**Оглавление**

[1. Извлечение и сортировка данных 2](#_Toc187329156)

[2. Фильтрация данных 2](#_Toc187329157)

[3. Создание вычисляемых полей 3](#_Toc187329158)

[4. Текстовые функции 4](#_Toc187329159)

[5. Числовые функции 6](#_Toc187329160)

[6. Функции даты и времени 8](#_Toc187329161)

[7. Дополнительные функции 13](#_Toc187329162)

[8. Условные конструкции 14](#_Toc187329163)

[9. Агрегатные функции и группировка 15](#_Toc187329164)

[10. Операторы JOIN, LEFT JOIN, RIGHT JOIN, CROSS JOIN, UNION 18](#_Toc187329165)

[11. Обновление, удаление, добавление данных 21](#_Toc187329166)

[12. Создание таблиц 24](#_Toc187329167)

[13. Удаление, переименование и обновление таблиц 28](#_Toc187329168)

[14. Индексы 30](#_Toc187329169)

[15. Тригеры 32](#_Toc187329170)

[16. Представления 33](#_Toc187329171)

[17. Обобщенные табличные выражения 34](#_Toc187329172)

[18. Пользовательские функции, процедуры и переменные 37](#_Toc187329173)

[19. Хранимые процедуры 43](#_Toc187329174)

[20. Оконные функции 46](#_Toc187329175)

[21. Регулярные выражения 53](#_Toc187329176)

1. Извлечение и сортировка данных

LIMIT – ограничение записей

OFFSET – индекс начала извлечения

SELECT trackname

FROM Songs

ORDER BY trackname DESC 3

LIMIT 3 OFFSET 2;

Выражение LIMIT 3 OFFSET 2 говорит о том, что должно быть извлечено три записи, начиная с записи с индексом два.

Если псевдоним состоит из нескольких слов, он должен быть представлен строкой, заключенной в апострофы

Ключевое слово DESC — это сокращение от DESCENDING (убывание)ASC – сокращение от ASCENDING (возрастание)

1. Фильтрация данных

|  |  |
| --- | --- |
| **Оператор** | **Проверка** |
| = | Равенство |
| <=> | Эквивалентность |
| != или <> | Неравенство |
| < | Меньше |
| <= | Меньше или равно |
| > | Больше |
| >= | Больше или равно |
| BETWEEN | Вхождение в диапазон |
| IS NULL | Значение NULL |
| IS NOT NULL | Не значение NULL |

Псевдонимы полей не могут быть использованы в блоке оператора WHERE для фильтрации записей.

Логические операторы AND, OR, NOT, IN и NOT IN имеют разный приоритет. В таблице ниже они представлены в порядке уменьшения их приоритета:

|  |
| --- |
| **Оператор** |
| IN , NOT IN |
| NOT |
| AND |
| OR |

Знак процента (%), в шаблоне поиска соответствует последовательности любых символов, число символов в последовательности может быть от 0 и более.

SELECT trackname, artist

FROM Songs

WHERE trackname LIKE 'You %' OR

trackname LIKE '% You %' OR

trackname LIKE '% You';

Знак нижнего подчеркивания (\_), в шаблоне поиска соответствует одному любому символу. С его помощью можно составить запрос, который поможет отыскать информацию, названия которой начинаеся со слова из заданного количества букв.

SELECT trackname, artist

FROM Songs

WHERE trackname LIKE '\_\_\_\_ %';

В случаях, когда регистр символов важен, используется оператор LIKE BINARY, который работает аналогично оператору LIKE, но учитывает регистр символов, используемых в шаблоне поиска.

SELECT trackname, artist

FROM Songs

WHERE trackname LIKE BINARY '%you%';

1. Создание вычисляемых полей

Для объединения нескольких значений используется функция CONCAT(). Она принимает переменное количество аргументов, выполняет их конкатенацию и возвращает полученный результат. Результатом функции CONCAT() всегда является строка, аргументы функции могут принадлежать любым типам.

​SELECT CONCAT(artist, ' - ', trackname) as song

FROM Songs

ORDER BY song DESC;

Функция — CONCAT\_WS() не только объединяет значения, но и добавляет между ними разделитель. Разделитель указывается в качестве первого аргумента, объединяемые значения — в качестве всех остальных.

SELECT CONCAT\_WS(', ', id, artist, trackname) AS song

FROM Songs;

В SQL поддерживаются основные математические операторы:

+ (сложение)

- (вычитание)

\* (умножение)

/ (деление)

1. Текстовые функции

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Аргументы** |
| CHAR\_LENGTH() | используется для вычисления длины строки | - |
| LOWER() | используется для перевода строки в нижний регистр | - |
| UPPER() | используется для перевода строки в верхний регистр | - |
| LTRIM() | используется для отсечения пробелов из левой части строки | - |
| RTRIM() | используется для отсечения пробелов из правой части строки | - |
| REVERSE() | используется для переворота строки | - |
| REPEAT() | используется для повторения строки | str — исходная строка  count — количество повторений |
| LPAD() | используется для увеличения длины строки слева до определенного значения путем добавления в ее начало другой строки | str — исходная строка  len — желаемая длина строки  padstr — дополняющая строка |
| RPAD() | используется для увеличения длины строки справа до определенного значения путем добавления в ее начало другой строки | str — исходная строка  len — желаемая длина строки  padstr — дополняющая строка |
| LEFT() | используется для извлечения определенного количества символов из начала строки | str — исходная строка  count — количество извлекаемых символов |
| RIGHT() | используется для извлечения определенного количества символов из конца строки | str — исходная строка  count — количество извлекаемых символов |
| LOCATE() | используется для определения местоположения подстроки в строке | substr — искомая подстрока  str — исходная строка  start — позиция начала поиска |
| REPLACE() | используется для замены подстроки в строке | str — исходная строка  from\_str — заменяемая подстрока  to\_str — заменяющая подстрока |
| SUBSTRING() | используется для извлечения подстроки из строки | str — исходная строка  start — позиция первого извлекаемого элемента  len — длина извлекаемой подстроки |
| SUBSTRING\_INDEX() | используется для извлечения подстроки из строки с помощью разделителя | str — исходная строка  delimiter — разделител  count — количество появлений разделителя |
| TRIM( | используется для удаления всех вхождений подстроки из начала и/или конца строки | TRIM("аргумент" что удаляем FROM откуда удаляем)  аргументы:  LEADING — из начала строки  TRAILING  — из конца строки  BOTH — из начала и конца строки |
| ORD() | используется для получения числового кода символа в ASCII-таблице | - |
| CHAR() | используется для получения символа в ASCII-таблице из числового кода | - |

1. Числовые функции

Оператор DIV используется для целочисленного деления

Оператор MOD используется для вычисления остатка от деления

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Аргументы** |
| ABS() | используется для вычисления модуля числа | число |
| ROUND() | используется для округления числа | num — число  decimals — количество знаков после запятой |
| POW() | используется для возведения числа в определенную степень | num — число  degree — показатель степени |
| SQRT() | используется для  вычисления квадратного  корня | число |
| RAND() | используется для  генерации случайных чисел | При вызове без аргументов она возвращает случайное число с плавающей точкой в диапазоне [0, 1) |
| FLOOR() | используется для округления числа в меньшую сторону | число |
| CEILING() | используется для округления числа в большую сторону | число |
| DEGREES() | используется для перевода значения из радиан в градусы | число |
| RADIANS() | используется для перевода значения из градусов в градианы | число |
| PI() | используется для получения числа пи (π) |  |
| SIN()  COS()  TAN() | используются для вычисления синуса, косинуса и тангенса | число |
| LEAST() | используется для поиска минимального значения | принимает переменное количество аргументов |
| GREATEST() | используется для поиска максимального значения | принимает переменное количество аргументов |
| CONV() | используется для перевода числа из одной системы счисления в другую | num — число  from\_base — система счисления, в которой представлено число  to\_base — система счисления, в которую нужно перевести число |
| FORMAT() | используется для округления и форматирования числа | num — число  decimals — количество знаков после запятой |

Для вычисления рандомных значений из таблицы можно использовать алгоритм

SELECT FLOOR(ROUND(RAND(), 2)\*(столбец\_b - столбец\_a) + столбец\_a) as random\_value

FROM Table

1. Функции даты и времени

Временной интервал

Интервал INTERVAL - представляет собой некоторый промежуток времени, выраженный в тех или иных единицах измерения

SELECT '2023-01-01 14:00:00' - INTERVAL '01:30' HOUR\_MINUTE;

Поддерживаемые единицы измерения

|  |  |
| --- | --- |
| **Единица измерения** | **Ключевое слово** |
| микросекунда | MICROSECOND |
| секунда | SECOND |
| минута | MINUTE |
| час | HOUR |
| день | DAY |
| неделя | WEEK |
| месяц | MONTH |
| квартал | QUARTER |
| год | YEAR |

Поддерживаемые составные единицы измерения

|  |  |  |
| --- | --- | --- |
| **Единица измерения** | **Ключевое слово** | **Формат величины** |
| секунда и микросекунда | SECOND\_MICROSECOND | секунды.микросекунды |
| минута, секунда и микросекунда | MINUTE\_MICROSECOND | минуты:секунды.микросекунды |
| минута и секунда | MINUTE\_SECOND | минуты:секунды |
| час, минута, секунда и микросекунда | HOUR\_MICROSECOND | часы:минуты:секунды.микросекунды |
| час, минута и секунда | HOUR\_SECOND | часы:минуты:секунды |
| час и минута | HOUR\_MINUTE | часы:минуты |
| день, час, минута, секунда и микросекунда | DAY\_MICROSECOND | дни часы:минуты:секунды.микросекунды |
| день, час, минута и секунда | DAY\_SECOND | дни часы:минуты:секунды |
| день, час и минута | DAY\_MINUTE | дни часы:минуты |
| день и час | DAY\_HOUR | дни часы |
| год и месяц | YEAR\_MONTH | годы-месяцы |

Функции

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Аргументы** |
| NOW() | используется для получения текущей даты и времени | - |
| CURDATE() | используется для получения текущей даты | - |
| CURTIME() | используется для получения текущего времени | - |
| UTC\_TIMESTAMP() | используется для получения текущей даты и времени в UTC (нулевой часовой пояс) | - |
| UTC\_DATE() | используется для получения текущей даты в UTC (нулевой часовой пояс) | - |
| UTC\_TIME() | используется для получения текущего времени в UTC (нулевой часовой пояс) | - |
| MAKEDATE() | используется для создания даты | year — год  day — номер дня в году |
| MAKETIME() | используется для создания временного значения | hours — часы  minutes — минуты  seconds — секунды |
| TIME\_TO\_SEC() | используется для преобразования временного значения в секунды | время |
| TO\_DAYS() | используется для преобразования даты в количество дней | дата |
| TO\_SECONDS() | используется для преобразования даты в количество секунд | дата |
| LAST\_DAY() | используется для замены дня на последний день месяца | дата |
| SEC\_TO\_TIME() | используется для выполняет обратного преобразования — количество секунд во временное значение | число секунд |
| DATE() | используется для получения даты из даты и времени | дата и время |
| TIME() | используется для получения времени из даты и времени | дата и время |
| HOUR()  MINUTE()  SECOND()  MICROSECOND() | используется для определения количества часов, минут, секунд и микросекунд во времени | время |
| YEAR()  MONTH()  DAY() | используется для извлечения значения года, месяца, дня из даты | дата |
| DAYOFYEAR() | используется для вычисления номера дня в году (от 1 до 366) на основе даты | дата |
| WEEK() | используется для вычисления номера недели в году | дата |
| WEEKDAY() | используется для вычисления номера дня недели (от 0 до 6) | дата |
| QUARTER() | используется для вычисления номера квартала (от 1 до 4) | дата |
| DAYNAME() | используется для вычисления названия дня недели | дата |
| MONTHNAME() | используется для определения названия месяца | дата |
| ADDDATE()/SUBDATE() | используется для прибавления/вычитания временного интервала к дате и времени | datetime — дата и время  interval — временной интервал |
| ADDTIME()/SUBTIME() | используется для прибавления временного значения к дате и времени. | datetime — дата и время  time — временное значение |
| DATEDIFF() | используется для вычисления разницы в днях между двумя значениями даты и времени | datetime1 — первая дата и время  datetime2 — вторая дата и время |
| TIMEDIFF() | используется для вычисления разницы во времени между двумя значениями даты и времени | datetime1 — первая дата и время  datetime2 — вторая дата и время |
| TIMESTAMPDIFF() | используется для вычисления разницы между двумя значениями даты и времени в определенных единицах измерения | unit — единица измерения, в которой требуется получить разницу  datetime1 — первая дата и время  datetime2 — вторая дата и время |
| TIMESTAMP() | используется для объединения даты и временного значения | date — дата  time — временное значение |
| DATE\_FORMAT() | используется для форматирования даты и времени | datetime — дата и время  format — строка формата |
| TIME\_FORMAT() | используется для форматирования времени | time — временное значение  format — строка формата |
| STR\_TO\_DATE() | используется для извлечения даты и времени из строки (операция обратная форматированию) | string — строка с компонентами даты и времени  format — строка формата |
| GET\_FORMAT() | используется для получения определенной строки формата для даты, времени или даты и времени | type — тип объекта, для которого требуется строка формата (DATE, TIME или DATETIME)  format — непосредственно формат (USA, JIS, ISO или EUR) |

*Спецификаторы формата*

|  |  |
| --- | --- |
| **Спецификатор** | **Описание** |
| %M | Полное название месяца (January..December) |
| %b | Сокращенное название месяца (Jan..Dec) |
| %W | Полное название дня недели (Sunday..Saturday) |
| %a | Сокращенное название дня недели (Sun..Sat) |
| %Y | Год (0000..9999) |
| %m | Номер месяца (00..12) |
| %d | День месяца (00..31) |
| %w | Номер дня недели (0..6) |
| %H | Часы (00..23) |
| %i | Минуты (00..59) |
| %S | Секунды (00..59) |
| %f | Микросекунды (000000..999999) |
| %T | Время в формате HH:MM:SS |

Функции, предназначенные в большей степени для работы только с датой или только с временем, могут использоваться для работы с датой и временем.

В MySQL временные значения ограничены диапазоном [-838:59:59; 838:59:59].

В MySQL имеется специальная переменная lc\_time\_names, которая управляет локализацией. Она определяет, как будут отображаться названия дней недели и месяцев при использовании таких функций, как DAYNAME() и MONTHNAME().

Переменная lc\_time\_names может принимать различные значения в зависимости от того, какая локализация необходима: например, 'ru\_RU' — русская, 'en\_US' — английская. Изменение ее значения выполняется с помощью следующей конструкции:

SET lc\_time\_names = <языковая локаль>;

1. Дополнительные функции

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Аргументы** |
| ISNULL() | используется для проверки значения на NULL | произвольное значение |
| IF() | спользуется для выбора одного из двух значений в зависимости от результата некоторого условного выражения | condition — условное выражение  value1 — произвольное значение  value2 — произвольное значение |
| COALESCE() | используется для поиска первого непустого значения | переменное количество аргументов и возвращает первый из них, не равный NULL |
| IFNULL() | используется для замены некоторого значения на альтернативное, если исходное значение равняется NULL | value — исходное значение  alternative\_value — альтернативное значение |
| NULLIF() | используется для сравнения двух значений | value1 — произвольное значение  value2 — произвольное значение |
| CONVERT() | используется для преобразования значения в необходимый тип данных | value — значение, которое необходимо преобразовать  type — желаемый тип |

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| DATE | значение типа DATE |
| DATETIME | значение типа DATETIME |
| TIME | значение типа TIME |
| DECIMAL(M, D) | значение типа DECIMAL |
| CHAR(N) | значение типа CHAR |
| SIGNED | значение типа BIGINT с учетом знака |
| UNSIGNED | значение типа BIGINT без учета знака |
| YEAR | значение типа YEAR |

Одним из примеров ее использования функции NULLIF() является реализация безопасного деления путем замены делителя на значение NULL, если тот равен нулю:

<делимое> / NULLIF(<делитель>, 0)

1. **Условные конструкции**

*Простая форма*

Простая форма подразумевает, что после оператора CASE указывается некоторое значение, которое последовательно сравнивается на равенство с другими значениями. Каждое сравниваемое значение указывается после ключевого слова WHEN, а результат, который будет возвращен, если исходное значение и сравниваемое окажутся равными, — после ключевого слова THEN. Если исходное значение не совпадет ни с одним из сравниваемых значений, результатом будет значение по умолчанию, которое указывается после ключевого слова ELSE.

CASE <значение>

WHEN <первое сравниваемое значение> THEN <результат>

WHEN <второе сравниваемое значение> THEN <результат>

ELSE <значение по умолчанию>

END AS <псевдоним>

*Усложненная форма*

Ключевая разница усложненной формы от простой заключается в том, что она не требует какого-либо значения вначале, которое будет сравниваться с другими. По сути, усложненная форма представляет собой набор произвольных условий, которые последовательно проверяются на истинность. Если проверяемое условие истинно, то результатом будет соответствующее ему значение, если ложно — проверка перейдет к следующему условию. Если все условия окажутся ложными, результатом будет значение по умолчанию.

CASE

WHEN <первое условие> THEN <результат>

WHEN <второе условие> THEN <результат>

ELSE <значение по умолчанию>

END AS <псевдоним>

*Оператор CASE как функция*

Оператор CASE может рассматриваться как функция, поскольку результатом его выполнения, независимо от формы, всегда является определенное значение. Также можно помещать в различные функции для последующего преобразования результата.

SELECT <первое поле>, <второе поле>,

CASE

WHEN <первое условие> THEN <результат>

WHEN <второе условие> THEN <результат>

ELSE <значение по умолчанию>

END AS <псевдоним третьего поля>

FROM <имя таблицы>;

Также оператор CASE можно использовать в блоке оператора ORDER BY для определения порядка сортировки по определенным критериям.

1. Агрегатные функции и группировка

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Аргументы** |
| COUNT() | используется для подсчета количества записей в таблице или количества значений в поле | Для определения количества записей в таблице в функцию COUNT() в качестве аргумента необходимо передать звездочку (\*)  Для определения количества записей в столбце в функцию COUNT() в качестве аргумента необходимо передать название столбца |
| AVG() | используется для вычисления среднего арифметического числовых значений поля | название столбца |
| MIN()  MAX() | используются для поиска минимального и максимального значений в поле | название столбца |
| SUM() | используется для вычисления суммы числовых значений поля | название столбца |
| GROUP\_CONCAT() | используется для перечисления значений поля через запятую | название столбца  Для замены разделителя после аргумента SEPARATOR необходимо передать разделитель |

*Внутри функций AVG(), COUNT(), SUM() и GROUP\_CONCAT() можно использовать ключевое слово DISTINCT, чтобы в итоговых вычислениях участвовали лишь уникальные значения поля*

*Агрегатные функции не могут использоваться в блоке WHERE*

SELECT GROUP\_CONCAT(DISTINCT trackname ORDER BY trackname) AS songs

FROM Songs

WHERE id <= 5;

GROUP BY – группировка по найденым записям

HAVING – фильтрация группы

SELECT artist, SUM(streams) AS streams

FROM Songs

GROUP BY artist

HAVING SUM(streams) > 50000;

В запросе записи сперва группируются по полю artist, однако в результирующую таблицу попадают группы, сумма значений поля streams которых превышает 50000. тоговом значении группы, а не на значениях извлеченных из таблицы записей.

Оператор WHERE фильтрует записи до того, как данные будут сгруппированы, а оператор HAVING — после того, как данные были сгруппированы.

Для сортировки групп не предусмотрен какой-либо отдельный оператор, поэтому она выполняется с помощью оператора ORDER BY

При выполнении SQL-запроса операторы выполняются в следующем порядке

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| FROM | Таблица для извлечения данных |
| WHERE | Фильтрация на уровне записей |
| GROUP BY | Создание групп |
| HAVING | Фильтрация на уровне групп |
| SELECT | Данные для извлечения |
| ORDER BY | Порядок сортировки результатов |
| LIMIT | Ограничение количества записей |

При формировании SQL-запроса необходимо соблюдать порядок используемых операторов

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| SELECT | Данные для извлечения |
| FROM | Таблица для извлечения данных |
| WHERE | Фильтрация на уровне записей |
| GROUP BY | Создание групп |
| HAVING | Фильтрация на уровне групп |
| ORDER BY | Порядок сортировки результатов |
| LIMIT | Ограничение количества записей |

Группировать оператором GROUP BY можно по псевдониму.

При фильтрации и сортировке групп также можно пользоваться псевдонимами.

Если псевдоним вычисляемого поля совпадает с именем какого-либо поля таблицы, группировка и фильтрация будет выполняться именно по значениям поля таблицы.

1. Операторы JOIN, LEFT JOIN, RIGHT JOIN, CROSS JOIN, UNION

В SQL соединение выполняется с помощью оператора JOIN и ключевого слова ON. Сначала указывается первая таблица, затем оператор JOIN, а после вторая таблица. Завершается выражение ключевым словом ON, после которого располагается условие соединения.

SELECT title, author

FROM Books JOIN Authors ON Books.author\_id = Authors.id;

Соединения бывают внутренними и внешними  
Внутренние соединения выполняются с помощью оператора JOIN и в финальную таблицу включают записи, удовлетворяющие условие после ON.  
Внешние соединения выполняются операторами LEFT JOIN, RIGHT JOIN, CROSS JOIN.

При использовании оператора LEFT JOIN к левой таблице будут добавляться записи, удовлетворяющие условию после ON. Однако записи правой таблицы, не попадающие под условие будут добавлены со значением NULL.

SELECT \*

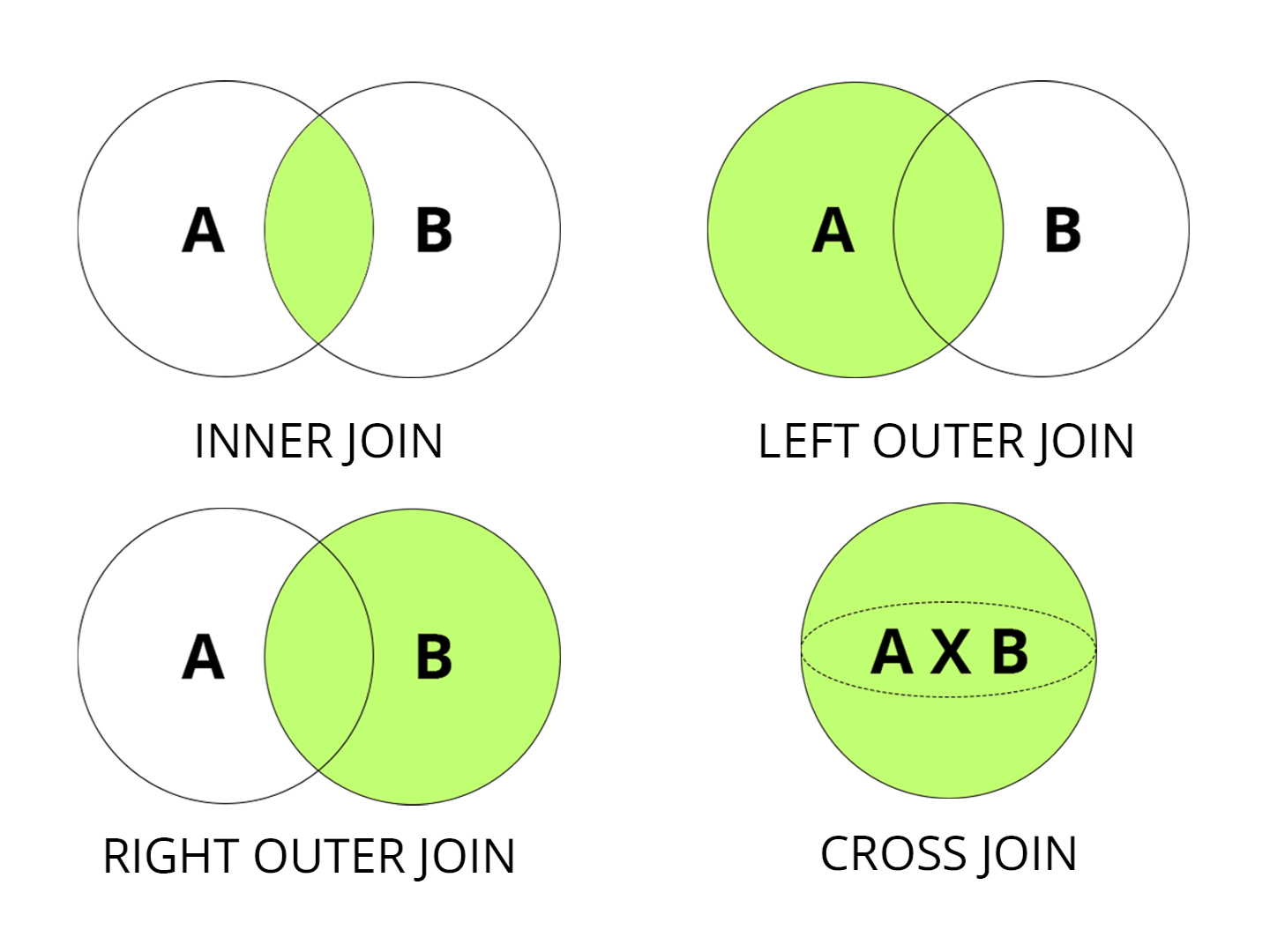
FROM Books LEFT JOIN Authors ON Books.author\_id = Authors.id;

При использовании оператора RIGHT JOIN к правой таблице будут добавляться записи, удовлетворяющие условию после ON. Однако записи левой таблицы, не попадающие под условие будут добавлены со значением NULL

SELECT \*

FROM Books RIGHT JOIN Authors ON Books.author\_id =Authors.id;

Левое и правое внешние соединения являются взаимозаменяемыми. Необходимо лишь изменить порядок обращения к таблицам.  
  
Перекрёстное соединение CROSS JOIN является декартовым произведением множеств. То есть позволяет перемешать каждую запись правой таблицы с каждой записью левой таблицы. Поэтому нет необходимости использовать условие после ON. Также порядок использования таблиц не имеет значения.



Оператор UNION дает возможность объединить несколько результатов запросов в один результирующий набор. Он указывается между запросами, результаты которых необходимо объединить.  
При использовании оператора UNION объединяемые результаты запросов обязательно должны иметь одинаковое количество полей. Названия полей в объединяемых результатах запросов необязательно должны быть одинаковыми, поскольку в качестве итоговых названий полей используются названия самого первого результата запроса. Типы значений в соответствующих полях объединяемых результатов могут быть различными

После оператора UNION может быть указано одно из ключевых слов DISTINCT или ALL. Если указано ключевое слово DISTINCT, то после объединения результатов запросов будет выполнено дополнительное удаление одинаковых записей. Если за оператором UNION следует ключевое слово ALL, то после объединения результатов запросов дополнительное удаление одинаковых записей выполнено не будет.  
Чтобы воспользоваться оператором ORDER BY и/или LIMIT в одном или нескольких запросах, расположенных между оператором UNION, эти запросы необходимо заключить в круглые скобки. Если оператор ORDER BY появляется в одном или нескольких объединяемых запросах без оператора LIMIT, то в целях оптимизации сортировка даже не выполняется.

(SELECT id, name, surname

FROM Students

ORDER BY id DESC)

UNION

(SELECT id, name, surname

FROM Teachers

ORDER BY id DESC);

Хоть с помощью оператора ORDER BY и нельзя выполнить сортировку каждого отдельного результата запроса, с помощью него можно отсортировать полученную после объединения всех результатов таблицу. Для этого нужно единожды указать оператор ORDER BY после самого последнего объединяемого запроса.

SELECT id, name, surname

FROM Students

UNION

SELECT id, name, surname

FROM Teachers ORDER BY id;

1. Обновление, удаление, добавление данных

Обновление отдельных записей

UPDATE Books

SET price = 1.99

WHERE amount = 1;

Обновление с вычисляемым значением

UPDATE Books

SET price = price \* 0.9;

Обновление с условной конструкцией

UPDATE Books

SET price = CASE

WHEN amount < 5 THEN price \* 0.9

WHEN amount BETWEEN 5 AND 10 THEN price \* 0.7

ELSE price \* 0.5

END;

Обновление на основе данных из других таблиц

UPDATE Books

SET price = price \* 0.5

WHERE author\_id =(SELECT id FROM Authors WHERE author = 'Stephen King');

Ограничение обновляемых записей

UPDATE Books

SET price = price + 2

LIMIT 1;

Обновление с сортировкой

UPDATE Books

SET id = id + 1

ORDER BY id DESC;

При обновлении данных таблицы на основе данных из других таблиц с помощью соединения нельзя использовать сортировку и оператор LIMIT

При обновлении данных таблицы на основе данных из других таблиц с помощью подзапроса в подзапросе нельзя обращаться к обновляемой таблице.

Удаление отдельных записей

Удаление происходит с помощью оператора DELETE

DELETE FROM Books

WHERE author\_id = (SELECT id FROM Authors WHERE author = 'Chuck Palahniuk');

Для того чтобы воспользоваться соединением, необходимо после названия таблицы, записи которой требуют удаления, указать ключевое слово USING, а затем предоставить выражение, выполняющее соединение.

DELETE FROM Books

USING Books JOIN Authors ON Books.author\_id = Authors.id WHERE Authors.author = 'Chuck Palahniuk';

Для того чтобы корректно удалить данные, от которых могут зависеть другие данные, сперва необходимо избавиться от всех потенциальных зависимостей или убедиться, что таких зависимостей нет. Для удаления информации об авторе Jerome Salinger сначала нужно удалить информацию обо всех принадлежащих ему книгах.

DELETE FROM Books

WHERE author\_id = (SELECT id FROM Authors WHERE author = 'Jerome Salinger');

DELETE FROM Authors

WHERE author = 'Jerome Salinger';

Если необходимо удалить все записи таблицы, рекомендуется использовать оператор TRUNCATE, а не DELETE

TRUNCATE Books

При удалении данных из таблиц на основе данных из других таблиц с помощью соединения нельзя использовать сортировку.

Добавление записей

Добавление происходит с помощью оператора INSERT

Если при добавлении записи в таблицу для какого-либо ее поля нет соответствующего значения, следует использовать значение NULL.

INSERT INTO Books (id, title, author, price)

VALUES (6, 'Animal Farm', 'George Orwell', NULL),

(7, 'Lord of the Flies', 'William Golding', 9.99 \* 0.6);

Сочетание операторов INSERT и SET позволяет добавить в таблицу лишь одну запись.

INSERT INTO Books

SET id = 6,

title = 'Animal Farm',

author = 'George Orwell',

price = 9.99 \* 0.6;

Добавление данных из других таблиц

INSERT INTO Books (id, title, author, price)

SELECT id, title, author, price

FROM NewBooks;

Оператор UPDATE, INSERT и DELETE поддерживает дополнительное ключевое слово IGNORE, которое позволяет не прерывать процесс обновления данных даже при возникновении ошибок.

Если поле имеет значение по умолчанию, то при добавлении записи значение для такого поля, как и само поле, можно не указывать.

В SQL существует оператор REPLACE, который работает как INSERT и UPDATE одновременно: запись добавляется в таблицу, если значение ее первичного ключа уникально. Если же оно совпадает со значением первичного ключа другой записи, то добавляемая запись заменяет эту запись.

1. Создание таблиц

Таблицы и поля всегда следует называть осмысленно и прозрачно. Подробнее с правилами именования можно [ознакомиться в руководстве по ссылке.](https://www.sqlstyle.guide/ru/#соглашения-о-наименовании)

CREATE TABLE IF NOT EXISTS Books

(

id INT UNIQUE,

title VARCHAR(40) DEFAULT 'Untitled',

author VARCHAR(40) DEFAULT 'Unknown',

fulltitle VARCHAR(40) DEFAULT (CONCAT(title, ' by ', author)),

price INT NOT NULL,

pieces INT CHECK (pieces > 0)

);

Если в качестве значения по умолчанию используется выражение, оно должно быть заключено в круглые скобки

Чтобы перед созданием таблицы выполнить проверку того, что в базе данных нет таблицы с таким же именем, нужно после оператора CREATE TABLE указать сочетание ключевых слов IF NOT EXISTS

Ограничение UNIQUE используется для того, чтобы запретить полю хранить повторяющиеся значения. Если поле имеет ограничение UNIQUE, это значит, что при добавлении новой записи или обновлении существующей записи значение этого поля должно быть уникальным.

Ограничение DEFAULT используется для того, чтобы определить значение по умолчанию, которое примет поле в том случае, если при добавлении записи его значение не будет указано явно. Если поле имеет значение по умолчанию, тип этого значения должен соответствовать типу самого поля.

Ограничение NOT NULL используется для того, чтобы запретить полю хранить значение NULL. Если поле имеет ограничение NOT NULL, это значит, что при добавлении новой записи или обновлении существующей записи значение этого поля не должно быть равным NULL.

Ограничение CHECK используется для того, чтобы запретить полю хранить значения, не удовлетворяющие заданному условию. Если поле имеет ограничение CHECK, это значит, что при добавлении новой записи или обновлении существующей записи значение этого поля перед установкой будет проверено на корректность. Условие, используемое для проверки, представляет собой выражение, заключенное в круглые скобки, которое указывается после ключевого слова CHECK. Оно может быть составлено с использованием функций и различных операторов сравнения (=,>, LIKE и так далее), то есть иметь произвольную сложность.

Ограничение CHECK может ссылаться только на значение того поля, для которого применяется.

При использовании ограничений их можно комбинировать.

У каждого ограничения CHECK имеется свое имя, которое СУБД автоматически определяет следующим образом:

<название таблицы>\_chk\_<порядковый нормер ограничения>

Для удобства ограничения CHECK можно именовать вручную. Для этого необходимо вынести определение ограничения из определения поля и воспользоваться ключевым словом CONSTRAINT.

CREATE TABLE Books

(

id INT,

title VARCHAR(40),

author VARCHAR(40),

CONSTRAINT positive\_id CHECK (id > 0)

);

Если поле не имеет явно определенного значения по умолчанию, но поддерживает значение NULL, значением по умолчанию такого поля будет NULL.

Чтобы при создании таблицы определить какое-либо ее поле как первичный ключ, можно воспользоваться сочетанием ограничений NOT NULL и UNIQUE. Также в SQL для определения первичного ключа таблицы существует отдельное ограничение PRIMARY KEY

CREATE TABLE Books

(id INT PRIMARY KEY,

title VARCHAR(40),

author VARCHAR(40)

);

При необходимости задать первичный ключ, включающий два и более полей, нужно использовать другой синтаксис

CREATE TABLE Books

(title VARCHAR(40),

author VARCHAR(40),

PRIMARY KEY (title, author)

);

Для автоматического заполнения поля в качестве первичного ключа можно использовать ограничение AUTO\_INCREMENT. Ограничение AUTO\_INCREMENT может быть указано в определении лишь одного поля, причем это поле должно быть первичным ключом. Также ограничение AUTO\_INCREMENT не сочетается с ограничением DEFAULT

CREATE TABLE Books

(id INT PRIMARY KEY AUTO\_INCREMENT,

title VARCHAR(40),

author VARCHAR(40)

);

При добавлении записи в таблицу Books значение поля id можно не указывать, оно само примет нужное последовательное значение.

Ограничение AUTO\_INCREMENT определяет значение поля id, основываясь лишь на очередности добавления записи. Например, третья по счету добавляемая запись в качестве значения поля id примет число 3 даже в том случае, если какие-либо добавленные ранее записи были удалены. При использовании оператора TRUNCATE счетчик ограничения AUTO\_INCREMENT сбрасывается до 1.

Ограничение FOREIGN KEY предназначено для определения поля как внешнего ключа. Помимо построения связей между таблицами, оно выполняет постоянную поддержку согласованности этих связей. Если пренебрегать ограничением FOREIGN KEY, то в процессе работы с таблицами базы данных можно прийти к ситуации, в которой согласованность связанных данных будет нарушена, что, например, может означать то, что какая-либо таблица будет ссылаться на несуществующие данные.

CREATE TABLE Authors (

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(40),

surname VARCHAR(40)

);

CREATE TABLE Books

(id INT PRIMARY KEY AUTO\_INCREMENT,

title VARCHAR(40),

author\_id INT,

FOREIGN KEY (author\_id) REFERENCES Authors (id)

);

При создании таблицы Books существует ограничение FOREIGN KEY, после которого в скобках указывается поле, являющееся внешним ключом. В нашем случае это поле author\_id. Затем следует ключевое слово REFERENCES, а после него — имя родительской таблицы и вновь в скобках поле этой таблицы, на которое ссылается внешний ключ. В нашем случае внешний ключ ссылается на поле id таблицы Authors.

Внешний ключ и поле, на которое оно ссылается, должны иметь одинаковые типы данных. Размер и знак типов с фиксированной точностью, таких как INTEGER и DECIMAL, должны быть одинаковыми.

Внешний ключ не может принимать значения, которых нет в поле, на которое он ссылается.

Когда выполняется попытка изменить запись в родительской таблице (а именно значение ее поля, связанного с внешним ключом), от которой зависят записи в дочерней таблице, происходит ошибка.

Чтобы определить действие, которое должно быть выполнено при обновлении, нужно после определения внешнего ключа указать дополнительный оператор ON UPDATE, а чтобы определить действие, которое должно быть выполнено при удалении, — оператор ON DELETE

Само же действие определяется одним из трех ключевых слов:

CASCADE

SET NULL

RESTRICT

Оно указывается сразу после оператора ON UPDATE (ON DELETE) и задает соответствующее поведение при обновлении (удалении) связанных данных.

При использовании ключевого слова RESTRICT изменение или удаление связанных данных приводит к ошибке. Другими словами, данное ключевое слово задает поведение по умолчанию.

Если используется ключевое слово CASCADE, то при изменении или удалении данных в родительской таблице аналогичные действия будут автоматически применены и к связанным записям в дочерней таблице

Если используется ключевое слово SET NULL, то при изменении или удалении данных в родительской таблице связанные записи в дочерней таблице в качестве значения внешнего ключа примут значение NULL.

CREATE TABLE Authors

(

id INT AUTO\_INCREMENT,

name VARCHAR(40),

surname VARCHAR(40),

PRIMARY KEY (id, surname)

);

CREATE TABLE Books

(

id INT PRIMARY KEY AUTO\_INCREMENT,

author\_surname VARCHAR(40),

author\_id INT,

title VARCHAR(40),

FOREIGN KEY (author\_id, author\_surname) REFERENCES Authors (id, surname)

ON UPDATE CASCADE

ON DELETE SET NULL

);

Если необходимо получить информацию о полях таблицы, можно воспользоваться оператором DESCRIBE

1. Удаление, переименование и обновление таблиц

Удаление таблиц

DROP TABLE IF EXISTS Books;

Переименование таблицы

RENAME TABLE <текущее имя таблицы> TO <новое имя таблицы>

Обновление таблицы

Любой запрос, модифицирующий таблицу, начинается с оператора ALTER TABLE, после которого указывается имя таблицы, подлежащей модификации.

Удаление поля таблицы выполняется с помощью оператора DROP COLUMN.

ALTER TABLE Books

DROP COLUMN author;

Переименование поля таблицы выполняется с помощью оператора RENAME COLUMN

ALTER TABLE Books

RENAME COLUMN author TO writer;

Таблицу можно расширить, добавив ей новое поле оператором ADD COLUMN

ALTER TABLE Books

ADD COLUMN release\_year INT CHECK (release\_year > 0);

По умолчанию оператор ADD COLUMN добавляет новое поле в конец таблицы. Если после определения поля указать ключевое слово FIRST, то новое поле будет добавлено в начало таблицы.

ALTER TABLE Books

ADD COLUMN release\_year INT CHECK (release\_year > 0) FIRST;

Также новое поле можно поместить в конкретную часть таблицы. Для этого нужно указать ключевое слово AFTER, а затем предоставить название поля, после которого должно быть расположено новое поле.

ALTER TABLE Books

ADD COLUMN release\_year INT CHECK (release\_year > 0) AFTER title;

*Важно заметить, что если поле, которое использует ограничение NOT NULL и не имеет значения по умолчанию, добавляется в непустую таблицу, то для такого поля будет автоматически выбрано значение по умолчанию. Например, для целочисленного типа таким значением является 0, для строкового — пустая строка.*

Обновление поля

Поле существующей таблицы можно полностью переопределить, для этого необходимо воспользоваться оператором MODIFY COLUMN, а затем предоставить новое определение желаемого поля.

ALTER TABLE Books

MODIFY COLUMN title VARCHAR(60) CHECK (title != '');

*Изменение типа поля может привести к изменению хранимых в нем данных. К примеру, изменение типа поля с FLOAT на INT приведет к тому, что числа в этом поле будут округлены до целых значений.*

При необходимости задать полю значение по умолчанию предназначена специальная связка операторов ALTER COLUMN и SET DEFAULT

ALTER COLUMN <название поля> SET DEFAULT <значение по умолчанию>

Воспользовавшись связкой операторов ALTER COLUMN и DROP DEFAULT, можно удалить значение по умолчанию, которым обладает поле.

ALTER COLUMN <название поля> DROP DEFAULT

Добавление и удаление первичного ключа

Для этого необходимо воспользоваться оператором ADD PRIMARY KEY, а затем в скобках указать поле, которое должно стать первичным ключом. Если первичный ключ должен состоять из нескольких полей, их нужно перечислить через запятую.

Удаление выполняется даже проще, чем добавление, для этого всего лишь необходимо воспользоваться оператором DROP PRIMARY KEY.

ALTER TABLE Books

ADD PRIMARY KEY (id);

ALTER TABLE Books

DROP PRIMARY KEY;

*Если таблица имеет первичный ключ, то при попытке добавить ей другой первичный ключ произойдет ошибка. Поэтому, если требуется обновить первичный ключ таблицы, сначала следует избавиться от имеющегося.*

Добавление и удаление внешнего ключа

Для этого необходимо воспользоваться оператором ADD FOREIGN KEY, а затем определить внешний ключ.

ALTER TABLE Books

ADD FOREIGN KEY (publisher) REFERENCES Publishers (name);

ALTER TABLE Books

DROP FOREIGN KEY publisher;

Добавление и удаление дополнительной проверки

Ограничение CHECK, как и значение по умолчанию, может быть добавлено отдельно и не требует полного переопределения поля. Для того чтобы добавить ограничение CHECK, нужно воспользоваться оператором ADD CHECK, а затем в скобках указать проверяющее выражение.

ALTER TABLE Books

ADD CHECK (title != '');

При необходимости добавить в таблицу именованное ограничение CHECK можно воспользоваться оператором ADD CONSTRAINT, синтаксис которого имеет следующий вид:

ADD CONSTRAINT <имя ограничения> CHECK (<проверяющее выражение>)

Ограничение CHECK может быть как добавлено, так и удалено. Удаление выполняется даже проще, чем добавление, для этого всего лишь необходимо воспользоваться оператором DROP CONSTRAINT, а затем указать имя ограничения CHECK, которое требуется удалить.

ALTER TABLE Books

DROP CONSTRAINT title\_is\_not\_empty\_string;

Оператор ALTER TABLE в своем теле может содержать как одно действие, так и несколько. Во втором случае все действия перечисляются через запятую.

ALTER TABLE Books

ADD COLUMN release\_year INT,

ADD COLUMN publisher VARCHAR(40);

Если необходимо сразу переименовать и переопределить поле, можно воспользоваться оператором CHANGE COLUMN, который выполняет эти операции за один раз. Синтаксис данного оператора имеет следующий вид:

CHANGE COLUMN <текущее имя поля> <новое определение поля, включающее имя, тип и набор ограничений>

1. Индексы

Индекс — это структура данных, предназначенная для ускорения поиска и сортировки данных. Индексы позволяют быстро находить нужные записи таблицы на основе значений одного или нескольких полей. Без индексов поиск данных в больших таблицах может быть медленным и ресурсоемким, поскольку подразумевает просмотр всей таблицы от первой записи до последней.

Кластеризованный индекс

Кластеризованные индексы хранят записи таблицы, отсортированные по их ключевым значениям. Эти ключевые значения — это поля, включенные в определение индекса. Существует только один кластеризованный индекс для каждой таблицы, так как записи могут храниться в единственном порядке.

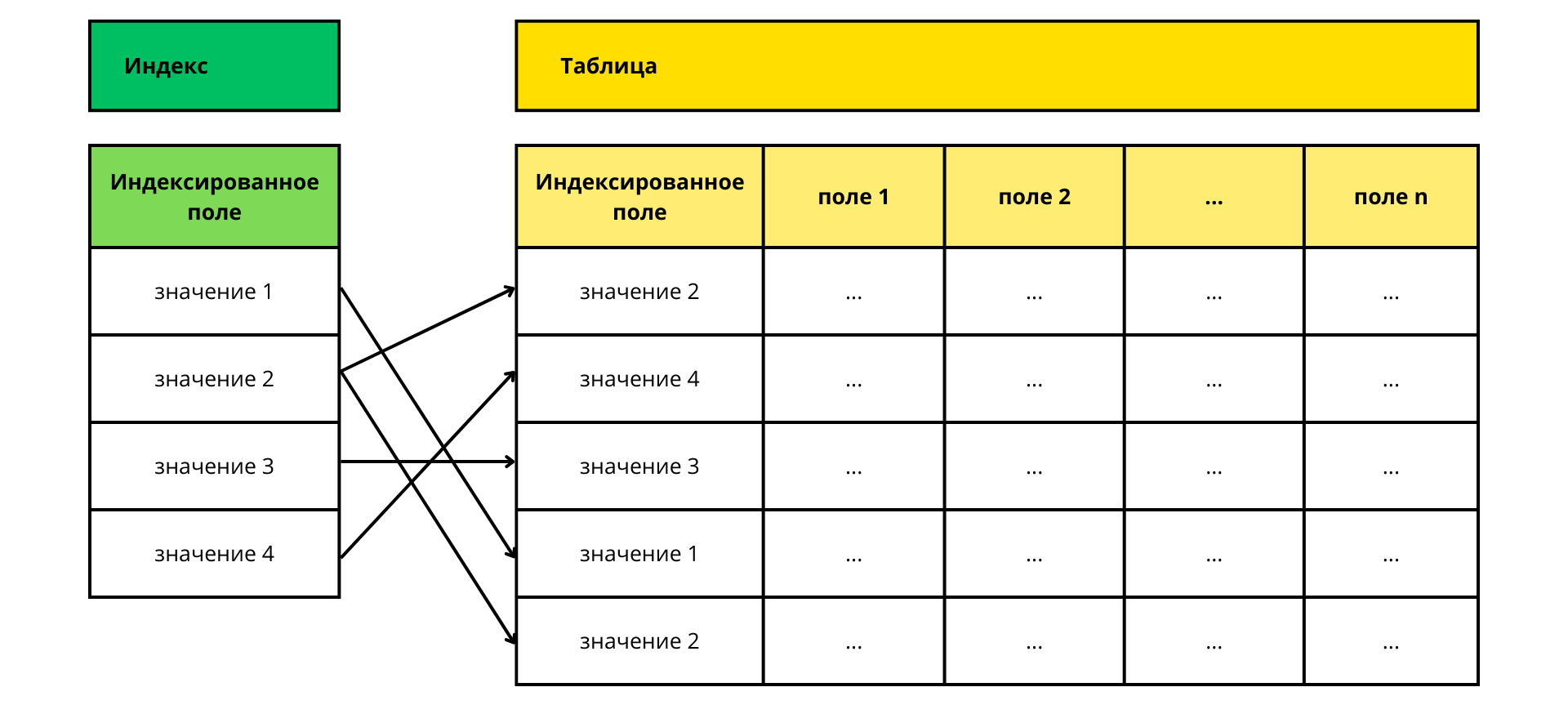
СУБД использует первичный ключ для определения порядка хранения записей в кластеризованном индексе. Если первичный ключ отсутствует, СУБД автоматически выбирает в качестве первичного ключа первое уникальное непустое поле таблицы и впоследствии также использует его для определения порядка хранения записей в индексе. Если таблица не имеет ни первичного ключа, ни уникального непустого поля, СУБД генерирует скрытое суррогатное (синтетическое) поле для хранения целочисленных идентификаторов записей и использует его в качестве первичного ключа.

Кластеризованный индекс организует физический порядок хранения записей таблицы. Это означает, что записи, имеющие близкие ключевые значения, также будут близко расположены в памяти. Данная особенность кластеризованного индекса обеспечивает быстрый поиск записей (посредством бинарного поиска) по значениям первичного ключа, особенно по диапазонным значениям.

Некластеризованный индекс

Кластеризованный индекс позволяет выполнять быстрый поиск и сортировку записей таблицы, однако лишь на основе значений первичного ключа. Для применения преимуществ кластеризованного индекса к любому полю таблицы используются некластеризованные индексы.

Некластеризованный индекс представляет собой структуру данных, которая содержит уникальные значения поля таблицы, указанного в качестве индекса. Каждое значение некластеризованного индекса связано со всеми записями таблицы, которые имеют такое же значение в индексированном поле. Также для дополнительной эффективности некластеризованный индекс хранит значения индексированного поля в упорядоченном виде.



Сам некластеризованный индекс хранится отдельно от таблицы в качестве самостоятельного объекта и содержит лишь значения индексированного поля и ссылки на соответствующие записи. Это означает, что данные таблицы могут быть организованы на диске в произвольном порядке, но некластеризованный индекс все равно позволит эффективно находить нужные записи.

Таким образом, основная идея некластеризованного индекса состоит в том, чтобы создать вспомогательную структуру данных, связанную с некоторым полем таблицы и ее записями, которая обеспечит быстрый поиск и сортировку записей таблицы по данному полю.

Создание индексов

CREATE INDEX <имя некластеризованного индекса>

ON <имя таблицы>(<имя индексируемого поля>);

Кластеризованный индекс не требует создания, поскольку им обладает любая таблица. Поэтому в SQL вручную можно создать лишь некластеризованный индекс.

Некластеризованный индекс может быть создан на основе как одного поля, так и нескольких.

Удаление индексов

DROP INDEX <текущее имя индекса>

ON <имя таблицы>;

Переименование индексов

ALTER TABLE <имя таблицы>

RENAME INDEX <текущее имя индекса> TO <новое имя индекса>;

*Индексы не стоит использовать для тех полей, которые редко используются при фильтрации записей, а также тех, которые часто обновляются.*

*Индексы ускоряют выполнение операции извлечения, однако при этом они могут замедлить операции вставки, обновления и удаления, так как эти операции требуют обновления индексов.*

1. Тригеры

Создание триггеров

Триггер — это запрос, который автоматически выполняется во время применения операций INSERT, DELETE или UPDATE по отношению к определенной таблице.

Общий шаблон для создания триггера имеет следующий вид:

DELIMITER //

CREATE TRIGGER <имя триггера>

<время срабатывания триггера>

<операция, которая вызывает триггер>

ON <имя таблицы>

FOR EACH ROW

BEGIN <тело триггера>;

END // DELIMITER ;

Шаблон начинается не совсем с создания триггера, а с ключевого слова DELIMITER. Оно используется для изменения разделителя, которым по умолчанию является точка с запятой (;). Здесь в качестве нового разделителя устанавливается двойной слэш (//).

Изменение разделителя перед созданием триггера является важной частью, поскольку запрос в теле триггера должен быть завершен символом ;. Для разделения фрагментов запроса в теле триггера также используется точка с запятой. Если заранее не переопределить разделитель, СУБД посчитает, что символ ; в теле триггера завершает не запрос, а создание самого триггера, что приведет к ошибке, поскольку в таком случае создание триггера будет оборвано на половине.

Непосредственно создание триггера начинается с оператора CREATE TRIGGER, после которого указывается имя триггера. Затем определяются время срабатывания триггера и операция, которая вызывает триггер. Время срабатывания определяется одним из операторов:

BEFORE INSERT — имеет доступ к значениям добавляемой записи; может изменять значения добавляемой записи

AFTER INSERT — имеет доступ к значениям добавленной записи

BEFORE UPDATE — имеет доступ к старым и новым значениям обновляемой записи; может изменять новые значения обновляемой записи

AFTER UPDATE — имеет доступ к старым и новым значениям обновленной записи

BEFORE DELETE — имеет доступ к значениям удаляемой записи

AFTER DELETE — имеет доступ к значениям удаленной записи

После операции, вызывающей триггер, указывается ключевое слово ON и название таблицы, для которой создается триггер. Затем следует набор ключевых слов FOR EACH ROW. Внутрь последующего блока BEGIN END помещается запрос, который и будет выполняться каждый раз при срабатывании триггера. Завершается создание триггера обновленным разделителем // и возвратом к стандартному разделителю в виде символа ;.

Для обращения к старым значениям записи используется переменная OLD, к новым — NEW.

Также некоторые тригеры триггер могут изменять записи. Выполняется это с помощью оператора SET, после которого указывается имя поля, а затем знак равенства и новое значение.

Удаление триггеров

Для этого необходимо воспользоваться оператором DROP TRIGGER, а затем указать имя удаляемого триггера.

DROP TRIGGER name\_and\_surname\_formatting;

Если тело триггера представляет собой одиночный запрос, то можно воспользоваться упрощенным синтаксисом создания триггера:

CREATE TRIGGER <имя триггера>

<время срабатывания триггера>

<операция, которая вызывает триггер>

ON <имя таблицы>

FOR EACH ROW

<тело триггера>

При использовании упрощенного синтаксиса не нужно помещать тело триггера в блок BEGIN END, а также изменять стандартный разделитель.

1. Представления

Представление — это виртуальная таблица. В отличие от обычных таблиц, содержащих данные, представление содержит запрос, который динамически извлекает данные тогда, когда это необходимо. Представления можно использовать точно так же, как и обычные таблицы: извлекать данные из них, фильтровать и сортировать, объединять с другими таблицами или представлениями.

Представления не содержат фактических данных, они извлекают их из других таблиц. Поэтому, если данные этих таблиц изменятся, изменится и содержимое связанных с ними представлений.

Создание представлений

Для создания представлений используется оператор CREATE VIEW, синтаксис которого имеет следующий вид:

CREATE VIEW <имя представления> <(имя\_псевдоним поля1, имя\_псевдоним поля2)> AS <извлекающий запрос>

Представления, как и таблицы, должны иметь уникальные имена. Они не могут быть названы так же, как другие таблицы или представления.

Если в базе данных содержится некоторое представление, то при попытке создать представление с таким же именем произойдет ошибка. Чтобы ее избежать, можно воспользоваться расширенным оператором CREATE OR REPLACE VIEW. Данный оператор создаст представление только в том случае, если оно еще не существует. Если же представление с создаваемым именем существует, оно будет заменено новым.

*При определении представления нельзя использовать подзапрос в блоке оператора FROM.*

Изменение представлений

Запрос, который используется в представлении, может быть изменен. Для этого используется оператор ALTER VIEW, синтаксис которого имеет следующий вид:

ALTER VIEW <имя представления> AS <новый извлекающий запрос>

Удаление представлений

Представление может быть удалено. Для этого используется оператор DROP VIEW, синтаксис которого имеет следующий вид:

DROP VIEW IF EXISTS <имя представления>

*С помощью представлений можно ограничивать доступ к данным, предоставляя пользователю или приложению права не на таблицу, а на представление. Также представления могут использоваться для обеспечения интерфейса обратной совместимости: если структура таблицы, с которой работало приложение, изменилась, с помощью представления можно смоделировать ее прежний вид.*

1. Обобщенные табличные выражения

Обычное обобщенное табличное выражение

Обобщенное табличное выражение или CTE (Common Table Expressions) — это временная таблица, к которой можно обращаться в рамках одного запроса. Для простоты CTE можно считать именованным подзапросом, который определен отдельно от основного запроса. Использование CTE позволяет писать сложные запросы в более читаемой форме путем их разбиения на небольшие логические шаги.

Для написания запросов с использованием CTE предназначен оператор WITH, синтаксис которого имеет следующий вид:

WITH <имя CTE> <(имя\_псевдоним поля1, имя\_псевдоним поля2)> AS (

<извлекающий запрос, определяющий содержимое CTE>

)

<основной запрос, который может обращаться к CTE>

Определение CTE не отделяется от основного запроса точкой с запятой, поскольку вся конструкция (определение CTE и основной запрос) воспринимается как одно целое.

После оператора WITH допустимо определить не только одно CTE, но и несколько, для этого достаточно перечислить их определения через запятую. Если блок WITH включает определение нескольких CTE, то их создание выполняется последовательно: сначала создается первое CTE, затем второе, и так далее. Таким образом, во время определения очередного CTE, все определенные ранее CTE уже являются созданными, поэтому каждое следующее определенное CTE может ссылаться на любое предыдущее.

*CTE является физической таблицей, которая перед выполнением основного запроса явно создается и помещается в оперативную память системы, пусть и на довольно небольшой промежуток времени.*

*Несмотря на сходство представлений и CTE, между ними есть различие.*

*Представление является реальным объектом базы данных — виртуальной таблицей, которая определяется единожды, а затем повсеместно используется при необходимости. CTE является временной таблицей, которая привязывается к одному конкретному запросу и существует лишь в рамках этого запроса.*

*Представление является составляющей базы данных, CTE — составляющей одного запроса.*

Рекурсивное обобщенное табличное выражение

Рекурсивное обобщенное табличное выражение или рекурсивное CTE — это CTE, содержимое которого последовательно формируется на основе самого себя. Тело рекурсивного CTE, включает два извлекающих запроса. Первый запрос используется для однозначного определения содержимого первой записи CTE, второй — для определения правила формирования всех остальных записей. Содержимое каждой записи рекурсивного CTE, кроме первой, зависит только от содержимого предыдущей записи.

Синтаксис создания рекурсивного CTE практически повторяет синтаксис создания обычного CTE и имеет следующий вид

WITH RECURSIVE <имя CTE> AS (

<извлекающий запрос, определяющий первую запись CTE>

<пара ключевых слов UNION ALL или UNION DISTINCT>

<извлекающий запрос, определяющий правило формирования очередной записи CTE (второй, третьей, и так далее)>

)

<основной запрос, который может обращаться к рекурсивному CTE>

При использовании связки ключевых слов UNION ALL CTE будет включать повторяющиеся записи, если при формировании такие возникнут.

При использовании UNION DISTINCT все повторяющиеся записи будут удалены.

По умолчанию рекурсивный запрос в рекурсивном CTE может выполнить не более 1000 итераций

Увеличив значение системной переменной cte\_max\_recursion\_depth, отвечающей за максимально допустимое количество итераций рекурсивного запроса, данное ограничение можно обойти.

SET @@cte\_max\_recursion\_depth := 10000;

Типы данных полей рекурсивного CTE определяются в нерекурсивном запросе. Об этом полезно помнить в том случае, когда CTE имеет строковое поле, поскольку в рекурсивном запросе могут происходить попытки добавить в поле строки, длина которых превышает поддерживаемые полем значения.

Для решения данной проблемы перед добавлением значения в поле field можно сконвертировать его в более широкий тип, например, с помощью функции CONVERT()(см. Пример\_1)

Пример\_1

WITH RECURSIVE Beegeek AS (

SELECT CONVERT('bee', CHAR(50)) AS field -- пятидесяти символьное строковое поле

UNION ALL

SELECT CONCAT(field, field)

FROM Beegeek

LIMIT 5

)

SELECT \* FROM Beegeek;

Пример\_2

WITH RECURSIVE NaturalNumbers AS (

SELECT 1 AS number -- нерекурсивный запрос

UNION ALL

SELECT number + 1 -- рекурсивный запрос

FROM NaturalNumbers WHERE number < 10

)

SELECT \* FROM NaturalNumbers;

Пример\_3

WITH RECURSIVE OddNumbers AS(

SELECT 1 AS number

UNION ALL

SELECT number + 2

FROM OddNumbers

LIMIT 10

)

SELECT \* FROM OddNumbers;

*В рекурсивном запросе рекурсивного CTE не могут использоваться агрегатные функции, а также операторы GROUP BY, ORDER BY и DISTINCT*

1. Пользовательские функции, процедуры и переменные

Переменная — это именованная область памяти, которая хранит определенные данные и позволяет неоднократно к ним обращаться.

Каждая переменная имеет уникальное имя, по которому к ней можно обращаться. Конкретные данные, которые хранятся в переменной, называют значением переменной.

Переменные делятся на два типа:

1. Пользовательские переменные
2. Системные переменные
   1. Сессионные системные
   2. Глобальные системные

Системные переменные используются для хранения технической информации и настройки параметров СУБД в соответствии с требованиями конкретного пользователя или приложения.

Глобальные переменные определяет администратор базы данных. Такие переменные создаются во время запуска базы данных, и для всех пользователей, которые в будущем подключатся к ней для работы, значения глобальных переменных будут равны и изменить их они не смогут.

Сессионные переменные создаются во время каждого подключения к базе данных, и их значения пользователям изменять допускается.

Сессионные переменные создаются во время подключения к базе данных для каждого подключения отдельно. Если один пользователь изменит значение сессионной переменной, то у других пользователей оно НЕ изменится.

[*Полный список системных переменных и их описание*](https://vds-admin.ru/mysql/sistemnye-peremennye-mysql-servera-fail-mycnf#time_format)

Обладая правами администратора, с системной переменной можно работать как глобально, так и сессионно.

Пользовательские переменные определяются программистом, а не СУБД, и используются при составлении запросов для определенных целей.

Создание и обращение к пользовательской переменной

Перед именем пользовательской переменной располагается знак @. Его необходимо использовать при создании пользовательской переменной и при обращении к ней.

SET @variable1 := 1,

@variable2 := 2,

@min\_price := (SELECT MIN(price) FROM Books);

SELECT @min\_price, @variable1, @variable2;

Определить пользовательскую переменную можно прямо внутри извлекающего запроса в блоке оператора SELECT без использования ключевого слова SET.

SELECT MIN(price), MAX(price) INTO @min\_price, @max\_price

FROM Books;

*Пользовательские переменные нечувствительны к регистру*

*Пользовательские переменные нельзя использовать в блоке оператора LIMIT*

*Пользовательская переменная, определенная одним пользователем, не видна другим пользователям.*

Создание и обращение к системной переменной

Перед именем системной переменной располагается знак @@. Его необходимо использовать при создании системной переменной и при обращении к ней.

SET @@session.sort\_buffer\_size := 262146;

SELECT @@global.sort\_buffer\_size, -- глобальное значение

@@session.sort\_buffer\_size; -- сессионное значение

Пользовательские функции

Пользовательские функции делятся на два вида:

1. Детерминированные – функция не обращается ни к каким данным кроме как к входным. Что-то подали на вход и получили предсказуемый результат на выходе.
2. Недетерминированные – функция обращается к каким-то данным из вне, например к какой-то таблице. Следовательно, изменились данные в таблице – изменился результат на выходе.

Создание пользовательских функций

Общий шаблон для создания пользовательской функции:

DELIMITER //

CREATE FUNCTION <имя функции>(<имя 1 параметра и его тип>, <имя 2 параметра и его тип>, ...)

RETURNS <тип возвращаемого значения>

<вид функции: детерминированная или недетерминированная>

BEGIN

<тело функции>;

RETURN <возвращаемое значение>;

END //

DELIMITER ;

После определения имени функции и всех ее параметров следует ключевое слово RETURNS, за которым указывается тип возвращаемого функцией значения. Затем определяется вид функции: детерминированная или недетерминированная. Если функция является детерминированной, указывается ключевое слово DETERMINISTIC, если недетерминированной — NOT DETERMINISTIC.

Если используется пара ключевых слов NOT DETERMINISTIC, обязательно нужно пояснить, почему функция является недетерминированной. Если функция работает с информацией, хранящейся в таблицах базы данных, после NOT DETERMINISTIC с новой строки необходимо добавить READS SQL DATA. Если функция не обращается к таблицам базы данных, но при этом выполняет вычисления со случайным результатом, после NOT DETERMINISTIC с новой строки необходимо добавить NO SQL.

Если тело пользовательской функции достаточно непростое и включает несколько этапов, то для хранения результатов промежуточных вычислений можно использовать дополнительно определенные локальные переменные. Поведение локальных переменных несколько похоже на поведение пользовательских переменных, поскольку значения им присваиваются с помощью той же связки операторов SET и :=. Однако локальные переменные имеют важную особенность: перед тем, как им присвоить значение, их необходимо определить. При определении локальной переменной для нее можно указать значение по умолчанию, которое переменная примет сразу же после определения. Для этого после типа данных переменной необходимо указать ключевое слово DEFAULT и предоставить желаемое значение.

Определение локальной переменной внутри функций выполняется с помощью оператора DECLARE, синтаксис использования которого имеет следующий вид:

DECLARE <имя переменной> <тип данных значения, хранимого в переменной> DEFAULT <значение переменной по умолчанию> ;

Пользовательскую функцию можно дополнить строкой документации, которая полезна тем, что позволяет программисту сразу понять, что именно делает функция, не вдаваясь в подробности ее реализации. Для этого перед блоком BEGIN END необходимо указать ключевое слово COMMENT.

Пользовательская функция может быть удалена после создания. Для этого нужно воспользоваться оператором DROP FUNCTION, а затем указать имя удаляемой функции.

DROP FUNCTION IS\_EVEN <имя функции> ;

Пример\_1:

DELIMITER //

CREATE FUNCTION CAPITALIZE(string TEXT)

RETURNS TEXT

DETERMINISTIC

BEGIN

RETURN CONCAT(UPPER(LEFT(string, 1)), LOWER(SUBSTRING(string, 2)));

END //

DELIMITER ;

SELECT CAPITALIZE(‘Hallow world’);

Пример\_2:

DELIMITER //

CREATE FUNCTION MIN\_MAX\_GRADE()

RETURNS TEXT

NOT DETERMINISTIC

READS SQL DATA

COMMENT 'Функция возвращает строку с максимальной и минимальной оценкой'

BEGIN

DECLARE min\_grade INT DEFAULT 0;

DECLARE max\_grade INT;

SET min\_grade := (SELECT MIN(grade)

FROM Math);

SET max\_grade := (SELECT MAX(grade)

FROM Math);

RETURN CONCAT('Lowest grade: ', min\_grade, ', Highest grade: ', max\_grade);

END //

DELIMITER;

SELECT MIN\_MAX\_GRADE();

DROP FUNCTION MIN\_MAX\_GRADE;

Пример\_3:

DELIMITER //

CREATE FUNCTION CALCULATE(a FLOAT, b FLOAT, operation TEXT)

RETURNS FLOAT

DETERMINISTIC

BEGIN

DECLARE res FLOAT;

CASE

WHEN operation = '+' THEN SET res := a + b;

WHEN operation = '-' THEN SET res := a - b;

WHEN operation = '\*' THEN SET res := a \* b;

WHEN operation = '/' THEN SET res := a / b;

END CASE;

RETURN res;

END //

DELIMITER;

Условная конструкция IF-ELSEIF-ELSE

При необходимости проверить одно или несколько условий и в зависимости от истинности одного из них выполнить ряд операций, используется условная конструкция IF-ELSEIF-ELSE.

Синтаксис данной конструкции близок к синтаксису оператора CASE и по большому счету отличается лишь тем, что вместо возвращаемых значений в нем указываются целые операции.

Синтаксис условной конструкции IF-ELSEIF-ELSE:

IF <первое проверяемое условие> THEN

<операция №1>;

<операция №2>;

<операция №3>;

ELSEIF <второе проверяемое условие> THEN

<операция №1>;

<операция №2>;

ELSEIF <третье проверяемое условие> THEN

<операция №1>;

<операция №2>;

<операция №3>;

ELSE <операция №1>;

END IF;

Цикл WHILE

Синтаксис цикла WHILE:

<имя цикла>: WHILE <проверяемое условие> DO

IF <проверяемое условие для досрочного выхода> THEN

LEAVE <имя цикла>;

ELSEIF < проверяемое условие для пропуска (аналог CONTINUE)> THEN

ITERATE <имя цикла>;

END IF;

<тело цикла>;

END WHILE <имя цикла WHILE>;

*Условную конструкцию IF-ELSEIF-ELSE и цикл WHILE нельзя использовать вне функций.*

1. **Хранимые процедуры**

Подпрограмма — одно из ключевых понятий в программировании. Она представляет собой набор операций, который решает определенную задачу. Подпрограмма может многократно вызываться при любой необходимости выполнить заключенные в ней операции. Назначение подпрограмм заключается в структурировании кода, разделении его на логические части и повторном использовании одного и того же кода.

Как и функции, процедуры заключают в себе определенный набор операций, который выполняется при их вызове. Также процедуры могут принимать различные аргументы, которые могут быть использованы в их теле. Однако, процедуры не имеют возвращаемого значения.

Шаблон для создания хранимой процедуры имеет следующий вид:

DELIMITER //

CREATE PROCEDURE <имя процедуры> ()

BEGIN

<тело процедуры>;

END //

DELIMITER ;

CALL <имя процедуры>;

При создании хранимой процедуры ее можно определить как детерминированную или недетерминированную. Пояснять, почему процедура является недетерминированной необязательно. Также можно дополнить строкой документации.

DELIMITER //

CREATE PROCEDURE HIGH\_GRADE\_STUDENTS()

NOT DETERMINISTIC

COMMENT 'Процедура извлекает информацию о студентах, получивших оценку 4 или выше'

BEGIN

SELECT \*

FROM Math

WHERE grade >= 4;

END //

DELIMITER ;

Хранимая процедура может быть удалена после создания.

DROP PROCEDURE HIGH\_GRADE\_STUDENTS;

Хранимые процедуры с входными параметрами

Поведение входных параметров процедуры полностью совпадает с поведением обычных параметров функции.

Шаблон для создания хранимой процедуры с параметрами:

DELIMITER //

CREATE PROCEDURE <имя процедуры> (IN <1 параметр> <тип 1 параметр>,

IN <2 параметр> <тип 2 параметр>)

BEGIN

<тело процедуры>;

END //

DELIMITER ;

CALL <имя процедуры>(<1 параметр>, <2 параметр>);

Хранимые процедуры с выходными параметрами

Несмотря на отсутствие возвращаемых значений, получить результаты выполнения операция теле процедуры возможно

При вызове процедуры в качестве значения выходного параметра передается не конкретное значение, а пользовательская переменная. Если процедура в своем теле присвоит выходному параметру какое-либо значение, такое же значение примет и переменная, указанная при вызове процедуры. То есть фактически процедура запишет какое-то значение в ранее определенную переменную.

--создание процедуры

DELIMITER //

CREATE PROCEDURE <имя процедуры> (OUT <1 параметр> <тип 1 параметр>,

OUT <2 параметр> <тип 2 параметр>)

BEGIN

<тело процедуры>;

--запись переменных в заданные параметры

SET <1 параметр> := <значение вычислений 1>

SET <2 параметр> := <значение вычислений 2>

END //

DELIMITER ;

--определение параметров

SET @<любое имя для 1 параметр>:= NULL,

@< любое имя для 2 параметр>:= NULL;

--вызов процедуры с записью значений в параметры

CALL <имя процедуры>(<1 параметр>, <2 параметр>);

SELECT @<1 параметр>, @<2 параметр>

Хранимые процедуры поддерживают дополнительный вид параметров, определяемый с помощью ключевого слова INOUT. Параметр, перед именем которого указано такое ключевое слово, ведет себя как входной и выходной одновременно.

Входной-выходной параметр по функционалу похож на выходной параметр, поскольку в качестве его аргумента также указывается пользовательская переменная. Однако если начальное значение выходного параметра всегда равно NULL, то начальное значение входного-выходного параметра определяется значением пользовательской переменной, которое передается ему в качестве аргумента.

Пример\_1

DELIMITER //

CREATE PROCEDURE UPDATE\_ORDER\_STATUS(IN order\_id INT, OUT updated\_status TEXT)

BEGIN

UPDATE Orders

SET status = CASE

WHEN status = 'Created' THEN 'Shipped'

WHEN status = 'Shipped' THEN 'Delivered'

WHEN status = 'Delivered' THEN 'Completed'

END

WHERE id = order\_id;

SET updated\_status := (SELECT status FROM Orders WHERE id = order\_id);

END //

DELIMITER;

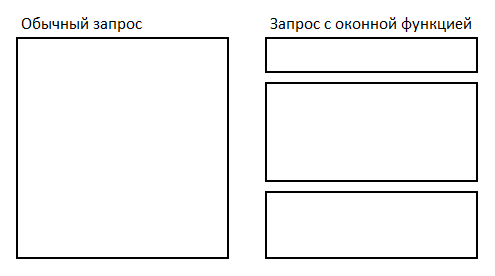
1. Оконные функции

Оконная функция — функция, которая работает с выделенным набором строк (окном, партицией) и выполняет вычисление для этого набора строк в отдельном столбце.

Набор записей, с которым работает оконная функция, называют окном.

При обычном запросе, все множество строк обрабатывается как бы единым «цельным куском», для которого считаются агрегаты. А при использовании оконных функций, запрос делится на части (окна) и уже для каждой из отдельных частей считаются свои агрегаты.

При использовании оконных функций количество строк в запросе не уменьшается по сравнении с исходной таблицей.



Определение окна выполняется с помощью оператора WINDOW, синтаксис использования которого имеет следующий вид:

Вид\_1:

SELECT <название поля 1>, <название поля 2>,

<оконная функция («какие-то поля таблицы»)> OVER <имя окна> AS <имя добавляемого столбца>

FROM <имя таблицы>

WINDOW < имя окна> AS (<спецификация окна>);

Вид\_2 (более наглядный так как отсутствует агрегат WINDOW):

SELECT <название поля 1>, <название поля 2>,

<оконная функция («какие-то поля таблицы»)> OVER (<спецификация окна>) AS <имя добавляемого столбца>

FROM <имя таблицы>;

С помощью оператора WINDOW можно определить как одно окно, так и несколько.

Определение окна происходит после выполнения операций:

JOIN, FROM, WHERE, GROUP BY, HAVING.

Определение окна в рамках запроса выполняется после группировки (или фильтрации групп) и до сортировки, поэтому оператор WINDOW в запросе располагается после оператора GROUP BY (или HAVING) и до оператора ORDER BY.

Спецификация окна, заключаемая в круглые скобки, может включать три базовых элемента:

* секционирование - PARTITION BY
* упорядочивание - ORDER BY
* определение границ окна - ROWS|RANGE

Если ни один из них не указан, то содержимое окна будет сформировано по умолчанию.

Разбиение окна на секции с помощью оператора PARTITION BY выполняется без учета регистра.

Оконные функции обычно подразделяют на три основные группы:

* ранжирующие функции — это функции, которые ранжируют значение для каждой строки в окне. Например, их можно использовать для того, чтобы присвоить порядковый номер строке или составить рейтинг
* агрегатные функции — это функции, которые выполняют на наборе данных арифметические вычисления и возвращают итоговое значение
* функции смещения — это функции, которые позволяют перемещаться и обращаться к разным строкам в окне, относительно текущей строки, а также обращаться к значениям в начале или в конце окна

Если подвести общий итог в работе оконных функций, то работа очень похожа на логику работы функции группировки GROUPE BY.

Однако вместо объединения и работы с уже сгруппированными полями, оконная функция:

1. разделяет все поля по заданным правилам после «OVER <спецификация окна>»
2. осуществляет какие-то действия в «<оконная функция («какие-то поля таблицы»)>»
3. добавляет новый столбец к существующим строкам под именем после «AS»

То есть в рамках работы исходная таблица не меняет количество строк, но прибавляет столбцы.

Ранжирующие функции

Функция **ROW\_NUMBER()** вычисляет порядковый номер записи в рамках указанного окна (начиная с 1).

SELECT ROW\_NUMBER () OVER (PARTITION BY director ORDER BY release\_year) AS num,

title, director, release\_year

FROM Films

ORDER BY director, num DESC;

1. Окно дробится на подокна с помощью спецификации PARTITION BY director
2. Записи в подокнах располагаются в порядке возрастания ORDER BY release\_year
3. К каждой записи добавляется столбец num с порядковым номером начиная с 1. В каждом подокне нумерация начинается заново.
4. Всё объединяется заново в одно окно
5. Общее окно располагается я в порядке возрастания director и убывания num

Функция **DENSE\_RANK()** вычисляет ранг записи в рамках указанного окна. Функция требует, чтобы окно, к которому она применяется, было упорядочено, поскольку значение поля (или нескольких полей), по которому было выполнено упорядочивание, будет использоваться для вычисления ранга записи. Непосредственно вычисление ранга выполняется следующим образом: функция DENSE\_RANK() присваивает первой записи окна ранг, равный 1, и увеличивает его каждый раз, когда очередная запись в поле (или полях) упорядочивания содержит значение, отличное от значения предыдущей записи в том же поле.

SELECT full\_name, department, salary,

DENSE\_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS place

FROM Employees;

Функция **NTILE()** используется для разбиения окна на заданное количество групп. Она принимает в качестве аргумента целое число k, разбивает окно на k групп и определяет номер группы, к которой относится запись.

SELECT id, full\_name,

NTILE(5) OVER (ORDER BY id) AS group\_number

FROM Employees;

Функция NTILE() всегда пытается разбить окно на группы равного размера, однако если окно из n записей делится на k групп, причем окно невозможно поделить на равные группы (n не делится на k нацело), то первые n % k (остаток от деления n на k) групп будут содержать ровно на 1 запись больше, чем остальные группы.

Например, если окно из 10 записей делится на 3 группы, то первая группа будет содержать 4 записи, а две остальные — по 3.

Агрегатные функции

Агрегатные оконные функции представляют собой те же самые агрегатные функции группировки, но применяемые не к группам, а к окнам. Использование агрегатных функций в контексте окон выгодно тем, что позволяет выполнять агрегатные вычисления на основе набора записей и при этом не терять какую-либо информацию о каждой записи из набора.

Функция **AVG()** вычисляет среднее арифметическое числовых значений, хранящихся в определенном поле окна. В качестве примера ее использования напишем запрос, который зарплате каждого сотрудника ставит в соответствие среднюю зарплату среди всех сотрудников.

Функция **SUM(**) вычисляет сумму числовых значений, хранящихся в определенном поле окна.

å

Функции AVG() и SUM() возвращают значение 0.0, если применяются к нечисловым полям.

Функция **COUNT()**, используемая для подсчета записей или непустых значений в поле, в контексте окон ведет себя ровно так же: если ее аргументом является звездочка (\*), функция вычисляет количество записей в окне, если название поля — количество непустых значений в этом поле окна.

Функция **MIN()** вычисляет минимальное значение, хранящиеся в определенном поле окна, функция **MAX()** — максимальное. При работе со строками функция MIN() считает минимальной ту строку, которая была бы первой, если бы эти строки были отсортированы в лексикографическом порядке. Функция MAX(), напротив, считает максимальной ту строку, которая была бы последней, если бы строки были отсортированы в лексикографическом порядке.

Если все значения поля, переданного в качестве аргумента в агрегатную оконную функцию, имеют значение NULL, возвращаемым значением функции также будет значение NULL.

Функции MIN(), MAX(), AVG(), SUM() и COUNT() игнорируют значения NULL при вычислениях.

Функции смещения

Функция **FIRST\_VALUE()** используется для получения значения, которое содержится в определенном поле первой записи окна.

Функция **LAST\_VALUE**() используется для получения значения, которое содержится в определенном поле последней записи окна.

Во время использования функции LAST\_VALUE() очень важно определить границы окна.

Если границы будут определены по умолчанию, оконная функция возвращает значение поля самой же записи, поскольку в таких рамках окна, в которых выполняется функция, текущая запись является последней.

Функция NTH\_VALUE() используется для получения значения, которое содержится в определенном поле n-ой записи окна начиная с 1.

Во время использования функции NTH\_VALUE(), как и в случае с функцией LAST\_VALUE(), важно следить за тем, как определены границы окна, в противном случае оконная функция может не дать нужного результата.

SELECT id, full\_name, salary,

NTH\_VALUE(salary, 2) OVER id\_asc AS second\_employee\_salary,

NTH\_VALUE(salary, 3) OVER id\_asc AS third\_employee\_salary

FROM Employees

WINDOW id\_asc AS (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING);

Функция NTH\_VALUE() возвращает значение NULL, если пытается получить значение записи, выходящей за рамки окна.

Функция **LAG()** используется для получения значения, которое содержится в определенном поле записи окна, отстающей от текущей на n.

Функция LAG() возвращает значение NULL, если пытается получить значение записи, выходящей за рамки окна.

В случае выхода за рамки окна функция LAG() умеет возвращать не только NULL, но и любое другое значение. Для этого при вызове функции достаточно указать необходимое значение в качестве третьего аргумента.

SELECT id, full\_name, salary,

LAG(salary, 1, 0) OVER (ORDER BY id) AS prev\_employee\_salary

FROM Employees;

Функция **LEAD()** используется для получения значения, которое содержится в определенном поле записи окна, опережающей текущую на n.

Функция LEAD(), может принимать третий аргумент, определяющий значение, которое будет возвращено при попытке получить значение записи, выходящей за рамки окна.

SELECT id, full\_name, salary,

LEAD(salary, 1, 0) OVER (ORDER BY id) AS next\_employee\_salary

FROM Employees;

Функции, LAG() и LEAD(), игнорируют установленные границы и выполняют свои вычисления в рамках всего окна (или секции окна, если было применено секционирование).

Функции LAG() и LEAD(), входящие в группу функций смещения, практически всегда используются только с упорядоченными окнами.

Если во время использования функции LAG()/ LEAD() опустить значение, определяющее то, на сколько запись отстает от текущей, оно по умолчанию примет значение 1.

*Определенные в окне границы учитывают только агрегатные функции, а также функции смещения FIRST\_VALUE(), LAST\_VALUE() и NTH\_VALUE(). Все остальные функции установленные границы игнорируют и выполняют свои вычисления в рамках всего окна (или секции окна, если было применено секционирование).*

Определение границ окна

С помощью данной спецификации можно детально обозначить, с какими именно записями окна должна взаимодействовать оконная функция. Определение границ окна выполняется путем указания двух граничных точек: начальной и конечной. При указанных границах оконная функция будет работать только с теми записями окна (или секции окна, если было применено секционирование), которые заключены между граничными, включая сами границы.

Границы всегда определяются относительно той записи, с которой на данный момент взаимодействует оконная функция. Поэтому, например, начальной границей может являться предыдущая запись относительно текущей, а конечной — следующая. Таким образом, для каждой записи, с которой взаимодействует оконная функция, границы определяются индивидуально.

Часть окна, получаемая в результате применения установленных границ, никогда не выходит за пределы окна, а если сталкивается с ними, то обрезается.

Часть окна, получаемая в результате применения установленных границ, называется фреймом.

Определить границы окна можно с помощью оператора **ROWS**:

ROWS BETWEEN <начальная граничная точка> AND <конечная граничная точка>

Начальная граничная точка может быть представлена одним из следующих значений:

* CURRENT ROW — текущая запись
* n PRECEDING — n-ая запись перед текущей
* n FOLLOWING — n-ая запись после текущей
* UNBOUNDED PRECEDING — самая первая запись окна (или секции окна, если было применено секционирование)

Конечная граничная точка может быть представлена одним из следующих значений:

* CURRENT ROW — текущая запись
* n PRECEDING — n-ая запись перед текущей
* n FOLLOWING — n-ая запись после текущей
* UNBOUNDED FOLLOWING — самая последняя запись окна (или секции окна, если было применено секционирование)

Оператор ROWS в определении окна должен располагаться после операторов PARTITION BY и ORDER BY.

Определенные в окне границы учитывают только агрегатные функции, а также некоторые функции смещения. Все остальные функции установленные границы игнорируют и выполняют свои вычисления в рамках всего окна (или секции окна, если было применено секционирование).

Если часть окна, полученная в результате применения установленных границ, не содержит ни одной записи, то результатом вызова оконной функции относительно данной части окна будет значение NULL.

Определить границы окна также можно с помощью оператора **RANGE**:

RANGE BETWEEN <начальная граничная точка> AND <конечная граничная точка>

Для указания граничных точек используются те же ключевые слова CURRENT ROW, PRECEDING и FOLLOWING.

Принципы работы операторов RANGE и ROWS значительно отличаются. Оператор RANGE, в отличие от ROWS, ориентируется не на расположение записей (например, предыдущая или следующая), а на их значение в том поле, по которому было упорядочено окно.

Оператор RANGE совместно с ключевыми словами PRECEDING и FOLLOWING позволяет определять не столько границы, сколько диапазоны.

Если некоторое окно упорядочено по условному полю field, а текущая запись в поле field содержит значение x, тогда определение границ следующим образом:

… OVER (ORDER BY x RANGE BETWEEN a PRECEDING AND b FOLLOWING)

будет говорить о том, что в ограниченную часть окна должны быть включены те записи, которые в поле field содержат значение, входящее в диапазон [x - a; x + b]. То есть фактически ограничение окна идет не по номеру записи, а по значению в определенном поле.

При использовании оператора RANGE связка ключевых слов CURRENT ROW обозначает не текущую запись, а набор записей, которые в поле упорядочивания содержат то же значение, что и текущая запись. Другими словами, CURRENT ROW в контексте оператора RANGE можно трактовать как 0 PRECEDING или 0 FOLLOWING.

Окно, границы которого определяются с помощью оператора RANGE, должно быть упорядочено только по одному полю.

Поле, которое используется оператором RANGE для определения границ, должно содержать только числа, даты, временные значения или значения даты и времени.

Окно всегда имеет границы, определенные по умолчанию, которые зависят от того, упорядочено окно или нет.

Если окно не упорядочено, границы определяются следующим образом:

RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

Если упорядочено — следующим образом:

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

1. Регулярные выражения

Регулярные выражения (regular expression, regex, регулярка) – это способ обработки текста, позволяющий находить в строках подстроки по сложным условиям. Формально, регулярное выражение — это строка, задающая шаблон поиска подстрок в тексте. Термин регулярные выражения является переводом английского словосочетания regular expressions. Перевод не очень точно отражает смысл, правильнее было бы шаблонные выражения.

Метасимволы

\d — задает любую цифру

\D — задает любой символ НЕ цифру

. — соответствует одному любому символу

\ — символ экранирования

\n — символ переноса строки

\s — пробельный символ (пробел, табуляция, конец строки)

\S — НЕ пробельный символ

\w — символ, соответствующий буквам (в верхнем и нижнем регистре), цифрам и символу подчеркивания \_

\W — символ, соответствующий НЕ буквам, НЕ цифрам и НЕ символу подчеркивания \_

\b — метасимвол соответствует началу/концу (границе) слова

\B — метасимвол соответствует НЕ началу/ НЕ концу (НЕ границе) слова

^ — метасимвол соответствует началу строки

$ — метасимвол соответствует концу строки

[A-Za-z0-9] — символы, определяющие наборы от A до Z, от a до z, от 0 до 9

[^0-9] — символы, определяющие наборы всех символов КРОМЕ цифр от 0 до 9

| — символ перечисления всех необходимых шаблонов (ИЛИ)

+ — устанавливает соответствие с одним или более символами

\* — устанавливает соответствие с нулем или более символами

? — устанавливает соответствие с нулем символов или одному символу

{} — метасимволы определяющие интервалы

() — метасимволы определяющие подвыражения

*Под границей слова подразумевается пустое пространство между двумя символами, одним из которых является буква, цифра или символ подчеркивания (\w), а вторым — символ, который НЕ является буквой, цифрой или символом подчеркивания (\W). Символ \b в основном используется для определения начала и конца слова.*

Метасимволы \* и + являются жадными, то есть они ищут наибольшее возможное соответствие. Если нам не нужно жадное поведение, то необходимо использовать ленивые аналоги этих метасимволов (они называются ленивыми, потому что устанавливают соответствие с наименьшим, а не с наибольшим возможным количеством символов).

Ленивые метасимволы определяются путем добавления символа ? к используемому метасимволу, причем для каждого жадного метасимвола имеется ленивый эквивалент.

|  |  |
| --- | --- |
| **Жадный** | **Ленивый** |
| \* | \*? |
| + | +? |
| ? | ? ? |
| {m,n} | {m,n}? |
| {,n} | {,n}? |
| {m,} | {m,}? |

*Буква ё не входит в диапазон [a-я], а буква Ё — в диапазон [A-Я].*

*В пределах квадратных скобок перед символы можно не экранировать косой чертой \.*

Интервалы

Если требуется найти точное количество совпадений одного символа или количество совпадений в определенном диапазоне, то эти метасимволы окажутся бесполезны. Чтобы обеспечить большую свободу управления повторением совпадений, в регулярных выражениях имеется возможность использования интервалов. Интервалы определяются между символами {и}.

Например:

\d{3} = \d\d\d

[0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]= [0-9A-Fa-f]{4}

Также интервалами можно задавать диапазон максимального и минимального значения

Например:

\d{2,4} = \d\d|\d\d\d|\d\d\d\d

В случае указания одной цифры и запятой, можно получить выражение «не менее» или «не более»

Например:

\d{2,} = \d\d|\d\d\d|\d\d\d\d|\d\d\d\d\d… и до бесконечности

\d{,3} = \d|\d\d|\d\d\d

*Интервалы могут начинаться с нуля.*

Подвыражения

Подвыражения — это части большого выражения, которые группируются так, чтобы они обрабатывались как единый объект. Подвыражения заключаются между символами ( и ).

Значение в подвыражение интерпретируется как единый объект и может быть использовано как в качестве ссылок назад. **Ссылки назад** позволяют в регулярном выражении обратиться к предыдущим совпадениям (в данном случае к ранее найденному слову).

Например в выражение (\b\w+\b)[ ]+\1 подвыражение (\b\w+\b) соответствует словам, состоящим из одного или нескольких алфавитно-цифровых символов, а выражение [ ]+ соответствует любому количеству пробелов в конце этих слов. Подвыражение (\b\w+\b) используется для того, чтобы сгруппировать символы в одно выражение, пометить его и идентифицировать для использования в дальнейшем. Заключительная часть этого регулярного выражения (\1) — это и есть ссылка назад на подвыражение, и всякий раз, когда подвыражению (\b\w+\b) соответствует слово, точно такому же слову соответствует и \1.

Функции для работы с регулярными выражениями

*Регулярные выражения являются* ***регистрозависимыми****, но в SQL регулярные выражения по умолчанию являются****регистроНЕзависимыми****.*

Функция **REGEXP\_LIKE()** используется для того, чтобы узнать, есть ли совпадение с шаблоном регулярного выражения в строке. Принимает три аргумента в следующем порядке:

* str — исходная строка
* pattern — шаблон регулярного выражения
* flags — флаги сопоставления (может не указываться)

Функция возвращает 1, если в строке str есть совпадение с шаблоном регулярного выражения pattern, или 0 в противном случае.

Необязательный аргумент flags изменяет поведение синтаксического анализа регулярных выражений, позволяя еще больше уточнить сопоставление с образцом. Функциональность регулярных выражений расширяется за счет флагов.

* Если требуется учитывать регистр, необходимо использовать флаг «c». **Consider (учитывать)**
* Флаг «i» используется по умолчанию и служит для игнорирования регистра. **Ignore (игнорировать)**
* Чтобы правильным образом обрабатывать символы \n, тем самым находя начало и конец каждой новой строки с учетом символа \n, необходимо использовать флаг «m». **Multiline (многострочный)**
* Метасимволу «.» соответствует любой символ кроме символа \n. Для того, чтобы учитывался и этот символ, необходимо использовать флаг «n». То есть если есть это флаг, то под символом «.» также будет учитываться и символ переноса строки. **Newline (новая строка)**

Флаги сопоставления можно комбинировать.

Функцию REGEXP\_LIKE() можно использовать совместно с оператором NOT.

Для поиска символа точка «.» соответствующий метасимвол должен быть экранирован двойным символом «\\», а не одинарным «\.» Это связано с тем, что аргумент pattern — это обычная строка SQL, и перед тем, как интерпретировать ее как регулярное выражение, СУБД выполняет преобразование. Символ «\» в строках SQL используется для экранирования, поэтому в ходе этого преобразования последовательность «\.» становится обычной точкой «.», которая в регулярных выражениях обозначает любой символ. Чтобы это избежать, необходимо экранировать сам символ «\.»

SELECT REGEXP\_LIKE('Foo\nbar.', '^[A-Z].\*\\.$') AS result1,

REGEXP\_LIKE('Foo\nbar.', '^[A-Z].\*\\.$', 'm') AS result2,

REGEXP\_LIKE('Foo\nbar.', '^[A-Z].\*\\.$', 'cm') AS result3,

REGEXP\_LIKE('Foo\nbar.', '^[A-Z].\*\\.$', 'mc') AS result4;

Функция **REGEXP\_INSTR()** используется для определения местоположения подстроки, которая совпадает с шаблоном регулярного выражения. Она принимает шесть аргументов в следующем порядке:

* str — исходная строка
* pattern — шаблон регулярного выражения
* start — позиция начала поиска (может не указываться, в таком случае поиск выполняется с начала строки)
* occurence — номер совпадения (может не указываться, в таком случае осуществляется поиск первого совпадения)
* return\_option — тип возвращаемой позиции (может не указываться, в таком случае будет возвращена позиция совпадения)
* flags — флаги сопоставления (может не указываться)

Функция выполняет поиск подстрок, которые совпадают с шаблоном регулярного выражения pattern в строке str, начиная с позиции start, и возвращает позицию первой подстроки.

При использовании функции REGEXP\_INSTR() символы исходной строки нумеруются с 1, а не с 0.

Если в строке нет подстрок, которые совпадают с шаблоном регулярного выражения, функция REGEXP\_INSTR() вернет значение 0.

Аргумент occurence определяет, позицию какой по счету подстроки необходимо вернуть. Если значение аргумента больше количества подстрок, функция вернет значение 0.

Если необходимо изменить значение только одного из необязательных аргументов, все равно нужно указывать все предыдущие необязательные аргументы, передав им в качестве значений их значения по умолчанию.

Аргумент return\_option определяет, какую позицию должна вернуть функция. Его значением по умолчанию является 0. В этом случае функция вернет позицию найденной подстроки. Если указать значение 1, будет возвращена позиция символа, стоящего после найденной подстроки.

SELECT REGEXP\_INSTR('beegeek', 'e{3}') AS result1,

REGEXP\_INSTR('beegeek', '^geek') AS result2,

REGEXP\_INSTR('beegeek', 'bee$') AS result3;

SELECT REGEXP\_INSTR('beegeek', 'e', 1, 1) AS result1, -- позиция первой подстроки, поведение по умолчанию

REGEXP\_INSTR('beegeek', 'e', 1, 2) AS result2, -- позиция второй подстроки

REGEXP\_INSTR('beegeek', 'e', 1, 3) AS result3, -- позиция третьей подстроки

REGEXP\_INSTR('beegeek', 'e', 1, 4) AS result4, -- позиция четвертой подстроки

REGEXP\_INSTR('beegeek', 'e', 1, 5) AS result5; -- позиция пятой подстроки

SELECT REGEXP\_INSTR('beegeek', 'e{2}', 1, 1, 1) AS result1, -- пример 1

REGEXP\_INSTR('beegeek', 'e{2}', 1, 2, 1) AS result2; -- пример 2

В первом примере осуществляется поиск подстрок, которые совпадают с шаблоном регулярного выражения e{2}, то есть с двумя подряд идущим символами e. Поиск осуществляется с начала строки (третий аргумент start равен 1), при этом происходит поиск первой подстроки (четвертый аргумент occurence равен 1). Пятый аргумент return\_option равен 1, поэтому осуществляется поиск позиции символа, стоящего после найденной подстроки. Этим символом является g, позиция которого равна 4, поэтому возвращаемым значением функции также является 4.

Во втором примере также осуществляется поиск позиции символа, стоящего после подстроки, которая совпадает с шаблоном регулярного выражения, но только в этот раз после второй подстроки, так как четвертый аргумент occurence равен 2. Этим символом является k, позиция которого равна 7, поэтому возвращаемым значением функции также является 7.

Аргумент flags используется для указания флагов сопоставления.

Функция **REGEXP\_SUBSTR()** используется для извлечения подстроки, которая совпадает с шаблоном регулярного выражения. Она принимает пять аргументов в следующем порядке:

* str — исходная строка
* pattern — шаблон регулярного выражения
* start — позиция начала поиска (может не указываться, в таком случае поиск выполняется с начала строки)
* occurence — номер совпадения (может не указываться, в таком случае осуществляется поиск первого совпадения)
* flags — флаги сопоставления (может не указываться)

Функция извлекает из строки str первую подстроку, которая совпадает с шаблоном регулярного выражения pattern, начиная с позиции start, и возвращает полученный результат.

При использовании функции REGEXP\_SUBSTR() символы исходной строки нумеруются с 1, а не с 0.

Если в строке нет подстрок, которые совпадают с шаблоном регулярного выражения, функция REGEXP\_SUBSTR() вернет значение NULL.

Аргумент occurence определяет, какую по счету подстроку необходимо вернуть. Если значение аргумента больше количества подстрок, функция вернет значение NULL.

Аргумент flags используется для указания флагов сопоставления.

SELECT REGEXP\_SUBSTR('beegeeek', 'e{2,}', 1, 1) AS result1, -- первая подстрока, поведение по умолчанию

REGEXP\_SUBSTR('beegeeek', 'e{2,}', 1, 2) AS result2, -- вторая подстрока

REGEXP\_SUBSTR('beegeeek', 'e{2,}', 1, 3) AS result3; -- третья подстрока

Функция **REGEXP\_REPLACE()** используется для замены подстроки, которая совпадает с шаблоном регулярного выражения. Она принимает шесть аргументов в следующем порядке:

* str — исходная строка
* pattern — шаблон регулярного выражения
* to\_str — строка замены
* start — позиция начала поиска (может не указываться, в таком случае поиск выполняется с начала строки)
* occurence — номер совпадения (может не указываться, в таком случае осуществляется поиск всех совпадений)
* flags — флаги сопоставления (может не указываться)

Функция заменяет в строке str все подстроки, которые совпадают с шаблоном регулярного выражения pattern, на подстроку to\_str, начиная с позиции start, и возвращает полученный результат.

При использовании функции REGEXP\_REPLACE() символы исходной строки нумеруются с 1, а не с 0

Если в строке нет ни одной подходящей подстроки, функция вернет исходную строку.

Аргумент occurence определяет, какую по счету подстроку необходимо заменить. Если значение аргумента больше количества подстрок, функция вернет исходную строку.

Строка замены — это обычная строка, а не регулярное выражение, поэтому она будет записана в том виде, в котором указана.

Аргумент flags используется для указания флагов сопоставления.

SELECT REGEXP\_REPLACE('beeegeeek', 'e{3,}', 'ee', 1, 0) AS result1, -- все подстроки, поведение по умолчанию

REGEXP\_REPLACE('beeegeeek', 'e{3,}', 'ee', 1, 1) AS result2, -- первая подстрока

REGEXP\_REPLACE('beeegeeek', 'e{3,}', 'ee', 1, 2) AS result3, -- вторая подстрока

REGEXP\_REPLACE('beeegeeek', 'e{3,}', 'ee', 1, 3) AS result3; -- третья подстрока