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 №8

По дисциплине: «Системы хранения и обработки данных»

Тема: «Разработка физической структуры базы данных»

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

подпись, дата

Принял: Короленко В.В.

подпись, дата

Воронеж 2024

Цель работы: изучить и освоить процесс разработки физической структуры базы данных с использованием системы управления базами данных Postgres.

Основные задачи:

• создание таблиц в СУБД Postgres с помощью SQL-запросов с атрибутами, связями, первичными и внешними ключами в соответствии с разработанной в предыдущей лабораторной работе логической структурой базы данных;

• построение графической диаграммы для отображения физической структуры базы данных.

Ход работы:

PostgreSQL — это мощная объектно-реляционная система управления базами данных (ОРСУБД), широко используемая благодаря своей надежности, гибкости и открытости. Вот основные аспекты, которые стоит изучить:

1. **Установка и настройка PostgreSQL**:
   * Установка PostgreSQL на вашу операционную систему (Windows, Linux, macOS).
   * Создание пользователей и баз данных.
   * Настройка доступа и прав пользователей.
2. **Основы SQL-запросов**:
   * SELECT, INSERT, UPDATE, DELETE — базовые операции с данными.
   * Использование JOIN для объединения таблиц.
   * Агрегационные функции (COUNT, SUM, AVG и др.).
   * Группировка данных (GROUP BY, HAVING).
   * Работа с подзапросами и CTE (Common Table Expressions).
3. **Создание и изменение таблиц**:
   * CREATE TABLE — создание новой таблицы.
   * ALTER TABLE — изменение существующей таблицы (добавление/удаление колонок, изменение типов данных и т.п.).
   * CONSTRAINTS — ограничения целостности данных (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK и др.).
4. **Индексы**:
   * Понимание необходимости индексов для ускорения запросов.
   * Создание и использование индексов (CREATE INDEX).
5. **Транзакции**:
   * Понятие транзакций и их важность для обеспечения целостности данных.
   * BEGIN TRANSACTION, COMMIT, ROLLBACK — работа с транзакциями.
6. **Резервное копирование и восстановление**:
   * pg\_dump и pg\_restore — инструменты для резервного копирования и восстановления баз данных.
7. **Оптимизация запросов**:
   * EXPLAIN и EXPLAIN ANALYZE — анализ производительности запросов.
   * Оптимизация структуры базы данных и запросов для повышения скорости выполнения.

**Изучение инструментария для работы с СУБД (DBeaver или аналог)**

DBeaver — это мощный инструмент для работы с различными СУБД, включая PostgreSQL. Он предоставляет удобный графический интерфейс для выполнения большинства задач администрирования и разработки баз данных. Вот основные моменты, которые следует изучить:

1. **Подключение к базе данных**:
   * Создание нового подключения к PostgreSQL-серверу.
   * Управление подключениями и сохранение настроек.
2. **Работа с объектами базы данных**:
   * Просмотр списка таблиц, представлений, функций и других объектов.
   * Редактирование свойств объектов (например, изменение структуры таблицы).
3. **Выполнение SQL-запросов**:
   * Открытие редактора SQL-запросов.
   * Выполнение запросов и просмотр результатов.
   * Сохранение и повторное выполнение запросов.
4. **Импорт и экспорт данных**:
   * Импорт данных из файлов CSV, Excel и других форматов.
   * Экспорт данных в различные форматы.
5. **Администрирование базы данных**:
   * Управление пользователями и ролями.
   * Мониторинг активности сервера и производительности.
   * Резервное копирование и восстановление баз данных.
6. **Отладка и профилирование**:
   * Анализ плана выполнения запросов.
   * Профилирование запросов для выявления узких мест.

Изучив эти аспекты, мы теперь можем эффективно работать с PostgreSQL и использовать DBeaver для выполнения лабораторной работы.

Сформируем SQL-запросы для создания таблиц в СУБД СУБД Postgres, соответствующих логической структуре базы данных, разработанной в предыдущей лабораторной работе. Для этого будем использовать такую программу, как Visual Studio Code.

Создаем SQL-запрос для создания таблицы «Врачи»:

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)

);

Создаем SQL-запрос для создания таблицы «Пациенты»:

CREATE TABLE пациенты (

    id\_пациента SERIAL PRIMARY KEY,

    фамилия VARCHAR(50) NOT NULL,

    имя VARCHAR(50),

    отчество VARCHAR(50),

    дата\_рождения DATE,

    адрес TEXT

);

Создаем SQL-запрос для создания таблицы «Приемы»:

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

);

Создаем SQL-запрос для создания таблицы «Платежи»:

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

);

Создаем SQL-запрос для создания таблицы «Зарплаты врачей»:

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)

);

После того, как мы создали SQL-запросы, сохраняем наш файл в расширении .sql. Займемся разбором каждого запроса отдельно.

1. Таблица «Врачи»

Запрос:

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)

);

Этот запрос создает таблицу «Врачи», которая будет содержать информацию о врачах, а так же просчитывает процента отчислений врачу с каждого приема.

Составляющие запроса:

**id\_врача SERIAL PRIMARY KEY**: Это уникальный идентификатор врача, который автоматически увеличивается при добавлении новой записи (SERIAL означает автоинкремент). PRIMARY KEY указывает, что этот столбец является первичным ключом таблицы.

**фамилия VARCHAR(50) NOT NULL**: Поле для хранения фамилии врача. Тип данных VARCHAR(50) ограничивает длину строки до 50 символов. NOT NULL требует обязательного заполнения этого поля.

**имя VARCHAR(50)**: Поле для имени врача. Ограничение длины — 50 символов. В отличие от предыдущего поля, оно может быть пустым.

**отчество VARCHAR(50)**: Поле для отчества врача. Также ограничено длиной в 50 символов и может быть пустым.

**специальность VARCHAR(100) NOT NULL**: Поле для специальности врача. Ограничено длиной в 100 символов и обязательно к заполнению.

**стоимость\_приема DECIMAL(10, 2) NOT NULL**: Поле для стоимости приема у данного врача. Тип данных DECIMAL(10, 2) означает, что число может иметь максимум 8 цифр перед десятичной точкой и 2 цифры после нее. Поле также обязательно к заполнению.

**процент\_отчислений DECIMAL(5, 2) NOT NULL CHECK (процент\_отчислений BETWEEN 0 AND 100)**: Поле для процента отчислений врачу с каждого приема. имеет тип данных DECIMAL(5, 2), что означает, что максимальное количество цифр до десятичного знака равно 3, а после него — 2. Условие NOT NULL говорит о том, что это поле должно всегда содержать какое-то значение, то есть оно не может оставаться пустым. После чего идет проверка условия (CHECK) на диапазон значений. Условие BETWEEN 0 AND 100 проверяет, чтобы введенное значение находилось в пределах от 0 до 100 включительно. Если пользователь попытается ввести значение меньше 0 или больше 100, база данных выдаст ошибку и не позволит сохранить запись. Таким образом, это поле предназначено для хранения процентов отчислений врачу, которые должны находиться в диапазоне от 0% до 100%.

2. Таблица «Пациенты»

Запрос:

CREATE TABLE пациенты (

id\_пациента SERIAL PRIMARY KEY,

фамилия VARCHAR(50) NOT NULL,

имя VARCHAR(50),

отчество VARCHAR(50),

дата\_рождения DATE,

адрес TEXT

);

Этот запрос создает таблицу пациенты, содержащую информацию о пациентах, а точнее, их персональные данные.

Составляющие запроса:

**id\_пациента SERIAL PRIMARY KEY**: Уникальный идентификатор пациента, который автоматически увеличивается при добавлении новой записи. Является первичным ключом таблицы.

**фамилия VARCHAR(50) NOT NULL**: Поле для хранения фамилии пациента. Обязательное к заполнению.

**имя VARCHAR(50)**: Поле для имени пациента. Может быть пустым.

**отчество VARCHAR(50)**: Поле для отчества пациента. Может быть пустым.

**дата\_рождения DATE**: Поле для даты рождения пациента. Тип данных DATE используется для хранения дат.

**адрес TEXT**: Поле для адреса пациента. Тип данных TEXT позволяет хранить длинные строки текста без ограничения на количество символов.

3. Таблица «Приемы»

Запрос:

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

);

Этот запрос создает таблицу приемы, содержащую информацию о приемах врачей. Кто был записан на прием, когда был проведен и сколько стоил.

Составляющие запроса:

**id\_приема SERIAL PRIMARY KEY**: Уникальный идентификатор приема, который автоматически увеличивается при добавлении новой записи. Является первичным ключом таблицы.

**id\_врача INT REFERENCES врачи(id\_врача)**: Внешний ключ, ссылающийся на таблицу «Врачи». Этот столбец хранит идентификатор врача, проводящего прием.

**id\_пациента INT REFERENCES пациенты(id\_пациента)**: Внешний ключ, ссылающийся на таблицу «Пациенты». Этот столбец хранит идентификатор пациента, пришедшего на прием.

**дата\_приема DATE NOT NULL**: Это поле содержит дату проведения приема. Оно имеет тип данных DATE, что означает, что в нем хранится информация исключительно о дате. NOT NULL говорит о том, что данное поле обязательно должно быть заполнено при создании новой записи в таблице. База данных не позволит вставить строку, где поле дата\_приема останется пустым.

**стоимость\_приема DECIMAL(10, 2) NOT NULL**: В этом поле содержится стоимость приема врача, который вел данный прием. Тип данных DECIMAL(10, 2) означает, что число может иметь до 10 знаков в целом числе и 2 знака после десятичной точки. Атрибут NOT NULL указывает, что это поле обязательно должно быть заполнено при создании новой записи в таблице.

4. Таблица «Платежи»

Запрос:

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

);

Этот запрос создает таблицу платежи, содержащую информацию о платежах за приемы. Так же устанавливает по умолчанию дату и время

Составляющие запроса:

**id\_платежа SERIAL PRIMARY KEY**: Уникальный идентификатор платежа, который автоматически увеличивается при добавлении новой записи. Является первичным ключом таблицы.

**id\_приема INT REFERENCES приемы(id\_приема)**: Внешний ключ, ссылающийся на таблицу приемы. Этот столбец хранит идентификатор приема, за который был произведен платеж.

**сумма DECIMAL(10, 2) NOT NULL**: В этом поле содержится сумма платежа. Имеет тип данных DECIMAL(10, 2) означает, что число может иметь до 10 знаков в целом числе и 2 знака после десятичной точки. Атрибут NOT NULL указывает, что это поле обязательно должно быть заполнено при создании новой записи в таблице.

**дата\_оплаты TIMESTAMP DEFAULT CURRENT\_TIMESTAMP**:

Данное поле, имеет тип данных TIMESTAM оно хранит информацию о дате и времени вместе. По умолчанию, если явного значения не задано при вставке новой записи, система автоматически установит текущую дату и время, используя функцию CURRENT\_TIMESTAMP. Эта функция возвращает точное время и дату на момент выполнения операции.

Однако, если необходимо указать конкретную дату и время оплаты, которая отличается от текущего момента, то это значение можно изменить вручную при внесении записи в базу данных. Таким образом, хотя по умолчанию используется текущее время, администратор или пользователь может задать любое другое подходящее значение.

5. Таблица «Зарплаты врачей»

Запрос:

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)

);

Этот запрос создает таблицу зарплаты\_врачей, содержащую информацию о начислениях заработной платы врачам.

Составляющие запроса:

**id\_начисления SERIAL PRIMARY KEY**: Уникальный идентификатор начисления, который автоматически увеличивается при добавлении новой записи. Является первичным ключом таблицы.

**id\_врача INT REFERENCES врачи(id\_врача)**: Внешний ключ, ссылающийся на таблицу «Врачи». Этот столбец хранит идентификатор врача, которому начисляется зарплата.

**id\_приема INT REFERENCES приемы(id\_приема)**: Внешний ключ, ссылающийся на таблицу «Приемы». Этот столбец хранит идентификатор приема, за который производится начисление.

**брутто\_зарплата DECIMAL(10, 2)**: имеет тип данных DECIMAL(10, 2), что означает, что оно может хранить числовые значения с точностью до двух знаков после запятой. Максимальное количество цифр до десятичного знака составляет 8, а после — 2. Зарплата врача, до вычета налогов.

**нетто\_зарплата DECIMAL(10, 2)**: имеет тип данных DECIMAL(10, 2), что означает, что оно может хранить числовые значения с точностью до двух знаков после запятой. Максимальное количество цифр до десятичного знака составляет 8, а после — 2. Это зарплата после вычета налогов (13%).

Таким образом, все эти запросы создают структуру базы данных для управления информацией о врачах, пациентах, приемах, платежах и зарплатах врачей. Мы создали только непосредственно сами таблицы. Теперь займемся наполнением нашей таблицы вычислением зарплат врачей.

1. Функция calculate\_salary()

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;

Этот запрос создает или заменяет существующую функцию под названием calculate\_salary. Функция предназначена для вычисления брутто-зарплаты и нетто-зарплаты врача на основе стоимости приема и процента отчислений. Она выполняется как триггер перед вставкой новых записей в таблицу зарплаты\_врачей.

Составляющие запроса:

**CREATE OR REPLACE FUNCTION calculate\_salary()**: Создает новую функцию с именем calculate\_salary или заменяет уже существующую функцию с таким же именем.

**RETURNS TRIGGER**: Указывает, что эта функция должна возвращать результат, который затем будет использоваться триггером.

**$$ АS…$$** : Оператор начала и конца блока кода функции.

**LANGUAGE plpgsql**: Определяет язык программирования функции как PL/pgSQL (Procedural Language/PostgreSQL).

Внутренние элементы функции:

**BEGIN:** Начало основного блока кода функции

**NEW**: Специальный объект, представляющий новые данные, которые будут вставлены в таблицу. Используется для доступа к полям новой записи.

**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**;: Выражение для расчета брутто-зарплаты. Сначала выбираются “стоимости приема” из таблицы «приемы» и “процент отчислений” из таблицы «врачи», соответствующие текущей записи. Затем эти значения умножаются и делятся на 100, чтобы получить итоговую брутто-зарплату.

**NEW.нетто\_зарплата := NEW.брутто\_зарплата - NEW.брутто\_зарплата \* 0.13;**: Вычитает 13% налога из брутто-зарплаты для получения нетто-зарплаты.

**RETURN NEW**;: Возвращает обновленный объект NEW, содержащий рассчитанные значения брутто- и нетто-зарплат.

**END**: Завершение основного блока кода функции.

1. Триггер trg\_calculate\_salary

Запрос:

CREATE TRIGGER trg\_calculate\_salary

BEFORE INSERT ON зарплаты\_врачей

FOR EACH ROW

EXECUTE FUNCTION calculate\_salary();

Этот запрос создаёт триггер с именем trg\_calculate\_salary, который срабатывает перед вставкой новой записи в таблицу зарплаты\_врачей. Когда происходит попытка вставки новой записи, триггер вызывает функцию calculate\_salary, которая производит необходимые расчеты и заполняет поля брутто\_зарплата и нетто\_зарплата.

Составляющие запроса:

**CREATE TRIGGER trg\_calculate\_salary**: Создает триггер с указанным именем.

**BEFORE INSERT ON зарплаты\_врачей**: Указывает, что триггер срабатывает перед операцией вставки новой записи в таблицу зарплаты\_врачей.

**FOR EACH ROW**: Указание на то, что триггер должен срабатывать для каждой отдельной строки, которая добавляется в таблицу.

**EXECUTE FUNCTION calculate\_salary();**: Вызывает созданную ранее функцию calculate\_salary для выполнения расчетов.

Когда происходит вставка новой записи в таблицу зарплаты\_врачей, триггер trg\_calculate\_salary срабатывает перед добавлением этой записи. Он вызывает функцию calculate\_salary, которая выполняет следующие шаги:

1. Извлекает стоимость приёма из таблицы приемы и процент отчислений из таблицы врачи на основе соответствующих идентификаторов.
2. Рассчитывает брутто-зарплату как произведение стоимости приёма и процента отчислений, делённое на 100.
3. Рассчитывает нетто-зарплату, вычитая 13% налога из брутто-зарплаты.
4. Заполняет поля брутто\_зарплата и нетто\_зарплата новой записи этими рассчитанными значениями.
5. Возвращает обновлённую запись, которая затем вставляется в таблицу.

Теперь, после того, как мы создали и написали весь код целиком для нашей новой базы данных. Обернем ее в docker-контейнер, с помощью файла docker-compose.yml и подключим к DBeaver (рисунок 1).

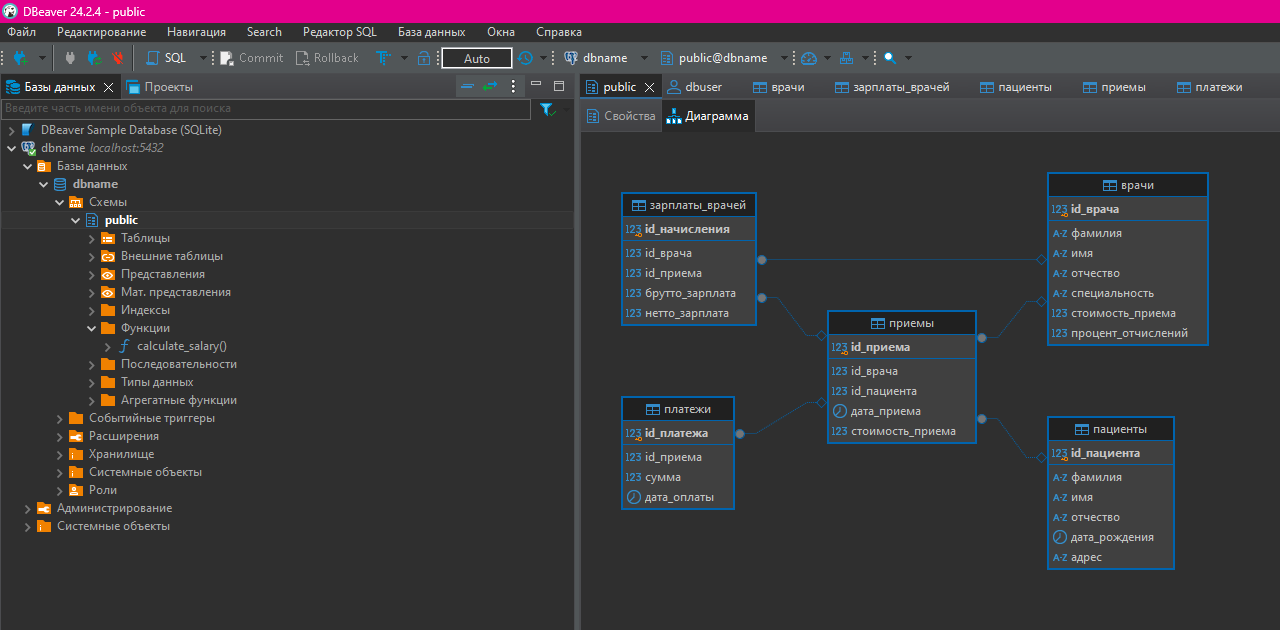


Рисунок 1 – Отображение созданной БД, через графическую диаграмму

Проверим корректность работы созданной нашей Базы данных. Как видно из рисунка 2, она корректно отображает все связные таблицы. Так же видно владельца данной БД.

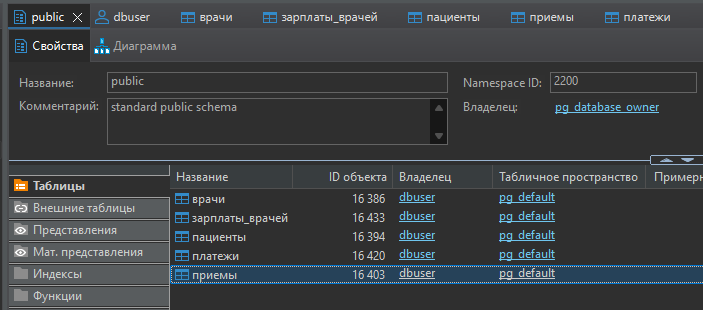


Рисунок 2 – Отображение таблиц в DBeaver

Рассмотрим подробнее нашу БД. Начнем с таблицы Врачи. Рассматривая рисунок 3, можно заметить, что тут самостоятельно можно внести данные для каждого нового врача. Видно, так же корректность введённых типов данных для каждого созданного атрибута.

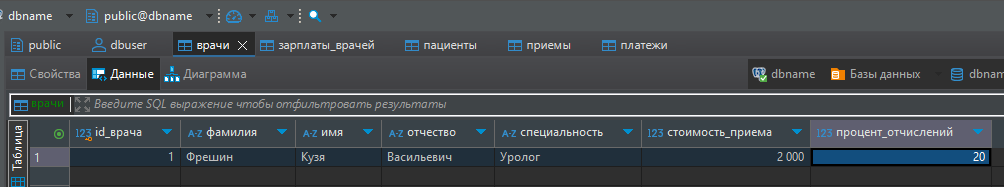


Рисунок 3 – Отображение данных таблицы «Врачи»

Тоже самое можно увидеть и в данных на рисунках 4-6.

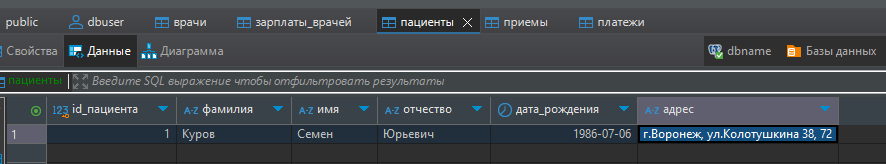


Рисунок 4 – Отображение данных таблицы «Пациенты»

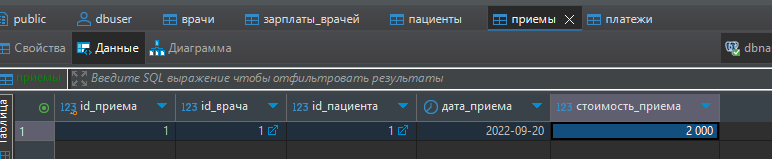


Рисунок 5 – Отображение данных таблицы «Приемы»

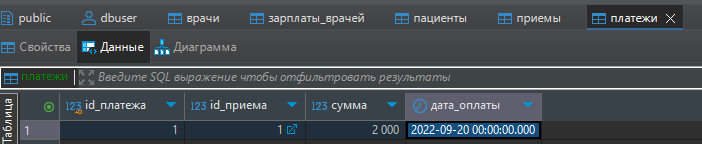


Рисунок 6 – Отображение данных таблицы «Платежи»

Рассмотрим подробнее данные из таблицы «Зарплаты врачей» (рисунок 7). Нам нет необходимости вводить данные из столбцов “брутто\_зарплата” и “нетто\_зарплата”. Когда мы вводим данные в первые три столбца, то созданная нами функция, самостоятельно просчитываем значения для них.

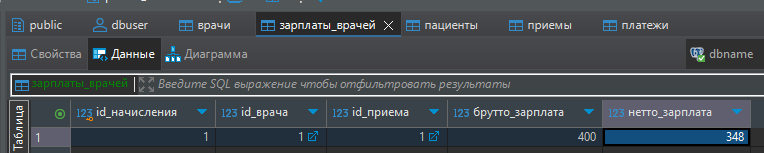


Рисунок 7 – Отображение данных таблицы «Зарплаты врачей»

Можно также проверить наличие нашей функции и триггера в DBeaver (рисунок 8-9).

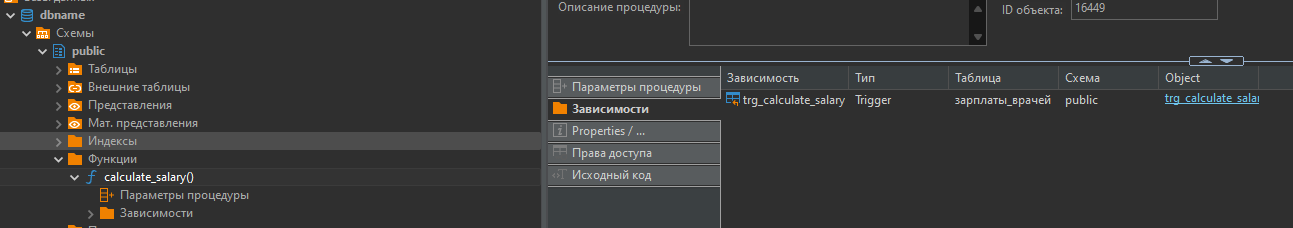


Рисунок 9 – Функция calculate\_salary

Как видно из рисунка 9, в созданной нами функции, имеется зависимость ввиде триггера, который относится и работает в таблице «Зарплаты врачей».

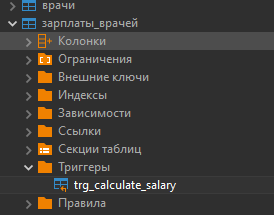


Рисунок 10 – Триггер trg\_calculate\_salary

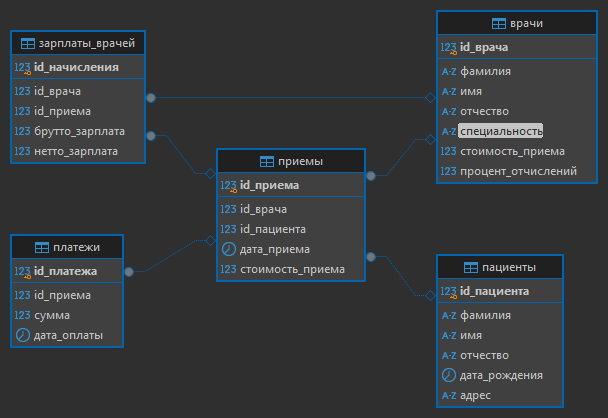


Рисунок 11 – Диаграмма отображения физической структуры базы данных

Эта диаграмма базы данных создана для медицинской информационной системы, которая позволяет управлять информацией о врачах, пациентах, приемах, платежах и начислениях зарплаты врачам. На ней изображены пять основных таблиц: врачи, пациенты, приемы, платежи и зарплаты врачей.

Основные функции системы:

1. Хранение информации о врачах:
   * Включает данные о врачах, такие как их фамилии, имена, отчества, специальности, стоимость приема и процент отчислений.
2. Хранение информации о пациентах:
   * Включает данные о пациентах, такие как их фамилии, имена, отчества, даты рождения и адреса.
3. Регистрация приемов:
   * Хранит информацию о каждом приеме, включая данные о враче и пациенте, дату приема и стоимость.
4. Учет платежей:
   * Регистрирует платежи за приемы, включая идентификатор платежа, сумму и дату оплаты.
5. Расчет зарплат врачей:
   * Ведет учет начислений зарплаты врачам, включая идентификаторы начисления, врача, приема, брутто- и нетто-зарплаты.

Для данной лабораторной работы, согласно заданию, были выбраны сущности: Врачи, пациенты, приемы, платежи, зарплаты врачей. Рассмотрим каждую сущность более подробно ниже.

**Врачи**

* **id\_врача**: уникальный идентификатор врача.
* **фамилия**: фамилия врача.
* **имя**: имя врача.
* **отчество**: отчество врача.
* **специальность**: специализация врача.
* **стоимость\_приема**: стоимость приема у данного врача.
* **процент\_отчислений**: процент отчислений врачу, применяемый к стоимости приема.

Эта сущность представляет врачей, работающих в клинике, и содержит информацию о каждом враче, включая его личные данные, специализацию и финансовые параметры.

**Пациенты**

* **id\_пациента**: уникальный идентификатор пациента.
* **фамилия**: фамилия пациента.
* **имя**: имя пациента.
* **отчество**: отчество пациента.
* **дата\_рождения**: дата рождения пациента.
* **адрес**: адрес пациента.

Эта сущность представляет пациентов, посещающих клинику, и содержит информацию о каждом пациенте, включая его личные данные и контактную информацию.

**Приемы**

* **id\_приема**: уникальный идентификатор приема.
* **id\_врача**: внешний ключ, ссылающийся на врача, проводившего прием.
* **id\_пациента**: внешний ключ, ссылающийся на пациента, проходившего прием.
* **дата\_приема**: дата приема.
* **стоимость\_приема**: стоимость приема.

Эта сущность представляет приемы, которые проводят врачи, и содержит информацию о каждом приеме, включая данные о враче и пациенте, а также стоимость приема.

**Платежи**

* **id\_платежа**: уникальный идентификатор платежа.
* **id\_приема**: внешний ключ, ссылающийся на прием, за который был совершен платеж.
* **сумма**: сумма платежа.
* **дата\_оплаты**: дата оплаты.

Эта сущность представляет платежи, сделанные за приемы, и содержит информацию о каждом платеже, включая ссылку на прием и сумму платежа.

**Зарплаты врачей**

* **id\_начисления**: уникальный идентификатор начисления зарплаты.
* **id\_врача**: внешний ключ, ссылающийся на врача, которому начисляется зарплата.
* **id\_приема**: внешний ключ, ссылающийся на прием, за который начисляется зарплата.
* **брутто\_зарплата**: зарплата врача до вычетов налогов
* **нетто\_зарплата**: зарплата врача после вычетов налогов

Эта сущность представляет начисления зарплаты врачам и содержит информацию о каждом начислении, включая ссылки на врача и прием, а также суммы зарплат.

Вывод: таким образом, мы изучили и освоили процессы разработки физической структуры базы данных с использованием системы управления базами данных Postgres.Так же научились создавать таблицы в СУБД Postgres с помощью SQL-запросов с атрибутами, связями, первичными и внешними ключами в соответствии с разработанной в предыдущей лабораторной работе логической структурой базы данных. И научились строить графической диаграммы для отображения физической структуры базы данных.

1. Особенности PostgreSQL

PostgreSQL — это мощная и гибкая реляционная система управления базами данных (СУБД) с открытым исходным кодом. Вот некоторые особенности PostgreSQL:

Расширяемость: PostgreSQL поддерживает расширение функциональности через различные расширения (extensions), включая географические данные (PostGIS), полнотекстовый поиск и другие.

Поддержка сложных типов данных: PostgreSQL поддерживает широкий спектр встроенных типов данных, таких как массивы, JSON, XML, а также позволяет пользователям определять свои собственные типы данных.

Конкурентоспособность: PostgreSQL использует механизм многоверсионности (MVCC), что обеспечивает высокую степень параллелизма и изоляции транзакций.

Безопасность: PostgreSQL предлагает мощные механизмы аутентификации и авторизации, а также шифрование данных как на уровне соединения, так и на уровне отдельных столбцов.

Производительность: PostgreSQL оптимизирован для работы с большими объёмами данных и высокой нагрузкой, поддерживая кластеризацию, репликацию и другие методы повышения производительности.

2. Что такое DDL?

DDL (Data Definition Language) — это набор команд SQL, используемых для определения структуры базы данных. Команды DDL позволяют создавать, изменять и удалять объекты базы данных, такие как таблицы, индексы, представления и др. Команды DDL:

* CREATE — создание объектов базы данных.
* ALTER — изменение существующих объектов.
* DROP — удаление объектов.
* TRUNCATE — очистка содержимого таблицы.

3. Что такое DML?

DML (Data Manipulation Language) — это набор команд SQL, предназначенных для манипуляции данными в базе данных. Они используются для добавления, изменения, удаления и выборки данных из таблиц. Команды DML:

* SELECT — выборка данных из одной или нескольких таблиц.
* INSERT — вставка новых строк в таблицу.
* UPDATE — обновление существующих данных в таблице.
* DELETE — удаление данных из таблицы.

4. Что такое DCL?

DCL (Data Control Language) — это набор команд SQL, используемых для управления доступом к данным и объектам базы данных. С помощью DCL можно предоставлять и отменять права доступа пользователей к различным ресурсам базы данных. Команды DCL:

* GRANT — предоставление прав доступа.
* REVOKE — отзыв прав доступа.

5. С какой команды, как правило, начинается запрос?

Большинство запросов начинаются с команды SELECT. Эта команда используется для выборки данных из одной или нескольких таблиц.

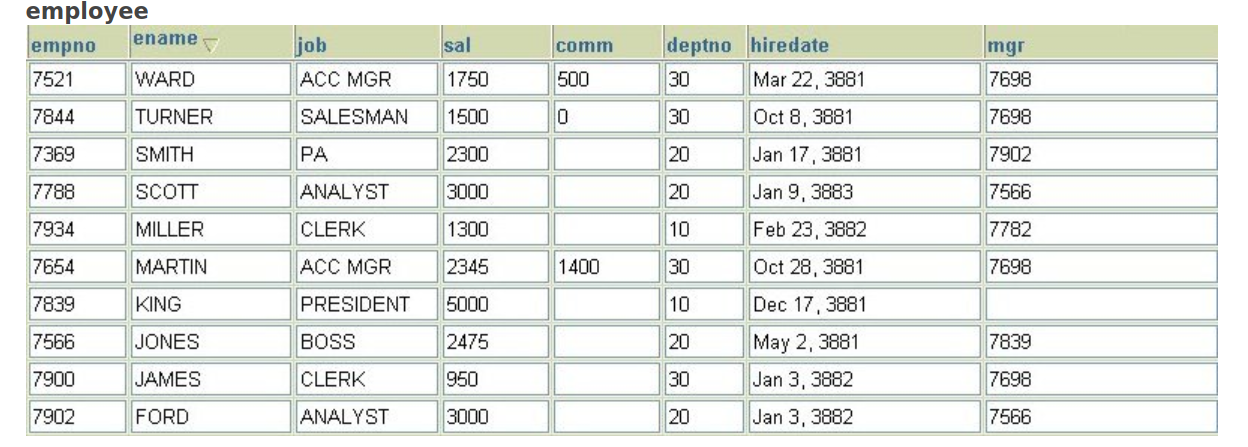
6. Какая команда используется для создания базы данных?

Для создания базы данных используется команда CREATE DATABASE “название базы данных”.

7. Какая команда используется для создания таблицы в базе данных?

Для создания таблицы используется команда CREATE TABLE “название таблицы”.

1. Существует следующая таблица:



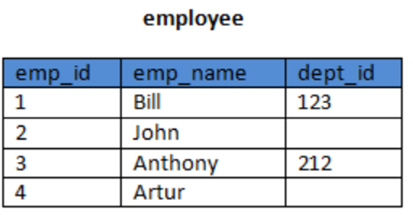
Из таблицы employee необходимо показать сотрудников с empno 7844, 7788 и 7902 одним запросом. Какие из приведенных ниже запросов соответствуют данному условию?

* SELECT \* FROM employee WHERE mod(mod(sal, 23), 5) = 0;
* SELECT \* FROM employee WHERE mod(sal, 23) = 0;
* SELECT \* FROM employee WHERE sal = 3000;
* SELECT \* FROM employee WHERE sal = 1500;
* **SELECT \* FROM employee WHERE sal IN (1500, 3000);**

Правильный запрос для отображения сотрудников с empno 7844, 7788 и 7902 одним запросом следующий:

SELECT \* FROM employee WHERE sal IN (1500, 3000);

1. В базе данных компании хранятся данные о работниках и подразделениях, к которым они относятся.



Ваша задача: достать весь список работников, при этом все работники без подразделения должны быть в начале списка. Какой из представленных запросов выполнит эту задачу?

* SELECT \* FROM employee ORDER BY dept\_id NULLS LAST;
* SELECT \* FROM employee ORDER BY dept\_id ASC;
* **SELECT \* FROM employee ORDER BY dept\_id NULLS FIRST**;
* SELECT \* FROM employee ORDER BY dept\_id;
* SELECT \* FROM employee ORDER BY dept\_id NULLS FIRST ASC;

Правильный запрос, для отображения списка работников, при этом чтобы все работники без подразделения были в начале списка, одним запросом следующий:

SELECT \* FROM employee ORDER BY dept\_id NULLS FIRST

1. Какие из следующих значений подходят для типа char в PostgreSQL?

* 12
* '1'
* 1.22
* 'a'
* 'char'

Значение, подходящее для типа CHAR в PostgreSQL:

'a' — строка, состоящая из одного символа.

Остальные значения не подходят, так как CHAR предназначен для хранения строк фиксированной длины.

1. Какие из перечисленных конструкций запроса НЕ являются верными? Выбрать можно несколько вариантов ответа.

* Select список полей from список таблиц where условия order by поля для сортировки
* **Select список полей from список таблиц union Select список полей from список таблиц order by поля для сортировки**
* Select список полей from список таблиц having условия группировки group by поля для группировки
* Select список полей from список таблиц where условия group by поля для группировки order by поля для сортировки
* **Select список полей from список таблиц order by поля для сортировки union Select список полей from список таблиц order by поля для сортировки**

Неверные конструкции:

* Select список полей from список таблиц union Select список полей from список таблиц order by поля для сортировки
* Select список полей from список таблиц order by поля для сортировки union Select список полей from список таблиц order by поля для сортировки

Причина неверности заключается в неправильном использовании оператора UNION. После UNION не следует указывать команду ORDER BY для каждой из частей запроса. Правильная конструкция предполагает использование ORDER BY только один раз в конце всего запроса.

1. Необходимо вычислить средний объем продаж (advance) книг и годовой доход (sales) от продажи книг по всем категориям (type) из таблицы titles. Какой из перечисленных фрагментов кода позволит решить поставленную задачу?

* Select avg (advance), sum (sales) from titles where type between ’business’ or ’internet’
* Select avg (advance), sum (sales) from titles
* **Select avg (advance), sum (sales) from titles where type between ’business’ and ’internet’**
* Select avg (advance), sum (sales) from titles group by type
* Select avg (advance), sum (sales) from titles where type = ’business’ and type = ’internet’

Правильным запросом, для поставленной задачи является:

Select avg (advance), sum (sales) from titles where type between ’business’ and ’internet’

Т.к, он выбирает среднюю стоимость аванса и общую сумму продаж для всех категорий, находящихся между business и internet.