Паттерны хранения данных

Существует множество уже готовых архитектурных шаблонов, позволяющих сделать процесс хранения и взаимодействия с данными удобнее и проще для разработчика.

В данном документе мы разберем два наиболее часто встречающихся.

Логическое удаление

Зачастую возникает задача исключить запись из расчетов, скажем у нас есть таблица с клиентами и нам необходимо "удалить" клиентов, которые от нас ушли. Однако велика вероятность, что клиент к нам вернется и при обычном удалении через команду **delete** нам придется собирать данные о клиенте с нуля, для множества процессов в нашей компании (построение рекомендаций, формирование групп клиентов и т.д.)

Однако есть подход, позволяющий после удаления восстановить запись о пользователе (**логическое удаление**). Для реализации данного подхода мы добавляем поле deleted_flg в нашу таблицу, которое может принимать в качестве значения только 0 или 1, где 0 обозначает, что запись не удалена, а 1 напротив, что запись удалена.

Так, мы можем сформировать представление с небольшим условием, которое будет хранить только актуальные данные, а при необходимости мы легко восстановим запись о клиенте изменив значение deleted_flg.

Ниже представлен скрипт на Python с примером реализации логического удаления.

```
import sqlite3

con = sqlite3.connect('sber.db')
cursor = con.cursor()

def init_table():
  cursor.execute('''
    CREATE TABLE if not exists product(
      id integer primary key,
      title varchar(128),
      price integer,
      deleted_flg integer default 0
    )''')
```

```
  cursor.execute('''
    CREATE VIEW if not exists v_product as
      select id, title, price from product
      where deleted_flg = 0;
    ''')

def deleteRow(id):
  cursor.execute('''
      update product
      set deleted_flg = 1
      where id = ?
    ''', [id])
  con.commit()

def repairRow(id):
  cursor.execute('''
      update product
      set deleted_flg = 0
      where id = ?
    ''', [id])
  con.commit()

def addRow(title, price):
  cursor.execute('''
    INSERT INTO product(title, price) values(?, ?);
    ''', [title, price])
  con.commit()

def showTable(table):
  cursor.execute(f'select * from {table}')

  for row in cursor.fetchall():
    print(row)


init_table()
# addRow('велосипед1', 9000)
# addRow('велосипед2', 9000)
# addRow('велосипед3', 9000)
# addRow('велосипед4', 9000)
# addRow('велосипед5', 9000)
deleteRow(3)
deleteRow(5)
repairRow(3)
showTable('v_product')
```

В скрипте выше представлены следующие функции

init_table() - данная функция создает (в случае, если их еще нет) таблицу и представление с актуальным срезом.

deleteRow(id) - данная функция логически удаляет строку по id (которое получает в качестве аргумента)

repairRow(id) - данная функция логически восстанавливает строку по id (которое получает в качестве аргумента)

addRow(title, price) - данная функция получает в качестве аргумента название товара и его цену и добавляет запись о нем в таблицу.

showTable(table) - данная функция получает в качестве аргумента название таблицы и выводит данные из нее в командную строку.

Хранение истории.

Иногда для обеспечения корректной работы процесса нам бывает необходимо хранить историю изменения данных. Реализуется данный процесс с помощью полей **start_dttm** и **end_dttm.** Они позволяют хранить информацию о начале и конце периода, когда эта строка была актуальной. В случае, если запись является актуальной на данный момент (нет даты конца актуальности записи) **end_dttm** присваивается значение технической бесконечности (2999-12-31 23:59:59).

При таком способе хранения данных для формирования актуального среза вам достаточно просто указать дату, на какой период вас интересует состояние и проверить start_dttm и end_dttm через between.

Ниже представлен скрипт на Python с примером реализации хранения истории.

```
import sqlite3

con = sqlite3.connect('sber_1.db')
cursor = con.cursor()

def init_table():
  cursor.execute('''
    CREATE TABLE if not exists product(
      id integer primary key,
      title varchar(128),
      price integer,
      start_dttm datetime default current_timestamp,
      end_dttm datetime default (datetime('2999-12-31 23:59:59'))
    );
    ''')

def addProduct(title, price):
  cursor.execute('''
    update product
    set end_dttm = datetime('now', '-1 second')
    where title = ?
    and end_dttm = datetime('2999-12-31 23:59:59')
```

```
    ''', [title])

  cursor.execute('''
    INSERT INTO product (title, price) values(?, ?)
    ''', [title, price])

  con.commit()




def showTable(table):
  cursor.execute(f'''select * from {table} ''')

  for row in cursor.fetchall():
    print(row)

init_table()
# addProduct('велосипед', 30000)
# addProduct('ролики', 10000)
addProduct('велосипед', 40000)
showTable('product')
```

Обратите внимание на функцию **addProduct** в ней происходит основной процесс.

1) закрывается текущей датой запись по товару, которая до этого считалась актуальной

2) добавляется новая запись.

Обратите внимание!

Важно избежать нахлеста нескольких промежутков дат. Иначе могут возникнуть дубли. Для этого при указании закрывающей даты необходимо вычесть из нее секунду.

Строчка из кода. **set end_dttm = datetime('now', '-1 second')**

Задание

Создайте таблицу User (id, name, lastname, age, start_dttm, end_dttm) ключом продукта являются имя и фамилия. Id - ключ записи.

и таблицу Products с полями (id, title, price, deleted_flg)

Добавьте следующие функции.

1) удалить товар по id

2) восстановить товар по id

3) добавить пользователя (если добавляется пользователь с уже существующими именем и фамилией, то мы считаем эту запись новой версией данных о пользователе).