Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОННИКИ

Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Дисциплина: Базы данных

ОТЧЁТ
по лабораторной работе №6
«Создание приложения для базы данных»
на тему

«Континентальная хоккейная лига»

Студент М.А. Бекетова

Преподаватель Д.В. Куприянова

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные информационные системы немыслимы без использования баз данных (БД), которые обеспечивают структурированное хранение, эффективное управление и быстрый доступ к большим объемам данных. Приложения для работы с базами данных играют ключевую роль в автоматизации бизнес-процессов, аналитике и поддержке принятия решений. Одной из наиболее популярных систем управления базами данных (СУБД) является PostgreSQL, благодаря своей надежности, открытому исходному коду и поддержке расширенных функций, таких как транзакции, триггеры и полнотекстовый поиск.

Разработанное программное обеспечение предоставляет пользователям интуитивно понятную и наглядную среду для выполнения основных и продвинутых операций с базой данных. Интерфейс приложения ориентирован как на опытных пользователей, так и на тех, кто только начинает работу с СУБД, устраняя необходимость ручного написания SQL-запросов для типовых операций.

В рамках задачи предполагается реализовать базовые операции управления данными: создание и удаление таблиц, редактирование их структуры, резервное копирование и восстановление информации. Особое внимание уделяется механизмам экспорта данных в форматы, удобные для анализа (например, Excel), а также сохранению и повторному использованию пользовательских запросов. Это позволяет повысить гибкость работы с информацией и адаптировать приложение под конкретные сценарии использования.

Разрабатываемое приложение демонстрирует интеграцию PostgreSQL с выбранным языком программирования, подчеркивая возможности взаимодействия между клиентской частью и СУБД. Важным аспектом является обеспечение отказоустойчивости через резервное копирование и реализацию транзакций для сохранения целостности данных.

1 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

1.1 Системные требования

Операционная система Windows 11 и выше, Python 3.12 с библиотеками psycopg2, pandas, tkinter, PostgreSQL.

1.2 Обзор интерфейса приложения

В левом верхнем углу окна представлена область для подключения к базе данных. В правом углу – область для отображения списка таблиц. В нижней части представлены кнопки для взаимодействия и управления таблицами. На рисунке 1.1 представлен интерфейс приложения.

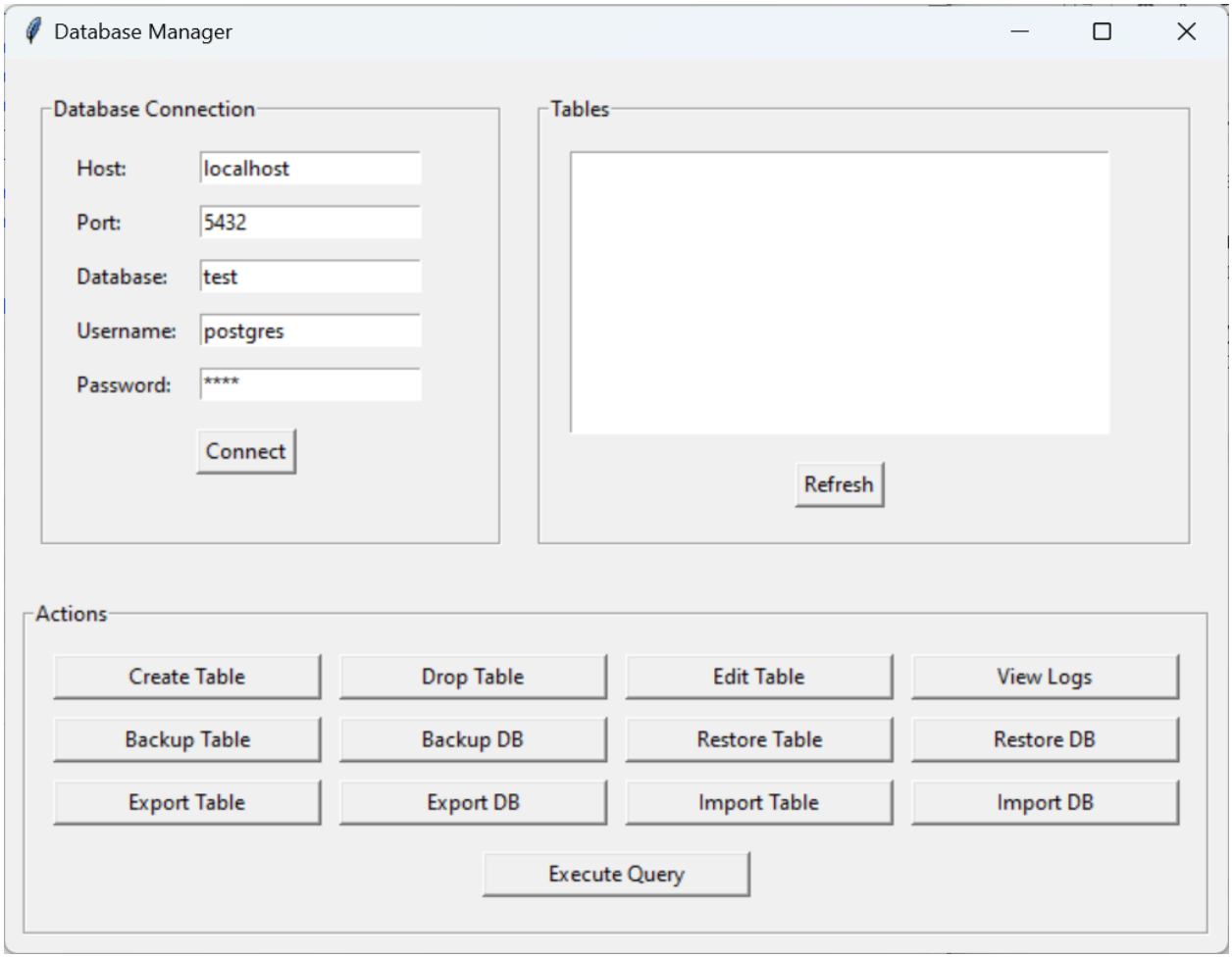

Рисунок 1.1 – Интерфейс приложения

1.3 Отображение таблицы

Для отображения определенной таблицы необходимо в окне таблиц выбрать из списка нужную таблицу и нажать кнопку «Edit table». Пример списка показан на рисунке 1.2.

После выбора таблицы, откроется окно, где отобразится содержимое таблицы. Пример вывода таблицы coach представлен на рисунке 1.3.

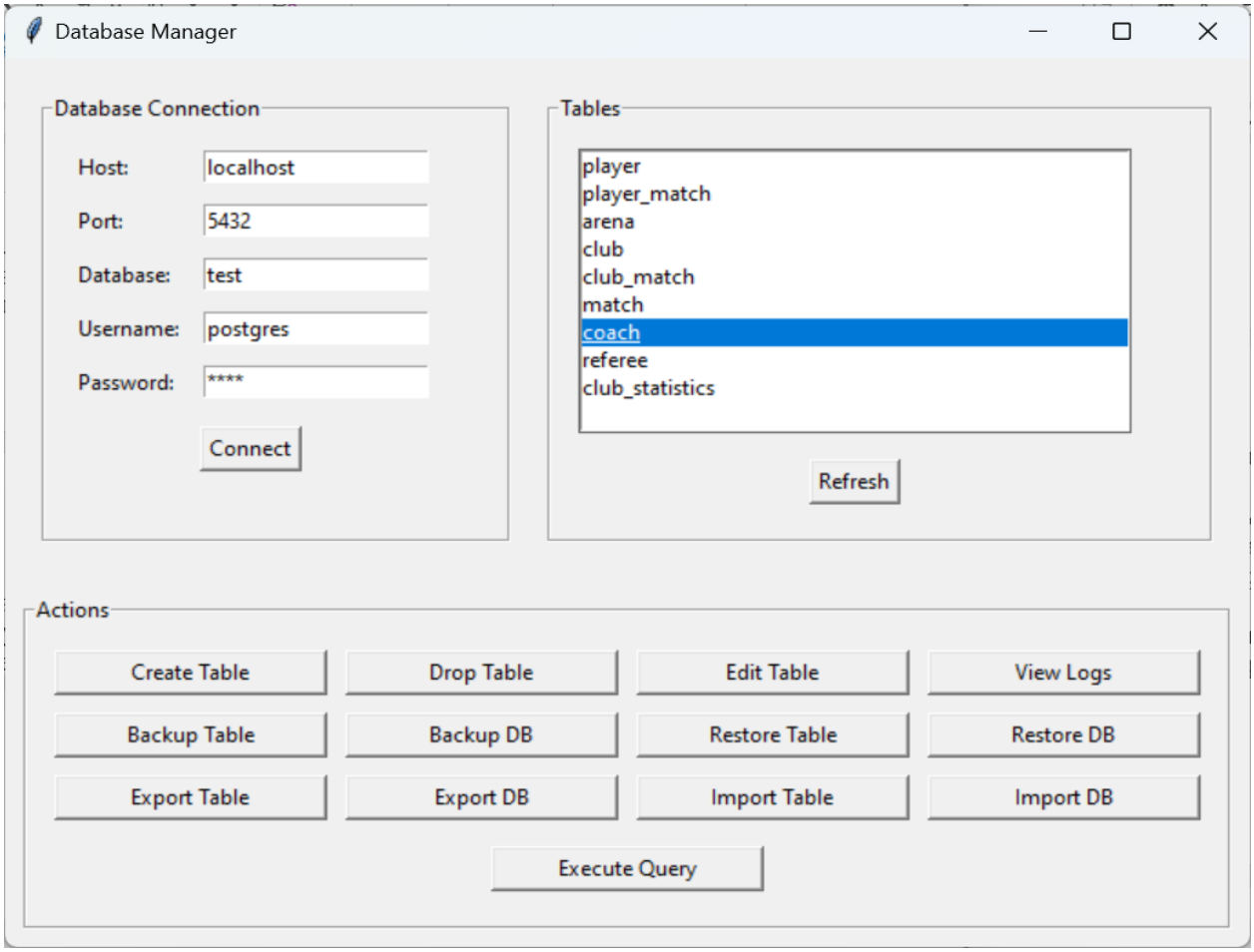

Рисунок 1.2 – Выбор таблицы

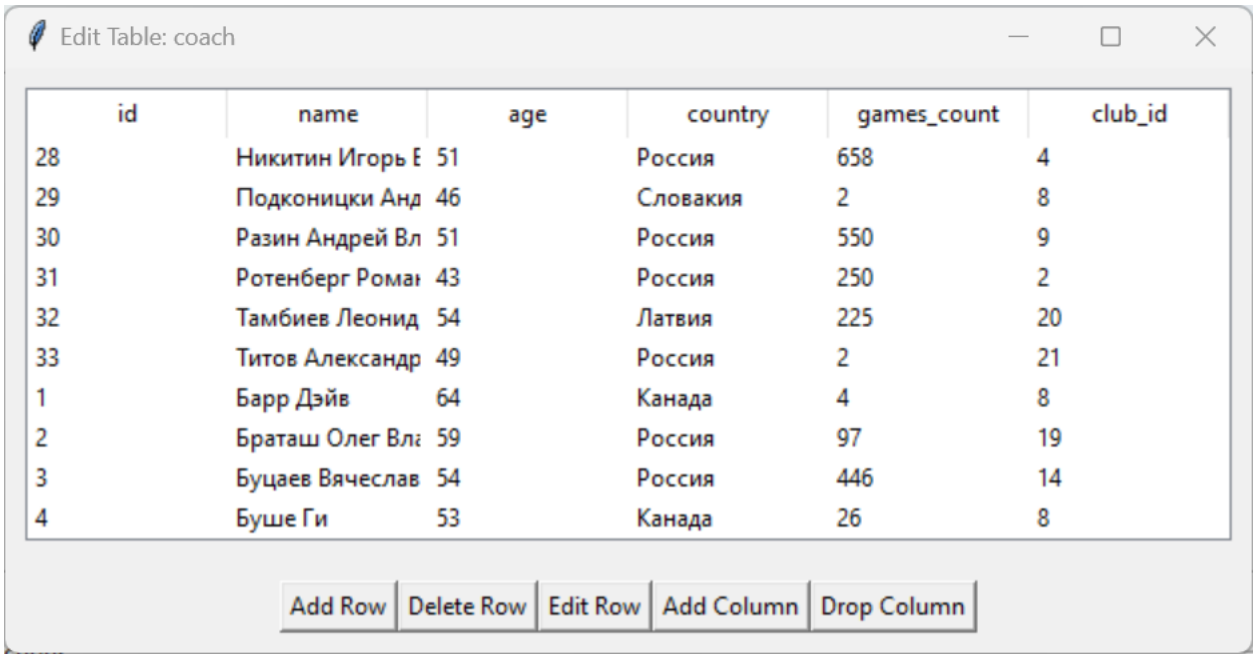

Рисунок 1.3 – Вывод таблицы coach

1.4 Редактирование, добавление и удаление записей таблицы

Для добавления и удаления записей в меню окна имеются специальные кнопки. Для удаления требуется сначала выбрать запись, которую нужно удалить, а после нажать на соответствующую кнопку. Для добавления записи требуется только нажать необходимую кнопку и ввести информацию, которая будет запрошена в диалоговом окне. Если требуется изменить существующую запись, следует ее выбрать из списка всех записей и нажать «Edit Row». После чего нужно последовательно ввести новые значения, где это необходимо.

На рисунке 1.4 показано окно редактирования записи на примере записи из таблицы coach.

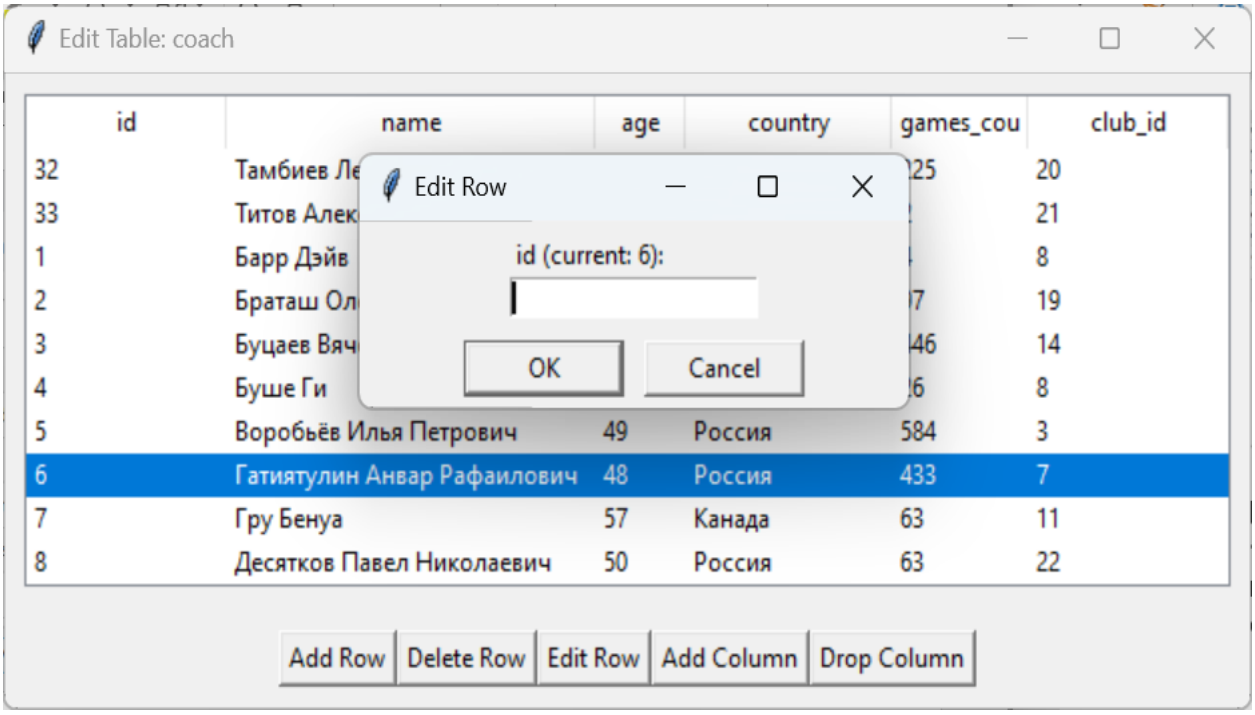

Рисунок 1.4 – Окно редактирования записи в таблице

1.5 Добавление таблицы

Для добавления таблицы в базу данных нажмите кнопку «Create Table».

После нажатия кнопки появляется окно добавления таблицы в базу данных, где сперва требуется ввести название таблицы, а после названия полей и типов данных.

Основные типы данных для столбцов:

- integer – целые числа;
- real – числа с плавающей запятой;
- text – текстовая информация;
- boolean – логический тип данных;
- date – тип данных даты;
- money – денежный тип данных.

При нажатии кнопки «Ok», если какие-то данные введены некорректно или уже существуют, появится окно с предупреждением.

На рисунке 1.5 представлено окно для ввода названий полей.

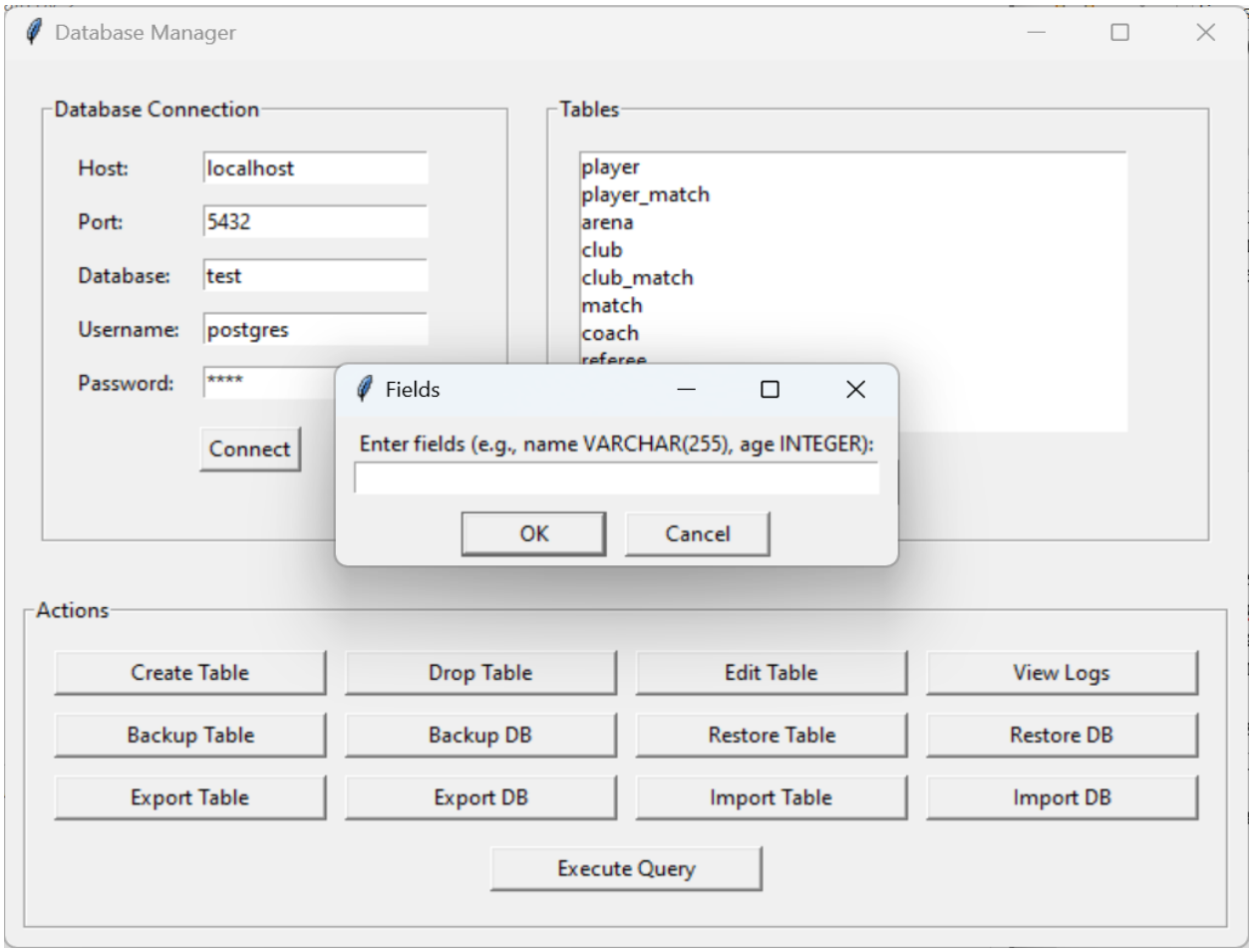

Рисунок 1.5 – Окно создания таблицы

1.6 Удаление таблицы из базы данных

Для удаления таблицы выберите нужную вам таблицу и нажмите кнопку «Drop Table».

После нажатия кнопки появится окно с подтверждением удаления выбранной пользователем таблицы. Пример окна показан на рисунке 1.6.

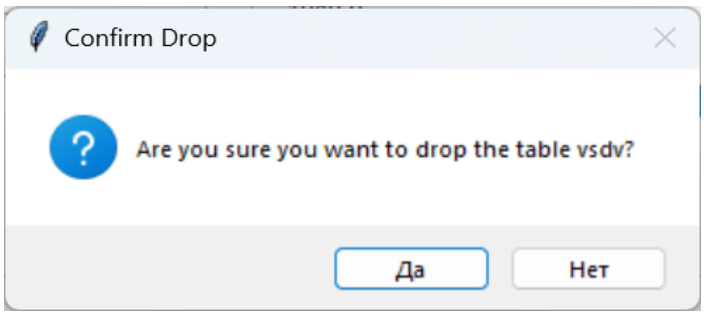

Рисунок 1.6 – Подтверждение удаления таблицы vsdv

1.7 SQL запросы

В нижней части окна приложения есть кнопка «Execute Query». При нажатии на кнопку открывается окно для применения и создания произвольных SQL запросов. На рисунке 1.7 представлено данное окно.

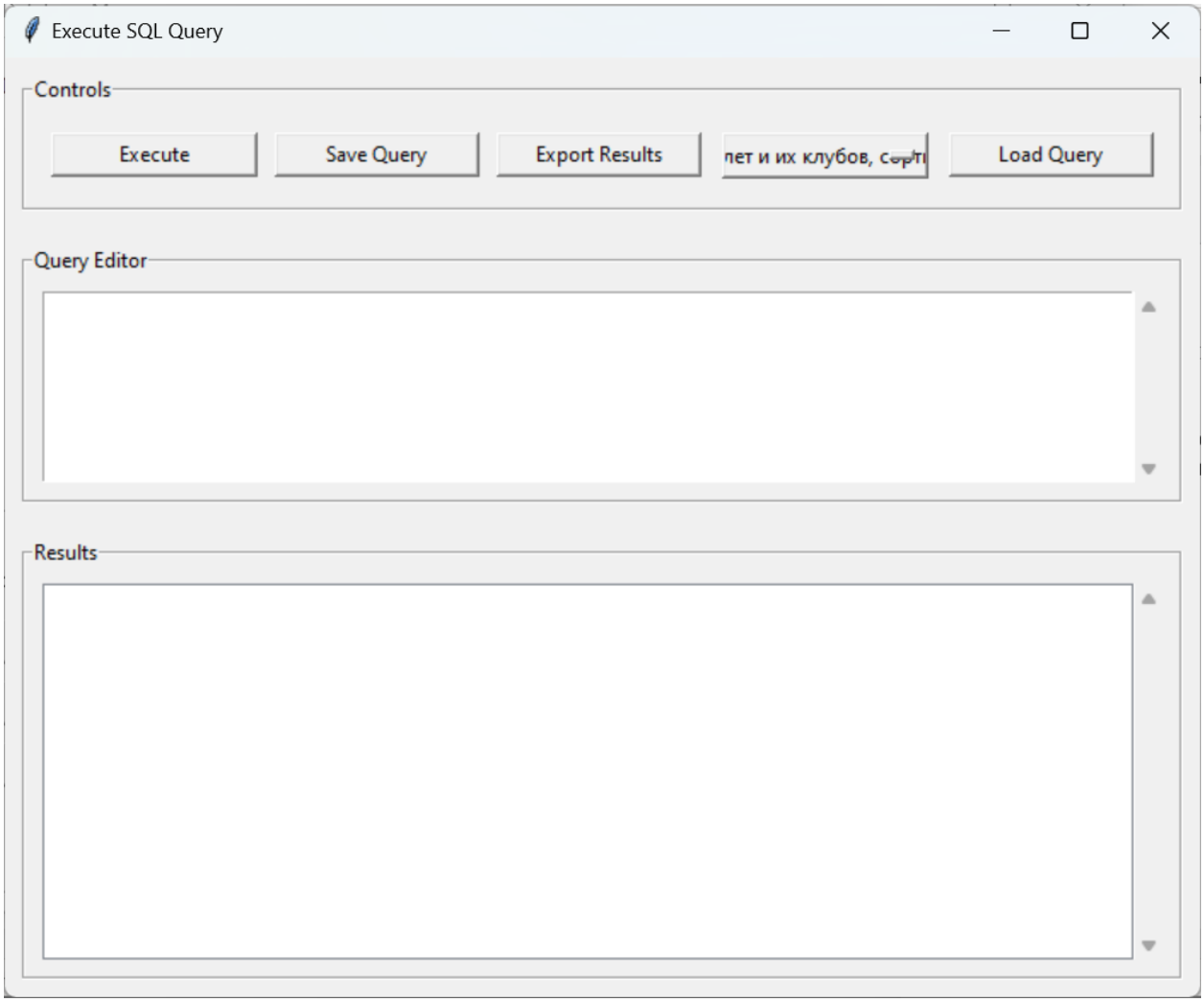

Рисунок 1.7 – Область для работы с SQL запросами

Окно включает в себя три части. Первая часть содержит кнопки управления. Вторая часть, центральная, представляет собой окно для редактирования запросов. А в самой нижней части после выполнения запросов можно увидеть их результаты.

Для выполнения SQL запроса нужно либо выбрать один из уже существующих запросов из выпадающего списка, либо написать новый. В первом случае потребуется выбрать запрос, нажать кнопку «Load Query», а после нажать «Execute». Во втором случае достаточно ввести запрос вручную и нажать кнопку «Execute», после чего можно сохранить запрос нажав «Save Query». В обоих случаях результат выполнения запроса можно экспортировать в файл используя пункт меню «Export Results». На рисунке 1.8 показан список реализованных SQL запросов из лабораторных работ №4 и №5.

Пример выполнения одного из запросов представлен на рисунке 1.9.

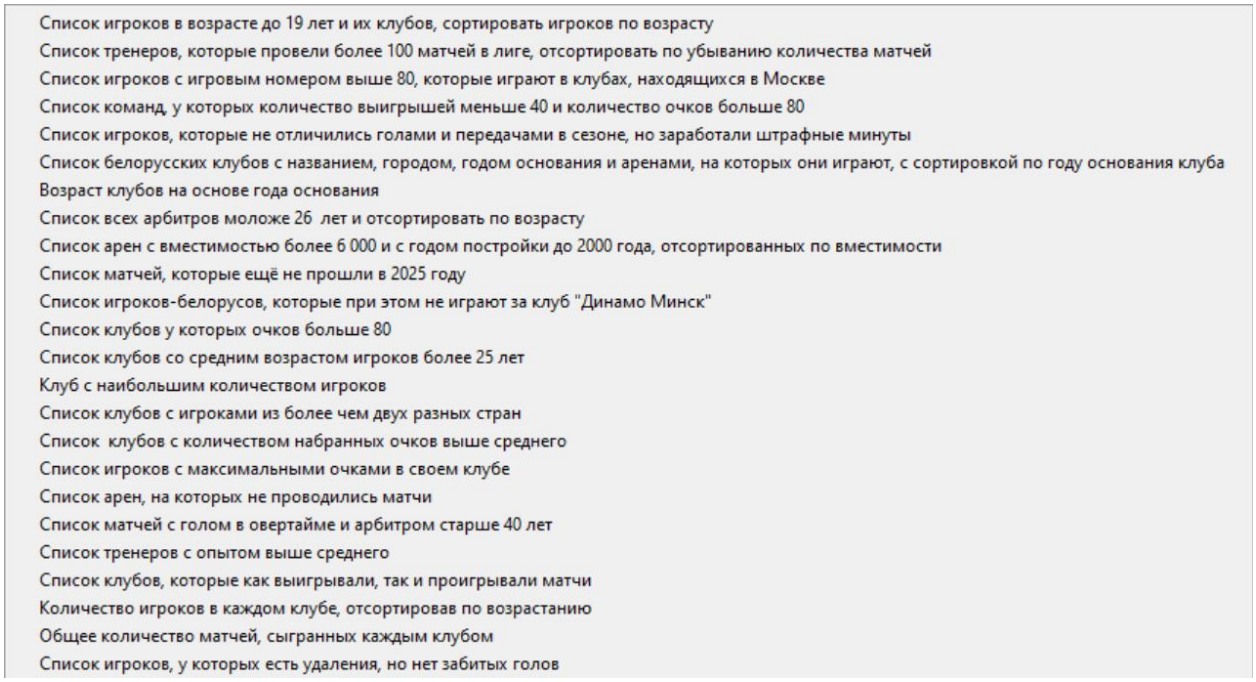

Рисунок 1.8 – Список скриптом из лабораторных работ №4 и №5

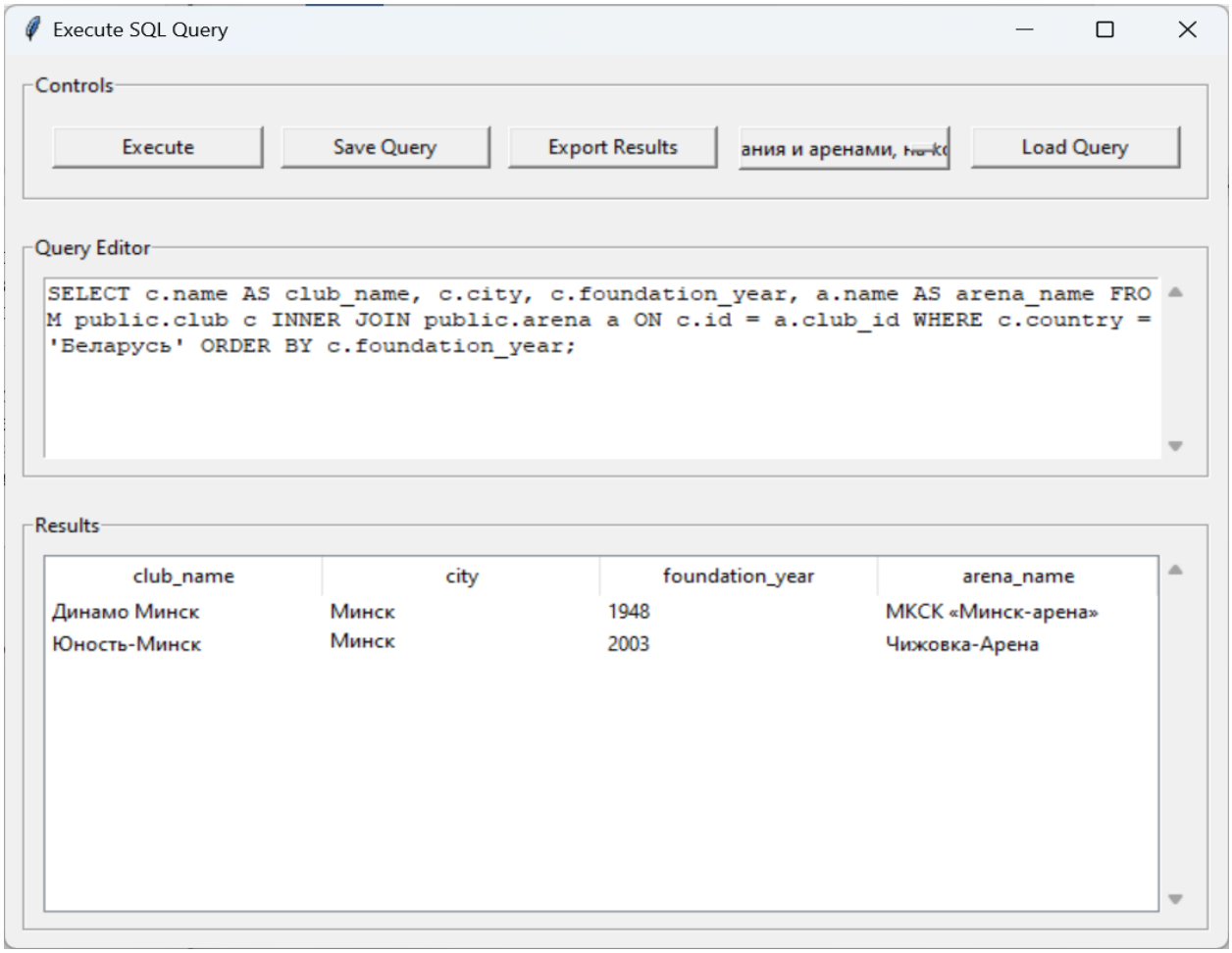

Рисунок 1.9 – Область SQL скриптов после выбора скрипта из списка

1.8 Экспорт в Excel

В основном меню есть четыре кнопки, позволяющие работать с Excel: «Export Table», «Export DB», «Import Table» и «Import DB». Перечисленные пункты меню позволяют экспортировать данные таблиц и всей базы данных в Excel файл, а также наоборот импортировать данные из файлов. Пример сохранения файла показан на рисунке 1.10.

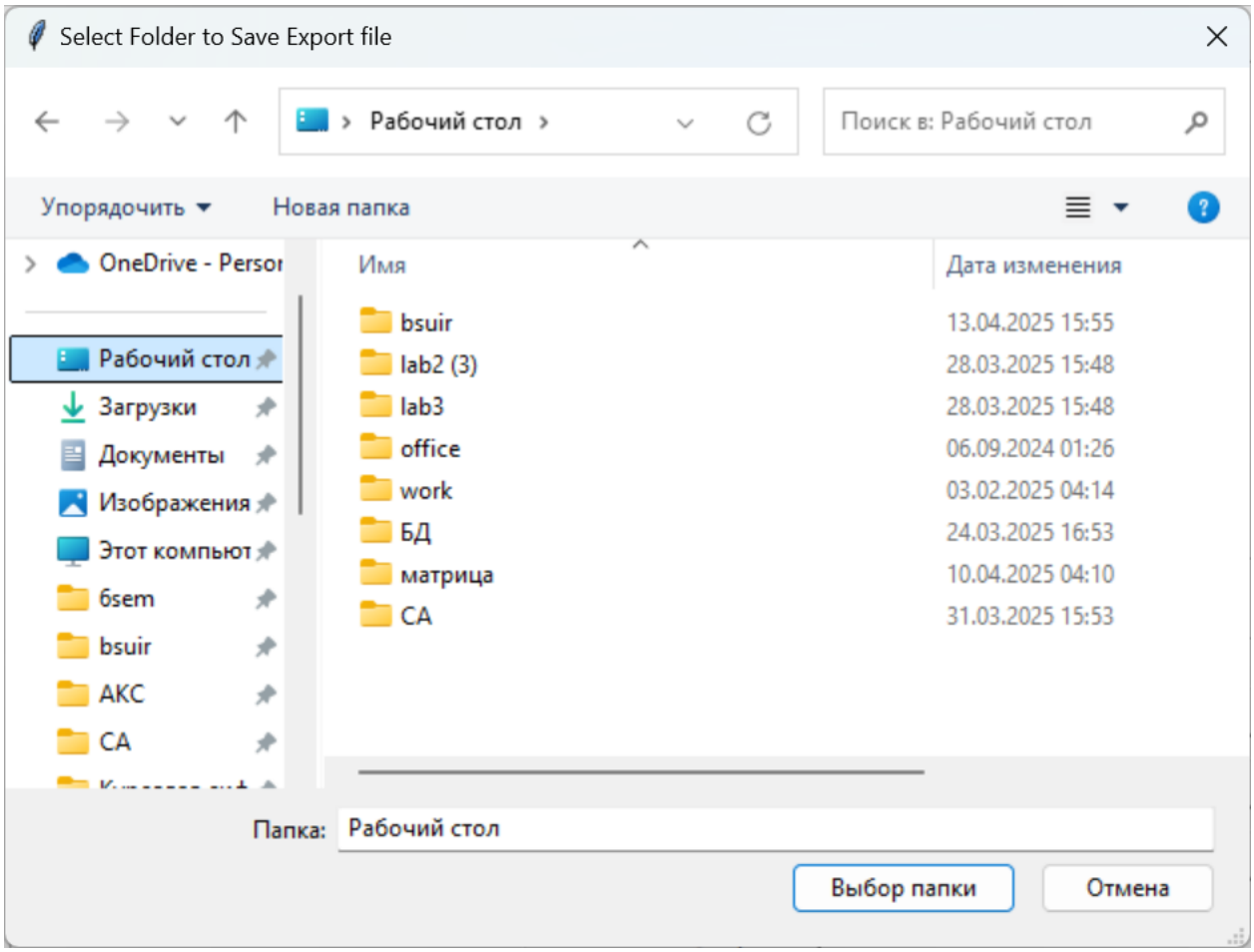

Рисунок 1.10 – Выбор места для экспорта таблицы

1.9 Создание резервной копии

Выбрав пункты основного меню «Backup Table», «Backup DB», «Restore Table» и «Restore DB». Данные кнопки продемонстрированы на рисунке 1.11.

Рисунок 1.11 – Кнопки резервного копирования

Для создания резервной копии таблицы или всей базы данных нужно нажать кнопки «Backup Table» и «Backup DB» соответственно. После этого

пользователю, как и в случае с созданием Excel файлов, предложат выбрать место для сохранения копии. Название файла резервной копии генерируется автоматически.

Восстановить таблицы используя резервные копии можно с помощью кнопок «Restore Table» и «Restore DB».

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы было разработано настольное приложение для взаимодействия с базой данных, написанное на языке программирования Python с использованием библиотеки tkinter для создания графического интерфейса. Основное назначение программы – обеспечение удобного и наглядного интерфейса для выполнения операций с базой данных PostgreSQL и визуализации результатов запросов.

Приложение реализует возможность создания, редактирования и удаления таблиц, а также управления отдельными записями. Пользователю предоставлена возможность добавления и удаления столбцов, а также работы со структурой таблиц. Кроме того, поддерживается создание резервных копий как отдельных таблиц, так и всей базы данных с возможностью последующего восстановления. Это позволяет обеспечить стабильную и безопасную работу с данными, снижая риск их потери.

Встроенный модуль управления SQL-запросами позволяет не только выполнять предопределённые запросы, но и создавать пользовательские, с возможностью их сохранения для повторного использования. Результаты запросов могут быть экспортированы в файл Excel-формата, что удобно для анализа, хранения или последующей обработки данных.

Таким образом, разработанное приложение охватывает все ключевые функции, необходимые для эффективной работы с реляционной базой данных в рамках учебного проекта. Оно сочетает в себе мощный функционал с простотой и доступностью пользовательского интерфейса, что делает его удобным как для студентов, так и для преподавателей. Структура программы также предполагает возможность дальнейшего масштабирования и функционального расширения.

ПРИЛОЖЕНИЕ А
(обязательное)

Листинг программного кода

```
import tkinter as tk
from tkinter import messagebox, filedialog, simpledialog
import psycopg2
import pandas as pd
import os
from tkinter.ttk import Treeview
from tkinter.scrolledtext import ScrolledText
import logging
import re


class DatabaseApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Database Manager")
        self.conn = None
        self.cursor = None
        self.tables = []
        self.create_widgets()

        # Настройка логирования SQL-запросов
        logging.basicConfig(
            filename="sql_queries.log",
            level=logging.INFO,
            format="%(asctime)s - %(message)s",
        )

    def create_widgets(self):
        # Part 1 & 2 - Connection Panel + Tables List (Top Row)
        self.top_frame = tk.Frame(self.root)
        self.top_frame.grid(
            row=0,  column=0,  columnspan=2,  padx=10,  pady=10,
sticky="ew"
        )

        # Connection Panel
        self.connection_frame = tk.LabelFrame(
            self.top_frame,  text="Database  Connection",  padx=10,
pady=10
        )
        self.connection_frame.grid(row=0,   column=0,   padx=10,
pady=10, sticky="nsew")

        self.host_label     =     tk.Label(self.connection_frame,
text="Host:")
        self.host_label.grid(row=0, column=0, sticky="w", padx=5,
pady=5)
```

```
        self.host_entry = tk.Entry(self.connection_frame)
        self.host_entry.insert(0, "localhost")
        self.host_entry.grid(row=0, column=1, padx=5, pady=5)

        self.port_label     =     tk.Label(self.connection_frame,
text="Port:")
        self.port_label.grid(row=1, column=0, sticky="w", padx=5,
pady=5)
        self.port_entry = tk.Entry(self.connection_frame)
        self.port_entry.insert(0, "5432")
        self.port_entry.grid(row=1, column=1, padx=5, pady=5)

        self.db_label      =      tk.Label(self.connection_frame,
text="Database:")
        self.db_label.grid(row=2,  column=0,  sticky="w",  padx=5,
pady=5)
        self.db_entry = tk.Entry(self.connection_frame)
        self.db_entry.insert(0, "test")
        self.db_entry.grid(row=2, column=1, padx=5, pady=5)

        self.user_label     =     tk.Label(self.connection_frame,
text="Username:")
        self.user_label.grid(row=3, column=0, sticky="w", padx=5,
pady=5)
        self.user_entry = tk.Entry(self.connection_frame)
        self.user_entry.insert(0, "postgres")
        self.user_entry.grid(row=3, column=1, padx=5, pady=5)

        self.password_label   =   tk.Label(self.connection_frame,
text="Password:")
        self.password_label.grid(row=4,   column=0,   sticky="w",
padx=5, pady=5)
        self.password_entry   =   tk.Entry(self.connection_frame,
show="*")
        self.password_entry.insert(0, "1313")
        self.password_entry.grid(row=4, column=1, padx=5, pady=5)

        self.connect_button = tk.Button(
            self.connection_frame,                text="Connect",
command=self.connect_to_db
        )
        self.connect_button.grid(row=5, columnspan=2, pady=10)

        # Tables List Panel
        self.tables_frame = tk.LabelFrame(
            self.top_frame, text="Tables", padx=10, pady=10
        )
        self.tables_frame.grid(row=0, column=1, padx=10, pady=10,
sticky="nsew")

        self.tables_listbox    =    tk.Listbox(self.tables_frame,
height=10, width=50)
        self.tables_listbox.grid(row=0, column=0, padx=5, pady=5)
```

```
        self.refresh_button = tk.Button(
            self.tables_frame,                    text="Refresh",
command=self.refresh_tables
        )
        self.refresh_button.grid(row=1, column=0, pady=10)

        # Part 3 - Actions Panel (Bottom)
        self.actions_frame       =       tk.LabelFrame(self.root,
text="Actions", padx=10, pady=10)
        self.actions_frame.grid(
            row=1,  column=0,  columnspan=2,  padx=10,  pady=10,
sticky="ew"
        )

        actions = [
            ("Create Table", self.create_table),
            ("Drop Table", self.drop_table),
            ("Edit Table", self.edit_table),
            ("View Logs", self.view_logs),
            ("Backup Table", self.backup_table),
            ("Backup DB", self.backup_db),
            ("Restore Table", self.restore_table),
            ("Restore DB", self.restore_db),
            ("Export Table", self.export_table),
            ("Export DB", self.export_db),
            ("Import Table", self.import_table),
            ("Import DB", self.import_db),
        ]

        # Установим фиксированную ширину для всех кнопок
        button_width = 20

        for i, (text, command) in enumerate(actions):
            button = tk.Button(
                self.actions_frame,  text=text,  command=command,
width=button_width
            )
            button.grid(row=i // 4, column=i % 4, padx=5, pady=5,
sticky="ew")

        execute_button = tk.Button(
            self.actions_frame,
            text="Execute Query",
            command=self.execute_query,
            width=button_width,
        )
        execute_button.grid(row=3,    column=1,    columnspan=2,
padx=5, pady=10)

        # Настройка весовых коэффициентов для выравнивания фреймов
        self.top_frame.columnconfigure(0, weight=1)
        self.top_frame.columnconfigure(1, weight=1)
```

```
        self.top_frame.rowconfigure(0, weight=1)

    def connect_to_db(self):
        try:
            host = self.host_entry.get()
            port = self.port_entry.get()
            dbname = self.db_entry.get()
            user = self.user_entry.get()
            password = self.password_entry.get()

            # Establishing connection to the database
            self.conn = psycopg2.connect(
                host=host, port=port, dbname=dbname, user=user,
password=password
            )
            self.cursor = self.conn.cursor()

            messagebox.showinfo("Success", "Connected to the
database successfully")
            self.refresh_tables()

        except Exception as e:
            messagebox.showerror(
                "Error", f"Unable to connect to the database:
{str(e)}"
            )

    def refresh_tables(self):
        if self.conn is not None:
            try:
                self.cursor.execute(
                    "SELECT table_name FROM
information_schema.tables WHERE table_schema = 'public'"
                )
                tables = self.cursor.fetchall()
                self.tables = [table[0] for table in tables]

                self.tables_listbox.delete(0, tk.END)
                for table in self.tables:
                    self.tables_listbox.insert(tk.END, table)

            except Exception as e:
                messagebox.showerror("Error", f"Failed to
retrieve tables: {str(e)}")

    def create_table(self):
        table_name = simpledialog.askstring("Table Name", "Enter
the table name:")
        if not table_name:
            return

        if not re.match(r"^[a-zA-Z_][a-zA-Z0-9_]*$", table_name):
            messagebox.showerror(
```

```
                "Invalid Name", "Table name contains invalid
characters."
            )
            return

        fields_input = simpledialog.askstring(
            "Fields", "Enter fields (e.g., name VARCHAR(255), age
INTEGER):"
        )
        if not fields_input:
            return

        try:
            query = f'CREATE TABLE "{table_name}"
({fields_input});'
            self.cursor.execute(query)
            self.conn.commit()
            messagebox.showinfo("Success", f"Table {table_name}
created successfully.")

            self.refresh_tables()
            logging.info(f"Executed: {query}")

        except Exception as e:
            self.conn.rollback()
            messagebox.showerror("Error", f"Failed to create
table:\n{e}")

    def drop_table(self):
        selected_table = self.tables_listbox.get(tk.ACTIVE)
        if selected_table:
            confirm = messagebox.askyesno(
                "Confirm Drop",
                f"Are you sure you want to drop the table
{selected_table}?",
            )
            if confirm:
                try:
                    query = f"DROP TABLE IF EXISTS
{selected_table};"
                    self.cursor.execute(query)
                    self.conn.commit()
                    messagebox.showinfo(
                        "Success", f"Table {selected_table}
dropped successfully"
                    )
                    self.refresh_tables()

                    # Log the query
                    logging.info(f"Executed: {query}")
                except Exception as e:
                    self.conn.rollback()
```

```
                    messagebox.showerror("Error",   f"Failed   to
drop table: {str(e)}")

    def edit_table(self):
        selected_table = self.tables_listbox.get(tk.ACTIVE)
        if not selected_table:
            messagebox.showerror("Error", "No table selected!")
            return

        # Окно для редактирования таблицы
        edit_window = tk.Toplevel(self.root)
        edit_window.title(f"Edit Table: {selected_table}")

        # Отображение данных таблицы
        self.cursor.execute(f"SELECT * FROM {selected_table}")
        rows = self.cursor.fetchall()
        columns = [desc[0] for desc in self.cursor.description]

        tree     =     Treeview(edit_window,     columns=columns,
show="headings")
        for col in columns:
            tree.heading(col, text=col)
            tree.column(col, width=100)
        for row in rows:
            tree.insert("", tk.END, values=row)

        button_frame = tk.Frame(edit_window)
        button_frame.grid(row=1, column=0, columnspan=5, pady=10)
        tree.grid(row=0, column=0, padx=10, pady=10)

        # Кнопки управления
        tk.Button(
            button_frame,
            text="Add Row",
            command=lambda: self.add_row(selected_table, tree),
        ).grid(row=0, column=0)

        tk.Button(
            button_frame,
            text="Delete Row",
            command=lambda:       self.delete_row(selected_table,
tree),
        ).grid(row=0, column=1)

        tk.Button(
            button_frame,
            text="Edit Row",
            command=lambda: self.edit_row(selected_table, tree),
        ).grid(row=0, column=2)

        tk.Button(
            button_frame,
            text="Add Column",
```

```
            command=lambda:       self.add_column(selected_table,
tree),
        ).grid(row=0, column=3)

        tk.Button(
            button_frame,
            text="Drop Column",
            command=lambda:      self.drop_column(selected_table,
tree),
        ).grid(row=0, column=4)

    def add_row(self, table_name, tree):
        self.cursor.execute(f'SELECT * FROM "{table_name}" LIMIT
1')
        columns = [desc[0] for desc in self.cursor.description]

        row_data = []
        for col in columns:
            value  =  simpledialog.askstring("Add  Row",  f"Enter
value for '{col}':")
            row_data.append(value if value != "" else None)

        placeholders = ",".join(["%s"] * len(columns))
        query = (
            f'INSERT  INTO  "{table_name}"  ({",".join(columns)})
VALUES ({placeholders});'
        )
        try:
            self.cursor.execute(query, row_data)
            self.conn.commit()
            messagebox.showinfo("Success",       "Row       added
successfully.")
            logging.info(f"Executed:    {query}    with    values
{row_data}")
            tree.insert("", tk.END, values=row_data)
        except Exception as e:
            self.conn.rollback()
            messagebox.showerror("Error",    f"Failed    to    add
row:\n{e}")

    def delete_row(self, table_name, tree):
        selected_item = tree.selection()
        if not selected_item:
            return

        values = tree.item(selected_item)["values"]
        if not values:
            return

        pk_column = tree["columns"][0]
        pk_value = values[0]

        try:
```

```
            query = f'DELETE FROM "{table_name}" WHERE
"{pk_column}" = %s;'
            self.cursor.execute(query, (pk_value,))
            self.conn.commit()
            logging.info(f"Executed: {query} with {pk_value}")
            tree.delete(selected_item)
            messagebox.showinfo("Success", "Row deleted.")
        except Exception as e:
            self.conn.rollback()
            messagebox.showerror("Error", f"Failed to delete
row:\n{e}")

    def edit_row(self, table_name, tree):
        selected_item = tree.selection()
        if not selected_item:
            return

        values = tree.item(selected_item)["values"]
        columns = tree["columns"]
        new_values = []

        for i, col in enumerate(columns):
            new_val = simpledialog.askstring(
                "Edit Row", f"{col} (current: {values[i]}):"
            )
            new_values.append(new_val if new_val != "" else
values[i])

        pk_column = columns[0]
        pk_value = values[0]

        set_clause = ", ".join([f'"{col}" = %s' for col in
columns])
        query = f'UPDATE "{table_name}" SET {set_clause} WHERE
"{pk_column}" = %s;'

        try:
            self.cursor.execute(query, new_values + [pk_value])
            self.conn.commit()
            logging.info(f"Executed: {query} with {new_values +
[pk_value]}")
            tree.item(selected_item, values=new_values)
            messagebox.showinfo("Success", "Row updated.")
        except Exception as e:
            self.conn.rollback()
            messagebox.showerror("Error", f"Failed to edit
row:\n{e}")

    def view_logs(self):
        log_window = tk.Toplevel(self.root)
        log_window.title("SQL Query Logs")

        log_text = ScrolledText(log_window, width=80, height=20)
```

```
        log_text.grid(row=0, column=0, columnspan=2)

        def refresh_logs():
            log_text.delete("1.0", tk.END)
            with open("sql_queries.log", "r") as f:
                log_text.insert(tk.END, f.read())

        tk.Button(log_window,                       text="Refresh",
command=refresh_logs).grid(
            row=1, column=0
        )
        tk.Button(
            log_window,
            text="Clear Logs",
            command=lambda:              [open("sql_queries.log",
"w").close(), refresh_logs()],
        ).grid(row=1, column=1)

        refresh_logs()

    def backup_table(self):
        if not self.conn:
            messagebox.showerror("Error",   "Not   connected   to
database")
            return

        selected_table = self.tables_listbox.get(tk.ACTIVE)
        if not selected_table:
            messagebox.showerror("Error", "No table selected")
            return

        folder = filedialog.askdirectory(title="Select Folder to
Save SQL Backup")
        if not folder:
            return

        try:
            # Получаем структуру таблицы
            self.cursor.execute(
                f"""
                SELECT 'CREATE TABLE "{selected_table}" (' ||
                string_agg('"'   ||   column_name   ||   '"   '   ||
data_type, ', ') || ');'
                FROM information_schema.columns
                WHERE table_name = %s
            """,
                (selected_table,),
            )
            create_stmt = self.cursor.fetchone()[0]

            # Получаем содержимое таблицы
            self.cursor.execute(f"SELECT          *          FROM
{selected_table}")
```

```
            rows = self.cursor.fetchall()
            col_names    =    [f'"{desc[0]}"'    for    desc    in
self.cursor.description]

            insert_stmts = ""
            for row in rows:
                values = []
                for val in row:
                    if val is None:
                        values.append("NULL")
                    else:
                        values.append("'" + str(val).replace("'",
"''") + "'")
                insert_stmts += f'INSERT INTO "{selected_table}"
({", ".join(col_names)}) VALUES ({", ".join(values)});\n'

            sql = create_stmt + "\n" + insert_stmts

            path              =              os.path.join(folder,
f"{selected_table}_backup.sql")
            with open(path, "w", encoding="utf-8") as f:
                f.write(sql)

            messagebox.showinfo("Success",   f"Backup   saved   as
{path}")
        except Exception as e:
            messagebox.showerror("Error",    f"Backup    failed:
{str(e)}")

    def backup_db(self):
        if not self.conn:
            messagebox.showerror("Error",   "Not   connected   to
database")
            return

        folder = filedialog.askdirectory(title="Select Folder to
Save SQL Backup")
        if not folder:
            return

        try:
            self.cursor.execute(
                "SELECT table_name FROM information_schema.tables
WHERE table_schema = 'public'"
            )
            tables = [t[0] for t in self.cursor.fetchall()]

            sql_full = ""
            for table in tables:
                # CREATE
                self.cursor.execute(
                    f"""
                    SELECT 'CREATE TABLE "{table}" (' ||
```

```
                    string_agg('"' || column_name || '" ' ||
data_type, ', ') || ');'
                    FROM information_schema.columns
                    WHERE table_name = %s
                """,
                    (table,),
                )
                create_stmt = self.cursor.fetchone()[0]

                # INSERT
                self.cursor.execute(f"SELECT * FROM {table}")
                rows = self.cursor.fetchall()
                col_names   =   [f'"{desc[0]}"'   for   desc   in
self.cursor.description]

                insert_stmts = ""
                for row in rows:
                    values = []
                    for val in row:
                        if val is None:
                            values.append("NULL")
                        else:
                            values.append("'"                    +
str(val).replace("'", "''") + "'")
                    insert_stmts += f'INSERT INTO "{table}" ({",
".join(col_names)}) VALUES ({", ".join(values)});\n'

                sql_full += create_stmt + "\n" + insert_stmts +
"\n\n"

            path = os.path.join(folder, "database_backup.sql")
            with open(path, "w", encoding="utf-8") as f:
                f.write(sql_full)

            messagebox.showinfo("Success", f"Full database backup
saved as {path}")
        except Exception as e:
            messagebox.showerror("Error",    f"Backup    failed:
{str(e)}")

    def restore_table(self):
        file_path = filedialog.askopenfilename(
            title="Select   SQL   File   to   Restore   Table",
filetypes=[("SQL files", "*.sql")]
        )
        if not file_path:
            return

        try:
            with open(file_path, "r", encoding="utf-8") as f:
                sql_script = f.read()

            # Разделяем по ;
```

```
            statements = [
                stmt.strip() for stmt in sql_script.split(";") if
stmt.strip()
            ]
            for stmt in statements:
                self.cursor.execute(stmt)

            self.conn.commit()
            messagebox.showinfo("Success",     "Table     restored
successfully.")
            self.refresh_tables()
        except Exception as e:
            self.conn.rollback()
            messagebox.showerror("Error",    f"Restore    failed:
{str(e)}")

    def restore_db(self):
        file_path = filedialog.askopenfilename(
            title="Select SQL File to Restore Database",
            filetypes=[("SQL files", "*.sql")],
        )
        if not file_path:
            return

        try:
            with open(file_path, "r", encoding="utf-8") as f:
                sql_script = f.read()

            statements = [
                stmt.strip() for stmt in sql_script.split(";") if
stmt.strip()
            ]
            for stmt in statements:
                self.cursor.execute(stmt + ";")

            self.conn.commit()
            messagebox.showinfo("Success",   "Database   restored
successfully.")
            self.refresh_tables()
        except Exception as e:
            self.conn.rollback()
            messagebox.showerror("Error",    f"Restore    failed:
{str(e)}")

    def export_table(self):
        if not self.conn:  # Проверяем наличие соединения
            messagebox.showerror("Error",   "Not   connected   to
database")
            return

        selected_table = self.tables_listbox.get(tk.ACTIVE)

        if not selected_table:
```

```
            messagebox.showerror("Error", "No table selected")
            return

        backup_folder = filedialog.askdirectory(
            title="Select Folder to Save Export file"
        )
        if not backup_folder:
            return
        try:
            query = f"COPY {selected_table} TO STDOUT WITH (FORMAT
CSV, HEADER TRUE, DELIMITER ';', QUOTE '\"', ENCODING 'UTF8')"
            export_file = os.path.join(
                backup_folder,
f"{selected_table}_export_file.xlsx"
            )
            with open(export_file, "w", encoding="utf-8-sig") as
f:
                self.cursor.copy_expert(query, f)
            messagebox.showinfo(
                "Success",
                f"Export file for table {selected_table} saved
successfully at {export_file}",
            )
        except Exception as e:
            messagebox.showerror("Error",  f"Failed  to  export
table: {str(e)}")

    def export_db(self):
        if not self.conn:  # Проверяем наличие соединения
            messagebox.showerror("Error",  "Not  connected  to
database")
            return
        folder = filedialog.askdirectory(title="Select Folder to
Save Export file")
        if folder:
            try:
                self.cursor.execute(
                    "SELECT           table_name           FROM
information_schema.tables WHERE table_schema = 'public'"
                )
                tables = [t[0] for t in self.cursor.fetchall()]
                excel_path        =        os.path.join(folder,
"database_export_file.xlsx")
                with                  pd.ExcelWriter(excel_path,
engine="openpyxl") as writer:
                    for table in tables:
                        df  =  pd.read_sql_query(f'SELECT  *  FROM
"{table}"', self.conn)
                        df.to_excel(writer,
sheet_name=table[:31], index=False)
                messagebox.showinfo(
                    "Success",
```

```
                    f"Database export file saved successfully as
{excel_path}",
                )
            except Exception as e:
                messagebox.showerror("Error", f"Failed to export
database: {str(e)}")

    def import_table(self):
        file_path = filedialog.askopenfilename(
            title="Select Excel File",
            filetypes=[("Excel files", "*.xlsx *.xls")],
        )

        if file_path:
            try:
                # Загружаем все листы
                excel_data       =       pd.read_excel(file_path,
sheet_name=None)
                table_names = list(excel_data.keys())

                selected_table = simpledialog.askstring(
                    "Table Selection",
                    f"Available                          tables:\n{',
'.join(table_names)}\n\nEnter the table name",
                )
                if   selected_table   and   selected_table   in
excel_data:
                    self._import_table_from_df(
                        selected_table,
excel_data[selected_table]
                    )
                else:
                    messagebox.showwarning(
                        "Cancel", "No table selected or the name
is incorrect."
                    )
                    return

                self.conn.commit()
                messagebox.showinfo("Success", "Import completed
successfully.")
                self.refresh_tables()

            except Exception as e:
                self.conn.rollback()
                messagebox.showerror("Error",   f"Import   failed:
{str(e)}")

    def import_db(self):
        file_path = filedialog.askopenfilename(
            title="Select Excel File",
            filetypes=[("Excel files", "*.xlsx *.xls")],
        )
```

```
        if file_path:
            try:
                # Загружаем все листы
                excel_data       =       pd.read_excel(file_path,
sheet_name=None)

                # Восстановление всей базы
                for table_name, df in excel_data.items():
                    self._import_table_from_df(table_name, df)

                self.conn.commit()
                messagebox.showinfo("Success", "Import completed
successfully.")
                self.refresh_tables()

            except Exception as e:
                self.conn.rollback()
                messagebox.showerror("Error",   f"Import   failed:
{str(e)}")

    def _import_table_from_df(self, table_name, df):
        self.cursor.execute(f'DROP      TABLE      IF      EXISTS
"{table_name}"')
        columns   =   ",   ".join([f'"{col}"   TEXT'   for   col   in
df.columns])
        self.cursor.execute(f'CREATE     TABLE     "{table_name}"
({columns})')
        for _, row in df.iterrows():
            placeholders = ", ".join(["%s"] * len(row))
            columns   =   ",   ".join([f'"{col}"'   for   col   in
df.columns])
            insert_query = (
                f'INSERT  INTO  "{table_name}"  ({columns})  VALUES
({placeholders})'
            )
            self.cursor.execute(insert_query, tuple(row))

    def execute_query(self):
        query_window = tk.Toplevel(self.root)
        query_window.title("Execute SQL Query")
        query_window.geometry("700x550")  # Фиксированный размер
окна

        query_map = {}

        def load_queries_from_file(filename):
            try:
                with open(filename, "r", encoding="utf-8") as f:
                    for line in f:
                        if "===" in line:
                            name, sql = line.strip().split("===",
1)
```

```
                            query_map[name] = sql
            except FileNotFoundError:
                pass

        load_queries_from_file("queries.txt")
        query_names = list(query_map.keys())
        selected_query = tk.StringVar()
        selected_query.set(query_names[0] if query_names else "")

        # Часть 1: Панель управления
        control_frame        =        tk.LabelFrame(query_window,
text="Controls", padx=10, pady=10)
        control_frame.grid(row=0,  column=0,  padx=10,  pady=10,
sticky="ew")
        control_frame.grid_columnconfigure((0,  1,  2,  3,  4),
weight=1)

        btn_width = 18

        tk.Button(
            control_frame,
            text="Execute",
            width=btn_width,
            command=lambda:  self.run_query(query_text.get("1.0",
tk.END), result_tree),
        ).grid(row=0, column=0, padx=5, pady=5)
        tk.Button(
            control_frame,
            text="Save Query",
            width=btn_width,
            command=lambda: self.save_query(query_text.get("1.0",
tk.END)),
        ).grid(row=0, column=1, padx=5, pady=5)
        tk.Button(
            control_frame,
            text="Export Results",
            width=btn_width,
            command=lambda: self.export_results(result_tree),
        ).grid(row=0, column=2, padx=5, pady=5)

        query_menu = tk.OptionMenu(control_frame, selected_query,
*query_names)
        query_menu.config(width=btn_width - 2)
        query_menu.grid(row=0, column=3, padx=5, pady=5)

        tk.Button(
            control_frame,
            text="Load Query",
            width=btn_width,
            command=lambda: [
                query_text.delete("1.0", tk.END),
                query_text.insert(tk.END,
query_map[selected_query.get()]),
```

```
            ],
        ).grid(row=0, column=4, padx=5, pady=5)

        # Часть 2: Редактор запросов
        query_frame = tk.LabelFrame(query_window, text="Query
Editor", padx=10, pady=10)
        query_frame.grid(row=1, column=0, padx=10, pady=10,
sticky="nsew")
        query_frame.grid_rowconfigure(0, weight=1)
        query_frame.grid_columnconfigure(0, weight=1)

        query_text = ScrolledText(query_frame, width=100,
height=10)
        query_text.grid(row=0, column=0, sticky="nsew")

        # Часть 3: Результаты
        result_frame = tk.LabelFrame(query_window,
text="Results", padx=10, pady=10)
        result_frame.grid(row=2, column=0, padx=10, pady=10,
sticky="nsew")
        result_frame.grid_rowconfigure(0, weight=1)
        result_frame.grid_columnconfigure(0, weight=1)

        result_tree = Treeview(result_frame, show="headings",
height=15)
        result_tree.grid(row=0, column=0, sticky="nsew")

        scrollbar = tk.Scrollbar(
            result_frame, orient="vertical",
command=result_tree.yview
        )
        scrollbar.grid(row=0, column=1, sticky="ns")
        result_tree.configure(yscrollcommand=scrollbar.set)

        # Настройка растяжения
        query_window.grid_rowconfigure(1, weight=1)
        query_window.grid_rowconfigure(2, weight=2)
        query_window.grid_columnconfigure(0, weight=1)

    def run_query(self, query, tree):
        query = query.strip()
        if not query:
            messagebox.showerror("Error", "Query is empty!")
            return
        try:
            for item in tree.get_children():
                tree.delete(item)
            tree["columns"] = ()

            self.cursor.execute(query)
            self.conn.commit()

            if query.upper().startswith("SELECT"):
```

```
                rows = self.cursor.fetchall()
                if not rows:
                    messagebox.showinfo(
                        "Info", "Query executed, but no results
returned."
                    )
                    return

                columns     =     [desc[0]     for     desc     in
self.cursor.description]
                tree["columns"] = columns
                for col in columns:
                    tree.heading(col, text=col)
                    tree.column(col, width=120, anchor="w")
                for row in rows:
                    tree.insert("", tk.END, values=row)

                logging.info(f"Executed SELECT: {query}")
            else:
                messagebox.showinfo("Success",   "Query   executed
successfully.")
                logging.info(f"Executed: {query}")
        except Exception as e:
            self.conn.rollback()
            messagebox.showerror("Error",   f"Failed   to   execute
query:\n{str(e)}")

    def save_query(self, query):
        if not query.strip():
            messagebox.showerror("Error", "Query is empty!")
            return

        name = simpledialog.askstring("Save Query", "Enter query
name/description:")
        if not name:
            return

        # Проверка на повтор
        if os.path.exists("queries.txt"):
            with open("queries.txt", "r", encoding="utf-8") as f:
                lines = f.readlines()
                for line in lines:
                    if line.startswith(name + "==="):
                        messagebox.showwarning("Warning",  "Query
name already exists!")
                        return
        with open("queries.txt", "a", encoding="utf-8") as f:
            f.write(f"\n{name}==={query.strip()}\n")

        messagebox.showinfo("Success",        "Query        saved
successfully.")

    def export_results(self, tree):
```

```
        if not tree.get_children():
            messagebox.showerror("Error",    "No    results    to
export!")
            return
        folder = filedialog.askdirectory(title="Select Folder to
Save Excel File")
        if folder:
            try:
                df = pd.DataFrame(
                    [tree.item(item)["values"]   for   item   in
tree.get_children()],
                    columns=tree["columns"],
                )
                excel_path         =         os.path.join(folder,
"query_results.xlsx")
                df.to_excel(excel_path, index=False)
                messagebox.showinfo("Success", "Results exported
to " + excel_path)
            except Exception as e:
                messagebox.showerror("Error", f"Failed to export
results: {str(e)}")

    def export_to_excel(self):
        selected_table = self.tables_listbox.get(tk.ACTIVE)
        if selected_table:
            export_folder = filedialog.askdirectory(
                title="Select Folder to Export Excel File"
            )
            if export_folder:
                try:
                    query = f"SELECT * FROM {selected_table};"
                    self.cursor.execute(query)
                    rows = self.cursor.fetchall()

                    df = pd.DataFrame(
                        rows,   columns=[desc[0]   for   desc   in
self.cursor.description]
                    )
                    excel_path    =    os.path.join(export_folder,
f"{selected_table}.xlsx")
                    df.to_excel(excel_path, index=False)
                    messagebox.showinfo(
                        "Success",
                        f"Table   {selected_table}   exported   to
Excel successfully",
                    )
                except Exception as e:
                    messagebox.showerror(
                        "Error",  f"Failed  to  export  table  to
Excel: {str(e)}"
                    )

    def __del__(self):
```

```
        if self.conn:
            self.cursor.close()
            self.conn.close()


if __name__ == "__main__":
    root = tk.Tk()
    app = DatabaseApp(root)
    root.mainloop()
```