Урок QT SQL 2

Введение в БД, работа с SQL-таблицами и отображение данных в PyQT. Часть 2

Аннотация

В уроке мы продолжим знакомство с базами данных и работой с ними при помощи SQL и Python. Сделаем основной упор на управление данными: запись, изменение и удаление записей.

1. SQL. Продолжение

Давайте снова вернемся в SQLiteStudio, откроем модельную **базу данных** с информацией о фильмах и выполним несколько запросов.

Добавление записей

На прошлом уроке мы познакомились только с одной частью работы с SQL — получение информации с помощью специального запроса `SELECT`. Но перед тем, как получать информацию из базы, необходимо ее туда поместить.

Для этого существует оператор `INSERT`. Его синтаксис в общем случае выглядит так:

```
INSERT INTO имя_таблицы(названия_полей*) VALUES(значения)
```

Названия полей могут не указываться, тогда значения по умолчанию подставятся в поля по порядку. Давайте рассмотрим несколько примеров:

```
INSERT INTO genres(id,title) VALUES(42,'Фантасмагория')
```

```
INSERT INTO genres(title) VALUES('Лекции')
```

И теперь, когда мы захотим вывести таблицу жанров, мы увидим, что там появились новые значения.

21	23	исторический
22	24	нуар
23	42	Фантасмагория
24	43	Лекции

Но откуда взялось значение id, равное 43? Мы же его нигде не указывали? Дело в том, что в нашей таблице поле id является **автоинкрементным**, то есть при создании нового элемента берется максимальный из уже созданных индексов, увеличивается на единицу и присваивается новому элементу.

Часто в таблицу необходимо вставить не одно значение, а несколько. Для это не нужно вызывать оператор `INSERT` несколько раз, а достаточно через запятую перечислить все значения, которые необходимо добавить. Например, так:

```
INSERT INTO genres VALUES (45, 'Научные'), (46, 'Сказки')
```

Изменение записей

Часто случается, что информация, хранящаяся в базе данных, нуждается в изменении. Для таких запросов используется оператор `UPDATE`. Его синтаксис в общем случае выглядит следующим образом:

```
UPDATE имя_таблицы
SET название_колонки = новое_значение
WHERE условие
```

Обратите внимание: если не указать условие обновления, изменятся **абсолютно все** записи в таблице.

Перейдем к примеру. Предположим, что мы посмотрели режиссерскую версию фильма «Аватар», и теперь хотим указать новую продолжительность фильма. Разумеется, можно сохранить информацию, удалить запись и создать новую. Можно, но это крайне неудобно. Так что обновим значение поля `duration` для фильма «Аватар» .

```
UPDATE films
SET duration = '162'
WHERE title = 'Аватар'
```

	id	title	year	genre	duration
1	54	Аватар	2009	11	162

Удаление записей

Но бывают и ситуации, когда какая-либо информация уже совершенно точно не нужна. Тогда ее необходимо удалить из БД, чтобы она не занимала лишнее пространство в памяти. Для этого используется оператор `DELETE`.

Например, чтобы удалить все фильмы, выпущенные до 1985 года, необходимо написать следующий запрос:

```
DELETE from films
where year < 1985
```

Обратите внимание: если не указать условие, будут удалены **абсолютно все** записи в таблице. Во многие менеджеры даже встроен запрос подтверждения действия при выполнении запроса `DELETE` без части `WHERE`.

2. Модификация из Python

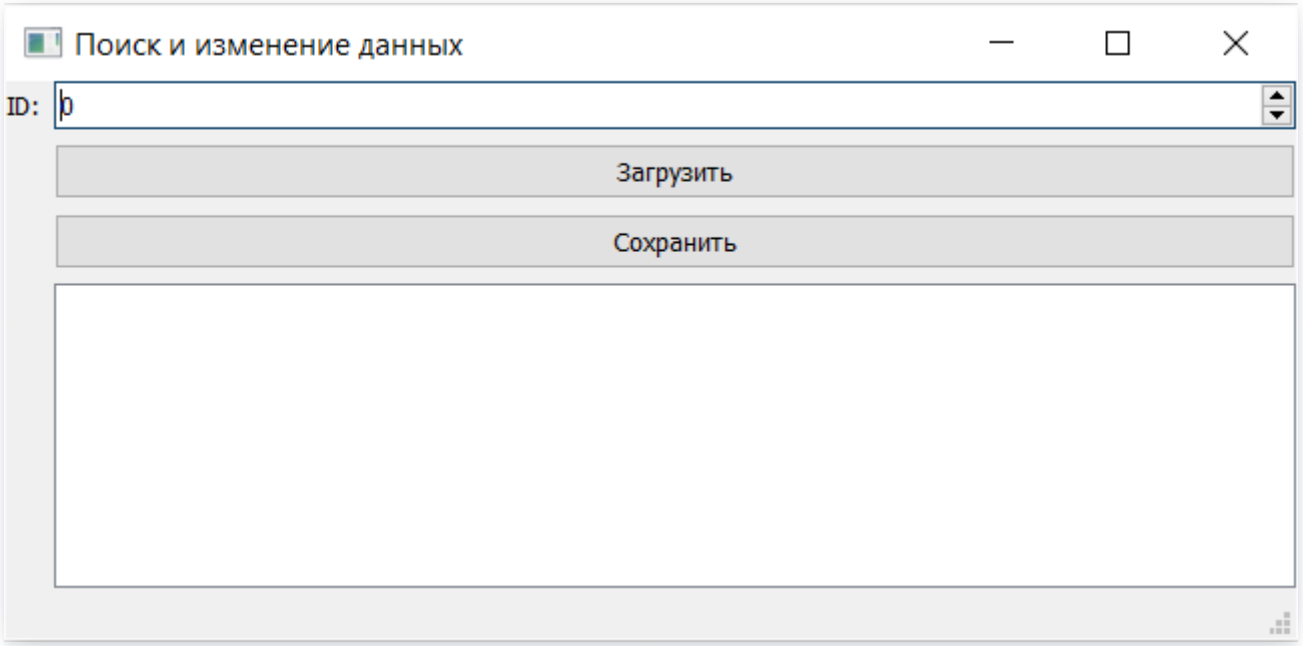
Редактирование данных

В прошлом уроке мы научились получать информацию из БД и отображать ее в таблице. Но достаточно часто пользователям нужно не только просматривать, но и модифицировать и удалять информацию. Напишем программу, которая позволит получить фильм по его идентификатору, изменить его поля и сохранить.

Создадим интерфейс с помощью QtDesigner. Разместим на форме:

- Виджет для ввода id записи, например, QSpinBox
- Две кнопки. Одну — для получения информации из базы данных, другую — для сохранения изменений
- QTableWidget для отображения информации

Упакуем это все с использованием какого-нибудь Layout. В результате у нас должно получиться что-то вроде такого:



Начнем писать наше приложение с создания класса и заглушек для методов.

```
import sqlite3
import sys

from PyQt5 import uic
from PyQt5.QtWidgets import QApplication
from PyQt5.QtWidgets import QMainWindow, QTableWidgetItem


class MyWidget(QMainWindow):
    def __init__(self):
        super().__init__()
        uic.loadUi("UI1.ui", self)
        self.con = sqlite3.connect("films_db.sqlite")
        self.pushButton.clicked.connect(self.update_result)
        self.tableWidget.itemChanged.connect(self.item_changed)
        self.pushButton_2.clicked.connect(self.save_results)
        self.modified = {}
        self.titles = None

    def update_result(self):
```

```
        pass

    def item_changed(self, item):
        pass

    def save_results(self):
        pass


if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = MyWidget()
    ex.show()
    sys.exit(app.exec())
```

Для того чтобы обработать изменение содержимого ячейки, воспользуемся сигналом `self.tableWidget.itemChanged`.

Начнем с метода получения данных `update_result()`. Тут все просто. Единственное, на что стоит обратить внимание — получение данных о заголовках столбцов из параметра курсора.

```
def update_result(self):
    cur = self.con.cursor()
    # Получили результат запроса, который ввели в текстовое поле
    result = cur.execute("SELECT * FROM films WHERE id=?",
                         (item_id := self.spinBox.text(), )).fetchall()
    # Заполнили размеры таблицы
    self.tableWidget.setRowCount(len(result))
    # Если запись не нашлась, то не будем ничего делать
    if not result:
        self.statusBar().showMessage('Ничего не нашлось')
        return
    else:
        self.statusBar().showMessage(f"Нашлась запись с id = {item_id}")
    self.tableWidget.setColumnCount(len(result[0]))
    self.titles = [description[0] for description in cur.description]
    # Заполнили таблицу полученными элементами
    for i, elem in enumerate(result):
        for j, val in enumerate(elem):
            self.tableWidget.setItem(i, j, QTableWidgetItem(str(val)))
    self.modified = {}
```

Теперь напишем метод, который будет отслеживать изменения ячеек. Будем записывать в словарь пару, состоящую из наименования поля и нового значения.

```
def item_changed(self, item):
    # Если значение в ячейке было изменено,
    # то в словарь записывается пара: название поля, новое значение
    self.modified[self.titles[item.column()]] = item.text()
```

Обратите внимание на параметр `item`. При изменении ячейки в него будет попадать объект класса `QTableWidgetItem`, у которого можно получить интересующую нас информацию с помощью методов `column()`

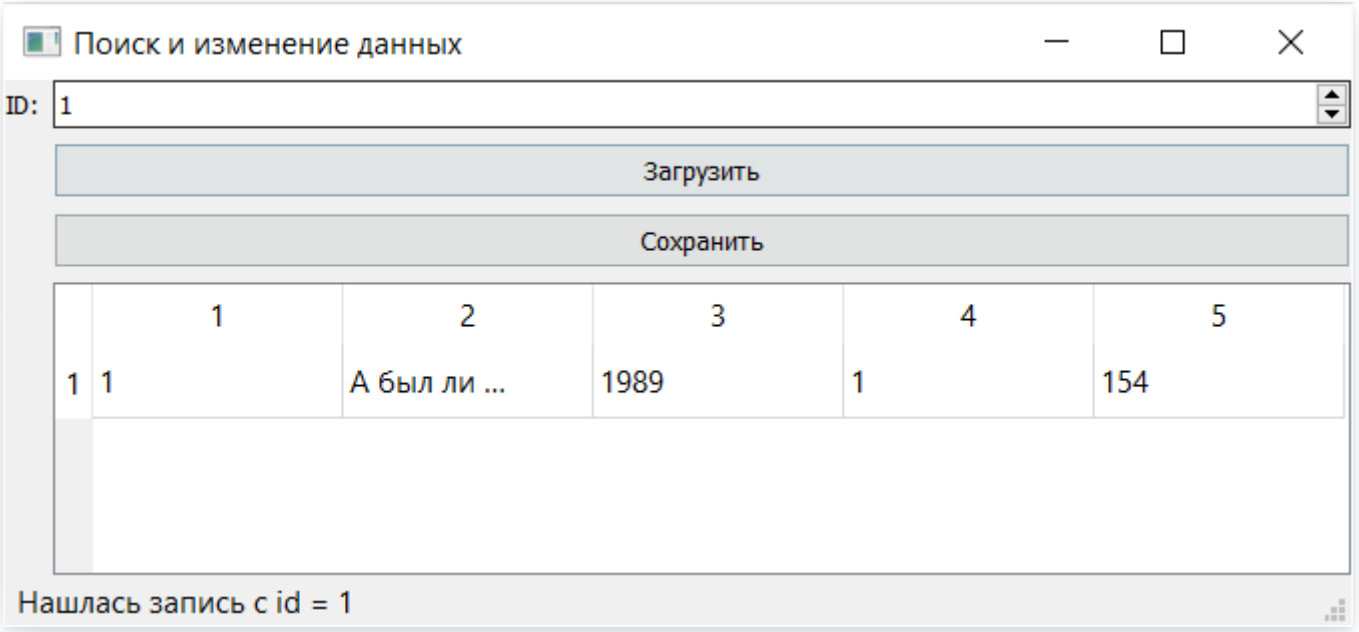
и `text()`.

Затем нам необходимо сохранить полученные результаты в БД. Так как заранее мы не знаем, какие поля были модифицированы, то мы не можем использовать конструкцию с вопросительным знаком. Так что просто «склеим» необходимый нам запрос, например, используя обыкновенную конкатенацию строк.

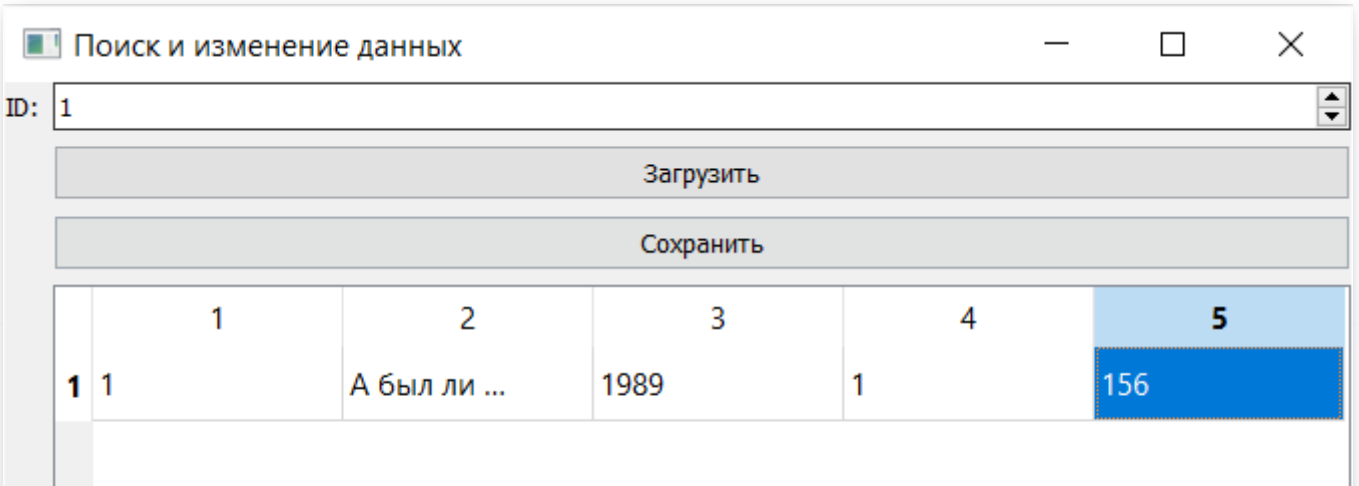
```
def save_results(self):
    if self.modified:
        cur = self.con.cursor()
        que = "UPDATE films SET\n"
        que += ", ".join([f"{key}='{self.modified.get(key)}'"
                          for key in self.modified.keys()])
        que += "WHERE id = ?"
        print(que)
        cur.execute(que, (self.spinBox.text(),))
        self.con.commit()
        self.modified.clear()
```

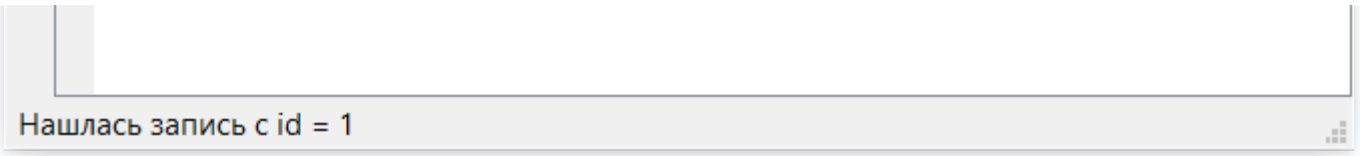
Одна из самых важных строк в `save_results` — вызов метода `commit()` у объекта-соединения с базой данных. По умолчанию все изменения с базой данных проводятся в памяти и не записываются в сам файл, поэтому если мы совершим манипуляцию с данными и перезапустим приложение, все правки пропадут. Чтобы такого не происходило, изменения надо зафиксировать. За это как раз и отвечает метод `commit()`.

Изначальное отображение:



После изменения и сохранения:





На практике изменение записи непосредственно в таблице используется не так часто. Обычно для редактирования существующей или создания новой записи производится в отдельной форме. Так лучше делать по нескольким причинам:

1. Более широкие возможности для пользователя благодаря правильному подбору виджетов для каждого отдельного типа данных.
2. Меньше возможности пользователю отредактировать не те данные.
3. Проще отслеживать и сохранять изменения.

Удаление данных

Теперь рассмотрим программу для удаления выделенного элемента.

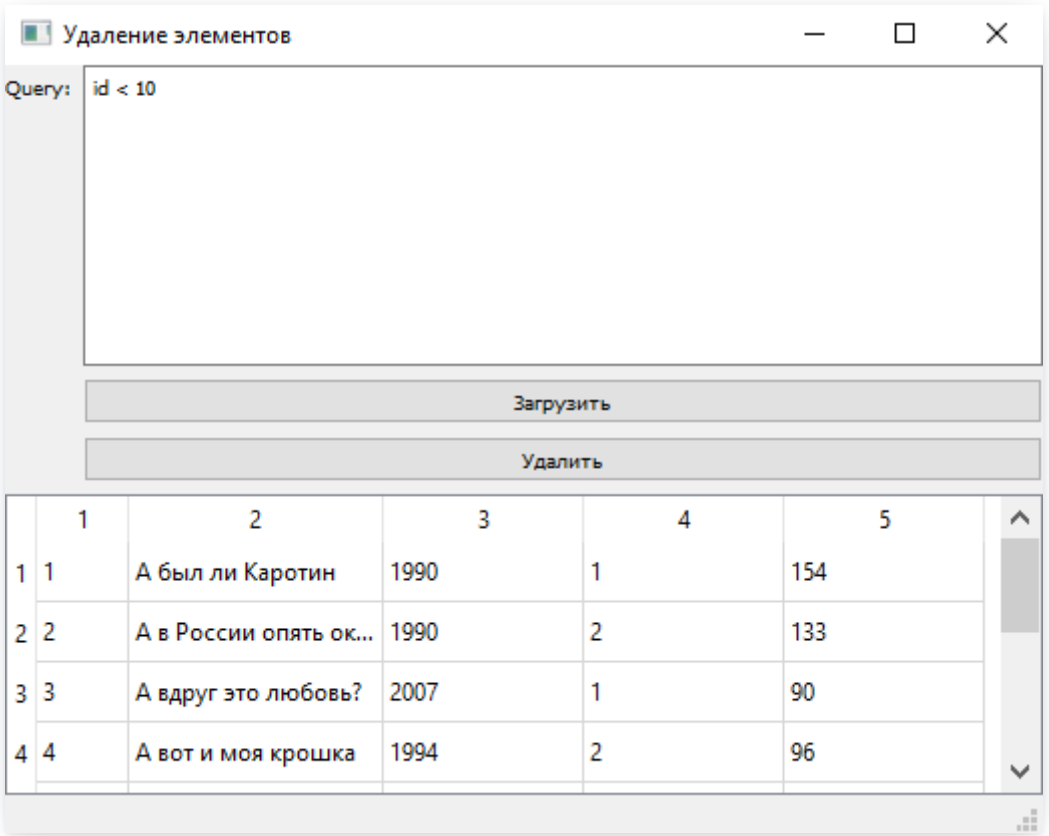
Часто перед удалением какой-либо информации различные программы запрашивают у пользователя подтверждение. Сделаем программу с похожей функциональностью. Будем работать только с таблицей films. Создадим интерфейс нашей программы. Разместим на форме следующие виджеты:

– Виджет для ввода параметров фильтрации, например, например, QTextEdit

– Две кнопки. Одну — для получения информации из базы данных, другую — для удаления записей

– QTableWidget для отображения информации

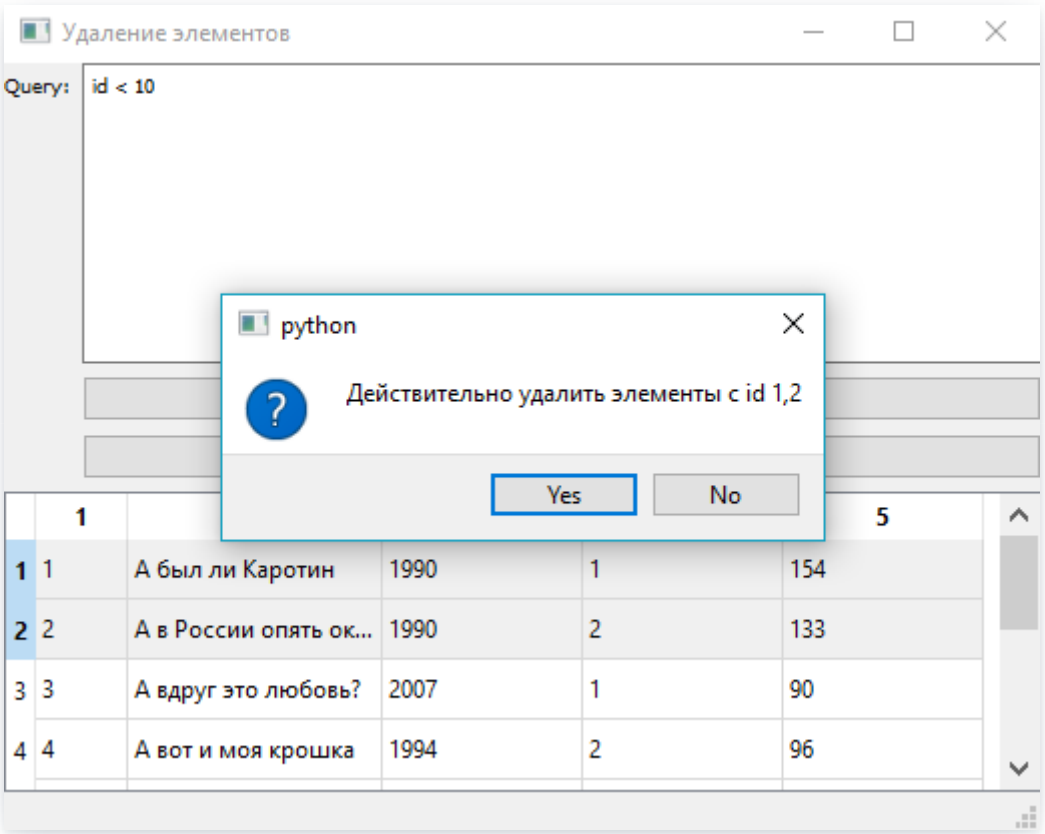
Снова запакуем в какой-нибудь Layout. Результат будет выглядеть как-то так:



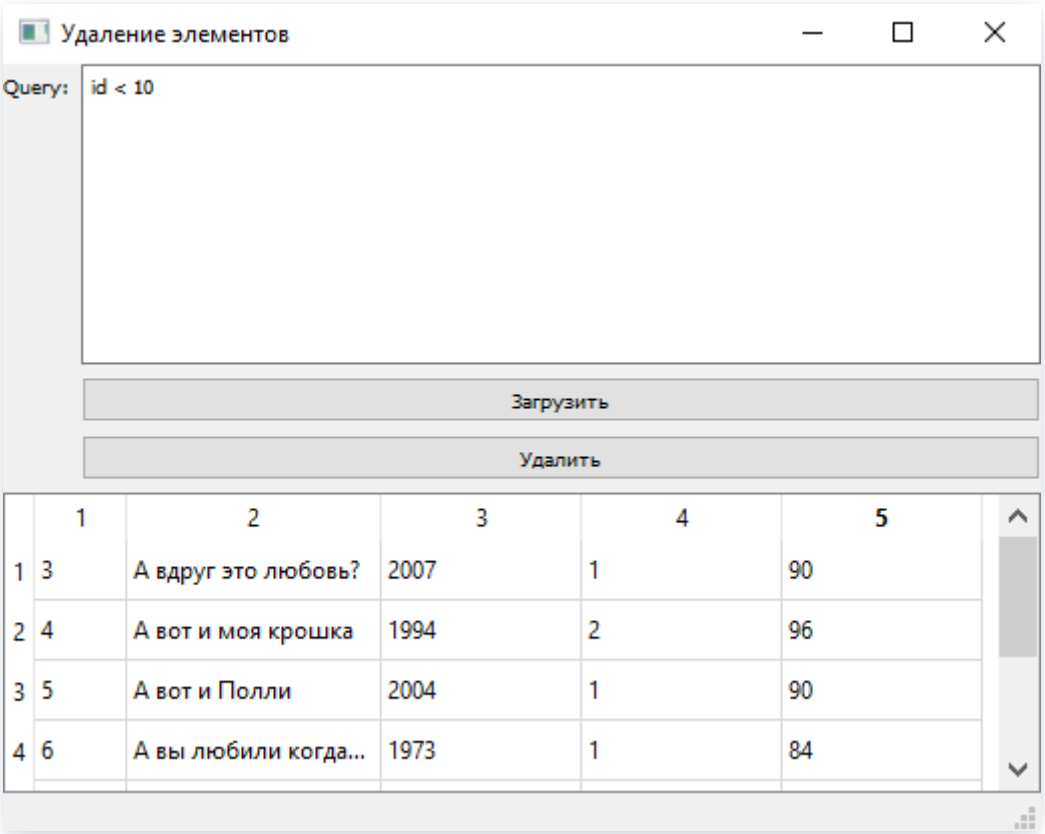
В поле для ввода будем вводить только условие (то, что будет идти после оператора WHERE). А сам запрос будет формироваться так:

```
queue = "SELECT * FROM films WHERE " + self.textEdit.toPlainText()
```

Пользователь может выделить одно или несколько значений, а затем нажать на кнопку «Удалить». После этого открывается окно с подтверждением, в котором указаны все ID, которые были выделены.



В случае положительного ответа при повторном запросе удаленные элементы уже не отобразятся.



Основной метод, отличающий этот пример от предыдущего — `delete_elem()` для удаления выделенных элементов. Рассмотрим его поподробнее.

```
def delete_elem(self):
    # Получаем список элементов без повторов и их id
    rows = list(set([i.row() for i in self.tableWidget.selectedItems()]))
    ids = [self.tableWidget.item(i, 0).text() for i in rows]
```

```
    # Спрашиваем у пользователя подтверждение на удаление элементов
    valid = QMessageBox.question(
        self, '', "Действительно удалить элементы с id " + ",".join(ids),
        QMessageBox.Yes, QMessageBox.No)
    # Если пользователь ответил утвердительно, удаляем элементы.
    # Не забываем зафиксировать изменения
    if valid == QMessageBox.Yes:
        cur = self.con.cursor()
        cur.execute("DELETE FROM films WHERE id IN (" + ", ".join(
            '?' * len(ids)) + ")", ids)
        self.con.commit()
```

В первой строке мы получаем список строк без повторов. Откуда взялись повторы? Выделяя строку, мы выделяем все элементы строки (все ячейки), соответственно, в список добавляется номер строки столько раз, сколько в ней элементов.

Поскольку номера строк очень часто не совпадают с идентификаторами, во второй строке мы получаем id записей, по которым мы и будем производить удаление, непосредственно из значений ячейки в нулевой колонке.

Теперь нам необходимо показать пользователю форму с подтверждением своего действия. Можно воспользоваться уже знакомым нам `QInputDialog` и написать что-то вроде:

```
answer, ok_pressed = QInputDialog.getItem(
    self, "Подтверждение удаления",
    "Вы точно хотите удалить элементы с id " + ",".join(ids),
    ("нет", "да"), 1, False)
```

Однако для отображения пользователю диалога о подтверждении какого-либо действия или информационного сообщения о каком-нибудь системном событии (например, об ошибке) в PyQT есть более привычный и подходящий инструмент. Это класс `QMessageBox`, у которого с `QInputDialog` общий родитель — `QDialog`. Поэтому импортируем его из PyQt5.QtWidgets, вызовем метод `question()`, в который передаются следующие параметры:

- Родитель — `self`
- Заголовок — обычно передается пустое поле, если мы хотим задать пользователю вопрос
- Текст вопроса
- Варианты ответов — `QMessageBox.Yes`, `QMessageBox.No`

Возможности `QMessageBox` достаточно широки, рекомендуем ознакомиться с ними в **документации**.

После того как пользователь нажмет на одну из кнопок, результат будет занесен в переменную `valid`. А затем будет выполнена проверка и удаление.

Важно обратить внимание на то, что текст запроса формируется с использованием и конкатенации строк, и оператора `"?"`. В данной задаче мы также столкнулись с методом `commit()` у соединения с базой данных. Не забывайте фиксировать изменения после изменения данных или их удаления.

3. Заключение

Мы рассмотрели основные запросы SQL и теперь умеем все, что необходимо для написания полноценных приложений для работы с данными, хранящимися в SQLite базе данных. Конечно, возможности SQL

значительно шире, а уже рассмотренные операторы умеют больше. Уверенные знания языка запросов — «must have» для значительной части разработчиков программного обеспечения, поэтому настоятельно рекомендуем вам самостоятельно углубиться в эту тему. К счастью, в интернете существует огромное количество материалов, в том числе и бесплатных.

Справка