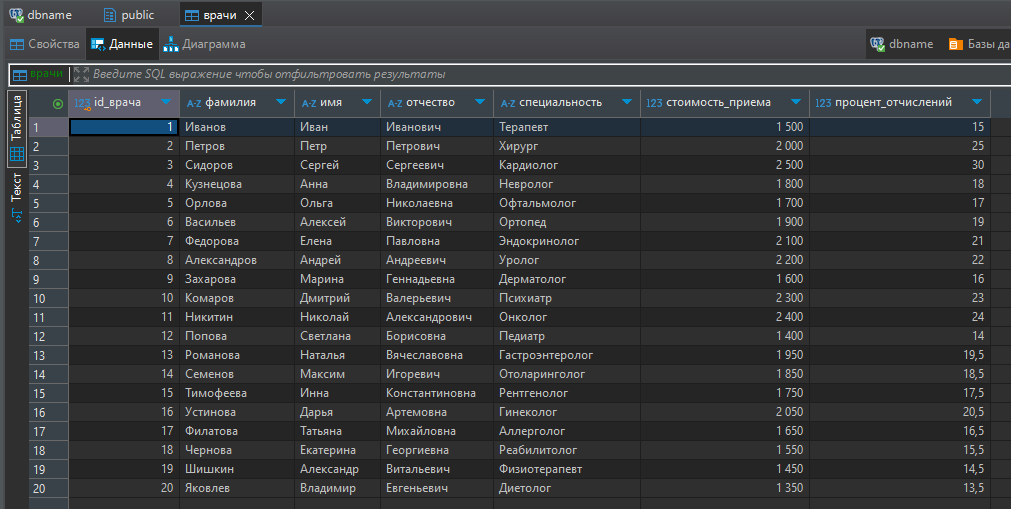


Рисунок 13 – Отображение данных, в сущности, «Врачи»

Как можно заметить из рисунка 13, все данные корректно отображаются в своих строчках и столбцах.

Далее перейдем к заполнению таблицы «Пациенты»:

INSERT INTO пациенты (id\_пациента, фамилия, имя, отчество, дата\_рождения, адрес)

VALUES

(1, 'Иванов', 'Иван', 'Иванович', '1980-01-05', 'Москва, ул. Тверская, д. 12'),

(2, 'Петров', 'Петр', 'Петрович', '1990-02-10', 'Санкт-Петербург, пр-т Невский, д. 34'),

(3, 'Сидоров', 'Сергей', 'Сергеевич', '1975-03-15', 'Новосибирск, ул. Красный проспект, д. 56'),

(4, 'Кузнецова', 'Анна', 'Владимировна', '1985-04-20', 'Екатеринбург, ул. Малышева, д. 78'),

(5, 'Орлова', 'Ольга', 'Александровна', '1995-05-25', 'Казань, ул. Баумана, д. 9'),

(6, 'Васильев', 'Алексей', 'Николаевич', '1960-06-30', 'Нижний Новгород, ул. Большая Покровская, д. 21'),

(7, 'Федорова', 'Марина', 'Павловна', '1970-07-14', 'Самара, ул. Молодогвардейская, д. 43'),

(8, 'Комаров', 'Дмитрий', 'Валерьевич', '1986-08-27', 'Омск, ул. Маркса, д. 65'),

(9, 'Зайцева', 'Елена', 'Викторовна', '1991-09-11', 'Ростов-на-Дону, ул. Пушкинская, д. 87'),

(10, 'Романова', 'Светлана', 'Георгиевна', '1965-10-19', 'Краснодар, ул. Красная, д. 32'),

(11, 'Леонова', 'Наталья', 'Игоревна', '1983-11-22', 'Челябинск, пр-т Ленина, д. 54'),

(12, 'Миронов', 'Михаил', 'Андреевич', '1993-12-03', 'Волгоград, ул. Мира, д. 76'),

(13, 'Соколова', 'Татьяна', 'Евгеньевна', '1955-01-17', 'Пермь, ул. Ленина, д. 98'),

(14, 'Семенов', 'Владимир', 'Борисович', '1940-02-29', 'Воронеж, ул. Карла Маркса, д. 111'),

(15, 'Павлова', 'Инна', 'Геннадьевна', '1978-03-28', 'Красноярск, ул. Мира, д. 123'),

(16, 'Григорьева', 'Юлия', 'Антоновна', '1997-04-16', 'Саратов, ул. Советская, д. 135'),

(17, 'Попов', 'Александр', 'Владимирович', '1989-05-26', 'Тюмень, ул. Республики, д. 147'),

(18, 'Максимова', 'Оксана', 'Анатольевна', '1967-06-07', 'Барнаул, ул. Ленина, д. 159'),

(19, 'Степанов', 'Артем', 'Михайлович', '1999-07-18', 'Ижевск, ул. Пушкинская, д. 171'),

(20, 'Ковалева', 'Полина', 'Денисовна', '1982-08-29', 'Иркутск, ул. Байкальская, д. 183');

Проверим корректность написанных данных, в нашей сущности «Пациенты» (рисунок 14).

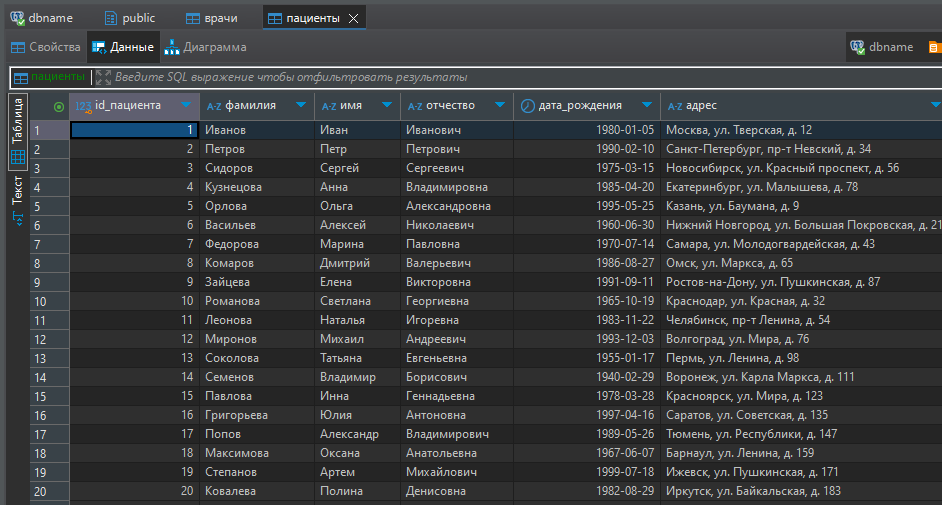


Рисунок 14 – Отображение данных, в сущности, «Пациенты»

Заполним таким же образом остальные оставшиеся сущности.

Сущность «Приемы»:

INSERT INTO приемы (id\_врача, id\_пациента, дата\_приема, стоимость\_приема) VALUES

(1, 15, '2023-10-26', 1500.00),

(2, 3, '2023-10-27', 2000.00),

(3, 7, '2023-10-28', 2500.00),

(4, 12, '2023-10-29', 1800.00),

(5, 14, '2023-10-30', 1700.00),

(6, 8, '2023-10-31', 1900.00),

(7, 19, '2023-11-01', 2100.00),

(8, 4, '2023-11-02', 2200.00),

(9, 2, '2023-11-03', 1600.00),

(10, 12, '2023-11-04', 2300.00),

(11, 13, '2023-11-05', 2400.00),

(12, 9, '2023-11-06', 1400.00),

(13, 17, '2023-11-07', 1950.00),

(14, 10, '2023-11-08', 1850.00),

(15, 6, '2023-11-09', 1750.00),

(16, 5, '2023-11-10', 2050.00),

(17, 1, '2023-11-11', 1650.00),

(18, 10, '2023-11-12', 1550.00),

(19, 5, '2023-11-13', 1450.00),

(20, 11, '2023-11-14', 1350.00),

(1, 20, '2023-11-15', 1500.00),

(2, 14, '2023-11-16', 2000.00),

(3, 8, '2023-11-17', 2500.00),

(4, 3, '2023-11-18', 1800.00),

(5, 16, '2023-11-19', 1700.00);

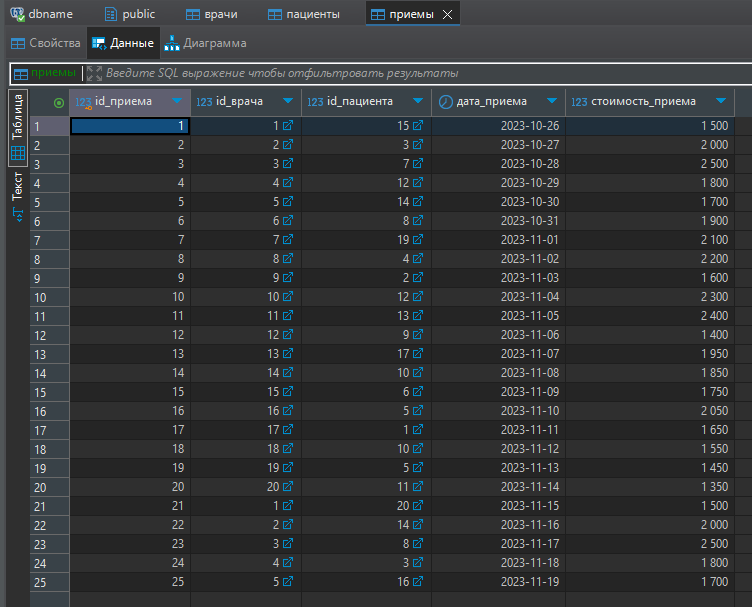


Рисунок 15 – Отображение данных, в сущности, «Приемы»

Данные, для сущности «Платежи»:

INSERT INTO платежи (id\_приема, сумма, дата\_оплаты) VALUES

(1, 1500.00, '2023-10-01 08:00:00'),

(2, 2000.00, '2023-10-02 08:15:00'),

(3, 2500.00, '2023-10-03 08:30:00'),

(4, 1800.00, '2023-10-04 08:45:00'),

(5, 1700.00, '2023-10-05 09:00:00'),

(6, 1900.00, '2023-10-06 09:15:00'),

(7, 2100.00, '2023-10-07 09:30:00'),

(8, 2200.00, '2023-10-08 09:45:00'),

(9, 1600.00, '2023-10-09 10:00:00'),

(10, 2300.00, '2023-10-10 10:15:00'),

(11, 2400.00, '2023-10-11 10:30:00'),

(12, 1400.00, '2023-10-12 10:45:00'),

(13, 1950.00, '2023-10-13 11:00:00'),

(14, 1850.00, '2023-10-14 11:15:00'),

(15, 1750.00, '2023-10-15 11:30:00'),

(16, 2050.00, '2023-10-16 11:45:00'),

(17, 1650.00, '2023-10-17 12:00:00'),

(18, 1550.00, '2023-10-18 12:15:00'),

(19, 1450.00, '2023-10-19 12:30:00'),

(20, 1350.00, '2023-10-20 12:45:00'),

(21, 1500.00, '2023-10-21 13:00:00'),

(22, 2000.00, '2023-10-22 13:15:00'),

(23, 2500.00, '2023-10-23 13:30:00'),

(24, 1800.00, '2023-10-24 13:45:00'),

(25, 1700.00, '2023-10-25 14:00:00');

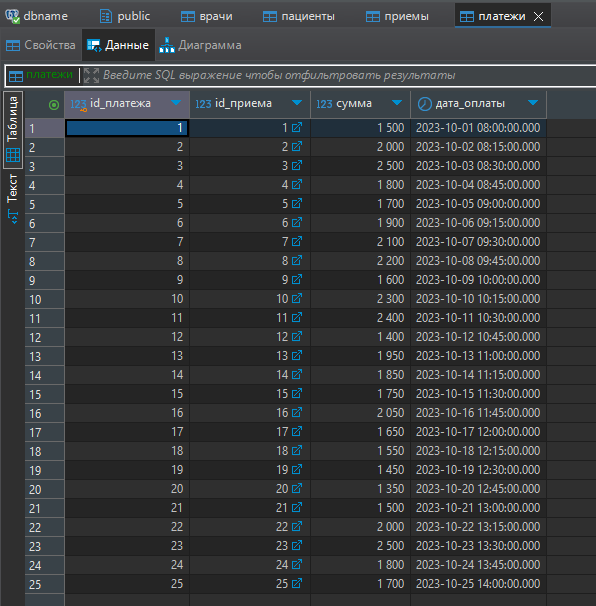


Рисунок 16 – Отображение данных, в сущности, «Платежи»

Сущность «Зарплаты врачей»:

INSERT INTO зарплаты\_врачей (id\_врача, id\_приема) VALUES

(1, 1),

(2, 2),

(3, 3),

(4, 4),

(5, 5),

(6, 6),

(7, 7),

(8, 8),

(9, 9),

(10, 10),

(11, 11),

(12, 12),

(13, 13),

(14, 14),

(15, 15),

(16, 16),

(17, 17),

(18, 18),

(19, 19),

(20, 20),

(1, 21),

(2, 22),

(3, 23),

(4, 24),

(5, 25);

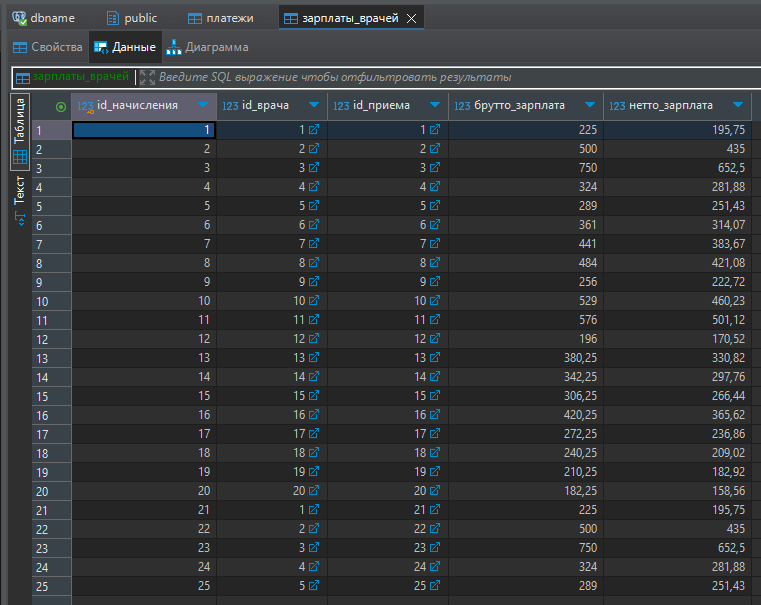


Рисунок 17 – Отображение данных, в сущности, «Зарплаты врачей»

Аналогично тренировочной задачи, пропишем SQL-запрос, который возвращает список пациентов (имя, фамилия, отчество) и суммарную стоимость приемов каждого пациента.

SELECT

    v.фамилия,

    v.имя,

    v.отчество,

    SUM(p.стоимость\_приема) AS общая\_стоимость\_приемов

FROM

    врачи v

JOIN

    приемы p ON v.id\_врача = p.id\_врача

GROUP BY

    v.фамилия, v.имя, v.отчество, v.id\_врача;

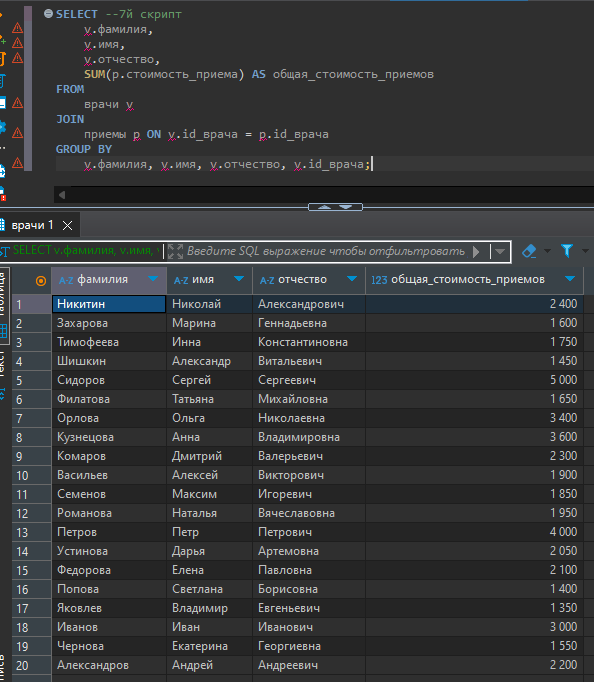


Рисунок 18 – Вывод пациентов, с суммарной стоимостью приемов

Теперь, создадим SQL-запрос, который отсортирует полученный список по убыванию суммарной стоимости приемов пациента (рисунок 19).

SELECT --8й скрипт

    v.фамилия,

    v.имя,

    v.отчество,

    SUM(p.стоимость\_приема) AS общая\_стоимость\_приемов

FROM

    врачи v

JOIN

    приемы p ON v.id\_врача = p.id\_врача

GROUP BY

    v.фамилия, v.имя, v.отчество, v.id\_врача

ORDER BY

    SUM(p.стоимость\_приема) DESC;

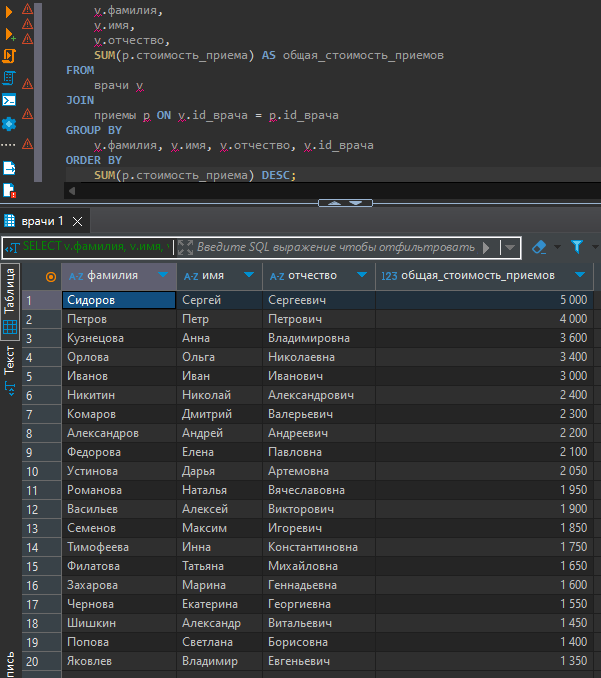


Рисунок 19 – Вывод пациентов, с суммарной стоимостью приемов по убыванию

В данный запрос, мы добавили сортировку по стоимости приема (ORDER BY), которая происходит в порядке убывания (DESC). Этот запрос возвращает идентификаторы пациентов (id), их фамилии и имена, а также общую стоимость приемов. Сортировка по убыванию общей стоимости позволяет получить пациентов, у которых общая стоимость приемов самая высокая, сверху вниз.

Далее добавим столбец со средней суммарной стоимостью приемов:

SELECT

    v.фамилия,

    v.имя,

    v.отчество,

  ROUND(SUM(p.стоимость\_приема)::NUMERIC, 2) AS общая\_стоимость\_приемов,

    ROUND(AVG(SUM(p.стоимость\_приема)) OVER (), 2) AS средняя\_общая\_стоимость\_приемов

FROM

    врачи v

JOIN

    приемы p ON v.id\_врача = p.id\_врача

GROUP BY

    v.фамилия, v.имя, v.отчество, v.id\_врача

ORDER BY

    SUM(p.стоимость\_приема) DESC;

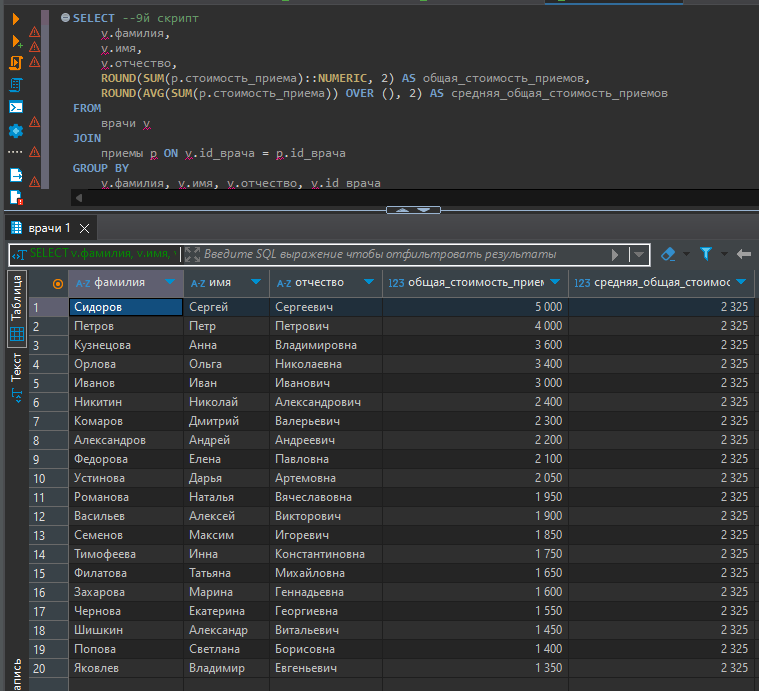


Рисунок 20 – Добавление столбца средней суммарной стоимостью приемов

Научимся выводить одного определенного пациента, у которого будет наибольшая суммарная стоимость приема:

WITH patient\_totals AS (--10й скрипт

    SELECT

        п.фамилия,

        п.имя,

        SUM(пр.стоимость\_приема) AS общая\_стоимость\_приемов

    FROM

        пациенты п

    JOIN

        приемы пр ON п.id\_пациента = пр.id\_пациента

    GROUP BY

        п.фамилия, п.имя, п.id\_пациента

)

SELECT \*

FROM patient\_totals

ORDER BY общая\_стоимость\_приемов DESC

LIMIT 1;

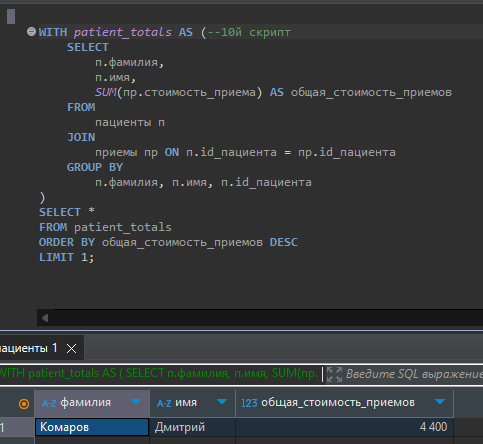


Рисунок 21 – Вывод пациента с наибольшей суммарной стоимостью заказов

Сортируем результаты по общей стоимости приемов в порядке убывания (ORDER BY). И ограничиваем вывод первой записи с помощью оператора LIMIT.

Теперь пациента с наибольшей суммарной стоимостью приемов вывести список его приемов (номер приема и стоимость) в порядке возрастания стоимости проема.

WITH top\_patient AS (--11й скрипт

    SELECT

        p.фамилия,

        p.имя,

        p.отчество,

        p.id\_пациента

    FROM

        пациенты p

    INNER JOIN

        приемы pr ON p.id\_пациента = pr.id\_пациента

    GROUP BY

        p.фамилия, p.имя, p.отчество, p.id\_пациента

    ORDER BY

        SUM(pr.стоимость\_приема) DESC

    LIMIT 1

)

SELECT

    pr.id\_приема,

    pr.стоимость\_приема

FROM

    приемы pr

INNER JOIN

    top\_patient tp ON pr.id\_пациента = tp.id\_пациента

ORDER BY

    pr.стоимость\_приема ASC;

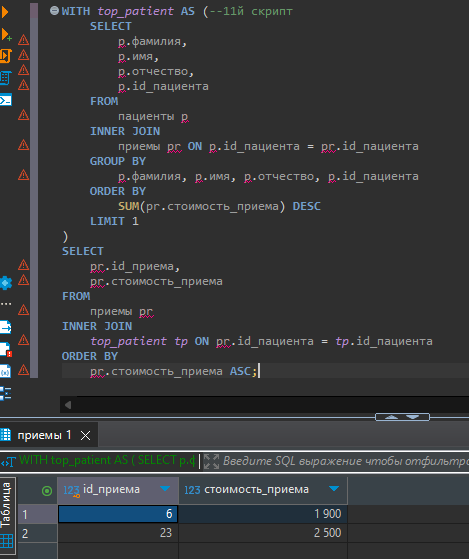


Рисунок 22 – Отображение стоимости приемов пациента, в порядке возрастания стоимости

Как видно на рисунке выше, отображается номер приема и его стоимость, в порядке возрастания цены. В подзапросе top\_patient выбираются фамилия, имя, отчество и идентификатор пациента, который потратил больше всего денег на приёмы. Для этого используется агрегация по стоимости всех приёмов каждого пациента (SUM(pr.стоимость\_приема)), а затем результат сортируется по убыванию суммы затрат и ограничивается одним пациентом с максимальной суммой. Основной запрос выбирает все записи о приёмах выбранного пациента, включая их стоимость, отсортированные по возрастанию стоимости приёма.

WITH patient\_totals AS (--12 скрипт

    SELECT

        п.фамилия,

        п.имя,

        SUM(пр.стоимость\_приема) AS общая\_стоимость\_приемов

    FROM

        пациенты п

    JOIN

        приемы пр ON п.id\_пациента = пр.id\_пациента

    GROUP BY

        п.фамилия, п.имя, п.id\_пациента

),

avg\_cost AS (

    SELECT

        ROUND(AVG(общая\_стоимость\_приемов)) AS средняя\_стоимость\_приемов

    FROM

        patient\_totals

)

SELECT

    п.фамилия,

    п.имя,

    п.общая\_стоимость\_приемов,

    ac.средняя\_стоимость\_приемов

FROM

    patient\_totals п

CROSS JOIN

    avg\_cost ac

WHERE

    п.общая\_стоимость\_приемов > ac.средняя\_стоимость\_приемов;

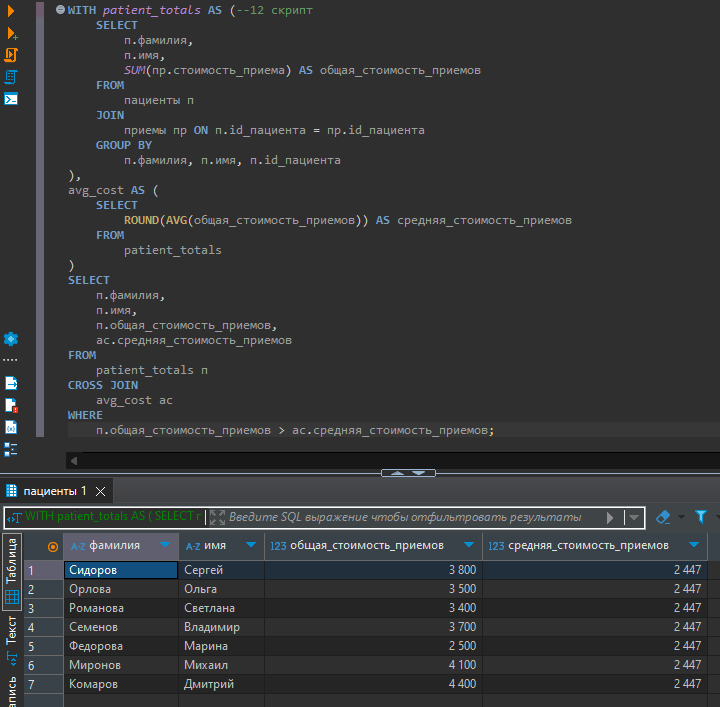


Рисунок 23 – Отображение вывода пациентов, с средней стоимостью приемов

В первом подзапросе patient\_totals происходит группировка данных по каждому пациенту (по фамилии, имени и идентификатору). Затем рассчитывается сумма затрат на все его приёмы (SUM(пр.стоимость\_приема)). Во втором подзапросе avg\_cost вычисляется среднее значение общих затрат на приёмы среди всех пациентов (AVG(общая\_стоимость\_приемов)). Результат округляется до целого числа. Основной запрос объединяет результаты обоих подзапросов через операцию CROSS JOIN, чтобы сопоставить среднюю стоимость приёмов со стоимостью каждого пациента. Затем фильтруются те пациенты, у которых общая стоимость приёмов превышает среднюю по базе данных. В итоге запрос возвращает список пациентов, чья общая стоимость приёмов выше среднего значения по всем пациентам, вместе с информацией об их затратах и среднем значении.

Вывод: таким образом, мы изучили синтаксис и основные команды для формирования запросов к базе данных, освоить процесс формирования SQL-запросов. Так же научились разрабатывать структуру базы данных в СУБД Postgres, наполнять ее данными, научились писать запросы к базе данных для отработки навыка формирования SQL-запросов. Поняли, как ставить задачи для выборки необходимых данных из базы данных и решать её с помощью SQL-запросов.

**Контрольные вопросы:**

1. Подзапрос — это SQL-запрос, который содержится внутри другого SQL-запроса. Он используется для получения данных, которые затем используются в основном запросе. Подзапросы могут использоваться для фильтрации данных, агрегирования данных, соединения таблиц и т.д.
2. Для заполнения таблицы данными можно использовать команды INSERT, UPDATE и DELETE. INSERT используется для добавления новых строк в таблицу, UPDATE используется для изменения существующих строк, а DELETE используется для удаления строк из таблицы.
3. Агрегатные функции — это функции, которые применяются к группе строк для получения сводных значений. Примеры агрегатных функций включают SUM, AVG, COUNT, MIN и MAX. Они используются для подсчета количества строк, суммирования значений, нахождения среднего значения, минимального и максимального значений в группе строк.
4. Для объединения таблиц в запросе используется оператор JOIN. Существуют различные типы join'ов, такие как INNER JOIN, LEFT JOIN, RIGHT JOIN и FULL OUTER JOIN. Каждый из них определяет, какие строки будут выбраны из каждой таблицы и как они будут объединены.
5. Способы объединения таблиц включают использование различных типов JOIN'ов, таких как INNER JOIN, LEFT JOIN, RIGHT JOIN и FULL OUTER JOIN. INNER JOIN выбирает строки из обеих таблиц, где совпадают условия соединения. LEFT JOIN выбирает все строки из левой таблицы и соответствующие строки из правой таблицы. RIGHT JOIN аналогичен LEFT JOIN, но наоборот. FULL OUTER JOIN выбирает все строки из обеих таблиц.
6. План запроса — это последовательность операций, которую выполняет СУБД для выполнения запроса. Планировщик запросов оптимизирует план запроса, чтобы минимизировать время выполнения и использовать ресурсы эффективно.
7. Оптимизация запроса включает анализ плана запроса и изменение его для улучшения производительности. Это может включать изменение индексов, добавление или удаление условий WHERE, изменение порядка сортировки и т.д.
8. Оконные функции — это функции, которые работают с подгруппами строк внутри одного результирующего множества. Они используются для анализа данных в контексте окна, которое определяется пользователем.
9. Для группировки данных используется ключевое слово GROUP BY. Оно позволяет агрегировать данные по определенным столбцам и применять агрегатные функции к этим группам.
10. Сортировка данных осуществляется с использованием ключевого слова ORDER BY. Оно позволяет упорядочивать результаты запроса по одному или нескольким столбцам.
11. Условия для выборки данных (WHERE, HAVING) используются для фильтрации строк перед применением агрегатных функций. WHERE применяется для фильтрации строк перед группировкой, а HAVING — для фильтрации групп после применения агрегатных функций.