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

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.

1 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Исходный код программы приведен в приложении А.

1.1 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.

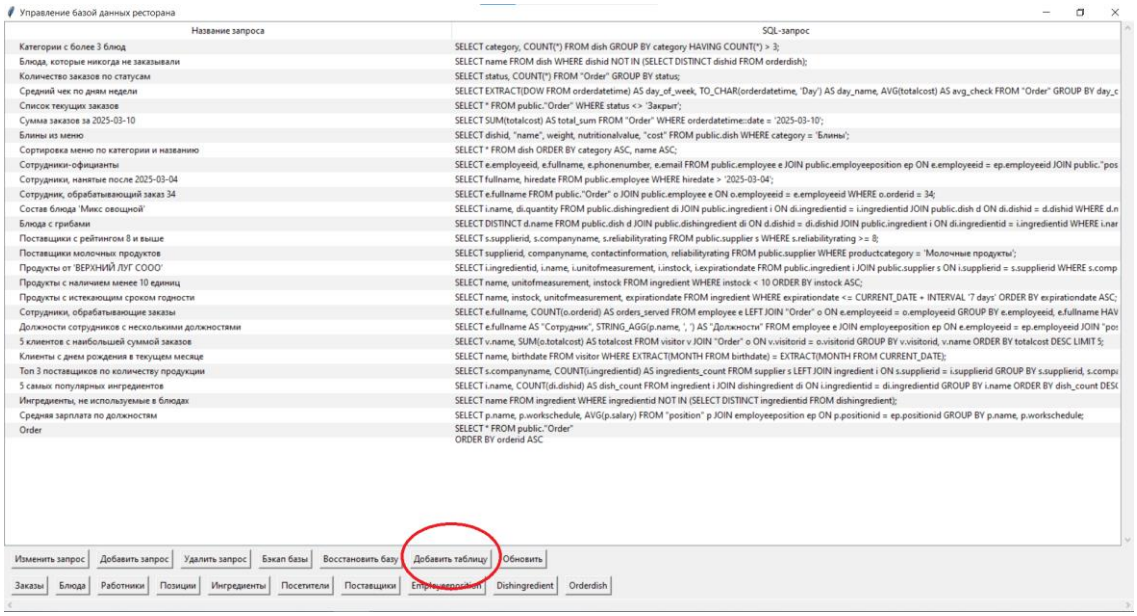


Рисунок 1.1 –Добавление таблицы

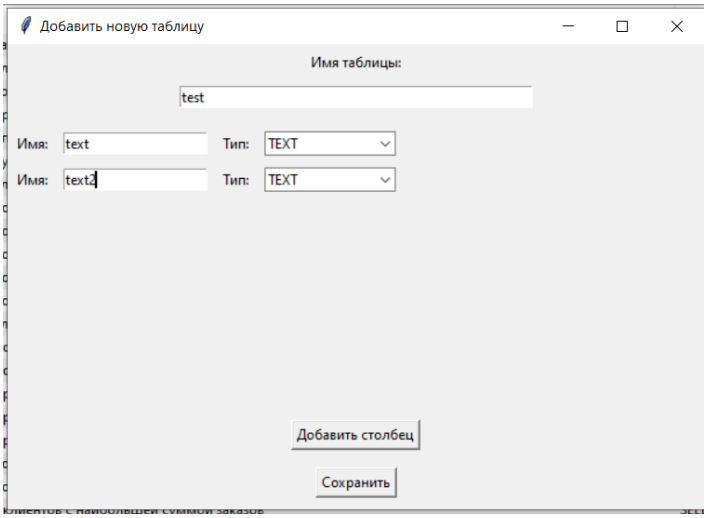


Рисунок 1.2 –Конфигурирование таблицы

1.2 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.

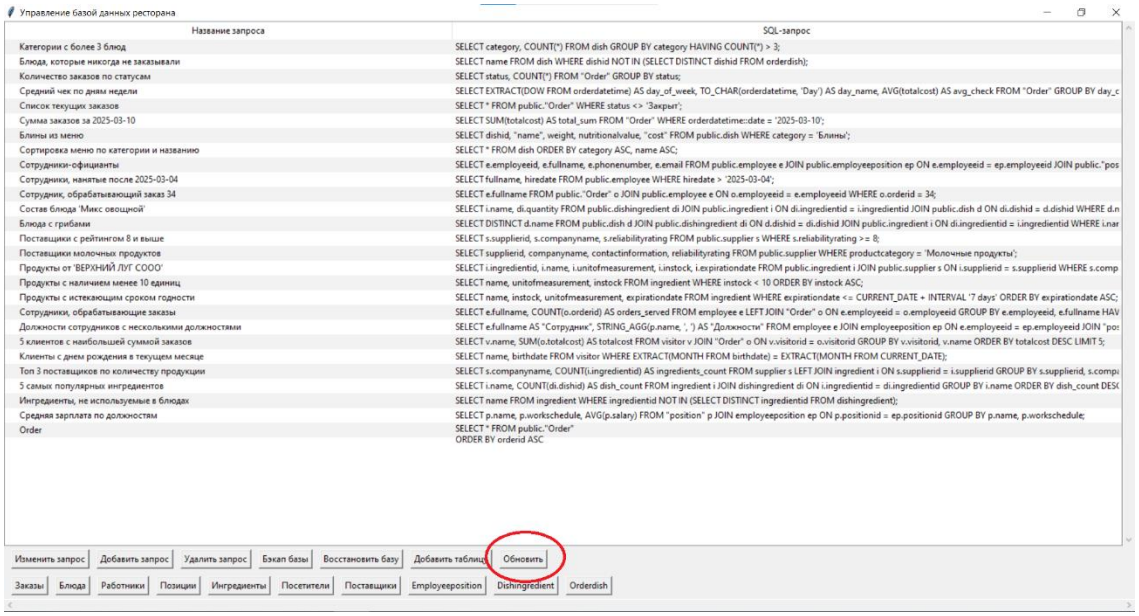


Рисунок 1.3 – Обновление приложения

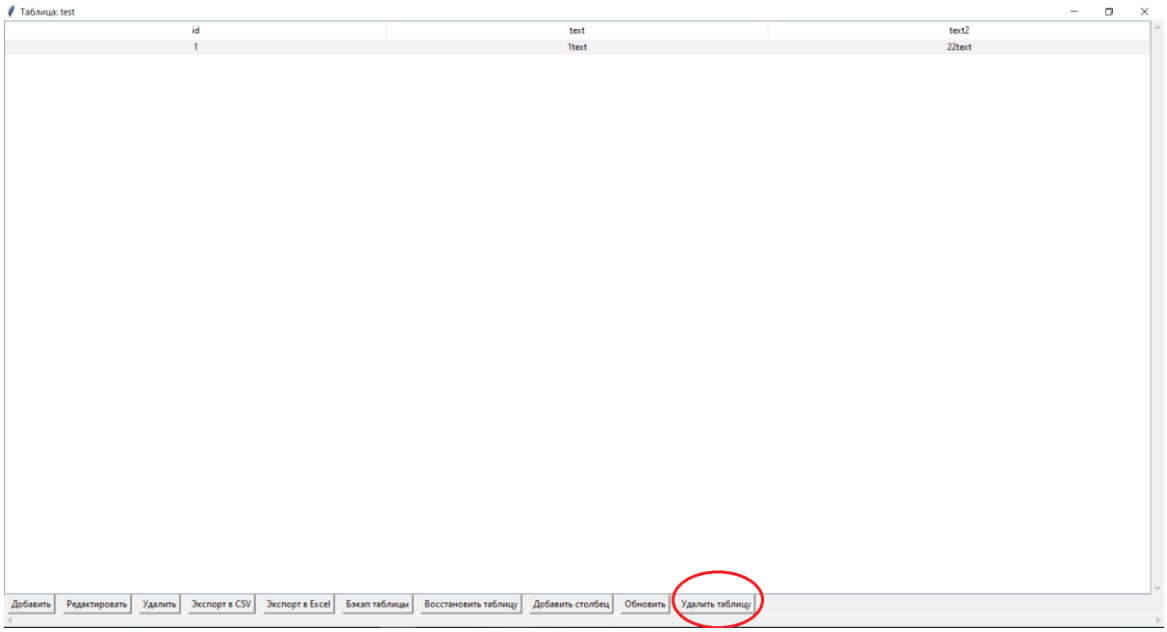


Рисунок 1.4 – Удаление таблицы

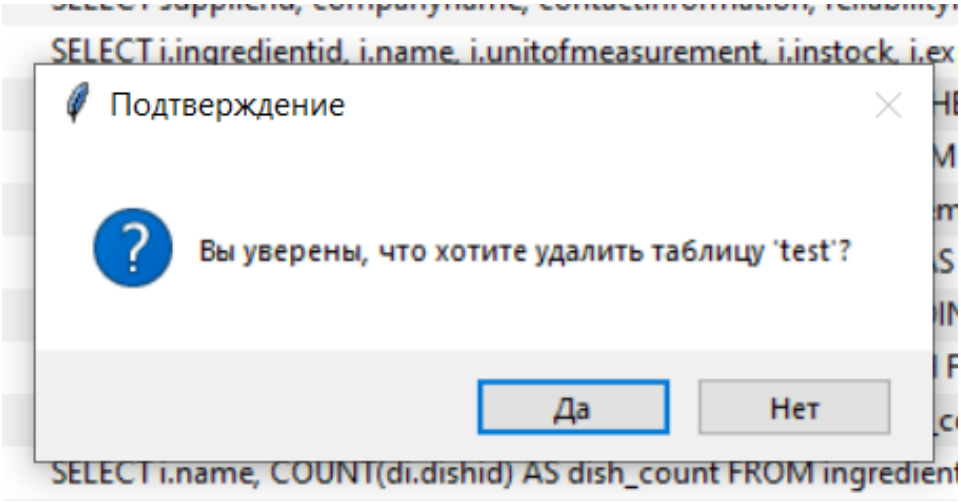


Рисунок 1.5 –Подтверждение удаления

1.3 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.

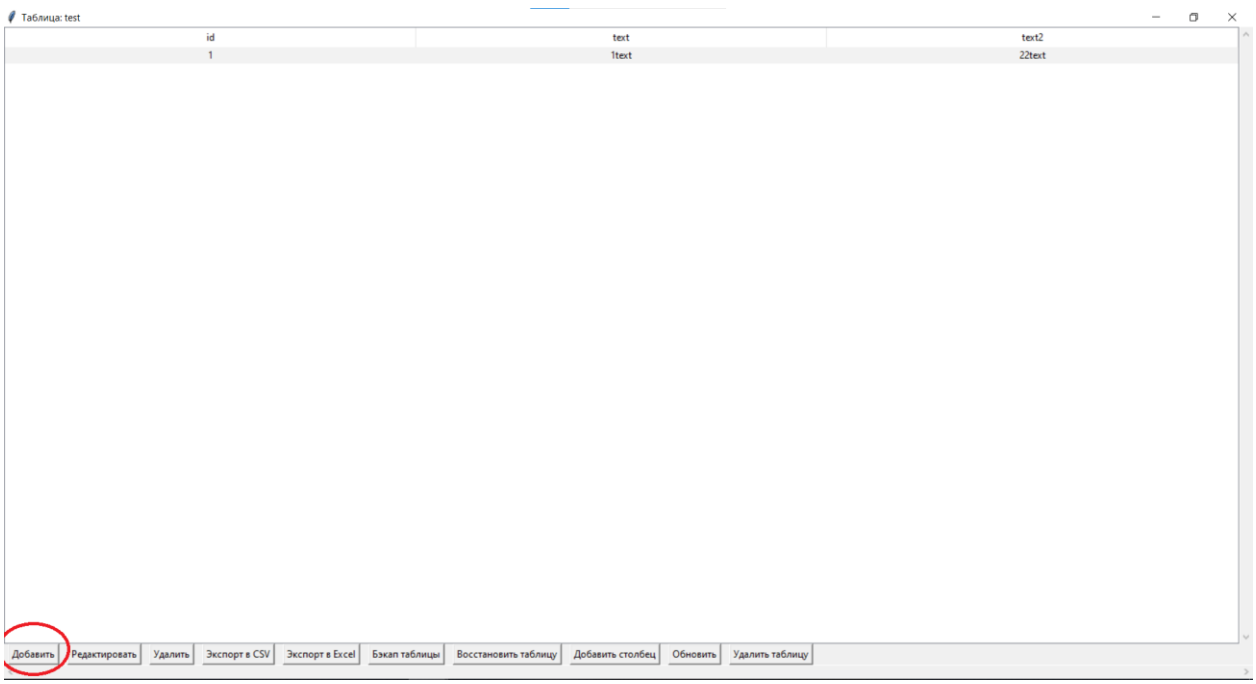


Рисунок 1.6 – Добавить строку

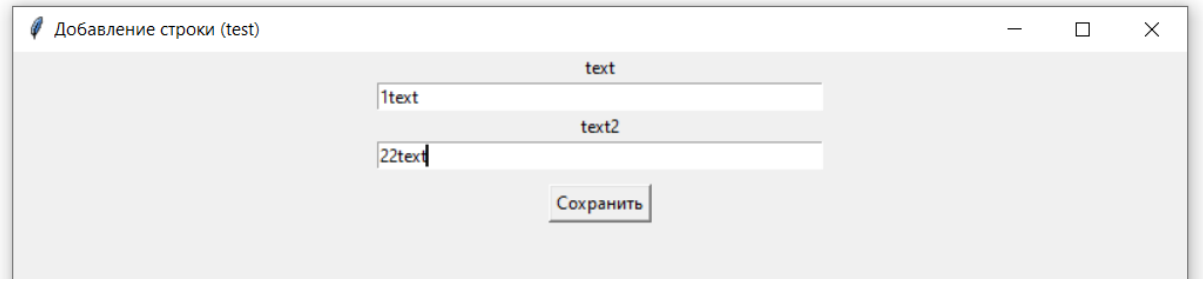


Рисунок 1.7 – Добавление строки

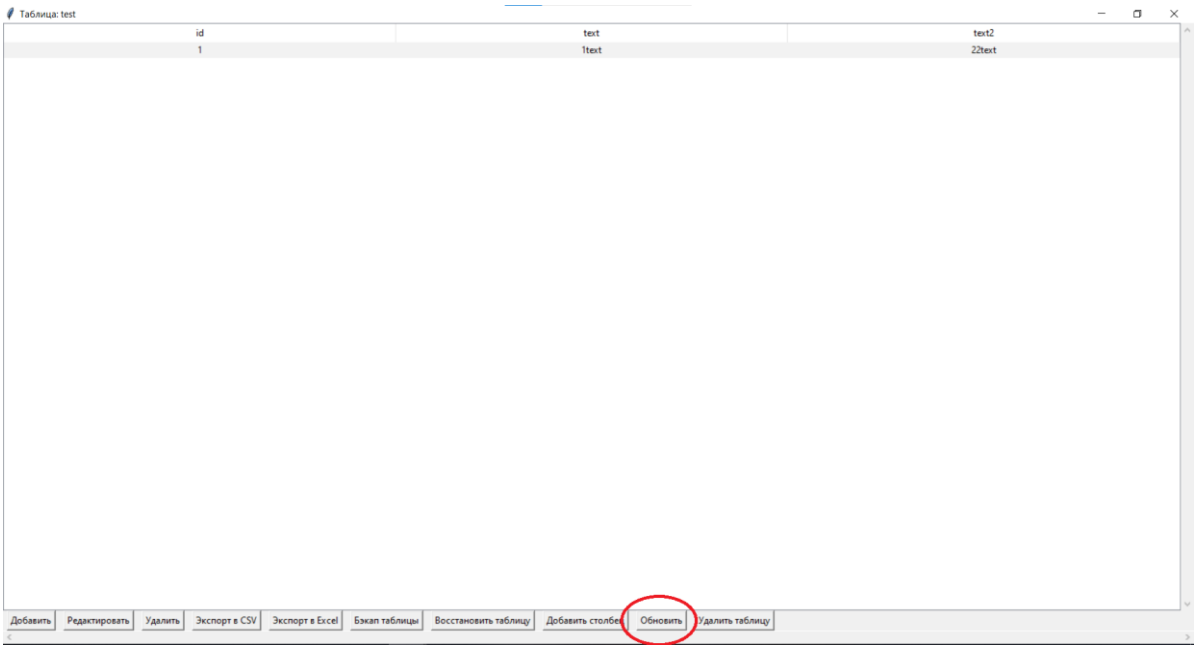


Рисунок 1.8 –Обновить таблицу

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.

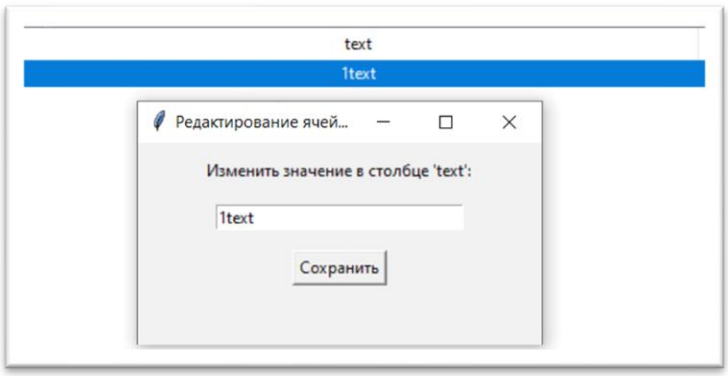


Рисунок 1.9 – Редактирование клетки двойным нажатием

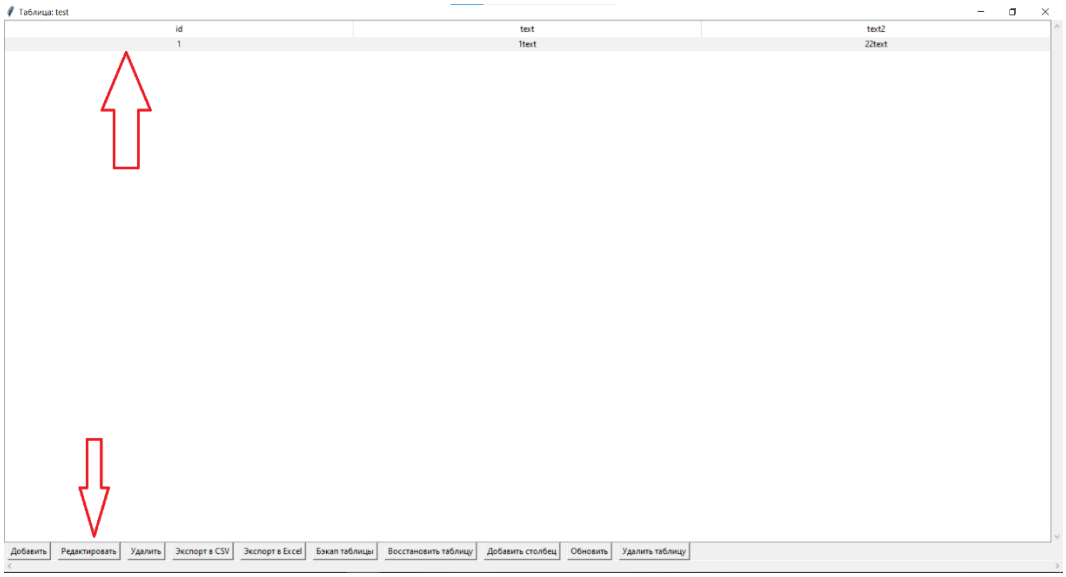


Рисунок 1.10 –Редактирование строки

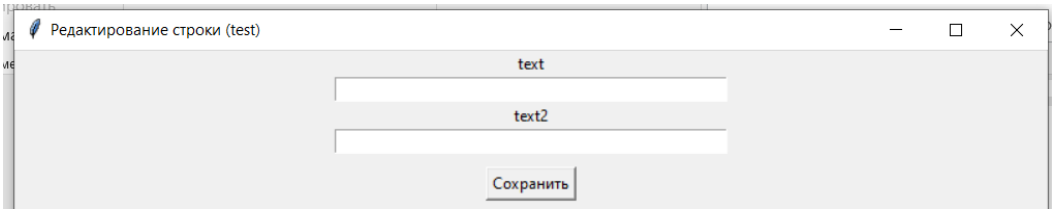


Рисунок 1.11 – Внесение новых данных строки

Для удаления строки необходимо:
1 Обновить таблицу.
2 Выделить строку нажатием, нажать «Удалить» –рисунок 1.12.

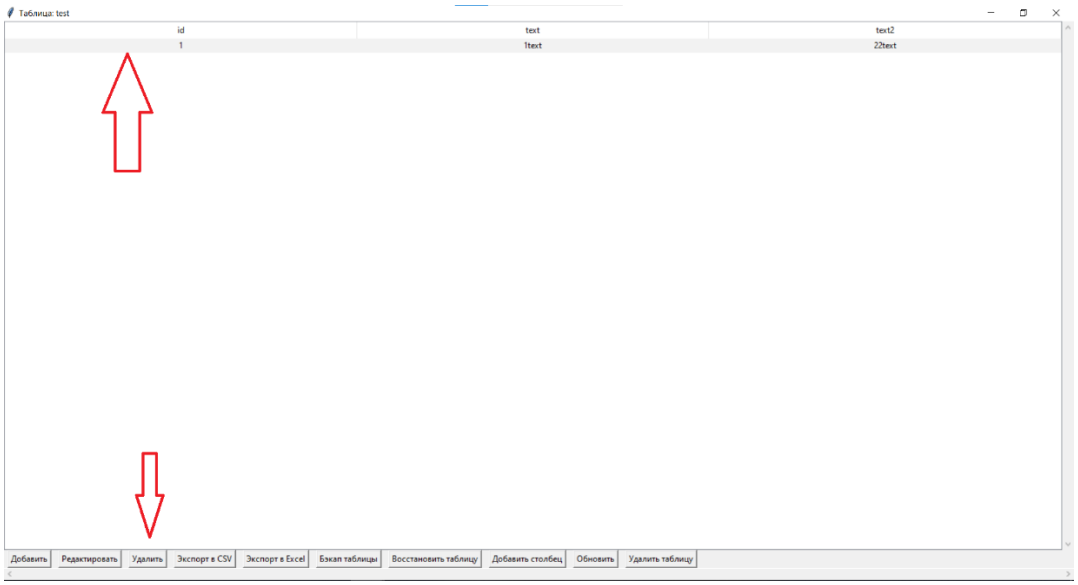


Рисунок 1.12 –Удаление строки

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

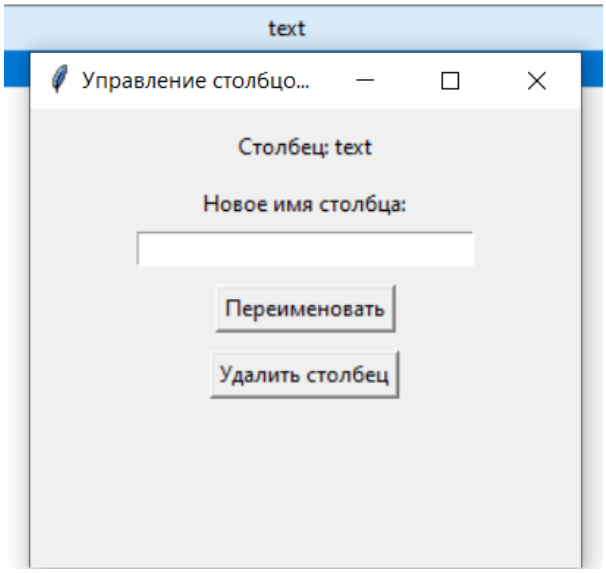


Рисунок 1.13 –Изменение столбца

1.4 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».

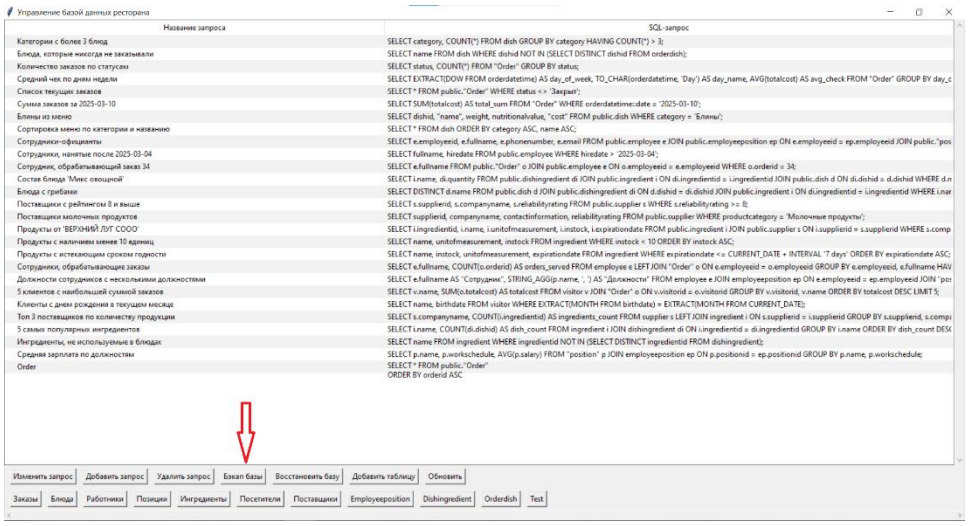


Рисунок 1.14 –Создание резервной копии базы данных

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

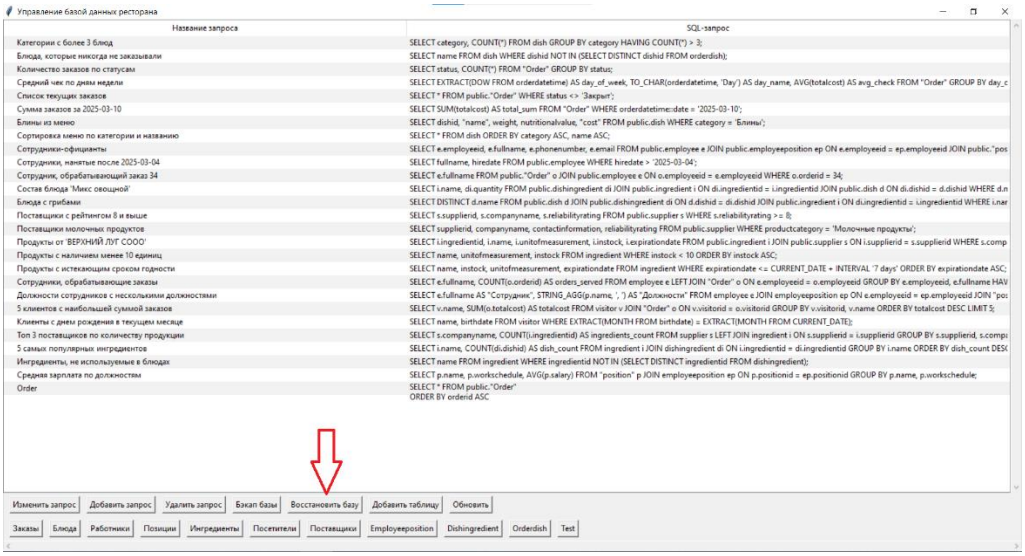


Рисунок 1.15 –Восстановить базу данных

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.

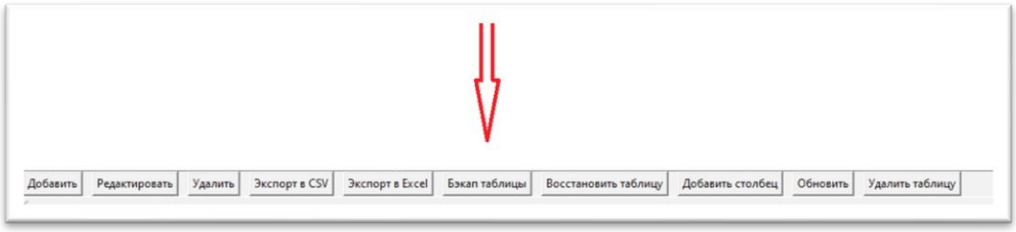


Рисунок 1.16 –Резервное сохранение таблицы

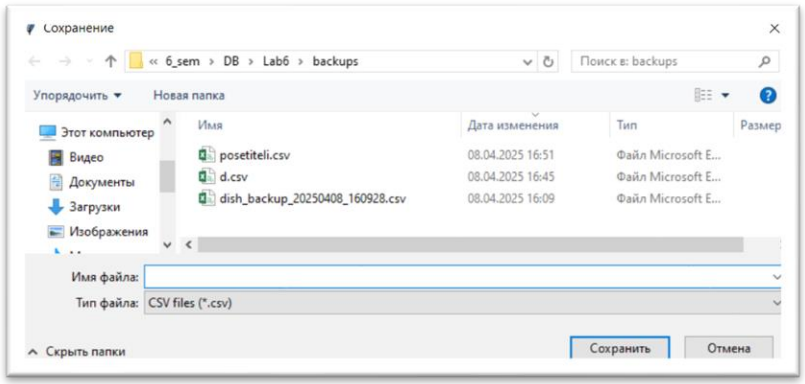


Рисунок 1.17 –Пример выбора пути

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

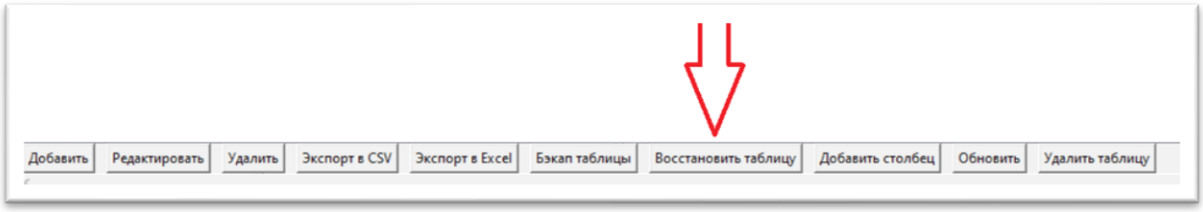


Рисунок 1.18 – восстановление таблицы

1.5 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 «Excel».

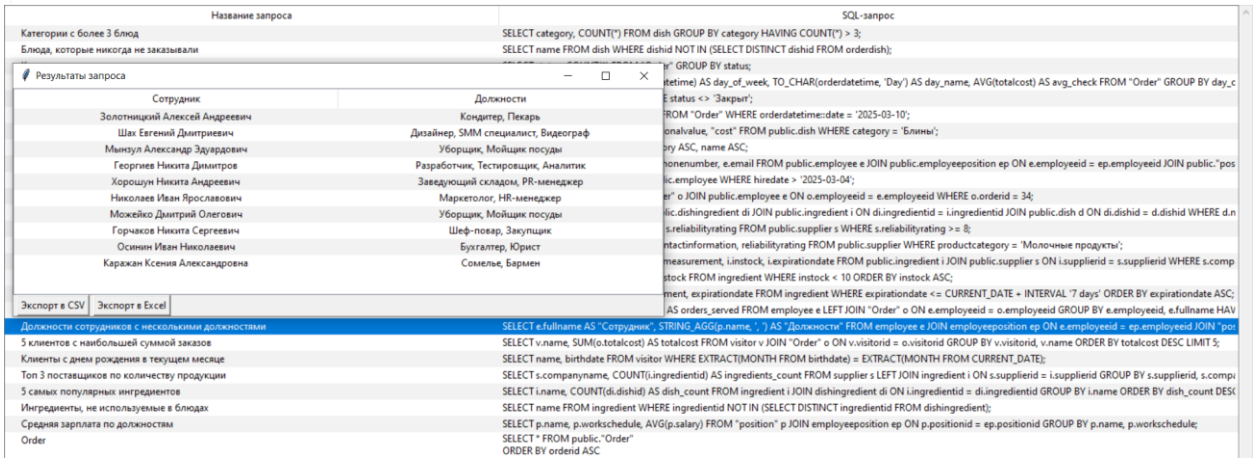


Рисунок 1.19 – Выполнение существующего запроса

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

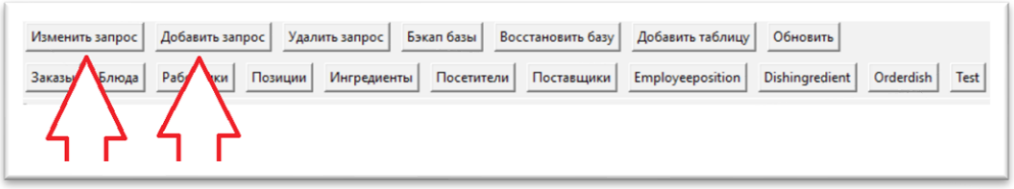


Рисунок 1.20 – Изменение запросов

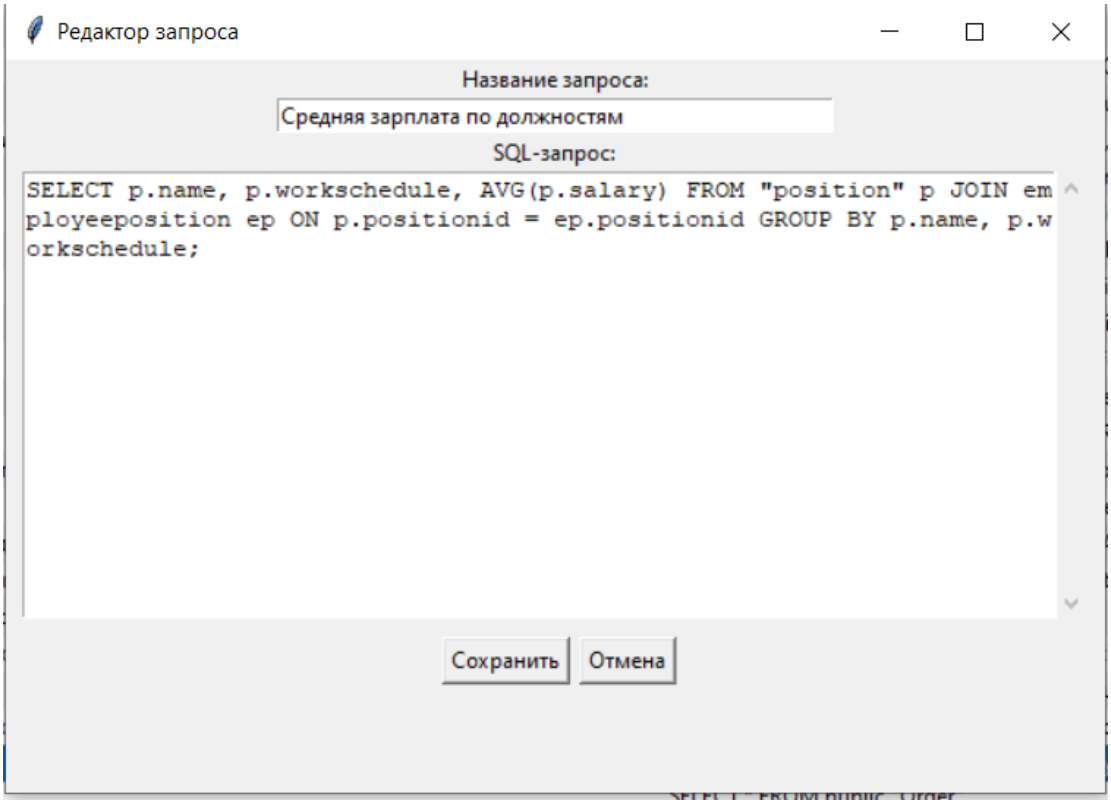


Рисунок 1.21 –Окно редактора запроса

1.6 Экспорт данных

Экспорт доступен для любых запросов и таблиц. Экспорт производится в файлы двух форматов: «SVG» и «Excel». Кнопки экспорта располагаются 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.

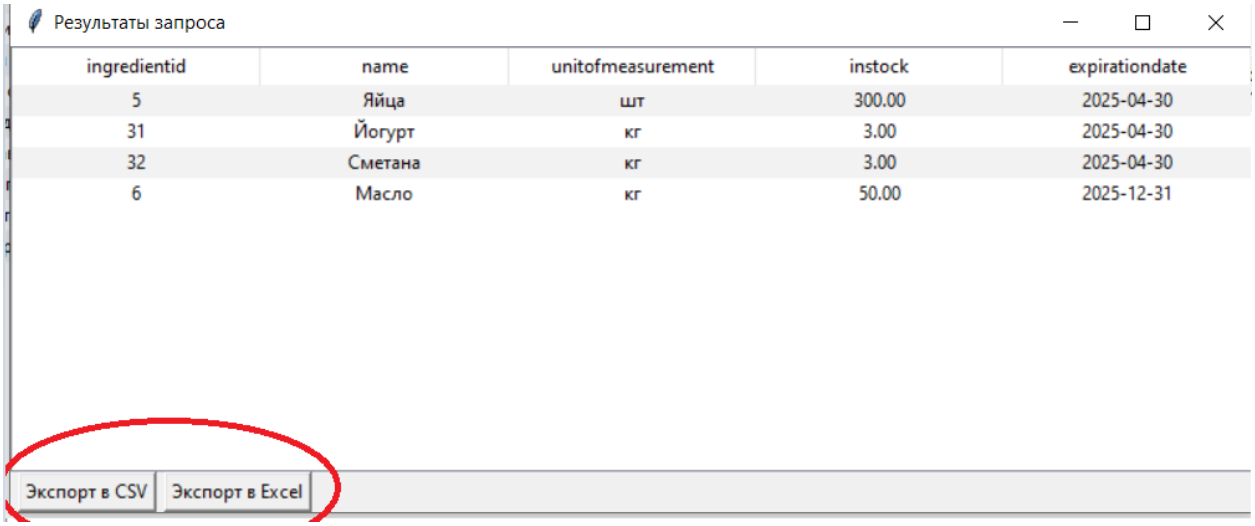
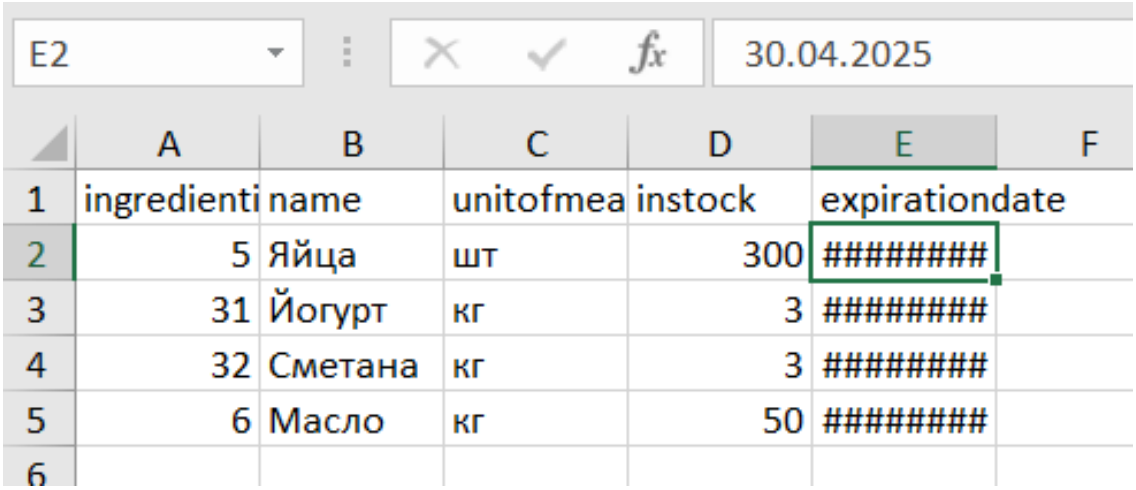
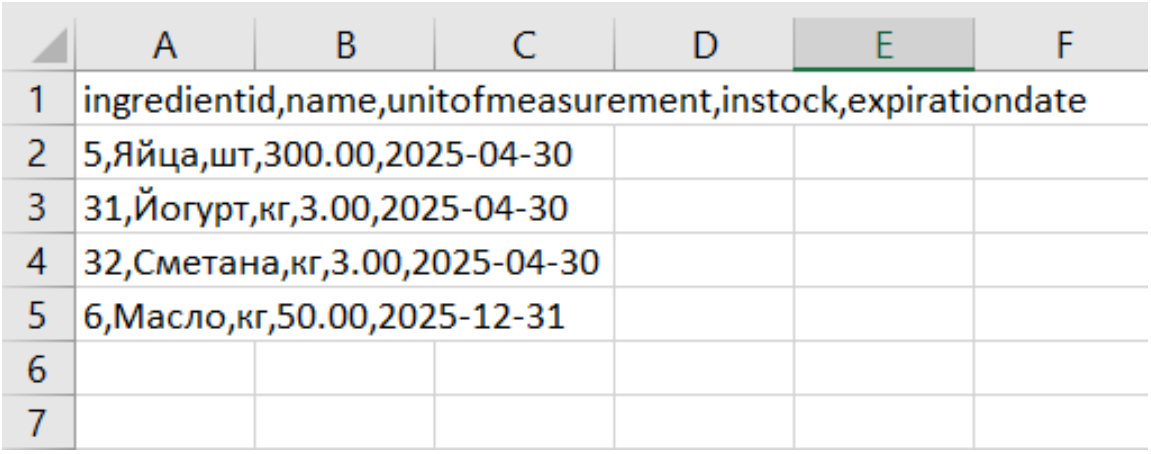


Рисунок 1.22 – Кнопки экспорта результата запроса



	A	B	C	D	E	F
1	ingredienti	name	unitofmea	instock	expirationdate	
2	5	Яйца	шт	300	########	
3	31	Йогурт	кг	3	########	
4	32	Сметана	кг	3	########	
5	6	Масло	кг	50	########	
6						

Рисунок 1.23 –Результат экспорта в «Excel»



	A	B	C	D	E	F
1	ingredientid,name,unitofmeasurement,instock,expirationdate					
2	5,Яйца,шт,300.00,2025-04-30					
3	31,Йогурт,кг,3.00,2025-04-30					
4	32,Сметана,кг,3.00,2025-04-30					
5	6,Масло,кг,50.00,2025-12-31					
6						
7						

1.24 –Результат экспорта в «CSV»

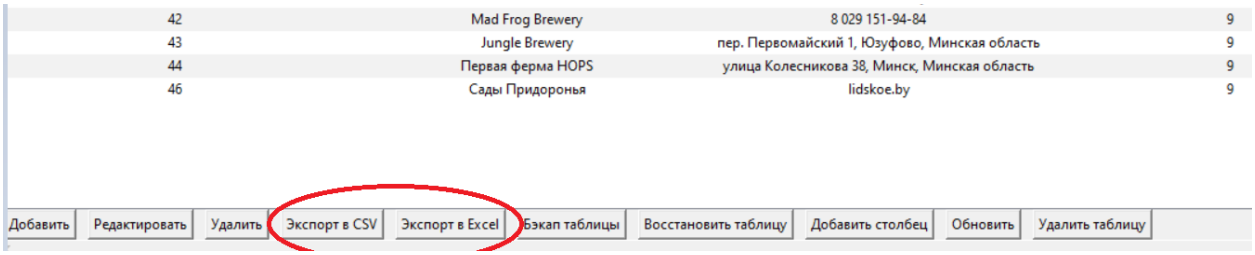


Рисунок 1.25 – Кнопки экспорта таблицы

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()
```