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 3](#_Toc195867673)

[1 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 4](#_Toc195867674)

[1.1 Системные требования 4](#_Toc195867675)

[1.2 Обзор интерфейса приложения 4](#_Toc195867676)

[1.3 Отображение таблицы 4](#_Toc195867677)

[1.4 Редактирование, добавление и удаление записей таблицы 6](#_Toc195867678)

[1.5 Добавление таблицы 6](#_Toc195867679)

[1.6 Удаление таблицы из базы данных 7](#_Toc195867680)

[1.7 SQL запросы 8](#_Toc195867681)

[1.8 Экспорт в Excel 10](#_Toc195867682)

[1.9 Создание резервной копии 10](#_Toc195867683)

[ЗАКЛЮЧЕНИЕ 12](#_Toc195867684)

[ПРИЛОЖЕНИЕ А 13](#_Toc195867685)

**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. **Системные требования**

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

* 1. **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.

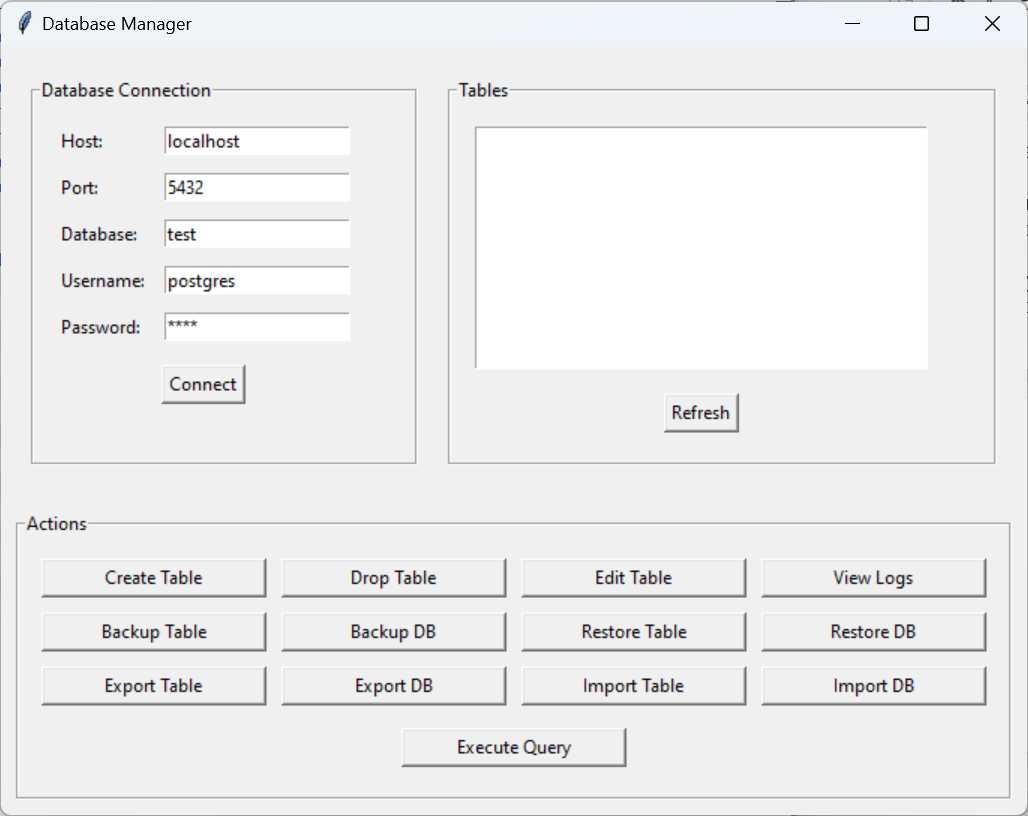


Рисунок 1.1 – Интерфейс приложения

* 1. **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.

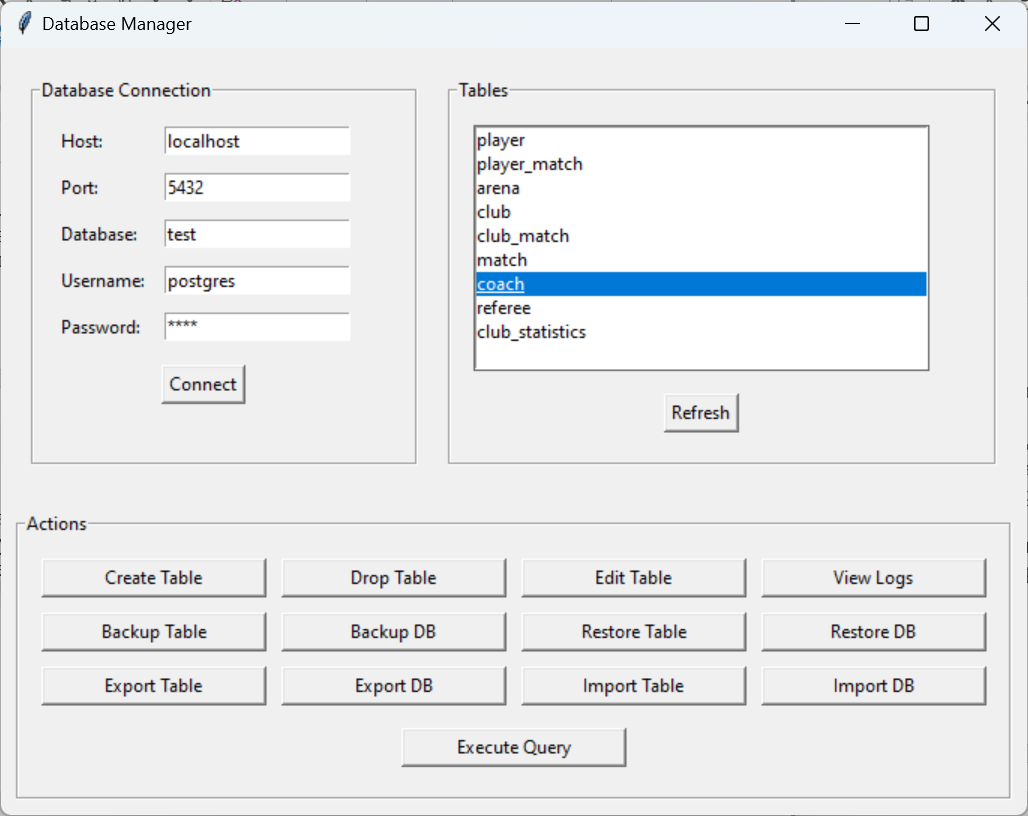


Рисунок 1.2 – Выбор таблицы

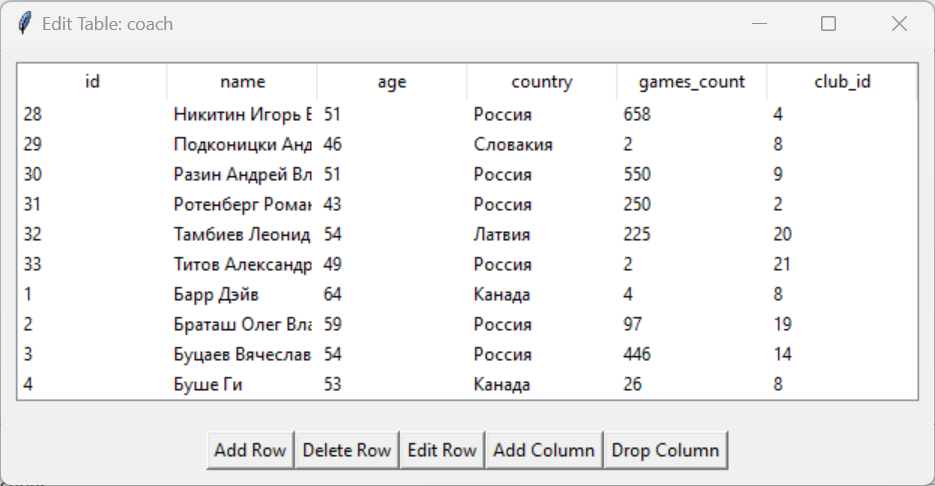


Рисунок 1.3 – Вывод таблицы coach

* 1. **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.

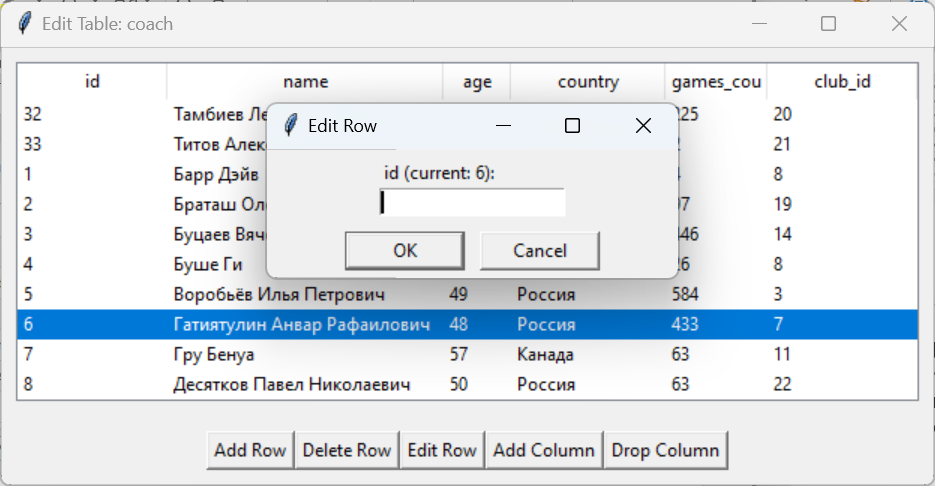


Рисунок 1.4 – Окно редактирования записи в таблице

* 1. **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.

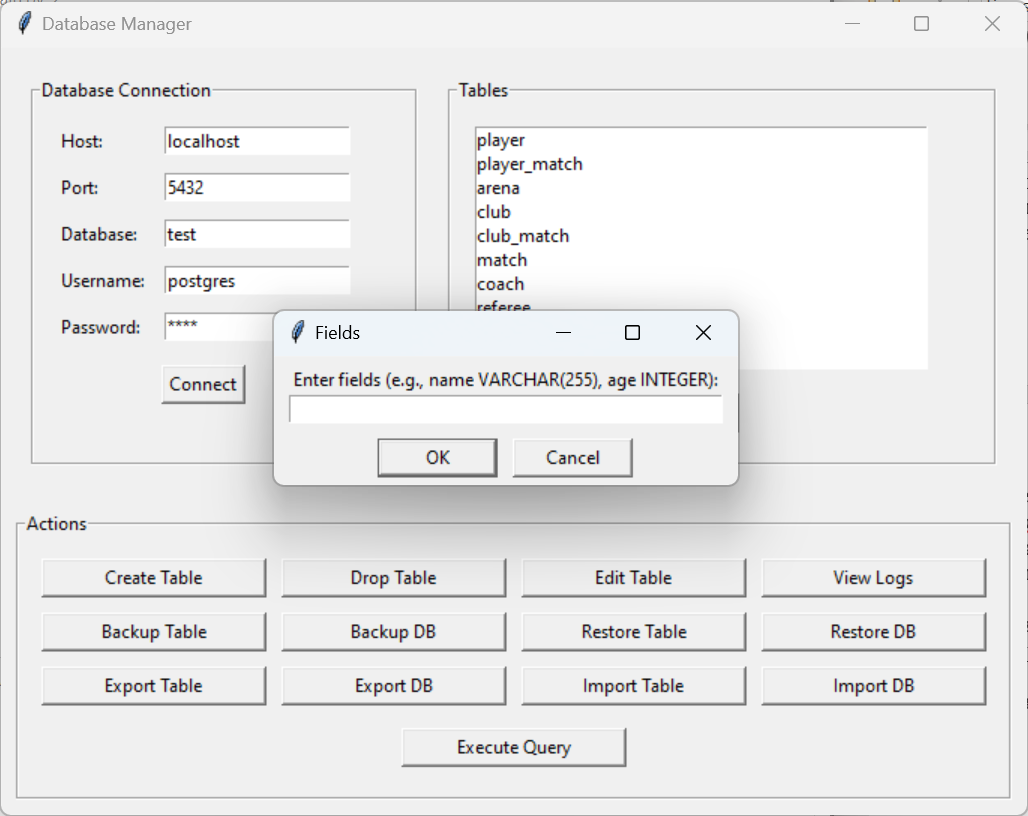


Рисунок 1.5 – Окно создания таблицы

* 1. **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.

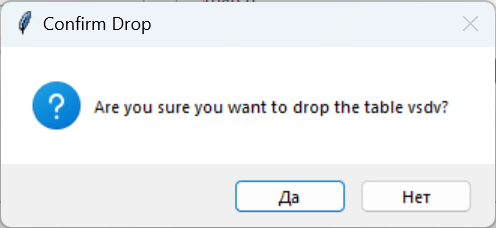


Рисунок 1.6 – Подтверждение удаления таблицы vsdv

* 1. **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.

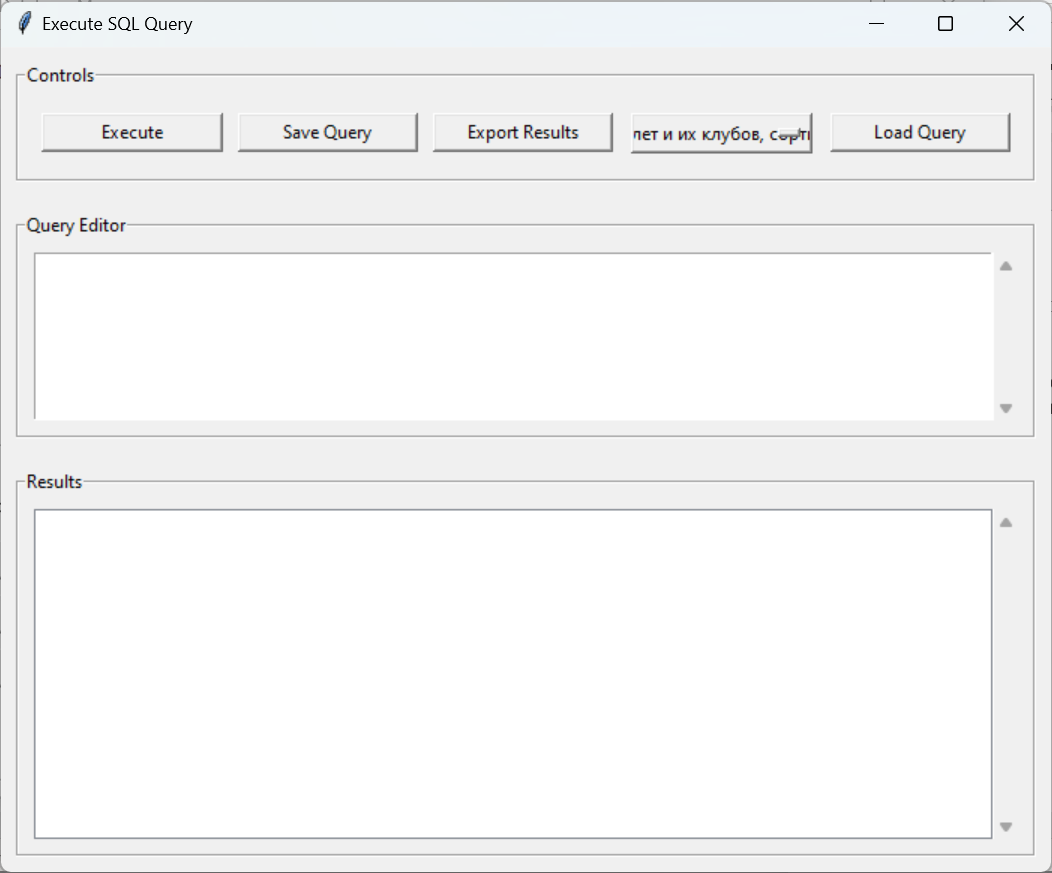


Рисунок 1.7 – Область для работы с SQL запросами

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.

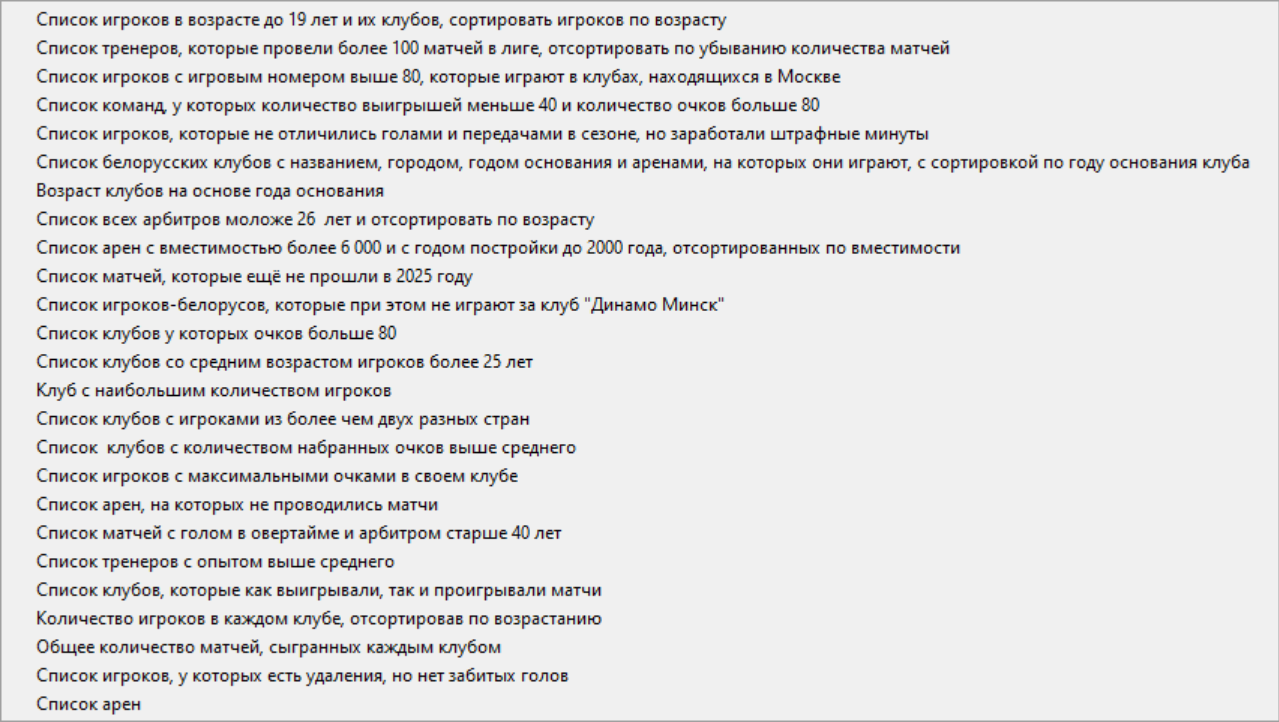


Рисунок 1.8 – Список скриптом из лабораторных работ №4 и №5

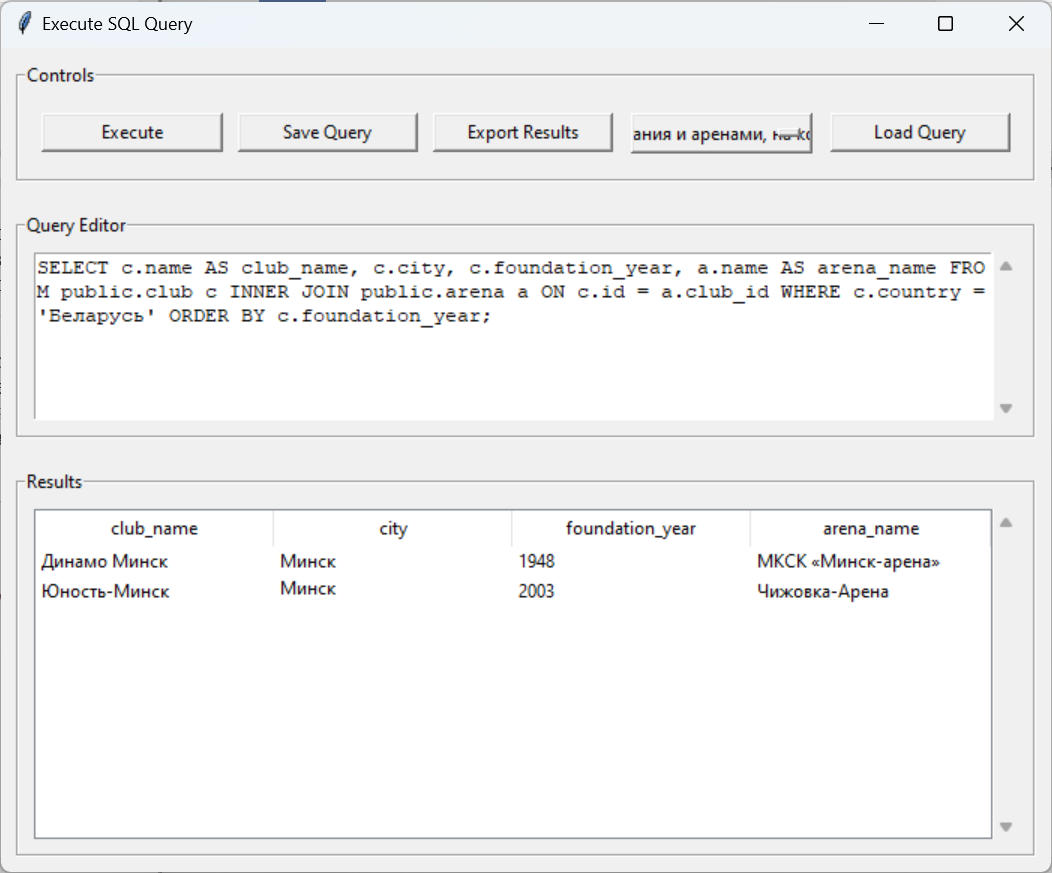


Рисунок 1.9 – Область SQL скриптов после выбора скрипта из списка

* 1. **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.

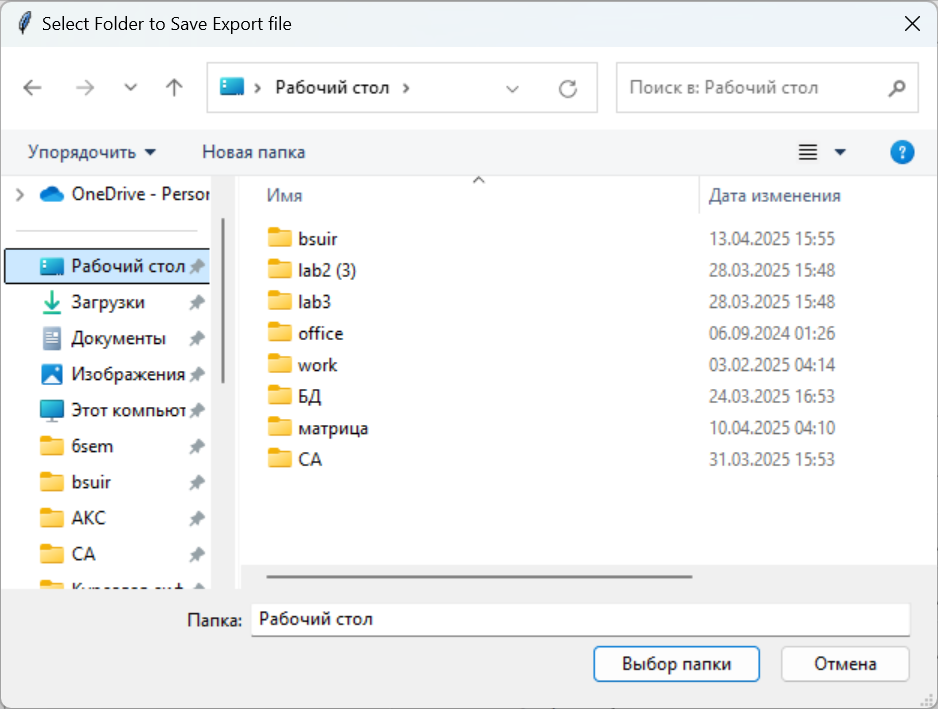


Рисунок 1.10 – Выбор места для экспорта таблицы

* 1. **Создание резервной копии**

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

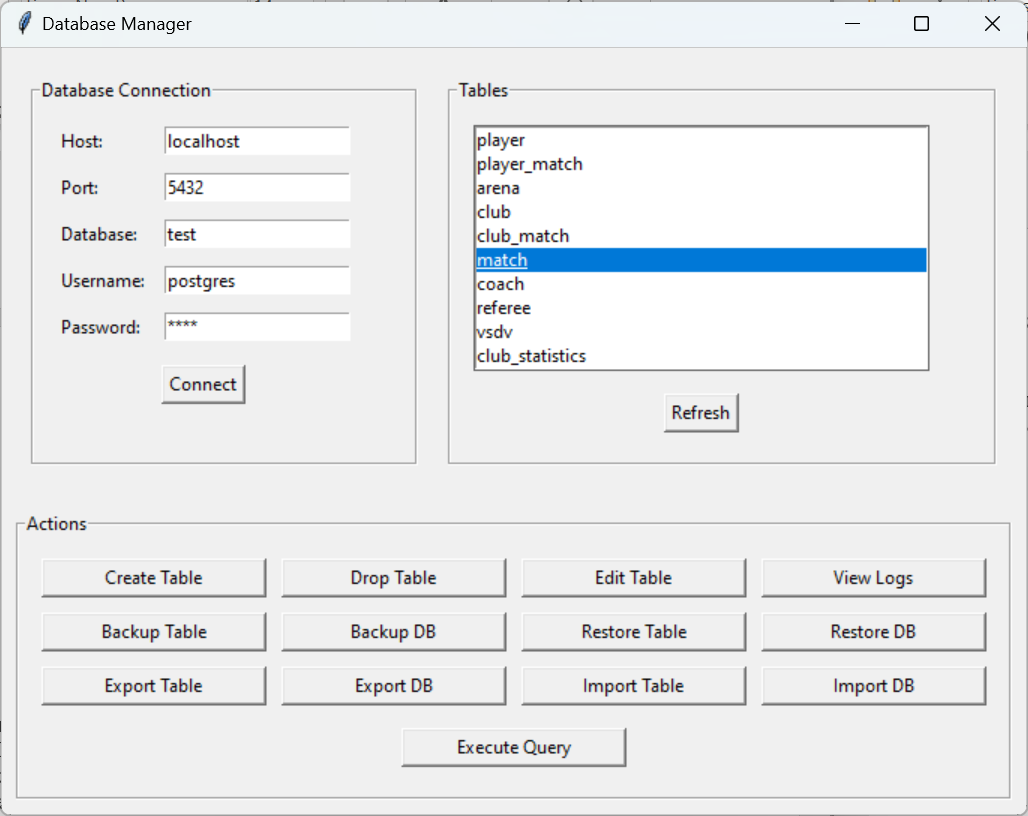


Рисунок 1.11 – Кнопки резервного копирования

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()