



# Курс базы данных и SQL. Лекции 3 и 4



На 3 и 4 лекции будем более детально рассматривать типы данных, рассмотрим структуру запроса, поговорим о понятии "сортировка" и "группировка". Мы научимся ограничивать вывод строк с помощью оператора LIMIT, применим агрегатные функции. Будем получать только уникальные данные из вашей таблицы, применим оператор соединения таблиц(он не такой сложный, но очень интересный), рассмотрим общую структуру подзапросов. Поймем, как соединять два запроса в 1 с помощью UNION и UNION ALL.

## Что мы узнаем:

- Немного вспомним типы данных
- Структура запроса
- Сортировка
- Ограничения выборки. LIMIT
- Агрегатные функции
- Уникальные значения
- Подзапросы, подзапросы и еще раз - подзапросы
- UNION ALL и INION
- JOIN - соединения таблиц

## Доброго времени суток, уважаемые студенты!

Постарался приложить побольше картинок для вашего запоминания и углубить наши знания в рамках общего курса по SQL 😊

## Терминология

**NULL** соответствует понятию «пустое поле»null, то есть «поле, не содержащее никакого значения».

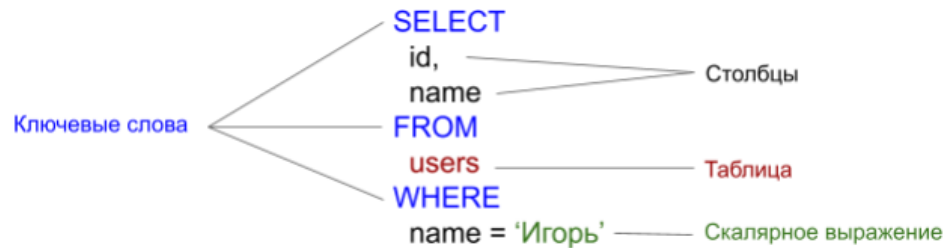
**JOIN** - оператор языка SQL, который является реализацией операции соединения реляционной алгебры.

**Группировка** — операция, которая создает из записей таблицы независимые группы записей, по которым проводится анализ.

**Агрегатные функции (агрегации)** — это функции, которые вычисляются от группы значений и объединяют их в одно результирующее.

## Структура любого запроса

Каждая инструкция SQL начинается с ключевого слова, которое описывает выполняемое действие.



После ключевого слова идет одно или несколько предложений. Предложение может описывать данные, с которыми работает инструкция, или содержать уточняющую информацию о действии, выполняемом инструкцией. Каждое предложение также начинается с ключевого слова, такого как **WHERE** (где), **FROM** (откуда, из какой таблицы).

**Скалярное выражение** — это сочетание символов и операторов, в результате вычисления которых возвращается одно значение. Фактически, это константы(строки и числа).


```
SELECT "Привет, мир!";
-- В запросе выше строка "Привет, мир!" — скалярное выражение
SELECT 2 + 2;
--Числа - тоже скалярные выражения
```

## И... немного обобщим типы данных

Более детально получить информацию о типах данных можно узнать по ссылке:

MySQL :: MySQL 8.0 Reference Manual :: 11 Data Types

MySQL supports SQL data types in several categories: numeric types, date and time types, string (character and byte) types, spatial types, and the data type. This chapter provides an overview and more detailed description of the properties of the types in each category, and a summary of the data type storage requirements.

 <https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

MySQL поддерживает несколько типов данных, которые можно разбить на пять групп:

- числовые данные (целые, вещественные или с плавающей точкой);
- строковые данные (фиксированного и переменного размера);
- специальный тип NULL, которое обозначает неопределенное значение, отсутствие информации;
- календарные данные предназначены для сохранения даты и времени;

- коллекционные типы позволяют сохранять множество значений или даже целые документы в виде JSON-полей.

Типы определяют характеристики сохраняемых значений, а также количество памяти, которая под них отводится. Многие типы могут сопровождаться дополнительными атрибутами.

Например, атрибуты `NULL` или `NOT NULL` задают ограничение на столбец, позволяя присваивать элементам неопределенное значение или, наоборот, запрещая такое поведение.

Атрибут `DEFAULT` позволяет задать полю значение по умолчанию, которое будет присваиваться в случае, если при создании записи значение не задано.

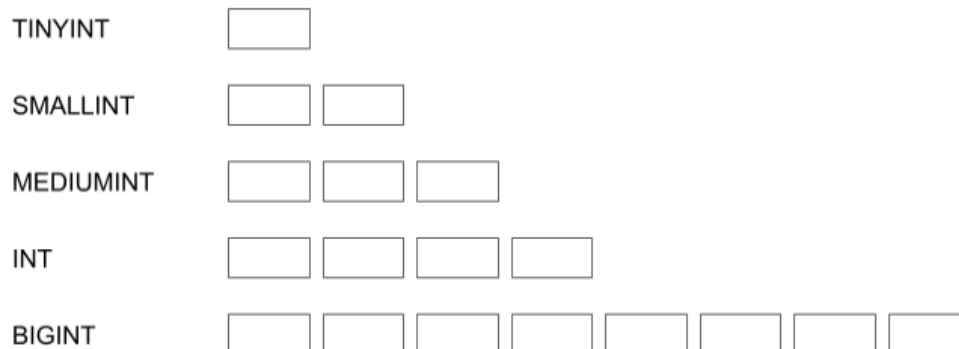
Атрибут `UNSIGNED` относится только к числовым значениям. Поле с таким атрибутом теряет возможность хранить отрицательные значения. Для многих полей, например, первичного ключа, отрицательные значения не требуются. Под кодирование знака отводится один бит. Отказ от него позволяет его высвободить под кодирование числа и увеличить максимально допустимый диапазон.



### Числовые типы

- целочисленные;
- вещественные, т.е., числа с плавающей точкой;
- точные — `DECIMAL`.

Целые числа обрабатываются быстрее всех, вещественные чуть медленнее, точные медленнее всех, так как это фактически строка, в которую записано число.



В MySQL предусмотрено целых 5 целых типов, прямоугольниками на рисунке показывается, сколько байт отводится под каждый из типов. Чем больше места занимает тип, тем объемнее будет конечная таблица и тем больше данные будут занимать места на жестком диске и в оперативной памяти. Кроме того, чем больше байт отводится под число, тем больший диапазон оно может обслуживать. В `TINYINT` один байт, т.е., 8 бит,

максимальное значение, которое он может обслуживать — от 0 до 2 в степени 8, т. е., 256 .  
Большое число в поле этого типа поместить не получится. Если при этом не используется атрибут UNSIGNED , то эту величину следует поделить пополам и допустимый диапазон — от -128 и до 127 .

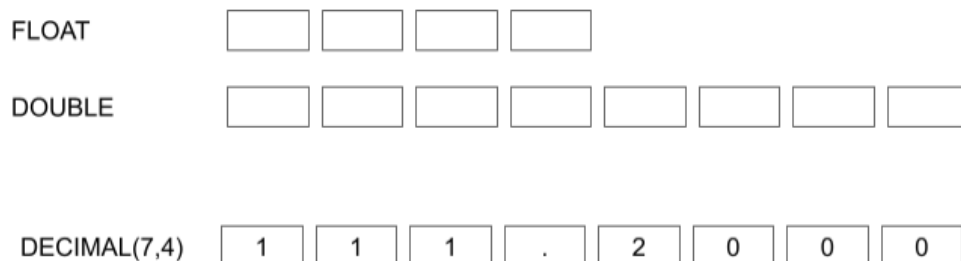
```
-- IF EXISTS - проверка на наличие таблицы или любого другого объекта.  
-- Таким образом можно запускать скрипт без ошибки о том, что объект уже создан.  
  
-- IF NOT EXISTS - проверка на отсутствие объекта  
  
CREATE TABLE IF NOT EXISTS tbl (id INT(8));  
-- Создадим таблицу, если ранее такой же не существовало  
  
INSERT INTO tbl VALUES (5);  
SELECT * FROM tbl;  
  
DROP TABLE IF EXISTS tbl; -- Удалить таблицу tbl, если она существует  
CREATE TABLE tbl (id INT(8) ZEROFILL);  
  
INSERT INTO tbl VALUES (5);  
INSERT INTO tbl VALUES (500000000);
```

При объявлении целого типа в круглых скобках можно задать количество отводимых под число символов. Это **необязательное** указание количества выводимых символов используется для дополнения пробелами слева. Однако ограничений ни на диапазон величин, ни на количество разрядов не налагается. Если количества символов, необходимых для вывода числа, будет недостаточно, под столбец будет выделено больше символов. Если дополнительно указан необязательный атрибут ZEROFILL , свободные позиции по умолчанию заполняются нулями слева.

Среди вещественных чисел различают FLOAT , который занимает 4 байта, и DOUBLE , занимающий 8 байт. Так как не вещественные числа можно закодировать при помощи двоичного кода, вычисления с участием вещественных чисел приводит к накоплению ошибок. Ряд областей, например, работа с деньгами очень чувствительна к таким ошибкам.

Поэтому в SQL предусмотрен специальный тип DECIMAL , в нем число хранится в виде строки, обрабатывается такой тип данных сильно медленнее, чем остальные числа, зато не теряется точность. Требуемая точность задается при объявлении столбца данных одного из этих типов.

На рисунке выше представлена схема типа DECIMAL , в котором под все число отводится 7 байт, а под дробную часть — 4 байта. Поместить сюда число больше 999 с четырьмя девятками после запятой уже не выйдет



### Строковые типы

Строковые типы можно условно разделить на:

- фиксированные строки, которые задаются типом CHAR, если при создании таблицы отводится 40 символов

— именно столько памяти и займет запись;

- переменные строки, которые задаются типом VARCHAR, не имеют фиксированного размера, занимаемый объем определяется размером строки; впрочем, допускается задание максимального объема строки в круглых скобках после запятой;
- BLOB-типы, которые изначально задумывались для хранения объемных бинарных данных, однако в результате были адаптированы для хранения текстовых значений.

#### Запись фиксированной длины

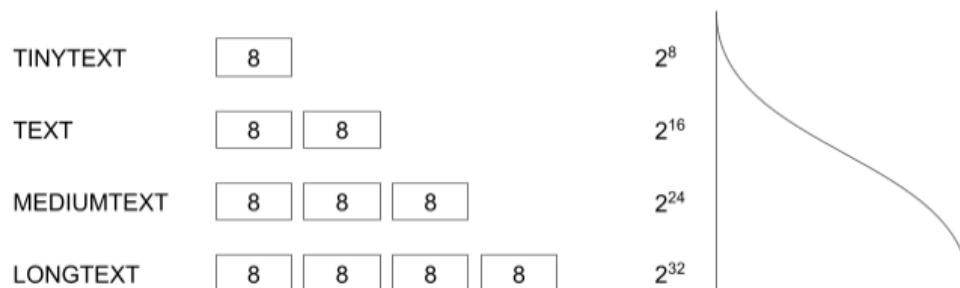
INT	INT	CHAR	CHAR
-----	-----	------	------

#### Запись переменной длины

INT	INT	VARCHAR, NULL
-----	-----	---------------

65536

Строковые типы имеют ограничения, записи в таблице представлены в виде структуры фиксированного размера. Это позволяет быстро переходить к нужной записи, так как размер известен заранее и мы можем перемещать указатель на нужный адрес. Под столбцы переменной длины отводится специальная область длиной **65 536** байт. Таким образом, нельзя в таблице создать столбцы **VARCHAR**, совокупный размер которых больше, чем эта специальная область. Так как для кодирования строк используется **UTF-8**, ситуация еще более печальная: для символов, отличных от английских, зачастую используется больше одного байта, например, русский текст кодируется аж двумя байтами.

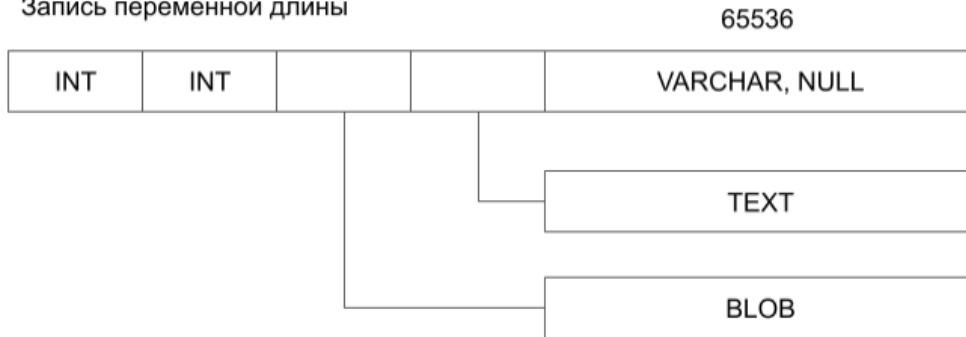


Поэтому для хранения объемного текста используется тип **TEXT**, который, как и **INT**, имеет несколько модификаций. На рисунке прямоугольниками указывается количество байт, которые используются для адресации внутри текстовой строки. Таким образом, при помощи семейства **TEXT**-типов можно закодировать как короткие строки на **256** символов, так и объемный текст вплоть до **4** Гб. Тип **TEXT** адаптирован под хранение текста из **BLOB**-типа, который используется для хранения бинарных данных. Впрочем, **BLOB** используется исключительно редко, гораздо чаще база данных используется для хранения текста.

Запись фиксированной длины

INT	INT	CHAR	CHAR
-----	-----	------	------

Запись переменной длины



Типы `TEXT` и `BLOB` еще медленнее, чем `VARCHAR`, так как они хранятся в отдельной области памяти, отделенной от данных основной таблицы. Поэтому при обращении к ним MySQL вынуждена осуществлять поиск этих данных и соединять их с основными данными записи. Поэтому прибегать к ним следует только, когда размера `VARCHAR` не хватает.

## Тип NULL

SQL поддерживает специальные типы данных, среди них выделяется `NULL` — неизвестное значение. Все операции с `NULL` в качестве результата возвращают `NULL`. `SELECT NULL + 2;`

```
SELECT NULL + 2;  
-- Любая операция с неизвестным значением, приводит к неизвестному результату
```

Имя столбца и таблицы придумать мы можем, но какие же типы данных бывают? Давайте разбираться, какие возможные типы данных можно применять для создания таблиц. После типов данных закрепим наши знания, заполнив БД, как минимум, 2 таблицами. Пример будет немного ниже.

## Календарные типы

MySQL поддерживает пять типов календарных типов:

- `TIME` предназначен для хранения времени в течение суток;
- `YEAR` хранит год;
- `DATE` хранит дату с точностью до дня;
- `DATETIME` хранит дату и время;
- `TIMESTAMP` также хранит дату и время, занимает в два раза меньше места, чем `DATETIME`, но может хранить только ограниченные даты — в интервале от 1970 года до 2038;

Кроме того, первый `TIMESTAMP`-столбец в таблице обновляется автоматически при операциях создания и обновления. `TIMESTAMP` хранит дату в UTC-формате.

В таблице представлены календарные типы, именно в таких форматах MySQL возвращает календарные значения.

Тип	Описание
YEAR	0000
DATE	'0000-00-00'
TIME	'00:00:00'
DATETIME	0000-00-00 00:00:00'
TIMESTAMP	'0000-00-00 00:00:00'

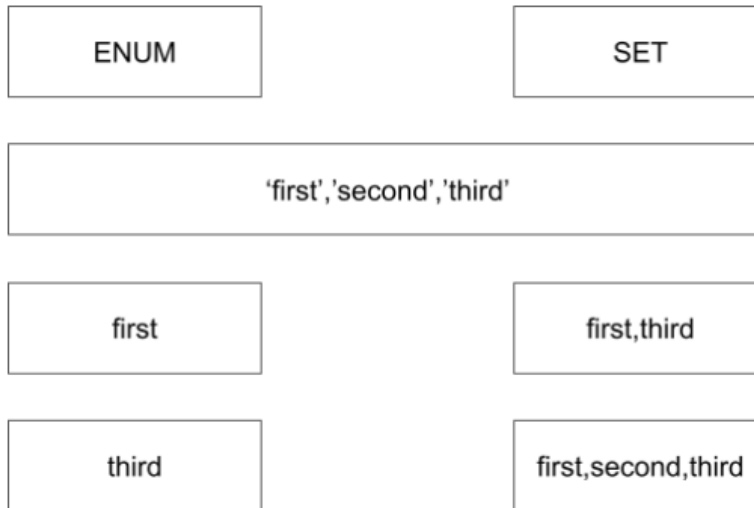
YEAR	<input type="text"/>
DATE	<input type="text"/> <input type="text"/> <input type="text"/>
TIME	<input type="text"/> <input type="text"/> <input type="text"/>
TIMESTAMP	<input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/>
DATETIME	<input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/>

С календарными типами можно проводить операции сложения и вычитания. Для этого используется специальная конструкция **INTERVAL** :

```
SELECT '2018-10-01 0:00:00' - INTERVAL 1 DAY;
SELECT '2018-10-01 0:00:00' + INTERVAL 1 WEEK;
SELECT '2018-10-01 0:00:00' + INTERVAL 1 YEAR;
SELECT '2018-10-01 0:00:00' + INTERVAL '1-1' YEAR_MONTH;
```

## Коллекционные типы

При объявлении списка допустимых значений **ENUM** и **SET** задаются списком строк, но во внутреннем представлении базы данных элементы множеств сохраняются в виде чисел. В случае **ENUM** поле может принимать лишь одно значение из списка. В случае **SET** — комбинацию заданных значений.



В последнее время большую популярность приобрел формат `JSON`, готовый объект языка JavaScript. Этот формат интенсивно используется для хранения и передачи коллекций. В `MySQL` предусмотрен столбец `JSON`-формата. Давайте добавим в таблицу `tbl` еще один столбец `JSON`-типа:

## JSON

JSON — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми. Формат JSON был разработан Дугласом Крокфордом.

🌐 <https://ru.wikipedia.org/wiki/JSON>



```
DESCRIBE tbl; -- Получим информацию о столбцах
ALTER TABLE tbl ADD collect JSON; -- Добавили столбец "collect" с типом JSON
DESCRIBE tbl; -- Получим информацию о столбцах с новым столбцом "collect"
```

- 1 • `DESCRIBE tbl;` -- Получим информацию о столбцах
- 2 • `ALTER TABLE tbl ADD collect JSON;` -- Добавили столбец "collect" с типом JSON
- 3 • `DESCRIBE tbl;` -- Получим информацию о столбцах с новым столбцом "collect"
- 4

Result Grid						
Filter Rows: <input type="text"/>						
Export:  Wrap Cell Content:						
	Field	Type	Null	Key	Default	Extra
▶	id	int	YES		<code>NULL</code>	
	collect	json	YES		<code>NULL</code>	

MySQL :: MySQL 8.0 Reference Manual :: 11.5 The JSON Data Type

<https://dev.mysql.com/doc/refman/8.0/en/json.html>



```
INSERT INTO tbl VALUES(1, '{"first": "Hello", "second": "World"}');
-- Пара ключ:значение
SELECT * FROM tbl;
SELECT collect->"$.first" FROM tbl; -- Получить значение по ключу "first" - "Hello"
SELECT collect->"$.second" FROM tbl -- Получить значение по ключу "second" - "World"
```

### Декодирование типа данных JSON MySQL

В этом посте мы собираемся исследовать тип данных JSON в MySQL 5.7 и во время погружения будем использовать фреймворк Laravel для построения запросов. Для начала, создадим новую таблицу: CREATE TABLE...

<https://habr.com/ru/post/279155/>

Хабр

Декодирование типа данных JSON MySQL

Разработка

## БД для работы. Интернет-магазин

CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP :

MySQL :: MySQL 8.0 Reference Manual :: 11.2.5 Automatic Initialization and Updating for TIMESTAMP and DATETIME

<https://dev.mysql.com/doc/refman/8.0/en/timestamp-initialization.html>

CURRENT\_TIMESTAMP :

MySQL :: MySQL 8.0 Reference Manual :: 11.2.5 Automatic Initialization and Updating for TIMESTAMP and DATETIME

<https://dev.mysql.com/doc/refman/8.0/en/timestamp-initialization.html>

```
-- CURRENT_TIMESTAMP возвращает текущую дату и время
-- COMMENT - комментарий к объекту
```

```
DROP TABLE IF EXISTS products;
```

```
CREATE TABLE products (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) COMMENT 'Название',
  description TEXT COMMENT 'Описание',
  price DECIMAL (11,2) COMMENT 'Цена',
  catalog_id INT UNSIGNED,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Товарные позиции';
```

```
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);
```

```
CREATE TABLE users (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  name VARCHAR(255) COMMENT 'Имя покупателя',  
  birthday_at DATE COMMENT 'Дата рождения',  
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,  
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP  
) COMMENT = 'Покупатели';  
INSERT INTO users (name, birthday_at)  
VALUES  
  ('Геннадий', '1990-10-05'),  
  ('Наталья', '1984-11-12'),  
  ('Александр', '1985-05-20'),  
  ('Сергей', '1988-02-14'),  
  ('Иван', '2001-01-12'),  
  ('Мария', '2002-08-29');
```

## 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. По умолчанию сортировка производится в прямом порядке `ASC`, однако, добавив после имени столбца ключевое слово `DESC`, можно добиться сортировки в обратном порядке.

Синтаксис:

```
SELECT expressions  
  
FROM tables  
  
[WHERE conditions]  
  
ORDER BY expression [ ASC | DESC ];
```

### Пример:

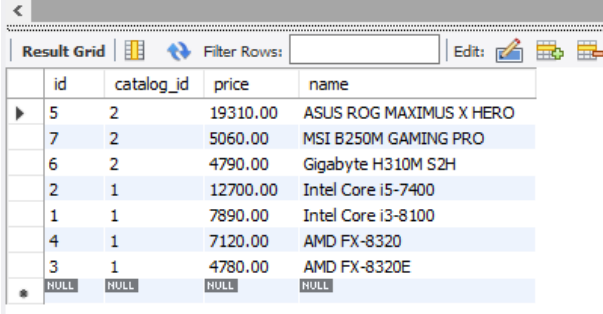
```
SELECT * FROM products ORDER BY id; -- Запрос сортирует результат выборки по полю id  
  
-- Чтобы отсортировать таблицу по каталогам,  
-- в рамках каждого каталога, по цене, мы можем указать  
-- после ключевого слова ORDER BY сначала поле catalog_id, а затем поле price:  
  
SELECT id, catalog_id, price, name FROM products ORDER BY catalog_id, price;
```

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;
```

Если в `ORDER BY` указаны 2 столбца, то сначала сортировка производится по первому столбцу, затем - по второму. Рассмотрим пример для запроса выше:

```
39 • SELECT id, catalog_id, price, name
40     FROM products
41     ORDER BY catalog_id DESC, price DESC
42
```



The screenshot shows a database query result grid. The query is: `SELECT id, catalog_id, price, name FROM products ORDER BY catalog_id DESC, price DESC`. The result grid has columns: id, catalog\_id, price, and name. The data is sorted by catalog\_id in descending order, and then by price in descending order for each catalog\_id. The rows are: (5, 2, 19310.00, ASUS ROG MAXIMUS X HERO), (7, 2, 5060.00, MSI B250M GAMING PRO), (6, 2, 4790.00, Gigabyte H310M S2H), (2, 1, 12700.00, Intel Core i5-7400), (1, 1, 7890.00, Intel Core i3-8100), (4, 1, 7120.00, AMD FX-8320), (3, 1, 4780.00, AMD FX-8320E), and a row with NULL values.

	id	catalog_id	price	name
▶	5	2	19310.00	ASUS ROG MAXIMUS X HERO
	7	2	5060.00	MSI B250M GAMING PRO
	6	2	4790.00	Gigabyte H310M S2H
	2	1	12700.00	Intel Core i5-7400
	1	1	7890.00	Intel Core i3-8100
	4	1	7120.00	AMD FX-8320
	3	1	4780.00	AMD FX-8320E
*	NULL	NULL	NULL	NULL

1. Сортировка по убыванию (от большего к меньшему) по столбцу `id_catalog` : от 2 до 1.
2. Для данных, которые отсортированы по убыванию, происходит еще 1 сортировка: по убыванию цены.

## Ограничения выборки. Оператор LIMIT

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. Для этого перед именем столбца можно использовать ключевое слово **DISTINCT**:

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

Данный запрос возвращает уникальные номера каталогов, без повторений

## Группировка данных

Мы уже знаем механизм получения уникальных значений. В языке SQL для работы с такими группами предназначено специальное ключевое слово **GROUP BY**.

Синтаксис: (в “[]” указаны необязательные части)

```
SELECT столбцы
FROM таблица
[WHERE условие_фильтрации_строк]
[GROUP BY столбцы_для_группировки]
[HAVING условие_фильтрации_групп]
[ORDER BY столбцы_для_сортировки]
```

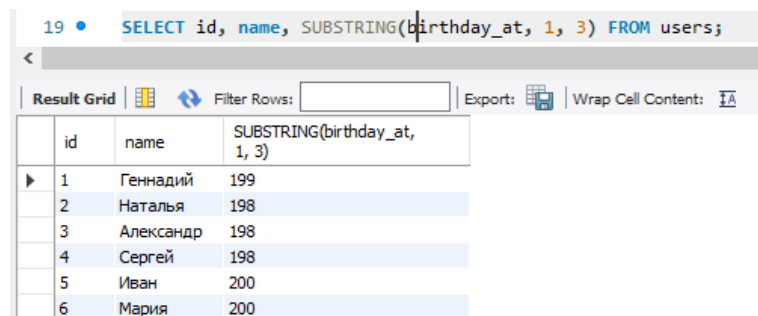
Пример:

```
-- Уникальные значения
SELECT DISTINCT catalog_id FROM products ORDER BY catalog_id;
-- Группировка данных через GROUP BY
SELECT catalog_id FROM products GROUP BY catalog_id;
```

В качестве значений для создания групп могут выступать не только столбцы таблицы, но и вычисляемые значения. Например, давайте разделим пользователей в таблице на три группы: родившихся в 80-х, 90-х и 2000-х годах. Для этого из даты рождения можно получить только первые 3 цифры. К примеру:

- 2001 год - первые 3 цифры - это “200” - обозначает, что человек родился в 2000 годах
- 1991 год - первые 3 цифры - это “199” - обозначает, что человек родился в 90 - х годах

```
SELECT id, name, SUBSTRING(birthday_at, 1, 3) FROM users;
```

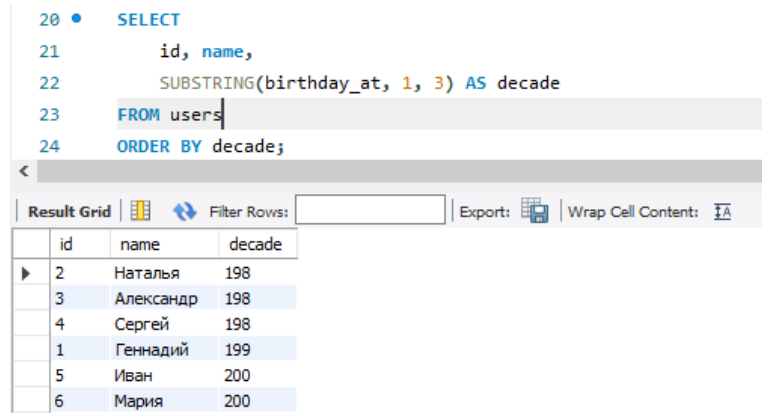


The screenshot shows a SQL query editor with the following query: `SELECT id, name, SUBSTRING(birthday_at, 1, 3) FROM users;`. Below the query, there is a table grid displaying the results. The table has three columns: `id`, `name`, and `SUBSTRING(birthday_at, 1, 3)`. The results are as follows:

	id	name	SUBSTRING(birthday_at, 1, 3)
▶	1	Геннадий	199
	2	Наталья	198
	3	Александр	198
	4	Сергей	198
	5	Иван	200
	6	Мария	200

Здесь мы преобразуем календарный тип `DATETIME` поля `birthday_at` к строковому значению и при помощи функции `SUBSTRING` `SUBSTRING` извлекаем первые три цифры года рождения. Давайте назначим вычисляемому значению псевдоним при помощи ключевого слова `AS` и отсортируем значения при помощи `ORDER BY`

```
SELECT
    id, name,
    SUBSTRING(birthday_at, 1, 3) AS decade
FROM users
ORDER BY decade;
```



The screenshot shows a SQL query editor with the following code:

```
20 • SELECT
21     id, name,
22     SUBSTRING(birthday_at, 1, 3) AS decade
23 FROM users
24 ORDER BY decade;
```

Below the editor is a 'Result Grid' with the following data:

	id	name	decade
▶	2	Наталья	198
	3	Александр	198
	4	Сергей	198
	1	Геннадий	199
	5	Иван	200
	6	Мария	200

Обратите внимание, что мы можем использовать псевдоним `decade` в конструкции `ORDER BY`. При помощи конструкции `GROUP BY` мы можем сгруппировать поля по декадам.

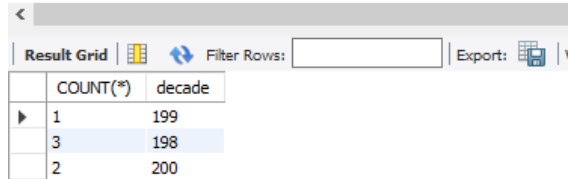
Каждая из групп содержит в себе несколько пользователей и непонятно, какого из них следует выводить. Ранее MySQL выводила случайного пользователя, однако сейчас такое поведение отменено. Такой режим по-прежнему можно включить, СУБД даже подсказывает в сообщении об ошибке, как это можно сделать. Однако лучше этого не делать, чтобы ваш SQL-код оставался совместимым с другими СУБД. Какую пользу можно извлечь из сгруппированных значений? MySQL предоставляет несколько функций, которые называются агрегатными. Они позволяют работать с содержимым групп, полученных `GROUP BY`. Например, мы можем подсчитать количество записей внутри каждой из групп:

```
SELECT COUNT(*),
    SUBSTRING(birthday_at, 1, 3) AS decade
FROM users
GROUP BY decade;
```

```

26 • SELECT COUNT(*),
27     SUBSTRING(birthday_at, 1, 3) AS decade
28 FROM users
29 GROUP BY decade;
30

```



	COUNT(*)	decade
1	199	
3	198	
2	200	

Таким образом, у нас 3 пользователя родились в 80-х, два в 90-х и один в 2000-х. Полученные значения мы по-прежнему можем сортировать при помощи конструкции `ORDER BY`.

```

SELECT
    COUNT(*) AS total,
    SUBSTRING(birthday_at, 1, 3) AS decade
FROM users
GROUP BY decade
ORDER BY total DESC;

```

Причем сортировать можно не группируемому значению, но и по любому другому полю. Например, давайте назовем функции `COUNT()` псевдоним `total` и отсортируем результаты по этому значению:

```

SELECT
    COUNT(*) AS total,
    SUBSTRING(birthday_at, 1, 3) AS decade
FROM users
GROUP BY decade
ORDER BY total DESC;

```

Посмотреть содержимое группы мы можем при помощи специальной функции `GROUP_CONCAT`:

```

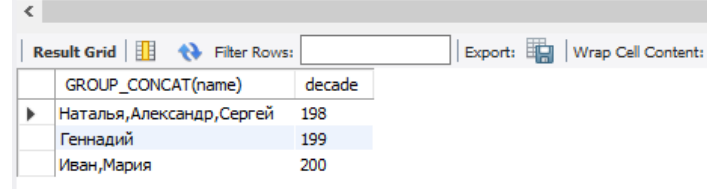
SELECT
    GROUP_CONCAT(name),
    SUBSTRING(birthday_at, 1, 3) AS decade
FROM users
GROUP BY decade;

```

```

26 • SELECT
27     GROUP_CONCAT(name),
28     SUBSTRING(birthday_at, 1, 3) AS decade
29 FROM users
30 GROUP BY decade;
31

```



GROUP_CONCAT(name)	decade
Наталья,Александр,Сергей	198
Геннадий	199
Иван,Мария	200

Функция `GROUP_CONCAT` допускает задание разделителя, для этого внутри функции используется ключевое слово `SEPARATOR`. Давайте зададим в качестве разделителя пробел:

```
SELECT
  GROUP_CONCAT(name SEPARATOR ' '),
  SUBSTRING(birthday_at, 1, 3) AS decade
FROM users
GROUP BY decade;
```

26 • **SELECT**

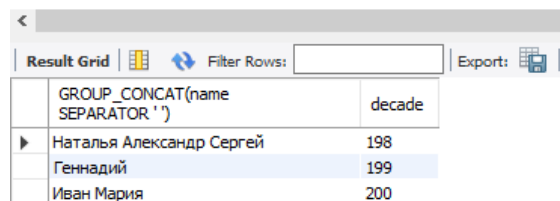
27     GROUP\_CONCAT(name SEPARATOR ' '),

28     SUBSTRING(birthday\_at, 1, 3) AS decade

29   FROM users

30   GROUP BY decade;

31



GROUP_CONCAT(name SEPARATOR ' ')	decade
Наталья Александр Сергей	198
Геннадий	199
Иван Мария	200

Ключевое слово `ORDER BY` позволяет отсортировать значения в рамках возвращаемой строки. Давайте отсортируем имена пользователей в обратном порядке:

```
SELECT
  GROUP_CONCAT(name ORDER BY name DESC SEPARATOR ' '),
  SUBSTRING(birthday_at, 1, 3) AS decade
FROM users
GROUP BY decade;
```

26 • **SELECT**

27     GROUP\_CONCAT(name ORDER BY name DESC SEPARATOR ' '),

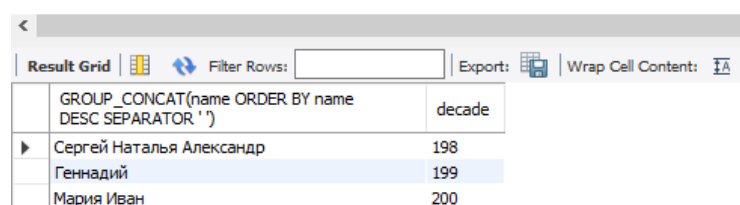
28     SUBSTRING(birthday\_at, 1, 3) AS decade

29   FROM users

30   GROUP BY decade;

31

32

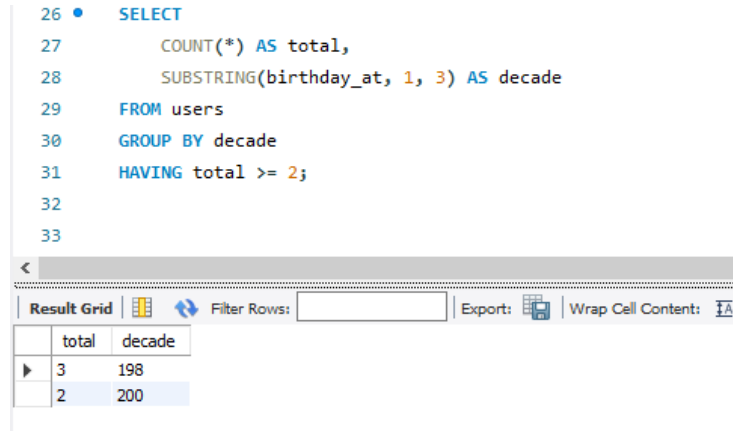


GROUP_CONCAT(name ORDER BY name DESC SEPARATOR ' ')	decade
Сергей Наталья Александр	198
Геннадий	199
Мария Иван	200

Агрегационные функции позволяют получать результаты для каждой из групп в отдельности. Чаше при составлении условий требуется ограничить выборку по результату функции, например выбрать группы, где количество записей больше или равно двум.

Использование для этих целей конструкции `WHERE` приводит к ошибке. Для решения этой проблемы вместо ключевого слова `WHERE` используется ключевое слово `HAVING`, которое располагается вслед за конструкцией `GROUP BY`. Выберем группы людей (рожденных в 80, 90, 2000х), где количество людей больше, чем 2 человека:

```
SELECT
    COUNT(*) AS total,
    SUBSTRING(birthday_at, 1, 3) AS decade
FROM users
GROUP BY decade
HAVING total >= 2;
```



## Агрегатные функции

**Агрегатные функции (агрегации)** — это функции, которые вычисляются от группы значений и объединяют их в одно результирующее.

Количество записей в таблице можно узнать при помощи функции `COUNT()`, которая принимает в качестве аргумента имя столбца. Функция возвращает число строк в таблице, значения столбца для которых отличны от `NULL`.

В качестве параметра функции наряду с именами столбцов может выступать символ звездочки (\*). При использовании символа \* будет возвращено число строк таблицы независимо от того, принимают какие-то из них значение NULL или нет.

```
SELECT COUNT(*) FROM catalogs; -- Количество строк в таблице
SELECT COUNT(id) FROM catalogs; -- Количество id-шников в таблице
```

Конструкция `GROUP BY` разбивает таблицу на отдельные группы. Функция `COUNT()` возвращает результат для каждой из этих групп.

```
SELECT
    catalog_id,
    COUNT(*) AS total
FROM products
GROUP BY catalog_id;
```



	catalog_id	total
▶	1	4
	2	3

Функции `MIN()` и `MAX()` возвращают минимальное и максимальное значения столбца.

Получим максимальную и минимальную цену:

```
SELECT
  MIN(price) AS min,
  MAX(price) AS max
FROM products;
```

Сверху - таблица искомая, снизу - результат запроса:

	id	name	description	price	catalog_id	created_at	updated_at
▶	1	Intel Core i3-8100	Процессор для настольных персональных ко...	7890.00	1	2023-03-17 13:33:44	2023-03-17 13:33:44
	2	Intel Core i5-7400	Процессор для настольных персональных ко...	12700.00	1	2023-03-17 13:33:44	2023-03-17 13:33:44
	3	AMD FX-8320E	Процессор для настольных персональных ко...	4780.00	1	2023-03-17 13:33:44	2023-03-17 13:33:44
	4	AMD FX-8320	Процессор для настольных персональных ко...	7120.00	1	2023-03-17 13:33:44	2023-03-17 13:33:44
	5	ASUS ROG MAXIMUS X HERO	Материнская плата ASUS ROG MAXIMUS X HE...	19310.00	2	2023-03-17 13:33:44	2023-03-17 13:33:44
	6	Gigabyte H310M S2H	Материнская плата Gigabyte H310M S2H, H31...	4790.00	2	2023-03-17 13:33:44	2023-03-17 13:33:44
	7	MSI B250M GAMING PRO	Материнская плата MSI B250M GAMING PRO, ...	5060.00	2	2023-03-17 13:33:44	2023-03-17 13:33:44
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

	min	max
▶	4780.00	19310.00

Функция `AVG()` возвращает среднее значение аргумента. Давайте подсчитаем среднюю цену товара в нашем магазине. Чтобы проверить поиск среднего значения, вспомним формулу: сумма всех значений / количество. Если решать задачу поиска среднего, то можно решить эту же задачу математически: `SUM(price) / COUNT(price)`

Для поиска суммы используем `SUM()` - поиск суммы внутри столбца.

```
SELECT
  AVG(price) AS "Средняя цена через AVG",
  SUM(price) AS "Сумма товаров через SUM",
  COUNT(price) AS "Количество всех товаров через COUNT",
  SUM(price) / COUNT(price) AS "Проверка ср. арифм"
FROM products;
```

	Средняя цена через AVG	Сумма товаров через SUM	Количество всех товаров через COUNT	Проверка ср. арифм
▶	8807.142857	61650.00	7	8807.142857

## Многотабличные запросы

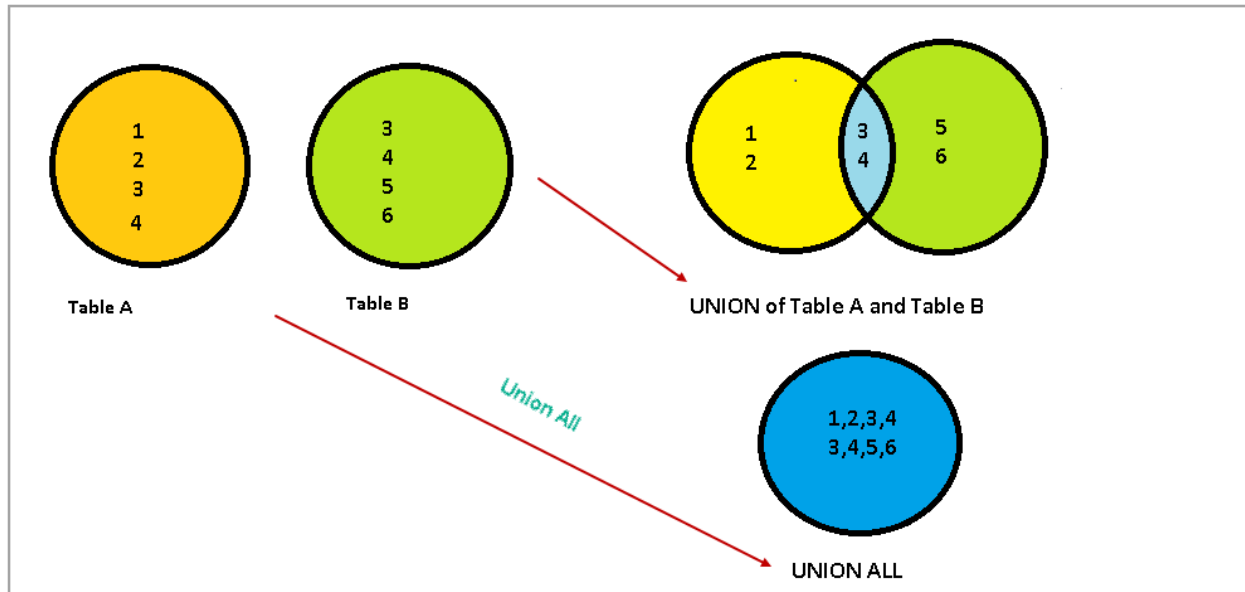
До этого мы обращались только к одной таблице, но настало время попробовать многотабличные запросы, результат в которых можно формировать из двух и более таблиц.

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.

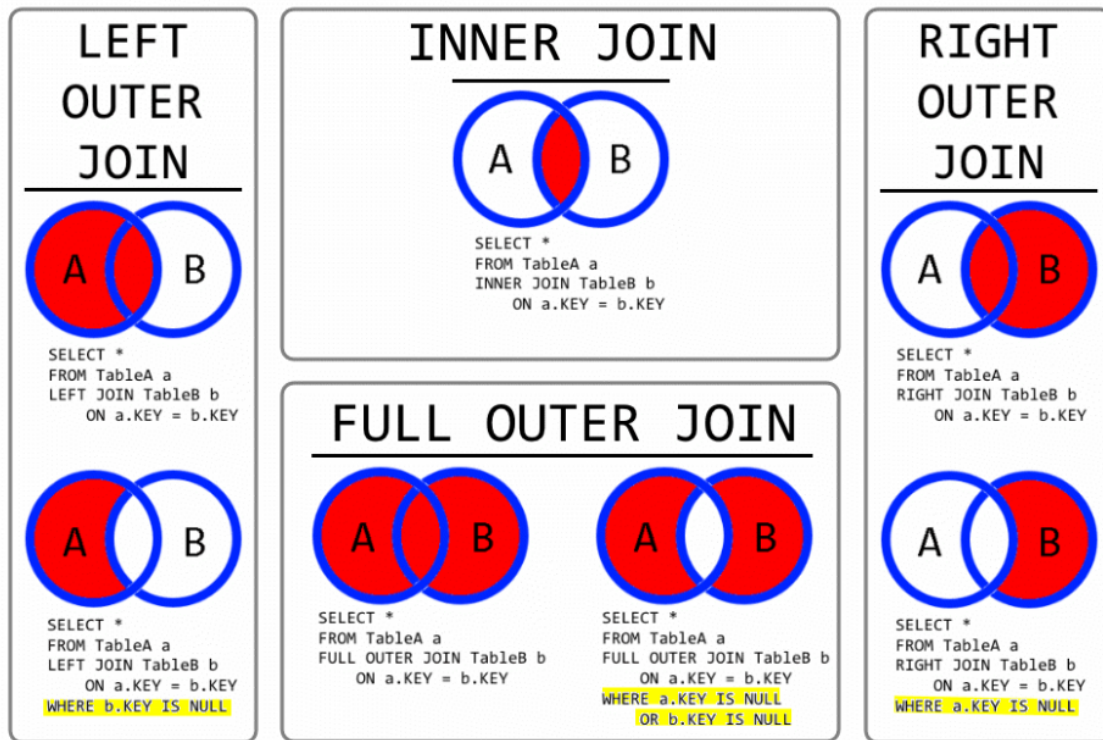


Вложенный запрос позволяет использовать результат, возвращаемый одним запросом, в другом. Здесь синим цветом представлены точки в запросе, где мы можем использовать вложенные запросы.

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

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».

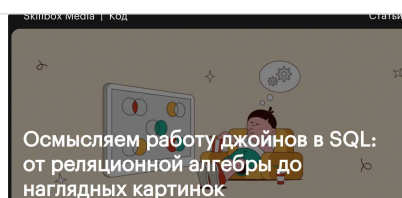
# ANSI SQL JOIN



Очень неплохое графическое представление JOIN-ов. При первом знакомстве, еще на курсе из 3 семинаров от Ильнара, показываю этот пример:

Осмыслием работу джойнов в SQL: от реляционной алгебры до наглядных картинок  
Выбираем, какие фильмы посмотреть, с помощью соединения данных в SQL.

[https://skillbox.ru/media/code/osmyslyaem\\_rabotu\\_dzhoynov\\_v\\_sql\\_ot\\_relyatsionnoy\\_algebry\\_d\\_o\\_naglyadnykh\\_kartinok/](https://skillbox.ru/media/code/osmyslyaem_rabotu_dzhoynov_v_sql_ot_relyatsionnoy_algebry_d_o_naglyadnykh_kartinok/)



## 1. 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 catalogs;  
CREATE TABLE catalogs (  
  id INT UNSIGNED,
```

```

    name VARCHAR(255) COMMENT 'Название раздела'
) COMMENT = 'Разделы интернет-магазина';

INSERT INTO catalogs (name) VALUES ('Процессоры');
INSERT INTO catalogs VALUES (0, 'Мат.платы');
INSERT INTO catalogs VALUES (NULL, 'Видеокарты');

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');
```

Давайте попробуем получить уникальные данные из 2 таблиц. Для этого используем **UNION**:

```

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

catalogs

name
Процессоры
Мат.платы
Видеокарты

rubrics

name
Видеокарты
Память

```

SELECT
name
FROM
catalogs
```

**UNION**

```

SELECT
name
FROM
rubrics
```

```

ORDER BY
name;
```

name
Видеокарты
Мат.платы
Память
Процессоры

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.

Если необходимо вывести все данные без повторения, используется оператор **UNION ALL**:

```

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

catalogs		
name	SELECT	
Процессоры	name	
Мат.платы	FROM	
Видеокарты	catalogs	
	UNION ALL	
rubrics	SELECT	
name	name	
Видеокарты	FROM	
Память	rubrics	
	ORDER BY	
	name;	
		name
		Видеокарты
		Видеокарты
		Мат.платы
		Память
		Процессоры

## 2. 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:

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

Чтобы СУБД могла отличать основной запрос и подзапрос, последний заключают в круглые скобки. Попробуем вернуть максимальную цену из всех товаров, но без агрегатной функции:

```
SELECT *,
  (SELECT MAX(price) FROM products ) AS 'max_price'
FROM products;
```

Здесь мы видим максимальную цену у товара "Материнская плата ASUS ROG MAXIMUS X HERO, Z370, Socket 1151-V2, DDR4, ATX" стоимостью 19310.00.

Большинство подзапрос мы будем писать на семинарах 😊

## 3. JOIN - соединения таблиц

1. **INNER JOIN**: возвращает записи с совпадающими значениями в обеих таблицах.
2. **LEFT JOIN**: возвращает все записи из левой таблицы и соответствующие записи из правой таблицы.
3. **RIGHT JOIN**: возвращает все записи из правой таблицы и соответствующие записи из левой таблицы.
4. **CROSS JOIN**: возвращает все записи из обеих таблиц.

Рассмотрим каждый из типов:

### 1. CROSS JOIN

Первый тип соединения - декартово произведение или CROSS JOIN.

В MySQL CROSS JOIN генерирует результирующий набор, который является произведением строк двух связанных таблиц. Каждая строка одной таблицы соединяется с каждой строкой второй таблицы, давая тем самым в результате все возможные сочетания строк двух таблиц.

```
CREATE TABLE fst(
  value VARCHAR(255)
);
INSERT INTO fst
VALUES
  ('fst1'),
  ('fst2'),
  ('fst3');
CREATE TABLE snd(
  value VARCHAR(255)
);
INSERT INTO snd
VALUES
  ('snd1'),
  ('snd2'),
  ('snd3');
```

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

fst

value
fst1
fst2
fst3

snd

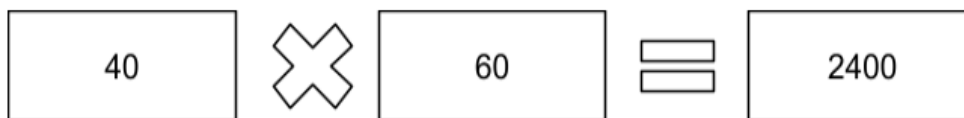
value
snd1
snd2
snd3

```
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

Размер в таком случае можно определить по формуле:  $M * N$ , где **M** - количество строк в первой таблице, **N** - количество строк во второй таблице. Если **M = 40**, **N = 60**, то **CROSS JOIN** вернет нам **2400** строк.

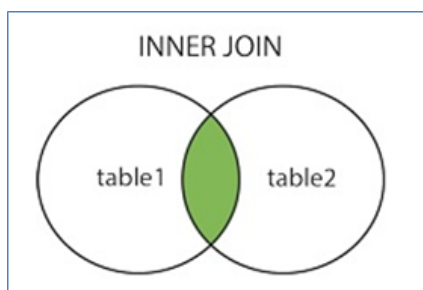


## 2. INNER JOIN

Оператору передаются две таблицы, и он возвращает их внутреннее пересечение по какому-либо критерию. Результатом будут записи, которые соответствуют обеим таблицам, — их перед отправкой объединят.

### Пример(для понимания):

Например, если в одной таблице будут перечислены черные животные, а в другой — собаки, то **INNER JOIN** вернет одну таблицу с перечислением черных собак. Столбцы будут «склеены» друг с другом, несмотря на то что в базе данные хранятся в разных таблицах. Это похоже на бинарное «и» из алгебры логики.



```
SELECT столбцы
FROM таблица1
  [INNER] JOIN таблица2
    ON условие1
  [[INNER] JOIN таблица3
    ON условие2]
```

### Пример(кодом):

Возвращает записи с совпадающими значениями в обеих таблицах. Модифицируем наши таблицы:

```
DROP TABLE IF EXISTS catalog;
CREATE TABLE catalog (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) COMMENT 'Название раздела',
  UNIQUE unique_name(name(10))
) COMMENT = 'Разделы интернет-магазина';

INSERT INTO catalog (name)
VALUES
  ('Процессоры'), -- id = 1
  ('Материнские платы'), -- id = 2
  ('Видеокарты'), -- id = 3
  ('Жесткие диски'), -- id = 4
  ('Оперативная память'); -- id = 5

DROP TABLE IF EXISTS product;
CREATE TABLE product (
  id INT PRIMARY KEY AUTO_INCREMENT,
```

```

name VARCHAR(255) COMMENT 'Название',
description TEXT COMMENT 'Описание',
price DECIMAL (11,2) COMMENT 'Цена',
catalog_id INT,
created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
FOREIGN KEY (catalog_id) REFERENCES catalog(id)
) COMMENT = 'Товарные позиции';
SELECT * FROM catalog;

INSERT product(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);

```

Выведем товар и его группу. Для этого выберем столбцы, которые нас интересуют из каждой таблицы. Мы видим, что в табличке products имеется ссылка на табличку catalogs . Если эти ссылки равны, то мы узнаем товар и принадлежность к группе.

Допустим, имеется

**('Intel Core i3-8100', 'Процессор для настольных персональных компьютеров, основанных на платформе Intel.', 7890.00, 1);**

В конце указана ссылка **catalog\_id(внешний ключ) - группа №1 из таблицы catalog.**

Это ссылка на первичный ключ из таблицы catalog (FOREIGN KEY (catalog\_id) REFERENCES catalog(id)). Указав равенство ключей, мы можем сделать вывод, что 'Intel Core i3-8100' принадлежит к группе с номером №1: 'Процессоры'.

```

SELECT p.name,
       p.price,
       c.name
FROM catalog AS c
JOIN product AS p
ON c.id = p.catalog_id;

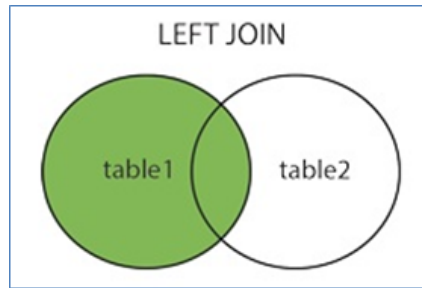
```

	name	price	name
►	Intel Core i3-8100	7890.00	Процессоры
	Intel Core i5-7400	12700.00	Процессоры
	AMD FX-8320E	4780.00	Процессоры
	AMD FX-8320	7120.00	Процессоры
	ASUS ROG MAXIMUS X HERO	19310.00	Материнские платы
	Gigabyte H310M S2H	4790.00	Материнские платы
	MSI B250M GAMING PRO	5060.00	Материнские платы

### 3. LEFT JOIN.

**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.





### Пример(для понимания):

Возвращает пересечение множеств и все элементы из левой таблицы. Например, человек хочет посмотреть кино, но на русский фильм согласен, только если это боевик. Фильтр вернет ему все фильмы из множества «боевики», фильмы из подмножества «русские боевики», но других фильмов из множества «русские» там не будет.

### Пример(кодом):

Допустим, попробуем получить полную информацию о каталоге товаров. Даже если товар не продается, мы будем его выводить. Для этого нам нужен LEFT JOIN. В запросе первая таблица всегда является **левой**, вторая - **правой**. Для запроса выше левая таблица - **catalog**, правая - **product**.

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. Обратите внимание, что в **catalog** есть группы видеокарт, жестких дисков и процессоров, а в табличке **product** таких товаров нет.

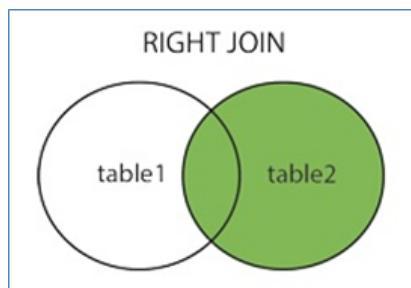
**Если совпадений не найдено, возвращается значение NULL.**

```
SELECT
  p.name,
  p.price,
  c.name
FROM catalog AS c -- Берется левая таблица и совпадения из правой
LEFT JOIN product AS p -- Правая таблица
ON c.id = p.catalog_id; -- Условие для совпадения (область пересечения 2 множеств)
```

```
43 • SELECT
44     p.name,
45     p.price,
46     c.name
47 FROM catalog AS c -- Берется левая таблица и совпадения из правой
48 LEFT JOIN product AS p -- Правая таблица
49 ON c.id = p.catalog_id; -- Условие для совпадения (область пересечения 2 множеств)
```

name	price	name
Intel Core i3-8100	7890.00	Процессоры
Intel Core i5-7400	12700.00	Процессоры
AMD FX-8320E	4780.00	Процессоры
AMD FX-8320	7120.00	Процессоры
ASUS ROG MAXIMUS X HERO	19310.00	Материнские платы
Gigabyte H310M S2H	4790.00	Материнские платы
MSI B250M GAMING PRO	5060.00	Материнские платы
NULL	NULL	Видеокарты
NULL	NULL	Жесткие диски
NULL	NULL	Оперативная память

#### 4. RIGHT JOIN



##### Пример(для понимания):

Работает по тому же принципу, но вместо левой таблицы — правая. То есть человек получит в результатах боевики, только если они русские.

##### Пример(кодом):

Аналогично, можем посмотреть товары и группы, к которым они принадлежат.

```
SELECT * FROM catalogs;
SELECT
  p.name,
  p.price,
  c.name
FROM catalog AS c -- Левая таблица
RIGHT JOIN product AS p -- Правая таблица
ON c.id = p.catalog_id; -- Условие для совпадения (область пересечения 2 множеств)
```

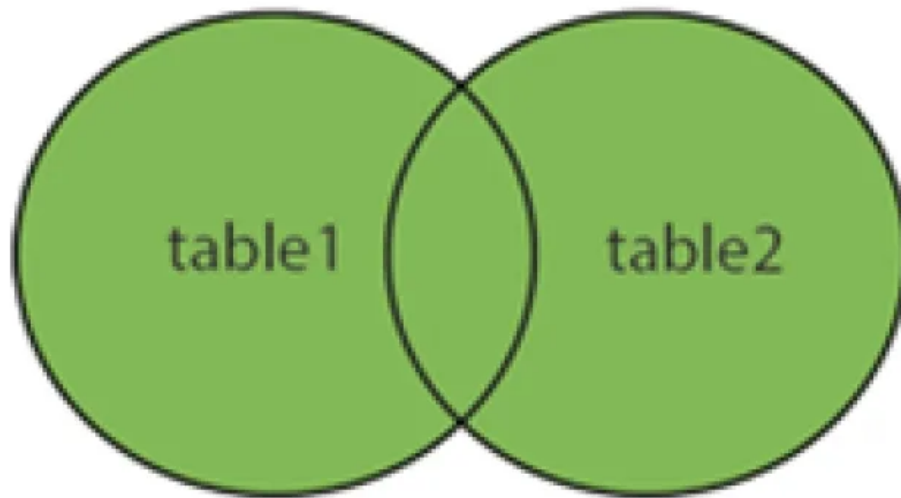
```
43 • SELECT
44     p.name,
45     p.price,
46     c.name
47 FROM catalog AS c -- Левая таблица
48 RIGHT JOIN product AS p -- Правая таблица
49 ON c.id = p.catalog_id; -- Условие для совпадения (область пересечения 2 множеств)
```

name	price	name
Intel Core i3-8100	7890.00	Процессоры
Intel Core i5-7400	12700.00	Процессоры
AMD FX-8320E	4780.00	Процессоры
AMD FX-8320	7120.00	Процессоры
ASUS ROG MAXIMUS X HERO	19310.00	Материнские платы
Gigabyte H310M S2H	4790.00	Материнские платы
MSI B250M GAMING PRO	5060.00	Материнские платы

Запрос очень напоминает INNER JOIN, но почему? Дело в том, что продукт имеет ссылку на каталог. Нет такого значения, которое бы не ссылалось на каталог (имеется внешний ключ, который ссылается ТОЛЬКО на реальные объекты в вашей БД).

Так же имеет смысл рассмотреть **FULL JOIN** в MySQL. **FULL JOIN** возвращает обе таблицы, объединенные в одну. Данный тип соединения не поддерживается, так как операция считается избыточной. Мы можем реализовать **FULL JOIN**, объединив два запроса при помощи **UNION**, оставив только уникальные значения

или **UNION ALL**, который не удаляет дубликаты. В первом запросе будем использовать **LEFT JOIN**, а во втором - **RIGHT JOIN**.



**Пример(для понимания):**

Например, человек хочет увидеть список из всех боевиков и всех русских фильмов, без исключений.

**Пример(кодом):**

Попробуем объединить табличку **product** с таблицей **catalog**, удалив дубликаты.

```
SELECT p.name, p.price, c.name
FROM catalog AS c
LEFT JOIN product AS p
ON c.id = p.catalog_id

UNION -- Удалили дубликаты

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

Иногда имеет смысл не удалять дубликаты. В таком случае используется оператор **UNION ALL**.

```
SELECT p.name, p.price, c.name
FROM catalog AS c
LEFT JOIN product AS p
ON c.id = p.catalog_id
```

```
UNION ALL -- дубликаты не удалили
```

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

## Парочка вопросов из реальных собеседований

1. Представьте, что запрос с оператором **UNION** и **UNION ALL** вернул одинаковое количество строчек (пусть вернулось 10 строчек). Какой из операторов сработал быстрее и почему?

**Ответ:** сработает быстрее UNION ALL, оператор не делает проверку на уникальность

2. Чем отличается **WHERE** от **HAVING**?

**Ответ:** HAVING очень похож на WHERE - это фильтр. Мы можем написать в HAVING name = 'AMD FX-8320', как и в WHERE. Ошибки не будет.

В **HAVING** и только в нём можно писать условия по агрегатным функциям (SUM, COUNT, MAX, MIN, AVG). Если мы хотим сделать что-то вроде **SUM(salary) > 1000000**, то это возможно сделать только в **HAVING**. Но мы могли оставить только **HAVING**, зачем сложности с оператором **WHERE**?

Загвоздка в выполнении запроса, работа с данными и порядок. **WHERE** выполняется на раннем этапе, до **GROUP BY**, чтобы экономить время выполнения запроса и ресурсы сервера.

Далее мы применяем GROUP BY, который делит данные на группы. На группы можно накладывать условия на результаты агрегатных функций.

Главное отличие HAVING от WHERE в том, что в HAVING можно наложить условия на результаты группировки, потому что порядок исполнения запроса устроен таким образом, что на этапе, когда выполняется WHERE, ещё нет групп, а HAVING выполняется уже после формирования групп. Оператор HAVING работает так же с агрегатными функциями, а WHERE - нет.

### Используемые источники

1. <https://dev.mysql.com/doc/refman/5.7/en/group-by-functions-and-modifiers.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.
9. <https://dev.mysql.com/doc/refman/5.7/en/union.html>.

10. <https://dev.mysql.com/doc/refman/5.7/en/subqueries.html>

11. <https://dev.mysql.com/doc/refman/5.7/en/join.html>

**Книги:**

- "Изучаем SQL", книга Бейли Л.
- Алан Бьюли "Изучаем SQL" (2007)
- Энтони Молинаро "SQL. Сборник рецептов" (2009)