MySQL

ВЫБОРКА, СОРТИРОВКА И ФИЛЬТРАЦИЯ ДАННЫХ

СОДЕРЖАНИЕ

ОСНОВЫ ВЫБОРКИ ДАННЫХ: ОПЕРАТОР SELECT

ВВЕДЕНИЕ

- Операция выборки данных фактически является самой главной из всех четырех операций в контексте СУБД MySQL (напомним эти операции - выборка, вставка, обновление и удаление). Само собой, можно возразить, что перед тем, как данные извлечь из таблиц, их сначала надо туда вставить - поэтому возможно стоило начать с операции вставки. Однако как мы дальше увидим, операции выборки активно участвуют во всех других операциях, Поэтому начнем мы именно с получения данных, т.е. с выборки.

- Главный и единственный оператор, который непосредственно отвечает за выборку данных в MySQL (как и во всех остальных реляционных СУБД), носит название **SELECT**. Сразу оговоримся, и отметим, что этот оператор может не только извлекать данные из таблиц баз данных, но и отображать результаты математических операций, выводить строки и числа, обработанные специальными функциями MySQL, а также показывать дату и время (о функциях - в особенности даты и времени - мы поговорим подробнее несколько позже). В качестве результата **SELECT** всегда возвращает данные в виде таблицы - даже если было запрошено какое-либо одно значение (как мы уже говорили, это одна из основных особенностей реляционных баз данных).

- Пример запроса результата простого математического выражения выглядит следующим образом:

```
SELECT 2 + 2 * 2;
```

  Как результат получится таблица из одного столбца и одной строки (если не учитывать строку с названием столбца), а названием столбца послужит само выражение:

2 + 2 * 2
6

- Если нам необходимо узнать несколько результатов выражений сразу, то мы можем сделать запрос, перечислив эти выражения

через запятую (дадим небольшое пояснение - функция **SQRT** требует в качестве единственного аргумента какое-либо число и возвращает корень из него, функция **POW** требует в качестве первого аргумента опять некое число, а в качестве второго аргумента степень, в которую мы будем возводить первый аргумент):

```
SELECT SQRT(4), 1 + 2, POW(3, 3);
```

В результате мы получим таблицу из трех столбцов и одной строки (если не учитывать строку с названиями столбцов):

SQRT(4)	1 + 2	POW(3, 3)
2	3	27

- Иногда в качестве названий столбцов нам могут понадобиться не конкретные выражения, а некие абстрактные названия операций - псевдонимы (например, не “1 + 2”, а “addition”). Это можно сделать двумя способами - с использованием ключевого слова **AS** и без него (**square root** взят в обратные кавычки для того, чтобы дать понять MySQL что это единое название):

```
-- 1-й способ
SELECT
    SQRT(4) AS `square root`,
    1 + 2 AS addition,
    POW(3, 3) AS exponentiation;

-- 2-й способ
SELECT
    SQRT(4) `square root`,
    1 + 2 addition,
    POW(3, 3) exponentiation;
```

В обоих случая мы получим одну и ту же таблицу:

square root	addition	exponentiation
2	3	27

РАБОТА С ТАБЛИЦАМИ

- Применение **SELECT** в контексте одиночных значений является интересной темой, но все же главное назначение **SELECT** - делать выборку из таблиц баз данных. Рассмотрим такую выборку подробнее. Сначала предположим, что у нас есть база данных *selection*, в которой существует таблица products:

```
CREATE TABLE products(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    ean VARCHAR(255),
    price DECIMAL(20,2),
    created_at DATE NOT NULL
);
```

- Чтобы таблица не была пустой, заполним ее данными (это несколько опережает то, что пока рассмотрели, но в данный момент у нас нет другого выхода) - для этого выполним следующую команду:

```
INSERT INTO products(name, ean, price, created_at)
VALUES
('iPhone', '74748484', 999.99, '2023-01-01'),
('Samsung', '9393983', 1099.99, '2023-02-02'),
('Xiaomi', '35333653', 899.99, '2023-03-03');
```

- Чтобы получить все строки и все колонки данной таблицы, мы должны выполнить следующий запрос с оператором **SELECT**:

```
SELECT * FROM products;
```

  В этом запросе оператор звездочка (*****) говорит MySQL о том, что мы хотим получить значения всех столбцов в том порядке, в каком они были объявлены при создании таблицы, оператор **FROM** обозначает, что мы хотим получить значение из таблицы, название которой указано после этого оператора (*products*). Если предположить, что в таблице уже было сохранено три уже упомянутых продукта, мы получим следующий результат (важно отметить, что в отличие от порядка столбцов, порядок строк не гарантирован - чтобы выводить строки в строго заданном порядке используются другие операторы, которые мы обсудим несколько позже):

id	name	ean	price	created_at
1	iPhone	74748484	999.99	2023-01-01
2	Samsung	9393983	1099.99	2023-02-03
3	Xiaomi	35333653	899.99	2023-03-03

Тот же самый результат мы могли бы получить, если бы явно запросили каждый столбец, перечислив их все через запятую:

```
SELECT id, name, ean, price, created_at FROM products;
```

- Если нам необходимо получить столбцы не в том порядке, в каком они были объявлены при создании таблицы, то оператор звездочка уже не подходит - нам ничего не остается, как только перечислить их через запятую:

```
SELECT ean, price, created_at  name, id FROM products;
```

ean	price	created_at	name	id
74748484	999.99	2023-01-01	iPhone	1
9393983	1099.99	2023-02-03	Samsung	2
35333653	899.99	2023-03-03	Xiaomi	3

- Было бы наивно полагать, что нам всегда понадобятся значения всех столбцов - иногда нам нужно получить только два или три столбца из пяти. Это сделать просто - надо указать после SELECT только необходимые столбцы (например, name и price):

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

Будет возвращена следующая таблица:

name	price
iPhone	999.99
Samsung	1099.99
Xiaomi	899.99

- Выборки в таблицах, как и выборки в рамках простых значений (без таблиц), поддерживают псевдонимы - легко понять, что данные псевдонимы нужно использовать для названий столбцов (это может быть необходимо, например, в том случае, если в приложении жестко зафиксированы названия получаемых значений, и эти названия по какой-то причине не совпадают с названиями в базе данных). Присвоение псевдонимов может происходить как с ключевым словом **AS**, так и без него:

```
SELECT name AS title, ean code, price total FROM products;
```

Заменив name на title (с применением AS), ean на code, а price - на total, мы получим следующий результат:

title	code	total
iPhone	74748484	999.99
Samsung	9393983	1099.99
Xiaomi	35333653	899.99

- При выборке можно менять не только результирующие названия столбцов, их количество и порядок. Важной особенностью оператора **SELECT** является возможность на лету менять значения столбцов (в базе данных они останутся неизменными, а в результате мы отобразим по-другому). Например, предположим, что у нас временно повысились цены на все продукты в два раза - мы не хотим менять их в таблице, но хотим их отображать в увеличенном виде. Для этого подойдет следующий запрос:

```
SELECT id, name, price * 2 FROM products;
```

Результат:

id	name	price * 2
1	iPhone	1999.98
2	Samsung	2199.98
3	Xiaomi	1799.98

- Еще одной интересной особенностью оператора **SELECT** является добавлять при выборке из таблиц несуществующие столбцы. В предыдущем примере мы просто меняли значение столбца price, а теперь оставим его без изменения, но затем запросим его вновь, но уже в увеличенном виде:

```
SELECT id, name, price, price * 2 FROM products;
```

Полученный результат:

id	name	price	price * 2
1	iPhone	999.99	1999.98
2	Samsung	1099.99	2199.98
3	Xiaomi	899.99	1799.98

- Кроме математических операций, мы также можем применять к значениям столбцов таблицы специальные функции MySQL. Например, в рекламных целях необходимо узнать, как длина названия продукта влияет на его продажи - для этого следует использовать функцию **CHAR_LENGTH**, которая принимает в качестве аргумента любой текст, а возвращает количество символов в этом тексте:

```
SELECT id, CHAR_LENGTH(name), price FROM products;
```

Полученная выборка выглядит следующим образом (при желании на столбец длины строки - **CHAR_LENGTH(name) -** можно было бы поставить какой-либо эстетически благозвучный псевдоним, например, name_size, но в этот раз обойдемся тем, что есть):

id	CHAR_LENGTH(name)	price
1	6	999.99
2	7	1099.99
3	6	899.99

- Мы уже видели, как **SELECT** можем видоизменять результат столбца, а также добавлять любые новые столбцы. Еще одной

возможностью можно назвать объединение значений нескольких столбцов в один столбец. Например, новое начальство решило, что отныне в название должен обязательно входить код **ean**. Чтобы не ломать уже существующую структуру, просто применим к запросу функцию **CONCAT**, которая принимает сколь угодно аргументов, а возвращает их соединенными вместе (в данном случае применен псевдоним full_name, но можно обойтись и без него):

```
SELECT id, CONCAT(name, ' (', ean, ')') AS full_name, price
FROM products;
```

Результат:

id	full_name	price
1	iPhone (74748484)	999.99
2	Samsung (9393983)	1099.99
3	Xiaomi (35333653)	899.99

- Возможно, что мы еще помним коварное специальное значение **NULL** - если мы посмотрим на таблицу products, то заметим, что столбцы name, ean и price не обязательно должны быть заполнены, т.е. потенциально туда можно этот **NULL** записать. Чтобы отследить такое положение дел для, например, столбца name, можно применить специальную конструкцию **IFNULL**. Если name будет равняться **NULL**, мы будем выводить какое-то значение по умолчанию, если нет - показывать оригинальное имя (применен псевдоним name):

```
SELECT id, IFNULL(name, 'default name') name, price
FROM products;
```

В нашем случае таблица вернется в неизменном виде, так как у нас нет значений **NULL** в столбце name, но если бы были, то вместо них был бы вставлен текст 'default name'.

- Кроме конструкции **IFNULL**, есть более универсальная конструкция **IF**, которая принимает три значения - в-первых, некое выражение, которое может быть истинным или ложным, во-вторых, значение, которое будет выведено при истинности первого выражения и, в-третьих, значение, которое будет

выведено при ложности первого выражения. Например, мы хотим вывести несуществующий столбец status, который показывает, дорогой у нас продукт или дешевый (продукт дороже 1000 будет считаться дешевым) - для этого **IF** подходит идеально:

```
SELECT id, name, price, IF(price > 1000, 'expensive',
'cheap') status FROM products;
```

Результат:

id	name	price	status
1	iPhone	999.99	cheap
2	Samsung	1099.99	expensive
3	Xiaomi	899.99	cheap

- Иногда конструкции IF бывает недостаточно - представим, что у нас должно быть не два статуса, а три - продукты дороже 900, но дешевле или равные 1000 должны иметь статус normal. Для решения этой проблемы подходит несколько необычная конструкция **CASE ... WHEN ... THEN ... END:**

```
SELECT
    id,
    name,
    price,
    CASE
    WHEN price > 1000 THEN 'expensive'
    WHEN price <= 1000 AND price > 900 THEN 'normal'
    ELSE 'cheap'
    END AS status
FROM products;
```

Результат:

id	name	price	status
1	iPhone	999.99	normal
2	Samsung	1099.99	expensive
3	Xiaomi	899.99	cheap

СОРТИРОВКА ДАННЫХ: ОПЕРАТОР ORDER BY

- Как уже было сказано выше, по умолчанию MySQL не гарантирует порядок строк в результате выполнения операций выборки (хотя в подавляющем большинстве случаев строки будут возвращены в том порядке, в каком они были вставлены в таблицу). Однако если мы укажем явно порядок сортировки строк, то полученная выборка будет гарантированно отсортирована необходимым для нас способом.

- Сортировка осуществляется при помощи оператора **ORDER BY**, после которого указываются столбец (или столбцы), по которым должна идти сортировка. Кроме того, каждому столбцу можно явно указывать порядок, в котором будет происходить сортировка (от большего к меньшему или наоборот). По умолчанию при использовании оператора **ORDER BY** MySQL выполняет сортировку от меньшего к большему, но это можно указать явно, поставив после столбца оператор **ASC**. Если мы желаем сортировку от большего к меньшему, то после столбца надо поставить оператор **DESC**.

- Попробуем отсортировать наши продукты из прошлого раздела по столбцу цены - от меньшей цены к большей. Для тренировки явно укажем порядок сортировки **ASC**, но, как уже говорилось, при таком порядке его можно и не указывать:

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

Как результат получим отсортированную по цене таблицу:

id	name	price
3	Xiaomi	↓ 899.99
1	iPhone	↓ 999.99
2	Samsung	↓ 1099.99

- Далее применим обратную сортировку по цене при помощи оператора DESC:

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

Закономерный результат - строки выведены в обратном в контексте цены порядке:

id	name	price
2	Samsung	↑ 1099.99
1	iPhone	↑ 999.99
3	Xiaomi	↑ 899.99

- Важно заметить, что MySQL разрешает осуществлять сортировку даже по тем столбцам, которые не участвуют в выборке - в примере попробуем отсортировать по цене, но не получать ее:

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

Цена действительно будет отсутствовать, но сортировка будет произведена правильно:

id	name
3	Xiaomi
1	iPhone
2	Samsung

- До этого мы производили сортировку только по числовым значениям, но MySQL также умеет сортировать и по текстовым столбцам, однако делает это в **лексикографическом порядке**. Символ (буква или цифра), который идет в рамках своего алфавита после какого-либо другого символа, имеет больший “вес”, чем предыдущий символ. Это означает, что А меньше чем Ж, а '2' меньше, чем '5'. Кроме того, латинские буквы всегда будут идти перед кириллицей. Важный момент - при упомянутой лексикографической сортировке сравнение происходит посимвольно, т.е. в тот момент, когда символ в одной строке меньше символа на этой же самой позиции в другой строке, то первая строка считается имеющей “меньший вес”, а дальнейшее сравнение прекращается. Проиллюстрируем это на примере сортировки по столбцу ean, который является текстовым, но в нашем случае хранит лишь числовые значения:

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

Результат будет несколько неожиданный, но закономерный - самый маленький ean - 9393983 - MySQL посчитает самым большим, т.к. он начинается с '9', а остальные хотя и имеют большее значение, начинаются '3' и '7' соответственно:

id	name	price	ean
3	Xiaomi	899.99	↓ 35333653
1	iPhone	999.99	↓ 74748484
2	Samsung	1099.99	↓ 9393983

- А что, если нам надо, чтобы текстовые строки сортировались в “нормальном” порядке? Решение для этого есть, но в данном случае нам надо сначала отсортировать ean по его длине, т.е. сначала мы гарантированно получим порядок, в котором строки с меньшим количеством символов будут стоять вначале, а затем произведем стандартную сортировку по самому **ean** (т.е. среди строк с одинаковым количеством символов):

```
SELECT
    id, name, price, ean
FROM products
ORDER BY CHAR_LENGTH(ean), ean;
```

Мы применили функцию CHAR_LENGTH, чтобы узнать количество символов (это показывает, что при сортировке можно также использовать и функции) и получили нужный нам результат:

id	name	price	ean
2	Samsung	1099.99	↓ 9393983
3	Xiaomi	899.99	↓ 35333653
1	iPhone	999.99	↓ 74748484

- В прошлом примере мы видели, как сортировка происходила сразу по двум столбцам - по некоему виртуальному столбцу, которого нет в таблице, а также по столбцу ean. Рассмотрим подробнее логику сортировки в таких случаях (сортировка по

двум или более столбцам). Чтобы это осуществить, сначала занесем в таблице еще две записи, где опять фигурирует продукт Samsung (но уже с большей ценой), а также продукт Xiaomi (с меньшей ценой):

```
INSERT INTO products(name, ean, price, created_at)
VALUES
('Samsung', '5394983', 1199.99, '2023-04-04'),
('Xiaomi', '65322653', 799.99, '2023-05-05');
```

Далее нам необходимо получить выборку продуктов таким образом, чтобы названия шли в алфавитном порядке (от меньшего к большего), а цены в обратном порядке (от большего к меньшего). Для этого применим сортировку по двум столбцам - названию и цене:

```
SELECT id, name, price, ean
FROM products
ORDER BY name ASC, price DESC;
```

Произошло следующее - MySQL отсортировал сначала имена в алфавитном порядке и получил три группы - с именами iPhone (1 элемент), Samsung (2 элемента) и Xiaomi (2 элемента). Затем в контексте каждой группы произошла сортировка по цене в обратном порядке. Продукты в одной группе не могли “перепрыгнуть” в другую группу, они могли поменять свое положение только в рамках своей группы с одинаковым именем:

id	name	price	ean
1	↓ iPhone	↑↑ 999.99	74748484
4	↓ Samsung	↑↑ 1199.99	5394983
2	↓ Samsung	↑↑ 1099.99	9393983
3	↓ Xiaomi	↑↑ 899.99	35333653
5	↓ Xiaomi	↑↑ 799.99	65322653

- Еще одной особенностью оператора **ORDER BY** можно назвать возможность сортировать не по имени столбца, а по его номеру в контексте выборки. Например, мы хотим получить выборку с участием трех столбцов - **id**, **price**, **name**. Мы можем указать

name в качестве того поля, по которому мы будет сортировать, а можем указать номер 3. Понятно, что в данном случае нельзя сортировать по тем полям, которые не запрашиваются - у них нет номера, кроме того, к номерам нельзя применять функции.

```
SELECT id, price, name FROM products ORDER BY 3 ASC;
```

id	price	name
1	999.99	iPhone
2	1099.99	Samsung
4	1199.99	Samsung
3	899.99	Xiaomi
5	799.99	Xiaomi

- В завершение раздела кратко коснемся темы верхнего и нижнего регистра (большие и маленькие буквы) при сортировке. Дело в том что он зависит только от атрибута **COLLATE** (сопоставление) который всегда явно или неявно следует за объявлением кодировки базы данных, таблицы или столбца. По умолчанию основная кодировка **utf8mb4** имеет **COLLATE** равное **utf8mb4_0900_ai_ci**, которое не учитывает регистр при сравнении строк (т.е. считает 'A' и 'a' одинаковыми). Чтобы регистр учитывался, можно ставить **utf8mb4** сущностям в качестве сопоставления **utf8mb4_0900_as_cs**:

```
CREATE DATABASE example DEFAULT CHARSET utf8mb4 COLLATE
utf8mb4_0900_as_cs;

CREATE TABLE example (
    name VARCHAR(255) CHARACTER SET utf8mb4 COLLATE
utf8mb4_0900_as_cs
) DEFAULT CHARSET utf8mb4 COLLATE utf8mb4_0900_as_cs;
```

Если мы не хотим менять сопоставление в сущностях, то можем это сделать на лету в момент выборки:

```
SELECT id, name FROM products ORDER BY name COLLATE
utf8mb4_0900_as_cs;
```

ФИЛЬТРАЦИЯ ДАННЫХ ПРИ ВЫБОРКЕ

ВЫБОРКА УНИКАЛЬНЫХ СТРОК

- Если мы захотим узнать только имена продуктов из таблицы предыдущей главы, то мы должны следующий запрос:

```
SELECT name FROM products;
```

Мы получим таблицу из пяти строк, но в этой таблице значения Samsung и Xiaomi будут повторяться дважды:

name
iPhone
Samsung
Xiaomi
Samsung
Xiaomi

Однако иногда могут потребоваться только уникальные значения (т.е. значения без повторений). Можно решить эту проблему с помощью оператора **DISTINCT**, который должен располагаться после оператора **SELECT** и перед названиями столбцов. Оператор **DISTINCT** гарантирует, что будут возвращены только те строки, комбинации значений столбцов которых уникальны. Если у нас запрашивается имя, то вернутся только уникальные имена, если же запрашивается имя и цена, а у нас несколько строк с одинаковой именем и ценой - то вернется только одна строка с таким именем и ценой. Решим же нашу проблему с именем:

```
SELECT DISTINCT name FROM products;
```

name
iPhone
Samsung
Xiaomi

- В завершение этой подглавы заметим, что на работу оператора **DISTINCT** влияют уже упоминавшиеся сопоставления (**COLLATE**). По умолчанию у нас не учитывается регистр, но если же мы хотим, чтобы он учитывался в рамках запроса на уникальную выборку, то можно сделать это следующим образом (подставим сопоставление после названия столбца):

```
SELECT
    DISTINCT name COLLATE utf8mb4_0900_as_cs AS name
FROM products;
```

ОГРАНИЧЕНИЕ КОЛИЧЕСТВА СТРОК

- Очень часто нам надо получить не все строки в таблице, а только какую-то часть - например, первые две строки. Для задач такого рода предусмотрен специальный оператор **LIMIT** - воспользуемся им для решения нашей задачи:

```
SELECT id, name, price FROM products LIMIT 2;
```

Как и было запланировано, получим только первые две строки:

id	name	price
1	iPhone	999.99
2	Samsung	1099.99

Однако мы помним, что MySQL без явной сортировки может вернуть строки таблицы в любом порядке. Чтобы это пресечь, воспользуемся оператором **ORDER BY** и отсортируем строки по цене в возрастающем порядке, а уже после этого получим первые две строки. Перефразируя предыдущее предложение, можно сказать, что мы хотим получить два наиболее дешевых продукта. Дополнительно стоит упомянуть, что при таких случаях (когда вместе используются сортировка и лимитирование) сначала будет происходить сортировка, а только потом лимитирование количества строк.

```
SELECT id, name, price FROM products ORDER BY price LIMIT 2;
```

Результат - Xiaomi оказались самыми дешевыми продуктами:

id	name	price
5	Xiaomi	799.99
3	Xiaomi	899.99

- Очень часто, например при создании постраничной навигации, нам необходимо получить строки не с самого начала, а с каким-либо отступом - не первые две строки, а две строки, начиная с четвертой (т.е. пропустив первые три строки). Чтобы это реализовать, в MySQL есть оператор **OFFSET**, который всегда работает в связке с **LIMIT**. После **OFFSET** мы указываем количество строк, которые мы хотим пропустить. Также существует альтернативный синтаксис в контексте самого **LIMIT** - после него можно указать через запятую две цифры - первая означает количество строк, которое мы хотим пропустить, а вторая цифра - количество строк, которое мы хотим получить.

```
-- 1-й вариант
SELECT
    id, name, price
FROM products
ORDER BY price
LIMIT 2 OFFSET 3;

-- 2-й вариант
SELECT
    id, name, price
FROM products
ORDER BY price
LIMIT 3,2;
```

Результат:

id	name	price
2	Samsung	1099.99
4	Samsung	1199.99

ФИЛЬТРАЦИЯ СТРОК СОГЛАСНО СОБСТВЕННЫМ УСЛОВИЯМ

- Выборка только уникальных строк и ограничение количества выбираемых строк - замечательные возможности, которые сильно помогают в работе с таблицами баз данных. Однако они не могут покрыть все возможные требования - очень часто нам необходимо применять к выборке очень нестандартные фильтры. Однако, как и всегда, в MySQL существует оператор **WHERE**, который позволяет решить эту проблему. Важно сразу же отметить, что этот оператор срабатывает еще до того, как будут выполнены **ORDER BY** и **LIMIT** (если они включены в один и тот же запрос).

- Синтаксис **WHERE** крайне прост - сразу после него можно писать условия, согласно которым будут выбираться строки из таблицы. Если условие окажется ложно в рамках какой-то конкретной строки, то эта строка не будет включена в результат. Самые простые условия - сравнение значение какого-то столбца с каким-либо значением при помощи конструкций > (больше), >= (больше или равно), < (меньше), <= (меньше или равно), = (равно) или <> (не равно).

- Попробуем применить полученные знания, чтобы получить все продукты, цена которых больше и равна 900, а затем отсортируем по полученной цене:

```
SELECT id, name, price
FROM products
WHERE price >= 900
ORDER BY price;
```

id	name	price
1	iPhone	999.99
2	Samsung	1099.99
4	Samsung	1199.99

- **WHERE** позволяет использовать несколько условий одновременно - например, мы хотим выборку, в которой присутствуют продукты с именем Samsung и ценой более 1100 (эти два условия должны выполняться сразу, а не по отдельности). Чтобы отделять условия, применяется оператор **AND**:

```
SELECT id, name, price
FROM products
WHERE name = 'Samsung' AND price > 1100;
```

В результате получим один продукт:

id	name	price
4	Samsung	1199.99

- Иногда нам необходимо применить несколько условий, но мы согласны на то, чтобы истинным оказалось хотя бы одно из них (первое или второе или третье и т.д.). Для таких задач используется оператор **OR**. Попробуем выбрать все продукты, именем которых является iPhone или которые стоят дешевле 900:

```
SELECT id, name, price, created_at
FROM products
WHERE name = 'iPhone' OR price < 900;
```

В выборке окажется один iPhone (цена больше 900, но имеет название iPhone) и два Xiaomi (название отличается от iPhone, но цена меньше 900, что соответствует условию):

id	name	price	created_at
1	iPhone	999.99	2023-01-01
3	Xiaomi	899.99	2023-03-03
5	Xiaomi	799.99	2023-05-05

- Если после **WHERE** будет установлено несколько условий, которые будут разделены как **OR**, так и **AND**, то сначала будут проверены те условия, которые разделены **AND**, а уже потом **OR**:

```
SELECT id, name, price, created_at
FROM products
WHERE name = 'Samsung'
OR price = 999.99
AND created_at > '2023-02-03';
```

Несмотря на то, что под комбинацию двух последних условий не подходит не один продукт, нам будет возвращены два продукта Samsung, т.к. они подходят под первое условие, после которого стоит **OR** (или) и которое было выполнено последним:

id	name	price	created_at
2	Samsung	1099.99	2023-02-02
4	Samsung	1199.99	2023-04-04

- Стандартное поведение операторов **OR** и **AND** можно изменить, если расставить скобки. Например, следующий запрос идентичен предыдущему во всем, кроме скобок, но он вернет только один продукт Samsung, т.к. скобки поменяли смысл фильтрации:

```
SELECT id, name, price, created_at
FROM products
WHERE (name = 'Samsung'
OR price = 999.99)
AND created_at > '2023-02-03';
```

Результат:

id	name	price	created_at
4	Samsung	1199.99	2023-04-04

- Помимо операций сравнения (больше, меньше и т.д.), в MySQL существует еще ряд других операторов, которые используются в контексте главной директивы **WHERE**. Один из самых интересных таких операторов носит название **LIKE**. Он позволяет искать строки не по конкретным значениям, а по части значения какого-либо текстового столбца. Например, нам надо получить выборку всех строк, которые начинаются с буквы X, после которой могут идти любые другие символы (или вообще ничего не идти). Любые другие символы (или их отсутствие) обозначается символом **%**:

```
SELECT id, name, price
FROM products
WHERE name LIKE 'X%';
```

В качестве закономерного результата получим все продукты Xiaomi:

id	name	price
3	Xiaomi	899.99
5	Xiaomi	799.99

Символ % может также стоять в самом начале шаблона поиска - попробуем найти все продукты, которые заканчиваются на g:

```
SELECT id, name, price
FROM products
WHERE name LIKE '%g';
```

Само собой, мы получим все продукты Samsung:

id	name	price
2	Samsung	1099.99
4	Samsung	1199.99

Приведем пример символа % в нескольких местах одновременно - попробуем найти все продукты, которые содержат букву n (т.е. что угодно может стоять как до этой буквы, так и после):

```
SELECT id, name, price
FROM products
WHERE name LIKE '%n%';
```

В качестве результата мы получим один продукт iPho**n**e и два продукта Samsu**n**g:

id	name	price
1	iPhone	999.99
2	Samsung	1099.99
4	Samsung	1199.99

Еще один символ, который используется в контексте оператора **LIKE** - символ нижнего подчеркивания _. В отличие от % он означает не сколько угодно символов, а любой, но один символ:

```
SELECT id, name, price
FROM products
WHERE name LIKE 'iPho_';
```

В нашей таблице после символов iPho могут идти только два символа (ne), а не один. Поэтому в данном случает мы получим пустую таблицу. Если же мы хотим получить, что-то осмысленное, то добавим нижнее подчеркивание два раза:

```
SELECT id, name, price
FROM products
WHERE name LIKE 'iPho__';
```

Теперь мы уже увидим наш закономерный iPhone:

id	name	price
1	iPhone	999.99

Заканчивая часть об операторе **LIKE**, отметим, что он также зависит от сопоставлений (**COLLATE**), т.е. верхний и нижний регистр букв при сравнении будет зависеть от того, какое сопоставление установлено у столбца, таблицы или всей базы данных. Если же мы хотим установить свое сопоставление только на время конкретного запроса, то это делается следующим образом:

```
SELECT id, name, price
FROM products
WHERE name LIKE 'iPho__' COLLATE utf8mb4_0900_as_cs;
```

- Еще одним оператором, использующимся в связке с глобальной директивой **WHERE**, является **BETWEEN**. Как можно понять из названия, он помогает выявить те значения столбцов, которые находятся в определенном интервале (от и до включая). Именно поэтому BETWEEN осуществляет сравнение значения не с один

другим значением, а с двумя и при этом использует уже упоминавшийся оператор AND, но уже в другом контексте. Например, нам надо получить те продукты, цена который между (BETWEEN) 900 и (AND) 1100, включая оба значения:

```
SELECT id, name, price
FROM products
WHERE price BETWEEN 900 AND 1100;
```

Как результат увидим таблицу, с двумя устраивающими нас результатами:

id	name	price
1	iPhone	999.99
2	Samsung	1099.99

Кроме чисел, оператор **BETWEEN** можно использовать и для нахождения значений в неком текстовом интервале (с учетом лексикографического сравнения и сопоставлений), а также временных типов данных (**DATETIME**, **DATE** и т.д.). Например, получим те продукты, которые были добавлены в таблицу (created_at) в период времени 2 февраля 2023 года и 4 апреля 2023 года (включая оба значения) - для наглядности включим в выборку сам столбец created_at:

```
SELECT id, name, price, created_at
FROM products
WHERE created_at BETWEEN '2023-02-02' AND '2023-04-04';
```

В ответ получим абсолютно правильную таблицу:

id	name	price	created_at
2	Samsung	1099.99	2023-02-02
3	Xiaomi	899.99	2023-03-03
4	Samsung	1199.99	2023-04-04

- Следующая конструкция - оператор **IN** - позволяет определить определенный набор (список) разрешенных значений - если значение столбца равно хотя бы одному из этих разрешенных

значений, то вся строка, имеющая столбец с разрешенным значением, попадет в результирующую таблицу. Например, попытаемся получить все продукты, id которых могут иметь следующие значения 2, 20, 1000, 5, '1':

```
SELECT id, name, price
FROM products
WHERE id IN (2, 20, 1000, 5, '1');
```

В результате мы увидим в результате строки с id равными 1, 2, 5. Особо отметим, что в разрешенные значения мы вписали 1 как строку, а столбец id является целочисленным. Это означает, что MySQL пытается преобразовать разрешенные значения в нужный тип - если у него это получается, то подходящие значения будут вставлены в ответ:

id	name	price
1	iPhone	999.99
2	Samsung	1099.99
5	Xiaomi	799.99

- У оператора **IN** есть конструкция-антипод **NOT IN**, которая делает обратное - определяет список значений, которым значение столбца равняться не должно. Повторим запрос из предыдущего примера, но вместо **IN**, подставим **NOT IN**:

```
SELECT id, name, price
FROM products
WHERE id NOT IN (2, 20, 1000, 5, '1');
```

Как и ожидалось, в этот раз ни одной строки из предыдущего результата мы не увидим:

id	name	price
3	Xiaomi	899.99
4	Samsung	1199.99

- Последняя пара конструкций, которую мы рассмотрим в этой подглаве и которые используются совместно с директивой WHERE - **IS NULL** и **IS NOT NULL**. По названию можно понять, что они используются для сравнения со столбом, который потенциально может содержать **NULL** в качестве значения. Однако сразу же возникает вопрос - зачем нам такие конструкции, если у нас уже есть = и <>? Чтобы на этот вопрос ответить, внесем в нашу таблицу products еще один продукт (шестой по счету), у которого ean будет равен **NULL**:

```
INSERT INTO products(name, ean, price, created_at)
VALUES
('Nokia', NULL, 950.99, '2023-06-06');
```

После успешной вставки, попробуем выбрать все строки у которых ean равен **NULL** при помощи простого сравнения =:

```
SELECT id, name, price, ean
FROM products
WHERE ean = NULL;
```

С удивлением мы обнаружим, что нам вернулась пустая таблица. Дело в том, что **NULL** - это специальное значение, которое, как уже упоминалось, обозначает “отсутствие значение” (своеобразную черную дыру, неизвестность). Поэтому **NULL** не может быть равен какому-либо другому значению, даже другому **NULL**. Однако если нам надо понять, что в столбце сохранен именно **NULL**, то нам следует применять конструкцию **IS NULL**, что мы и сделаем:

```
SELECT id, name, price, ean
FROM products
WHERE ean IS NULL;
```

В качестве ответа получим наш закономерный шестой продукт:

id	name	price	ean
6	Nokia	950.99	NULL

Далее приведем пример использования конструкции **IS NOT NULL** - она, как и ожидается, вернет все строки, кроме той, которую мы вставили последней:

```
SELECT id, name, price, ean
FROM products
WHERE ean IS NOT NULL;
```

id	name	price	ean
1	iPhone	999.99	74748484
2	Samsung	1099.99	9393983
3	Xiaomi	899.99	35333653
4	Samsung	1199.99	5394983
5	Xiaomi	799.99	65322653

- В самом конце упомянем, что директива **WHERE** и все связанные с ней конструкции могут осуществлять выборку и по значениям тех столбцов, которые не перечислены после **SELECT**. Возможно, для кого-то это очевидно, но на всякий случай подтвердим утверждение примером. Выберем столбцы name и price для тех строк, у которых значение id - четное число (т.е. условие ориентируется на столбец id, а выбираются name и price):

```
SELECT name, price
FROM products
WHERE id % 2 = 0;
```

Результат:

name	price
Samsung	1099.99
Samsung	1199.99

ВЫБОРКА ИЗ СТОЛБЦОВ С ТИПОМ JSON

- При выборке данных тип JSON по причине своего несколько нестандартного формата должен быть обработан специфичным образом. Чтобы это продемонстрировать, создадим таблицу documents с двумя полями - id (целочисленный тип, первичный ключ) и data (JSON):

```
CREATE TABLE documents (
    id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    data JSON NOT NULL
);
```

Затем занесем в нашу таблицу два документа:

```
INSERT INTO documents (data)
    VALUES
('{
    "title": "1st Document",
    "content": {"en": "Text 1", "ru": "Текст 1"},
    "paragraphs": [10, 55, 18]
}'),
('{
    "title": "2nd Document",
    "content": {"en": "Text 2", "ru": "Текст 2"},
    "paragraphs": [16, 47, 98]
}');
```

А теперь попробуем получить первую выборку по условию **WHERE** (в следующем запросе значение после **WHERE** идентично тому значению, которое мы вставляли первым):

```
SELECT id, data FROM documents
WHERE data = '{
    "title": "1st Document",
    "content": {"en": "Text 1", "ru": "Текст 1"},
    "paragraphs": [10, 55, 18]
}';
```

Удивительно, но мы получим в ответ пустую таблицу. JSON нельзя получить, если мы пытаемся сравнить значение столбца со строкой - тут требуется дополнительная обработка и синтаксис, который, например, должен быть ориентирован на какое-то значение внутри самой сохраненной JSON строки.

- Если мы ожидаем, что внутри JSON строки (столбец data) есть поле под названием 'title' и со значением '1st Document', то запрос необходимо сделать вот так:

```
SELECT id, data FROM documents
WHERE data->'$.title' = '1st Document';
```

Вот теперь мы уже получим первый документ, у которого title равен значению '1st Document':

id	data
1	{ "title": "1st Document", "content": { "en": "Text 1", "ru": "Текст 1" }, "paragraphs": [10, 55, 18] }

Если у нас значение имеет вложенность, то части пути в запросе разделяются точкой. Попробуем получить документ, к которого в поле en поля content хранится значение 'Text 2':

```
SELECT id, data FROM documents
WHERE data->'$.content.en' = 'Text 2';
```

В результате получим абсолютно правильный второй документ:

id	data
2	{ "title": "2nd Document", "content": { "en": "Text 2", "ru": "Текст 2" }, "paragraphs": [16, 47, 98] }

Если у нас значение входит в массив, то надо указать индекс этого значения (в массивах нумерация начинается с нуля):

```
SELECT id, data FROM documents
WHERE data->'$.paragraphs[1]' = 55;
```

- Мы уже рассматривали ситуацию, когда мы хотели получить данные по значению всего JSON. Это можно сделать, но сравнивать надо не со строкой, а со специально сконструированным объектом, который отображает весь JSON, хранимый в таблице:

```
SELECT id, data FROM documents
WHERE data = JSON_OBJECT(
    "title", "1st Document",
    "content", JSON_OBJECT("en", "Text 1", "ru", "Текст 1"),
    "paragraphs",  JSON_ARRAY(10, 55, 18)
);
```

- Иногда нам надо получить не весь JSON, а значение какого-либо поля внутри JSON строки. Чтобы этого добиться, необходимо использовать синтаксис, похожий на тот, который уже использовался в директиве **WHERE**, но с использованием функции **JSON_EXTRACT**. Для начала получим значение поля title:

```
SELECT id, JSON_EXTRACT(data, '$.title') AS title
FROM documents;
```

id	title
1	1st Document
2	2nd Document

Далее получим вложенное значение (части пути разделяется точкой):

```
SELECT id, JSON_EXTRACT(data, '$.content.en') AS content_en
FROM documents;
```

id	content_en
1	Text 1
2	Text 2

А теперь получим значение массива **paragraphs** по индексу 2 (т.е. третий элемент) внутри JSON:

```
SELECT
    id,
    JSON_EXTRACT(data, '$.paragraphs[2]') AS third_paragraph
FROM documents;
```

id	third_paragraph
1	18
2	98