GeekBrains

Базы данных

Урок 5

Сложные запросы

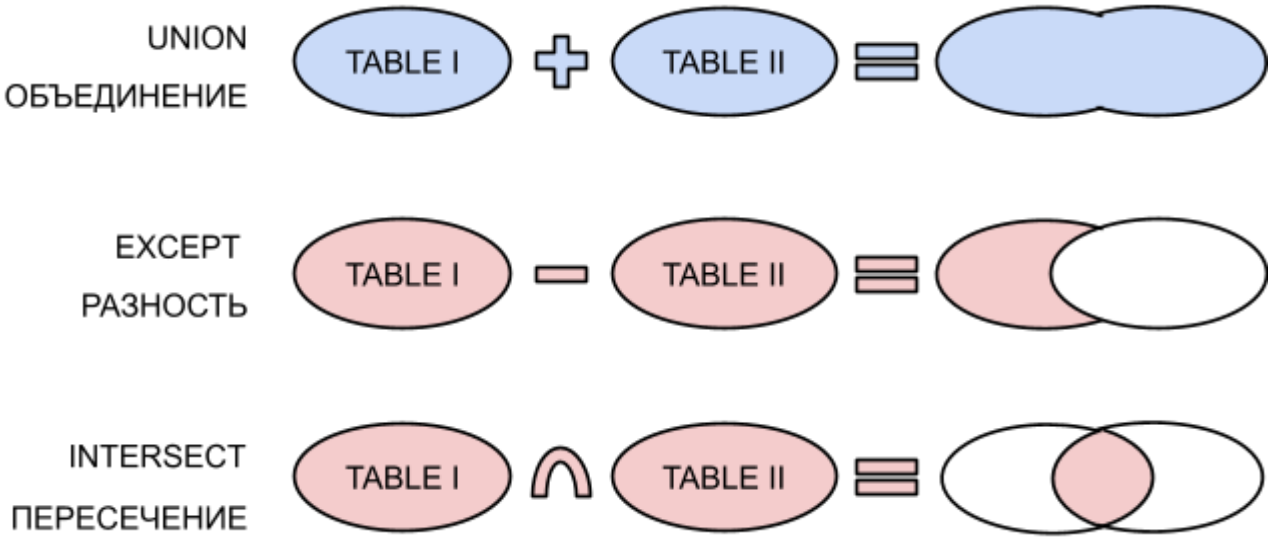
Многотабличные запросы

До этого мы обращались только к одной таблице, но настало время попробовать многотабличные запросы, результат в которых можно формировать из двух и более таблиц.

Многотабличные запросы условно можно поделить на три большие группы:

- объединение **UNION**,
- вложенные запросы,
- JOIN-соединения.

Сильная сторона SQL — то, что в его основе лежит теория множества. В отличие от других языков программирования, мы оперируем не отдельными значениями, а их наборами. В теории множеств описывается, как можно складывать и вычитать такие наборы или получать их пересечения.

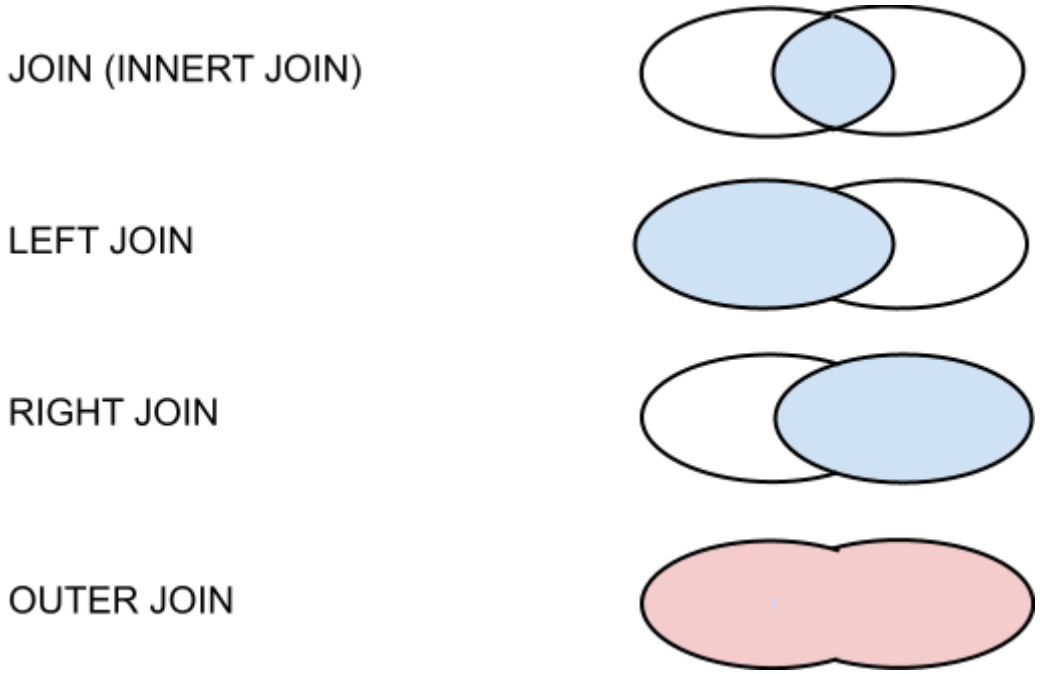


SQL поддерживает все эти операции, предоставляя операторы **UNION**, **EXCEPT** и **INTERSECT**. К сожалению, MySQL поддерживает только **UNION**, поддержка **EXCEPT** и **INTERSECT** не реализована.

```
SELECT
  id,
  <SUBQUERY>
FROM
  <SUBQUERY>
WHERE
  <SUBQUERY>
GROUP BY
  id
HAVING
  <SUBQUERY>
```

Еще один тип запросов — это вложенные запросы. Вложенный запрос позволяет использовать результат, возвращаемый одним запросом, в другом. Здесь синим цветом представлены точки в запросе, где мы можем использовать вложенные запросы.

И, наконец, третий тип запросов — это JOIN-соединения. Они очень похожи на UNION-запросы, однако вместо объединения однотипных результатов, допускают соединения совершенно разноплановых таблиц, задействуя связь «первичный-внешний ключ».



Объединение UNION

Если формат результирующих таблиц совпадает, возможно объединение результатов выполнения двух операторов SELECT в одну результирующую таблицу. Для этого используется оператор UNION.

Важное условие — совпадение всех параметров результирующих запросов. Количество, порядок следования и тип столбцов должны совпадать. Для демонстрации работы UNION-запроса создадим таблицу **rubrics**, структура которой полностью совпадает с таблицей **catalogs**:

```
DROP TABLE IF EXISTS rubrics;
CREATE TABLE rubrics (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) COMMENT 'Название раздела'
) COMMENT = 'Разделы интернет-магазина';

INSERT INTO rubrics VALUES
  (NULL, 'Видеокарты'),
  (NULL, 'Память');
```

Выведем содержимое таблицы **catalogs**:

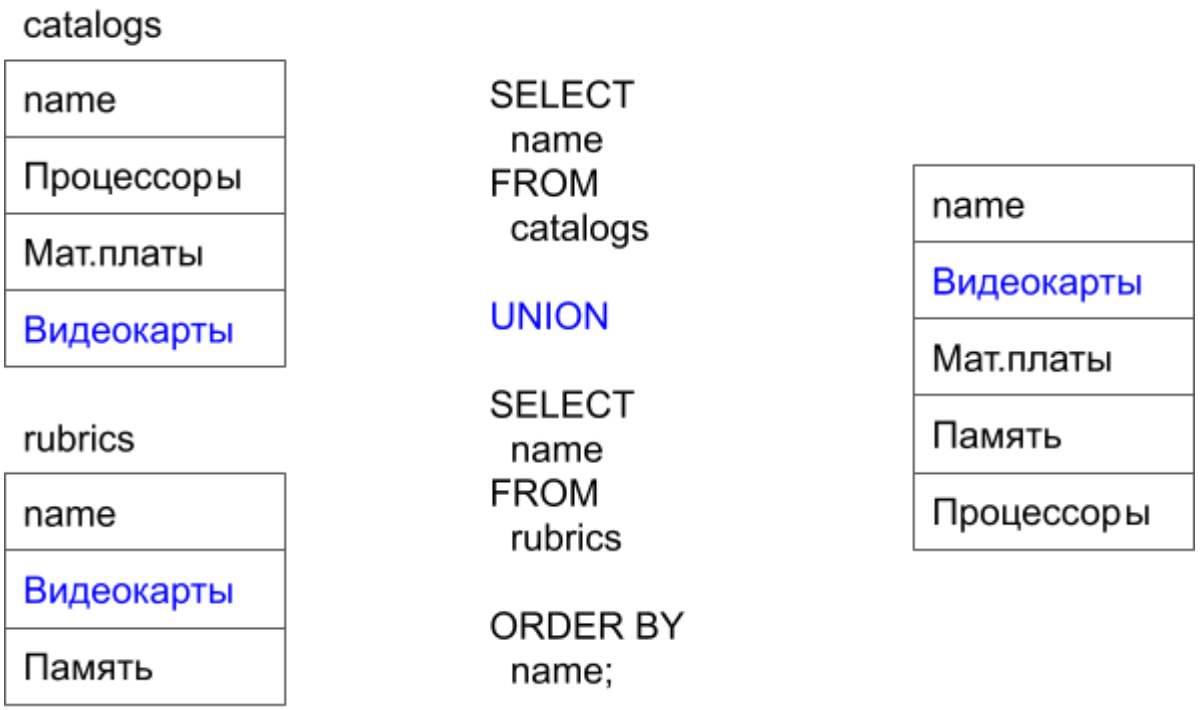
```
SELECT * FROM catalogs;
```

И таблицы **rubrics**:

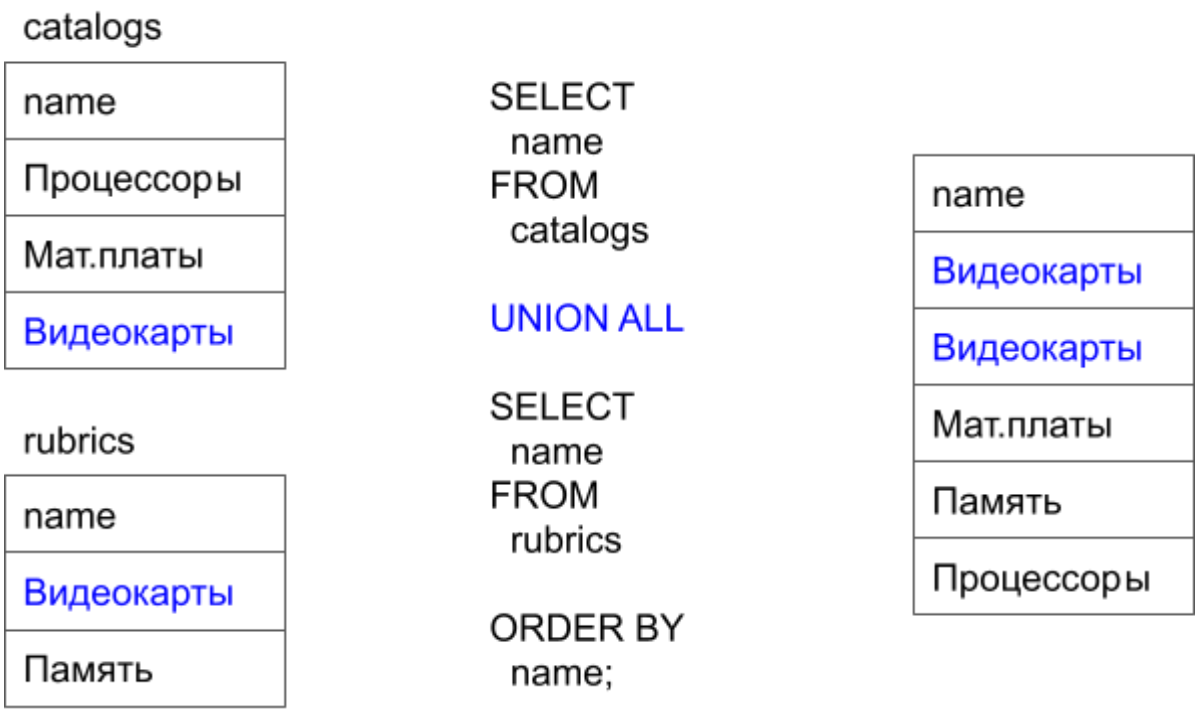
```
SELECT * FROM rubrics;
```

Чтобы объединить два результирующих запроса в один, воспользуемся **UNION**.

```
SELECT name FROM catalogs
UNION
SELECT name FROM rubrics;
```



Обратите внимание, что в результирующий запрос попадают только не повторяющиеся результаты. Несмотря на то, что раздел «Видеокарты» присутствует и в первой, и во второй таблицах, в результирующий запрос этот раздел попал в единственном экземпляре.

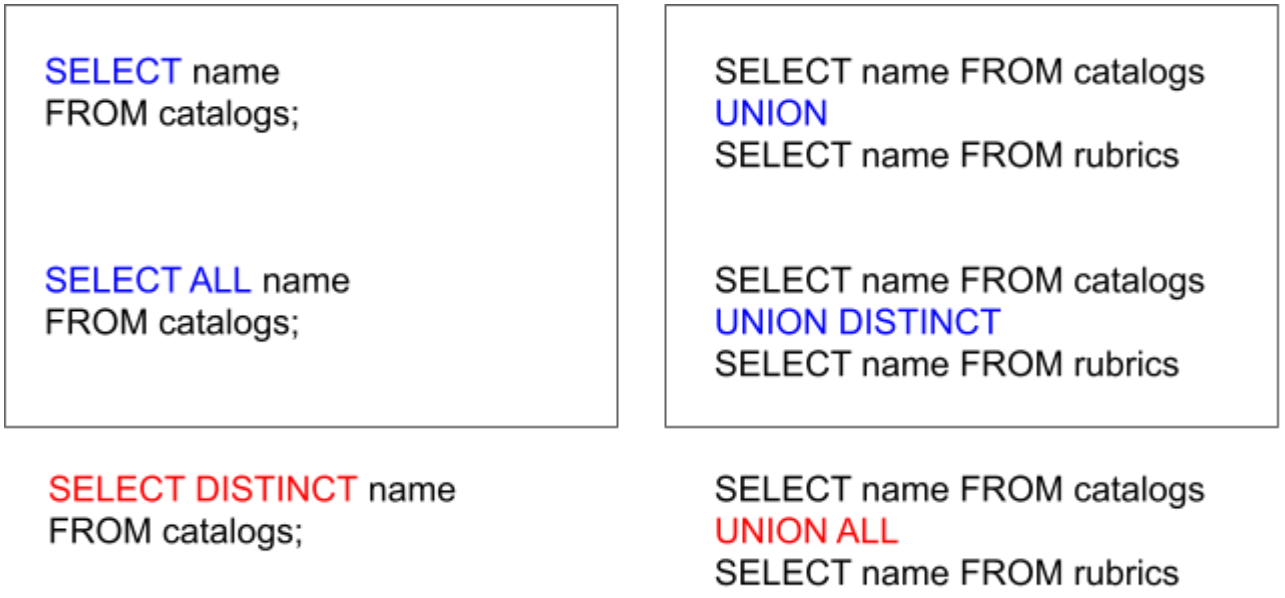


Чтобы предотвратить такое поведение, после **UNION** следует указать ключевое слово **ALL**. В этом случае в результирующий запрос будет содержать ровно столько записей, сколько находится в обеих исходных таблицах, несмотря на их дублирование.

В консоли убедимся, что запрос **UNION ALL** работает именно так, как мы описали.

```
SELECT name FROM catalogs
UNION ALL
SELECT name FROM rubrics;
```

У нас появилось две записи «Видеокарты»: одна из таблицы **catalogs**, другая — из таблицы **rubrics**:



Ключевые слова **ALL** и **DISTINCT** являются взаимозаменяемыми: если вы видите где-то в SQL-синтаксисе **ALL**, его можно заменить на **DISTINCT**. И наоборот, часто SQL-команда по умолчанию выбирает то или иное поведение, например, выводить все, включая дубли (**ALL**) или выводить только уникальные значения (**DISTINCT**).

В том случае, если ключевое слово можно опустить, его почти всегда опускают. Например, мы не пишем **SELECT ALL**, если можно писать просто **SELECT**. Точно так же в случае **UNION** мы используем **UNION ALL**, когда хотим получить все записи. Однако для получения уникальных значений используем краткую форму **UNION**, хотя могли бы писать **UNION DISTINCT**.

Если мы используем ключевое слово **ORDER BY** для сортировки, оно действует на весь результат запроса, а не на отдельные таблицы:

```
SELECT name FROM catalogs
UNION ALL
SELECT name FROM rubrics
ORDER BY name;
```

Или в обратном порядке:

```
SELECT name FROM catalogs
UNION ALL
SELECT name FROM rubrics
ORDER BY name DESC;
```

То же самое касается ключевого слова **LIMIT**: сначала происходит объединение результатов и лишь затем применяется ограничение **LIMIT**:

```
SELECT name FROM catalogs
UNION ALL
SELECT name FROM rubrics
ORDER BY name DESC
LIMIT 2;
```

Обойти это ограничение в рамках синтаксиса **UNION** нельзя. В рамках вложенных запросов мы можем использовать сначала сортировку и ограничение и лишь затем использовать полученные результаты в **UNION**.

Чтобы превратить запросы во вложенные, SELECT-команды следует поместить в круглые скобки:

```
(SELECT name FROM catalogs
ORDER BY name DESC
LIMIT 2)

UNION ALL

(SELECT name FROM rubrics
ORDER BY name DESC
LIMIT 2);
```

Если структура таблиц не совпадает, объединить их при помощи **UNION** не получится.

```
SELECT * FROM catalogs
UNION
SELECT * FROM products;
```

Попытавшись объединить таблицы **catalogs** и **products**, мы получаем сообщение об ошибке, в котором говорится, что в этих таблицах разное количество столбцов. Однако мы можем подобрать условия таким образом, что содержимое таблиц будет объединено:

```
SELECT * FROM catalogs
UNION
SELECT id, name FROM products;
```

Следует иметь в виду, что первый SELECT-запрос определяет название столбцов.

```
SELECT * FROM catalogs
UNION
SELECT id, name AS 'product' FROM products;
```

Как бы ни назывались столбцы второй таблицы, в результирующей таблице для названия столбцов будет использоваться первая. Так как мы объединяем между собой не сами таблицы, а результаты запроса к ним, мы можем объединять полностью эквивалентные запросы.

```
SELECT * FROM catalogs
UNION ALL
SELECT * FROM catalogs;
```

В **UNION** не обязательно должно участвовать только две таблицы. Используя несколько ключевых слов **UNION**, можно объединять три и более таблиц.

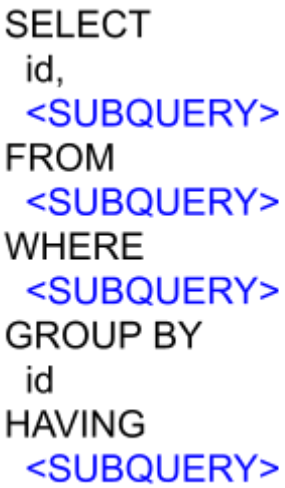
```
SELECT * FROM catalogs
UNION
SELECT id, name FROM products
UNION
SELECT id, name FROM users;
```

Не следует злоупотреблять UNION-объединениями. Как правило, UNION-запросы довольно медленно выполняются. Промежуточная таблица **UNION** в MySQL почти всегда размещается на жестком диске, поэтому все операции фильтрации и сортировки также будут осуществляться во временном файле.

По возможности после ключевого слова **SELECT** следует указывать только те столбцы, которые нужны в результирующей таблице или для составления запроса. Чем меньше столбцов указано, тем меньше размер промежуточной таблицы и тем быстрее проходят операции с ней.

Вложенные запросы

Вложенный запрос позволяет использовать результат, возвращаемый одним запросом, в другом. Синтаксис основного запроса остается неизменным, однако в местах помеченным синим цветом, можно использовать подзапрос или, как еще говорят, вложенный запрос:



Чтобы СУБД могла отличать основной запрос и подзапрос, последний заключают в круглые скобки.

Пусть мы хотим выяснить список всех товаров в разделе «Процессоры». Например, у нас имеется таблица разделов каталогов:

```
SELECT * FROM catalogs;
```

Она связана через внешний ключ **catalog_id** с таблицей товарных позиций **products**.

```
SELECT id, name, catalog_id FROM products;
```

Если мы хотим извлечь все товары, относящиеся к разделу «Процессоры», мы можем воспользоваться следующим условием:

```
SELECT id, name, catalog_id
FROM products
WHERE catalog_id = 1;
```

Таким образом, для извлечения списка процессоров нам пришлось выполнить несколько запросов. При помощи вложенных запросов мы можем объединить их в один. Извлечем первичный ключ каталога «Процессоры»:

```
SELECT id FROM catalogs WHERE name = "Процессоры";
```

Теперь превратим его во вложенный запрос:

```
SELECT
```

```
  id, name, catalog_id
FROM
  products
WHERE
  catalog_id = (SELECT id FROM catalogs WHERE name = "Процессоры");
```

При помощи вложенных запросов мы можем решать целый спектр задач. Например, найдем в таблице products товар с самой высокой ценой. Для начала найдем максимальную цену при помощи функции MAX():

```
SELECT MAX(price) FROM products;
```

Теперь мы можем сформировать вложенный запрос:

```
SELECT
  id, name, catalog_id
FROM
  products
WHERE
  price = (SELECT MAX(price) FROM products);
```

Для вложенных запросов, которые возвращают единичное значение, можно использовать не только оператор равенства, но и любой другой логический оператор. Например, найдем все товары, чья цена ниже среднего:

```
SELECT
  id, name, catalog_id
FROM
  products
WHERE
  price < (SELECT AVG(price) FROM products);
```

Для этого удобно воспользоваться агрегатной функцией **AVG**, которая возвращает среднее значение. Вложенные запросы можно использовать не только в условиях, но и, например, после ключевого слова **SELECT**.

Для каждого из товаров извлечем название каталога. Для начала выведем список товарных позиций:

```
SELECT
  id, name, catalog_id
FROM
  products;
```

Для замены внешнего ключа разделом, к которому принадлежит товар, мы можем воспользоваться следующим запросом:

```
SELECT name FROM catalogs WHERE id = 1;
```

Только вместо единицы мы должны подставить **catalog_id**:

```
SELECT
  id,
  name,
  (SELECT name FROM catalogs WHERE id = catalog_id) AS 'catalog'
FROM
  products;
```

Обратите внимание, что в WHERE-условии столбец **id** принадлежит таблицы **catalogs**, а столбец **catalog_id** — таблице **catalogs**. В случае конфликтов мы можем явно использовать квалификационные имена:

```
SELECT
  products.id,
  products.name,
  (SELECT
      catalogs.name
   FROM
      catalogs
   WHERE
      catalogs.id = products.catalog_id) AS 'catalog'
FROM
  products;
```

Если подзапрос использует столбец из внешнего запроса, его называют коррелированным. Особенность коррелированных запросов — СУБД вынуждена их вычислять для каждой строки внешнего запроса. Это может быть довольно накладно для объемных таблиц.

```
SELECT
  products.id,
  products.name,
  (SELECT MAX(price) FROM products) AS 'max_price'
FROM
  products;
```

Вложенный запрос, который вычисляет максимальную цену товара, не будет коррелированным, СУБД выполнит его один раз в начале и в каждую строку будет добавлен результат из заранее выполненного запроса.

До этого момента мы рассматривали вложенные запросы, которые всегда возвращали лишь одно значение. Если там, где СУБД ожидает одно значение, мы попробуем передать несколько, MySQL вернет сообщение об ошибке:

```
SELECT
  id, name, catalog_id
FROM
  products
WHERE
  catalog_id = (SELECT id FROM catalogs);
```

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

Чтобы воспользоваться вложенным запросом в таких условиях, нам потребуется воспользоваться специальными ключевыми словами, например, **IN**:

```
SELECT
  id, name, catalog_id
FROM
  products
WHERE
  catalog_id IN (1, 2);
```

Содержимое скобок в таком запросе можно заменить на вложенный запрос:

```
SELECT
  id, name, catalog_id
FROM
  products
WHERE
  catalog_id IN (SELECT id FROM catalogs);
```

Оператор **IN** используется, если необходимо применить оператор равенства в отношении множеств. Однако, помимо оператора равенства, могут использоваться другие логические операторы: больше, меньше, больше-равно, меньше-равно. Для реализации таких сравнений используется ключевое слово **ANY**.

Например, давайте выясним, есть ли среди товаров раздела «Материнские платы» товарные позиции, которые дешевле любой позиции из раздела «Процессоры».

```
SELECT
  id, name, price, catalog_id
FROM
  products
WHERE
  catalog_id = 2 AND
  price < ANY (SELECT price FROM products WHERE catalog_id = 1);
```

Мы получили только две позиции. Давайте выведем все:

```
SELECT id, name, price, catalog_id
FROM products
ORDER BY catalog_id, price;
```

Как видим, в результат предыдущего запроса попали позиции 6 и 7, а позиция 5 не попала:

price < ANY (SELECT price FROM products WHERE catalog_id = 1)

4790.00 < 4780.00
4790.00 < 7120.00
4790.00 < 7890.00
4790.00 < 12700.00

19310.00 < 4780.00
19310.00 < 7120.00
19310.00 < 7890.00
19310.00 < 12700.00

Подзапрос возвращает 4 цены из раздела «Процессоры» и сравнивает каждую из цен с этим списком, если хотя бы в одном случае условие срабатывает, ANY-выражение считается истинным. Для ключевого слова **ANY** существует синоним **SOME**:

```
SELECT
  id, name, price, catalog_id
FROM
  products
WHERE
  catalog_id = 2 AND
  price < SOME (SELECT price FROM products WHERE catalog_id = 1);
```

В ключевых словах **ANY** и **SOME** фактически работает логика «или»: если срабатывает хотя бы одно сравнения со множеством значений, выражение считается истинным.

Иногда требуется логика И, когда выражение должно быть истинным, когда все сравнения возвращают истину усли хотя бы одно из сравнению оказалось ложным, весь результат считается ложным. В этом случае используется ключевое слово ALL.

Найдем все товары из раздела «Материнские платы», которые дороже любого товара из раздела «Процессоры»:

```
SELECT
  id, name, price, catalog_id
FROM
  products
WHERE
  catalog_id = 2 AND
  price > ALL (SELECT price FROM products WHERE catalog_id = 1);
```

Мы получаем единственную товарную позицию с идентификатором 5.

price > ALL (SELECT price FROM products WHERE catalog_id = 1)

4790.00 > 4780.00
4790.00 > 7120.00
4790.00 > 7890.00
4790.00 > 12700.00

19310.00 > 4780.00
19310.00 > 7120.00
19310.00 > 7890.00
19310.00 > 12700.00

Каждая цена из раздела «Материнские платы» сравнивается с каждой из цен раздела «Процессоры». Если хотя бы одно из выражений ложное, как в столбце слева, такой товар отбрасывается. В конечную выборку попадают только те товарные позиции, для которых все сравнения истинны.

Результирующая таблица, которая возвращается вложенным запросом, может быть пустой, т. е., не содержать ни одной строки. Для проверки этого используются ключевые слова **EXISTS** и **NOT EXISTS**.

Извлечем те разделы каталога, для которых есть хотя бы одна товарная позиция:

```
SELECT * FROM catalogs
WHERE EXISTS (SELECT * FROM products WHERE catalog_id = catalogs.id);
```

Если вложенный запрос возвращает более одной строки, **EXISTS** возвращает истину. **EXISTS** в действительности не использует результаты вложенного запроса, проверяется только число возвращаемых строк.

Это означает, что в списке столбцов, следующих после ключевого слова **SELECT** вложенного запроса, вместо символа ***** может быть расположено любое допустимое имя — например, просто цифра 1:

```
SELECT * FROM catalogs
WHERE EXISTS (SELECT 1 FROM products WHERE catalog_id = catalogs.id);
```

Такой запрос выполняется гораздо быстрее.

Допускается использование отрицания **NOT EXISTS**. Извлечем каталоги, для которых нет ни одной товарной позиции:

```
SELECT * FROM catalogs
WHERE NOT EXISTS (SELECT 1 FROM products WHERE catalog_id = catalogs.id);
```

До сих пор рассматривались вложенные запросы, возвращающие единственный столбец. Однако в СУБД MySQL реализованы так называемые строчные запросы, которые возвращают более одного столбца.

```
SELECT id, name, price, catalog_id FROM products
WHERE (catalog_id, 5060.00) IN (SELECT id, price FROM catalogs);
```

Выражение в скобках перед **IN** называется конструктором строки, его можно записывать с использованием ключевого слова **ROW**:

```
SELECT id, name, price, catalog_id FROM products
WHERE ROW(catalog_id, 5060.00) IN (SELECT id, price FROM catalogs);
```

Однако, как правило, это ключевое слово **ROW** опускают.

Вложенные запросы возвращают результирующую таблицу, которая становится предметом дальнейших запросов. Стандарт **SQL** разрешает использование вложенных запросов везде, где допускаются ссылки на таблицы. В частности, вложенный запрос может указываться вместо имени таблицы в предложении **FROM.**

Получим товарные позиции из раздела «Процессоры» следующим образом:

```
SELECT * FROM products WHERE catalog_id = 1;
```

Этот запрос может стать своеобразной промежуточной таблицей, например, извлечем среднюю цену по разделу:

```
SELECT
  AVG(price)
FROM
  (SELECT * FROM products WHERE catalog_id = 1) AS prod;
```

Обратите внимание, что в ключевом слове **FROM** мы обязаны назначать вложенному запросу псевдоним при помощи ключевого слова **AS**. Это задачу мы могли бы выполнить и без вложенного запроса:

```
SELECT AVG(price)
FROM products
WHERE catalog_id = 1;
```

Как правило, к вложенным запросам прибегают, когда без них обойтись сложно. Например, если нам требуется вычислить минимальные цены в разделах и получить среднюю минимальную цену. В этом случае мы можем сначала получить минимальные цены разделов:

```
SELECT catalog_id, MIN(price)
FROM products
GROUP BY catalog_id;
```

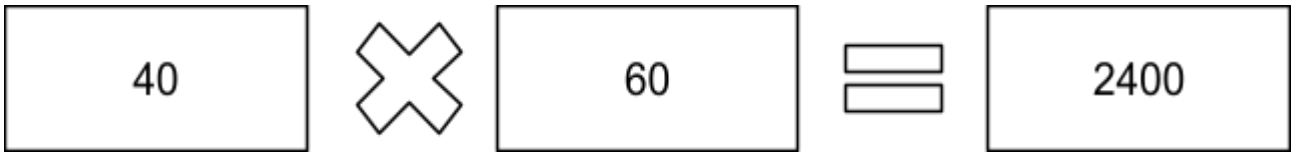
А потом использовать полученную результирующую таблицу в ключевом слове FROM:

```
SELECT
  AVG(price)
```

```
FROM
  (SELECT MIN(price) AS price
   FROM products
   GROUP BY catalog_id) AS prod;
```

JOIN-соединения таблиц

При соединении получается промежуточная таблица, в котором каждая строка одной таблицы объединяется с каждой строкой другой, создавая тем самым все возможные комбинации строк обеих таблиц.



Результирующая таблица содержит число столбцов, равное сумме столбцов в объединяемых таблицах. Если в первой таблице у нас будет 40 строк, а во второй — 60, то результирующая таблица будет содержать 2400 строк.

Создадим две таблицы — **tbl1** и **tbl2**, которые будут содержать единственный столбец **value**:

```
CREATE TABLE tbl1 (
  value VARCHAR(255)
);
INSERT INTO tbl1
VALUES ('fst1'), ('fst2'), ('fst3');

CREATE TABLE tbl2 (
  value VARCHAR(255)
);
INSERT INTO tbl2
VALUES ('snd1'), ('snd2'), ('snd3');
```

Посмотрим содержимое таблиц.

```
SELECT * FROM tbl1;
SELECT * FROM tbl2;
```

Чтобы создать соединение этих двух таблиц, их имена следует перечислить после ключевого слова **FROM** через запятую.

```
SELECT * FROM tbl1, tbl2;
```

Вместо запятой можно использовать ключевое слово **JOIN**:

```
SELECT * FROM tbl1 JOIN tbl2;
```

fst

value
fst1
fst2
fst3

snd

value
snd1
snd2
snd3

```
SELECT
  *
FROM
  fst, snd;
```

```
SELECT
  *
FROM
  fst
JOIN
  snd;
```

value	value
fst1	snd1
fst2	snd1
fst3	snd1
fst1	snd2
fst2	snd2
fst3	snd2
fst1	snd3
fst2	snd3
fst3	snd3

На рисунке видна логика соединения двух таблиц: каждой строке одной таблицы (синий цвет) сопоставляется строка другой таблицы (красный).

У нас в каждой таблице хранится по 3 записи, поэтому результирующая таблица запроса содержит 9 записей. Синим цветом показаны значения из первой таблицы, красным — из второй.

Если мы попробуем явно запросить поле **value**, мы получим сообщение об ошибке:

```
SELECT value FROM tbl1, tbl2;
```

СУБД не может определить, столбец какой таблицы — **tbl1** или **tbl2** — имеется в виду. Чтобы исключить неоднозначность, можно использовать квалификационные имена:

```
SELECT tbl1.value, tbl2.value FROM tbl1, tbl2;
```

Для символа звездочки также можно использовать квалификационное имя:

```
SELECT tbl1.*, tbl2.* FROM tbl1, tbl2;
```

В этом случае будут выводиться столбцы всех соединяемых таблиц. Таблицам можно назначать псевдонимы при помощи ключевого слова **AS**:

```
SELECT t1.value, t2.value FROM tbl1 AS t1, tbl2 AS t2;
```

Такой подход позволяет использовать в качестве имен таблиц более короткие имена. Содержимое промежуточной таблицы можно фильтровать, например при помощи WHERE-условия. В качестве демонстрации попробуем соединить таблицы **catalogs** и **products**:

```
SELECT
  p.name, p.price, c.name
```

```
FROM
  catalogs AS c
JOIN
  products AS p;
```

Нам редко требуется выводить всевозможные комбинации строк соединяемых таблиц. Чаще количество строк в результирующей таблице ограничивается при помощи условия.

```
SELECT
  p.name,
  p.price,
  c.name
FROM
  catalogs AS c
JOIN
  products AS p
WHERE
  c.id = p.catalog_id;
```

При использовании соединения вместо WHERE-условия используется **ON**:

```
SELECT
  p.name,
  p.price,
  c.name
FROM
  catalogs AS c JOIN products AS p
ON
  c.id = p.catalog_id;
```

Разница в том, что ON-условие работает в момент соединения, т. е., у нас промежуточная таблица сразу получается небольшой. WHERE-условие всегда действует после соединения, т. е., сначала получается промежуточная таблица с декартовым произведением исходных таблиц и лишь затем идет фильтрация. Поэтому фильтрацию соединений по возможности следует проводить при помощи ON-фильтрации.

Можно делать запросы с участием одной и той же таблицы, назначая ей разные псевдонимы.

```
SELECT
  *
FROM
  catalogs AS fst
JOIN
  catalogs AS snd;
```

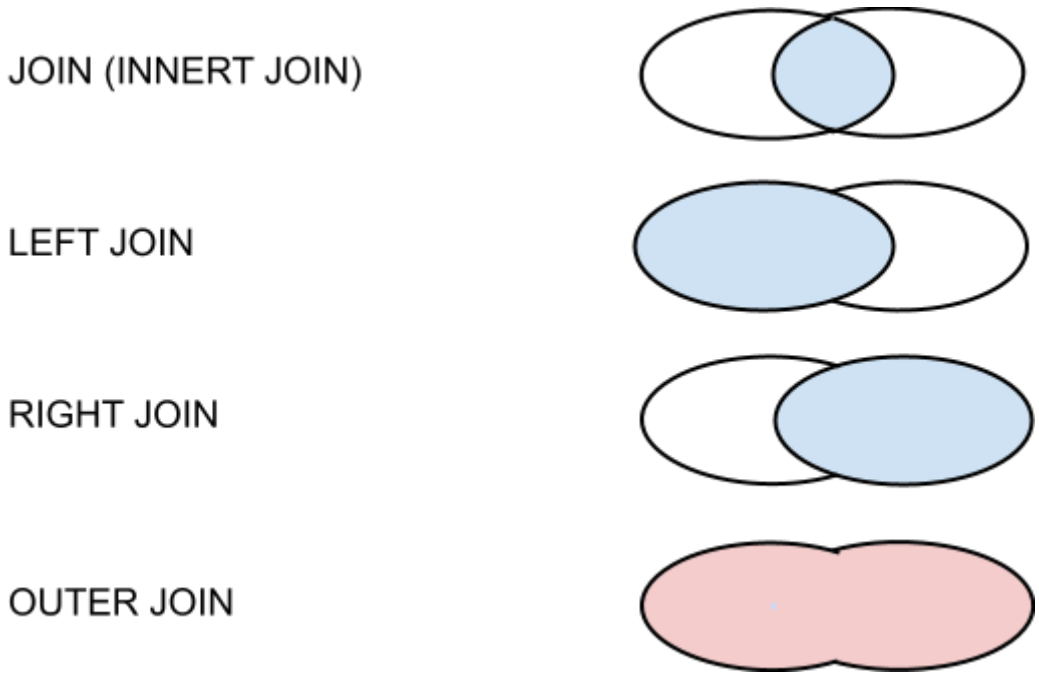
Такие запросы называют самообъединением таблицы. Избавимся от повторов:

```
SELECT
  *
FROM
```

```
  catalogs AS fst
JOIN
  catalogs AS snd
ON
  fst.id = snd.id;
```

Обратите внимание, что названия столбцов в ON-условии совпадают. Различаются только названия таблиц. В этом случае допускается использования ключевого слова **USING**:

```
SELECT
  *
FROM
  catalogs AS fst
JOIN
  catalogs AS snd
USING(id);
```



При использовании условий у нас появляется ограничение, согласно которому строки одной таблицы сопоставляются строкам другой. Существует несколько режимов соединения по условию.

Без дополнительных ключевых слов **JOIN** осуществляет перекрестное соединение таблиц, если для записи одной таблицы отсутствует сопоставление в другой таблице. Такая запись отбрасывается.

LEFT JOIN и **RIGHT JOIN** осуществляют левое и правое соединение, в результирующей таблице присутствуют все записи левой или правой таблицы, даже если им нет подходящего сопоставления.

CROSS JOIN производит соединения записей обеих таблиц, даже если нет подходящего сопоставления. К сожалению, этот тип соединения не поддерживается MySQL. Таким образом, в MySQL различают только три типа соединений.

```
SELECT
```

```
  p.name,
  p.price,
  c.name
FROM
  catalogs AS c
JOIN
  products AS p
ON
  c.id = p.catalog_id;
```

Давайте рассмотрим **LEFT JOIN** на примере соединения таблиц **catalogs** и **products**:

```
SELECT * FROM catalogs;
```

В таблице **catalogs** у нас три записи, для раздела «Видеокарты» сопоставления в таблице **products** нет, поэтому эта запись не попадает в результирующую таблицу JOIN-соединения:

```
SELECT
  p.name,
  p.price,
  c.name
FROM
  catalogs AS c
LEFT JOIN
  products AS p
ON
  c.id = p.catalog_id;
```

Однако, если мы заменим **JOIN** на **LEFT JOIN**, в результате появляется раздел «Видеокарты», несмотря на то, что для него нет соответствующих строк в таблице **products**. Недостающие поля заполняются неопределенным значением **NULL**. Порядок таблиц имеет значение, таблица **catalogs** должна располагаться слева от **LEFT JOIN**. Если мы поменяем местами таблицы **catalogs** и **products**, для получения такого результата нам потребуется использовать соединение RIGHT JOIN:

```
SELECT
  p.name,
  p.price,
  c.name
FROM
  products AS p
RIGHT JOIN
  catalogs AS c
ON
  c.id = p.catalog_id;
```

Многотабличные запросы можно использовать не только для извлечения, но и для обновления данных, например, если мы захотим снизить цены на 10 % для материнских плат, мы можем воспользоваться следующим UPDATE-запросом:

```
UPDATE
  catalogs
JOIN
  products
ON
  catalogs.id = products.catalog_id
SET
  price = price * 0.9
WHERE
  catalogs.name = 'Мат.платы';
```

Схожим образом действует и многотабличное удаление. Однако в нем необходимо явно указать, из каких таблиц мы будем удалять записи:

```
DELETE
  products, catalogs
FROM
  catalogs
JOIN
  products
ON
  catalogs.id = products.catalog_id
WHERE
  catalogs.name = 'Мат.платы';
```

Таким запросом мы удалили 4 записи: 3 товара из таблицы **products** и один раздел «Материнские платы» из таблицы **catalogs**.

Если мы не хотим удалять из таблицы **catalogs** записи, то после ключевого слова **DELETE** мы должны указать только одну таблицу **products.** Давайте удалим процессоры

```
DELETE
  products
FROM
  catalogs
JOIN
  products
ON
  catalogs.id = products.catalog_id
WHERE
  catalogs.name = 'Процессоры';
```

В таблице **products** у нас должны исчезнуть все записи. При этом таблица **catalogs** должна остаться нетронутой.

Внешние ключи и ссылочная целостность

В SQL довольно много механизмов поддержания целостности данных. Одним из самых важных механизмов является ограничении внешнего ключа.

У нас таблицы **catalogs** и **products** связаны отношением «один ко многим». Одному каталогу могут соответствовать множество товарных позиций, в то время как у каждой товарной позиции может быть только один каталог.

Для такой связи в таблице **products** в поле **catalog_id** мы храним значение первичного ключа из таблицы **catalogs**.

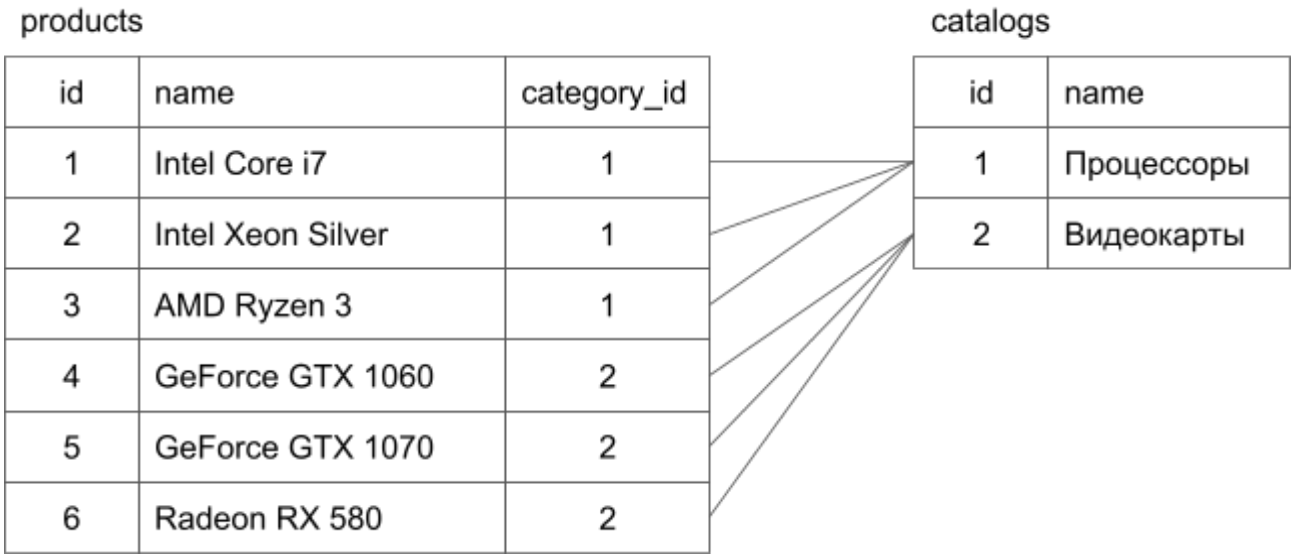
Давайте представим, что мы удаляем каталог с идентификатором 1:

```
DELETE FROM catalogs WHERE id = 1;
SELECT id, name, price, catalog_id FROM products;
```

При этом в таблице **products** остаются записи, которые соответствуют данному каталогу. Таким образом, запрос на удаление из таблицы **catalogs** привел к тому, что база данных перестала быть согласованной. Мы нарушили целостность данных.

При удалении таблицы **catalogs** нам необходимо помнить, что требуется внести изменения в таблицу **products**. Например, удалить записи с внешним ключом, который ссылается на несуществующую запись или, как вариант, присвоить внешнему ключу значение **NULL.**

СУБД не знает, какую стратегию вы предпочитаете, поэтому по умолчанию не делает ничего, оставляя решение вопроса на откуп разработчику. У нас довольно простая база данных и помнить о необходимости обновления таблицы **products** не сложно. Однако база данных может содержать десятки и сотни таблиц, которые ссылаются на **catalogs**. Даже если мы все их помним, появляется очень высокая вероятность, что мы ошибемся и забудем выполнить запрос к одной из баз данных. Даже если мы выполним все запросы правильно, на момент выполнения запросов нарушается целостность данных.



Для решения такого рода проблем в SQL предназначено ограничение внешнего ключа. В рамках реляционной модели таблицу с первичным ключом **catalogs** называют предком, а таблицу **catalogs** — потомком:

products

id	name	category_id
1	Intel Core i7	1
2	Intel Xeon Silver	1
3	AMD Ryzen 3	1
4	GeForce GTX 1060	2
5	GeForce GTX 1070	2
6	Radeon RX 580	2

catalogs

id	name
1	Процессоры
2	Видеокарты

У нас возможны две ситуации, которые могут приводить к нарушению целостности данных: удаление строки-предка и обновление первичного ключа в строке-предке.

Чтобы задать реакцию на эти ситуации, в таблицу добавляется внешний ключ, ограничение которого задается ключевым словом **FOREIGN KEY**.

```
FOREIGN KEY [name_key] (col1, ...) REFERENCES tbl (tbl_col, ...)
[ON DELETE {CASCADE|SET NULL|NO ACTION|RESTRICT|SET DEFAULT}]
[ON UPDATE {CASCADE|SET NULL|NO ACTION|RESTRICT|SET DEFAULT}]
```

После ключевого слова **FOREIGN KEY** указывается название ключа и в скобках столбцы, которые играют роль внешнего ключа. После ключевого слова **REFERENCES** указывается имя таблицы и в скобках столбцы, которые играют роль первичного ключа.

Необязательные конструкции **ON DELETE** и **ON UPDATE** позволяют задать поведение СУБД при удалении и обновлении строк из таблицы-предка, соответственно.

После ключевых слов **ON DELETE** и **ON UPDATE** указывается, какое действие нужно предпринять при выполнении **DELETE** и **UPDATE** запросов. Всего предусмотрено пять режимов:

- CASCADE
- SET NULL
- NO ACTION
- RESTRICT
- SET DEFAULT

При использовании ключевого слова **CASCADE** при обновлении или удалении записей в таблице-предке, соответствующие записи в таблице-потомке удаляются или обновляются автоматически.

В случае **SET NULL** при удалении или обновлении записи, содержащей первичный ключ, в таблице-потомке значения устанавливаются в **NULL**.

При использовании действия **NO ACTION** при удалении или обновлении записей, содержащих первичный ключ, с таблицей-потомком никаких дополнительных действий не производится. Мы просто обозначаем логическую связь таблиц, не вводя ограничений.

Ограничение **RESTRICT** приводит к тому, что если в таблице-потомке имеются записи, ссылающиеся на первичный ключ таблицы-предка, при удалении или обновлении записей с таким первичным ключом возвращается ошибка.

СУБД не позволяет изменять или удалять запись с первичным ключом, пока не останется ни одной ссылки из таблицы-потомка.

Последнее ключевое слово **SET DEFAULT** очень похоже на **SET NULL**, только вместо **NULL** устанавливается DEFAULT-значение столбца.

Давайте посмотрим на эти случаи на практике. Воспользуемся оператором **SHOW CREATE TABLE**, чтобы посмотреть структуру таблицы **products**:

```
SHOW CREATE TABLE catalogs\G
```

Итак, в таблице у нас есть внешний ключ **catalog_id**, предлагаю добавить ограничение внешнего ключа. Для этого можно воспользоваться **ALTER TABLE**:

```
ALTER TABLE products
ADD FOREIGN KEY (catalog_id)
REFERENCES catalogs (id)
ON DELETE NO ACTION
ON UPDATE NO ACTION;
```

Мы получаем ошибку, связана она с тем, что типы первичного ключа catalogs и products отличаются. Все идентификаторы у нас имеют тип **BIGINT**, в то время как внешние ключи имеют тип **INT**.

Давайте исправим тип у внешнего ключа **catalog_id**:

```
ALTER TABLE products
CHANGE catalog_id catalog_id BIGINT UNSIGNED DEFAULT NULL;
```

Попробуем добавить внешний ключ повторно:

```
ALTER TABLE products
ADD FOREIGN KEY (catalog_id)
REFERENCES catalogs (id)
ON DELETE NO ACTION
ON UPDATE NO ACTION;
```

Ключ добавлен. Давайте посмотрим на структуру таблицы:

```
SHOW CREATE TABLE products\G
```

Итак, у нас появилось ограничение внешнего ключа, которое не предпринимает пока никаких действий. Обратите внимание, что имя ключу назначено автоматически **products_ibfk_1**. Это имя нам может пригодиться, если мы захотим удалить ограничение из таблицы:

```
ALTER TABLE products
DROP FOREIGN KEY products_ibfk_1;
```

Имя для ограничения **FOREIGN KEY** мы можем задавать и явно. Для этого мы можем его указать после необязательного ключевого слова **CONSTRAINT**:

```
ALTER TABLE products
ADD CONSTRAINT fk_catalog_id
FOREIGN KEY (catalog_id)
REFERENCES catalogs (id)
ON DELETE NO ACTION
ON UPDATE NO ACTION;
```

Например, тут мы назначаем в качестве имени **fk_catalog_id**. Давайте убедимся, что ограничение успешно добавлено в таблицу:

```
SHOW CREATE TABLE products\G
```

Теперь, если мы захотим изменить или удалить ограничение, мы можем воспользоваться нашим собственным именем **fk_catalog_id**, а не назначенным СУБД MySQL. Действие **NO ACTION** не очень интересно по своему функционалу: что бы ни происходило со связанными таблицами, ограничение не будет срабатывать.

Давайте добавим ограничение **CASCADE**, которое позволяет каскадно удалять и обновлять данные. Опять воспользуемся **ALTER TABLE**.

```
ALTER TABLE products
DROP FOREIGN KEY fk_catalog_id;
```

Удалим ограничение **fk_catalog_id** и создадим его по новой, указав каскадный режим для операций удаления и обновления:

```
ALTER TABLE products
ADD CONSTRAINT fk_catalog_id
FOREIGN KEY (catalog_id)
REFERENCES catalogs (id)
ON DELETE CASCADE
ON UPDATE CASCADE;

SELECT * FROM catalogs;
```

Итак, у нас есть три каталога, давайте изменим первичный ключ для Процессоров с 1 на 4:

```
SELECT id, name, price, catalog_id FROM products;
```

При этом у нас в таблице **products** имеются внешний ключ **catalog_id**, который ссылается на первичный ключ таблицы **products**. Так как у нас включен каскадный режим обновления, у нас должны обновиться ключи в обеих таблицах.

```
UPDATE catalogs SET id = 4 WHERE name = 'Процессоры';
```

Давайте убедимся, что изменения внесены.

```
SELECT * FROM catalogs;
SELECT id, name, price, catalog_id FROM products;
```

Мы обновили запись в **catalogs**, а изменения каскадно отразились на таблице **products**. Если мы сейчас удалим раздел «Процессоры», все товары из таблицы products тоже будут удалены.

```
DELETE FROM catalogs WHERE name = 'Процессоры';

SELECT * FROM catalogs;
SELECT id, name, price, catalog_id FROM products;
```

Раздел «Процессоры» и все относящиеся к нему товары удалены. Давайте посмотрим, как работает поле ограничения **SET NULL**. Давайте удалим текущее ограничение.

```
ALTER TABLE products
DROP FOREIGN KEY fk_catalog_id;
```

И добавим новое:

```
ALTER TABLE products
ADD CONSTRAINT fk_catalog_id
FOREIGN KEY (catalog_id)
REFERENCES catalogs (id)
ON DELETE SET NULL;
```

Обратите внимание, что мы можем устанавливать только одно ограничение, например на удаление, а для обновления задать совершенно другой тип ограничения.

Давайте удалим раздел видеокарт:

```
DELETE FROM catalogs WHERE name = 'Мат.платы';
```

Посмотрим на результаты:

```
SELECT * FROM catalogs;
SELECT id, name, price, catalog_id FROM products;
```

Обратите внимание, что внешний ключ получил значения **NULL**. Ограничение первичного ключа — далеко не единственный механизм поддержания целостности данных.

Часть механизмов, которые описывает стандарт SQL просто не реализованы в MySQL, например, CHECK-ограничения, Часть поддерживается очень хорошо и будет рассмотрена на следующих уроках.

Используемые источники

1. https://dev.mysql.com/doc/refman/5.7/en/union.html
2. https://dev.mysql.com/doc/refman/5.7/en/subqueries.html
3. https://dev.mysql.com/doc/refman/5.7/en/join.html
4. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.
5. Гровф, Джеймс Р., Вайнберг, Пол Н., Оппель, Эндрю Дж. SQL: полное руководство, 3-е изд. : Пер. с англ. — М.: ООО "И.Д. Вильямс", 2015. — 960 с.
6. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. — Пер. с англ. — СПб.: Символ-Плюс, 2010. — 480 с.
7. Кузнецов М.В., Симдянов И.В. MySQL на примерах. — СПб.: БХВ-Петербург, 2007. — 592с.
8. Кузнецов М.В., Симдянов И.В. MySQL 5. — СПб.: БХВ-Петербург, 2006. — 1024с.
9. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.
10. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.