Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОННИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Базы данных

Тема «Столовая Лидо»

Лабораторная работа №6

Создание прикладной программы для работы с базой данных

Студент: М.С. Патюпин

Преподаватель: Д.В. Куприянова

МИНСК 2025

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc195096873)

[1 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 4](#_Toc195096874)

[1.1 Добавление новой таблицы 4](#_Toc195096875)

[1.2 Удаление существующей таблицы 5](#_Toc195096876)

[1.3 Работа с таблицей 6](#_Toc195096877)

[1.4 Создание резервных копий, восстановление 9](#_Toc195096878)

[1.5 Исполнение запросов 11](#_Toc195096879)

[1.6 Экспорт данных 12](#_Toc195096880)

[ЗАКЛЮЧЕНИЕ 14](#_Toc195096881)

[ПРИЛОЖЕНИЕ А 15](#_Toc195096882)

# ВВЕДЕНИЕ

Данная лабораторная работа нацелена на изучение работы с базами данных со стороны прикладного приложения (язык программирования python).

Приложения должно соответствовать следующим требованиям:

– выполнять заданные транзакции;

– возможность добавление новой таблицы;

– возможность удаление существующих таблиц;

– работа с таблицами (редактирование, удаление, добавление полей);

– создание резервной копии для восстановления удаленной таблицы, строк, всей базы данных;

– вывод созданных новых запросов и возможность их сохранения для последующего использования;

– предусмотреть механизм экспорта каждой таблицы, результатов запросов в файл.

Реализовать управление через многостраничный графический интерфейс, включающий в себя:

– окно с запросами;

– окно для просмотра каждой таблицы;

–окно просмотра результата запроса.

# РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Исходный код программы приведен в приложении А.

## Добавление новой таблицы

Для добавления новой таблицы, необходимо выполнить следующие шаги:

1 Запустить приложение.

2 Нажать кнопку «Добавить таблицу» – рисунок 1.1.

3 В открывшимся окне вписать имя новой таблицы на латинице и добавить необходимые столбцы, нажать сохранить – рисунок 1.2.

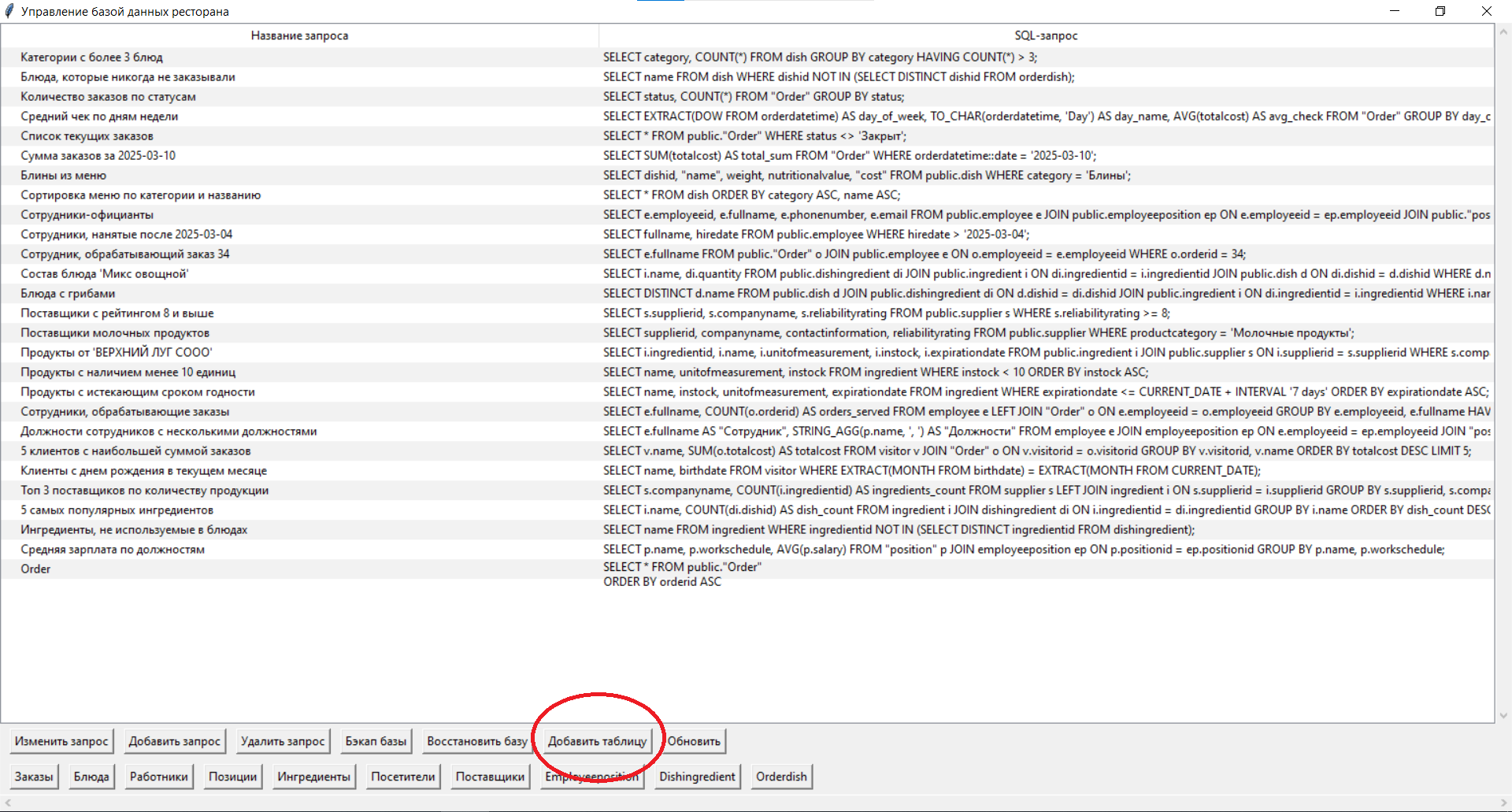


Рисунок 1.1 –Добавление таблицы

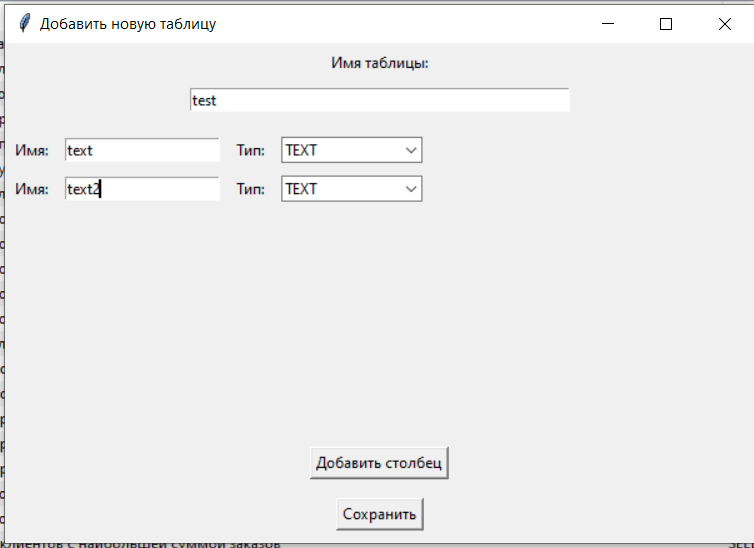


Рисунок 1.2 –Конфигурирование таблицы

## Удаление существующей таблицы

Для удаления существующей таблицы необходимо:

1 Обновить приложение – нажать на кнопку «Обновить», приложение перезапустится, рисунок 1.3.

2 Выбрать желаемую таблицу и нажать по ней, откроется меню редактирования таблицы.

3 Нажать удаление таблицы, рисунок 1.4.

4 Подтвердить, рисунок 1.5.

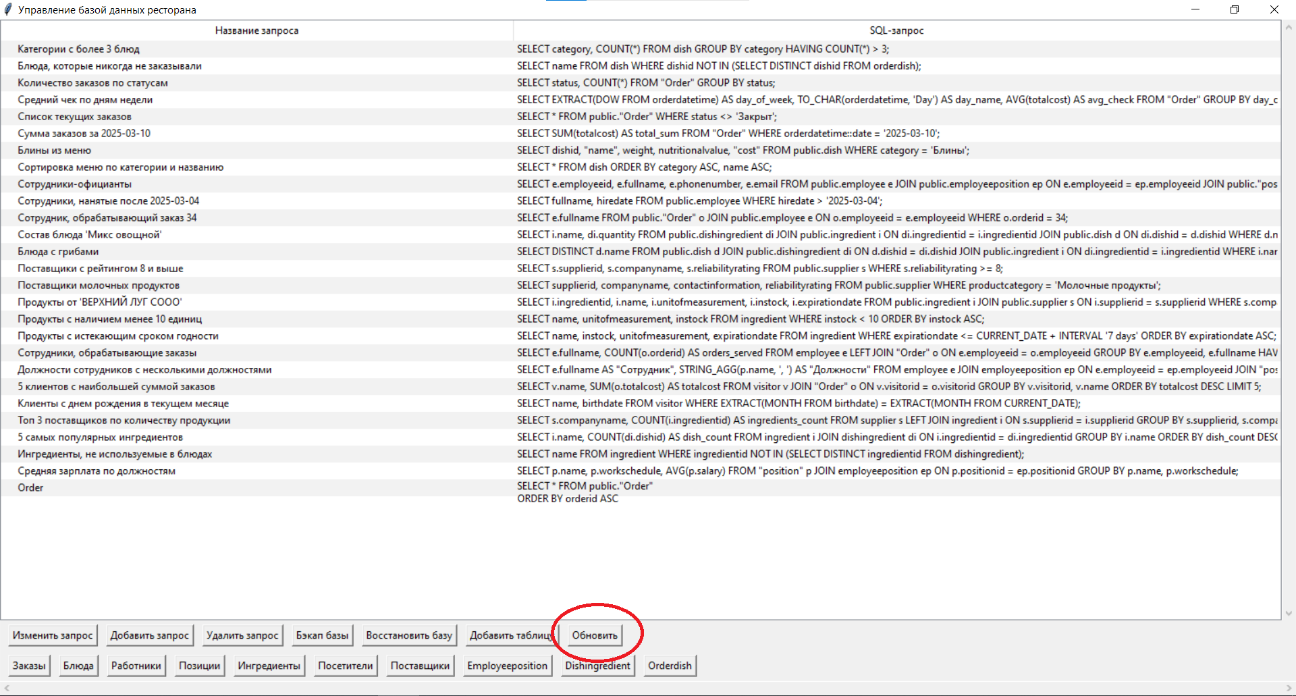


Рисунок 1.3 – Обновление приложения

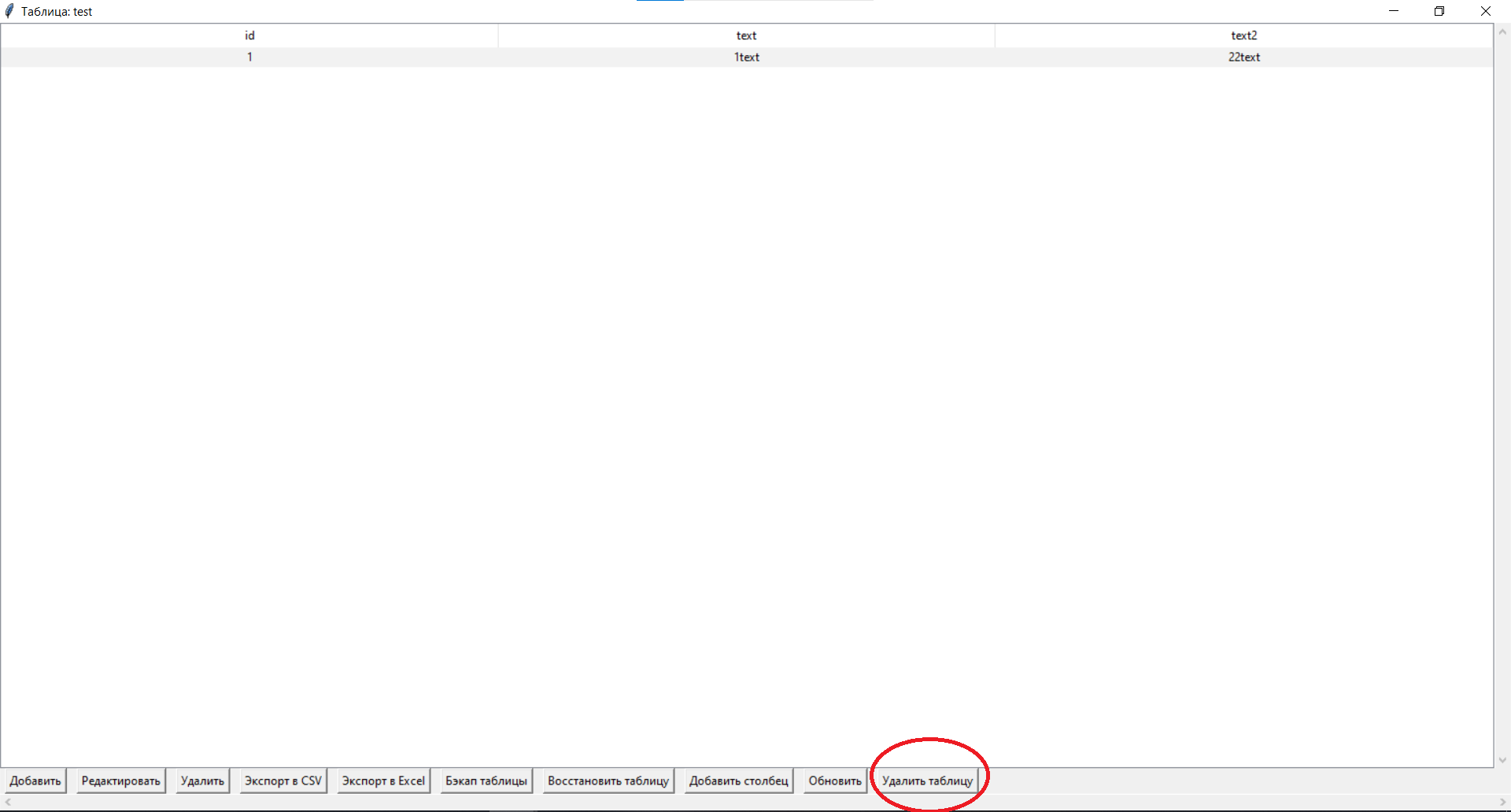


Рисунок 1.4 – Удаление таблицы

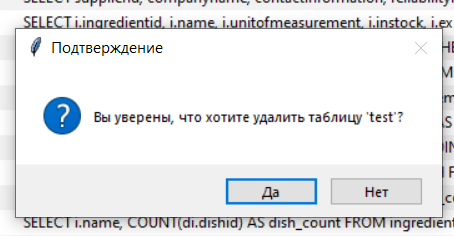


Рисунок 1.5 –Подтверждение удаления

## Работа с таблицей

Для добавления новых строк необходимо:

1 Обновить приложение – нажать на кнопку «Обновить», приложение перезапустится, рисунок 1.3.

2 Выбрать желаемую таблицу и нажать по ней, откроется меню редактирования таблицы.

3 Нажать на кнопку «Добавить», рисунок 1.6.

4 В предложенном окне заполнить данные, рисунок 1.7, нажать «Сохранить».

5 Выполнить обновление таблицы, рисунок 1.8.

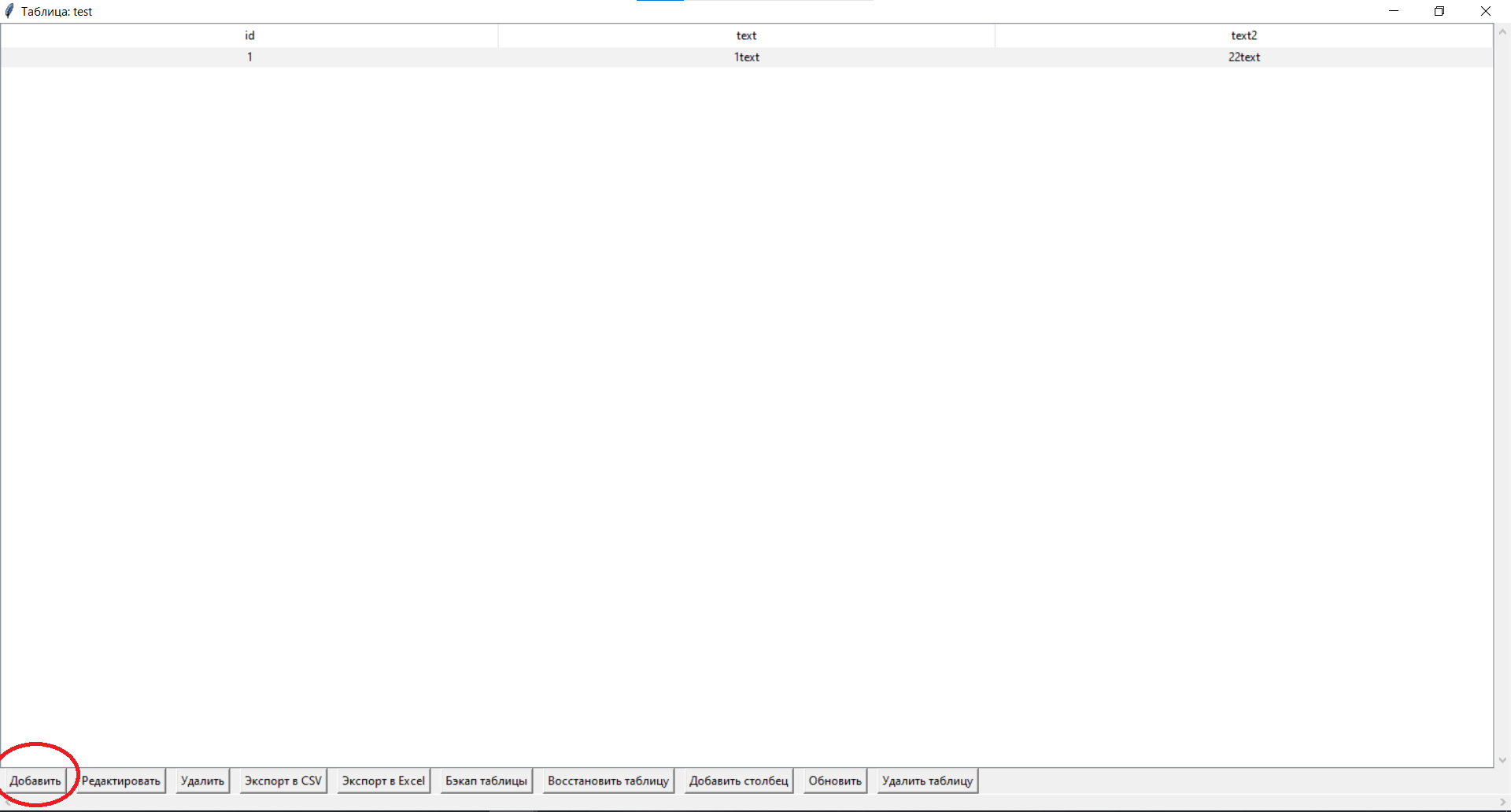


Рисунок 1.6 – Добавить строку

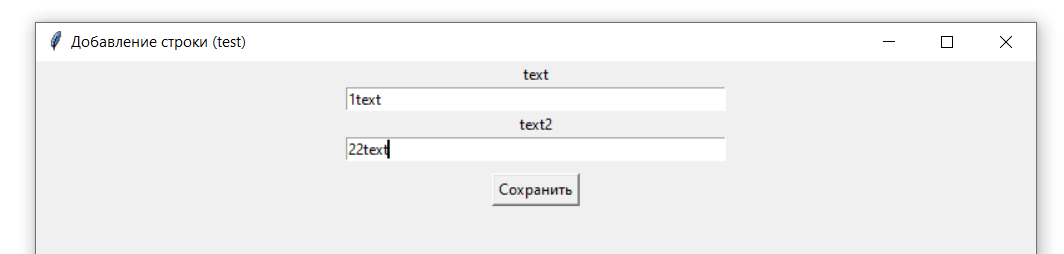


Рисунок 1.7 – Добавление строки

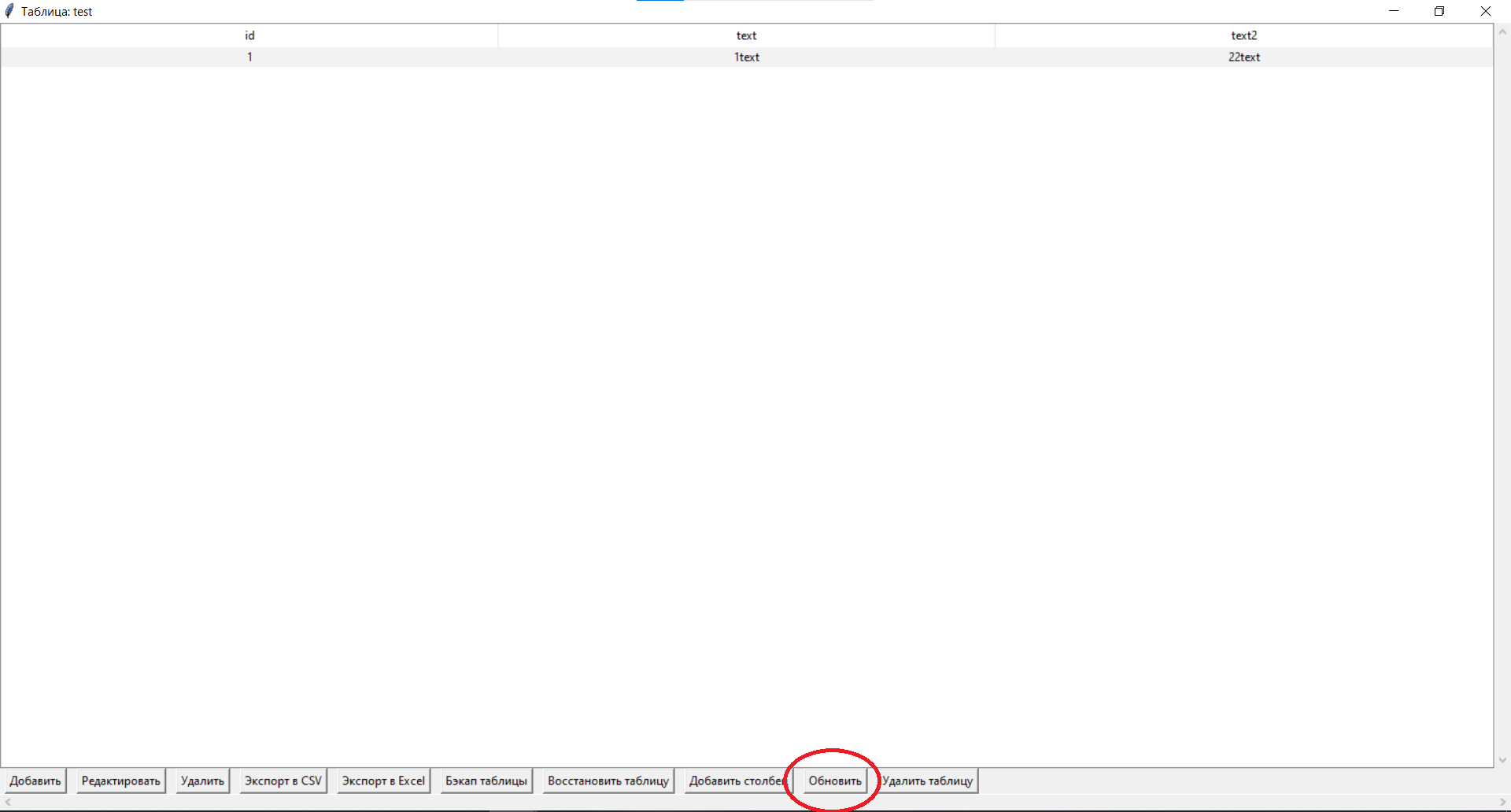


Рисунок 1.8 –Обновить таблицу

Для редактирования строк необходимо выполнить двойное нажатие по интересующей клетке в таблице – рисунок 1.9, или выбрать строку, и нажать редактирование – рисунок 1.10, и внести новые данных в открывшиеся окно, нажать сохранить – рисунок 1.11.

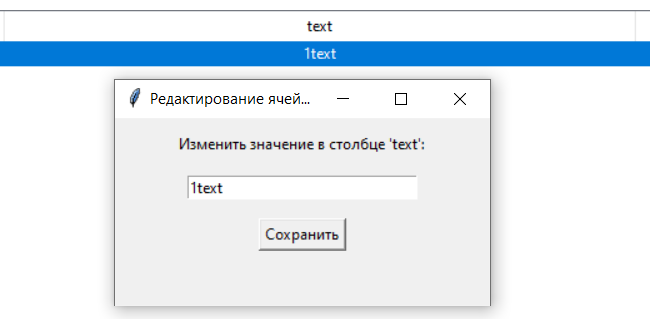


Рисунок 1.9 – Редактирование клетки двойным нажатием

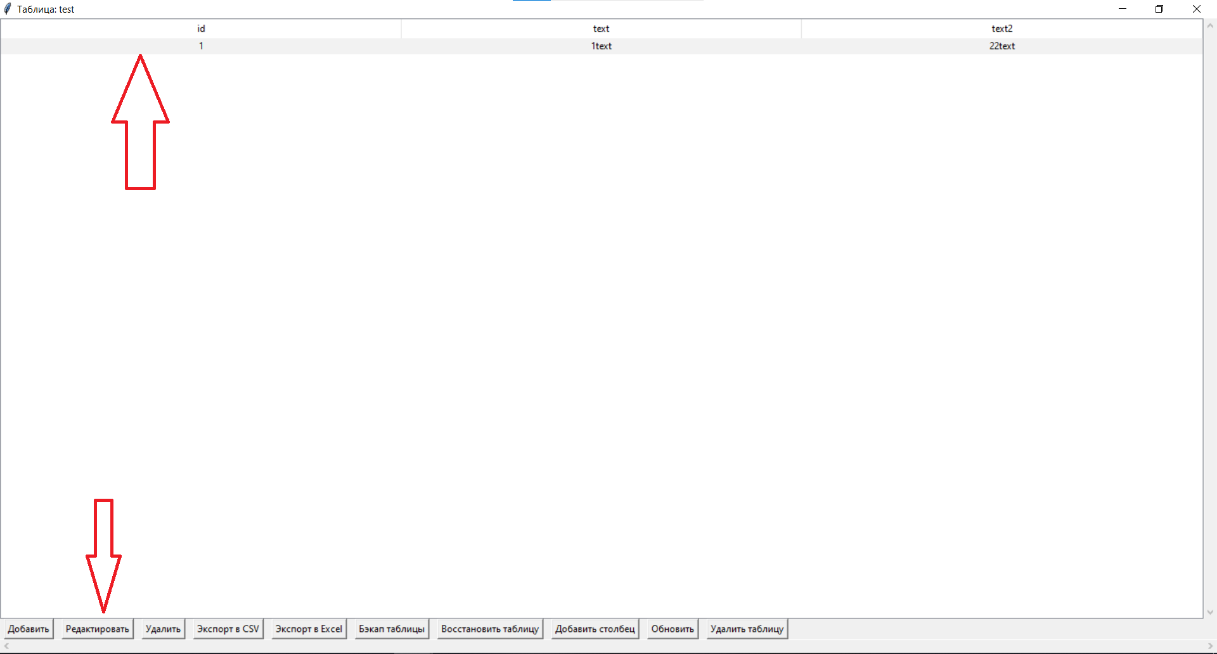


Рисунок 1.10 –Редактирование строки

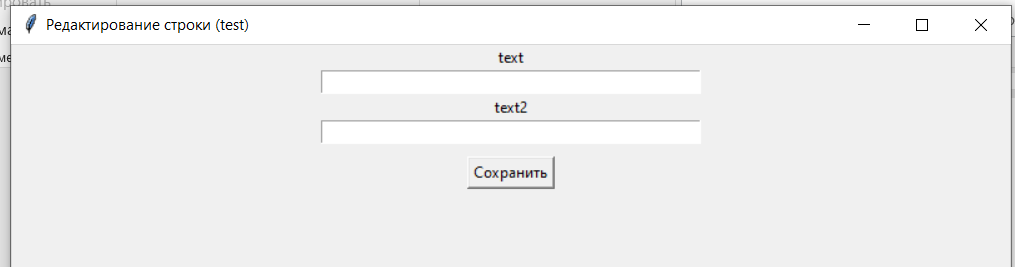


Рисунок 1.11 – Внесение новых данных строки

Для удаления строки необходимо:

1 Обновить таблицу.

2 Выделить строку нажатием, нажать «Удалить» –рисунок 1.12.

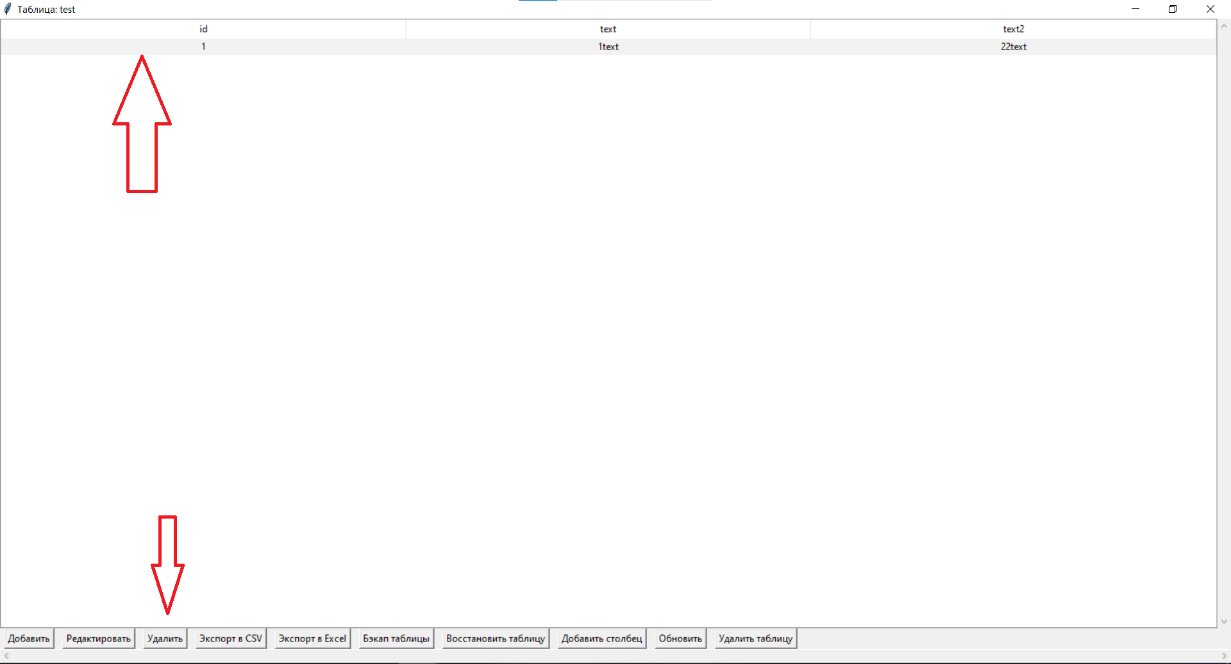


Рисунок 1.12 –Удаление строки

Для редактирования столбца необходимо выполнить двойное нажатие, и по выбору либо переименовать столбец или удалить, рисунок 1.13.

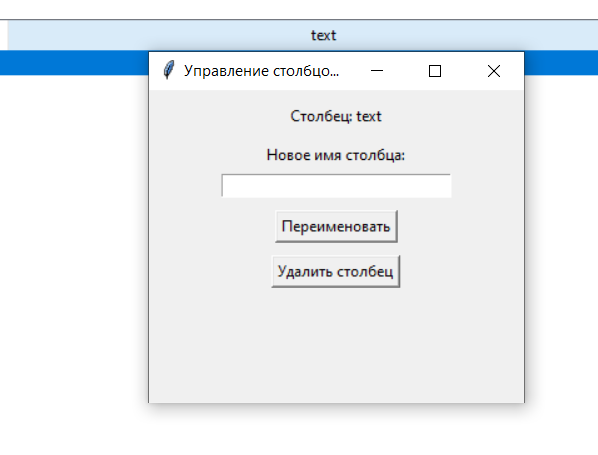


Рисунок 1.13 –Изменение столбца

## Создание резервных копий, восстановление

Для создания резервной копии базы данных необходимо на главном экране приложения нажать кнопку «Бэкап базы» – рисунок 1.14, после чего будет предложено выбрать место сохранения, название будет содержать дату и время выполнения команды, формат файла «.sql».

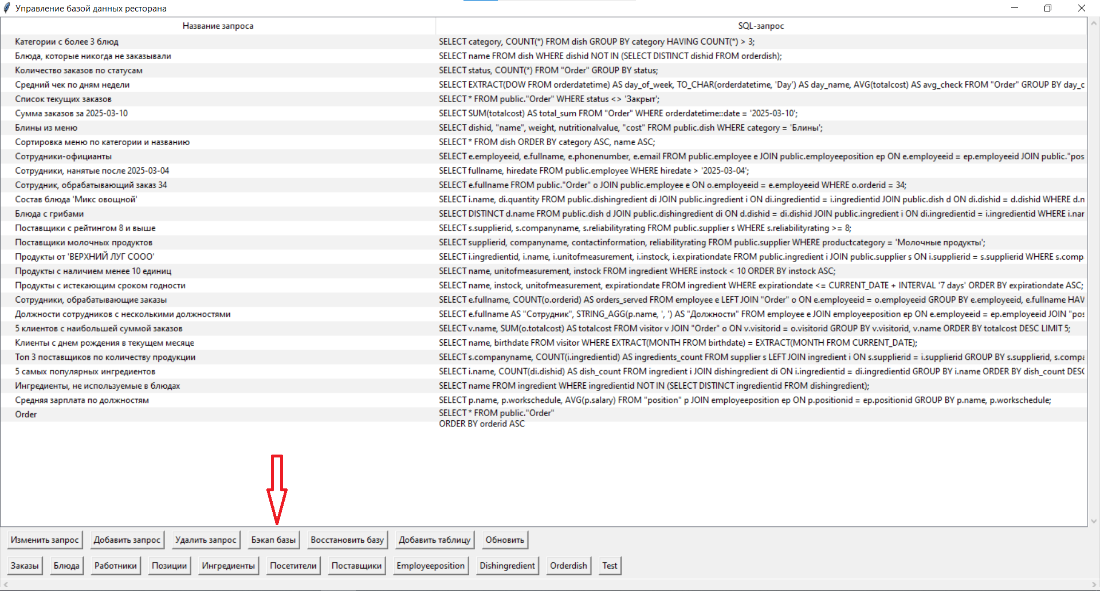


Рисунок 1.14 –Создание резервной копии базы данных

Для восстановления из резервной копии необходимо нажать на главном экране кнопку «Восстановить базу» –рисунок 1.15 и выбрать необходимый файл для восстановления.

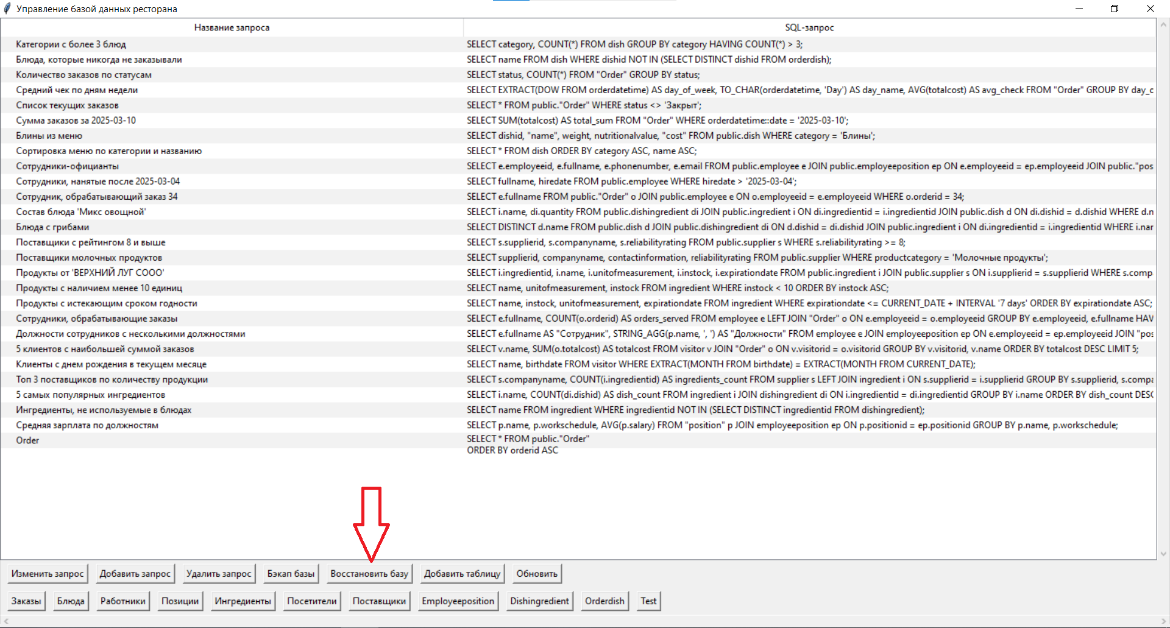


Рисунок 1.15 –Восстановить базу данных

Для сохранения данных таблиц необходимо:

1 Обновить приложение – нажать на кнопку «Обновить», приложение перезапустится, рисунок 1.3.

2 Выбрать желаемую таблицу и нажать по ней, откроется меню редактирования таблицы.

3 Выбрать кнопку «Бэкап таблицы», и сохранить по необходимому пути, рисунок 1.16, 1.17.

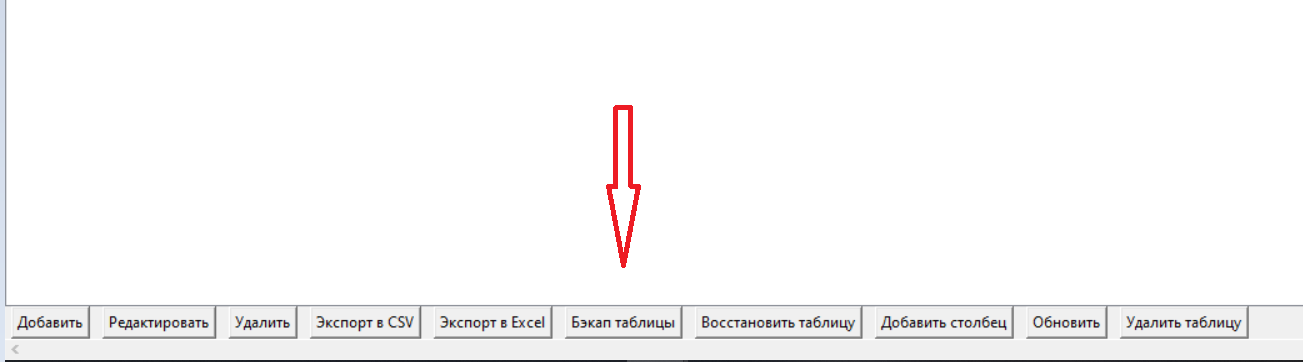


Рисунок 1.16 –Резервное сохранение таблицы

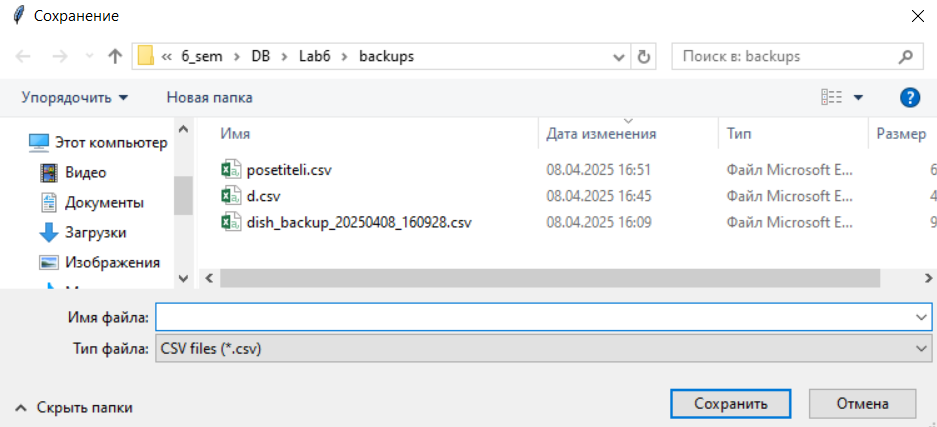


Рисунок 1.17 –Пример выбора пути

Для восстановления данных таблицы, необходимо открыть интересующею таблицу и нажать «Восстановить таблицу» – рисунок 1.18, выбрать нужный файл восстановления.

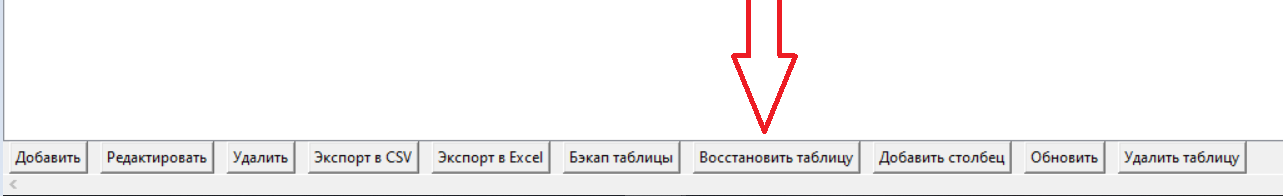


Рисунок 1.18 – восстановление таблицы

## Исполнение запросов

Для исполнения существующих запросов, необходимо на главном экране выбрать интересующий запрос и выполнить двойное нажатие, после чего откроется результат выполнения, рисунок 1.19. Есть возможность экспорта результата в формате «CSG» и «Exсel».

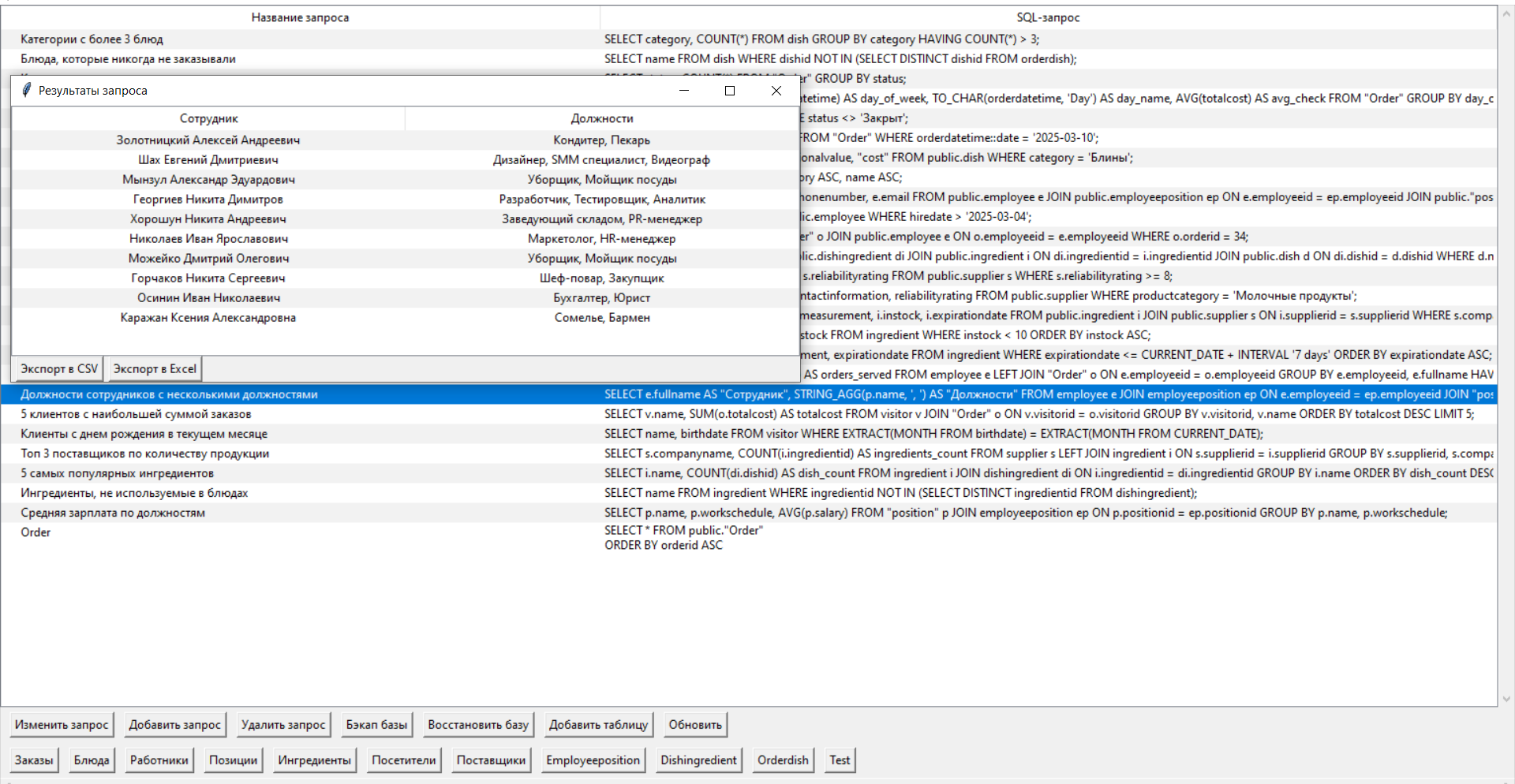


Рисунок 1.19 – Выполнение существующего запроса

Для создания нового запроса или изменения, существующего можно воспользоваться соответствующими кнопками на панели управления, рисунок 1.20. На рисунке 1.21 вид редактора запроса.

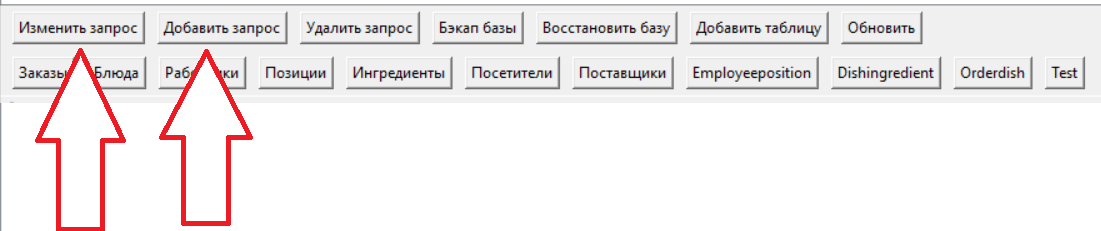


Рисунок 1.20 – Изменение запросов

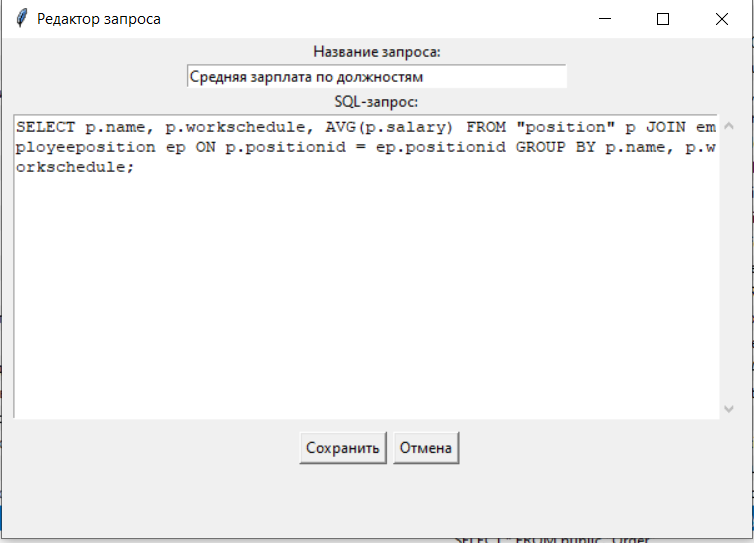


Рисунок 1.21 –Окно редактора запроса

## Экспорт данных

Экспорт доступен для любых запросов и таблиц. Экспорт производится в файлы двух форматов: «SVG» и «Exсel». Кнопки экспорта распологаются на соответствующих тулбарах – рисунки 1.22, 1.25.

На русинках 1.23, 1.24 приведены снимки результата экспорта для запроса, результаты экспорта для таблиц анаологичные.

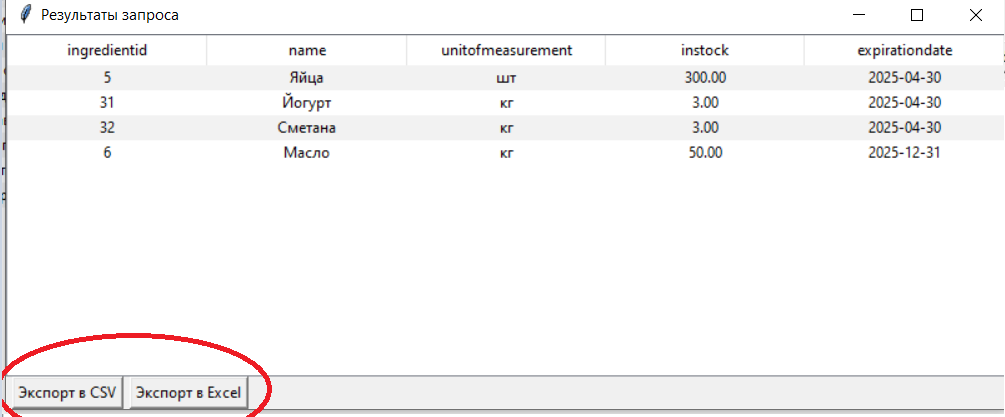


Рисунок 1.22 – Кнопки экспорта результата запроса

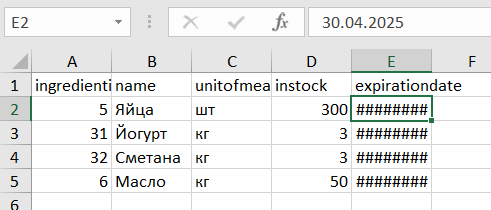
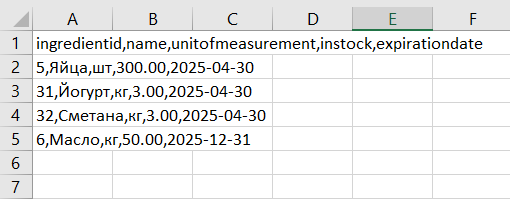


Рисунок 1.23 –Результат экспорта в «Exсel»



1.24 –Результат экспорта в «CSV»

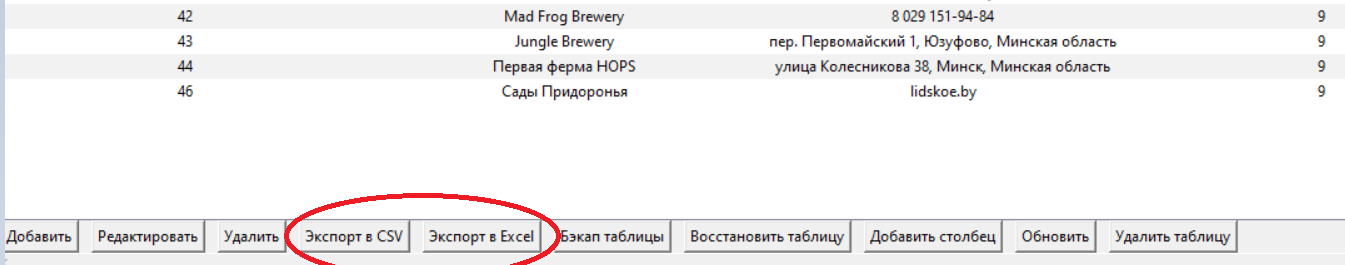


Рисунок 1.25 – Кнопки экспорта таблицы

# ЗАКЛЮЧЕНИЕ

Разработанное программа представляет собой современное и эффективное решение для работы с реляционными базами данных, сочетающее в себе простоту использования и мощный функционал

Благодаря интуитивно понятному графическому интерфейсу, приложение позволяет значительно упростить и ускорить выполнение рутинных операций, таких как просмотр, редактирование, добавление и удаление данных, без необходимости написания сложных SQL-запросов вручную.

Важным преимуществом программа является его способность экономить время и снижать количество ошибок, связанных с ручным вводом данных. Автоматизация многих процессов и продуманный интерфейс минимизируют риски возникновения неточностей, что особенно критично при работе с большими объемами информации. Приложение не только упрощает взаимодействие с базами данных, но и повышает продуктивность пользователей, позволяя им сосредоточиться на анализе данных, а не на технических деталях их обработки.

Таким образом, данное приложение не только отвечает актуальным требованиям к работе с базами данных, но и задает новый стандарт удобства и эффективности. Оно является примером того, как современные технологии могут сделать сложные процессы доступными и понятными для пользователей с разным уровнем подготовки, способствуя повышению качества и скорости работы с информацией.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Исходный код программы**

Файл app.py:

import tkinter as tk

from tkinter import ttk, messagebox, filedialog, scrolledtext

import psycopg2

import csv

import json

from datetime import datetime

import os

import subprocess

import threading

# Класс для работы с базой данных

class DBManager:

    def \_\_init\_\_(self, config):

        self.config = config

        self.conn = None

        self.connect()

        self.backup\_dir = "backups/"

        os.makedirs(self.backup\_dir, exist\_ok=True)

    def connect(self):

        # Подключение к базе данных

        try:

            self.conn = psycopg2.connect(\*\*self.config)

        except Exception as e:

            messagebox.showerror("Ошибка подключения", str(e))

    def execute\_query(self, query, params=None):

        # Выполнение SQL-запроса

        try:

            with self.conn.cursor() as cur:

                cur.execute(query, params or ())

                if cur.description:

                    return cur.fetchall(), [desc[0] for desc in cur.description]

                self.conn.commit()

                return None, None

        except Exception as e:

            self.conn.rollback()

            messagebox.showerror("Ошибка запроса", str(e))

            return None, None

    def backup\_table(self, table\_name):

        # Создание резервной копии таблицы

        try:

            timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

            filename = f"{self.backup\_dir}{table\_name}\_{timestamp}.sql"

            with open(filename, 'w') as f:

                with self.conn.cursor() as cur:

                    cur.execute(f'SELECT \* FROM "{table\_name}"')

                    rows = cur.fetchall()

                    colnames = [f'"{desc[0]}"' for desc in cur.description]

                    f.write(f'CREATE TABLE "{table\_name}" ({", ".join(colnames)});\n')

                    for row in rows:

                        f.write(f'INSERT INTO "{table\_name}" VALUES {row};\n')

            return True

        except Exception as e:

            messagebox.showerror("Ошибка резервирования", str(e))

            return False

    def get\_fk\_values(self, table\_name, key\_column, display\_column):

        # Получение значений для внешнего ключа

        query = f'SELECT {key\_column}, {display\_column} FROM "{table\_name}"'

        data, \_ = self.execute\_query(query)

        return data or []

    def backup\_database(self):

        # Создание резервной копии всей базы данных

        try:

            timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

            filename = f"{self.backup\_dir}database\_backup\_{timestamp}.sql"

            command = [

                "pg\_dump",

                "-U", self.config["user"],

                "-h", self.config["host"],

                "-p", self.config["port"],

                "-d", self.config["dbname"],

                "-F", "c",

                "-f", filename

            ]

            env = os.environ.copy()

            env["PGPASSWORD"] = self.config["password"]

            subprocess.run(command, env=env, check=True)

            return filename

        except subprocess.CalledProcessError as e:

            messagebox.showerror("Ошибка резервирования", f"Ошибка выполнения команды: {str(e)}")

            return None

        except Exception as e:

            messagebox.showerror("Ошибка резервирования", str(e))

            return None

    def restore\_database(self, backup\_file):

        # Восстановление базы данных из резервной копии

        def restore\_task():

            try:

                command = [

                    "pg\_restore",

                    "-U", self.config["user"],

                    "-h", self.config["host"],

                    "-p", self.config["port"],

                    "-d", self.config["dbname"],

                    "-c",

                    backup\_file

                ]

                env = os.environ.copy()

                env["PGPASSWORD"] = self.config["password"]

                process = subprocess.Popen(

                    command, env=env, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True

                )

                stdout, stderr = process.communicate()

                if process.returncode == 0:

                    messagebox.showinfo("Успех", "База данных успешно восстановлена.")

                else:

                    messagebox.showerror("Ошибка восстановления", f"Ошибка: {stderr}")

            except Exception as e:

                messagebox.showerror("Ошибка восстановления", str(e))

        threading.Thread(target=restore\_task).start()

    def backup\_table\_to\_file(self, table\_name):

        # Создание резервной копии таблицы в файл

        try:

            timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

            filename = f"{self.backup\_dir}{table\_name}\_backup\_{timestamp}.sql"

            query = f"COPY \"{table\_name}\" TO STDOUT WITH CSV HEADER"

            with open(filename, 'w', encoding='utf-8') as f:

                with self.conn.cursor() as cur:

                    cur.copy\_expert(query, f)

            return filename

        except Exception as e:

            messagebox.showerror("Ошибка резервирования", str(e))

            return None

    def restore\_table\_from\_file(self, table\_name, backup\_file):

        # Восстановление таблицы из файла резервной копии

        try:

            self.execute\_query("SET session\_replication\_role = 'replica'")

            query = f"TRUNCATE TABLE \"{table\_name}\" CASCADE"

            self.execute\_query(query)

            with open(backup\_file, 'r', encoding='utf-8') as f:

                reader = csv.reader(f)

                columns = next(reader)

                included\_columns = [col for col in columns if not col.lower().endswith("id")]

                if not included\_columns:

                    messagebox.showwarning("Ошибка", f"Нет подходящих столбцов для вставки в таблице {table\_name}.")

                    return

                placeholders = ", ".join(["%s"] \* len(included\_columns))

                query = f'INSERT INTO "{table\_name}" ({", ".join(included\_columns)}) VALUES ({placeholders})'

                for row in reader:

                    filtered\_row = [row[columns.index(col)] for col in included\_columns]

                    self.execute\_query(query, filtered\_row)

            self.conn.commit()

            messagebox.showinfo("Успех", f"Таблица {table\_name} успешно восстановлена.")

        except Exception as e:

            self.conn.rollback()

            messagebox.showerror("Ошибка восстановления", str(e))

        finally:

            self.execute\_query("SET session\_replication\_role = 'origin'")

    def export\_table\_data(self, table\_name, export\_file):

        # Экспорт данных таблицы в CSV-файл

        try:

            query = f'SELECT \* FROM "{table\_name}"'

            data, columns = self.execute\_query(query)

            if not data or not columns:

                messagebox.showwarning("Ошибка", "Нет данных для экспорта.")

                return False

            with open(export\_file, 'w', newline='', encoding='utf-8') as f:

                writer = csv.writer(f)

                writer.writerow(columns)

                writer.writerows(data)

            return True

        except Exception as e:

            messagebox.showerror("Ошибка экспорта", str(e))

            return False

    def import\_table\_data(self, table\_name, import\_file):

        # Импорт данных из CSV-файла в таблицу

        try:

            with open(import\_file, 'r', encoding='utf-8') as f:

                reader = csv.reader(f)

                columns = next(reader)

                self.execute\_query(f'TRUNCATE TABLE "{table\_name}" RESTART IDENTITY')

                for row in reader:

                    placeholders = ", ".join(["%s"] \* len(row))

                    query = f'INSERT INTO "{table\_name}" ({columns}) VALUES ({placeholders})'

                    self.execute\_query(query, row)

            messagebox.showinfo("Успех", f"Данные успешно импортированы в таблицу {table\_name}.")

            return True

        except Exception as e:

            messagebox.showerror("Ошибка импорта", str(e))

            return False

class EditRowDialog(tk.Toplevel):

    # Диалог для редактирования строки

    def \_\_init\_\_(self, parent, db\_manager, table\_name, row\_data=None):

        super().\_\_init\_\_(parent)

        self.db = db\_manager

        self.table\_name = table\_name

        self.row\_data = row\_data

        self.title(f"Редактирование строки ({table\_name})" if row\_data else f"Добавление строки ({table\_name})")

        self.geometry("800x600")

        query = f"""

            SELECT column\_name

            FROM information\_schema.columns

            WHERE table\_name = '{table\_name}'

            AND column\_name != 'id'

        """

        columns, \_ = self.db.execute\_query(query)

        self.columns = [col[0] for col in columns]

        self.entries = {}

        for col in self.columns:

            tk.Label(self, text=col).pack()

            entry = tk.Entry(self, width=50)

            entry.pack()

            self.entries[col] = entry

        tk.Button(self, text="Сохранить", command=self.save).pack(pady=10)

    def save(self):

        # Сохранение изменений строки

        values = {}

        for col in self.columns:

            values[col] = self.entries[col].get()

        if self.row\_data:

            set\_clause = ", ".join([f'"{col}" = %s' for col in values.keys()])

            query = f'UPDATE "{self.table\_name}" SET {set\_clause} WHERE id = %s'

            params = list(values.values()) + [self.row\_data[0]]

        else:

            columns = ", ".join([f'"{col}"' for col in values.keys()])

            placeholders = ", ".join(["%s"] \* len(values))

            query = f'INSERT INTO "{self.table\_name}" ({columns}) VALUES ({placeholders})'

            params = list(values.values())

        try:

            self.db.execute\_query(query, params)

            self.destroy()

            messagebox.showinfo("Успех", "Изменения успешно сохранены.")

        except Exception as e:

            messagebox.showerror("Ошибка", f"Не удалось сохранить данные: {str(e)}")

class EditOrderDialog(tk.Toplevel):

    # Диалог для редактирования заказа

    def \_\_init\_\_(self, parent, db\_manager, table\_name, row\_data):

        super().\_\_init\_\_(parent)

        self.db = db\_manager

        self.table\_name = table\_name

        self.row\_data = row\_data

        self.title(f"Изменение строки ({table\_name})")

        self.geometry("800x600")

        self.columns = self.get\_table\_columns()

        self.row\_data\_dict = {

            "orderid": row\_data[0],

            "status": row\_data[1],

            "totalcost": row\_data[2],

            "numberofguests": row\_data[3],

            "orderdatetime": row\_data[4],

            "visitorid": row\_data[5],

            "employeeid": row\_data[6]

        }

        self.entries = {}

        self.create\_fields()

        self.create\_dish\_editor()

        tk.Button(self, text="Сохранить", command=self.save).pack(pady=10)

    def get\_table\_columns(self):

        # Получение списка столбцов таблицы

        query = f"SELECT column\_name FROM information\_schema.columns WHERE table\_name = '{self.table\_name}'"

        columns, \_ = self.db.execute\_query(query)

        return [col[0] for col in columns]

    def create\_fields(self):

        # Создание полей для редактирования данных заказа

        for col in self.columns:

            if col == "orderid":

                continue

            tk.Label(self, text=col).pack()

            current\_value = self.row\_data\_dict.get(col, "")

            if col == "numberofguests":

                self.entries[col] = self.create\_number\_field(current\_value)

            elif col == "status":

                self.entries[col] = self.create\_status\_field(current\_value)

            elif col == "visitorid":

                self.entries[col] = self.create\_fk\_field("visitor", "visitorid", "name", current\_value)

            elif col == "employeeid":

                self.entries[col] = self.create\_employee\_field(current\_value)

            else:

                self.entries[col] = self.create\_text\_field(current\_value)

    def create\_number\_field(self, value):

        # Создание поля для ввода числового значения

        entry = tk.Entry(self, width=50, validate="key")

        entry.insert(0, value)

        entry.pack()

        entry.configure(validatecommand=(self.register(self.validate\_number), "%P"))

        return entry

    def create\_status\_field(self, value):

        # Создание выпадающего списка для статуса

        combobox = ttk.Combobox(self, values=["Закрыт", "Ожидает", "Готовится", "Готов"], width=50)

        combobox.set(value)

        combobox.pack()

        return combobox

    def create\_fk\_field(self, table\_name, key\_column, display\_column, value):

        # Создание выпадающего списка для внешнего ключа

        fk\_values = self.db.get\_fk\_values(table\_name, key\_column, display\_column)

        fk\_values.insert(0, (None, "Неизвестный гость"))

        combobox = ttk.Combobox(self, values=[f"{key} - {value}" for key, value in fk\_values], width=50)

        combobox.set(f"{value} - {dict(fk\_values).get(value, '')}")

        combobox.pack()

        return (combobox, fk\_values)

    def create\_employee\_field(self, value):

        # Создание выпадающего списка для выбора сотрудника

        query = """

            SELECT e.employeeid, e.fullname

            FROM employee e

            JOIN employeeposition ep ON e.employeeid = ep.employeeid

            JOIN "position" p ON ep.positionid = p.positionid

            WHERE p.name = 'Официант'

        """

        fk\_values, \_ = self.db.execute\_query(query)

        combobox = ttk.Combobox(self, values=[f"{key} - {value}" for key, value in fk\_values], width=50)

        combobox.set(f"{value} - {dict(fk\_values).get(value, '')}")

        combobox.pack()

        return (combobox, fk\_values)

    def create\_text\_field(self, value):

        # Создание текстового поля

        entry = tk.Entry(self, width=50)

        entry.insert(0, value)

        entry.pack()

        return entry

    def create\_dish\_editor(self):

        # Создание интерфейса для редактирования блюд в заказе

        tk.Label(self, text="Блюда и количество").pack()

        self.dish\_frame = tk.Frame(self)

        self.dish\_frame.pack(fill=tk.BOTH, expand=True)

        self.add\_dish\_button = tk.Button(self.dish\_frame, text="Добавить блюдо", command=self.add\_dish\_row)

        self.add\_dish\_button.pack()

        self.dish\_rows = []

        self.load\_order\_dishes()

    def load\_order\_dishes(self):

        # Загрузка блюд, связанных с заказом

        query = """

            SELECT od.dishid, d.name, od.count

            FROM orderdish od

            JOIN dish d ON od.dishid = d.dishid

            WHERE od.orderid = %s

        """

        dishes, \_ = self.db.execute\_query(query, (self.row\_data[0],))

        for dish\_id, dish\_name, count in dishes:

            self.add\_dish\_row(dish\_id, dish\_name, count)

    def add\_dish\_row(self, dish\_id=None, dish\_name=None, count=None):

        # Добавление строки для редактирования блюда

        row\_frame = tk.Frame(self.dish\_frame)

        row\_frame.pack(fill=tk.X, pady=5)

        fk\_values = self.db.get\_fk\_values("dish", "dishid", "name")

        dish\_combobox = ttk.Combobox(row\_frame, values=[f"{key} - {value}" for key, value in fk\_values], width=30)

        if dish\_id and dish\_name:

            dish\_combobox.set(f"{dish\_id} - {dish\_name}")

        dish\_combobox.pack(side=tk.LEFT, padx=5)

        quantity\_entry = tk.Entry(row\_frame, width=10)

        if count:

            quantity\_entry.insert(0, count)

        quantity\_entry.pack(side=tk.LEFT, padx=5)

        self.dish\_rows.append((dish\_combobox, quantity\_entry))

    def validate\_number(self, value):

        # Валидация числового значения

        return value.isdigit() or value == ""

    def save(self):

        # Сохранение изменений в заказе

        updated\_columns = []

        params = []

        for col, widget in self.entries.items():

            if col in ["visitorid", "employeeid"]:

                combobox, fk\_values = widget

                selected\_value = combobox.get().split(" - ")[0]

                new\_value = None if selected\_value == "None" else selected\_value

            elif col == "status":

                new\_value = widget.get()

            else:

                new\_value = widget.get()

            current\_value = self.row\_data\_dict.get(col, "")

            if str(current\_value) != str(new\_value):

                updated\_columns.append(f"{col} = %s")

                params.append(new\_value)

        if updated\_columns:

            set\_clause = ", ".join(updated\_columns)

            query = f'UPDATE "{self.table\_name}" SET {set\_clause} WHERE orderid = %s'

            params.append(self.row\_data[0])

            self.db.execute\_query(query, params)

        query = 'DELETE FROM "orderdish" WHERE "orderid" = %s'

        self.db.execute\_query(query, (self.row\_data[0],))

        for dish\_combobox, quantity\_entry in self.dish\_rows:

            dish\_id = dish\_combobox.get().split(" - ")[0]

            quantity = quantity\_entry.get()

            if dish\_id and quantity:

                query = 'INSERT INTO "orderdish" ("orderid", "dishid", "count") VALUES (%s, %s, %s)'

                self.db.execute\_query(query, (self.row\_data[0], dish\_id, quantity))

        self.destroy()

        messagebox.showinfo("Успех", "Изменения успешно сохранены.")

class TableWindow(tk.Toplevel):

    # Окно для работы с таблицей

    def \_\_init\_\_(self, parent, db\_manager, table\_name):

        super().\_\_init\_\_(parent)

        self.db = db\_manager

        self.table\_name = table\_name

        self.title(f"Таблица: {table\_name}")

        self.geometry("800x800")

        self.state("zoomed")

        frame = tk.Frame(self)

        frame.pack(fill=tk.BOTH, expand=True)

        self.tree = ttk.Treeview(frame)

        self.tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        y\_scroll = ttk.Scrollbar(frame, orient=tk.VERTICAL, command=self.tree.yview)

        y\_scroll.pack(side=tk.RIGHT, fill=tk.Y)

        x\_scroll = ttk.Scrollbar(self, orient=tk.HORIZONTAL, command=self.tree.xview)

        x\_scroll.pack(side=tk.BOTTOM, fill=tk.X)

        self.tree.configure(yscrollcommand=y\_scroll.set, xscrollcommand=x\_scroll.set)

        toolbar = tk.Frame(self)

        toolbar.pack(fill=tk.X)

        tk.Button(toolbar, text="Добавить", command=self.add\_row).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Редактировать", command=self.edit\_row).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Удалить", command=self.delete\_row).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Экспорт в CSV", command=self.export\_csv).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Экспорт в Excel", command=self.export\_excel).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Бэкап таблицы", command=self.backup\_table).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Восстановить таблицу", command=self.restore\_table).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Добавить столбец", command=self.add\_column).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Обновить", command=self.reload\_table).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Удалить таблицу", command=self.delete\_table).pack(side=tk.LEFT, padx=5)

        self.load\_data()

        self.tree.bind("<Double-1>", self.on\_double\_click)

    def load\_data(self):

        # Загрузка данных таблицы

        query = f'SELECT \* FROM "{self.table\_name}"'

        data, columns = self.db.execute\_query(query)

        if data:

            self.tree['columns'] = columns

            self.tree['show'] = 'headings'

            for col in columns:

                self.tree.heading(col, text=col)

                self.tree.column(col, width=150, anchor='center')

            for i, row in enumerate(data):

                tag = 'even' if i % 2 == 0 else 'odd'

                self.tree.insert('', 'end', values=row, tags=(tag,))

            self.tree.tag\_configure('even', background='#f2f2f2')

            self.tree.tag\_configure('odd', background='#ffffff')

    def on\_double\_click(self, event):

        # Обработчик двойного клика для редактирования ячейки или заголовка столбца

        region = self.tree.identify("region", event.x, event.y)

        if region == "heading":

            self.on\_column\_header\_double\_click(event)

        elif region == "cell":

            self.on\_cell\_double\_click(event)

    def on\_column\_header\_double\_click(self, event):

        # Обработчик двойного клика для заголовка столбца

        column = self.tree.identify\_column(event.x)

        column\_index = int(column.replace("#", "")) - 1

        column\_name = self.tree["columns"][column\_index]

        column\_window = tk.Toplevel(self)

        column\_window.title(f"Управление столбцом '{column\_name}'")

        column\_window.geometry("300x250")

        tk.Label(column\_window, text=f"Столбец: {column\_name}").pack(pady=10)

        tk.Label(column\_window, text="Новое имя столбца:").pack()

        rename\_entry = tk.Entry(column\_window, width=30)

        rename\_entry.pack(pady=5)

        def rename\_column():

            new\_name = rename\_entry.get()

            if not new\_name:

                messagebox.showwarning("Ошибка", "Введите новое имя столбца.")

                return

            query = f'ALTER TABLE "{self.table\_name}" RENAME COLUMN "{column\_name}" TO "{new\_name}"'

            self.db.execute\_query(query)

            messagebox.showinfo("Успех", f"Столбец '{column\_name}' переименован в '{new\_name}'.")

            column\_window.destroy()

            self.reload\_table()

        tk.Button(column\_window, text="Переименовать", command=rename\_column).pack(pady=5)

        def delete\_column():

            if not messagebox.askyesno("Подтверждение", f"Вы уверены, что хотите удалить столбец '{column\_name}'?"):

                return

            query = f'ALTER TABLE "{self.table\_name}" DROP COLUMN "{column\_name}"'

            self.db.execute\_query(query)

            messagebox.showinfo("Успех", f"Столбец '{column\_name}' успешно удален.")

            column\_window.destroy()

            self.reload\_table()

        tk.Button(column\_window, text="Удалить столбец", command=delete\_column).pack(pady=5)

    def reload\_table(self):

        # Перезагрузка данных таблицы

        for item in self.tree.get\_children():

            self.tree.delete(item)

        self.load\_data()

    def on\_cell\_double\_click(self, event):

        # Обработчик двойного клика для редактирования ячейки

        selected\_item = self.tree.selection()

        if not selected\_item:

            return

        item = self.tree.item(selected\_item[0])

        column = self.tree.identify\_column(event.x)

        column\_index = int(column.replace("#", "")) - 1

        column\_name = self.tree["columns"][column\_index]

        current\_value = item["values"][column\_index]

        edit\_window = tk.Toplevel(self)

        edit\_window.title("Редактирование ячейки")

        edit\_window.geometry("300x150")

        tk.Label(edit\_window, text=f"Изменить значение в столбце '{column\_name}':").pack(pady=10)

        entry = tk.Entry(edit\_window, width=30)

        entry.insert(0, current\_value)

        entry.pack(pady=5)

        def save\_value():

            new\_value = entry.get()

            if new\_value == current\_value:

                edit\_window.destroy()

                return

            primary\_key\_column = self.tree["columns"][0]

            primary\_key\_value = item["values"][0]

            query = f'UPDATE "{self.table\_name}" SET "{column\_name}" = %s WHERE "{primary\_key\_column}" = %s'

            self.db.execute\_query(query, (new\_value, primary\_key\_value))

            item["values"][column\_index] = new\_value

            self.tree.item(selected\_item[0], values=item["values"])

            messagebox.showinfo("Успех", "Значение успешно обновлено.")

            edit\_window.destroy()

        tk.Button(edit\_window, text="Сохранить", command=save\_value).pack(pady=10)

    def add\_row(self):

        # Добавление новой строки

        EditRowDialog(self, self.db, self.table\_name)

    def edit\_row(self):

        # Редактирование выбранной строки

        selected\_item = self.tree.selection()

        if not selected\_item:

            messagebox.showwarning("Ошибка", "Выберите строку для редактирования.")

            return

        row\_data = self.tree.item(selected\_item[0], 'values')

        if self.table\_name == "Order":

            EditOrderDialog(self, self.db, self.table\_name, row\_data)

        else:

            EditRowDialog(self, self.db, self.table\_name, row\_data)

    def delete\_row(self):

        # Удаление выбранной строки

        selected\_item = self.tree.selection()

        if not selected\_item:

            messagebox.showwarning("Ошибка", "Выберите строку для удаления.")

            return

        row\_data = self.tree.item(selected\_item[0], 'values')

        primary\_key = self.tree['columns'][0]

        try:

            related\_tables = []

            if self.table\_name == "Order":

                query = 'SELECT COUNT(\*) FROM "orderdish" WHERE "orderid" = %s'

                count, \_ = self.db.execute\_query(query, (row\_data[0],))

                if count and count[0][0] > 0:

                    related\_tables.append(("orderdish", "orderid"))

            query = f"""

                SELECT ccu.table\_name, ccu.column\_name

                FROM information\_schema.constraint\_column\_usage ccu

                JOIN information\_schema.referential\_constraints rc

                ON ccu.constraint\_name = rc.constraint\_name

                JOIN information\_schema.key\_column\_usage kcu

                ON rc.unique\_constraint\_name = kcu.constraint\_name

                WHERE kcu.table\_name = %s AND kcu.column\_name = %s

            """

            additional\_related\_tables, \_ = self.db.execute\_query(query, (self.table\_name, primary\_key))

            related\_tables.extend(additional\_related\_tables)

            if related\_tables:

                choice = messagebox.askyesno(

                    "Связанные записи",

                    "Существуют связанные записи. Хотите удалить их вместе с основной записью?"

                )

                if not choice:

                    return

                for related\_table, related\_column in related\_tables:

                    delete\_related\_query = f'DELETE FROM "{related\_table}" WHERE "{related\_column}" = %s'

                    self.db.execute\_query(delete\_related\_query, (row\_data[0],))

        except Exception as e:

            messagebox.showerror("Ошибка", f"Не удалось проверить связанные записи: {str(e)}")

            return

        if not messagebox.askyesno("Подтверждение", "Вы уверены, что хотите удалить выбранную строку?"):

            return

        query = f'DELETE FROM "{self.table\_name}" WHERE "{primary\_key}" = %s'

        self.db.execute\_query(query, (row\_data[0],))

        self.tree.delete(selected\_item[0])

        messagebox.showinfo("Успех", "Строка успешно удалена.")

    def export\_csv(self):

        # Экспорт данных таблицы в CSV

        filename = filedialog.asksaveasfilename(defaultextension=".csv",

                                                filetypes=[("CSV files", "\*.csv")])

        if not filename:

            return

        query = f'SELECT \* FROM "{self.table\_name}"'

        data, columns = self.db.execute\_query(query)

        if not data or not columns:

            messagebox.showwarning("Ошибка", "Нет данных для экспорта.")

            return

        try:

            # Используем кодировку utf-8-sig для корректного отображения русских символов

            with open(filename, 'w', newline='', encoding='utf-8-sig') as f:

                writer = csv.writer(f)

                writer.writerow(columns)

                writer.writerows(data)

            messagebox.showinfo("Успех", f"Данные успешно экспортированы в файл: {filename}")

        except Exception as e:

            messagebox.showerror("Ошибка", f"Не удалось экспортировать данные: {str(e)}")

    def export\_excel(self):

        # Экспорт данных таблицы в Excel

        filename = filedialog.asksaveasfilename(defaultextension=".xlsx",

                                                filetypes=[("Excel files", "\*.xlsx")])

        if not filename:

            return

        query = f'SELECT \* FROM "{self.table\_name}"'

        data, columns = self.db.execute\_query(query)

        if not data or not columns:

            messagebox.showwarning("Ошибка", "Нет данных для экспорта.")

            return

        try:

            from openpyxl import Workbook

            wb = Workbook()

            ws = wb.active

            ws.title = self.table\_name

            ws.append(columns)

            for row in data:

                ws.append(row)

            wb.save(filename)

            messagebox.showinfo("Успех", f"Данные успешно экспортированы в файл: {filename}")

        except Exception as e:

            messagebox.showerror("Ошибка", f"Не удалось экспортировать данные: {str(e)}")

    def backup\_table(self):

        # Создание резервной копии таблицы

        export\_file = filedialog.asksaveasfilename(defaultextension=".csv",

                                                   filetypes=[("CSV files", "\*.csv")])

        if not export\_file:

            return

        if self.db.export\_table\_data(self.table\_name, export\_file):

            messagebox.showinfo("Успех", f"Данные таблицы успешно экспортированы: {export\_file}")

    def restore\_table(self):

        # Восстановление таблицы из резервной копии

        import\_file = filedialog.askopenfilename(filetypes=[("CSV files", "\*.csv")])

        if not import\_file:

            return

        if self.db.import\_table\_data(self.table\_name, import\_file):

            self.reload\_table()

    def add\_column(self):

        # Добавление нового столбца

        column\_window = tk.Toplevel(self)

        column\_window.title("Добавить столбец")

        column\_window.geometry("300x200")

        tk.Label(column\_window, text="Имя нового столбца:").pack(pady=5)

        add\_column\_entry = tk.Entry(column\_window, width=30)

        add\_column\_entry.pack(pady=5)

        tk.Label(column\_window, text="Тип нового столбца:").pack(pady=5)

        add\_column\_type = ttk.Combobox(column\_window, values=["INTEGER", "TEXT", "DATE", "BOOLEAN"], width=27)

        add\_column\_type.set("TEXT")

        add\_column\_type.pack(pady=5)

        def save\_column():

            new\_column\_name = add\_column\_entry.get()

            column\_type = add\_column\_type.get()

            if not new\_column\_name or not column\_type:

                messagebox.showwarning("Ошибка", "Введите имя и тип нового столбца.")

                return

            query = f'ALTER TABLE "{self.table\_name}" ADD COLUMN "{new\_column\_name}" {column\_type}'

            self.db.execute\_query(query)

            messagebox.showinfo("Успех", f"Столбец '{new\_column\_name}' добавлен.")

            column\_window.destroy()

            self.reload\_table()

        tk.Button(column\_window, text="Сохранить", command=save\_column).pack(pady=10)

    def delete\_table(self):

        # Удаление текущей таблицы

        if not messagebox.askyesno("Подтверждение", f"Вы уверены, что хотите удалить таблицу '{self.table\_name}'?"):

            return

        try:

            query = f'DROP TABLE "{self.table\_name}" CASCADE'

            self.db.execute\_query(query)

            messagebox.showinfo("Успех", f"Таблица '{self.table\_name}' успешно удалена.")

            self.destroy()

        except Exception as e:

            messagebox.showerror("Ошибка", f"Не удалось удалить таблицу: {str(e)}")

class AddTableDialog(tk.Toplevel):

    # Диалог для добавления новой таблицы

    def \_\_init\_\_(self, parent, db\_manager):

        super().\_\_init\_\_(parent)

        self.db = db\_manager

        self.title("Добавить новую таблицу")

        self.geometry("600x400")

        self.columns = []

        tk.Label(self, text="Имя таблицы:").pack(pady=5)

        self.table\_name\_entry = tk.Entry(self, width=50)

        self.table\_name\_entry.pack(pady=5)

        self.columns\_frame = tk.Frame(self)

        self.columns\_frame.pack(fill=tk.BOTH, expand=True, pady=10)

        tk.Button(self, text="Добавить столбец", command=self.add\_column).pack(pady=5)

        tk.Button(self, text="Сохранить", command=self.save\_table).pack(pady=10)

    def add\_column(self):

        # Добавление строки для описания нового столбца

        row\_frame = tk.Frame(self.columns\_frame)

        row\_frame.pack(fill=tk.X, pady=5)

        tk.Label(row\_frame, text="Имя:").pack(side=tk.LEFT, padx=5)

        column\_name\_entry = tk.Entry(row\_frame, width=20)

        column\_name\_entry.pack(side=tk.LEFT, padx=5)

        tk.Label(row\_frame, text="Тип:").pack(side=tk.LEFT, padx=5)

        column\_type\_combobox = ttk.Combobox(row\_frame, values=["INTEGER", "TEXT", "DATE", "BOOLEAN"], width=15)

        column\_type\_combobox.set("TEXT")

        column\_type\_combobox.pack(side=tk.LEFT, padx=5)

        self.columns.append((column\_name\_entry, column\_type\_combobox, None, None, None))

    def save\_table(self):

        # Сохранение новой таблицы в базе данных

        table\_name = self.table\_name\_entry.get()

        if not table\_name:

            messagebox.showwarning("Ошибка", "Введите имя таблицы.")

            return

        columns\_definitions = ['"id" SERIAL PRIMARY KEY']

        for column\_name\_entry, column\_type\_combobox, \_, \_, \_ in self.columns:

            column\_name = column\_name\_entry.get()

            column\_type = column\_type\_combobox.get()

            if not column\_name or not column\_type:

                messagebox.showwarning("Ошибка", "Заполните все поля для столбцов.")

                return

            columns\_definitions.append(f'"{column\_name}" {column\_type}')

        query = f'CREATE TABLE "{table\_name}" ({", ".join(columns\_definitions)})'

        try:

            self.db.execute\_query(query)

            messagebox.showinfo("Успех", f"Таблица '{table\_name}' успешно создана.")

            self.destroy()

        except Exception as e:

            messagebox.showerror("Ошибка", f"Не удалось создать таблицу: {str(e)}")

class QueryManagerApp(tk.Tk):

    # Главное окно приложения

    def \_\_init\_\_(self, db\_manager):

        super().\_\_init\_\_()

        self.db = db\_manager

        self.title("Управление базой данных ресторана")

        self.geometry("800x600")

        self.state("zoomed")

        self.saved\_queries = []

        self.load\_queries()

        frame = tk.Frame(self)

        frame.pack(fill=tk.BOTH, expand=True)

        toolbar\_frame = tk.Frame(self)

        toolbar\_frame.pack(fill=tk.X)

        management\_toolbar = tk.Frame(toolbar\_frame)

        management\_toolbar.pack(fill=tk.X, padx=5, pady=5)

        self.edit\_btn = tk.Button(management\_toolbar, text="Изменить запрос", command=self.edit\_query)

        self.edit\_btn.pack(side=tk.LEFT, padx=5)

        self.add\_btn = tk.Button(management\_toolbar, text="Добавить запрос", command=self.add\_query)

        self.add\_btn.pack(side=tk.LEFT, padx=5)

        self.delete\_btn = tk.Button(management\_toolbar, text="Удалить запрос", command=self.delete\_query)

        self.delete\_btn.pack(side=tk.LEFT, padx=5)

        tk.Button(management\_toolbar, text="Бэкап базы", command=self.backup\_database).pack(side=tk.LEFT, padx=5)

        tk.Button(management\_toolbar, text="Восстановить базу", command=self.restore\_database).pack(side=tk.LEFT, padx=5)

        tk.Button(management\_toolbar, text="Добавить таблицу", command=self.add\_table).pack(side=tk.LEFT, padx=5)

        tk.Button(management\_toolbar, text="Обновить", command=self.restart\_app).pack(side=tk.LEFT, padx=5)

        table\_toolbar = tk.Frame(toolbar\_frame)

        table\_toolbar.pack(fill=tk.X, padx=5, pady=5)

        tk.Button(table\_toolbar, text="Заказы", command=lambda: self.open\_table\_window("Order")).pack(side=tk.LEFT, padx=5)

        tk.Button(table\_toolbar, text="Блюда", command=lambda: self.open\_table\_window("dish")).pack(side=tk.LEFT, padx=5)

        tk.Button(table\_toolbar, text="Работники", command=lambda: self.open\_table\_window("employee")).pack(side=tk.LEFT, padx=5)

        tk.Button(table\_toolbar, text="Позиции", command=lambda: self.open\_table\_window("position")).pack(side=tk.LEFT, padx=5)

        tk.Button(table\_toolbar, text="Ингредиенты", command=lambda: self.open\_table\_window("ingredient")).pack(side=tk.LEFT, padx=5)

        tk.Button(table\_toolbar, text="Посетители", command=lambda: self.open\_table\_window("visitor")).pack(side=tk.LEFT, padx=5)

        tk.Button(table\_toolbar, text="Поставщики", command=lambda: self.open\_table\_window("supplier")).pack(side=tk.LEFT, padx=5)

        self.add\_remaining\_table\_buttons(table\_toolbar)

        self.tree = ttk.Treeview(frame, columns=('query',), show='tree headings')

        self.tree.heading('#0', text='Название запроса')

        self.tree.column('#0', width=200)

        self.tree.heading('query', text='SQL-запрос')

        self.tree.column('query', width=500)

        self.tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        y\_scroll = ttk.Scrollbar(frame, orient=tk.VERTICAL, command=self.tree.yview)

        y\_scroll.pack(side=tk.RIGHT, fill=tk.Y)

        x\_scroll = ttk.Scrollbar(self, orient=tk.HORIZONTAL, command=self.tree.xview)

        x\_scroll.pack(side=tk.BOTTOM, fill=tk.X)

        self.tree.configure(yscrollcommand=y\_scroll.set, xscrollcommand=x\_scroll.set)

        self.load\_queries\_ui()

        self.tree.bind('<Double-1>', self.run\_query)

    def restart\_app(self):

        # Перезапуск приложения

        self.destroy()

        new\_app = QueryManagerApp(self.db)

        new\_app.mainloop()

    def add\_remaining\_table\_buttons(self, toolbar):

        # Добавление кнопок для оставшихся таблиц

        excluded\_tables = {"Order", "dish", "employee", "position", "ingredient", "visitor", "supplier"}

        query = """

            SELECT table\_name

            FROM information\_schema.tables

            WHERE table\_schema = 'public'

        """

        tables, \_ = self.db.execute\_query(query)

        for table in tables:

            table\_name = table[0]

            if table\_name not in excluded\_tables:

                tk.Button(toolbar, text=table\_name.capitalize(), command=lambda t=table\_name: self.open\_table\_window(t)).pack(side=tk.LEFT, padx=5)

    def open\_table\_window(self, table\_name):

        # Открытие окна для работы с таблицей

        TableWindow(self, self.db, table\_name)

    def load\_queries(self):

        # Загрузка сохраненных запросов

        try:

            with open('queries.json', 'r', encoding='utf-8') as f:

                self.saved\_queries = json.load(f)

        except FileNotFoundError:

            self.saved\_queries = []

    def save\_queries(self):

        # Сохранение запросов в файл

        with open('queries.json', 'w', encoding='utf-8') as f:

            json.dump(self.saved\_queries, f)

    def load\_queries\_ui(self):

        # Загрузка запросов в интерфейс

        for item in self.tree.get\_children():

            self.tree.delete(item)

        for i, query in enumerate(self.saved\_queries):

            tag = 'even' if i % 2 == 0 else 'odd'

            self.tree.insert('', 'end', text=query['name'], values=(query['sql'],), tags=(tag,))

        self.tree.tag\_configure('even', background='#f2f2f2')

        self.tree.tag\_configure('odd', background='#ffffff')

    def add\_query(self):

        # Добавление нового запроса

        EditQueryDialog(self, self.db)

    def run\_query(self, event=None):

        # Выполнение выбранного запроса

        selected\_item = self.tree.selection()

        if not selected\_item:

            messagebox.showwarning("Ошибка", "Выберите запрос для выполнения.")

            return

        query = self.tree.item(selected\_item[0], 'values')[0]

        ResultWindow(self, self.db, query)

    def edit\_query(self):

        # Редактирование выбранного запроса

        selected\_item = self.tree.selection()

        if not selected\_item:

            messagebox.showwarning("Ошибка", "Выберите запрос для редактирования.")

            return

        query\_name = self.tree.item(selected\_item[0], 'text')

        query\_sql = self.tree.item(selected\_item[0], 'values')[0]

        EditQueryDialog(self, self.db, query={'name': query\_name, 'sql': query\_sql})

    def delete\_query(self):

        # Удаление выбранного запроса

        selected\_item = self.tree.selection()

        if not selected\_item:

            messagebox.showwarning("Ошибка", "Выберите запрос для удаления.")

            return

        query\_name = self.tree.item(selected\_item[0], 'text')

        self.saved\_queries = [query for query in self.saved\_queries if query['name'] != query\_name]

        self.save\_queries()

        self.load\_queries\_ui()

        messagebox.showinfo("Успех", f"Запрос '{query\_name}' успешно удален.")

    def backup\_database(self):

        # Создание резервной копии базы данных

        backup\_file = self.db.backup\_database()

        if backup\_file:

            messagebox.showinfo("Успех", f"Бэкап успешно создан: {backup\_file}")

    def restore\_database(self):

        # Восстановление базы данных из резервной копии

        backup\_file = filedialog.askopenfilename(filetypes=[("SQL files", "\*.sql")])

        if not backup\_file:

            return

        self.db.restore\_database(backup\_file)

        messagebox.showinfo("Успех", "База данных успешно восстановлена.")

    def restore\_backup(self):

        # Восстановление базы данных из резервной копии

        filename = filedialog.askopenfilename(filetypes=[("SQL files", "\*.sql")])

        if filename:

            try:

                pg\_restore\_path = r"C:\Program Files\PostgreSQL\16\bin\pg\_restore.exe"

                if not os.path.exists(pg\_restore\_path):

                    raise FileNotFoundError(f"pg\_restore.exe не найден по пути: {pg\_restore\_path}")

                drop\_db\_query = f"DROP DATABASE IF EXISTS {self.db.config['dbname']}"

                create\_db\_query = f"CREATE DATABASE {self.db.config['dbname']}"

                with psycopg2.connect(

                    dbname="postgres",

                    user=self.db.config["user"],

                    password=self.db.config["password"],

                    host=self.db.config["host"],

                    port=self.db.config["port"]

                ) as conn:

                    conn.autocommit = True

                    with conn.cursor() as cur:

                        cur.execute(drop\_db\_query)

                        cur.execute(create\_db\_query)

                cmd = [

                    pg\_restore\_path,

                    '-U', self.db.config["user"],

                    '-h', self.db.config["host"],

                    '-p', self.db.config["port"],

                    '-d', self.db.config["dbname"],

                    filename

                ]

                env = os.environ.copy()

                if self.db.config["password"]:

                    env["PGPASSWORD"] = self.db.config["password"]

                process = subprocess.run(cmd, env=env, stderr=subprocess.PIPE, stdout=subprocess.PIPE, text=True)

                if process.returncode != 0:

                    raise Exception(f"Ошибка pg\_restore: {process.stderr.strip()}")

                messagebox.showinfo("Успех", "База данных успешно восстановлена")

            except FileNotFoundError as e:

                messagebox.showerror("Ошибка", str(e))

            except Exception as e:

                messagebox.showerror("Ошибка восстановления", f"Произошла ошибка: {str(e)}")

    def add\_table(self):

        # Открытие диалога для добавления новой таблицы

        AddTableDialog(self, self.db)

class EditQueryDialog(tk.Toplevel):

    # Окно редактирования запроса

    def \_\_init\_\_(self, parent, db\_manager, query=None):

        super().\_\_init\_\_(parent)

        self.parent = parent

        self.db = db\_manager

        self.query = query or {'name': '', 'sql': ''}

        self.title("Редактор запроса")

        self.geometry("600x400")

        tk.Label(self, text="Название запроса:").pack()

        self.name\_entry = tk.Entry(self, width=50)

        self.name\_entry.pack()

        self.name\_entry.insert(0, self.query['name'])

        tk.Label(self, text="SQL-запрос:").pack()

        self.sql\_editor = scrolledtext.ScrolledText(self, width=70, height=15)

        self.sql\_editor.pack()

        self.sql\_editor.insert('1.0', self.query['sql'])

        self.sql\_editor.bind("<Control-v>", self.paste\_text)

        self.sql\_editor.bind("<Control-c>", self.copy\_text)

        self.sql\_editor.bind("<Control-x>", self.cut\_text)

        btn\_frame = tk.Frame(self)

        btn\_frame.pack(pady=10)

        tk.Button(btn\_frame, text="Сохранить", command=self.save).pack(side=tk.LEFT, padx=5)

        tk.Button(btn\_frame, text="Отмена", command=self.destroy).pack(side=tk.LEFT)

    def paste\_text(self, event=None):

        # Вставка текста

        try:

            self.sql\_editor.insert(tk.INSERT, self.clipboard\_get())

        except tk.TclError:

            pass

        return "break"

    def copy\_text(self, event=None):

        # Копирование текста

        try:

            selected\_text = self.sql\_editor.get(tk.SEL\_FIRST, tk.SEL\_LAST)

            self.clipboard\_clear()

            self.clipboard\_append(selected\_text)

        except tk.TclError:

            pass

        return "break"

    def cut\_text(self, event=None):

        # Вырезание текста

        try:

            selected\_text = self.sql\_editor.get(tk.SEL\_FIRST, tk.SEL\_LAST)

            self.clipboard\_clear()

            self.clipboard\_append(selected\_text)

            self.sql\_editor.delete(tk.SEL\_FIRST, tk.SEL\_LAST)

        except tk.TclError:

            pass

        return "break"

    def save(self):

        # Сохранение изменений запроса

        name = self.name\_entry.get()

        sql = self.sql\_editor.get('1.0', tk.END).strip()

        if not name or not sql:

            messagebox.showwarning("Ошибка", "Заполните все поля")

            return

        if self.query in self.parent.saved\_queries:

            self.parent.saved\_queries.remove(self.query)

        self.parent.saved\_queries.append({'name': name, 'sql': sql})

        self.parent.save\_queries()

        self.parent.load\_queries\_ui()

        self.destroy()

class ResultWindow(tk.Toplevel):

    # Окно для отображения результатов запроса

    def \_\_init\_\_(self, parent, db\_manager, query):

        super().\_\_init\_\_(parent)

        self.title("Результаты запроса")

        self.geometry("800x600")

        self.db = db\_manager

        self.query = query

        self.tree = ttk.Treeview(self)

        self.tree.pack(fill=tk.BOTH, expand=True)

        toolbar = tk.Frame(self)

        toolbar.pack(fill=tk.X)

        tk.Button(toolbar, text="Экспорт в CSV", command=self.export\_csv).pack(side=tk.LEFT, padx=5)

        tk.Button(toolbar, text="Экспорт в Excel", command=self.export\_excel).pack(side=tk.LEFT)

        self.load\_data()

    def load\_data(self):

        # Загрузка данных запроса

        data, columns = self.db.execute\_query(self.query)

        if data:

            self.tree['columns'] = columns

            self.tree['show'] = 'headings'

            for col in columns:

                self.tree.heading(col, text=col)

                self.tree.column(col, width=100, anchor='center')

            for i, row in enumerate(data):

                tag = 'even' if i % 2 == 0 else 'odd'

                self.tree.insert('', 'end', values=row, tags=(tag,))

            self.tree.tag\_configure('even', background='#f2f2f2')

            self.tree.tag\_configure('odd', background='#ffffff')

    def export\_csv(self):

        # Экспорт результатов в CSV

        filename = filedialog.asksaveasfilename(defaultextension=".csv",

                                                filetypes=[("CSV files", "\*.csv")])

        if not filename:

            return

        data, columns = self.db.execute\_query(self.query)

        if not data or not columns:

            messagebox.showwarning("Ошибка", "Нет данных для экспорта.")

            return

        try:

            with open(filename, 'w', newline='', encoding='utf-8') as f:

                writer = csv.writer(f)

                writer.writerow(columns)

                writer.writerows(data)

            messagebox.showinfo("Успех", f"Данные успешно экспортированы в файл: {filename}")

        except Exception as e:

            messagebox.showerror("Ошибка", f"Не удалось экспортировать данные: {str(e)}")

    def export\_excel(self):

        # Экспорт результатов в Excel

        filename = filedialog.asksaveasfilename(defaultextension=".xlsx",

                                                filetypes=[("Excel files", "\*.xlsx")])

        if not filename:

            return

        data, columns = self.db.execute\_query(self.query)

        if not data or not columns:

            messagebox.showwarning("Ошибка", "Нет данных для экспорта.")

            return

        try:

            from openpyxl import Workbook

            wb = Workbook()

            ws = wb.active

            ws.title = "Результаты"

            ws.append(columns)

            for row in data:

                ws.append(row)

            wb.save(filename)

            messagebox.showinfo("Успех", f"Данные успешно экспортированы в файл: {filename}")

        except Exception as e:

            messagebox.showerror("Ошибка", f"Не удалось экспортировать данные: {str(e)}")

config = {

    'dbname': 'Lido',

    'user': 'postgres',

    'password': '1234',

    'host': 'localhost',

    'port': '5432'

}

if \_\_name\_\_ == "\_\_main\_\_":

    db = DBManager(config)

    app = QueryManagerApp(db)

    app.mainloop()