

# БАЗЫ ДАННЫХ. РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ. SQL

## Оглавление

1 Типы баз данных. Реляционные БД.....	2
1.1 Реляционные базы данных .....	2
1.2 Типы данных в таблицах.....	3
1.3 Relation -> связь .....	3
1.4 Нереляционные базы данных (NoSQL).....	5
2 Основные команды в SQL. CRUD.....	5
2.1 Создание таблицы и добавление записей: CREATE, INSERT .....	6
2.2 Получение данных: SELECT, FROM, WHERE, DISTINCT. Значение NULL.....	7
2.3 Сортировка, ограничение и сдвиг выборки: ORDER BY, LIMIT, OFFSET.....	9
3 Агрегирующие функции, группировка, фильтрация .....	12
3.1 Агрегирующие функции COUNT, MIN, MAX, AVG, SUM.....	12
3.2 Группировка GROUP BY и фильтрация HAVING.....	14
4 Отношение между таблицами .....	17
4.1 Связь «один-к-одному» (1:1).....	19
4.2 Связь «один-ко-многим» (1:M) .....	22
4.3 Связь «многие-ко-многим» (M:M).....	23
5 Объединение таблиц: JOIN.....	26
5.1 INNER JOIN .....	27
5.1 LEFT OUTER JOIN.....	28
5.3 RIGHT OUTER JOIN .....	28
5.4 FULL OUTER JOIN .....	29
5.5 CROSS JOIN.....	30

## 1 Типы баз данных. Реляционные БД.

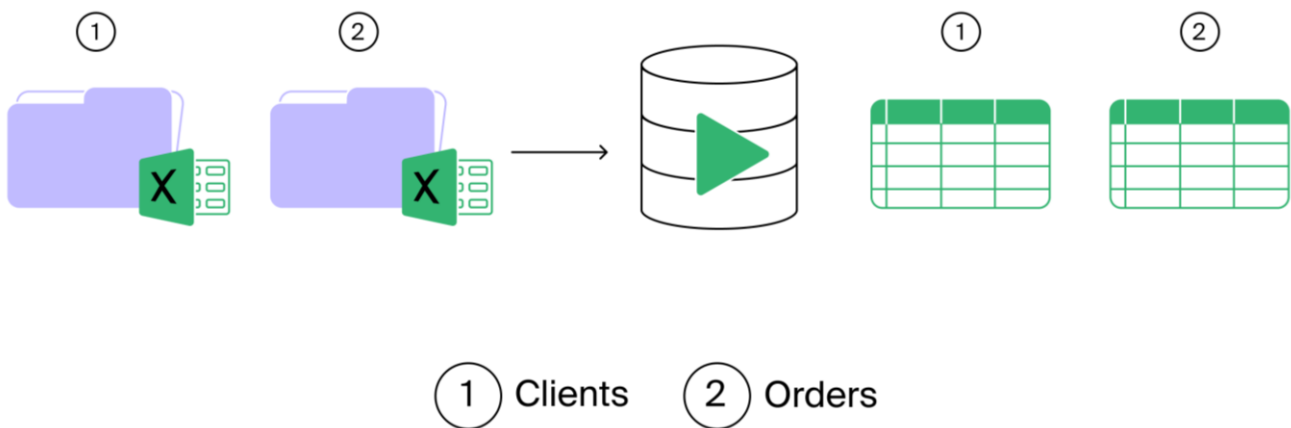
Работа с данными — это большая часть работы с любым проектом. Информация о пользователях, описания и цены на товары и географические координаты и многое другое — всё это данные, которые необходимо сохранять, обрабатывать и предоставлять пользователям.

Для хранения информации придумали **базы данных** — системы для хранения информации и доступа к ней.

По способу хранения информации базы данных разделяют на две большие группы: реляционные и нереляционные.

### 1.1 Реляционные базы данных

Реляционную базу данных можно представить в виде **набора таблиц с информацией**, хранящихся в одном пространстве — вроде excel-таблиц, хранящихся в одной папке.



Каждая строка таблицы в БД — это отдельная запись о каком-то объекте. В полях записи (в ячейках строки) могут храниться различные характеристики этого объекта. Каждый столбец таблицы должен иметь уникальное имя.

Вот простая таблица **movies**, в ней собрана информация о нескольких фильмах. Столбцам принято давать имена в единственном числе, предельно короткие и содержательные, точно как переменным в Python.

**movies**

name	type	release_year
Весёлые Мелодии	Мультсериал	1930
Кто подставил кролика Роджера	Фильм	1988
Безумные Мелодии Луни Тюнз	Мультсериал	1931
Розовая пантера: Контроль за вредителями	Мультсериал	1969

Элементы таблицы в реляционных базах данных принято называть так:

- **Запись (строка)** — это строка в таблице. Таблица **movies** содержит четыре строки/четыре записи.
- **Поле (столбец)** — это столбец таблицы или ячейка одной записи. Например, в **столбце** `type` хранится тип фильма. Или так в **поле** `release_year` хранится дата выхода фильма.



## 1.2 Типы данных в таблицах

В столбце таблицы реляционной БД хранятся данные одного типа. Тип данных задаётся при создании таблицы. Например, колонка `release_year` в приведённом примере хранит данные типа «число», и при попытке записать в это поле значение другого типа возникнет ошибка.

Самые востребованные типы полей:

- **Целое число**, как `int` в Python.
- **Число с плавающей точкой**, как `float` в Python.
- **Строка**, как `str` в Python (обычно применяется для коротких строк: названий, заголовков).
- **Булев тип**: логический тип данных. `True` или `False`.
- **Дата и время**. В базах данных обычно есть несколько специализированных типов для хранения дат, времени или временных интервалов.

Помимо базовых типов могут применяться и специальные, например — для хранения картографической информации, работы с деньгами, адресов компьютеров в сети или документов в формате JSON или XML.

Не во всех базах данных поддерживается полный набор типов данных. Например, **SQLite** поддерживает только пять из них:

- **INTEGER** — целое число;
- **REAL** — дробное число;
- **TEXT** — текст;
- **BLOB** — двоичные данные;
- **NULL** — специальное значение, означающее отсутствие информации.

Поэтому при работе с SQLite приходится идти на всякие хитрости.

## 1.3 Relation -> связь

Реляционная база данных может содержать множество таблиц с различными данными; эти данные могут быть связаны между собой.

Вернёмся к таблице с фильмами: у каждого фильма есть режиссёр (а то и несколько), и о каждом режиссёре в БД можно хранить какую-то дополнительную информацию.

Для примера создадим таблицу, где будет имя режиссёра и год его рождения:

## directors

id	name	birthday_year
1	Текс Эйвери	1908
2	Роберт Земекис	1952
3	Джерри Чиникей	1912

Каждый из этих режиссёров снимал какой-то фильм из таблицы **movies**, название фильма должно быть связано с его именем. Для этого в каждой записи таблицы **movies** можно указать, с каким режиссёром из таблицы **directors** связана эта запись.

Для организации связей между таблицами нужно твёрдо знать, какая запись на какую ссылается. Для точной «адресации» у каждой записи в каждой таблице должен быть уникальный идентификатор, **первичный ключ (primary key, PK)**. Этот ключ должен быть уникален в пределах таблицы; в таблице **directors** PK режиссёров хранятся в колонке *id*.

В таблице, которая ссылается на другую таблицу, создают поле с атрибутом **внешний ключ** или **FK (англ. foreign key)**; в этом поле указывается, на какой **PK** в другой таблице ссылается это поле.

Поэтому давайте в таблице **movies** создадим дополнительную колонку *director\_id* — это будет **Foreign Key**, колонка, ссылающаяся на таблицу **directors**; в этой колонке в каждой строке укажем идентификатор того режиссёра, который снял фильм.

Заодно добавим в таблицу **movies** и колонку *id*, в ней будут храниться первичные ключи (уникальные идентификаторы) записей этой таблицы. Пригодятся.

## movies

id	name	type	release_year	director_id
1	Весёлые Мелодии	Мультсериал	1930	1
2	Кто подставил кролика Роджера	Фильм	1988	2
3	Безумные Мелодии Луни Тюнз	Мультсериал	1931	1
4	Розовая пантера: Контроль за вредителями	Мультсериал	1969	3

Теперь можно получить из таблицы **movies** запись о каком-нибудь фильме, и, взяв значение из поля *director\_id*, обратиться к нужной записи в таблице **directors**, получив из неё данные о режиссёре этого фильма. Связи могут быть гораздо сложнее описанного примера, но общий принцип остаётся неизменным.

В Django-проектах часто используются такие реляционные БД:

- [MySQL](#).
- [PostgreSQL](#).

- [SQLite](#).

В этом курсе мы будем работать исключительно с реляционными базами данных. Начнём работу с **SQLite**, а позже поработаем с **PostgreSQL**.

## 1.4 Нереляционные базы данных (NoSQL)

Реляционные БД требуют чёткой структуры. А сама информация, которую надо хранить может быть очень разнородной. Для работы с большими объёмами разнородной информации были придуманы **нереляционные базы данных**. Такие базы данных ещё называют **NoSQL** (*Not Only SQL*).

Пионером в масштабном использовании нереляционных баз данных была компания Google. В начале 2000-х годов с их помощью решались проблемы масштабируемости и параллельной обработки данных в поисковой системе и в приложениях GMail, Maps, Earth.

Существует несколько типов NoSQL базы данных:

- **Ключ-значение** (*Redis, Memcached, Amazon DynamoDB*). Базу данных этого типа можно представить в виде таблицы, в каждой ячейке которой хранятся значения произвольного типа. Главное преимущество таких баз — поиск по ключу и быстрое получение данных. Обычно используются для кэшей объектов.
- **Документо-ориентированные** (*MongoDB, CouchDB, Amazon DocumentDB*). Похожи на БД типа ключ-значение, но единица хранения — «документ»; в документ можно записать любой набор данных. Такие базы часто используются для хранения информации о товарах с разнородными характеристиками.
- **Столбцовые (или колоночные)** (*BigTable, HyperTable, Apache HBase, Apache Cassandra*). В таких БД данные хранятся не в строчках, а в столбцах. Это позволяет выполнять быстрый поиск по базе. Область применения — большой объём данных, аналитика.
- **Графовые** (*neo4j, AllegroGraph, ActiveRDF*). Информация в таких БД представлена в виде узлов и отношений между узлами. Графовые БД позволяют эффективно работать с большими объёмами связанных данных; используются, например, для создания соцсетей.

## 2 Основные команды в SQL. CRUD

База данных чем-то похожа на библиотеку: информация упорядочена, структурирована и разложена по полочкам. Но нужен специальный инструмент, чтобы управлять данными: что толку хранить информацию, если ей нельзя воспользоваться.

Роль библиотекаря в базах данных выполняет **система управления базами данных, СУБД**. Стоит дать ей команду, и она:

- создаст базу или таблицу в базе;
- внесёт новые данные или удалит ненужные;
- получит из базы нужную информацию по заданному условию;
- обеспечит безопасный доступ к данным.

Основной инструмент для работы с реляционными базами данных — специальный язык запросов: **Structured Query Language, SQL**.

У SQL своеобразный синтаксис: в нём описываются условия **запросов-команд**: «база данных, сделай то-то и то-то с такими-то условиями».

Вот пример команды на языке SQL:

```
-- Верни названия (значения поля name) тех фильмов из таблицы movies,  
-- у которых в поле type записано «Фильм», а дата выпуска — ранее 1990 года  
SELECT name FROM movies WHERE type = 'Фильм' AND release year < 1990;
```

На которую получим ответ:

	name
1	Кто подставил кролика Роджера
2	Хороший, плохой, злой

Язык SQL — это стандарт, который применяется в реляционных базах данных. В разных СУБД могут применяться дополнительные команды SQL, расширяющие стандартные возможности языка. Такие расширенные версии называют «диалектами» SQL. В нашем курсе мы будем говорить на «классическом SQL» — применять команды, которые универсальны для всех диалектов SQL.

Есть такое понятие **CRUD** - акроним, обозначающий четыре базовые функции, используемые при работе с базами данных:

- **create** — создание — CREATE/INSERT;
- **read** — чтение — SELECT;
- **update** — обновление записи — UPDATE;
- **delete** — удаление записи — DELETE.

Пройдемся по каждой функции подробнее.

## 2.1 Создание таблицы и добавление записей: CREATE, INSERT

Для создания таблицы применяется оператор **CREATE TABLE**, после которого указывается название таблицы. Далее в скобках через запятую перечисляются названия полей с их свойствами (тип поля и дополнительные свойства). Свойства одного поля отделяются просто пробелами. Давайте создадим таблицу **movies**:

```
CREATE TABLE IF NOT EXISTS movies
(
    id            INTEGER PRIMARY KEY,
    name          TEXT,
    type          TEXT,
    release_year  INTEGER
);
```

В базе данных не может быть двух таблиц с одинаковым названием. Поэтому перед созданием таблицы нужно сделать проверку **IF NOT EXISTS**. В результате новая таблица будет создана только в том случае, если таблицы с таким именем в базе нет. Если убрать из кода проверку **IF NOT EXISTS**, то в первый раз всё сработает, как надо — в базе появятся таблица **movies**. А вот при втором запуске программа упадёт с сообщением «*OperationalError: table movies already exists*».

Для добавления записи в таблицу применяется конструкция:

```
INSERT INTO <таблица> (<перечень полей>), VALUES (<перечень значений>)
```

Давайте добавим одну запись в таблицу **movies**:

```
INSERT INTO movies(id, name, type, release_year)
VALUES (1, 'Весёлые мелодии', 'Мультсериал', 1930);
```

Можно добавить сразу несколько записей в таблицу:

```
INSERT INTO movies(id, name, type, release_year)
VALUES (2, 'Безумные Мелодии Луни Тюнз', 'Мультсериал', '1931'),
(3, 'Кто подставил кролика Роджера', 'Фильм', '1988'),
(4, 'Хороший, плохой, злой', 'Фильм', '1967'),
(5, 'Последний киногерой', 'Фильм', '1993'),
(6, 'Она написала убийство', 'Сериал', '1984'),
(7, 'Лас-Вегас', 'Сериал', '2003'),
(8, 'Паркер Люис не проигрывает', 'Сериал', '1990'),
(9, 'Розовая пантера: Контроль за вредителями', 'Мультфильм', '1969'),
(10, 'Койот против Аспе', 'Фильм', '2023');
```

В итоге таблица **movies** будет иметь следующий вид и наполнение:

	id	name	type	release_year
1	1	Весёлые мелодии	Мультсериал	1930
2	2	Безумные Мелодии Луни Тюнз	Мультсериал	1931
3	3	Кто подставил кролика Роджера	Фильм	1988
4	4	Хороший, плохой, злой	Фильм	1967
5	5	Последний киногерой	Фильм	1993
6	6	Она написала убийство	Сериал	1984
7	7	Лас-Вегас	Сериал	2003
8	8	Паркер Люис не проигрывает	Сериал	1990
9	9	Розовая пантера: Контроль за вредителями	Мультфильм	1969
10	10	Койот против Аспе	Фильм	2023

Немного про стиль оформления SQL запросов:

1. Операторы SQL пишутся в верхнем регистре: это позволяет визуально отделить их от ключевых слов.
2. Каждый оператор (а их в запросе может быть много) пишется с новой строки.

Пример создания таблицы **movies** без форматирования:

```
create table if not exists movies (id integer primary key , name text, type text,
release_year integer);
```

Этот запрос будет выполнен и создаст таблицу с нужными полями (SQL – регистр независимый), но прочесть такой код — сложная задача: а если будет 20 полей и у каждого поля по несколько дополнительных свойств.

## 2.2 Получение данных: SELECT, FROM, WHERE, DISTINCT. Значение NULL

Мы создали таблицу и добавили записи, настало время их получить из БД. Для этого после оператора **SELECT** указывают названия полей, значения которых должны вернуться в ответе; после **FROM** — названия таблиц, в которых надо искать данные.

**SELECT** <перечень столбцов, значения из которых надо получить>

**FROM** <перечень таблиц, из которых надо получить данные>;

В ответ на запрос **SELECT** СУБД возвращает данные в структурированном табличном виде. Чтобы не путаться в понятиях «таблица БД» и «таблица с ответом», возвращаемые данные будем называть «**результатирующая выборка**» или просто «**выборка**».

В ответ на запрос:

```
-- Вернуть только поля name, release_year
SELECT name,
        release_year
-- из всех записей в таблице movies
FROM movies;
```

SQL вернёт такую результирующую выборку:

	name	release_year
1	Весёлые мелодии	1930
2	Безумные Мелодии Луни Тюнз	1931
3	Кто подставил кролика Роджера	1988
4	Хороший, плохой, злой	1967
5	Последний киногерой	1993
6	Она написала убийство	1984
7	Лас-Вегас	2003
8	Паркер Люис не проигрывает	1990
9	Розовая пантера: Контроль за вредителями	1969
10	Койот против Аспе	2023

Обычно при запросах к БД не нужны все поля записей. Например, на какой-то странице сайта нужно показать только названия и год выпуска для всех фильмов из базы. В такой ситуации нужно запросить из базы не все поля, а лишь два из них; для этого названия нужных столбцов перечисляют через запятую после **SELECT**.

### Фильтрация по строкам

В запросе можно отфильтровать записи, удовлетворяющие заданным условиям. Найдём все фильмы, снятые после 1980 года. Это можно сделать при помощи оператора **WHERE**.

```
SELECT name,  
       release_year  
FROM movies  
-- ГДЕ значение поля release_year больше 1980  
WHERE release_year > 1980;
```

Получим:

	name	release_year
1	Кто подставил кролика Роджера	1988
2	Последний киногерой	1993
3	Она написала убийство	1984
4	Лас-Вегас	2003
5	Паркер Люис не проигрывает	1990
6	Койот против Асте	2023

В результирующую выборку попадут только те записи, которые соответствуют условиям, описанным после оператора **WHERE**.

Для описания условий запроса применяют специальные операторы. Вот несколько популярных:

- «=» — оператор сравнения, а не присваивания (это аналог «==» в Python)
- «>», «<» — больше и меньше, «>=», «<=» — больше или равно, меньше или равно.
- «<>» или «!=» — «не равно».
- BETWEEN** <начало\_диапазона> **AND** <конец\_диапазона> - «между», для проверки значения в диапазоне:

```
-- Найдём фильмы, выпущенные с 1980 по 1990 год включительно:  
WHERE release_year BETWEEN 1980 AND 1990;
```

- IN** — вхождение в список:

```
-- Найдём в базе все фильмы,  
-- у которых значение поля type - 'Сериал' или 'Фильм':  
WHERE type IN ('Сериал', 'Фильм');
```

- LIKE** — поиск строки по шаблону; в шаблонах можно применять символы-«маски»: знак процента «%» заменяет любой набор символов; символ подчёркивания «\_» заменяет один любой символ (цифру, букву, пробел, пунктуационный или любой другой символ).

```
WHERE type LIKE 'Мульт%';
```

Этот запрос выберет все записи, где значение поля **type** — **Мультфильмы** или **Мультсериалы**. Символы «\_» и «%» можно комбинировать в одном шаблоне.

В разных СУБД могут быть различные наборы текстовых масок и дополнительные функции для преобразования строк и поиска с учётом строения языка.

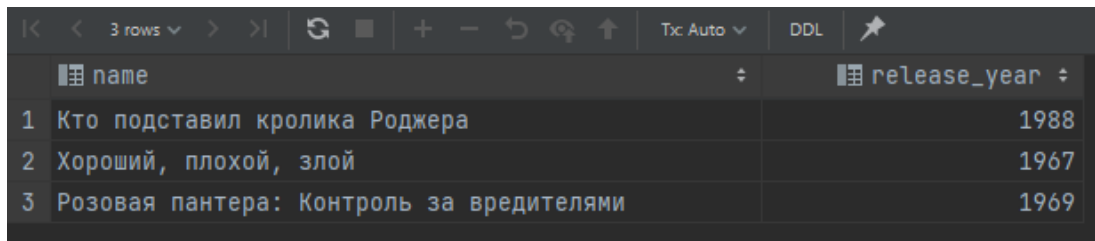
Оператор **WHERE** позволяет использовать множество условий одновременно. Для объединения сразу нескольких условий используются операторы **AND**, **OR** или **NOT**, они работают так же, как в Python.

```
SELECT name,  
       release_year
```



```
FROM movies
WHERE (release_year BETWEEN 1965 AND 1990)
AND type LIKE '%фильм';
```

Получим:



	name	release_year
1	Кто подставил кролика Роджера	1988
2	Хороший, плохой, злой	1967
3	Розовая пантера: Контроль за вредителями	1969

### Специальное значение NULL

Значение **NULL** означает отсутствие данных. Основная особенность **NULL** в том, что это значение не равно ничему; любая попытка сравнить **NULL** с чем угодно, даже с **NULL**, вернёт **False**.

Для работы со специальными значениями используют отдельные операторы — **IS NULL** и **IS NOT NULL**. Оператор **IS NULL** при встрече с **NULL** вернёт **True**, а оператор **IS NOT NULL** наоборот: вернёт **True**, если значение не равно **NULL**.

Операторы **IS NULL** и **IS NOT NULL** можно применять в условиях, чтобы фильтровать данные.

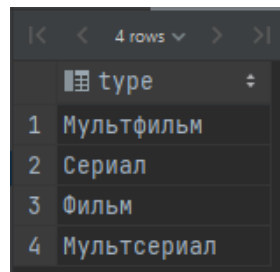
### Оператор DISTINCT

Этот оператор используется, чтобы получить уникальные значения из заданного столбца.

Выберем уникальные значения из колонки **type** в таблице **movies**:

```
SELECT DISTINCT type
FROM movies;
```

Получим перечень типов фильмов, без повторов:



	type
1	Мультфильм
2	Сериал
3	Фильм
4	Мультсериал

### Порядок операторов

В SQL-запросах важно указывать операторы в правильном порядке. Логика работы такая:

1. Выбрали столбцы из таблицы **SELECT FROM**.
2. В них выбрали значения, которые соответствуют условиям **WHERE**.

**SELECT** <перечень столбцов через запятую>

**FROM** <название таблицы>

**WHERE** <условия>;

SQL строг, и неправильный порядок операторов вызовет ошибку.

## 2.3 Сортировка, ограничение и сдвиг выборки: ORDER BY, LIMIT, OFFSET

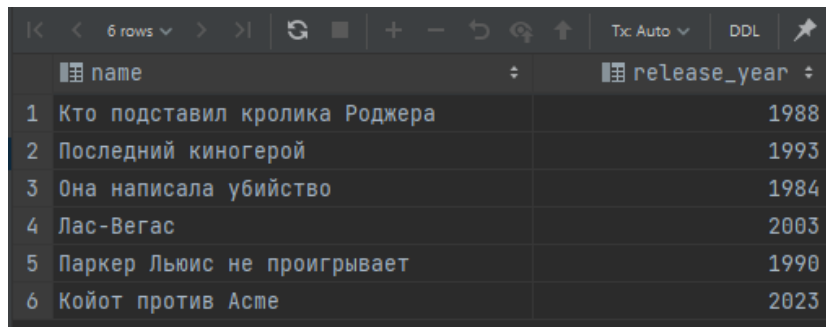
Зачастую требуется не просто получить весь набор найденных записей или значений полей, но и:

- отсортировать выборку по какому-то принципу;
- ограничить выборку по количеству строк.

Если мы запросим, например, список фильмов, выпущенных после 1980 года:

```
SELECT name,  
       release_year  
FROM movies  
WHERE release_year > 1980;
```

То получим отфильтрованную выборку, но в хаотичном порядке:



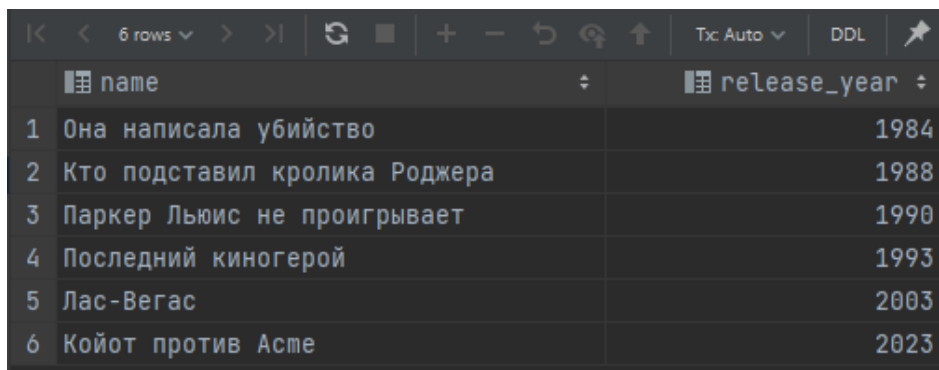
	name	release_year
1	Кто подставил кролика Роджера	1988
2	Последний киногерой	1993
3	Она написала убийство	1984
4	Лас-Вегас	2003
5	Паркер Люис не проигрывает	1990
6	Койот против Аспе	2023

### Сортировка данных: ORDER BY

Для сортировки результатов запроса в SQL есть оператор **ORDER BY**. Обратите внимание: оператор сортировки пишется после условия **WHERE**. Это логично — ведь сначала нужно выбрать данные, и только после этого выстраивать их по порядку.

```
SELECT name,  
       release_year  
FROM movies  
WHERE release_year > 1980  
ORDER BY release_year;
```

Получим отсортированный по дате выхода список фильмов (по возрастанию):



	name	release_year
1	Она написала убийство	1984
2	Кто подставил кролика Роджера	1988
3	Паркер Люис не проигрывает	1990
4	Последний киногерой	1993
5	Лас-Вегас	2003
6	Койот против Аспе	2023

### Направление сортировки

По умолчанию, команда **ORDER BY** выполняет сортировку по возрастанию. Чтобы управлять направлением сортировки вручную, после имени столбца указывается ключевое слово **ASC** («*ascending*» - по возрастанию) или **DESC** («*descending*» - по убыванию).

Вывести фильмы в порядке по убыванию (от новых к старым) можно так:

```
...  
ORDER BY release_year DESC;
```

А оба этих запроса — равнозначны:

```
...  
ORDER BY release_year ASC;
```

```
...  
ORDER BY release_year;
```

Результирующую выборку можно отсортировать и по текстовым полям, тогда результаты будут выведены в алфавитном порядке. Сортировка с ключевым словом **DESC** «развернёт» порядок, и записи будут выведены в порядке «от Я до А».

При сортировке можно указать несколько полей, перечислив их через запятую. Сначала результаты отсортируются по первому полю, а потом — по второму.

Отсортируем выборку по полю **type** в обратном порядке (от Я к А), а фильмы с одинаковым типом отсортируем по алфавиту:

```
SELECT type,
       name
FROM movies
WHERE release_year > 1980
ORDER BY type DESC, name;
```

Сначала сортируется тип картины в обратном алфавитном порядке (от Я к А), а внутри **type** — алфавитная сортировка по названиям.

	type	name
1	Фильм	Койот против Аспе
2	Фильм	Кто подставил кролика Роджера
3	Фильм	Последний киногерой
4	Сериал	Лас-Вегас
5	Сериал	Она написала убийство
6	Сериал	Паркер Люис не проигрывает

Сортировка очень часто применяется на практике. В любом современном интернет-магазине, на новостном сайте или в блоге всегда есть сортировка по цене, рейтингу или дате публикации материала.

### Ограничение и сдвиг выборки: LIMIT и OFFSET

В ответ на запрос может вернуться большая выборка; но зачастую нужно получить лишь несколько записей из неё. Самый простой пример — постраничное отображение информации на сайте; допустим, на каждую страницу нужно вывести по два фильма из нашей базы данных. При запросе нужно получить только две записи, а не полный их список.



Для ограничения количества строк, возвращаемых в ответ на SQL-запрос, применяют оператор **LIMIT** (англ. «ограничение»); и **OFFSET** (англ. «смещение»).

Ограничим количество возвращаемых записей; выводить будем по алфавиту:

```
SELECT name,
       type
FROM movies
ORDER BY name
LIMIT 2;
```

Получим:

	name	type
1	Безумные Мелодии Луни Тюнз	Мультсериал
2	Весёлые мелодии	Мультсериал

При постраничном размещении информации эти две записи можно выкладывать на первую страницу. На вторую страницу нужно вывести третью и четвёртую записи.

Если нужно вернуть определённое число записей, но начинать нужно не с первой записи в выборке — после **LIMIT** ставят оператор **OFFSET** и в нём указывают, сколько записей нужно пропустить.

Вернём две записи из нашей выборки, но две первые пропустим:

```
SELECT type,
       name
FROM movies
ORDER BY name
LIMIT 2 OFFSET 2;
```

В результате вернулись третья и четвёртая записи из отсортированного списка фильмов:

	type	name
1	Фильм	Койот против Аспе
2	Фильм	Кто подставил кролика Роджера

Переведем последний запрос с SQL на русский:

1. возьми значения полей type и name из таблицы movies
2. отсортируй результаты по полю name в алфавитном порядке,
3. покажи две строки, но пропусти две и начни с третьей.

Использование ограничений **LIMIT** и **OFFSET** без явной сортировки **ORDER BY** не вызовет ошибки, но может привести к непредсказуемым результатам. Ограничивать выборку лучше после сортировки.

Порядок операторов важен. Нельзя сортировать значения и ограничивать выборку без того, чтобы сначала не указать таблицу и столбцы, которые должны попасть в выборку.

Порядок должен быть таким:

- SELECT
- FROM
- WHERE
- ORDER BY
- LIMIT OFFSET

Если не держать перед глазами таблицу — решить такую задачу будет трудновато. При запросах без **ORDER BY** в результирующей выборке записи будут расположены в порядке их размещения в БД. Предвидеть этот порядок почти невозможно, поэтому сортировка — залог предсказуемости при получении выборки.

## 3 Агрегирующие функции, группировка, фильтрация

### 3.1 Агрегирующие функции COUNT, MIN, MAX, AVG, SUM

Отсортированная и ограниченная выборка — это удобно, но при работе с базой данных всегда хочется большего. В корзине интернет-магазина нужно посчитать сумму всех товаров, а для аналитика в ресторане важно узнать, как менялось число заказов и динамика среднего чека.

Для всех этих целей в SQL существуют **агрегирующие** или **агрегатные** (от англ. *aggregate*, «сгруппированный, совокупный») функции. Такие функции выполняют вычисления на наборе значений и возвращают не набор данных, а одиночное значение — результат вычислений.

В общем виде запрос с агрегирующей функцией выглядит так:

```
SELECT АГРЕГИРУЮЩАЯ_ФУНКЦИЯ(поле)
```

```
FROM Таблица;
```

Добавим в таблицу с фильмами столбец **gross**: в нём записаны кассовые сборы фильмов в долларах (по данным Кинопоиска). Для некоторых картин информации о сборах нет, поэтому в графе стоит **NULL** (в Python есть эквивалент **NULL - None**).

Выполним следующие команды:

```
ALTER TABLE movies ADD COLUMN gross INTEGER;
UPDATE movies SET gross=156452370 WHERE id=3;
UPDATE movies SET gross=25118063 WHERE id=4;
UPDATE movies SET gross=137298489 WHERE id=5;
```

Переведем с языка SQL на русский то, что мы сделали:

1. Изменить таблицу **movies**, а именно: добавить поле **gross** типа **INTEGER** (целое число);
2. Обновить таблицу **movies**, установив значение *156452370* в поле **gross** для записи с id=3;
3. Следующие две команды аналогичны 2 пункту.

Посмотрим, что по итогу стало с таблицей **movies**:

```
SELECT *
FROM movies
ORDER BY id;
```

id	name	type	release_year	gross
1	Весёлые мелодии	Мультсериал	1930	<null>
2	Безумные Мелодии Луни Тюнз	Мультсериал	1931	<null>
3	Кто подставил кролика Роджера	Фильм	1988	156452370
4	Хороший, плохой, злой	Фильм	1967	25118063
5	Последний киногерой	Фильм	1993	137298489
6	Она написала убийство	Сериал	1984	<null>
7	Лас-Вегас	Сериал	2003	<null>
8	Паркер Льюис не проигрывает	Сериал	1990	<null>
9	Розовая пантера: Контроль за вредителями	Мультфильм	1969	<null>
10	Койот против Аспе	Фильм	2023	<null>

Если в конструкции **UPDATE ... SET ...** не применить условие **WHERE**, то, например, в случае нашего запроса при выполнении команды:

```
update movies set gross=156452370;
```

значение *156452370* установилось бы во **ВСЕХ** записях таблицы, что очень небезопасно.

## COUNT

Агрегирующая функция **COUNT()** (англ. «подсчёт») возвращает количество строк в результирующей выборке.

Этот запрос переводится с SQL на русский как «СОСЧИТАЙ все строки в таблице **movies**»:

```
SELECT COUNT(*)
FROM movies;
```

В результате выполнения этого запроса вернётся число 10:

count
10

Через **COUNT** можно подсчитать число записей в любой выборке, например, с условием **WHERE**.

### MIN и MAX

Эти агрегирующие функции предназначены для поиска максимального или минимального значения в заданном столбце.

```
SELECT MIN(gross)
FROM movies;
-- Вернёт: 25118063
```

```
SELECT MAX(gross)
FROM movies;
-- Вернёт: 156452370
```

Функции **MIN** и **MAX** игнорируют значения **NULL**. Чего нет — то не может быть подсчитано.

Найти минимальное или максимальное значение можно и без агрегирующих функций — для этого сортируем таблицу по заданному полю и берём первое значение. Но такой запрос будет работать намного дольше, чем агрегатные функции.

### AVG и SUM

Часто требуется получить средние значения и итоговую сумму по какому-то столбцу. SQL может вернуть и такие данные: для этого есть агрегирующие функции **AVG** (от англ. average «среднее») и **SUM** (от англ. «сумма»). Эти функции тоже игнорируют значения **NULL**. Подсчитываемая выборка может быть ограничена с помощью **WHERE**.

```
SELECT AVG(gross)
FROM movies
WHERE release_year > 1980;
-- Вернёт: 146875429.5

SELECT SUM(gross)
FROM movies
WHERE release_year > 1980;
-- Вернёт: 293750859
```

В веб-разработке **SUM** часто используется для подсчёта итоговой суммы в заказе, а **AVG** — для работы со всевозможными рейтингами.

## 3.2 Группировка GROUP BY и фильтрация HAVING

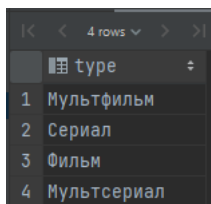
Для некоторых задач нужно группировать результирующую выборку по одному или нескольким столбцам. Такие группировки помогают, например, понять, насколько средний чек зависит от пола и возраста посетителя или от времени посещения ресторана.

### Группировка данных: GROUP BY

Самый простой вариант использования **GROUP BY** — найти «группы», объединённые одинаковым значением в заданной колонке:

```
SELECT type
FROM movies
GROUP BY type;
```

Все записи с одинаковыми значениями в поле **type** были объединены в группы, и в ответ на запрос вернулось по одной записи из каждой «группы»:



	type
1	Мультфильм
2	Сериал
3	Фильм
4	Мультсериал

	Name	Type	Release_year	Gross
Группа "Мультсериал"	Весёлые мелодии	Мультсериал	1930	NULL
	Безумные Мелодии Луни Тюнз	Мультсериал	1931	NULL
Группа "Фильм"	Кто подставил кролика Роджера	Фильм	1988	156 452 370
	Хороший, плохой, злой	Фильм	1967	25 118 063
	Последний киногерой	Фильм	1993	137 298 489
Группа "Сериал"	Она написала убийство	Сериал	1984	NULL
	Лас-Вегас	Сериал	2003	NULL
	Паркер Льюис не проигрывает	Сериал	1990	NULL
Группа "Мультфильм"	Розовая пантера: Контроль за вредителями	Мультфильм	1969	NULL
Группа "Фильм"	Койот против Асте	Фильм	2023	NULL

Теперь после разбивки записей на группы к каждой из групп мы можем применять агрегирующие функции (**COUNT**, **MIN**, **MAX**, **AVG** или **SUM**).

1. Посчитаем (**COUNT**), сколько в таблице фильмов каждого типа — сколько фильмов, сколько сериалов, сколько мультфильмов и мультсериалов:

```
SELECT type,
       COUNT(*)
FROM movies
GROUP BY type;
```

	type	count
1	Мультфильм	1
2	Сериал	3
3	Фильм	4
4	Мультсериал	2

2. Найдём самый старый фильм (**MIN**) в каждой группе:

```
SELECT type,
       MIN(release_year)
FROM movies
GROUP BY type;
```

Переведём запрос на русский: «в таблице **movies** собери в группы записи с одинаковым полем **type**; покажи значение **type** и минимальное значение **release\_year** для каждой из групп»:

	type	min
1	Мультфильм	1969
2	Сериал	1984
3	Фильм	1967
4	Мультсериал	1930

3. Просуммируем кассовые сборы (**SUM**) для фильмов разного типа:

```
SELECT type,
       SUM(gross)
FROM movies
GROUP BY type;
```

	type	sum
1	Мультфильм	<null>
2	Сериал	<null>
3	Фильм	318868922
4	Мультсериал	<null>

4. Данные, которые попадут в группировку, можно предварительно отфильтровать через **WHERE**. Объединим фильмы в группы по типам и просуммируем кассовые сборы для каждой группы; но в группы включим лишь фильмы, выпущенные после 1990-го года; выведем название группы и сумму сборов — **SUM(gross)**:

```
SELECT type,
       SUM(gross)
FROM movies
WHERE release_year > 1990
GROUP BY type;
```

	type	sum
1	Сериал	<null>
2	Фильм	137298489

Как видно, в нашей базе нет ни мультфильмов, ни мультсериалов младше 1990-го года.

5. Следует иметь в виду, что для **GROUP BY** все значения **NULL** трактуются как равные, то есть при группировке по полю, содержащему **NULL**-значения, все строки с **NULL** попадут в одну группу. Сгруппируем записи по одинаковым значениям в поле **gross** и выведем количество записей в каждой группе:

```
SELECT gross,
       COUNT(*)
FROM movies
GROUP BY gross;
```

	gross	count
1	<null>	7
2	137298489	1
3	156452370	1
4	25118063	1

### Фильтрация групп. HAVING

Оператор **HAVING** позволяет выполнить фильтрацию групп: он определяет, какие группы будут включены в результирующую выборку. Работа **HAVING** во многом аналогична применению **WHERE**; но **WHERE** применяется для фильтрации строк, а **HAVING** — для фильтрации групп.

Исключим из выборки все группы, в которых сумма кассовых сборов равна **NULL**:

```
SELECT type,
       SUM(gross)
FROM movies
GROUP BY type
HAVING SUM(gross) IS NOT NULL;
```

	type	sum
1	Фильм	318868922



Шпаргалка со списком самых востребованных операторов и ключевых слов, применяемых с получением данных из таблиц (**SELECT**) в SQL-запросах:

**SELECT** ('столбцы (\* - для выбора всех столбцов); обязательно')

(могут применяться агрегатные функции COUNT, MIN, MAX, AVG и SUM; необязательно)

(и ключевое слово **DISTINCT**; необязательно)

**FROM** ('таблица; обязательно')

**WHERE** ('условие/фильтрация; необязательно')

**GROUP BY** ('столбец, по которому нужно сгруппировать данные; необязательно')

**HAVING** ('условие/фильтрация на уровне сгруппированных данных; необязательно')

**ORDER BY** ('столбец, по которому нужно ранжировать вывод; необязательно')

**LIMIT** ('сколько записей показывать; необязательно')

**OFFSET** ('сколько записей в выборке пропустить; необязательно')

## 4 Отношение между таблицами

Превратим таблицу с перечнем фильмов в полноценную и гибкую базу данных; добавим информации и изменим архитектуру, создадим в базе несколько связанных таблиц.

Будут следующие таблицы:

- **movies** (Фильмы). В ней будут храниться русские названия фильмов и ссылки на записи из других таблиц.
- **original\_names** (Оригинальные названия) — перечень оригинальных названий фильмов из таблицы **movies**.
- **types** (Типы) — перечень возможных типов фильмов (фильм, мультфильм и другие).
- **directors** (Режиссёры) — имена режиссёров.

К каждому фильму из таблицы **movies** должно быть привязано оригинальное название, тип и режиссёр.

Для привязки данных из других таблиц создадим в таблице **movies** поля, в которых будем указывать **id** нужной записи из таблиц **original\_names**, **types** и **directors**.

### movies

id	name	original_name_id	type_id	director_id
1	Безумные Мелодии Луни Тюнз	3	2	1
2	Весёлые мелодии	6	2	1
3	Кто подставил кролика Роджера	5	3	2
4	Хороший, плохой, злой	4	3	3
5	Последний киногерой	1	3	4
6	Она написала убийство	2	4	5, 6, 7

Названия полей, ссылающихся на другие таблицы, могут быть любыми, но лучше составлять их по принципу *имяТаблицы\_имяПоля*.

В остальных таблицах будет храниться следующая информация:

### original\_names

id	name
1	Last Action Hero
2	Murder, She Wrote
3	Looney Tunes
4	Il Buono, il brutto, il cattivo
5	Who Framed Roger Rabbit
6	Merrie Melodies

### types

id	name
1	Мультфильм
2	Мультсериал
3	Фильм
4	Сериал

### directors

id	name
1	Текс Эйвери
2	Роберт Земекис
3	Серджо Леоне
4	Джон Мактирнан
5	Кори Аллен
6	Питер Фишер
7	Ричард Левинсон

Можно схематично изобразить ссылки на другие таблицы для фильма «Она написала убийство»:

id	name	original_name_id	type_id	director_id
1	Безумные Мелодии Луни Тюнз	3	2	1
2	Весёлые мелодии	6	2	1
3	Кто подставил кролика Роджера	5	3	2
4	Хороший, плохой, злой	4	3	3
5	Последний киногерой	1	3	4
6	Она написала убийство	2	4	5, 6, 7

original\_names

id	name
1	Last Action Hero
2	Murder, She Wrote
3	Looney Tunes
4	Il Buono, il brutto, il cattivo
5	Who Framed Roger Rabbit
6	Merrie Melodies

types

id	name
1	Мультфильм
2	Мультсериал
3	Фильм
4	Сериал

directors

id	name
1	Текс Эйвери
2	Роберт Земекис
3	Серджо Леоне
4	Джон Мактирнан
5	Кори Аллен
6	Питер Фишер
7	Ричард Левинсон

Зная связанные с записью идентификаторы из других таблиц, можно по цепочке извлечь дополнительные данные и получить полную информацию о фильме:

- **Русское название:** Она написала убийство
- **Оригинальное название:** Murder, She Wrote
- **Тип:** Сериал
- **Режиссёр:** Кори Аллен, Питер Фишер, Ричард Левинсон

Поля между собой связаны по-разному:

- У каждого фильма есть лишь одно оригинальное название, значит, два разных фильма не могут ссылаться на одну и ту же запись в **таблице original\_names**.
- Разные фильмы из таблицы **movies** могут ссылаться на одну и ту же запись в таблице **types**. Например, *Безумные Мелодии Луни Тюнз* и *Весёлые мелодии* — это два мультсериала.
- Есть ещё один вариант: у одного фильма может быть несколько режиссёров, то есть один фильм может ссылаться на несколько записей из таблицы **directors** (как, например, *Она написала убийство*); но и режиссёр может снять несколько фильмов (как *Текс Эйвери* в нашей базе).

Всё это — различные **типы связей**.

## 4.1 Связь «один-к-одному» (1:1)

В жизни отношения один-к-одному довольно часты:

- **Школа — директор.** В одной школе — один директор.
- **Машина — VIN-код.** У одного автомобиля — один уникальный номер.
- **Человек — рисунок радужной оболочки глаза.** Рисунок уникален для каждого человека — одинаковых глаз у разных людей быть не может.

Но в базах данных связь один-к-одному применяют редко: проще объединить две таблицы в одну, чем использовать эту связь.

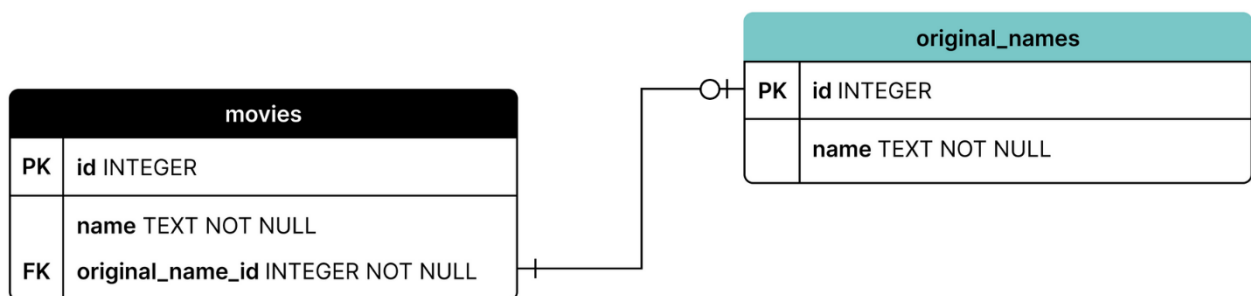
Свяжем каждый фильм с его оригинальным названием, которое хранится в другой таблице, для этого создадим новую БД с двумя таблицами — **movies** и **original\_names**.

В таблице **movies** создадим три поля:

- **id** — *primary key*;
- **name** — название фильма (обязательное поле);
- **original\_name\_id** — *foreign key*; через это поле таблица **movies** будет связана с таблицей **original\_names**.

В таблице **original\_names** будет два поля:

- **id** — *primary key*;
- **name** — оригинальное название фильма (обязательное поле);



```
CREATE TABLE IF NOT EXISTS original_names
(
    id    INTEGER PRIMARY KEY,
    name  TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS movies
(
    id                INTEGER PRIMARY KEY,
    name              TEXT    NOT NULL,
    original_name_id  INTEGER NOT NULL UNIQUE,
    FOREIGN KEY (original_name_id) REFERENCES original_names (id)
);
```

Вот как создаётся связь:

1. В таблице **movies** создаётся поле **original\_name\_id** типа **INTEGER** (только числа).
2. **UNIQUE** объявляет это поле уникальным в пределах таблицы: в ячейках этой колонки не может быть двух одинаковых значений — ведь два разных фильма не могут ссылаться на одно и то же оригинальное название.
3. **NOT NULL** — поле не может быть пустым, при создании каждой записи его обязательно нужно заполнять.
4. **FOREIGN KEY(original\_name\_id)** — объявляет поле **original\_name\_id** внешним ключом.
5. Ключевое слово **REFERENCES** указывает с каким полем связан ключ. В нашем случае это поле **id** в таблице **original\_names**.

Заполним таблицы. Для этого сопоставим русские названия фильмов с оригинальными. При заполнении таблицы **movies** важно указать правильный **original\_name\_id**, чтобы ссылка вела на нужную запись в таблице **original\_names**.

Русские названия	Оригинальные названия
Безумные Мелодии Луни Тюнз	Looney Tunes
Весёлые мелодии	Merrie Melodies
Кто подставил кролика Роджера	Who Framed Roger Rabbit
Хороший, плохой, злой	Il Buono, il brutto, il cattivo
Последний киногерой	Last Action Hero
Она написала убийство	Murder, She Wrote

```
INSERT INTO original_names(id, name)
VALUES (1, 'Last Action Hero'),
       (2, 'Murder, She Wrote'),
       (3, 'Looney Tunes'),
       (4, 'Il Buono, il brutto, il cattivo'),
       (5, 'Who Framed Roger Rabbit'),
       (6, 'Merrie Melodies');

INSERT INTO movies(id, name, original_name_id)
VALUES (1, 'Безумные Мелодии Луни Тюнз', 3),
       (2, 'Весёлые мелодии', 6),
       (3, 'Кто подставил кролика Роджера', 5),
       (4, 'Хороший, плохой, злой', 4),
       (5, 'Последний киногерой', 1),
       (6, 'Она написала убийство', 2);
```

В результате выполнения этого кода будут созданы две таблицы, связанные по типу «*один-к-одному*».

Выборку из таблиц, связанных 1:1, можно получить, например, таким запросом:

```
-- Вернуть поле name из таблицы movies и поле name из original_names
SELECT movies.name,
       original_names.name
-- ...из двух таблиц
FROM movies,
      original_names
-- Выводить только те значения полей, для которых верно условие
WHERE movies.original_name_id = original_names.id;
```

movies.name	original_names.name
1 Последний киногерой	Last Action Hero
2 Она написала убийство	Murder, She Wrote
3 Безумные Мелодии Луни Тюнз	Looney Tunes
4 Хороший, плохой, злой	Il Buono, il brutto, il cattivo
5 Кто подставил кролика Роджера	Who Framed Roger Rabbit
6 Весёлые мелодии	Merrie Melodies

При выборке из нескольких таблиц нужно указывать не только имя поля, но и имя таблицы, чтобы СУБД точно знала, о каком поле идёт речь.

Длинным названиям столбцов или таблиц лучше давать короткие псевдонимы (по англ. «*alias*») с помощью оператора **AS**:

```
SELECT m.name AS translation,
       o.name AS original
FROM movies as m,
      original_names as o
WHERE m.original_name_id = o.id
-- Добавим условие
AND o.name LIKE 'M%';
```

	translation	original
1	Весёлые мелодии	Merrie Melodies
2	Она написала убийство	Murder, She Wrote

## 4.2 Связь «один-ко-многим» (1:M)

Самая часто встречающаяся связь в реляционных БД — это отношение «*один-ко-многим*» и её брат-близнец «*многие-к-одному*».

Эта связь описывает, например, такие отношения:

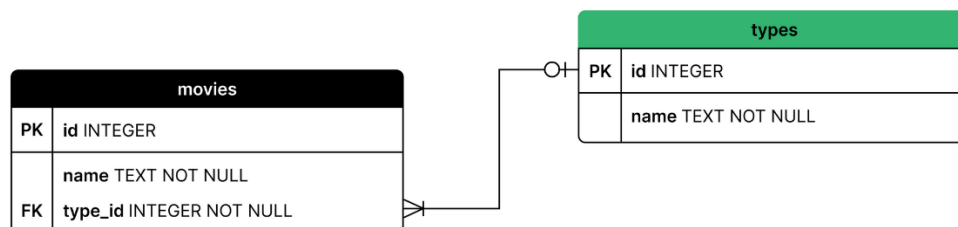
- **Классы в школе:** в одной школе есть много классов (*один-ко-многим*). Но каждый класс принадлежит только одной школе (*многие-к-одному*).
- **Кинотеатры и экраны:** в одном кинотеатре может быть много экранов (*один-ко-многим*), но множество экранов принадлежит только одному кинотеатру (*многие-к-одному*).
- **Место рождения:** в городе может родиться много детей, но каждый из этих детей рождён в одном городе.

Основная идея связи **1:M** в том, что с какой-то одной сущностью можно связать много других. Например, у фильма есть тип (мультфильм, фильм, сериал...); данные о типах хранятся в таблице **types**.

В нашей фильмотеке каждое произведение может относиться к одному из типов: мультфильм, фильм, сериал или мультсериал.

Название	Тип
Безумные Мелодии Луни Тюнз	Мультсериал
Весёлые мелодии	Мультсериал
Кто подставил кролика Роджера	Фильм
Хороший, плохой, злой	Фильм
Последний киногерой	Фильм
Она написала убийство	Сериал

Перечень типов фильмов сохраним в таблице **types**; к этой таблице должна «тянуться» связь от **movies**.



Забудем пока про поле **original\_name\_id** и сосредоточимся на связи с **types**. Для этого удалим таблицу **movies** и создадим ее заново, но уже с **FK-полем** **type\_id**, конечно, предварительно создав таблицу **types**:

```

DROP TABLE movies;

CREATE TABLE IF NOT EXISTS types
(
    id    INTEGER PRIMARY KEY,
    name  TEXT NOT NULL
);
  
```

```
CREATE TABLE IF NOT EXISTS movies
(
    id          INTEGER PRIMARY KEY,
    name        TEXT          NOT NULL,
    type_id     INTEGER NOT NULL,
    FOREIGN KEY (type_id) REFERENCES types (id)
);
```

Для поля **type\_id** не указан параметр **UNIQUE**; таким образом, несколько записей из **movies** могут ссылаться на одну и ту же запись в **types**. Таким образом, кинолента может относиться только к одному типу, и для любого фильма обязательно должен быть указан его тип — значение поля **type\_id** не может быть **NULL**. С другой стороны — к одному типу может относиться много фильмов из таблицы **movies**. Это связь «*один-ко-многим*».

Заполним таблицы данными:

```
INSERT INTO types(id, name)
VALUES (1, 'Мультфильм'),
       (2, 'Мультсериал'),
       (3, 'Фильм'),
       (4, 'Сериал');

INSERT INTO movies(id, name, type_id)
VALUES (1, 'Безумные Мелодии Луни Тюнз', 2),
       (2, 'Весёлые мелодии', 2),
       (3, 'Кто подставил кролика Роджера', 3),
       (4, 'Хороший, плохой, злой', 3),
       (5, 'Последний киногерой', 3),
       (6, 'Она написала убийство', 4);
```

Выбрать все названия кинолент из категории 'Фильм' можно таким запросом:

```
SELECT movies.name,
       types.name
FROM movies,
       types
WHERE movies.type_id = types.id
      AND types.name = 'Фильм';
```

	movies.name	types.name
1	Кто подставил кролика Роджера	Фильм
2	Хороший, плохой, злой	Фильм
3	Последний киногерой	Фильм

### 4.3 Связь «многие-ко-многим» (М:М)

Это самый сложный тип связей: нескольким записям из одной таблицы могут соответствовать несколько записей из другой.

- **Учителя-предметники.** Один учитель может преподавать в нескольких классах, в то же самое время в одном классе может преподавать несколько учителей. Несколько учителей — несколько классов.
- **Фильмы — режиссёры.** Каждый режиссёр может снимать разные фильмы. У одного фильма может быть несколько режиссёров.

В нашем случае у фильма может быть один или несколько режиссёров, а режиссёр может снять один фильм или несколько фильмов.

Соберём имена режиссёров в отдельную таблицу:

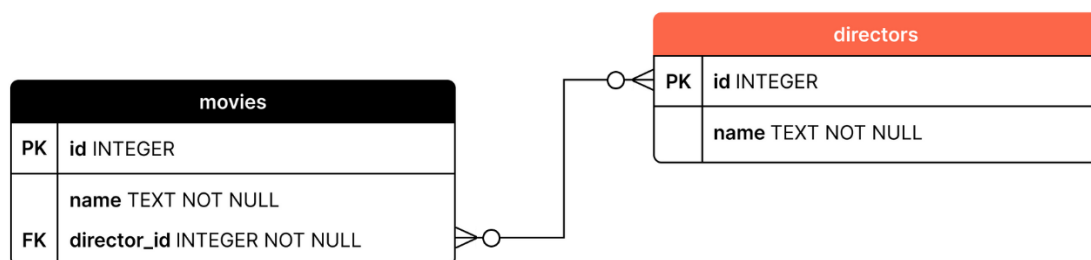
## directors

id	name
1	Текс Эйвери
2	Роберт Земекис
3	Серджо Леоне
4	Джон Мактирнан
5	Кори Аллен
6	Питер Фишер
7	Ричард Левинсон

Сопоставим названия фильмов с **id** режиссёров:

Название фильма	id режиссёра
Безумные Мелодии Луни Тюнз	1
Весёлые мелодии	1
Кто подставил кролика Роджера	2
Хороший, плохой, злой	3
Последний киногерой	4
Она написала убийство	5, 6, 7

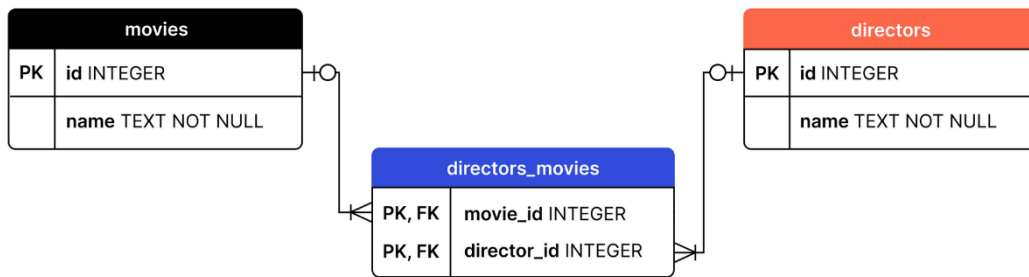
Упрощённо эту связь можно отобразить так: множественная связь к таблице **movies** и множественная — к таблице **directors**.



Но у фильма «Она написала убийство» целых три режиссёра, и чтобы указать их — в поле **director\_id** пришлось бы перечислить несколько **id**. Однако в одной ячейке типа **INTEGER** должно храниться только одно число.

На практике такую связь организуют через промежуточную таблицу: в ней сопоставляют PK записей из связанных таблиц.





Промежуточную таблицу принято называть по именам таблиц, которые связываются через неё: **directors\_movies**.

Она содержит

- поле **director\_id**, которое хранит *FK*, ссылающийся на колонку **id** в таблице **directors**;
- поле **movie\_id**, которое хранит *FK*, ссылающийся на колонку **id** в таблице **movies**.

movie_id	director_id
1	1
2	1
6	5
6	6
6	7

Эта таблица связывает фильмы с режиссёрами:

Режиссёры
Безумные Мелодии Луни Тюнз — Текс Эйвери
Весёлые мелодии — Текс Эйвери
Она написала убийство — Кори Аллен
Она написала убийство — Питер Фишер
Она написала убийство — Ричард Левинсон

Любой режиссер может снять один фильм только один раз. Ведь даже если он переснимет его спустя 10, 20, 1000 лет — это уже будет другой фильм. Поэтому здесь в качестве **PRIMARY KEY** используется сочетание (**director\_id, movie\_id**) — эта пара уникальна в пределах таблицы. Сочетание значений этих полей служит **композиционным** первичным ключом.

Создать таблицу с композиционным первичным ключом можно так:

```
CREATE TABLE IF NOT EXISTS movies
(
    id    INTEGER PRIMARY KEY,
    name  TEXT NOT NULL
);
```

```
CREATE TABLE IF NOT EXISTS directors
(
    id    INTEGER PRIMARY KEY,
    name  TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS directors_movies
(
    director_id INTEGER NOT NULL,
    movie_id    INTEGER NOT NULL,
    -- Пару полей назначаем композитным первичным ключом:
    PRIMARY KEY (director_id, movie_id),
    FOREIGN KEY (director_id) REFERENCES directors (id),
    FOREIGN KEY (movie_id) REFERENCES movies (id)
);
```

Перевод с SQL на русский выглядит следующим образом:

1. создаём таблицы **movies** и **directors** и промежуточную таблицу **directors\_movies**;
2. в промежуточной таблице указываем связь между записями из двух таблиц;
3. в таблице **directors\_movies** создаём **компози́тный** первичный ключ из полей **director\_id** и **movie\_id**; при этом решается и вторая задача: теперь каждая пара полей будет уникальна в пределах таблицы, и никто не сможет повторно связать фильм с режиссёром.

Теперь любую запись из таблицы **directors** можно связать с несколькими фильмами из **movies**, а фильм можно связать с несколькими режиссёрами.

## 5 Объединение таблиц: JOIN

Давайте посмотрим, что вернет следующий запрос:

```
select *
from movies,
types;
```

	movies.id	movies.name	type_id	types.id	types.name
1	1	Безумные Мелодии Луни Тюнз	2	1	Мультфильм
2	2	Весёлые мелодии	2	1	Мультфильм
3	3	Кто подставил кролика Роджера	3	1	Мультфильм
4	4	Хороший, плохой, злой	3	1	Мультфильм
5	5	Последний киногерой	3	1	Мультфильм
6	6	Она написала убийство	4	1	Мультфильм
7	1	Безумные Мелодии Луни Тюнз	2	2	Мультсериал
8	2	Весёлые мелодии	2	2	Мультсериал
9	3	Кто подставил кролика Роджера	3	2	Мультсериал
10	4	Хороший, плохой, злой	3	2	Мультсериал
11	5	Последний киногерой	3	2	Мультсериал
12	6	Она написала убийство	4	2	Мультсериал
13	1	Безумные Мелодии Луни Тюнз	2	3	Фильм
14	2	Весёлые мелодии	2	3	Фильм

В результирующую выборку включены все поля объединяемых таблиц; к каждой записи из **movies** присоединена каждая запись из **types**.

Выглядит не очень практично. Давайте получим из таблицы **movies** все записи: оставим только те строки, в которых значение поля **type\_id** в таблице **movies** равно значению **id** в таблице **type**:

```
SELECT *
FROM movies,
types
WHERE movies.type_id = types.id;
```

	movies.id	movies.name	type_id	types.id	types.name
1	1	Безумные Мелодии Луни Тюнз	2	2	Мультсериал
2	2	Весёлые мелодии	2	2	Мультсериал
3	3	Кто подставил кролика Роджера	3	3	Фильм
4	4	Хороший, плохой, злой	3	3	Фильм
5	5	Последний киногерой	3	3	Фильм
6	6	Она написала убийство	4	4	Сериал

В нашем запросе конструкция **FROM movies, slogans** соединяет таблицы, а **WHERE** — фильтрует получившуюся выборку.

По стандарту SQL92 принято отделять фильтрацию от условий соединения таблиц с помощью оператора **JOIN** (англ. «соединение»):

```
-- Верни все поля
SELECT *
-- из таблицы movies
FROM movies
-- ...но перед этим присоедини таблицу slogans так, чтобы в записях
-- совпадали значения полей movies.slogan_id и slogans.id
JOIN types ON movies.type_id = types.id;
```

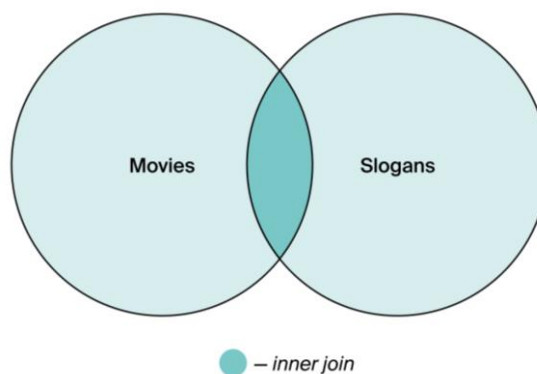
Результат будет тот же, а запрос выглядит лучше:

- Соединение и фильтрация разделены; это упрощает понимание запроса.
- Условия соединения для каждой пары таблиц содержатся в блоке ON, это уменьшает вероятность ошибки.

В классическом SQL существует пять типов JOIN. Рассмотрим каждый из них.

## 5.1 INNER JOIN

В примере выше был как раз **INNER JOIN** (или просто **JOIN**, без титулов). Объединение таблиц через INNER JOIN можно представить схематически:



Запрос через **JOIN** можно сделать и к трём таблицам:

```
SELECT movies.name,
       slogans.name,
       types.name
FROM movies
JOIN slogans ON movies.slogan_id = slogans.id
JOIN types ON movies.type_id = types.id;
```

	movies.name	slogans.name	types.name
1	Последний киногерой	This isn't the movies anymore	Фильм
2	Хороший, плохой, злой	For Three Men The Civil War Wasn't Hell. It Was Practice!	Фильм
3	Она написала убийство	Tonight on Murder She Wrote	Сериал

В этой выборке всего три фильма: мультики в неё не попали, ведь у них не указана связь с таблицей **slogans**.

Оператор **INNER JOIN** включает в результирующую таблицу только те записи, в которых выполняется условие, заданное в **ON**.

## 5.1 LEFT OUTER JOIN

При обработке запроса **LEFT OUTER JOIN** объединяемые таблицы условно называют «левая» и «правая». «Левая» — та, которая вызвана в блоке **FROM**, «правая» — та, что указана после ключевого слова **JOIN**. «Правых» таблиц может быть и несколько.

В этом запросе **OUTER** — не обязательное слово. Можно использовать сокращённую запись: **LEFT JOIN**.

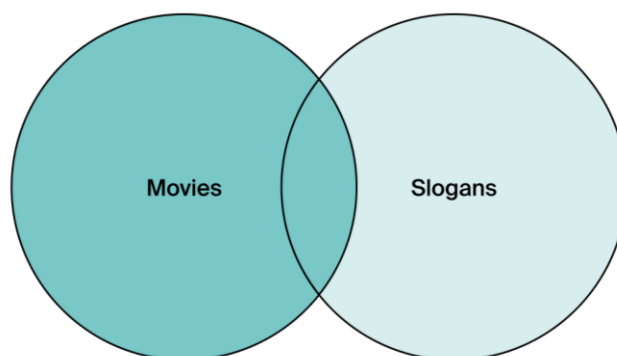
При выполнении запросов с **LEFT JOIN** возвращаются все строки левой таблицы. Данными из правой таблицы дополняются только те строки левой таблицы, для которых выполняются условия соединения, описанные после оператора **ON**. Для недостающих данных вместо строк правой таблицы вставляются **NULL**-значения.

```
SELECT movies.name,  
       slogans.name  
FROM movies  
LEFT JOIN slogans ON movies.slogan_id = slogans.id;
```

movies.name	slogans.name
1 Безумные Мелодии Луни Тюнз	<null>
2 Весёлые мелодии	<null>
3 Кто подставил кролика Роджера	<null>
4 Последний киногерой	This isn't the movies anymore
5 Хороший, плохой, злой	For Three Men The Civil War Wasn't Hell. It Was Practice!
6 Она написала убийство	Tonight on Murder She Wrote

Первые три записи остались в выборке, поскольку **LEFT JOIN** возвращает все записи левой таблицы без исключения.

Графически эта связь изображается так:



## 5.3 RIGHT OUTER JOIN

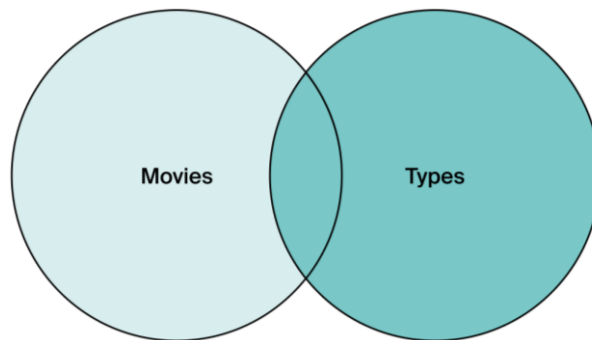
**RIGHT JOIN** — это такое же объединение, как и **LEFT JOIN**, но выводятся все записи из правой таблицы, а к ним добавляются только те данные из левой таблицы, в которых есть ключ объединения.

```
SELECT movies.name,  
       types.name  
FROM movies  
RIGHT JOIN types ON movies.type_id = types.id;
```

	movies.name	types.name
1	Безумные Мелодии Луни Тюнз	Мультсериал
2	Весёлые мелодии	Мультсериал
3	Кто подставил кролика Роджера	Фильм
4	Последний киногерой	Фильм
5	Хороший, плохой, злой	Фильм
6	Она написала убийство	Сериал
7	<null>	Мультфильм

Будут выведены все записи из правой таблицы (из **types**) и связанные с ними записи из левой (из **movies**). В таблице **movies** нет ни одной записи, ссылающейся на тип «*Мультфильм*», поэтому вместо значения **movies.name** в этой строке стоит **None**.

Графическое отображение **RIGHT JOIN**:



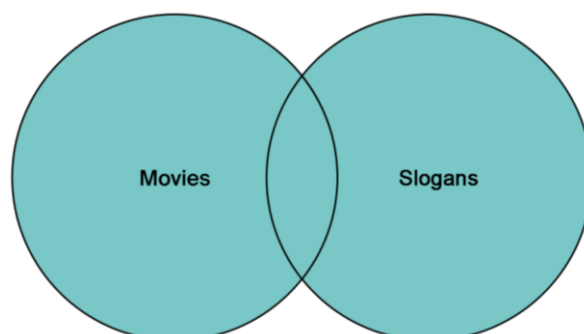
## 5.4 FULL OUTER JOIN

При запросе **FULL (OUTER) JOIN** выводятся все записи из объединяемых таблиц. Те записи, у которых запрошенные значения совпадают — выводятся парами, у остальных недостающее значение заменяется на **NULL**. Другими словами, **FULL JOIN == LEFT JOIN + RIGHT JOIN**.

```
SELECT movies.name,
       slogans.name
FROM movies
FULL JOIN slogans ON movies.slogan_id = slogans.id;
```

	movies.name	slogans.name
1	Безумные Мелодии Луни Тюнз	<null>
2	Весёлые мелодии	<null>
3	Кто подставил кролика Роджера	<null>
4	Последний киногерой	This isn't the movies anymore
5	Хороший, плохой, злой	For Three Men The Civil War Wasn't Hell. It Was Practice!
6	Она написала убийство	Tonight on Murder She Wrote
7	<null>	I'll be back

Графически **FULL JOIN** можно представить так:



## 5.5 CROSS JOIN

Объединение таблиц через **CROSS JOIN** возвращает декартово произведение таблиц — каждая запись левой таблицы объединится с каждой записью правой. Параметр **ON** при запросах **CROSS JOIN** не применяется.

```
SELECT movies.name,  
       slogans.name  
FROM movies  
CROSS JOIN slogans;
```

	movies.name	slogans.name
1	Безумные Мелодии Луни Тюнз	For Three Men The Civil War Wasn't Hell. It Was Practice!
2	Весёлые мелодии	For Three Men The Civil War Wasn't Hell. It Was Practice!
3	Кто подставил кролика Роджера	For Three Men The Civil War Wasn't Hell. It Was Practice!
4	Последний киногерой	For Three Men The Civil War Wasn't Hell. It Was Practice!
5	Хороший, плохой, злой	For Three Men The Civil War Wasn't Hell. It Was Practice!
6	Она написала убийство	For Three Men The Civil War Wasn't Hell. It Was Practice!
7	Безумные Мелодии Луни Тюнз	This isn't the movies anymore
8	Весёлые мелодии	This isn't the movies anymore
9	Кто подставил кролика Роджера	This isn't the movies anymore
10	Последний киногерой	This isn't the movies anymore
11	Хороший, плохой, злой	This isn't the movies anymore
12	Она написала убийство	This isn't the movies anymore
13	Безумные Мелодии Луни Тюнз	Tonight on Murder She Wrote
14	Весёлые мелодии	Tonight on Murder She Wrote
15	Кто подставил кролика Роджера	Tonight on Murder She Wrote
16	Последний киногерой	Tonight on Murder She Wrote
17	Хороший, плохой, злой	Tonight on Murder She Wrote
18	Она написала убийство	Tonight on Murder She Wrote
19	Безумные Мелодии Луни Тюнз	I'll be back
20	Весёлые мелодии	I'll be back
21	Кто подставил кролика Роджера	I'll be back
22	Последний киногерой	I'll be back
23	Хороший, плохой, злой	I'll be back
24	Она написала убийство	I'll be back

На практике **CROSS JOIN** применяется редко, но и он может быть полезен. Например, если в одной таблице сохранён список жидкостей, а во второй — список возможной тары для расфасовки, от маленькой баночки до цистерны. В выборке получим все возможные комбинации «жидкость — тара».