MySQL

ФУНКЦИИ

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

- В предыдущих уроках мы уже встречали ряд очень полезных инструментов внутри MySQL, которые, например, помогали нам объединять несколько строк в одну (**CONCAT**), узнавать длину строки (**CHAR_LENGTH**) или даже позволяли получать некое статистическое значение для колонок какой-либо таблицы (**SUM**, **AVG**, **MIN**, **MAX**). Такие инструменты называются функциями - специальные программы (они же блоки кода), которые могут принимать значения, производить над ними определенные операции, а затем возвращать единый результат этих операций. Мы должны сразу запомнить, что функция всегда возвращает один результат, она не может вернуть таблицу или набор значений. Кроме того, следует понимать, что функции не являются одноразовыми, они могут быть использованы многократно в любом запросе и практически в любом контексте (при выборке - **SELECT**, фильтрации - **WHERE** или сортировке - **ORDER BY**).

- Те функции, которые мы уже знаем, представляют собой лишь малую часть от того общего количества функций, которые нам разрешает использовать MySQL. Этого совершенно недостаточно для того, чтобы эффективно работать с таблицами баз данных и считаться знатоком СУБД хотя бы среднего уровня. Поэтому в рамках этого урока мы подробно рассмотрим самые популярные функции для работы с текстовыми данными, обратим внимание на ряд очень полезных функций для осуществления математических операций, а также пожалуй самое главное - попрактикуемся в использовании функций, которые посвящены работе с датой и временем.

- MySQL не ограничивает пользователей только использованием встроенных функций. Также разрешается создавать свои функции, которые могут применяться схожим со встроенными функциями способом, т.е. принимать (или не принимать - как мы потом увидим, это не является обязательным условием при объявлении функции) параметры и возвращать результат обработки этих параметров. Мы не можем пройти мимо такого полезного инструментария - создание собственных функций также будет рассмотрено в рамках этого урока.

ИСПОЛЬЗОВАНИЕ ВСТРОЕННЫХ ФУНКЦИЙ

ФУНКЦИИ ДЛЯ РАБОТЫ С ТЕКСТОМ

- Перед тем, как мы приступим непосредственно к обзору текстовых функций, создадим и заполним данными таблицу students, которая будет использоваться для практики и примеров:

```
CREATE TABLE students (
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    name VARCHAR(255),
    admission_date DATE,
    scholarship DECIMAL(20, 6),
    age TINYINT UNSIGNED,
    avg_mark DECIMAL(20, 6)
) DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_0900_as_cs;

INSERT INTO students VALUES
(1, ' Britney Baker ', '2022-09-06', 1000, 21, 8.8),
(2, 'Fang Yu', '2022-08-05', 1200, 17, 9.5),
(3, 'Иван Иванов', '2022-07-18', 900, 25, 7.9),
(4, 'Dolores Rivera ', '2022-04-22', 800, 23, 8.4),
(5, ' adriano Caruso', '2022-10-25', 500, 20, 6.3),
(6, 'Katrin Roth', '2023-03-03', 0, 18, NULL);
```

id	name	admission_date	scholarship	age	avg_mark
1	Britney Baker	2022-09-06	1000	21	8.8
2	Fang Yu	2022-08-05	1200	17	9.5
3	Иван Иванов	2022-07-18	900	25	7.9
4	Dolores Rivera	2022-04-22	800	23	8.4
5	adriano Caruso	2022-10-25	500	20	6.3
6	Katrin Roth	2023-03-03	0	18	NULL

- Вначале скажем несколько слов об уже упоминавшейся ранее функции **CHAR_LENGTH**. Дело в том, что в Интернете и даже в профильной литературе[1] для получения длины строки рекомендуется использовать функцию **LENGTH**. При поверхностном рассмотрении нам может показаться, что это действительно

[1] Шилдс, У. SQL: быстрое погружение. Спб: Питер, 2022. С. 131.

так. Например, попробуем узнать количество символов в имени ученика Fang Yu (с id = 2):

```
SELECT name, LENGTH(name) FROM students WHERE id = 2;
```

Кажется, что проблем никаких нет - количество символов в этом имени действительно равняется семи:

name	LENGTH(name)
Fang Yu	7

А теперь попробуем применить функцию **LENGTH** для определения количества символов в имени ученика Иван Иванов (с id = 3):

```
SELECT name, LENGTH(name) FROM students WHERE id = 3;
```

С удивлением мы обнаружим, что нам пришло не совсем то, что мы хотели - мы получили число 21 (должно быть явно меньше):

name	LENGTH(name)
Иван Иванов	21

Почему же так произошло? Дело в том, что функция **LENGTH** возвращает **количество байт** в тексте, а функция **CHAR_LENGTH** возвращает **количество символов** в тексте. Распространенная неточность в Интернете и литературе связана с тем, что для примеров используются тексты исключительно с английскими буквами, в которых одна буква занимает один байт. В других языках в популярных кодировках одна буква может занимать большее количество байт - именно поэтому в имени Иван Иванов 21 байт. Восстановим справедливость и узнаем, сколько действительно символов в имени Иван Иванов при помощи функции **CHAR_LENGTH**:

```
SELECT name, CHAR_LENGTH(name) FROM students WHERE id = 3;
```

name	CHAR_LENGTH(name)
Иван Иванов	11

- Далее рассмотрим две полезные функции, которые позволяют преобразовать тексты в верхний (только большие буквы) и нижний (только маленькие буквы) регистр - **UCASE** и **LCASE** (также у них есть псевдонимы **UPPER** и **LOWER**). Сразу скажем, что у эти функции работают только с буквами - все остальные символы остаются без изменения. Кроме того, запомним, что у этих функций нет никаких подводных камней - они работают одинаково для всех языков и кодировок. Далее напишем запрос, который получает все имена студентов в обоих регистрах:

```
SELECT UCASE(name), LCASE(name) FROM students;
```

UCASE(name)	LCASE(name)
BRITNEY BAKER	britney baker
FANG YU	fang yu
ИВАН ИВАНОВ	иван иванов
DOLORES RIVERA	dolores rivera
ADRIANO CARUSO	adriano caruso
KATRIN ROTH	katrin roth

- С функциями изменения регистра связан один приятный момент. Вспомним, что при создании базы данных и таблицы мы можем указать сопоставление (**COLLATE**), которое отвечает за то, будет ли учитываться регистр при поиске, фильтрации и сортировке. Функции **LCASE** и **UCASE** позволяют обойти препятствия строгого сравнения - например, мы можем сравнивать значение в нижнем регистре со с значением столбца, которое было приведено в нижний регистр при помощи функции **LCASE** - в этом случае мы по сути получим сравнение без учета регистра:

```
-- ничего не будет выведено - не совпадает регистр
SELECT * FROM students WHERE name = 'katrin roth';

-- применена обработка LCASE - вернется строка со студентом
SELECT * FROM students WHERE LCASE(name) = 'katrin roth';
```

- Следующие три функции (**TRIM**, **LTRIM** и **RTRIM**) убирают пробелы из начала и конца строки (если таковые пробелы имеются).

TRIM убирает пробелы с обеих сторон, **LTRIM** - слева, а **RTRIM** - только справа. Если с какое-либо из целевых сторон будет стоять не один, а два или более пробелов - они будут удалены все. В нашей таблицы три студента имеют проблемы с пробелами - Britney Baker имеет пробелы с обеих сторон, Dolores River - справа, а adriano Caruso - слева. Применим на каждого из них все три функции и посмотрим результаты:

```
SELECT TRIM(name), LTRIM(name), RTRIM(name)
FROM students WHERE id IN (1, 4, 5);
```

Все прошло, как и ожидалось - **TRIM** убрала все, **LTRIM** - только то, что слева, а **RTRIM** - только то, что справа:

TRIM(name)	LTRIM(name)	RTRIM(name)
Britney Baker	Britney Baker	Britney Baker
Dolores Rivera	Dolores Rivera	Dolores Rivera
adriano Caruso	adriano Caruso	adriano Caruso

- Функции **LEFT** и **RIGHT** позволяют взять определенное количество символов слева и справа соответственно. Например, мы хотим получить только год (первые 4 символа) из столбца admission_date - этот столбец является датой, но при обработке функцией **LEFT** он будет восприниматься как текст. Также получим отдельно месяц и число при помощи **RIGHT** (последние 5 символов) из того же столбца admission_date:

```
SELECT id, LEFT(admission_date, 4), RIGHT(admission_date, 5)
FROM students;
```

id	LEFT(admission_date, 4)	RIGHT(admission_date, 5)
1	2022	09-06
2	2022	08-05
3	2022	07-18
4	2022	04-22
5	2022	10-25

6	2023	03-03

- Функция **SUBSTRING** похожа на функцию **LEFT**, но она вырезает символы не с начала строки, а начиная с той позиции, которую мы явно укажем (нумерация символов начинается с единицы). Можно не указывать количество символов, которые мы будем брать - тогда будут взяты все символы до конца строки. Для примера возьмем месяц поступления студентов (пропустим 5 символов, начнем с шестого и возьмем два символа):

```
SELECT
    id,
    SUBSTRING(admission_date, 6, 2)
FROM students;
```

id	SUBSTRING(admission_date, 6, 2)
1	09
2	08
3	07
4	04
5	10
6	03

- Также в MySQL есть функция **SUBSTRING_INDEX** - она вырезает не количество символов, а весь текст до или после разделителя. При применении этой функции необходимо указать сколько таких разделителей должно быть, чтобы мы начали вырезать текст. Если это число (количество разделителей) положительное, то мы будем получать текст до разделителя, если отрицательное - после разделителя. Применим эту функцию, что получить отдельно имена (текст до первого пробела) и фамилии студентов (текст после первого пробела). Важно - перед передачей имен в функцию **SUBSTRING_INDEX**, обработаем их функцией **TRIM**, чтобы убрать лишние пробелы справа и слева:

```
SELECT
   SUBSTRING_INDEX(TRIM(name), ' ', 1) AS firstname,
   SUBSTRING_INDEX(TRIM(name), ' ', -1) AS lastname
FROM students;
```

firstname	lastname
Britney	Baker
Fang	Yu
Иван	Иванов
Dolores	Rivera
adriano	Caruso
Katrin	Roth

- Очередная функция - **REPLACE** - делает именно то, о чем говорит ее название - при выборке меняет в тексте один набор символов на другой. Допустим, нам надо получить студента Fang Yu как Zong Yu (предположим, что это один из видов произношения). Применим функцию **REPLACE**, чтобы осуществить замену Fang на Zong:

```
SELECT
   name AS original,
   REPLACE(name, 'Fang', 'Zong') AS changed
FROM students WHERE id = 2;
```

original	changed
Fang Yu	Zong Yu

- Существуют две интересные функции, которые могут дополнить значение столбца до определенной длины какими-либо символами. Первая функция - **LPAD** дополняет слева, а вторая функция - **RPAD** дополняет справа. Надо четко осознавать, что если в функцию передать длину, которая меньше длины оригинального текста, то этот текст будет обрезан. Попробуем дополнить имя студентов звездочками справа и слева до длины в 20 символов:

```
SELECT
   LPAD(name, 20, '*') AS from_left,
   RPAD(name, 20, '*') AS from_right
FROM students;
```

from_left	from_right
***** Britney Baker	Britney Baker *****
**********Fang Yu	Fang Yu**********
*********Иван Иванов	Иван Иванов*********
*****Dolores Rivera	Dolores Rivera *****
***** adriano Caruso	adriano Caruso*****
*********Katrin Roth	Katrin Roth*********

- Назначение функции **REVERSE** очевидно - она позволяет получить текст в обратном порядке (иногда это может быть полезно при контактах с людьми из арабских стран, где текст пишется справа налево). Получим же имя каждого студента в обратном порядке:

```
SELECT name, REVERSE(name) FROM students;
```

name	REVERSE(name)
Britney Baker	rekaB yentirB
Fang Yu	uY gnaF
Иван Иванов	вонавИ навИ
Dolores Rivera	areviR seroloD
adriano Caruso	osuraC onairda
Katrin Roth	htoR nirtaK

МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

- К сожалению, для объяснения математических функций нам не подойдет таблица из предыдущей главы, поэтому создадим новую. Представим, что у нас база данных большой компании, которая занимается продажей строительных материалов. Вот таблицу с разными материалами, ценами, размерами (в сантиметрах) мы и создадим:

```
CREATE TABLE materials (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    width DECIMAL(20, 6),
    height DECIMAL(20, 6),
    length DECIMAL(20, 6),
    price DECIMAL(20, 6),
    quantity INTEGER
);

INSERT INTO materials VALUES
(1, 'Wood Beam', 15, 7, 300, 30.56, 300),
(2, 'Iron Rod', 3, 3, 600, 35.12, 500),
(3, 'Brick', 12, 6.5, 25, -5.68, 280),
(4, 'Block', 30, 25, 60, 40.79, 650),
(5, 'Plank', 10, 4, 225, 20.50, 180);
```

id	name	width	height	length	price	quantity
1	Wood Beam	15	7	300	30.56	300
2	Iron Rod	3	3	600	35.12	500
3	Brick	12	6.5	25	-5.68	280
4	Block	30	25	60	40.79	650
5	Plank	10	4	225	20.50	180

- Одни из самых полезных функций для работы с числами - функции округления до целого числа. Таких в MySQL можно найти целых 3. Во-первых, это стандартное математическое округление **ROUND** (если дробная часть 0.5 или больше - округление в большую сторону, если меньше - то в меньшую), во-вторых, **CEILING** - всегда округляет в большую сторону и, в-третьих, **FLOOR** - всегда округляет в меньшую сторону. Часто округление применяется для бухгалтерских операций - например, при переводе из одной валюты в другую, не имеющую

дробных частей. Попробуем применить все три функции в одном запросе, чтобы округлить цену каждого материала:

```
SELECT
    name, price, ROUND(price), CEILING(price), FLOOR(price)
FROM materials;
```

Результаты очевидны для всех строк, кроме той, которая имеет отрицательное значение. Мы можем заметить, что функция **ROUND** округляет по модулю (-5.68 округлено в “большее число” -6, хотя оно и меньше, чем -5). В свою очередь **CEILING** и **FLOOR** учитывают знак числа - для **CEILING** большим числом(чем -5.68) оказалось -5, а для **FLOOR** меньшим числом оказалось -6:

name	price	ROUND(price)	CEILING(price)	FLOOR(price)
Wood Beam	30.56	31	31	30
Iron Rod	35.12	35	36	35
Brick	-5.68	-6	-5	-6
Block	40.79	41	41	40
Plank	20.50	21	21	20

- Функция **ROUND** умеет округлять не только до целого числа, но и до определенного знака после запятой - для этого надо только добавить позицию этого знака. Для наглядности попробуем округлить цену материалов до первого знака:

```
SELECT name, price, ROUND(price, 1)
FROM materials;
```

name	price	ROUND(price, 1)
Wood Beam	30.56	30.6
Iron Rod	35.12	35.1
Brick	-5.68	-5.7
Block	40.79	40.8
Plank	20.50	20.5

- Мы могли заметить, что в таблице для обозначения цены кирпича используется отрицательное число. Предположим, что это ошибка человека, который заносил цену. Однако нам пришел заказ на 122 кирпича и мы должны посчитать общую стоимость. Если это будет заказ только на кирпичи, то ошибка будет замечена сразу, а если это будет в рамках большого заказа, где есть и другие позиции? Бухгалтерия будет в полном хаосе. Что же делать? Ответ - использовать функцию **ABS**, которая всегда возвращает положительную версию числа, какое бы число в нее не передали. Посчитаем правильную цену 122 кирпичей:

```
SELECT ABS(price) * 122 FROM materials WHERE name = 'Brick';
```

ABS(price) * 122
692.96

- Также в MySQL есть три замечательные функции, которые очень полезны для геометрических вычислений. Это **POW** - вычисление степени числа (также для нее существует псевдоним **POWER**), **SQRT** - взятие квадратного корня из какого либо числа, а также функция **PI**, которая не принимает никаких значений, но всегда возвращает приближенное значение числа PI. Как мы можем их применить? Например, нам пришел запрос посчитать площадь поперечного сечения железного прута, чтобы узнать, подходит ли такой товар клиенту. У нас есть ширина и длина прута, которые одинаковы и по сути являются диаметрами. Формула расчета площади поперечного сечения совпадает с формулой площади круга: **PI * (радиус)^2**. Вспомним, что радиус - половина диаметра и напишем запрос, в котором узнаем площадь поперечного сечения с помощью формулы:

```
SELECT PI() * POW(width / 2, 2) FROM materials
WHERE name = 'Iron Rod';
```

PI() * POW(width / 2, 2)
7.0685834705770345

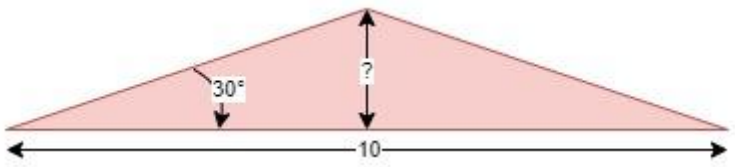
Чтобы продемонстрировать работу квадратного корня, представим, что человек купил доску и спрашивает, каким наиболее рациональным и гармоничным способом необходимо

прикрутить ее в нужное место, т.е. через сколько сантиметров надо вкручивать каждый шуруп. Для этого и можно использовать функцию SQRT, которая покажет на сколько частей можно разделить доску и какой надо делать отступ:

```
SELECT SQRT(length) FROM materials WHERE name = 'Plank';
```

SQRT(length)
15

- Говоря о математике в MySQL, нельзя не упомянуть возможность использования основных тригонометрических функций - **COS** (косинус), **SIN** (синус) и **TAN** (тангенс). Сразу заметим, что в качестве аргумента эти функции принимают не градусы, а радианы. Однако у нас есть удобная функция **RADIANS**, которая как раз преобразовывает градусы в радианы. Довольно сложно придумать пример задачи с такими функциями в контексте нашей таблицы, поэтому решим отвлеченную проблему. Представим, что нам надо рассчитать площадь части крыши (равнобедренный треугольник), если известно, что основание - 10 метров, а наклон крыши - 30°.



Решим эту задачу через тангенс, с помощью которого мы узнаем высоту, а потом, зная высоту и основание, получим площадь (площадь равняется длина основания умноженная на длину высоты и деленная пополам):

```
SELECT (10 / 2) * TAN(RADIANS(30)) * 10 / 2;
```

SELECT (10 / 2) * TAN(RADIANS(30)) * 10 / 2
14.433756729740644

- В качестве последней математической распространенной функции в MySQL рассмотрим функцию **RAND**. Она не принимает никаких аргументов, а в качестве ответа возвращает случайное дробное

число. Обычно используется для генерации тестовых данных или для определения победителя в какой-либо азартной игре. В качестве примера отобразим тестовую случайную цену для каждого материала в диапазоне от нуля до тысячи:

```
SELECT name, 1000 * RAND() FROM materials;
```

name	1000 * RAND()
Wood Beam	327.0156516945135
Iron Rod	967.2704599492909
Brick	855.3053753965585
Block	374.7155278685647
Plank	307.66154388679337

ФУНКЦИИ ДЛЯ РАБОТЫ С ДАТОЙ И ВРЕМЕНЕМ

- И в случае с временными функциями мы не можем использовать те таблицы, которые были созданы ранее - они не дадут нам показать все преимущества этих новых функций. Поэтому создадим таблицу - schedule, в которой будет помещено расписание движения поездов со временем отбытия и прибытия:

```
CREATE TABLE schedule (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    train VARCHAR(255),
    departed_at DATETIME,
    arrived_at DATETIME
);

INSERT INTO schedule VALUES
(1, 'Orient Express', '2023-02-01 17:30:00', '2023-02-03
13:45:00'),
(2, 'Blue Arrow', '2023-03-07 09:15:00', '2023-03-09
22:15:00'),
(3, 'Union Pacific', '2023-03-14 05:00:00', '2023-03-15
18:35:00'),
(4, 'Lone Star', '2023-04-02 02:25:00', '2023-04-04
23:55:00'),
(5, 'Golden Eagle', '2023-04-22 14:05:00', '2023-04-25
02:15:00');
```

id	train	departed_at	arrived_at
1	Orient Express	2023-02-01 17:30:00	2023-02-03 13:45:00
2	Blue Arrow	2023-03-07 09:15:00	2023-03-09 22:15:00
3	Union Pacific	2023-03-14 05:00:00	2023-03-15 18:35:00
4	Lone Star	2023-04-02 02:25:00	2023-04-04 23:55:00
5	Golden Eagle	2023-04-22 14:05:00	2023-04-25 02:15:00

- Однако начнем мы не с таблицы, а с функции **UNIX_TIMESTAMP.** Эта функция возвращает количество секунд с 1970-01-01 00:00:00 по настоящий момент во временной зоне UTC независимо от временной зоны MySQL или операционной системы. Это наиболее точное время из доступных нам в MySQL - на него можно ориентироваться при сравнениях с другими датами в базе данных. Если функции передать параметр в виде даты, например, '2008-08-07' или '2014-01-10 15:45', то она вернет количество секунд начиная с 1970 года и заканчивая переданной датой. Эта дата считается датой во временной зоне MySQL сервера. Попробуем реализовать все вышеперечисленное на практике, учитывая, что сейчас 17 февраля 2023 года:

```
SELECT
    UNIX_TIMESTAMP() AS simple,
    UNIX_TIMESTAMP('2008-08-07') AS from_date,
    UNIX_TIMESTAMP('2014-01-10 15:45') AS from_datetime;
```

simple	from_date	from_datetime
1676665183	1218056400	1389361500

- У функции **UNIX_TIMESTAMP** есть обратная функция **FROM_UNIXTIME,** которая позволяет из количества секунд получить дату во временной зоне MySQL сервера. Она поддерживает необязательный второй аргумент - строку, в которой можно передать формат ожидаемой даты:

```
SELECT
    FROM_UNIXTIME(1676665183) AS simple,
    FROM_UNIXTIME(1218056400, '%Y-%m-%d') AS dt,
    FROM_UNIXTIME(1389361500, '%Y-%m-%d %H:%i:%s') AS dtime;
```

simple	dt	dtime
2023-02-17 22:19:43	2008-08-07	2014-01-10 15:45:00

- Также в MySQL есть три удобные функции, которые позволяют получить отдельно время дня (час, минуту, секунду) - **CURRENT_TIME** (псевдоним **CURTIME**), дату (год, месяц и день) - **CURRENT_DATE** (псевдоним **CURDATE**), а также время и дату одновременно - **NOW** (год, месяц, день, а потом час, минуту, секунду) (псевдоним **CURRENT_TIMESTAMP**). Все эти инструкции предоставляют данные во временной зоне MySQL сервера, поэтому с ними надо быть особенно осторожными. Применим все упомянутые функции на практике:

```
SELECT CURRENT_TIME(), CURRENT_DATE(), NOW();
```

CURRENT_TIME()	CURRENT_DATE()	NOW()
22:19:43	2023-02-17	2023-02-17 22:19:43

- Теперь вернемся к нашей таблице с расписанием поездов. Во первых, попытаемся узнать, сколько дней каждый из поездов был в пути. Для этого можно использовать функцию **DATEDIFF**, которая принимает два параметра, каждый из которых может быть либо с типом **DATE**, либо с типом **DATETIME** (в schedule используется **DATETIME**) (часы и минуты просто отбрасываются при расчете, что может привести к неточностям):

```
SELECT train, DATEDIFF(arrived_at, departed_at)
FROM schedule;
```

train	DATEDIFF(arrived_at, departed_at)
Orient Express	2
Blue Arrow	2
Union Pacific	1
Lone Star	2
Golden Eagle	3

- Помимо разницы в часах, можно узнать разницу в часах, минутах и секундах. Для этого следует использовать функцию **TIMEDIFF**, которая имеет тот же синтаксис и такое же количество и такой же тип аргументов, как и **DATEDIFF**:

```
SELECT train, TIMEDIFF(arrived_at, departed_at)
FROM schedule;
```

train	TIMEDIFF(arrived_at, departed_at)
Orient Express	44:15:00
Blue Arrow	61:00:00
Union Pacific	37:35:00
Lone Star	69:30:00
Golden Eagle	60:10:00

- В MySQL есть еще одна функция - **TIMESTAMPDIFF**, которая позволяет узнать разницу между двумя датами в любых единицах (днях, часах, минутах, секундах и т.д.). Она принимает три параметра - временную единицу (**year**, **month**, **day**, **hour**, **minute**, **second**), затем первую дату, а в конце вторую. Обе даты могут быть с типом **DATE** или **DATETIME**. Попробуем узнать, сколько часов в пути был каждый поезд:

```
SELECT train, TIMESTAMPDIFF(minute, departed_at, arrived_at)
FROM schedule;
```

train	TIMESTAMPDIFF(minute,departed_at,arrived_at)
Orient Express	2655
Blue Arrow	3660
Union Pacific	2255
Lone Star	4170
Golden Eagle	3610

При желании можно узнать даже количество секунд, которые поезд провел в пути:

```
SELECT train, TIMESTAMPDIFF(second, departed_at, arrived_at)
FROM schedule;
```

train	TIMESTAMPDIFF(second,departed_at,arrived_at)
Orient Express	159300
Blue Arrow	219600
Union Pacific	135300
Lone Star	250200
Golden Eagle	216600

- Иногда нам надо узнать какое-либо время относительно другого времени (10 дней после 1 марта 2019 года или 10 секунд до 15 июня 2022). Для этого в MySQL есть функции **DATE_ADD** (прибавляет время) и **DATE_SUB** (вычитает время). У обеих функций одинаковый синтаксис - первым аргументом идет дата с типом **DATE** или **DATETIME**, а вторым ключевое слово **INTERVAL** и то количество временных единиц, которое мы хотим отнять или прибавить (например, **INTERVAL 10 hour**). Учитывая все вышесказанное, попробуем узнать те поезда, которые отправлялись в рейс между 1 марта и 31 марта 2023 года (**INTERVAL 1 month**):

```
SELECT train, departed_at FROM schedule
WHERE departed_at BETWEEN '2023-03-01' AND
DATE_ADD('2023-03-01', INTERVAL 1 month);
```

train	departed_at
Blue Arrow	2023-03-07 09:15:00
Union Pacific	2023-03-14 05:00:00

Также узнаем названия всех поездов, которые отправились в путь за 25 дней до 23 апреля 2023 года:

```
SELECT train, departed_at FROM schedule
WHERE departed_at BETWEEN DATE_SUB('2023-04-23', INTERVAL 25
day) AND '2023-04-23';
```

train	departed_at
Lone Star	2023-04-02 02:25:00
Golden Eagle	2023-04-22 14:05:00

- Вернемся к вопросу о временных зонах, которые в таблице имеют временной пояс +02:00. Что же делать, если нам нужно перевести их в другой временной пояс, например, +01:00? Для этого есть функция **CONVERT_TZ**, которая принимает три аргумента - дату в формате **DATETIME**, временную зону из которой будем конвертировать и временную зону, в которую будем конвертировать.

```
SELECT train, CONVERT_TZ(departed_at, '+02:00', '+01:00')
FROM schedule;
```

train	CONVERT_TZ(departed_at, '+02:00', '+01:00')
Orient Express	2023-02-01 16:30:00
Blue Arrow	2023-03-07 08:15:00
Union Pacific	2023-03-14 04:00:00
Lone Star	2023-04-02 01:25:00
Golden Eagle	2023-04-22 13:05:00

- В конце главы перечислим функции, которые могут получить из **DATETIME** только год, месяц, день, месяц, час, минуту и секунду. Это **YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE** и **SECOND** соответственно, которые принимают в качестве аргумента значение типа **DATETIME**. Получим же отдельно эти элементы у даты отправки поезда Orient Express:
-

```
SELECT YEAR(departed_at) y, MONTH(departed_at) mon,
DAY(departed_at) d, HOUR(departed_at) h, MINUTE(departed_at)
min, SECOND(departed_at) s FROM schedule WHERE train =
'Orient Express';
```

y	mon	d	h	min	s
2023	2	1	17	30	0

СОЗДАНИЕ ФУНКЦИЙ

- Как уже было сказано во введении, MySQL не только позволяет использовать встроенные функции, но и создавать свои. Можно, конечно, спросить - а зачем вообще это делать? Во-первых, бывают случаи, когда нам не хватает встроенных функций, чтобы решить какую-то проблему - например, в MySQL нет функции, которая позволяет узнать, сколько надо сделать операций (удаление, вставка замена) над одним текстом, чтобы он превратился в другой (т.н. расстояние Левенштейна). Во-вторых, очевидно, что если мы встречаемся с одной и той же проблемой, то не следует копировать ее решение из одного места в другое - лучше оформить это решение в виде функции. В-третьих, функции хранятся и исполняются на сервере, т.е. нам не надо передавать логику решения вместе с запросом, что уменьшает количество передаваемых данных на такой сервер и немного ускоряет запрос.

- Надо признать, что для новичков (и не только для новичков) синтаксис создания функции может показаться довольно запутанным (кроме основной команды **CREATE FUNCTION**). Ниже идет переведенная на русский язык копия схемы синтаксиса из официальной документации MySQL:[2]

```
CREATE
    [DEFINER = user]
    FUNCTION [IF NOT EXISTS] имя_функции ([аргумент_функции [,...]])
    RETURNS тип
    [характеристики ...] тело_функции

аргумент_функции:
    название тип

тип:
    любой тип MySQL

характеристики: {
    COMMENT 'string'
    LANGUAGE SQL
    [NOT] DETERMINISTIC
    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    SQL SECURITY { DEFINER | INVOKER }
}

тело_функции:
    Инструкции SQL без ошибок
```

[2] https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html

Однако не стоит пугаться - большинство из этих конструкций не являются обязательными, более того, некоторые не оказывают вообще никакого эффекта на функцию на данный момент (они предназначены для неких будущих планов). Но все же попытаемся подробно разобрать каждый элемент, который представлен в квадратных скобках, т.е. который не является обязательным (это может не пригодится для работы, но это могут спросить на собеседовании):

- **DEFINER** - здесь можно указать логин создателя функции, если не указывать, то будет вставлен логин того, кто действительно создал функцию (в нашем случае - 'root'@'localhost'). Может быть полезно для ограничения прав доступа к функции (см. **SQL SECURITY**).

- **АРГУМЕНТ ФУНКЦИИ** - в функцию можно передавать некоторые значения, которые помогают рассчитать финальный результат. Например, чтобы узнать сумму двух чисел, нам надо передать эти два числа в функцию. Аргумент определяется так - сначала пишется его название, а потом тип. Если аргументов несколько, их можно перечислить через запятую: first_num **INT**, second_num **INT**.

- **COMMENT -** первая из характеристик, которая просто позволяет задать комментарий для функции (пояснить, что функция делает).

- **LANGUAGE SQL** - показывает тот язык, на котором написана функция. На данный момент эта характеристика бессмысленна, т.к. MySQL поддерживает только SQL.

- **DETERMINISTIC** или **NOT DETERMINISTIC** - показывает, будет ли возвращенный результат один и тот же, если мы передаем (или не передаем) в функцию одни и те же аргументы. Например, функция **PI** вернет всегда одинаковый результат (3.14...), а функция **NOW** - всегда разный. В некоторых режимах MySQL этот параметр является обязательным, поэтому лучше его всегда прописывать явно. Используется MySQL для построения оптимальных запросов.

- **CONTAINS SQL** - указывает, что в функции нет выборки (**SELECT**) или вставки (**INSERT**) в таблицу - только SQL выражения - определение переменных (определение рассмотрим позднее), снятие блокировок и т.д.

- ➢ **NO SQL** – внутри нет SQL выражений (определение переменных и т.д.).

- ➢ **READS SQL DATA** – внутри есть выборка из какой-либо таблицы.

- ➢ **MODIFIES SQL DATA** – внутри есть вставка, обновление или удаление из таблицы.

- ➢ **SQL SECURITY DEFINER** или **SQL SECURITY INVOKER** – определяет, для которого будут проверяться права на команды внутри функции – для создателя функции (**DEFINER**) или для того, кто вызывает функцию (**INVOKER**).

- Напомним, что функция может возвращать только одно значение – тип этого значения определяется явно после ключевого слова **RETURNS**.

- Само тело функции (определение переменных, запросы в таблицы, возвращение результата) находится между ключевыми словами **BEGIN** и **END**. Возвращаемое значение указывается в самом конце тела функции при помощи оператора **RETURN**.

- Как и все команды в MySQL, команды внутри функции должны заканчиваться специальным символом – разделителем, который по умолчанию равняется точке с запятой (;). Однако если мы поставим этот разделитель внутри функции, то при создании этой функции произойдет ошибка, т.к. интерпретатор "подумает", что мы хотим завершить функцию, хотя еще не произошло возвращение значения (**RETURN**) и закрытие функции (**END**). Что же делать? "Обмануть" интерпретатор – до создания функции изменить глобально этот оператор на какой-либо другой символ (например, **|**) при помощи команды DELIMITER, в самой функции продолжить использовать стандартный разделитель, а после создания – поменять обратно на привычную для всех точку с запятой.

```
-- смена разделителя
DELIMITER |

-- создаем функцию

-- возвращение стандартного разделителя
DELIMITER ;
```

- Пришло время написать нашу первую функцию (попытаемся использовать минимальное количество характеристик). Предположим, что мы хотим получить имя самого быстрого поезда из таблицы schedule:

```
-- смена разделителя
DELIMITER |

-- начало функции
CREATE FUNCTION IF NOT EXISTS get_fastest_train()
RETURNS VARCHAR(255) -- указание типа возвращаемого значения
DETERMINISTIC -- возвращаемое значение не меняется

BEGIN -- начало тела функции

    -- определение переменной, которую мы будет возвращать
    DECLARE fastest_train VARCHAR(255) DEFAULT NULL;

    -- запрос записывает свой результат в переменную
    SELECT train INTO fastest_train
    FROM schedule
    ORDER BY TIMESTAMPDIFF(minute,departed_at,arrived_at) ASC
    LIMIT 1;

    RETURN(fastest_train); -- возвращение переменной

END | -- конец тела функции и самой функции

-- возвращение стандартного разделителя
DELIMITER ;
```

  Очевидно, что объявление функции требует дополнительных разъяснений. Во-первых, мы видим, что нам надо объявить переменную при помощи ключевого слова **DECLARE** - в эту переменную будет записан результат - название самого быстрого поезда. Во-вторых, во время выборки мы применили до сих пор не применявшуюся конструкцию **SELECT ... INTO**, которая и осуществила запись в переменную.

- На самом деле мы не обязаны записывать выборку в переменную, а только потом ее возвращать. Это хорошая практика, т.к. при таком подходе мы можем видоизменить переменную и использовать ее для каких-то дальнейших запросов внутри этой же функции. Но если мы уверены, что у нас нет дополнительных обработок и запросов, то мы можем пропустить создание переменной, и вставить запрос прямо в команду **RETURN(...)**:

```
DELIMITER |

CREATE FUNCTION IF NOT EXISTS get_fastest_train_without_var()
RETURNS VARCHAR(255)
DETERMINISTIC

BEGIN
    -- возвращение значения без переменной
    RETURN(SELECT train
    FROM schedule
    ORDER BY TIMESTAMPDIFF(minute,departed_at,arrived_at) ASC
    LIMIT 1);
END |

DELIMITER ;
```

- Попробуем получить значения из обеих функций (значения будут одинаковые, т.к. мы пробовали разный синтаксис, но придерживались одинаковой логики):

```
SELECT get_fastest_train(), get_fastest_train_without_var();
```

get_fastest_train()	get_fastest_train_without_var()
Union Pacific	Union Pacific

- Может случиться, что мы помним название функции, но совершенно не помним того, что она делает. Для этого существует специальная конструкция **SHOW CREATE FUNCTION название_функции**, которая позволяет увидеть подробную информацию о функции, включая ее тело. Результат, возвращаемый данной командой занимает много места по горизонтали, поэтому попытаемся развернуть его по вертикали, применив разделитель **\G**:

```
SHOW CREATE FUNCTION get_fastest_train_without_var\G
```

```
*************************** 1. row ***************************
            Function: get_fastest_train_without_var
            sql_mode:
ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZER
O_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION
     Create Function: CREATE DEFINER=`root`@`localhost`
FUNCTION `get_fastest_train_without_var`() RETURNS
```

```
varchar(255) CHARSET utf8mb4
    DETERMINISTIC
BEGIN

    RETURN(SELECT train
    FROM schedule
    ORDER BY TIMESTAMPDIFF(minute,departed_at,arrived_at) ASC
    LIMIT 1);
END
character_set_client: utf8mb4
collation_connection: utf8mb4_0900_ai_ci
  Database Collation: utf8mb4_0900_ai_ci
```

- Далее рассмотрим удаление функций. Тут нет ничего необычного - применяем команду **DROP FUNCTION**, которая может быть дополнена конструкцией **IF EXISTS**, которая обезопасит нас от ошибки в том случае, если функция не будет существовать:

```
-- удаление существующей функции без проверки
DROP FUNCTION get_fastest_train;

-- удаление существующей функции с проверкой проверки
DROP FUNCTION IF EXISTS get_fastest_train_without_var;

-- удаление несуществующей функции с проверкой
DROP FUNCTION IF EXISTS undefined_function;
```

- Мы рассмотрели создание функции без аргументов, а теперь попробуем создать простейшую функцию multiply, которая умножает два любые числа, которые переданы этой функции в качестве аргументов:

```
DELIMITER |

CREATE FUNCTION IF NOT EXISTS multiply(
    first_num DECIMAL(53, 10),
    second_num DECIMAL(53, 10)
)
RETURNS DECIMAL(53, 10)
DETERMINISTIC
BEGIN
    RETURN(first_num * second_num);
END |

DELIMITER ;
```

Попробуем применить новую функцию на практике, перемножив два числа - 17 и 22:

```
SELECT MULTIPLY(17, 22);
```

Заметим, что мы передали в функцию целые числа, хотя в определении функции требовались дробные числа. Ошибки не произошло, более того, результат был получен абсолютно правильно. Дело в том, что в MySQL целое число может быть представлено в виде дробного числа адекватным способом. Если бы потребовалось произвести обратную операцию - преобразовать дробное число в целов, ошибки бы не было - MySQL применила бы операцию округления к аргументам и мы бы получили не совсем точный результат:

MULTIPLY(17, 22)
374.0000000000

- Мы должны помнить, что встроенные функции могут быть использованы не только сами по себе и в других запросах (например, в **SELECT**). То же самое относится к тем функция, которые мы создали сами. Чтобы это продемонстрировать это, применим нашу новую функцию multiply, чтобы узнать общую стоимость каждого материала в нашей таблице materials (чтобы это узнать в общем случае, надо умножить цену за одну единицу на общее количество - это мы и сделаем). Учтем, что у нас имеются отрицательные цены и обернем результат в функцию ABS:

```
SELECT name, ABS(MULTIPLY(price, quantity)) AS total
FROM materials;
```

name	total
Wood Beam	9168.0000000000
Iron Rod	17560.0000000000
Brick	1590.4000000000
Block	26513.5000000000
Plank	3690.0000000000

- Теперь перейдем к тому, как использовать переданный в функцию аргумент в запросе, который находится внутри тела функции. Например, мы знаем идентификатор студента (из таблицы students) и по этому идентификатору хотим узнать его имя и среднюю оценку. Мы должны помнить, что функция может вернуть только одно значение, поэтому возвращать имя и оценку будем в виде объединенной строки:

```
DELIMITER |

CREATE FUNCTION IF NOT EXISTS get_student_info(
    student_id INT
)
RETURNS VARCHAR(255)
DETERMINISTIC
BEGIN
    DECLARE student_name VARCHAR(255) DEFAULT NULL;
    DECLARE student_avg_mark DECIMAL(20, 6) DEFAULT NULL;

    SELECT name, avg_mark INTO student_name, student_avg_mark
    FROM students
    WHERE id = student_id;

    RETURN(CONCAT(student_name, ' ', student_avg_mark));
END |

DELIMITER ;
```

Попробуем получить информацию о студенте с id = 2:

```
SELECT get_student_info(2);
```

get_student_info(2)
Fang Yu 9.500000

Если нам интересно, что произойдет в том случае, если мы передадим идентификатор студента, то мы должны вспомнить, как ведет себя MySQL в том случае, когда надо отобразить отсутствие значения. Можно догадаться, что будет возвращено специальное значение NULL:

```
SELECT get_student_info(99);
```

get_student_info(99)
NULL

- Функции можно использовать не только для получения данных, но и для их вставки. Напишем функцию, которая будет вставлять данные о новом поезде в таблицу schedule и возвращать идентификатор нового поезда:

```
DELIMITER |

CREATE FUNCTION IF NOT EXISTS create_new_train(
    train_name VARCHAR(255),
    train_departed_at DATETIME,
    train_arrived_at DATETIME
)
RETURNS INT
DETERMINISTIC
BEGIN
    INSERT INTO schedule(
        train, departed_at, arrived_at
    ) VALUES (
        train_name, train_departed_at, train_arrived_at
    );

    RETURN(SELECT LAST_INSERT_ID());
END |

DELIMITER ;
```

Попробуем вставить новый поезд:

```
SELECT create_new_train(
    'Roaring Bear',
    '2023-05-01 01:10:00',
    '2023-05-01 23:55:00'
);
```

create_new_train('Roaring Bear', '2023-05-01 01:10:00', '2023-05-01 23:55:00')
6

- Внутри функции можно помещать не только математические операции или работу с таблицами, но и другие конструкции, например, условные конструкции и циклы. Циклы мы рассмотрим в разделе о процедурах (как мы увидим, они почти идентичны с функциями), а сейчас обратим наше внимание на условные конструкции. Напишем функцию, которая принимает в качестве аргумента среднюю оценку студента и определяет, насколько он хорош в учебе:

```
DELIMITER |

CREATE FUNCTION get_student_level(
    student_avg_mark DECIMAL(20, 6)
)
RETURNS VARCHAR(255)
DETERMINISTIC
BEGIN
    DECLARE result VARCHAR(255);

    IF student_avg_mark > 9.0 THEN
        SET result = 'Excellent';
    ELSEIF student_avg_mark > 8.0 THEN
        SET result = 'Good';
    ELSEIF student_avg_mark > 7.0 THEN
        SET result = 'Mediocre';
    ELSE
        SET result = 'Bad';
    END IF;

    RETURN(result);
END |

DELIMITER ;
```

Теперь узнаем уровень каждого студента:

```
SELECT name, get_student_level(avg_mark) AS level
FROM students;
```

name	level
Britney Baker	Good
Fang Yu	Excellent
Иван Иванов	Mediocre

Dolores Rivera	Good
adriano Caruso	Bad
Katrin Roth	Bad