Базы данных и язык SQL

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Среда, 16 ноября года

План занятия

1. Основы основ
   - Текстовые файлы, БД и СУБД
   - Реляционные СУБД и простой SQL
   - Использование СУБД
   - Немного практики
2. Запросы посложнее
   - Группировка строк
   - Подзапросы
   - Соединения
   - Объединения результатов запросов

Постановка задачи

- Пусть мы пишем приложения для учёта товаров в магазине.
- Надо знать:
  1. Какой товар есть на складе и витринах.
  2. Где он лежит.
  3. По какой цене товар закуплен (могут быть разные партии).
  4. По какой цене товар сейчас продаётся.
  5. Какие покупки были сделаны (что куплено вместе, на какую сумму, в какое время).
- Возможные события:
  1. Приехала поставка со склада.
  2. Касса пробила чек — совершена покупка.
- Надо, чтобы приложение сохраняло состояние между перезапусками.
- Вопрос: как это сделать?

Вариант с файлами

- Создаём кучу внутренних структур для хранения объектов.
- При выходе из приложения пишем их в каком-то формате не диск, при запуске — считываем.

Проблемы:

Вариант с файлами

- Создаём кучу внутренних структур для хранения объектов.
- При выходе из приложения пишем их в каком-то формате не диск, при запуске — считываем.

Проблемы:

- На запуск и выход требуется существенное время (записать много данных).
- Если пропало питание, мы потеряли данные.

Решения:

Вариант с файлами

- Создаём кучу внутренних структур для хранения объектов.
- При выходе из приложения пишем их в каком-то формате не диск, при запуске — считываем.

Проблемы:

- На запуск и выход требуется существенное время (записать много данных).
- Если пропало питание, мы потеряли данные.

Решения:

- Меняем формат файла, чтобы можно было обновлять только изменившиеся кусочки.
- Храним структуры и объекты в разных маленьких файлах, чтобы изменения обрабатывала файловая система, а не мы.

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Возможные решения:

1. Каждая хранит данные независимо, а после закрытия мы склеиваем данные вместе:

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Возможные решения:

1. Каждая хранит данные независимо, а после закрытия мы склеиваем данные вместе:
   - Несложно пишется и сложно вызвать цепную реакцию из ошибок — ломается только в момент склейки.
   - Так раньше работали банки («ваш перевод будет обработан в течение X дней»).
   - Нет доступа к данным в реальном времени.

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Возможные решения:

2. Дать общий доступ ко всем файлам со всех компьютеров:

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Возможные решения:

2. Дать общий доступ ко всем файлам со всех компьютеров:
   - Появляется конкуренция за доступ и race condition: нельзя, чтобы две программы одновременно работали с одним файлом.
   - Если изменение затрагивает несколько файлов, то надо их все захватывать; не все сетевые ФС так умеют.

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Возможные решения:

3. Написать программу-сервер, которая как-то хранит данные на одном компьютере и обрабатывает сетевые запросы от клиентов:

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Возможные решения:

3. Написать программу-сервер, которая как-то хранит данные на одном компьютере и обрабатывает сетевые запросы от клиентов:
   - Получили абстракцию «хранилище данных».
   - Хранилище инкапсулирует то, как именно данные хранятся (хоть в памяти, хоть на десяти серверах распределённо).

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Общие проблемы:

Усложнение

- А теперь у нас десять касс и два компьютера в разных концах склада.
- Одной программы теперь недостаточно, надо несколько.

Общие проблемы:

- Хранилище сильно завязано на то, какие именно данные оно хранит: формат хранения на диске, протокол, внутренние структуры...
- Не очень много кода в хранилище действительно хочет знать о структуре данных что-то подробнее «число или строка».
- Для любого взаимодействия с хранилищем требуется писать своё новое приложение с поддержкой протокола.

СУБД

- *Система управления базами данных* (СУБД) — это сервис, который умеет хранить данные *произвольной структуры* (в определённых рамках, конечно, не совсем бессистемные).
- *База данных* — это описание данных **и их структуры**, которые хранятся в СУБД.
- СУБД обычно делят на два вида в зависимости от того, как они структурируют данные: реляционные (relational) и нереляционные (non-relational или NoSQL).
- Реляционные — это классика (существуют с 80-х годов), их и будем изучать.
- Нереляционные примерно того же возраста, но вошли в тренд только в последние лет десять.
- Примеры реляционных: MySQL, MariaDB, Oracle, MS SQL, Sqlite.
- Примеры нереляционных: MongoDB, Redis, Memcached, Cassandra.

Реляционные СУБД на практике

- СУБД хранит одну или несколько независимых БД (баз данных).
- Каждая БД — это набор таблиц, которые содержат данные.
- Таблица имеет фиксированный набор столбцов с названиями и типами.
- Фиксированный в каждый момент времени; вообще столбцы можно добавлять, менять, удалять, хоть это и сложные для СУБД операции.
- В таблице лежит неупорядоченный набор строк с данными.
- На столбцы (или их группы) могут накладываться дополнительные ограничения (например, «все значения в столбце различны»).
- Обычно запросы к реляционным СУБД формулируются на декларативном языке SQL (Structured Query Language).

Реляционная алгебра

Математическая модель происходящего в реляционных СУБД:

- Таблица называется *отношением* (relation, отсюда relational database).
- Есть операции над таблицами (образующие алгебру). Например, «выбрать какие-то строчки из таблицы».
- Обычно СУБД поддерживают гораздо более крутые и странные операции, чем в реляционной алгебре.
- Больше слова «реляционная алгебра» вам наверняка не пригодятся.

Классическая шутка

Three database admins walked into a NoSQL bar. A little while later they walked out because they could not find a table.

Типы данных

Смотрим на таблицу `Country`:

- `INTEGER` — целое число (размер варьируется от СУБД к СУБД, как и название).
- `REAL` — вещественное число с плавающей запятой.
- `VARCHAR(45)` — строка произвольной длины, но не длиннее 45 символов[1]. Если не влезает — поведение зависит от СУБД.

Вообще говоря, конкретно в SQLite значения в столбце могут иметь тип, не совпадающий со столбцом, но об этом лучше не думать.
За кадром остались типы:

- `BLOB` — бинарные данные любой длины.
- `TEXT` — строка произвольной длины без ограничений.
- `CHAR(10)` — строка фиксированной длины (может работать быстрее).

[1] А сколько символов занимает буква «ш»? А в какой кодировке?

Прочие типы данных

В каждой СУБД свои типы, они могут отличаться по поведению даже просто при разных настройках внутри одной базы данных. Но обычно они называются приблизительно так:

- `DECIMAL(10, 5)` — вещественное число с фиксированной запятой.
- Вариации на тему целых чисел: `SMALLINT`, `MEDIUMINT`, ...
- `FLOAT` — альтернатива `DOUBLE`.
- `DATE`, `TIME`, `DATETIME`, `TIMESTAMP` и вариации для хранения дат [2].

Мораль двух слайдов: сразу сказать, какой тип «правильный» в конкретной ситуации нельзя, надо хорошо понимать предметную область и СУБД, с которой вы работаете. Но для своих проектов по умолчанию можно ограничиться теми типам, что проще называются.

[2] Правильная работа с датами — тема отдельной лекции: 1, 2

Демонстрация SQL-запросов

- Запросы отделяются между собой точкой с запятой.
- Иногда интерфейс к СУБД не умеет делать несколько запросов одновременно и тогда точка с запятой не нужна.
- Комментарии — либо два дефиса в начале строки, либо `/* ... */`
- Результат запроса `SELECT` — тоже таблица, полученная из исходной.
- Наборы строк и столбцов в результате `SELECT` могут разительно отличаться от исходной.
- Можно фильтровать строки по условиям.
- Можно попросить не строки, а какую-то статистику.
- Если вы в SQL что-то написали, оно практически всегда либо упадёт при компиляции, либо как-то отработает на любых значениях.

LIMIT и OFFSET

- Порядок строк в таблицах и результате работы `SELECT` не определён.
- Но его можно явно задать при помощи `ORDER BY`, который сравнивает поля лексикографически в указанном порядке.
- Важно задавать `ORDER BY` так, чтобы он всегда отличал две строки. Этого можно добиться, только если мы знаем, какие наборы столбцов всегда отличаются.
- Другая стандартная проблема: пусть мы подгружаем бесконечную ленту новостей запросом `LIMIT 10 OFFSET already_loaded`.

LIMIT и OFFSET

- Порядок строк в таблицах и результате работы `SELECT` не определён.
- Но его можно явно задать при помощи `ORDER BY`, который сравнивает поля лексикографически в указанном порядке.
- Важно задавать `ORDER BY` так, чтобы он всегда отличал две строки. Этого можно добиться, только если мы знаем, какие наборы столбцов всегда отличаются.
- Другая стандартная проблема: пусть мы подгружаем бесконечную ленту новостей запросом `LIMIT 10 OFFSET already_loaded`.
- Новости в ленте могут добавляться и удаляться, и номера строк даже в идеально отсортированной таблице постоянно меняются.

Мораль: не стоит надеяться на номера строк, `LIMIT` и `OFFSET` обычно возникают только при выборке «топ-10».

Правильное решение задачи с лентой:

LIMIT и OFFSET

- Порядок строк в таблицах и результате работы `SELECT` не определён.
- Но его можно явно задать при помощи `ORDER BY`, который сравнивает поля лексикографически в указанном порядке.
- Важно задавать `ORDER BY` так, чтобы он всегда отличал две строки. Этого можно добиться, только если мы знаем, какие наборы столбцов всегда отличаются.
- Другая стандартная проблема: пусть мы подгружаем бесконечную ленту новостей запросом `LIMIT 10 OFFSET already_loaded`.
- Новости в ленте могут добавляться и удаляться, и номера строк даже в идеально отсортированной таблице постоянно меняются.

Мораль: не стоит надеяться на номера строк, `LIMIT` и `OFFSET` обычно возникают только при выборке «топ-10».

Правильное решение задачи с лентой:«верни топ-10 запросов после такой-то новости из ленты».

DELETE и INSERT

Удаление значений:

- `DELETE FROM Country` удалит **все** строки и не почешется.
- Надо писать `DELETE FROM Country WHERE ...`
- По-хорошему перед `DELETE` стоит сделать `SELECT` и посмотреть.

Добавление значений:

- После слова `VALUES` можно написать несколько кортежей со значениями, но надо знать точный порядок столбцов в таблице.
- На практике столбцы иногда (не часто, но иногда) меняются, добавляются и удаляются, поэтому всегда следует писать явно, каким столбцам что соответствует.
- При вставке может возникнуть ошибка, если на каком-то столбце были ограничения

Остальные запросы

- Также бывают запросы `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE` для работы с таблицами.
- Несмотря на наличие стандартов языка SQL, каждая база дополняет его по-своему, из-за чего получается множество несовместимых диалектов.
- Самые базовые команды (только что были) везде работают *примерно* одинаково (за исключением неоднозначных ситуаций).

Где и зачем

СУБД используются практически везде:

- Если данных или клиентов (которые запрашивают/меняют данные) будет очень много, то нам не надо изобретать велосипед и писать своё масштабируемое хранилище:
  1. Обычно одна СУБД обслуживает сразу несколько приложений.
  2. Можно создавать разных пользователей с разными правами.
  3. Можно прозрачно для приложений делать бэкапы или хранить данные на десяти серверах.

Где и зачем

СУБД используются практически везде:

- Если мы просто пишем приложение с какой-то нетривиальной схемой:
  1. Очень чётко отделяются данные от их обработки.
  2. SQL все знают (в отличие от логики программы), легко делать запросы к БД, зная только схему, и не зная ничего про приложение.
  3. SQL мощнее и читается лучше циклов for и list comprehension, которые ещё и не во всех языках есть.
  4. Не надо думать про хранение данных.

Где и зачем

СУБД используются практически везде:

- Если мы data scientist и/или хотим активно проверять гипотезы и много/просто работать с данными:
  1. Удобно, когда все данные лежат в БД с известным интерфейсом (SQL).
  2. Не надо писать никакой код и ни с чем интегрироваться, чтобы выполнить запрос.
  3. Не получится набагать в коде в обработке крайних случаев[3]

[3] Даже в SQL можно посадить сложный баг

В чём минусы

- Мы отдаём контроль за скоростью выполнения и потреблением памяти в руки СУБД (как и при любой абстракции). Это обычно приемлимый компромисс.
- Приложение сложнее запустить: нужно настроить СУБД, что обычно занимает несколько шагов. В нестадартных ситуациях — больше.
- Иногда приложение требует слишком хитрую настройку СУБД (например, для корректной работы с не-латиницей и датами).
- Многие инструменты заточены под промышленные решения и имеют слишком много рычажков и кнопок для простых целей.

Встраиваемые СУБД

- Самая известная встраиваемая СУБД — sqlite.
- Предназначена не для сетевого доступа, а для использования в рамках одной конкретной программы.
- Её можно просто вкомпилировать в своё приложение, не требуется никакой настройки.
- sqlite хранит каждую БД в отдельном файле определённого формата (последний — sqlite3).
- Формат sqlite3 один на все приложения, можно даже залезть в чужие БД и посмотреть.
- Занимает мало места в скомпилированном приложении.
- Используется во многих приложениях: под Android, в Firefox, в Chrome, в клиенте Dropbox[4]...

[4] ищите файлы `.db`, `.sqlite`, `.sqlite3`

Анонс домашнего задания

- Вам будет выдан файл с SQL-запросами, которые создают таблицы со странами (структуру разберём) и заполняют их данными.
- Вам нужно написать несколько SQL-запросов `SELECT`, которые что-то вычисляют.
- Тестировать можно на созданных тестовых данных.
- Как именно тестировать — сейчас покажу.

Консольная утилита

- Называется sqlite3. Это просто программа, которая умеет выполнять SQL-запросы на БД sqlite.
- По умолчанию создаёт пустую БД в памяти.
- Можно попросить открыть существующую БД в файле (или создать новый файл).
- При помощи перенаправления может выполнять SQL из файла.
- SQL-запрос должен заканчиваться точкой с запятой.

Графическая утилита

- Я выбрал DB Browser for SQLite.
- Иногда проще смотреть на таблице в графической оболочке, чем в консоли.
- Может открывать файлы с БД, все изменения идут в памяти.
- Можно откатывать изменения кнопкой «Revert Changes» до последнего сохранения.
- Можно сохранять изменения в файл кнопкой «Write Changes».
- Показывает таблицы, их структура, позволяет выполнять произвольные запросы.

Python

```
with sqlite3.Connection("literacy.sqlite3") as db:
  cursor = db.execute("SELECT * FROM Country LIMIT 3")
  print(cursor.description)
  print(list(cursor))
  print(list(cursor))  # Что-нибудь выведет?
```

- Терминология очень похожа во всех языках и СУБД.
- Обычно в языке есть стандартный интерфейс общения с любыми СУБД. А *драйвер* СУБД реализует этот интерфейс в языке.
- Сначала мы устанавливаем *соединение* с СУБД.
- Результатом запроса является *курсор* — это такой итератор по строчкам запроса.
- Что возвращают запросы, кроме `SELECT` — зависит от СУБД.
- Иногда считается, что не запрос возвращает курсор, а надо сначала создать курсор, а потом в нём выполнить запрос.

Упражнения

1. Скачайте файл literacy.sql.
2. Выберите имя для файла, где у вас будет лежать БД для тестов (например, `literacy.sqlite3`.
3. Выполните запросы из `literacy.sql`:

   ```
   sqlite3 literacy.sqlite3 < literacy.sql
   ```

   Каждый раз, когда вы будете выполнять команды из `literacy.sql`, таблицы будут полностью пересозданы. Не бойтесь что-то сломать.
4. Выполните SQL-запросы (либо в командной строке, либо в GUI):
   1. Всю информацию по всем странам.
   2. Первые десять стран (если сортировать по названию).
   3. Средние население и площадь страны.
   4. Площадь и население Франции.
   5. Количество французских территорий и их суммарную площадь и население.

NULL

Также есть специальное значение `NULL`, которое может лежать в любой колонке, если только на ней нет ограничения `NOT NULL`. Может обозначать:

- Отсутствие каких-либо данных (неизвестно население страны).
- Неопределённый результат вычисления (среднее значение пустого множества, деление на ноль).
- Что угодно ещё по желанию программиста.

Возникающая проблема: нет одного объяснения, как `NULL` себя ведёт в разных запросах.

- Если считать, что `NULL` распространяется как `NaN` (Not a Number), т.е. любое вычисление с `NULL` даёт `NULL`, то сложно писать запросы в ситуации, где наличие `NULL` — норма.
- Если `NULL` просто игнорировать, то про него легко забыть (так часто и делают); а он может где-то требовать специальной обработки.

Демонстрация NULL

- Разные функции обрабатывает `NULL` по-разному, общая цель — наиболее консистентное и адекватное поведение.
- Обычно в агрегирующих функциях игнорируется.
- Если вы пишете чуть-чуть несимметричный код (вроде `SUM(a) / COUNT(*)`), могут быть последствия.
- Очень легко забыть и получить какой-то правдоподобный, но неверный результат (особенно в соединениях — будут дальше).

SQL-инъекции

Пусть есть таблица с полями: владелец текста, его название, содержимое. Код для доступа к базе, выполняется на сервере:

```
with sqlite3.Connection("sql-injection.sqlite3") as db:
  key = input('Text key: ')
  cursor = db.execute("""SELECT * FROM Text
                         WHERE owner='user' AND key='{}'"""
                         .format(key))
  print(list(cursor))
```

В чём проблема?

SQL-инъекции

Пусть есть таблица с полями: владелец текста, его название, содержимое. Код для доступа к базе, выполняется на сервере:

```
with sqlite3.Connection("sql-injection.sqlite3") as db:
  key = input('Text key: ')
  cursor = db.execute("""SELECT * FROM Text
                         WHERE owner='user' AND key='{}'"""
                         .format(key))
  print(list(cursor))
```

В чём проблема?

Значение:	key1
Было:	SELECT ... WHERE owner='user' AND key='{}'
Стало:	SELECT ... WHERE owner='user' AND key='key1'

SQL-инъекции

Пусть есть таблица с полями: владелец текста, его название, содержимое. Код для доступа к базе, выполняется на сервере:

```
with sqlite3.Connection("sql-injection.sqlite3") as db:
  key = input('Text key: ')
  cursor = db.execute("""SELECT * FROM Text
                         WHERE owner='user' AND key='{}'"""
                         .format(key))
  print(list(cursor))
```

К коллайдеру! Упс.

SQL-инъекции

Пусть есть таблица с полями: владелец текста, его название, содержимое. Код для доступа к базе, выполняется на сервере:

```
with sqlite3.Connection("sql-injection.sqlite3") as db:
  key = input('Text key: ')
  cursor = db.execute("""SELECT * FROM Text
                         WHERE owner='user' AND key='{}'"""
                         .format(key))
  print(list(cursor))
```

К коллайдеру!

Значение:	' OR ''='
Было:	SELECT ... WHERE owner='user' AND key='{}'
Стало:	SELECT ... WHERE owner='user' AND key='' OR ''=''

Упс.

Классический комикс

А как правильно?

```
with sqlite3.Connection("sql-injection.sqlite3") as db:
  key = input('Text key: ')
  cursor = db.execute("SELECT * FROM Text WHERE owner='user' AN
  print(list(cursor))
```

Теперь драйвер базы данных знает, что `key` — это значение от пользователя, которое надо *заэкранировать*:

Значение:	`' OR ''='`
Было:	`... WHERE owner='user' AND key=?`
Стало:	`... WHERE owner='user' AND key='\' OR \'\'=\''`

Независимо от того, какой код мы напишем, SQL-инъекции не случится.

Мораль: никогда не собирайте SQL-запрос руками из переменных.

GROUP BY

- Полезно, когда мы хотим посчитать какую-то статистику по подмножествам строк.
- Каждая агрегирующая функция работает только внутри группы.
- Например, суммарное население стран с разным политическим строем.
- Можно группировать по нескольким полям.
- Условие `WHERE` применяется до группировки.
- `ORDER BY` применяется после (очевидно, так как группировка от порядка не зависит).
- В `SELECT` можно использовать только агрегирующие функции и колонки, по которым сделана группировка (иначе неясно, из какой строки выбирать). Некоторые СУБД ругаются, некоторые делают что-то.
- Можно дополнительно отфильтровать результаты после группировки при помощи `HAVING`.

Упражнения

Посчитайте:

1. Среднюю площадь страны в зависимости от формы правления.
2. Средний корень площади страны в зависимости от формы правления.
3. Количество стран с площадью порядка миллиона, порядка двух миллионов, и так далее.

Подзапросы-1

1.
   - Задача: хотим для города найти население страны, в которой он расположен.
   - В таблице городов есть только код страны, без населения.
   - Можно сделать подзапрос в условии `WHERE`.
2.
   - Задача: хотим найти средний уровень самой грамотной страны по годам.
   - Надо две агрегатных функции: сначала группируем по годам (чтобы найти победителя), а потом берём среднее.
   - Можно сделать подзапрос в `FROM` (ведь результат SELECT — тоже таблица, которую можно назвать).

Подзапросы-2

- Задача: хотим вывести информацию по странам плюс население самого большого города.
- Эта информация лежит в двух разных таблицах.
- SQL позволяет делать `SELECT FROM` сразу из нескольких таблиц (получается декартово произведение).
- Можно взять декартово произведение городов и стран, оставить только соответствующие, а по оставшимся взять агрегирующую функцию.
- Если имена колонок в разных таблицах совпадают, надо явно указывать, к какой мы обращаемся.
- Вообще лучше всегда явно указывать, из какой таблицы мы берём колонку, если таблиц несколько.

Такого сорта выборки происходят очень часто, в реляционной алгебре они зовутся «соединениями» (join).

Соединения

- Можно считать синтаксическим сахаром для взятия подмножества декартова произведения.
- Лучше отражает суть происходящего и проще читается.
- После `ON` может быть произвольное условие (это круче, чем в реляционной алгебре).
- Обычно там ставят условие «номер страны в первой таблице равен номеру страны во второй таблице».

Ключи и соединения-1

- Обычно сущностей в базе много и они как-то связны отношениями («каждый город лежит ровно в одной стране»).
- Не хочется дублировать информацию в разных таблицах (место занимает, изменять сложно).
- Поэтому информация о стране/городе отдельно.
- А запрос «получи объект по вот этому отношению» возникает.
- Так как свойства объектов часто меняются, то обычно каждому объекту выдают *первичный ключ*, по которому его можно опознать. Обычно это просто какое-то число (возможно, с автоинкрементом).

Ключи и соединения-2

- Тогда связи вроде «в какой стране лежит город» — это просто столбец «номер страны» в таблице с городом.
- Такой столбец называют *внешним ключом*.
- Это даже можно отразить в структуре таблицы (`REFERENCES`).
- И ещё можно указать, что делать при удалении того объекта, куда мы ссылаемся (`ON DELETE CASCADE`).

Упражнения

1. Вывести для каждой страны максимальный уровень её грамотности за все года.
2. Вывести для каждой страны номер города-столицы (см. таблицу `Capital`).
3. Вывести для каждой страны название города-столицы (потребуется два `JOIN`).

Соединения, NULL, отсутствие значений

1. Посчитаем количество стран (239).
2. Теперь для каждой страны посчитаем количество городов.
3. Теперь посмотрим, сколько строк получили в результате — 232.

- А всё потому что есть страны, в которых городов нет — про них в таблице `City` просто нет информации.
- И соединение не поможет — страна без городов отфильтруется.
- Но есть `LEFT (OUTER) JOIN` — он обязуется добавить в соединение все строчки из «левой» таблицы (а если не нашлось соответствующих строк, то поля второй в строчке соединения будут `NULL`).
- С ним надо быть осторожным — потому что теперь в строчках соединения могут оказаться `NULL`, которые, скорее всего, не надо учитывать (`COUNT(*)`, например, их учтёт).
- Ещё бывают аналогичные `RIGHT JOIN` и `FULL JOIN` (SQLite их не поддерживает).

Классическая шутка

UNION

- Хотим вывести названия всех географических объектов из БД.
- Делаем два (или больше) `SELECT` с одинаковым количеством столбцов в результате.
- Если объединяем их через `UNION` — то в результате не будет одинаковых строк вообще (даже если они были внутри одного `SELECT`).
- Если объединяем их через `UNION ALL` — то просто все результаты объединятся вместе.

На практике:

- Я ни разу не встречал.
- Может потребоваться, если в БД есть похожие таблицы (например, `UsersUS` и `UsersEU`).