GeekBrains

Базы данных. Интерактивный курс

# Урок 3

## Операторы, фильтрация, сортировка и ограничение

# Операторы

Под операторами подразумеваются конструкции языка, которые производят преобразование данных, например, операция сложения — **+**, вычитания — **-** и т. п. Данные, над которыми совершается операция, называются операндами.

## Арифметические операторы

К арифметическим операциям относятся сложение (**+**), вычитание (**-**), умножение (*****), деление (**/**). Кроме того, выделяют взятие остатка от деления, процент и целочисленное деление **DIV**.

| Оператор | Описание |
|---|---|
| 5 + 2 | Сложение |
| 5 - 2 | Вычитание |
| 2 * 3 | Умножение |
| 2 / 3 | Деление |
| 9 % 3 | Остаток от деления |
| 10 DIV 3 | Целочисленное деление |

Оператор **+** хорошо нам известен из школьного курса:

```
SELECT 3 + 5;
```

Для получения значения используется оператор **SELECT**. Результатом выступает таблица, состоящая из одного столбца и одной строки. Название столбца совпадает с вычисляемым значением: **3 + 5**. Однако при желании мы можем его переименовать при помощи ключевого слова **AS**:

```
SELECT 3 + 5 AS summ;
```

Применять арифметические операторы можно не только к обычным числам, но и к столбцам.

```
SELECT * FROM catalogs;
UPDATE catalogs SET id = id + 10;
SELECT * FROM catalogs;
```

Выше при помощи UPDATE-запроса значения идентификаторов во всех записях увеличивается на десять. Чаще всего такой прием используется для обновления счетчиков, календарных или денежных значений.

Важно отметить, что операция сложения числа с **NULL** снова дает **NULL**:

```
SELECT 3 + NULL;
```

Такое поведение вполне оправдано. **NULL** обозначает данные, значение которых не определено, поэтому прибавление к такому значению числа приводит опять к неопределенному значению, т. е., к **NULL**.

Если в качестве слагаемых будут выступать строки, они будут автоматически приведены к числам. Результатом в этом случае также будет число:

```
SELECT '3' + '5';
```

При этом, если строка не может быть приведена к числу, она интерпретируется как 0:

```
SELECT 'abc' + 'dfe';
```

Операция вычитания имеет те же особенности и ограничения, что и операция сложения.

```
SELECT 8 - 3;
```

Помимо бинарного оператора **-**, который производит вычитание, существует унарный оператор, который меняет знак операнда.

```
SELECT -7;
```

В операции умножения также нет ничего особенного:

```
SELECT 2 * 3;
```

Однако при умножении следует иметь в виду, что очень легко выйти за допустимые границы типа:

```
SELECT 18014398509481984 * 18014398509481984;
ERROR 1690 (22003): BIGINT value is out of range in '(18014398509481984 *
18014398509481984)'
```

Если мы выходим за границы типа **BIGINT**, MySQL возвращает ошибку.

В отличие от других языков программирования, деление на ноль не вызывает ошибки синтаксиса и остановки вычислений. В качестве результата возвращается **NULL**.

```
SELECT 8 / 0;
```

Кроме обычного деления, существует оператор целочисленного деления **DIV**.

```
SELECT 5 DIV 2, 5 / 2;
```

Результат деления при помощи **DIV** является целочисленным. Дробная часть просто отбрасывается и округления результата не производится. Чтобы получить остаток от деления, необходимо воспользоваться оператором **%**.

```
SELECT 5 % 2;
```

Результат запроса равен 1, т. к. без остатка на 2 делится только 4 (5 – 4 = 1). Оператор взятия остатка от деления **%** имеет еще две альтернативные формы написания:

- замена **%** на **MOD**;
- встроенная функция **MOD()**.

```
SELECT 5 % 2, 5 MOD 2;
```

# Операторы сравнения

Как и в любом другом языке программирования, в SQL большое значение имеет логический тип, который может принимать два значения: истину (**TRUE**) или ложь (**FALSE**).

```
SELECT TRUE;
SELECT FALSE;
```

MySQL поддерживает константы **TRUE** и **FALSE**. Однако, точно так же как и **SERIAL**, эти константы являются псевдонимами для 1 и 0 типа **TINYINT**.

Чаще всего логические значения получаются при помощи операторов сравнения, представленных в следующей таблице:

| Оператор | Описание |
|---|---|
| > | Больше |
| >= | Больше равно |
| < | Меньше |
| <= | Меньше равно |
| = | Равно |
| !=, <> | Не равно |
| <=> | Безопасное сравнение |

Ниже представлено типичное использование операторов сравнения.

```
SELECT 2 > 3;
SELECT 2 <= 3;
SELECT 2 = 2, 2 = 3;
SELECT 2 != 3, 2 <> 3;
```

Можно использовать отрицание, используя оператор **NOT**.

```
SELECT NOT TRUE, NOT FALSE;
SELECT ! TRUE, ! FALSE;
SELECT NOT 2 != 3, NOT 2 <> 3;
```

Сравнение с неопределенным значением **NULL** всегда возвращает **NULL** — неопределенное значение:

```
SELECT 2 = NULL, 2 != NULL;
```

Впрочем, существует специальный оператор **НЛО <=>**, который позволяет безопасно сравнивать со значением **NULL**:

```
SELECT 2 <=> NULL, NULL <=> NULL;
```

Более классические операторы сравнения с **NULL** — **IS NULL** и **IS NOT NULL**:

```
SELECT 2 IS NULL, 2 IS NOT NULL, NULL IS NULL, NULL IS NOT NULL;
```

## Логические операторы

Условия можно комбинировать при помощи логического И:

| AND | true | false |
|---|---|---|
| true | true | false |
| false | false | false |

Помимо логического И, поддерживается логическое ИЛИ:

| OR | true | false |
|---|---|---|
| true | true | true |
| false | true | false |

# Вычисляемые столбцы

Выражения можно сохранять в виде столбца таблицы. Оператор CREATE TABLE допускает создание столбцов, значение которых автоматически создается как арифметическое выражение других столбцов:

```
DROP TABLE IF EXISTS tbl;
CREATE TABLE tbl (
  x INT,
  y INT,
  summ INT AS (x + y)
);
INSERT INTO
  tbl (x, y)
VALUES
  (1, 1), (5, 6), (11, 12);
SELECT * FROM tbl;
```

По умолчанию значения в вычисляемом столбце не сохраняются на жесткий диск, они каждый раз вычисляются снова. Однако если добавить в определение столбца ключевое слово **STORED**, то такой столбец будет сохраняться на жесткий диск.

```
DROP TABLE IF EXISTS tbl;
CREATE TABLE tbl (
  x INT,
  y INT,
  summ INT AS (x + y) STORED
);
INSERT INTO
  tbl (x, y)
VALUES
  (1, 1), (5, 6), (11, 12);
SELECT * FROM tbl;
```

# Условная выборка

Ситуация, когда требуется изменить количество выводимых строк, встречается гораздо чаще, чем ситуация, когда требуется изменить число и порядок выводимых столбцов.

Для ввода в SQL-запрос такого рода ограничений в операторе **SELECT** предназначено специальное ключевое слово **WHERE**, после которого следует логическое условие. Если запись удовлетворяет такому условию, она попадает в результат выборки, в противном случае такая запись отбрасывается.

```
SELECT * FROM catalogs WHERE id_catalog > 2;
```

Условие может быть составным и объединяться при помощи логических операторов.

```
SELECT * FROM catalogs WHERE id > 2 AND id <= 4;
```

Для выборки записей из определенного интервала используется оператор **BETWEEN**. Ниже представлен запрос, полностью эквивалентный варианту выше с операторами сравнения.

```
SELECT * FROM catalogs WHERE id_catalog BETWEEN 3 AND 4;
```

Существует конструкция, противоположная **BETWEEN**, — **NOT BETWEEN**, которая возвращает записи, не попадающие в интервал.

```
SELECT * FROM catalogs WHERE id_catalog NOT BETWEEN 3 AND 4;
```

Иногда требуется извлечь записи, удовлетворяющие не диапазону, а списку, например, записи с **id** из списка (1,2,5). Для этого предназначена конструкция **IN**.

```
SELECT * FROM catalogs WHERE id IN (1,2,5);
```

Конструкция **IN** также возвращает **NULL**, если один из элементов списка равен **NULL**:

```
SELECT 2 IN (0,NULL,5,'wefwf');
SELECT 2 IN (2,NULL,5,'wefwf');
```

Конструкция **NOT IN** является противоположной оператору **IN** и возвращает **1** (истина), если проверяемое значение не входит в список, и **0** (ложь), если оно присутствует в списке.

```
SELECT * FROM catalogs WHERE id NOT IN (1,2,5);
```

Можно добавить отрицание и перед конструкцией NOT:

```
SELECT * FROM catalogs WHERE NOT id IN (1,2,5);
```

В конструкции **WHERE** могут использоваться не только числовые столбцы. Так ниже из таблицы **catalogs** извлекается запись, соответствующая элементу каталога «Процессоры».

```
SELECT * FROM catalogs WHERE name = 'Процессоры';
```

Зачастую условную выборку с участием строк удобнее производить не при помощи оператора равенства **=**, а при помощи оператора **LIKE**, который позволяет использовать простейшие шаблоны. Оператор часто используется в конструкции **WHERE** и возвращает **1** (истину), если шаблон соответствует выражению, и **0** (ложь) в противном случае.

```
SELECT 'MySQL' LIKE 'MySQL';
```

Главное преимущество оператора LIKE перед **=** заключается в использовании спецсимволов.

| Оператор | Описание |
|---|---|

| | |
|---|---|
| % | Любое количество символов или их отсутствие |
| _ | Ровно один символ |

```
SELECT 'Программист' LIKE 'Программ%';
SELECT 'Программа' LIKE 'Программ%';
SELECT 'Программ' LIKE 'Программ%';
```

Спецсимвол **%** заменяет собой любую последовательность символов. Поэтому шаблон «Программ%» удовлетворяет словам «Программист», «Программа» и «Программ» и будет удовлетворять любому слову, начинающемуся с выражения «Программ».

Спецсимвол **%** может быть размещен в любом месте шаблона, как в начале, так и в середине строки:

```
SELECT 'Программирование' LIKE 'П%е','Печенье' LIKE 'П%е';
SELECT 'Программирование' LIKE '%ние','Кодирование' LIKE '%ние';
```

Символ подчеркивания соответствует одному любому символу.

```
SELECT 'код' LIKE '___','рот' LIKE '___', 'абв' LIKE '___';
```

Так, шаблон из трех знаков подчеркивания **___** будет соответствовать любому слову, состоящему из трех символов: «код», «рот», «абв». Чтобы поместить в шаблон сами символы **%** и **_** без их специальной интерпретации, необходимо экранировать их при помощи обратного слеша:

```
SELECT '15 %' LIKE '15 \%', 'my_sql' LIKE 'my\_sql';
SELECT '15' LIKE '15 \%', 'my sql' LIKE 'my\_sql';
```

Рассмотрим использование оператора **LIKE** на примере таблицы catalogs. Извлечем имена каталогов, которые заканчиваются на символ **ы**.

```
INSERT INTO catalogs VALUES
  (NULL, 'Процессоры'),
  (NULL, 'Материнские платы'),
  (NULL, 'Видеокарты'),
  (NULL, 'Жесткие диски'),
  (NULL, 'Оперативная память');
SELECT * FROM catalogs WHERE name LIKE '%ы';
```

Оператор **NOT LIKE** противоположен по действию оператору **LIKE**:

```
SELECT * FROM catalogs WHERE name NOT LIKE '%ы';
```

Очень часто условия связаны с календарными столбцами. Давайте заполним таблицу пользователей:

```
INSERT INTO users (name, birthday_at) VALUES
  ('Геннадий', '1990-10-05'),
```

```
  ('Наталья', '1984-11-12'),
  ('Александр', '1985-05-20'),
  ('Сергей', '1988-02-14'),
  ('Иван', '1998-01-12'),
  ('Мария', '1992-08-29');
```

В таблице пользователей **users** есть поле **birthday_at**, которое соответствует дню рождения пользователей. Извлечем пользователей, которые родились в 90-е годы:

```
SELECT
  *
FROM
  users
WHERE
  birthday_at >= '1990-01-01' AND birthday_at < '2000-01-01';
```

Точно так же, как и в случае числовых данных, допускается использование оператора **BETWEEN**:

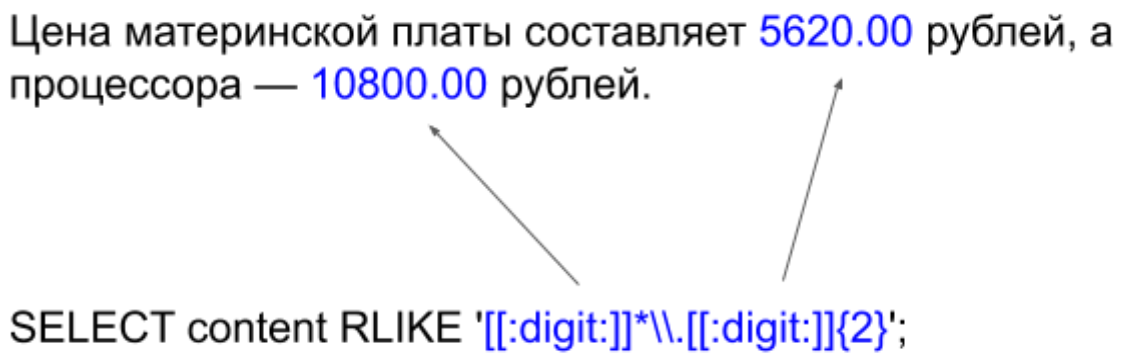
```
SELECT
  *
FROM
  users
WHERE
  birthday_at BETWEEN '1990-01-01' AND '2000-01-01';
```

При использовании оператора **LIKE** календарный столбец автоматически преобразуется к строке, поэтому представленный запрос мы можем записать еще более коротким способом:

```
SELECT
  *
FROM
  users
WHERE
  birthday_at LIKE '199%';
```

Оператор **RLIKE** (или его синоним **REGEXP**) позволяет производить поиск в соответствии с регулярными выражениями, которые предоставляют значительно более гибкие средства для поиска по сравнению с оператором **LIKE**. Обратной стороной медали является их более медленное выполнение.

Регулярные выражения — это специализированный язык поиска подстрок в тексте. Они оформляются в виде шаблона, который применяется к заданному тексту слева направо. Большая часть символов в таком шаблоне сохраняет свое значение, однако дополнительно вводятся символы, имеющие специальное значение: ограничители, классы, квантификаторы.




Слева от оператора **RLIKE** размещается текст, справа — регулярное выражение.

```
SELECT 'грамм' RLIKE 'грам', 'грампластинка' RLIKE 'грам';
```

Так, регулярное выражение, содержащее обычный текст, например «грам», соответствует строке, содержащей такую подстроку («грам»). Например, этому регулярному выражению будут соответствовать строки «программирование», «грамм», «грампластинка» и т. п.

Как видно, регулярное выражение «грам» осуществляет поиск по всему тексту, независимо от того, находится ли подстрока «грам» в начале, середине или конце слова.

Часто необходимо привязать регулярное выражение к началу слова, т. е., нужно, чтобы регулярное выражение «грам» соответствовало строке, начинающемуся с подстроки «грам», например, «грампластинка», но не соответствовало слову «программирование». Для этого используется символ **^**, соответствующий началу строки:

```
SELECT 'программирование' RLIKE '^грам', 'грампластинка' RLIKE '^грам';
```

Спецсимвол **$** позволяет привязать регулярное выражение к концу строки:

```
SELECT 'грампластинка' RLIKE '^грампластинка$';
```

Т. е., применение символов **^** и **$** позволяет указать, что регулярное выражение должно в точности соответствовать всей строке поиска от начала до конца.

```
SELECT 'грампластинка - это вам не программирование' RLIKE '^грампластинка$';
```

Символ вертикальной черты **|** применяется в регулярном выражении для задания альтернативных масок:

```
SELECT 'abc' RLIKE 'abc|абв', 'абв' RLIKE 'abc|абв';
```

Если шаблон должен включать символ **|** или любой другой, например рассмотренные выше **^** и **$**, то их необходимо экранировать при помощи двойного обратного слеша **** — в этом случае они теряют свое специальное значение и рассматриваются как обычные символы.

Для задания класса символов используются квадратные скобки, которые ограничивают поиск теми символами, которые в них заключены, например **[abc]**.

```
SELECT 'a' RLIKE '[abc]' AS a,
       'b' RLIKE '[abc]' AS b,
       'c' RLIKE '[abc]' AS c;
```

Регулярному выражению **[abc]** соответствует подстрока, содержащая один символ: либо **a**, либо **b**, либо **c**.

Так, для создания регулярного выражения, соответствующего всем буквам русского алфавита, можно, конечно, перечислить все буквы в квадратных скобках. Это допустимо, но утомительно и неэлегантно. Более кратко такое регулярное выражение можно записать следующим образом:

```
'[а-яё]'
```

Это выражение соответствует всем буквам русского алфавита, поскольку любые два символа, разделяемые дефисом, задают соответствие диапазону символов, находящихся между ними.

```
SELECT 'Л' RLIKE '[а-яё]' AS a;
SELECT 'z' RLIKE '[а-яё]' AS a;
```

Точно таким же образом задается регулярное выражение, соответствующее любому числу:

```
'[0-9]'
```

Это выражение эквивалентно:

```
'[0123456789]'
```

Кроме классов, которые могут создать разработчики, предусмотрены специальные конструкции классов:

```
SELECT '1' RLIKE '[[:digit:]]', 'a' RLIKE '[[:digit:]]';
SELECT '1' RLIKE '[[:alpha:]]', 'a' RLIKE '[[:alpha:]]';
```

Все, что находится в квадратных скобках, соответствует ровно одному символу. Чтобы распространить действие класса на несколько символов, используются квантификаторы, которые указываются сразу после квадратных скобок:

- **?** — символ входит ноль или один раз,
- ***** — любое количество вхождений, включая ноль,
- **+** — одно или более вхождений символа в строку.

```
SELECT '1' RLIKE '^[[:digit:]]+$', '453455234' RLIKE '^[[:digit:]]+$';
SELECT '' RLIKE '^[[:digit:]]+$', '45.3455234' RLIKE '^[[:digit:]]+$';
```

Помимо круглых и квадратных скобок, в регулярных выражениях также применяются фигурные скобки. Они предназначены для указания числа или диапазона повторения элемента.

Давайте создадим регулярное выражение для цены. Целая часть может состоять из любого числа цифр, а дробная часть всегда состоит из двух.

```
SELECT '123.90' RLIKE '^[[:digit:]]*\\.[[:digit:]]{2}$';
SELECT '123' RLIKE '^[[:digit:]]*\\.[[:digit:]]{2}$';
```

# Сортировка

Запрос выдает результаты в том порядке, в котором они хранятся в базе данных. Однако часто требуется отсортировать значения по одному из столбцов. Это делается при помощи конструкции **ORDER BY**. После конструкции **ORDER BY** указывается столбец (или столбцы), по которому следует сортировать данные.

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

Запрос сортирует результат выборки по полю id, можем отсортировать записи и по имени столбца:

```
SELECT id, name FROM catalogs ORDER BY name;
```

По умолчанию сортировка производится в прямом порядке, однако, добавив после имени столбца ключевое слово **DESC**, можно добиться сортировки в обратном порядке:

```
SELECT * FROM catalogs ORDER BY id DESC;
```

Сортировку записей можно производить и по нескольким столбцам. Давайте вставим несколько товарных позиций в таблицу **products**:

```
INSERT INTO products
  (name, description, price, catalog_id)
VALUES
  ('Intel Core i3-8100', 'Процессор для настольных персональных компьютеров,
основанных на платформе Intel.', 7890.00, 1),
  ('Intel Core i5-7400', 'Процессор для настольных персональных компьютеров,
основанных на платформе Intel.', 12700.00, 1),
  ('AMD FX-8320E', 'Процессор для настольных персональных компьютеров,
основанных на платформе AMD.', 4780.00, 1),
  ('AMD FX-8320', 'Процессор для настольных персональных компьютеров, основанных
на платформе AMD.', 7120.00, 1),
  ('ASUS ROG MAXIMUS X HERO', 'Материнская плата ASUS ROG MAXIMUS X HERO, Z370,
Socket 1151-V2, DDR4, ATX', 19310.00, 2),
  ('Gigabyte H310M S2H', 'Материнская плата Gigabyte H310M S2H, H310, Socket
1151-V2, DDR4, mATX', 4790.00, 2),
  ('MSI B250M GAMING PRO', 'Материнская плата MSI B250M GAMING PRO, B250, Socket
1151, DDR4, mATX', 5060.00, 2);
```

```
SELECT * FROM products;
```

Чтобы отсортировать таблицу по каталогам, в рамках каждого каталога, по цене, мы можем указать после ключевого слова **ORDER BY** сначала поле **catalog_id**, а затем поле **price**:

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

В рамках каждого каталога у нас сначала выводятся самые дешевые товарные позиции, а потом дорогие. Если мы захотим изменить порядок сортировки, мы можем добавить ключевое слово **DESC**:

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

Ключевое слово **DESC** относится только к полю **price**, и чтобы отсортировать оба столбца в обратном порядке, потребуется снабдить **DESC** как **id_catalog**, так и **price**.

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

# Ограничения

Результат выборки может содержать сотни и тысячи записей, их вывод и обработка занимают значительное время и серьезно нагружают сервер базы данных. Поэтому информацию часто разбивают на страницы и предоставляют ее пользователю порциями. Извлечение только части запроса требует меньше времени и вычислений, кроме того, пользователю часто бывает достаточно посмотреть первые несколько записей. Постраничная навигация используется при помощи ключевого слова **LIMIT**, за которым следует число выводимых записей.

```
SELECT * FROM products
ORDER BY name
LIMIT 2;
```

Здесь извлекаются первые две записи таблицы **products**, при этом записи сортируются по полю **name**.

Чтобы извлечь следующие две записи, используется ключевое слово **LIMIT** с двумя числами. Первое указывает позицию, начиная с которой необходимо вернуть результат, а второе — количество извлекаемых записей.

```
SELECT * FROM products
ORDER BY name
LIMIT 2, 2;
```

Существует и альтернативная форма записи такого оператора, с использованием ключевого слова **OFFSET**:

```
SELECT * FROM products
ORDER BY name
LIMIT 2 OFFSET 2;
```

# Уникальные значения

Очень часто возникает задача вывода уникальных значений из таблицы.

```
SELECT catalog_id FROM products ORDER BY catalog_id;
```

Данный запрос выдаст множество повторяющихся значений. Иногда удобнее, когда возвращаются только уникальные значения, Для этого перед именем столбца можно использовать ключевое слово **DISTINCT**:

```
SELECT DISTINCT catalog_id FROM products ORDER BY catalog_id;
```

Для ключевого слова **DISTINCT** имеется противоположное слово **ALL**, которое предписывает извлечение всех значений столбца, в том числе и повторяющихся. Поскольку такое поведение установлено по умолчанию, ключевое слово **ALL** часто опускают.

```
SELECT ALL catalog_id FROM products ORDER BY catalog_id;
```

Для решения схожих задача, часто используется группировка значений, при помощи ключевого слова **GROUP BY**, которому будет посвящена следующая тема. Условия и ограничения, которые мы рассмотрели выше, можно применять и в отношении команд обновления **UPDATE** и удаления **DELETE**.

Например, давайте уменьшим цену на 10 % для материнских плат, которые стоят больше 5000 рублей. Давайте сначала найдем все товарные позиции, подходящие к этому условию:

```
SELECT
  *
FROM
  products
WHERE
  catalog_id = 2 AND
  price > 5000;
```

А затем заменим команду **SELECT** на **UPDATE**:

```
UPDATE
  products
```

```
SET
  price = price * 0.1
WHERE
  catalog_id = 2 AND
  price > 5000;
```

Мы можем удалить две самые дорогие товарные позиции из таблицы **products**, для этого необходимо отсортировать таблицу при помощи ключевого слова **ORDER BY**:

```
SELECT
  *
FROM
  products
ORDER BY
  price DESC;
```

Ограничиваем выборку двумя строками:

```
SELECT
  *
FROM
  products
ORDER BY
  price DESC
LIMIT 2;
```

А затем заменить команду **SELECT** на **DELETE**:

```
DELETE FROM
  products
ORDER BY
  price DESC
LIMIT 2;
```

Таким образом, всегда можно подобрать подходящие условия при помощи SELECT-запроса, а потом поменять **SELECT** на **UPDATE** или **DELETE**.

# Предопределенные функции

MySQL, как и любая другая база данных, обладает большим числом предопределенных функций, т. е. готовых функций, которые предоставляют систему управления базами данных. Мы как разработчики можем писать и свои собственные функции, однако это тема следующих роликов.

Так же, как и в любом другом языке программирования, функции характеризуются именем и аргументами, которые перечисляются через запятую за именем в круглых скобках. Если аргументы у функции отсутствуют, круглые скобки все равно следует указывать. Результат функции подставляется в место вызова функции.

## Календарные функции

Например, одной из часто используемых является функция **NOW()**, которая позволяет получить текущую дату:

```
SELECT NOW();
```

Например, при вставке нового значения в таблицу **users** требуется 3 временных метки:

- дата рождения — **birthday_at**;
- дата создания записи — **created_at**;
- дата обновления записи — **updated_at**.

Таблица у нас создана таким образом, что две последние даты задаются неявно, однако мы могли бы задавать их при помощи функции **NOW()**:

```
INSERT INTO users VALUES (NULL, '1986-01-20', NOW(), NOW());
```

Вычисление текущего времени в рамках одного SQL-запроса производится только один раз, сколько бы раз они ни вызывались на протяжении данного запроса. Это приводит к тому, что временное значение в рамках всего запроса остается постоянным.

Календарных функций довольно много. Например, при помощи функции **DATE()** в полях **updated_at** и **created_at** можно отсекать время суток, таким образом, в результирующей таблице остаётся только дата.

```
SELECT id, name, birthday_at, DATE(created_at), DATE(updated_at) FROM users;
```

Обратите внимание на название столбцов с использованием функций **DATE**: они содержат название функций и аргументы. В результате оперировать названиями столбцов очень неудобно. Поэтому их часто переименовывают при помощи ключевого слова **AS**:

```
SELECT
  id,
  name,
  birthday_at,
  DATE(created_at) AS created_at,
  DATE(updated_at) AS updated_at
FROM
  users;
```

Допускается не указывать ключевое слово **AS**:

```
SELECT
  id,
  name,
  birthday_at,
  DATE(created_at) created_at,
  DATE(updated_at) updated_at
FROM
```

```
  users;
```

Эффект получается тот же самый, столбец получает новое название.

Для форматирования календарных типов используется функция **DATE_FORMAT(date, format)**, которая принимает в качестве первого аргумента время в одном из календарных типов, а в качестве второго — строку форматирования.

```
SELECT DATE_FORMAT('2018-06-12 01:59:59', 'На дворе %Y год');
```

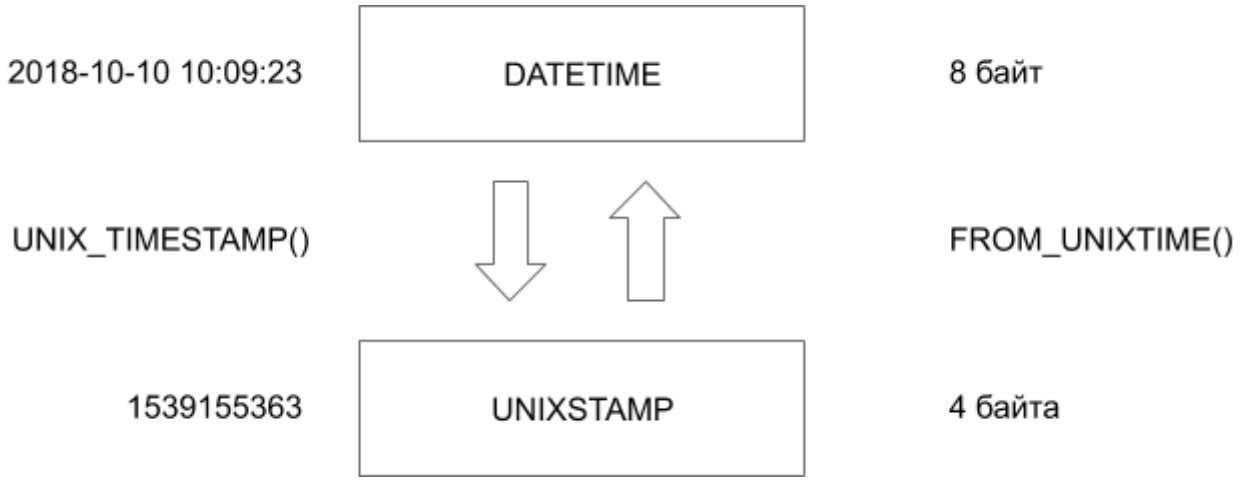
Скалярное значение даты можно заменить вызовом уже рассмотренной функции **NOW()**:

```
SELECT DATE_FORMAT('2018-06-12 01:59:59', 'На дворе %Y год');
```

Последовательность **%Y** отвечает за извлечение года из календарного значения и представления его в строковом виде. Таких последовательностей очень много, желательно познакомиться с ними по документации. Например, мы можем отформатировать день рождения пользователей в более привычном формате: день, месяц, год:

```
SELECT name, DATE_FORMAT(birthday_at, '%d.%m.%Y') AS birthday_at FROM users;
```

Еще часто встречается задача преобразования даты и времени в UNIXSTAMP-формат — количество секунд, которое прошло с полуночи 1 января 1970 года. Так как это целое число, его можно довольно быстро обрабатывать, индексировать и оно занимает мало места. Достаточно 4 байт, если нас устраивают даты от 1970 по 2038 год.




```
SELECT
  UNIX_TIMESTAMP('2018-10-10 10:09:23') AS TIMESTAMP,
  FROM_UNIXTIME(1539155363) AS DATETIME;
```

Пусть стоит задача вычисления текущего возраста пользователя. Один из вариантов состоит в преобразовании даты рождения и текущей даты в дни при помощи функции **TO_DAYS()** и делению на

число 365.25. В году 365 дней, дробное число 0.25 призвано компенсировать високосные года, которые случаются раз в четыре года.

```
SELECT
  name,
  (TO_DAYS(NOW()) - TO_DAYS(birthday_at))/365.25 AS age
FROM
  users;
```

Чтобы избавиться от дробной части, можно воспользоваться функцией **FLOOR()**.

```
SELECT
  name,
  FLOOR((TO_DAYS(NOW()) - TO_DAYS(birthday_at))/365.25) AS age
FROM
  users;
```

Можно добиться более точного результата, если воспользоваться специальной функцией **TIMESTAMPDIFF()**:

```
SELECT
  name,
  TIMESTAMPDIFF(YEAR, birthday_at, NOW()) AS age
FROM
  users;
```

## Случайное значение

Использование функций допускается не только после ключевого слова **SELECT()**. Везде, где используется имя столбца, можно задействовать функцию, например, для вывода записей в случайном порядке можно задействовав функцию **RAND()**, передав ее ключевому слову **ORDER BY**:

```
SELECT * FROM users ORDER BY RAND();
```

Получить случайное значение можно, если ограничить выборку при помощи ключевого слова **LIMIT**:

```
SELECT * FROM users ORDER BY RAND() LIMIT 1;
```

Существуют и информационные функции, например функция **VERSION()** возвращает текущую версию MySQL-сервера:

```
SELECT VERSION();
```

Обратите внимание, что мы часто не используем ключевое слово **FROM** и таблицу, когда нам требуется извлечь только одно значение, например, то, которое возвращает встроенная функция. В таком случае допускается использование псевдотаблицы **DUAL**, которая на самом деле не существует:

```
SELECT VERSION() FROM DUAL;
```

## Информационные функции

Часто в прикладных программах требуется узнать значение, присвоенное столбцу и снабженное атрибутом **AUTO_INCREMENT**. Это может потребоваться, чтобы использовать сгенерированное значение первичного ключа в качестве внешнего в другой таблице. Только что сгенерированное значение возвращает встроенная функция MySQL **LAST_INSERT_ID()**.

```
INSERT INTO catalogs VALUES (NULL, 'Процессоры');

INSERT INTO products
  (name, description, price, catalog_id)
VALUES
  ('Intel Core i3-8100', 'Процессор Intel.', 7890.00, LAST_INSERT_ID()),
  ('Intel Core i5-7400', 'Процессор Intel.', 12700.00, LAST_INSERT_ID()),
  ('AMD FX-8320E', 'Процессор AMD.', 4780.00, LAST_INSERT_ID()),
  ('AMD FX-8320', 'Процессор AMD.', 7120.00, LAST_INSERT_ID());

INSERT INTO catalogs VALUES (NULL, 'Материнские платы');

INSERT INTO products
  (name, description, price, catalog_id)
VALUES
  ('ASUS ROG MAXIMUS X HERO', 'Z370, Socket 1151-V2, DDR4, ATX', 19310.00,
LAST_INSERT_ID()),
  ('Gigabyte H310M S2H', 'H310, Socket 1151-V2, DDR4, mATX', 4790.00,
LAST_INSERT_ID()),
  ('MSI B250M GAMING PRO', 'B250, Socket 1151, DDR4, mATX', 5060.00,
LAST_INSERT_ID());

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

К информационным функциям относится также функция **DATABASE()**, которая возвращает текущую базу данных. Если текущая база данных не выбрана, **DATABASE()** возвращает **NULL**:

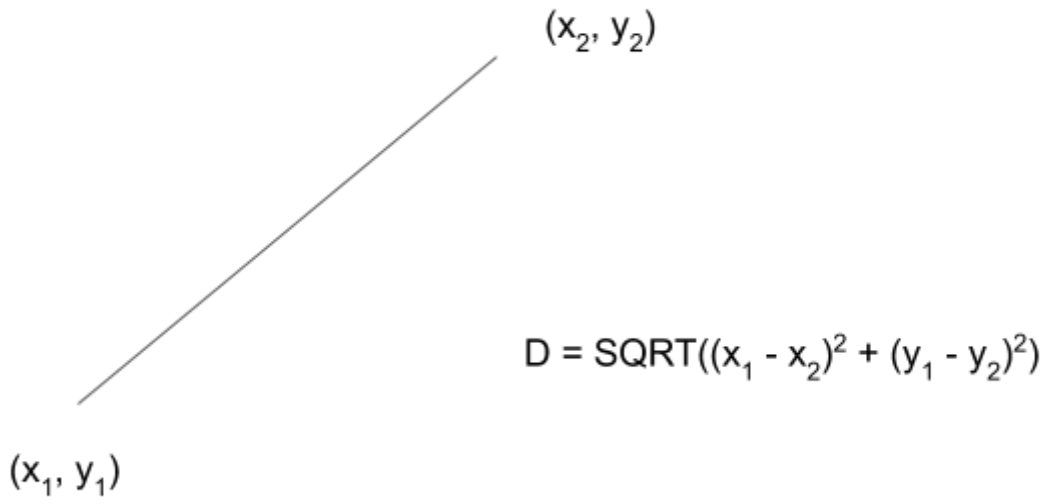
```
SELECT DATABASE();
USE shop
SELECT DATABASE();
```

Функция **USER()** возвращает аккаунт текущего пользователя:

```
SELECT USER();
```

# Математические функции

MySQL предоставляет огромное количество математических функций. С некоторыми, например функцией **RAND()**, предназначенной для получения случайных чисел, мы уже знакомы. Давайте познакомимся с некоторыми другими.




Например, функция **SQRT** позволяет получать квадратный корень числа. Давайте при помощи этой функции вычислим расстояние между двумя точками в декартовой системе координат. Пусть есть две точки с координатами X и Y, расстояние между ними вычисляется как разница квадратов расстояния по представленной на рисунке формуле.

```
DROP TABLE IF EXISTS distances;
CREATE TABLE distances (
  id SERIAL PRIMARY KEY,
  x1 INT NOT NULL,
  y1 INT NOT NULL,
  x2 INT NOT NULL,
  y2 INT NOT NULL,
  distance DOUBLE AS (SQRT(POW(x1 - x2, 2) + POW(y1 - y2, 2)))
) COMMENT = 'Расстояние между двумя точками';

INSERT INTO distances
  (x1, y1, x2, y2)
VALUES
  (1, 1, 4, 5),
  (4, -1, 3, 2),
  (-2, 5, 1, 3);

SELECT * FROM distances;
```

В качестве альтернативы можно использовать JSON-поля, отводя под каждую из точек отдельное поле, в котором будет JSON-коллекция, содержащая X и Y:

```
DROP TABLE IF EXISTS distances;
```

```
CREATE TABLE distances (
  id SERIAL PRIMARY KEY,
  a JSON NOT NULL,
  b JSON NOT NULL,
  distance DOUBLE AS (SQRT(POW(a->>'$.x' - b->>'$.x', 2) + POW(a->>'$.y' -
b->>'$.y', 2)))
) COMMENT = 'Расстояние между двумя точками';

INSERT INTO distances
  (a, b)
VALUES
  ('{"x": 1, "y": 1}', '{"x": 4, "y": 5}'),
  ('{"x": 4, "y": -1}', '{"x": 3, "y": 2}'),
  ('{"x": -2, "y": 5}', '{"x": 1, "y": 3}');

SELECT * FROM distances;
```

Для некоторых задач требуются тригонометрические функции. Например, если нам известен угол треугольника и длина двух его сторон, с использованием синуса мы можем вычислить его площадь. Давайте создадим таблицу **triangles**, которая будет состоять из трех столбцов:

```
DROP TABLE IF EXISTS triangles;
CREATE TABLE triangles (
  id SERIAL PRIMARY KEY,
  a DOUBLE NOT NULL COMMENT 'Сторона треугольника',
  b DOUBLE NOT NULL COMMENT 'Сторона треугольника',
  angle INT NOT NULL COMMENT 'Угол треугольника в градусах',
  square DOUBLE AS (a * b * SIN(RADIANS(angle)) / 2.0)
) COMMENT = 'Площадь треугольника';
```

Четвертый столбец мы сделаем вычисляемым, подставив в него формулу вычисления площади треугольника. Так как угол у нас задан в градусах, его потребуется преобразовать в радианы при помощи функции **RADIANS()**: синус будет ожидать значение именно в радианах.

```
INSERT INTO
  triangles (a, b, angle)
VALUES
  (1.414, 1, 45),
  (2.707, 2.104, 60),
  (2.088, 2.112, 56),
  (5.014, 2.304, 23),
  (3.482, 4.708, 38);

SELECT * FROM triangles;
```

Как видим, результат вычисления имеет до 16 знаков после запятой. Это не всегда удобно для восприятия: результат вычисления можно округлить при помощи помощи функции **ROUND()**.

Давайте при помощи оператора **ALTER TABLE** поменяем определение столбца **square** в таблице, добавив округление результата до четвертого знака после запятой.

```
ALTER TABLE triangles CHANGE square square DOUBLE AS (ROUND(a * b *
SIN(RADIANS(angle)) / 2.0, 4));
SELECT * FROM triangles;
```

Функция **ROUND()** осуществляет математическое округление, т. е., до ближайшего целого числа.

```
SELECT ROUND(2.4), ROUND(2.5), ROUND(2.6);
```

**ROUND()** — не единственная функция управления дробными числами. Функция **CEILING()** возвращает первое целое число, которое встречает справа от значения аргумента.

```
SELECT CEILING(-2.9), CEILING(-2.1), CEILING(2.1), CEILING(2.9);
```

Функция **FLOOR(X)** сходна по действию с функцией **CEILING(X)**, но сдвиг происходит в другую сторону.

```
SELECT FLOOR(-2.9), FLOOR(-2.1), FLOOR(2.1), FLOOR(2.9);
```

## Строковые функции

MySQL предоставляет большое количество функций, которые обслуживают строки. Очень часто требуется выбрать из таблицы не весь текст, а лишь несколько первых символов. Эту задачу удобно решать при помощи функции **SUBSTRING()**:

```
SELECT id, SUBSTRING(name, 1, 5) AS name FROM users;
```

Нумерация символов в строковых функциях всегда начинается с единицы. Для объединения строк предназначена функция **CONCAT**. Например, давайте выведем имя пользователя и его возраст через пробел:

```
SELECT id, CONCAT(name, ' ', TIMESTAMPDIFF(YEAR, birthday_at, NOW())) AS name
FROM users;
```

## Логические функции

Логические функции помогают преобразовать результат в зависимости от выполнения того или иного условия.

Давайте выведем слово «совершеннолетний» или «несовершеннолетний», в зависимости от того, достиг пользователь 18 лет или нет. Для этого можно воспользоваться функцией **IF**, которая принимает три аргумента:

- первый — логическое выражение,
- второй — результат, который выводится, если логическое выражение истинное,
- третий — если логическое выражение оказалось ложным.

Давайте посмотрим, как работает функция:

```
SELECT IF(TRUE, 'истина', 'ложь'), IF(FALSE, 'истина', 'ложь');
```

Теперь давайте решим задачу определения совершеннолетия пользователя:

```
SELECT
  name,
  IF(
      TIMESTAMPDIFF(YEAR, birthday_at, NOW()) >= 18,
      'совершеннолетний',
      'несовершеннолетний'
  ) AS status
FROM
  users;
```

Если условий больше, можно использовать выражение **CASE**, например, пусть у нас имеется таблица с цветами радуги:

```
DROP TABLE IF EXISTS rainbow;
CREATE TABLE rainbow (
  id SERIAL PRIMARY KEY,
  color VARCHAR(255)
) COMMENT = 'Цвета радуги';

INSERT INTO
  rainbow (color)
VALUES
  ('red'),
  ('orange'),
  ('yellow'),
  ('green'),
  ('blue'),
  ('indigo'),
  ('violet');

SELECT
  CASE
      WHEN color = 'red' THEN 'красный'
      WHEN color = 'orange' THEN 'оранжевый'
      WHEN color = 'yellow' THEN 'желтый'
      WHEN color = 'green' THEN 'зеленый'
      WHEN color = 'blue' THEN 'голубой'
      WHEN color = 'indigo' THEN 'синий'
      ELSE 'фиолетовый'
  END AS russian
FROM
  rainbow;
```

## Вспомогательные функции

MySQL предоставляет различные вспомогательные функции. Например функция **INET_ATON(address)** принимает IP-адрес **address** и представляет его в виде целого числа:

```
SELECT INET_ATON('62.145.69.10'), INET_ATON('127.0.0.1');
```

Функция **INET_NTOA** решает обратную задачу:

```
SELECT INET_NTOA(1049707786), INET_NTOA(2130706433);
```

Функция **UUID()** возвращает универсальный уникальный идентификатор. Идентификатор **UUID** реализован в виде числа, которое является глобально уникальным во времени и пространстве.

Два вызова функции **UUID()** вернут два разных значения, если они производятся одновременно на двух разных компьютерах или на одном и том же компьютере в разное время.

```
SELECT UUID();
SELECT UUID();
```

Это далеко не все функции, которые предоставляет MySQL. Часть функций, например агрегатные, мы будем рассматривать в следующих уроках.

# Используемые источники


1. https://dev.mysql.com/doc/refman/5.7/en/functions.html
2. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.
3. Грофф, 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.
4. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. — Пер. с англ. — СПб.: Символ-Плюс, 2010. — 480 с.
5. Кузнецов М.В., Симдянов И.В. MySQL на примерах. — СПб.: БХВ-Петербург, 2007. — 592с.
6. Кузнецов М.В., Симдянов И.В. MySQL 5. — СПб.: БХВ-Петербург, 2006. — 1024с.
7. Дейт, К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.
8. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.