**✅ База данных** — это организованная и определенным образом структурированная коллекция данных.

**✅ Данные** — это любая собранная и сохраненная информация об отдельной сущности (человеке, месте, предмете или любом другом объекте реального мира). К данным также относятся и атрибуты (свойства) этой сущности.

**✅ Возможности** баз данных:

* **Хранение и организация больших объемов данных.** Базы данных позволяют хранить огромное количество информации, обеспечивая эффективное использование ресурсов хранения.
* **Систематизация и структурирование данных.** Базы данных позволяют организовывать данные в согласованную структуру, определяя типы хранимых данных, связи между ними и различные правила целостности.
* **Обеспечение доступа к данным.** Базы данных поддерживают специальные языки запросов для быстрого и удобного доступа к данным.

**✅**На сегодняшний день существует огромное количество **разновидностей баз данных**, различающихся по множеству критериев, например:

* **Иерархическая модель** баз данных. В ней данные представляются в виде древовидной структуры, где каждая запись имеет родительскую запись, кроме верхнего уровня.
* **Сетевая модель** баз данных. В этой модели данные представляются в виде сети, где каждая запись может быть связана с несколькими другими записями.
* **Реляционная модель**баз данных**.**Реляционные базы данных организованы в виде набора связанных таблиц.
* **Модель** **ключ —** **значение**(хеш-таблица)**.**В базах данных этого типа данные хранятся в виде пар ключ — значение.
* **Документная** или **документоориентированная модель.**Основная идея этой модели заключается в хранении данных в виде **документов**, которые обычно представлены файлами в форматах вроде JSON или XML.

**✅ Реляционная модель** представляет собой совокупность данных, состоящую из набора **таблиц**. В теории множеств, которая легла в основу этой модели, таблице соответствует термин **отношение** (relation), а таблица является его физическим представлением. Отсюда и название модели — реляционная. Она была разработана в 1970 году **Эдгаром Коддом**, специалистом в области математики и информатики.

**✅ Любая таблица** в реляционной базе данных состоит из строк, которые называют **записями**, и столбцов, которые называют **полями**. На пересечении строк и столбцов находятся конкретные значения. Запись представляет собой набор значений, содержащий информацию о конкретной сущности, полем же является отдельная характеристика сущности.

**✅ Первичный ключ** — это такое поле, значения которого в таблице не повторяются. Первичный ключ, состоящий из одного поля, обычно называют **простым**. Если же первичный ключ включает несколько полей, его называют **составным**.

**✅ Таблица** может содержать информацию, используемую для связи с другой таблицей. Осуществляется такая навигация с помощью **внешних ключей**.

**✅ Таблица**, которая ссылается на данные из другой таблицы, называется **дочерней**. Таблица, на данные которой ссылается другая таблица, называется **родительской**.

**✅ Нормализация** — это процесс организации данных в реляционной базе данных в соответствии с определенными правилами, которые делают базу данных более гибкой, устраняя избыточность данных и потенциальные ошибки.

**✅ Ошибки**, которые могут возникнуть при работе с ненормализованной базой данных, называются **аномалиями**. Выделяют аномалии вставки, удаления и модификации, которые возникают при соответствующих операциях.

**✅**Существует несколько **наборов правил нормализации** баз данных. Каждый такой набор называется **нормальной формой**. Выделяют пять основных нормальных форм: **первая, вторая, третья, четвертая** и **пятая** нормальные формы. Иногда выделяют еще три дополнительные формы: шестая нормальная форма, нормальная форма Бойса — Кодда, доменно-ключевая нормальная форма.

**✅ Приведение базы данных** **к** какой-либо конкретной **нормальной форме** обязательно требует, чтобы эта база данных уже находилась **в предыдущей нормальной форме**.

**✅**База данных считается**нормализованной**, если она находится как минимум **в третьей нормальной форме**.

**✅ Первая нормальная форма** является начальным уровнем нормализации данных. Она включает следующие правила:

* в таблице не должно быть полей с одинаковым смыслом
* в каждой ячейке таблицы должно находиться одиночное несоставное значение
* таблица должна иметь первичный ключ

**✅ Вторая нормальная форма** является продолжением первой нормальной формы и включает следующие правила:

* таблица должна находиться в первой нормальной форме
* каждое неключевое поле таблицы должно зависеть только от полного первичного ключа, то есть от всех его полей

**✅ Третья нормальная форма** исправляет оставшиеся после приведения ко второй нормальной форме аномалии и включает следующие правила:

* таблица должна находиться во второй нормальной форме
* каждое неключевое поле таблицы должно зависеть только от первичного ключа и ни от какого другого неключевого поля

**✅ База данных** — это организованная и определенным образом структурированная коллекция данных.

**✅ СУБД (система управления базами данных)** — это комплексное программное обеспечение, которое служит интерфейсом между базой данных и разработчиком, конечным пользователем или программой, предоставляя им возможность удобно взаимодействовать с базой данных и выполнять с ней все необходимые действия.

**✅**На сегодняшний день существует немалое количество различных СУБД, каждая из них обладает рядом отличительных характеристик. Обычно **СУБД классифицируют по следующим признакам**:

* по расположению базы данных — **локальные** и **распределенные**
* по способу хранения и обработки данных — **клиент-серверные, файл-серверные** и **встраиваемые**
* по модели хранения данных — **иерархические**, **сетевые**, **реляционные** и так далее
* по используемому языку запросов — **SQL-ориентированные** и **NoSQL-ориентированные**

**✅ Реляционные СУБД** — одни из самых распространенных систем. Для управления данными они используют язык программирования под названием **SQL (Structured Query Language, язык структурированных запросов)**, который был специально разработан для работы с базами данных, принадлежащими реляционной модели.

**✅ Популярные реляционные СУБД:**

* **PostgreSQL** — клиент-серверная реляционная СУБД, которая подходит для масштабных проектов с большими массивами данных и высокой нагрузкой
* **MySQL** — клиент-серверная реляционная СУБД, которая хорошо подходит для средних и небольших проектов и команд
* **SQLite** — компактная СУБД, которая подходит для небольших проектов

**✅ Язык SQL** представляет собой набор операторов, которые делятся на определенные группы, и у каждой группы есть свое назначение. В сокращенном виде эти группы называются **DDL, DML, DCL** и **TCL.**

**✅ Data Definition Language (DDL)** – это группа операторов определения данных. С помощью операторов, входящих в эту группу, мы определяем структуру базы данных и работаем с объектами этой базы, то есть создаем, изменяем и удаляем их.

**✅ Data Manipulation Language (DML)** – это группа операторов для манипуляции данными. С помощью этих операторов мы можем добавлять, изменять, удалять и выгружать данные из базы, то есть манипулировать ими.

**✅ Data Control Language (DCL)** – группа операторов определения доступа к данным. С их помощью мы можем разрешать или запрещать выполнение определенных операций над объектами базы данных.

**✅ Transaction Control Language (TCL)** – группа операторов для управления транзакциями. Транзакция – это команда или блок команд, которые выполняются как единое целое. Если транзакция выполнена успешно, все модификации данных, сделанные в течение транзакции, принимаются и становятся постоянной частью базы данных. Если в результате выполнения транзакции происходит ошибка, все выполненные модификации данных будут отменены.

**✅ Строковые**типы данных**:**

* CHAR(n),где n – число в диапазоне [0; 255], представляющее допустимую длину строки.Хранит строку **фиксированной**длины, которая задается на этапе создания таблицы. Если строка имеет длину меньше, чем n, то оставшиеся символы заполняются пробелами.
* VARCHAR(n),где n – число в диапазоне [0; 65535], представляющее допустимую длину строки.Хранит строку **переменной** длины, которая также задается на этапе создания таблицы. В отличие от CHAR, строка не заполняется пробелами, если ее длина меньше, чем n.

Типы без указания максимальной длины (ограничены типом):

|  |  |  |
| --- | --- | --- |
| **Тип** | **Размер (в байтах)** | **Максимальное количество символов** |
| TINYTEXT | 255 | 255 |
| TEXT | 65535 | 2^16 - 1 |
| MEDIUMTEXT | 16777215 | 2^24 - 1 |
| LONGTEXT | 4294967295 | 2^32 - 1 |

**✅ Числовые**типы данных**:**

* **Целые числа:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Размер (в байтах)** | **Диапазон signed** | **Диапазон unsigned** |
| TINYINT | 1 | [-128; 127] | [0; 255] |
| SMALLINT | 2 | [-2^15; 2^15 - 1] | [0; 2^16 - 1] |
| MEDIUMINT | 3 | [-2^23; 2^23 - 1] | [0; 2^24 - 1] |
| INT | 4 | [-2^31; 2^31 - 1] | [0; 2^32 - 1] |
| BIGINT | 8 | [-2^63; 2^63 - 1] | [0; 2^64 - 1] |

* **Числа с плавающей точкой:**

|  |  |  |
| --- | --- | --- |
| **Тип** | **Размер (в байтах)** | **Диапазон** |
| FLOAT | 4 | [1.17 \* 10^-38; 3.4 \* 10^38] |
| DOUBLE | 8 | [2.2 \* 10^-308; 1.7 \* 10^308] |

* **Числа с фиксированной точкой:**  
  DECIMAL(M, D) и NUMERIC(M, D), где M – общее количество цифр (ограничено диапазоном [1; 65]), D – количество цифр после запятой.

**✅ Типы данных даты и времени:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Формат** | **Диапазон значений** | **Размер (в байтах)** |
| DATE | YYYY-MM-DD | 1000-01-01 – 9999-12-31 | 3 |
| TIME | HH:MM:SS | 00:00:00 – 23:59:59 | 3 |
| DATETIME | YYYY-MM-DD HH:MM:SS | 1000-01-01 00:00:00 – 9999-12-31 23:59:59 | 8 |
| TIMESTAMP | YYYY-MM-DD HH:MM:SS | 1970-01-01 00:00:00 – 2038-01-19 03:14:07 | 4 |
| YEAR | YYYY | 1901 – 2155 | 1 |

**✅ Тип данных NULL** — это специальное значение, которое используется в MySQL для обозначения отсутствия данных. Оно отличается от пустой строки или нулевого значения, так как в действительности означает полное отсутствие данных, в то время как ноль является допустимым числовым значением, а пустая строка — допустимой строкой нулевой длины.

**✅ В разных СУБД** могут использоваться **разные псевдонимы для типов данных**. Например, в MySQL тип INT также может использоваться как INTEGER, а в SQLite — только как INTEGER, но не как INT.

**✅ SQL-запрос**представляет собой команду, которую мы отправляем базе данных для выполнения определенной задачи, целью которой может являться создание, изменение или извлечение данных. Каждый запрос состоит из **операторов**, определяющих необходимые действия, и различных выражений, например, логических или арифметических.

**✅ Операторы** также называют **ключевыми словами**. Они представляют собой зарезервированные слова, которые имеют специальное значение и выполняют определенные функции в синтаксисе SQL. Операторы не могут использоваться в качестве имен таблиц, полей или других объектов базы данных.

**✅ SQL-запросы** могут быть сложными и состоять **из нескольких операций**, и для отделения их друг от друга используется символ ;.

**✅ Операторы** нечувствительны к регистру.

**✅ SELECT** — ключевое слово для извлечения данных из таблицы. Имя извлекаемого поля указывается сразу после ключевого слова SELECT. Для извлечения нескольких полей таблицы после оператора SELECT необходимо через запятую перечислить их имена. Для извлечения всех полей следует указать специальный групповой символ \*.

**✅ FROM** — ключевое слово для указания таблицы, из которой извлекаются данные.

**✅ DISTINCT** — ключевое слово для извлечения только уникальных записей. Указывается непосредственно перед именами полей.

**✅ LIMIT** — ключевое слово для ограничения количества извлекаемых записей.

**✅ OFFSET** — ключевое слово для задания начальной точки извлечения. Можно использовать только в паре с ключевым словом **LIMIT**. Выражение LIMIT n OFFSET m можно записать в более коротком виде: LIMIT m, n.

**✅ AS** — ключевое слово для создания **псевдонима**извлекаемого поля. Располагается между исходным именем поля и новым. Псевдонимы могут содержать до 255 знаков, включая пробелы, цифры и специальные символы.

**✅ SQL** поддерживает несколько видов комментариев:

* **однострочные** комментарии обозначаются двумя дефисами --
* **многострочные** комментарии обрамляются сочетанием символов /\*

**✅ ORDER BY** — оператор, отвечающий за сортировку информации при ее извлечении из базы данных. Достаточно дополнить запрос оператором ORDER BY и указать название поля, по которому необходимо осуществить сортировку. В запросе оператор ORDER BY должен следовать после операторов SELECT и FROM. По умолчанию оператор ORDER BY выполняет сортировку по возрастанию.

**✅ Поля, по которым выполняется сортировка**, необязательно должны быть извлечены, то есть сортировка может выполняться и по тем полям, которые не попадают в результирующую таблицу.

**✅ Сортировка** **может выполняться по нескольким полям**, указанным через запятую. Сначала сравниваются значения первого поля, затем — второго и так далее.

**✅ Вместо имен** **полей** оператор ORDER BY позволяет указывать **порядковые номера полей**, по которым необходимо выполнить сортировку. Однако сортировка таким методом возможна только по тем полям, которые присутствуют в результирующей таблице. Нумеруются поля с единицы: первое поле результирующей таблицы имеет порядковый номер 1, второе — 2 и так далее.

**✅ Сортировать данные** **можно** не только по возрастанию, но и **по убыванию**. Чтобы сортировка выполнялась в порядке убывания, необходимо после имени поля указать ключевое слово **DESC**. Если необходимо отсортировать записи в порядке убывания значений нескольких полей, нужно обязательно указать ключевое слово DESC после каждого из них.

**✅ Для сортировки** **в порядке возрастания** предусмотрено ключевое слово **ASC**, использовать которое необязательно.

**✅ Операторы ORDER BY и LIMIT** могут использоваться вместе для решения различных задач.

**✅ Сортировка** **является одной из завершающих операций**и выполняется уже после формирования результирующей таблицы, поэтому в блоке оператора ORDER BY можно указывать как фактические имена полей таблицы, из которой извлекаются данные, так и их псевдонимы. Если псевдоним состоит из нескольких слов, то при сортировке данных по этому псевдониму необходимо заключать его в обратные апострофы (``).

**✅ При сортировке** **строковых значений** регистр не учитывается. Данное поведение может быть изменено в настройках СУБД, однако выбирать, учитывать регистр или нет, с помощью оператора ORDER BY нельзя.

**✅ WHERE** — оператор, позволяющий указывать определенные условия при извлечении записей. Если запись удовлетворяет этим условиям, то она попадает в результирующую таблицу, в противном случае отбрасывается. Оператор WHERE указывается после названия таблицы.

**✅ Оператор WHERE** может использоваться совместно с оператором ORDER BY, в таком случае оператор ORDER BY должен располагаться после оператора WHERE.

**✅**В SQL поддерживается целый **набор операторов сравнения**, которые перечислены в таблице ниже:

|  |  |
| --- | --- |
| **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 |

**✅ Операторы != и <>** обычно взаимозаменяемы, однако не во всех СУБД поддерживаются обе формы оператора.

**✅ При использовании** **оператора BETWEEN** нужно указывать два значения: нижнюю и верхнюю границы диапазона. Оба значения должны быть разделены ключевым словом AND. При этом извлекаются все значения из диапазона, включая те, что равны граничным значениям.

**✅ Псевдонимы полей** не могут быть использованы в блоке оператора WHERE для фильтрации записей. Это связано с тем, что блок WHERE обрабатывается до блока SELECT, поэтому на момент выполнения блока WHERE псевдонимы полей еще не известны.

**✅ Результатом**операции сравнения могут являться следующие значения:

* 1, что эквивалентно истине
* 0, что эквивалентно лжи
* NULL, что эквивалентно неопределенности

**✅**По умолчанию операторы сравнения **не учитывают регистр** при сравнении строковых значений.

**✅ Оператор  LIMIT** при его использовании совместно с оператором WHERE  или сочетанием операторов WHERE и ORDER BY должен располагаться после них. Это связано с порядком, в котором запрос выполняет свои операции: сначала запрос извлекает из таблицы записи, удовлетворяющие заданным условиям, затем сортирует и только после этого ограничивает определенным количеством.

**✅ Фильтрация данных** при их извлечении не ограничивается лишь одним критерием. В SQL, как и во многих других языках программирования, можно использовать несколько условий, чтобы создавать более сложные фильтры. Для этого предназначены логические операторы **AND** и **OR**.

**✅ Оператор AND**используется для фильтрации данных, удовлетворяющих **всем** условиям.

**✅ Оператор OR** используется для фильтрации данных, удовлетворяющих**хотя бы одному** условию.

**✅ В большинстве СУБД** при использовании оператора OR второе условие не рассматривается, если выполняется первое.

**✅ После оператора WHERE** может содержаться любое количество логических операторов AND и OR.

**✅**При использовании **логических** **операторов AND и OR** рекомендуется всегда ставить скобки, чтобы точно сгруппировать условия.

**✅ Оператор IN** позволяет определить, совпадает ли значение поля с одним из перечисленных значений.

**✅ Оператор NOT** служит только одной цели — отрицать условие, следующее за ним.

**✅ Оператор NOT IN** используется для определения несоответствия списку значений.

**✅ 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 |

Если условие содержит несколько логических операторов с одинаковым приоритетом, они выполняются слева направо.

**✅ Логический оператор NOT** отрицает только то условие, перед которым указан.

**✅ Метасимволы**— это специальные знаки, которые позволяют создавать **шаблоны поиска** и находить значения, соответствующие не чему-либо конкретному, а именно этим шаблонам.

**✅**Чтобы использовать метасимволы **в условиях отбора записей**, необходимо задействовать **оператор** **LIKE**, который говорит о том, что следующий шаблон поиска необходимо анализировать с учетом метасимволов, а не искать точные совпадения.

**✅ Шаблоны поиска** представляют собой строки, состоящие из обычных символов, метасимволов или любой их комбинации, поэтому поиск с использованием метасимволов может осуществляться только в строковых полях.

**✅ Метасимвол %** в шаблоне поиска соответствует последовательности любых символов, причем число символов в последовательности может быть от 0 и более.

**✅ Метасимвол \_** в шаблоне поиска соответствует одному любому символу.

**✅**Метасимволы могут встречаться **в любом месте шаблона поиска**, причем в неограниченном количестве.

**✅ Важной особенностью** **оператора LIKE** является то, что при поиске с его помощью строк, соответствующих шаблону, не учитывается регистр символов, используемых в этом шаблоне. Когда регистр символов важен, шаблон поиска необходимо преобразовать в тип данных BINARY с помощью функций CAST() или CONVERT().

**✅**Чтобы отыскать **совпадение для самого метасимвола**, необходимо использовать экранирование с помощью ключевого слова **ESCAPE**, которое указывается после шаблона поиска и определяет символ, отвечающий за экранирование.

**✅ Метасимвол %** может соответствовать чему угодно, но есть одно исключение: NULL. Никакой шаблон поиска не позволит отобрать запись, проверяемое поле которой содержит значение NULL.

**✅ Вычисляемые поля** — поля, значения которых не хранятся в таблице базы данных, а вычисляются автоматически на основе данных из других полей таблицы. Вычисляемые поля используются для извлечения информации с дополнительными преобразованиями, подсчетами или форматированием.

**✅** **Вычисляемые поля** определяются с использованием выражений, которые могут включать математические операции и вызовы различных функций.

**✅ Функция** — программная единица, предназначенная для выполнения различных операций с данными. Каждая функция обладает **уникальным именем** и **набором аргументов**. Аргументы функции представляют собой данные, с которыми работает функция. Например, аргументами функции, возводящей число в определенную степень, являются само число и показатель степени.

**✅ Функция CONCAT()** предназначена для объединения (конкатенации) нескольких значений. Она принимает переменное количество аргументов, выполняет их конкатенацию и возвращает полученный результат. Результатом функции CONCAT() всегда является строка, однако аргументы функции могут принадлежать любым типам, поскольку функция неявно преобразует все аргументы в строки перед объединением.

**✅ Функция CONCAT\_WS()**предназначена для объединения (конкатенации) нескольких значений. В отличие от функции CONCAT(), она не только объединяет значения, но и добавляет между ними разделитель. Разделитель указывается в качестве первого аргумента, объединяемые значения — в качестве всех остальных.

**✅ Функция CONCAT\_WS()**, как и функция CONCAT(), умеет работать с аргументами любых типов, поскольку перед объединением неявно преобразует все аргументы в строки.

**✅**В SQL поддерживаются**основные математические операторы:** + (сложение), - (вычитание), \* (умножение) и / (деление). Для управления порядком обработки операторов можно использовать круглые скобки.

**✅**Если **хотя бы один из операндов** в математическом выражении **равен NULL**, результатом всего выражения также будет значение NULL.

**✅ Псевдонимы** **вычисляемых полей**, как и псевдонимы обычных полей, не могут быть использованы в блоке оператора WHERE для фильтрации записей.

**✅ Если псевдоним** **вычисляемого поля** совпадает с именем одного из полей таблицы, то сортировка будет осуществляться по значениям вычисляемого поля. Если необходимо осуществить сортировку по значениям поля таблицы, обращаться к полю следует через имя таблицы.

**✅ Все встроенные функции** в SQL можно разбить на четыре основных типа:

* **Текстовые функции.** Используются для обработки текста, например, для отсечения пробелов или преобразования символов в верхний регистр.
* **Числовые функции.** Используются для выполнения математических операций над числовыми данными: например, возведения в степень или извлечения квадратного корня.
* **Функции даты и времени.**Используются для обработки даты и времени, это может быть определение разницы между датами или проверка даты на корректность.
* **Системные функции.** Используются для получения специфичной для конкретной СУБД информации, к примеру, сведений об учетной записи пользователя.

**✅ Функция CHAR\_LENGTH()** используется для вычисления длины строки. Она принимает в качестве аргумента строку и возвращает количество символов в ней.

**✅ Функция LOWER()** используется для перевода строки в нижний регистр. Она принимает в качестве аргумента строку, преобразует все ее символы в нижний регистр и возвращает полученный результат.

**✅ Функция UPPER()** используется для перевода строки в верхний регистр. Она принимает в качестве аргумента строку, преобразует все ее символы в верхний регистр и возвращает полученный результат.

**✅ Функция LTRIM()** используется для отсечения ведущих пробелов. Она принимает в качестве аргумента строку, удаляет все пробелы из ее левой части и возвращает полученный результат.

**✅ Функция RTRIM()** используется для отсечения пробелов справа. Она принимает в качестве аргумента строку, удаляет все пробелы из ее правой части и возвращает полученный результат.

**✅ Функция REVERSE()** используется для переворота строки. Она принимает в качестве аргумента строку и возвращает ее с обратным порядком символов.

**✅ Функция REPEAT()** используется для повторения строки. Она принимает два аргумента в следующем порядке:

* str — исходная строка
* count — количество повторений

Функция возвращает строку, состоящую из строки str, повторяющейся count раз.

**✅ Функция LPAD()** используется для увеличения длины строки до определенного значения путем добавления в ее начало другой строки. Она принимает три аргумента в следующем порядке:

* str — исходная строка
* len — желаемая длина строки
* padstr — дополняющая строка

Функция добавляет строку padstr в начало строки str до тех пор, пока длина второй не станет равна len, и возвращает полученный результат.

**✅ Функция RPAD()** используется для увеличения длины строки до определенного значения путем добавления в ее конец другой строки. Она принимает три аргумента в следующем порядке:

* str — исходная строка
* len — желаемая длина строки
* padstr — дополняющая строка

Функция добавляет строку padstr в конец строки str до тех пор, пока длина второй не станет равна len, и возвращает полученный результат.

**✅ Функции** могут применяться как в блоке оператора SELECT, так и в блоках операторов WHERE и ORDER BY.

**✅ Функции** CHAR\_LENGTH(), LOWER(), UPPER(), LTRIM(), RTRIM(), REVERSE(), REPEAT(), LPAD() и RPAD() при вызове с аргументом NULL возвращают значение NULL.

**✅ Функции** CHAR\_LENGTH(), REVERSE(), REPEAT(), LPAD() и RPAD() также могут обрабатывать и **числовые данные**.

**✅ Функция LEFT()** используется для извлечения определенного количества символов из начала строки. Она принимает два аргумента в следующем порядке:

* str — исходная строка
* count — количество извлекаемых символов

Функция возвращает строку, состоящую из первых count символов строки str. Если количество извлекаемых символов меньше 1, функция вернет пустую строку, если больше длины строки — всю строку.

**✅ Функция RIGHT()** используется для извлечения определенного количества символов из конца строки. Она принимает два аргумента в следующем порядке:

* str — исходная строка
* count — количество извлекаемых символов

Функция возвращает строку, состоящую из последних count символов строки str. Если количество извлекаемых символов меньше 1, функция вернет пустую строку, если больше длины строки — всю строку.

**✅ Функция LOCATE()** используется для определения местоположения подстроки в строке. Она принимает три аргумента в следующем порядке:

* substr — искомая подстрока
* str — исходная строка
* start — позиция начала поиска (может не указываться, в таком случае поиск выполняется с начала строки)

Функция выполняет поиск подстроки substr в строке str, начиная с позиции start, и возвращает позицию ее первого вхождения. При этом символы исходной строки нумеруются с 1, а не с 0. Выполняет поиск позиции подстроки без учета регистра. Если искомой подстроки в строке нет, функция вернет значение 0.

**✅ Функция REPLACE()** используется для замены подстроки в строке. Она принимает три аргумента в следующем порядке:

* str — исходная строка
* from\_str — заменяемая подстрока
* to\_str — заменяющая подстрока

Функция заменяет все вхождения подстроки from\_str в строке str на подстроку to\_str и возвращает полученный результат. Если заменяемой подстроки в строке нет, функция вернет строку в исходном виде. Выполняет замену с учетом регистра.

**✅ Функция SUBSTRING()** используется для извлечения подстроки из строки. Она принимает три аргумента в следующем порядке:

* str — исходная строка
* start — позиция первого извлекаемого символа
* len — длина извлекаемой подстроки (может не указываться, в таком случае подстрока извлекается до конца)

Функция извлекает из строки str подстроку длины len, начиная с символа с позицией start, и возвращает полученный результат. При этом символы исходной строки нумеруются с 1, а не с 0. Позиция первого извлекаемого символа может быть отрицательным числом, в таком случае символы нумеруются с конца. Последнему символу соответствует позиция -1, предпоследнему — -2 и так далее.

**✅ Функция SUBSTRING\_INDEX()** используется для извлечения подстроки из строки с помощью разделителя. Она принимает три аргумента в следующем порядке:

* str — исходная строка
* delimiter — разделитель
* count — количество появлений разделителя

Функция возвращает подстроку, входящую в str, перед count появлением разделителя delimiter. Если указанное количество появлений разделителя больше количества раз, которое разделитель встречается в строке, функция вернет всю строку. Количество появлений разделителя может быть отрицательным числом, в таком случае извлечение подстроки происходит с конца. Если разделителя в строке нет, функция вернет строку в исходном виде. Выполняет поиск разделителя в строке с учетом регистра.

**✅ Функция TRIM()** используется для удаления всех вхождений подстроки из начала и/или конца строки. Синтаксис данной функции несколько отличается от предыдущих, так как при ее вызове аргументы не указываются через запятую, а связываются с помощью ключевого слова FROM:

TRIM(<ключевое слово LEADING, TRAILING или BOTH> <удаляемая подстрока> FROM <исходная строка>)

Ключевые слова LEADING, TRAILING и BOTH определяют, откуда будет происходить удаление подстроки:

* LEADING — из начала строки
* TRAILING  — из конца строки
* BOTH — из начала и конца строки

Ключевые слова могут не указываться, в таком случае удаление подстроки происходит из начала и конца строки. Удаляемая подстрока также может быть опущена, в таком случае ею считается пробел. Если не указаны и ключевое слово, и удаляемая подстрока, функция TRIM() выполнит удаление всех пробелов из начала и конца строки. При таком варианте использования функции ключевое слово FROM указывать не нужно.

**✅ Оператор DIV** используется для целочисленного деления одного числа на другое.

**✅ Оператор MOD** используется для вычисления остатка от деления одного числа на другое.

**✅ Для вычисления** остатка от деления можно использовать оператор % или функцию MOD().

**✅ Функция ABS()** используется для вычисления модуля числа. Она принимает в качестве аргумента число и возвращает его модуль.

**✅ Функция ROUND()** используется для округления числа. Она принимает два аргумента в следующем порядке:

* num — число
* decimals — количество знаков после запятой

Функция возвращает число num, округленное до decimals знаков после запятой. Функция ROUND() округляет число согласно математическим правилам округления, то есть в большую по модулю сторону. Количество знаков после запятой может быть не указано, в таком случае функция выполнит округление до целого числа.

**✅ Функция POW()** используется для возведения числа в определенную степень. Она принимает два аргумента в следующем порядке:

* num — число
* degree — показатель степени

Функция возвращает число num, возведенное в степень degree.

**✅ Функция SQRT()** используется для вычисления квадратного корня. Она принимает в качестве аргумента число, извлекает из него квадратный корень и возвращает полученный результат. Если переданное в качестве аргумента число меньше нуля, функция SQRT() вернет значение NULL.

**✅ Функция RAND()** используется для генерации случайных чисел. При вызове без аргументов она возвращает случайное число с плавающей точкой в диапазоне [0, 1). Может принимать один необязательный аргумент — целое число, определяющее начальное значение при генерации случайного числа. С одним и тем же аргументом функция будет возвращать одно и то же случайное число.

**✅**SQL поддерживает две **дополнительные функции для округления чисел**: FLOOR() и CEILING(). Первая функция используется для округления числа в меньшую сторону, вторая — в большую сторону.

**✅ Функция DEGREES()** используется для перевода значения из радиан в градусы. Она принимает в качестве аргумента значение в радианах, конвертирует его в градусы и возвращает полученный результат.

**✅ Функция RADIANS()** используется для перевода значения из градусов в радианы. Она принимает в качестве аргумента значение в градусах, конвертирует его в радианы и возвращает полученный результат.

**✅ Функция PI()** используется для получения числа пи (π*π*). Она не принимает никаких аргументов и возвращает число пи с точностью до шести знаков после запятой.

**✅ Функции SIN(), COS() и TAN()** используются для вычисления синуса, косинуса и тангенса соответственно. Каждая функция принимает в качестве аргумента значение в радианах и возвращает соответствующую тригонометрическую величину.

**✅ Функция LEAST()** используется для поиска минимального значения. Она принимает переменное количество аргументов (не меньше двух) и возвращает наименьший из них. Если поиск значения происходит среди действительных (типы FLOAT и DOUBLE) и целых чисел, функция сначала преобразует все числа в действительные. Если хотя бы один из переданных аргументов является строкой, функция перед поиском сначала преобразует все аргументы в строки. Если хотя бы один из переданных аргументов равняется NULL, функция вернет NULL.

**✅ Функция GREATEST()** используется для поиска максимального значения. Она принимает переменное количество аргументов (не меньше двух) и возвращает наибольший из них. Если поиск значения происходит среди действительных (типы FLOAT и DOUBLE) и целых чисел, функция сначала преобразует все числа в действительные. Если хотя бы один из переданных аргументов является строкой, функция перед поиском сначала преобразует все аргументы в строки. Если хотя бы один из переданных аргументов равняется NULL, функция вернет NULL.

**✅ Функция CONV()** используется для перевода числа из одной системы счисления в другую. Она принимает три аргумента в следующем порядке:

* num — число
* from\_base — система счисления, в которой представлено число
* to\_base — система счисления, в которую нужно перевести число

Функция переводит число num из системы счисления from\_base в систему счисления to\_base и возвращает полученный результат в виде строки.

**✅ Функция FORMAT()** используется для округления и форматирования числа. Она принимает два аргумента в следующем порядке:

* num — число
* decimals — количество знаков после запятой

Функция округляет число num до decimals знаков после запятой, форматирует его путем разбиения на разряды и возвращает полученный результат в виде строки. Она округляет число согласно математическим правилам округления, то есть в большую по модулю сторону. Если указанное количество знаков после запятой больше фактического количества знаков числа, функция заполнит недостающие знаки незначащими нулями.

**✅ В SQL для работы с датой и временем** часто используются **временные интервалы**. Они представляют собой некоторый промежуток времени, выраженный в тех или иных единицах измерения (день, год, минута), и используются для выполнения арифметических операций с датой и временем. Для записи временных интервалов применяется следующий синтаксис:

INTERVAL <величина интервала> <единица измерения>

**✅ Единица измерения** во временном интервале определяется ключевым словом. Для одиночных единиц измерения, таких как день или год, форматом величины является целое число. Для составных единиц измерения, таких как год и месяц, форматом величины является строка определенного вида, в которой перечислены все необходимые значения компонентов.

**✅ Одиночные** единицы измерения:

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

**✅ Составные** единицы измерения, а также формат величины для каждой из единиц:

|  |  |  |
| --- | --- | --- |
| **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 |

**✅ Функция NOW()** используется для получения текущей даты и времени. Она не принимает никаких аргументов и возвращает текущую дату и время в локальном часовом поясе.

**✅ Функция CURDATE()** используется для получения текущей даты. Она не принимает никаких аргументов и возвращает текущую дату в локальном часовом поясе.

**✅ Функция CURTIME()** используется для получения текущего времени. Она не принимает никаких аргументов и возвращает текущее время в локальном часовом поясе.

**✅ Функция UTC\_TIMESTAMP()** используется для получения текущей даты и времени. Она не принимает никаких аргументов и возвращает текущую дату и время в UTC, то есть в нулевом часовом поясе.

**✅ Функция UTC\_DATE()** используется для получения текущей даты. Она не принимает никаких аргументов и возвращает текущую дату в UTC, то есть в нулевом часовом поясе.

**✅ Функция UTC\_TIME()** используется для получения текущего времени. Она не принимает никаких аргументов и возвращает текущее время в UTC, то есть в нулевом часовом поясе.

**✅**Во время работы с датой и временем **рекомендуется использовать именно функции UTC\_TIMESTAMP(), UTC\_DATE() и UTC\_TIME()**, а приведение к определенному часовому поясу при необходимости выполнять вручную.

**✅ Функция MAKEDATE()** используется для создания даты. Она принимает два аргумента в следующем порядке:

* year — год
* day — номер дня в году

Функция возвращает дату, которая имеет порядковый номер day в году year. Если указанный номер дня в году больше количества дней в этом году, функция MAKEDATE() преобразует каждые 365-366 дней в полный год.

**✅ Функция MAKETIME()** используется для создания временного значения. Она принимает три аргумента в следующем порядке:

* hours — часы
* minutes — минуты
* seconds — секунды

Функция возвращает временное значение, состоящее из hours часов, minutes минут и seconds секунд. Может обрабатывать значение часов больше, чем есть в сутках, но не может обрабатывать значение минут и секунд больше 59. Если количество минут или секунд больше 59, вернет значение NULL.

**✅**Если хотя бы **один из аргументов**, переданных в функции MAKEDATE() и MAKETIME(), **являeтся значением NULL**, функции вернут значение NULL.

**✅ Функция TIME\_TO\_SEC()** используется для преобразования временного значения в секунды. Она принимает в качестве аргумента временное значение, преобразует его в количество секунд и возвращает полученный результат.

**✅ Функция SEC\_TO\_TIME()** используется для преобразования секунд во временное значение. Она принимает в качестве аргумента количество секунд, преобразует его во временное значение и возвращает полученный результат.

**✅ Функция TO\_DAYS()** используется для преобразования даты в количество дней. Она принимает в качестве аргумента дату, преобразует ее в количество дней и возвращает полученный результат.

**✅ Функция TO\_SECONDS()** используется для преобразования даты в количество секунд. Она принимает в качестве аргумента дату, преобразует ее в количество секунд и возвращает полученный результат.

**✅ Функция LAST\_DAY()** используется для замены дня на последний день месяца. Она принимает в качестве аргумента дату, заменяет в ней день на последний день месяца этой даты и возвращает полученный результат.

**✅ Функции** TIME\_TO\_SEC(), SEC\_TO\_TIME(), TO\_DAYS(), TO\_SECONDS() и LAST\_DAY() при вызове с аргументом NULL возвращают значение NULL.

**✅ Функция DATE()** используется для получения даты из даты и времени. Она принимает в качестве аргумента дату и время, извлекает из него дату и возвращает полученный результат.

**✅ Функция TIME()** используется для получения времени из даты и времени. Она принимает в качестве аргумента дату и время, извлекает из него время и возвращает полученный результат.

**✅ Функция HOUR()** используется для определения количества часов во времени. Она принимает в качестве аргумента временное значение, извлекает из него часы и возвращает полученный результат.

**✅ Функция MINUTE()** используется для определения количества минут во времени. Она принимает в качестве аргумента временное значение, извлекает из него минуты и возвращает полученный результат.

**✅ Функция SECOND()** используется для определения количества секунд во времени. Она принимает в качестве аргумента временное значение, извлекает из него секунды и возвращает полученный результат.

**✅ Функция MICROSECOND()** используется для определения количества микросекунд во времени. Она принимает в качестве аргумента временное значение, извлекает из него микросекунды и возвращает полученный результат.

**✅ Функция YEAR()** используется для извлечения значения года из даты. Она принимает в качестве аргумента дату, извлекает из нее год и возвращает полученный результат.

**✅ Функция MONTH()** используется для извлечения значения месяца из даты. Она принимает в качестве аргумента дату, извлекает из нее месяц и возвращает полученный результат.

**✅ Функция DAY()** используется для извлечения значения дня из даты. Она принимает в качестве аргумента дату, извлекает из нее день и возвращает полученный результат.

**✅ Функция DAYOFYEAR()** используется для вычисления номера дня в году (от 1 до 366) на основе даты. Она принимает в качестве аргумента дату, определяет, какой по счету день в году соответствует переданной дате, и возвращает полученный результат.

**✅ Функция WEEK()** используется для вычисления номера недели в году на основе даты. Она принимает в качестве аргумента дату, определяет, какая по счету неделя в году соответствует переданной дате, и возвращает полученный результат.

**✅ Функция WEEKDAY()** используется для вычисления номера дня недели (от 0 до 6) на основе даты. Она принимает в качестве аргумента дату, определяет день недели, который соответствует переданной дате, и возвращает его номер.

**✅ Функция QUARTER()** используется для вычисления номера квартала (от 1 до 4) на основе даты. Она принимает в качестве аргумента дату, определяет квартал, в котором находится переданная дата, и возвращает его номер.

**✅ Функция DAYNAME()** используется для вычисления названия дня недели на основе даты. Она принимает в качестве аргумента дату, определяет день недели, который соответствует переданной дате, и возвращает его полное название.

**✅ Функция MONTHNAME()** используется для определения названия месяца на основе даты. Она принимает в качестве аргумента дату и возвращает полное название ее месяца.

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

**✅ Функция ADDDATE()** используется для прибавления временного интервала к дате и времени. Она принимает два аргумента в следующем порядке:

* datetime — дата и время
* interval — временной интервал

Функция прибавляет временной интервал interval к дате и времени datetime и возвращает полученный результат. В качестве первого аргумента может принимать не только дату и время, но и просто дату.

**✅ Функция SUBDATE()** используется для вычитания временного интервала из даты и времени. Она принимает два аргумента в следующем порядке:

* datetime — дата и время
* interval — временной интервал

Функция вычитает временной интервал interval из даты и времени datetime и возвращает полученный результат. В качестве первого аргумента может принимать не только дату и время, но и просто дату.

**✅ Функция ADDTIME()** используется для прибавления временного значения к дате и времени. Она принимает два аргумента в следующем порядке:

* datetime — дата и время
* time — временное значение

Функция прибавляет временное значение time к дате и времени datetime и возвращает полученный результат. В качестве первого аргумента может принимать не только дату и время, но и просто время.

**✅ Функция SUBTIME()** используется для вычитания временного значения из даты и времени. Она принимает два аргумента в следующем порядке:

* datetime — дата и время
* time — временное значение

Функция вычитает временное значение time из даты и времени datetime и возвращает полученный результат. В качестве первого аргумента может принимать не только дату и время, но и просто время. Если первое временное значение меньше второго, функция вернет отрицательный результат.

**✅ Функция DATEDIFF()** используется для вычисления разницы в днях между двумя значениями даты и времени. Она принимает два аргумента в следующем порядке:

* datetime1 — первая дата и время
* datetime2 — вторая дата и время

Функция возвращает количество дней между значениями даты и времени datetime1 и datetime2. Если первая дата и время меньше второй, функция вернет отрицательный результат. При вычислении разницы функция использует только дату, поэтому каждый из аргументов может быть как датой и временем, так и просто датой.

**✅ Функция TIMEDIFF()** используется для вычисления разницы во времени между двумя значениями даты и времени. Она принимает два аргумента в следующем порядке:

* datetime1 — первая дата и время
* datetime2 — вторая дата и время

Функция возвращает разницу во времени между значениями даты и времени datetime1 и datetime2. Если первая дата и время меньше второй, функция вернет результат в виде отрицательного временного значения. Функция может использоваться не только для вычисления разницы во времени между двумя значениями даты и времени, но и просто между двумя временными значениями.

**✅ Функция TIMESTAMPDIFF()** используется для вычисления разницы между двумя значениями даты и времени в определенных единицах измерения. Она принимает три аргумента в следующем порядке:

* unit — единица измерения, в которой требуется получить разницу
* datetime1 — первая дата и время
* datetime2 — вторая дата и время

Функция возвращает разницу между значениями даты и времени datetime1 и datetime2 в единицах измерения unit. Если первая дата и время больше второй, функция вернет отрицательный результат. Единица измерения, передаваемая в функцию в качестве первого аргумента, должна быть одиночной. Каждый из аргументов, передаваемых в функцию TIMESTAMPDIFF(), может быть как датой и временем, так и просто датой, при этом дата имеет нулевые значения по всем компонентам времени.

**✅ Функция TIMESTAMP()** используется для объединения даты и временного значения. Она принимает два аргумента в следующем порядке:

* date — дата
* time — временное значение

Функция объединяет дату date и время time и возвращает полученный результат в виде единого объекта. При этом она автоматически конвертирует каждые 24 часа в 1 день. Временное значение может не указываться, в таком случае оно будет принято равным 00:00:00.

**✅ SQL позволяет** определить любой необходимый формат отображения даты и времени c помощью **строк формата**с использованием **спецификаторов формата.**

**✅** **Основные** 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 |

**✅ Функция DATE\_FORMAT()** используется для форматирования даты и времени. Она принимает два аргумента в следующем порядке:

* datetime — дата и время
* format — строка формата

Функция преобразует дату и время datetime в строку в соответствии с форматом format и возвращает полученный результат. В качестве первого аргумента может принимать не только дату и время, но и просто дату. Во втором случае считается, что дата имеет нулевые значения по всем компонентам времени.

**✅ Функция TIME\_FORMAT()** используется для форматирования времени. Она принимает два аргумента в следующем порядке:

* time — временное значение
* format — строка формата

Функция преобразует временное значение time в строку в соответствии с форматом format и возвращает полученный результат.

**✅ Функция STR\_TO\_DATE()** используется для выполнения операции, обратной форматированию. Она принимает два аргумента в следующем порядке:

* string — строка с компонентами даты и времени
* format — строка формата

Функция извлекает из строки string компоненты даты и времени в соответствии с форматом format и возвращает дату и время на основе извлеченных компонентов. Если в строке с компонентами представлены только компоненты даты, функция вернет дату без времени, если только компоненты времени — время без даты. При этом для создания даты или даты и времени должны быть определены все компоненты даты (год, месяц, число). Если хотя бы один компонент пропущен, функция вернет значение NULL. В случае с временным значением достаточно одного любого компонента, остальные по умолчанию примут значение 0. Во время использования функции необходимо следить за тем, чтобы строка с компонентами даты и времени полностью соответствовала строке формата, в противном случае функция вернет значение NULL.

**✅ Функция GET\_FORMAT()** используется для получения определенной строки формата для даты, времени или даты и времени. Она принимает два аргумента в следующем порядке:

* type — тип объекта, для которого требуется строка формата (DATE, TIME или DATETIME)
* format — непосредственно формат (USA, JIS, ISO или EUR)

Функция возвращает строку формата format для объекта типа type. Удобно использовать совместно с функциями DATE\_FORMAT() и TIME\_FORMAT().

**✅ Формат ISO** — международный стандарт, выпущенный организацией ISO (International Organization for Standardization), который описывает форматы дат и времени и дает рекомендации для их использования в международном контексте.

**✅ Функция ISNULL()** используется для проверки значения на NULL. Она принимает в качестве аргумента произвольное значение и возвращает 1, если переданным значением является NULL, или 0 в противном случае.

**✅ Функция IF()** используется для выбора одного из двух значений в зависимости от результата некоторого условного выражения. Она принимает три аргумента в следующем порядке:

* condition — условное выражение
* value1 — произвольное значение
* value2 — произвольное значение

Функция возвращает значение value1, если условное выражение condition истинно, или значение value2 в противном случае.

**✅ Функция COALESCE()** используется для поиска первого непустого значения. Она принимает переменное количество аргументов и возвращает первый из них, не равный NULL. Если все аргументы являются значением NULL, возвращаемым значением функции также будет значение NULL.

**✅ Функция IFNULL()** используется для замены некоторого значения на альтернативное, если исходное значение равняется NULL. Она принимает два аргумента в следующем порядке:

* value — исходное значение
* alternative\_value — альтернативное значение

Функция возвращает значение value, если оно не равняется NULL, или значение alternative\_value в противном случае.

**✅ Функция NULLIF()** используется для сравнения двух значений. Она принимает два аргумента в следующем порядке:

* value1 — произвольное значение
* value2 — произвольное значение

Функция возвращает значение NULL, если значения value1 и value2 совпадают, или значение value1 в противном случае.

**✅ Функция CONVERT()** принимает два аргумента в следующем порядке:

* value — значение, которое необходимо преобразовать
* type — желаемый тип

Функция приводит значение value к типу type и возвращает полученный результат. Функция CONVERT() умеет приводить значение к следующим типам:

|  |  |
| --- | --- |
| **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 |
| BINARY | значение типа BINARY |
| SIGNED | значение типа BIGINT с учетом знака |
| UNSIGNED | значение типа BIGINT без учета знака |
| YEAR | значение типа YEAR |

**✅ Функция CAST()** ведет себя так же, как и CONVERT(), только вместо запятой для разделения аргументов используется ключевое слово AS.

**✅ Оператор CASE**используется для проверки нескольких условий и в зависимости от истинности одного из них возвращает соответствующее значение. Он поддерживает две формы записи: **простую** и **усложненную**.

**✅ Простая форма** оператора CASE имеет следующий синтаксис:

CASE <значение>

    WHEN <первое сравниваемое значение> THEN <результат>

    WHEN <второе сравниваемое значение> THEN <результат>

...

WHEN <n-e сравниваемое значение> THEN <результат>

    ELSE <значение по умолчанию>

END

**✅ Усложненная форма** оператора CASE имеет следующий синтаксис:

CASE

    WHEN <первое условие> THEN <результат>

    WHEN <второе условие> THEN <результат>

...

WHEN <n-e условие> THEN <результат>

    ELSE <значение по умолчанию>

END AS <псевдоним>

**✅ Ключевое слово ELSE** может быть опущено, в таком случае значением по умолчанию считается NULL.

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

**✅ Оператор CASE** можно использовать в блоке оператора ORDER BY для определения порядка сортировки по определенным критериям.

**✅ Агрегатные функции**— функции, которые выполняют вычисления на наборе значений и возвращают одиночное значение.

**✅ Функция AVG()** используется для вычисления среднего арифметического числовых значений поля.

**✅ Функция 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 \*. Для определения количества значений в поле функции COUNT() в качестве аргумента необходимо передать это поле, при этом значения NULL не учитываются.

**✅ Функции MIN()** и **MAX()** используются для поиска минимального и максимального значений в поле соответственно. Несмотря на то что данные функции обычно применяются к полям, содержащим числовые значения или даты, они могут применяться и к строковым полям.

**✅ Функция SUM()** используется для вычисления суммы числовых значений поля. Функция игнорирует значения NULL при вычислении суммы значений поля.

**✅ Функция GROUP\_CONCAT()** используется для перечисления значений поля через запятую. Перечисляемые ею значения можно сортировать. Для этого после названия поля, значения которого необходимо перечислить, нужно указать оператор ORDER BY, а затем предоставить одно или несколько полей, по которым должна быть выполнена сортировка. Чтобы задать собственный разделитель, необходимо воспользоваться ключевым словом SEPARATOR, после которого предоставить строку, содержащую разделитель. При совместном использовании операторов ORDER BY и SEPARATOR сначала указываются правила сортировки, после определяется разделитель. Функция игнорирует значения NULL при перечислении значений поля.

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

**✅ В качестве аргумента** агрегатным функциям можно передавать**вычисляемые поля**.

**✅ Агрегатные функции** не могут использоваться в блоке WHERE.

**✅ Группировка** дает возможность разделить все данные на логические наборы, благодаря чему становится возможным выполнение итоговых вычислений отдельно по каждой группе.

**✅ Группа** представляет собой набор записей, которые определенным образом связаны друг с другом.

**✅ Создание групп** выполняется с помощью оператора **GROUP BY**, после которого указывается название поля, определяющего принцип группировки.

**✅**После оператора GROUP BY можно указывать **несколько** полей. В этом случае группировка будет выполняться **по всем** перечисленным полям.

**✅**Поля, указываемыепосле оператора GROUP BY, **могут быть** **вычисляемыми**.

**✅ Нельзя извлечь** поле, не участвующее в группировке, так как у двух записей в группе могут быть разные значения по этому полю.

**✅ Нельзя осуществлять** сортировку по полю, не участвующему в группировке.

**✅ Для фильтрации групп** в SQL используется **оператор** **HAVING.**Егоследует использовать только вместе с оператором GROUP BY.

**✅ Для сортировки** групп используется **оператор** **ORDER BY**.

**✅ При формировании** SQL-запроса всегда необходимо **соблюдать порядок** используемых операторов.В таблице ниже перечислены все изученные нами на данный момент операторы в том порядке, в котором они должны располагаться в запросе:

|  |  |
| --- | --- |
| **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 |

**✅ При выполнении** SQL-запроса операторы выполняются в **определенном порядке**. В таблице ниже представлен порядок их выполнения:

|  |  |
| --- | --- |
| **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 |

**✅**Для оператора GROUP BY **все** значения NULL трактуются как **равные**.

**✅**Если группировка записей выполняется по вычисляемому полю, причем этому полю присвоен псевдоним, то в операторе GROUP BY можно воспользоваться этим псевдонимом.

**✅ Подзапрос** — это запрос, вложенный в другой запрос. Подзапрос всегда заключен в круглые скобки и обычно выполняется перед основным запросом. Подзапросы являются обычными запросами, и для них не предусмотрен какой-либо отдельный синтаксис, поэтому для составления подзапросов достаточно уметь составлять обычные запросы.

**✅ Сравнение** одиночного значения **с таблицей, полученной в результате подзапроса**, возможно только в том случае, если эта таблица состоит из одного поля и одной записи, то есть содержит единственное значение. Однако с помощью **ключевых слов ALL и ANY** можно выполнять сравнение с таблицей, содержащей одно поле и произвольное количество записей, то есть сравнивать одиночное значение с набором значений.

**✅**При использовании **ключевого слова ALL** условие в операции сравнения должно быть истинно для всех значений, которые возвращаются подзапросом.

**✅**При использовании **ключевого слова ANY** условие в операции сравнения должно быть истинно хотя бы для одного из значений, которые возвращаются подзапросом. У ключевого слова ANY **имеется аналог** в виде ключевого слова **SOME**. По своему действию оба этих ключевых слова идентичны, поэтому применять можно любое из них.

**✅**Так как **результатом подзапроса** является таблица, она может быть указана после ключевого слова FROM. В таком случае основной запрос будет работать с той таблицей, которую вернул подзапрос. При использовании подзапроса совместно с ключевым словом FROM ему необходимо присвоить **псевдоним**, в противном случае при выполнении запроса произойдет ошибка.

**✅**Если результатом подзапроса является таблица **с единственным значением**, то это значение может быть извлечено с помощью оператора SELECT, например, как обычное число.

**✅**Если результатом подзапроса является **пустая таблица**, она приравнивается к значению NULL.

**✅ Некоррелированным** называется подзапрос, результат которого вычисляется единожды перед выполнением основного запроса.

**✅ Коррелированным** называется подзапрос, результат которого вычисляется для каждой рассматриваемой основным запросом записи отдельно.

**✅**Поскольку **коррелированный** подзапрос помимо значений полей собственной таблицы использует значения полей таблицы, принадлежащей основному запросу, должна быть возможность различать эти поля, если они имеют одинаковые названия. Для этого при обращении к полю явно указывают **название таблицы**. Если и основной запрос, и подзапрос обращаются к одной и той же таблице, то одной из таблиц дают **псевдоним**.

**✅**SQL позволяет сравнивать не только одиночные значения, но и пары значений, тройки значений и так далее. В подобных случаях используются подзапросы, возвращающие таблицы из нескольких полей.

**✅**Обращение к полю с явным указанием таблицы является лишь обращением по **полному имени**. Полные имена применяются в тех случаях, когда может возникнуть неопределенность в именовании полей, к примеру, когда поле подзапроса сравнивается с полем основного запроса и эти поля имеют одинаковые имена.

**✅**Если таблице присвоен **псевдоним**, то обращение к полям этой таблицы по полному имени должно происходить только с помощью данного псевдонима.

**✅**Если подзапрос возвращает таблицу из одного поля и нескольких записей, то помимо сравнения некоторого значения с этой таблицей с помощью ключевых слов ALL и ANY можно выполнить проверку на вхождение с помощью **оператора IN**.

**✅**Работа с несколькими таблицами практически не отличается от работы с одной таблицей. Разница заключается лишь в том, что для извлечения нужной информации из таблицы необходимо пользоваться данными не из этой же таблицы, а из других.

**✅ Оператор EXISTS** проверяет, есть ли записи, удовлетворяющие заданному в подзапросе условию, без извлечения записей. Если найдется хотя бы одна запись, проверка прервется и оператор EXISTS вернет значение 1, а если нет, то значение 0.

**✅ Оператор NOT EXISTS** проверяет отсутствие записей, удовлетворяющих заданному в подзапросе условию, без извлечения записей. Если найдется хотя бы одна запись, проверка прервется и оператор NOT EXISTS вернет значение 0, а если нет, то значение 1.

**✅**Операторы EXISTS и NOT EXISTS целесообразно использовать только с **коррелированными** подзапросами.

**✅**В отличие от операторов IN и NOT IN операторы EXISTS и NOT EXISTS **не зависят** от конкретных значений в наборе данных, их интересует только наличие записей. Поэтому они **корректно** обработают запрос, даже если в подзапросе присутствуют значения NULL.

**✅**Основной идеей **реляционной модели** хранения данных является разбиение информации по тематическим таблицам и установление связей между этими таблицами с помощью первичных и внешних ключей. Такие связи позволяют избежать избыточности информации, а также обеспечивают структурированный способ хранения данных и их целостность.

**✅ Первичные ключи** делятся на два вида:

* **суррогатный ключ** — это первичный ключ, созданный специально для идентификации записей таблицы, значения которого (часто числовые или символьные) не имеют прямого отношения к самим данным
* **естественный ключ** — это первичный ключ, значения которого непосредственно связаны с реальными данными или характеристиками хранимых сущностей

**✅ Первичный ключ**, состоящий из одного поля, обычно называют **простым**. Если же первичный ключ включает несколько полей, его называют **составным**.

**✅ Внешние ключи** могут ссылаться на первичный ключ не только другой таблицы, но и той, в которой находится сам внешний ключ. Во втором случае внешний ключ называют **рекурсивным**.

**✅**Выделяют три вида связей между таблицами: **"один к одному"**, **"один ко многим"** и **"многие ко многим​**​​​​​​".

**✅ Связь "один к одному"** предполагает, что каждая запись в одной таблице связана с одной записью в другой таблице. Это означает, что одна запись в первой таблице имеет только одну соответствующую ей запись во второй таблице, и наоборот.

**✅ Связь "один ко многим"** предполагает, что одна запись в одной таблице связана с несколькими записями в другой таблице. Это означает, что у одной записи в первой таблице может быть несколько соответствующих ей записей во второй таблице, но каждая запись во второй таблице может быть связана только с одной записью в первой таблице.

**✅ Связь "многие ко многим"** предполагает, что несколько записей в одной таблице могут быть связаны с несколькими записями в другой таблице. Это означает, что у одной записи в первой таблице может быть множество соответствующих ей записей во второй таблице, и наоборот.

**✅ Соединение** представляет собой операцию объединения двух таблиц в одну таблицу согласно определенному правилу. Таблица, получаемая в результате соединения, включает все поля соединяемых таблиц, что значительно упрощает работу с исходными данными, поскольку вся необходимая информация автоматически собирается вместе и предоставляется в виде одной таблицы.

**✅ Условие соединения**определяет,как относительно друг друга будут располагаться записи в полученной в результате соединения таблице.

**✅ При внутреннем соединении** двух таблиц отбираются все пары записей, для которых выполняется условие соединения. Если записи невозможно поставить в пару запись из другой таблицы так, чтобы условие соединения выполнялось, запись будет отброшена.

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

Общий синтаксис внутреннего соединения имеет следующий вид:

<первая таблица> INNER JOIN <вторая таблица> ON <условие соединения>

**✅**Запрос, который извлекает данные из таблицы, полученной в результате соединения, называют **многотабличным**.

**✅**Таблица, полученная в результате соединения, сама может быть соединена с другой таблицей, благодаря чему мы можем выполнять соединение не только двух таблиц, но и большего их количества.

**✅**Соединять одну таблицу можно не только с другой таблицей, но и с самой собой. Подобное может быть удобно во время работы с таблицей, содержащей рекурсивный внешний ключ.

**✅**Соединение **не является физической таблицей**, то есть оно не существует как реальная таблица в базе данных. Соединение создается СУБД по мере необходимости и сохраняется только на время выполнения запроса.

**✅**При использовании оператора INNER JOIN ключевое слово INNER (в переводе на русский "внутренний") является опциональным и может быть опущено.

**✅ Внешнее соединение** бывает двух видов — **левое** и **правое:**

* при **левом внешнем соединении** двух таблиц отбираются все пары записей, для которых выполняется условие соединения, а также все оставшиеся записи из **левой** таблицы, для которых пара не нашлась
* при **правом внешнем соединении** двух таблиц отбираются все пары записей, для которых выполняется условие соединения, а также все оставшиеся записи из **правой** таблицы, для которых пара не нашлась

**✅**При внешнем соединении **записям** **без пары** в соответствие ставятся значения NULL.

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

Общий синтаксис левого внешнего соединения имеет следующий вид:

<первая таблица> LEFT OUTER JOIN <вторая таблица> ON <условие соединения>

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

Общий синтаксис правого внешнего соединения имеет следующий вид:

<первая таблица> RIGHT OUTER JOIN <вторая таблица> ON <условие соединения>

**✅**Левое и правое внешние соединения являются **взаимозаменяемыми**.

**✅**При использовании операторов LEFT OUTER JOIN и RIGHT OUTER JOIN ключевое слово OUTER (в переводе на русский "внешний") является опциональным и может быть опущено.

**✅**При **перекрестном соединении** двух таблиц каждая запись из одной таблицы сопоставляется с каждой записью из другой таблицы. Перекрестное соединение отличается от внутреннего и внешнего тем, что не требует условия соединения, поскольку при перекрестном соединении сопоставление записей выполняется не по какому-либо правилу, а обычным перебором всех возможных комбинаций.

**✅**В SQL **перекрестное соединение** выполняется с помощью оператора CROSS JOIN. Сначала указывается первая таблица, затем оператор CROSS JOIN, а после вторая таблица.

Общий синтаксис перекрестного соединения имеет следующий вид:

<первая таблица> CROSS JOIN <вторая таблица>

**✅**При выполнении **перекрестного соединения** таблицы с самой собой необходимо использовать псевдонимы, в противном случае запрос завершится с ошибкой.

**✅ Оператор UNION**используется дляобъединения результатов нескольких  запросов в один результирующий набор. Он указывается между запросами, результаты которых необходимо объединить.

**✅**Запрос, который объединяет результаты выполнения нескольких запросов, называют **комбинированным**.

**✅**При использовании оператора UNION объединяемые результаты запросов обязательно должны иметь **одинаковое количество полей**, в противном случае произойдет ошибка.

**✅**Названия полей в объединяемых результатах запросов необязательно должны быть одинаковыми, поскольку в качестве итоговых названий полей используются названия **самого первого** результата запроса.

**✅**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. **По умолчанию** оператор UNION выполняет удаление одинаковых записей.

**✅ С помощью оператора ORDER BY**можно отсортировать полученную после объединения всех результатов таблицу. Для этого нужно единожды указать оператор ORDER BY после самого последнего объединяемого запроса. Аналогичным образом к полученной после объединения всех результатов таблице можно применить **оператор LIMIT** или связку операторов ORDER BY и LIMIT. Чтобы воспользоваться указанными операторами в одном или нескольких запросах, расположенных между оператором UNION, эти запросы необходимо заключить в круглые скобки.

**✅**При объединении результатов нескольких запросов можно использовать **вспомогательное поле** как для указания принадлежности записей к исходным таблицам, так и для обеспечения нужного порядка следования этих записей в итоговой таблице.

**✅**Таблица, получаемая после объединения результатов нескольких запросов, может быть указана в блоке **оператора FROM** для последующей работы с ней.

**✅ Оператор UNION** представляет одну из операций над множествами — **объединение**. В MySQL 8.0.31 были добавлены еще два оператора для выполнения операций над множествами: **INTERSECT (пересечение)** и **EXCEPT (разность)**. Они применяются так же, как и оператор UNION, и для них действует все те же правила.

**✅ Оператор INTERSECT** используется для получения записей, которые есть в **каждом** результате запроса.

**✅ Оператор EXCEPT** используется для получения всех записей **первого** результата запроса кроме тех, которые встречаются в следующих результатах запросов.

**✅ Оператор UPDATE**используется для изменения значений записей таблицы. Запрос начинается с оператора UPDATE, после которого указывается название таблицы, подлежащей обновлению. Затем используется **ключевое слово SET**, с помощью которого для поля определяется новое значение. Для изменения значений нескольких полей после ключевого слова SET нужно перечислить через запятую все изменяемые поля и их новые значения.

**✅**Дляобновления значений **нескольких** **определенных** записей или **одной** **конкретной** записи нужно воспользоваться **оператором WHERE**, который указывается после ключевого слова SET и определяет условия отбора записей, подлежащих обновлению.

**✅**В качестве **нового значения** поля допускается использовать не только константные значения, но и разного рода **выражения**. Выражения могут быть произвольной сложности и включать в себя как простые арифметические операции, так и вызовы различных функций.

**✅**При обновлении таблицы можно пользоваться **условными конструкциями**.

**✅**При обновлении данных одной таблицы допустимо пользоваться данными из **других таблиц**.

**✅ Ключевое слово LIMIT**, используемое для ограничения количества извлекаемых записей, также может использоваться для ограничения количества обновляемых записей.

**✅**При обновлении данных **ключевое слово OFFSET**  совместно с **ключевым словом LIMIT** использовать **нельзя**.

**✅ Оператор ORDER BY** может использоваться совместно **с оператором UPDATE** для определения порядка, в котором будут обновляться записи.

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

**✅ Результатом запроса**, обновляющего данные таблицы, **не является таблица**. Поэтому, если после обновления данных необходимо тут же их получить, следует использовать дополнительный извлекающий запрос.

**✅**Если поле обладает свойством, **запрещающим хранение значения NULL**, то при попытке добавить в него NULL путем изменения любого из имеющихся в поле значений произойдет ошибка. При этом, если после оператора UPDATE указать ключевое слово IGNORE, значение поля будет изменено на **значение** **по умолчанию** в зависимости от типа поля.

**✅**Если поле имеет **значение по умолчанию**, то при изменении значений этого поля им можно воспользоваться с помощью **ключевого слова DEFAULT**.

**✅**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**.

**✅ Оператор DELETE**предназначен для удаления данных из таблицы. Его можно использовать двумя способами:

* Для удаления **всех записей**необходимо указать оператор DELETE, а затем ключевое слово FROM и название таблицы, из которой должны быть удалены данные.
* Для удаления **определенных записей**оператор DELETE необходимо использовать совместно с **оператором WHERE** для определения условий отбора записей, подлежащих удалению.

**✅**При удалении данных из одной таблицы **допустимо** пользоваться данными из **других таблиц**.

**✅**При удалении данных допустимо использование **операторов ORDER BY** и **LIMIT** для определения порядка удаления записей и ограничения количества удаляемых записей соответственно.

**✅**При использовании внешних ключей **СУБД** **запрещает** удаление записей, которые обеспечивают корректность отношений между таблицами, и при попытке выполнить подобное удаление сообщает об ошибке.

**✅ Оператор DELETE** поддерживает дополнительное **ключевое слово IGNORE**, которое позволяет не прерывать процесс удаления данных даже при возникновении ошибок.

**✅**Если необходимо **удалить все** записи таблицы, рекомендуется использовать **оператор TRUNCATE**, а не DELETE, поскольку первый выполняет операцию очистки таблицы гораздо **быстрее**.

**✅ Оператор INSERT**предназначен для добавления данных в таблицу. Его можно использовать двумя способами:

* Для добавления **одной записи**запрос начинается с оператора INSERT, после которого следуют **ключевое слово INTO** и название таблицы, в которую выполняется добавление записи. Затем используется **ключевое слово VALUES** для указания значений добавляемой записи. Они должны быть заключены в круглые скобки и перечислены через запятую, причем для каждого поля обязательно должно быть указано какое-либо значение. Также при перечислении значений добавляемой записи должен соблюдаться их порядок, поскольку первое значение будет использовано в качестве значения первого поля, второе — в качестве значения второго поля и так далее. Либо для каждого значения должно быть указано поле (в круглых скобках после имени таблицы), в которое это значение должно быть добавлено.
* Для добавления **нескольких записей**после ключевого слова VALUES необходимо через запятую указать значения для каждой из них.

**✅**В качестве значений добавляемой записи можно использовать не только константные значения, но и результаты **арифметических операций**, а также возвращаемые значения **различных функций**.

**✅**Для добавления записи с помощью **ключевого слова SET** необходимо указать выражение INSERT INTO и название таблицы, в которую выполняется добавление, а затем ключевое слово SET, после которого определить значение для каждого поля добавляемой записи, расположив их рядом друг с другом и связав знаком равенства. Сочетание операторов INSERT и SET позволяет добавить в таблицу лишь **одну запись**.

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

**✅ Оператор INSERT** поддерживает дополнительное **ключевое слово IGNORE**, которое позволяет не прерывать процесс добавления данных даже при возникновении ошибок.

**✅**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 другую запись.

**✅ Оператор CREATE TABLE**предназначен для создания таблиц. Запрос начинается с оператора CREATE TABLE, после которого указывается название таблицы. Далее следует определение таблицы, которое заключается в круглые скобки и содержит определение каждого поля. Определения полей разделяются запятыми. Определение каждого поля начинается с имени поля, а за ним указывается тип данных.

**✅**Если во время создания очередной таблицы в базе данных содержится таблица, **имя которой совпадает** с именем создаваемой, **произойдет ошибка**. 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**.

**✅**Для обеспечения целостности хранимых в таблице данных используются **ограничения**. Ограничение представляет собой **ключевое слово** (или связку ключевых слов), которое указывается в определении поля и добавляет этому полю дополнительные свойства:

* **NOT NULL** используется для того, чтобы **запретить** полю хранить **значение NULL**
* **UNIQUE** используется для того, чтобы **запретить** полю хранить **повторяющиеся значения**
* **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
* **CHECK** используется для того, чтобы запретить полю хранить значения, не удовлетворяющие заданному условию, которое представляет собой выражение, заключенное в круглые скобки, и указывается после ключевого слова CHECK

**✅**При использовании ограничений их можно **комбинировать.**Перечисление ограничений происходит через **пробел**.

**✅**При создании таблицы необходимо определить **хотя бы одно** поле. Таблица может не иметь ни одной записи, но обязательно должна иметь хотя бы одно поле.

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

**✅**В SQL для определения **первичного ключа** таблицы существует **отдельное ограничение**—**PRIMARY KEY**, которое будет гарантировать, что поле содержит лишь уникальные непустые значения.

**✅ Если** **первичный ключ является составным**, то его определение нужно вынести в отдельное определение поля, в котором следует сначала указать ограничение PRIMARY KEY, а затем в скобках перечислить все поля, составляющие первичный ключ.

**✅ Ограничение AUTO\_INCREMENT** используется для автоматического заполнения поля рядом натуральных чисел. Оно может быть указано в определении лишь одного поля, причем это поле должно быть первичным ключом. Также ограничение AUTO\_INCREMENT **не сочетается с ограничением DEFAULT**.

**✅ Ограничение** **FOREIGN KEY** предназначено для определения поля как **внешнего ключа**. Определение внешнего ключа начинается с ограничения FOREIGN KEY, после которого в скобках указывается поле, являющееся внешним ключом. Затем следует **ключевое слово REFERENCES**, а после него — имя родительской таблицы и вновь в скобках поле этой таблицы, на которое ссылается внешний ключ.

**✅**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** или **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.

**✅**Если необходимо получить информацию о полях таблицы, можно воспользоваться **оператором DESCRIBE**. Для этого после оператора DESCRIBE указывается имя интересующей таблицы.

**✅**Внешний ключ может состоять как из одного, так и из нескольких полей. Разница лишь в том, что в первом случае в скобках после ограничения FOREIGN KEY и названия родительской таблицы указывается только одно поле, а во втором случае перечисляются несколько полей через запятую.

**✅**Для **удаления таблицы** нужно воспользоваться **оператором DROP TABLE**, а затем указать имя удаляемой таблицы. Если требуется удалить несколько таблиц одновременно, достаточно после оператора DROP TABLE перечислить через запятую их имена.

**✅ Оператор DROP TABLE** удаляет таблицу только в том случае, если она содержится в базе данных, в противном случае оператор завершает операцию удаления с ошибкой. Однако если воспользоваться дополнительным **оператором IF EXISTS**, таблица будет удалена только в том случае, если она существует, в противном случае операция удаления будет проигнорирована.

**✅** Для **переименования таблицы** необходимо воспользоваться **оператором RENAME TABLE**, синтаксис которого имеет следующий вид:

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

**✅**Любой запрос, **модифицирующий таблицу**, начинается с **оператора ALTER TABLE**, после которого указывается имя таблицы, подлежащей модификации. После имени таблицы указывается действие (с использованием соответствующего оператора), которое должно быть применено по отношению к этой таблице.

**✅ Удаление поля** таблицы выполняется с помощью **оператора DROP COLUMN**.

**✅ Переименование поля** таблицы синтаксически похоже на переименование самой таблицы и выполняется с помощью **оператора RENAME COLUMN**.

**✅**Для **добавления нового поля** необходимо воспользоваться **оператором ADD COLUMN**, а затем предоставить определение нового поля таблицы. **По умолчанию** оператор ADD COLUMN добавляет новое поле **в конец** таблицы. Если после определения поля указать **ключевое слово FIRST**, новое поле будет добавлено **в начало** таблицы. Для помещения нового поля **в конкретную** часть таблицы нужно указать **ключевое слово AFTER**, а затем предоставить название поля, после которого должно быть расположено новое поле.

**✅**Для **переопределения поля** существующей таблицы необходимо воспользоваться **оператором MODIFY COLUMN**, а затем предоставить новое определение желаемого поля.

**✅**Для **определения первичного ключа** после создания таблицы 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.

**✅**Для **определения внешнего ключа** после создания таблицы необходимо воспользоваться **оператором ADD FOREIGN KEY**, а затем привычным способом определить внешний ключ.

**✅**Для**удаления внешнего ключа** необходимо воспользоваться **оператором DROP FOREIGN KEY**, а затем указать имя внешнего ключа, которое необходимо удалить.

**✅**Для **добавления ограничения CHECK** нужно воспользоваться **оператором ADD CHECK**, а затем в скобках указать проверяющее выражение. Чтобы добавить в таблицу **именованное** ограничение CHECK, можно воспользоваться **оператором ADD CONSTRAINT**, синтаксис которого имеет следующий вид:

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

**✅**Для **удаления ограничения CHECK** нужно воспользоваться **оператором DROP CONSTRAINT**, а затем указать имя ограничения CHECK, которое требуется удалить.

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

**✅** 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>

**✅ Создание, чтение, модификация и удаление** — это четыре классические операции, которые в терминологии баз данных часто называют аббревиатурой **CRUD** (Create, Read, Update, Delete).

**✅ 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 ;

* **Ключевое слово DELIMITER** используется для изменения разделителя, которым по умолчанию является 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.
* **Создание** триггера начинается с **оператора CREATE TRIGGER**, после которого указывается имя триггера.
* **Время срабатывания** определяется одним из **операторов BEFORE** и **AFTER**, а **операция**, вызывающая триггер, — одним из **операторов INSERT, DELETE** и **UPDATE**. **Оператор BEFORE** используется в том случае, если нужно, чтобы триггер выполнялся **перед** вызывающей его операцией. При необходимости выполнения триггера **после** вызывающей его операции используется **оператор AFTER**.
* **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 ;.

**✅**Операция, вызывающая триггер, всегда связана с какой-либо записью таблицы, и триггер имеет доступ к значениям этой записи. Уровень доступа определяется типом триггера:

* **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

**✅**Для **удаления** триггера необходимо воспользоваться **оператором DROP TRIGGER**, а затем указать имя удаляемого триггера.

**✅ 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.

**✅ 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.

**✅ 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.

**✅**В отличие от кластеризованных, некластеризованные индексы могут создаваться **в любом** количестве.

**✅** Кластеризованный индекс **не требует создания**, поскольку им обладает любая таблица.

**✅ Для** **создания** некластеризованного индекса используется **оператор CREATE INDEX**, синтаксис которого имеет следующий вид:

CREATE INDEX <имя некластеризованного индекса>

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

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

**✅**Некластеризованный индекс хранит все значения индексированного поля, с помощью которого ссылается на любые записи таблицы. Поэтому чем больше количество полей в индексе, тем больше памяти он занимает.

**✅ Для удаления** индекса используется **оператор DROP INDEX**.

**✅** **Для переименования** индекса используется связка **операторов ALTER TABLE** и **RENAME INDEX**, синтаксис которых имеет следующий вид:

ALTER TABLE <имя таблицы>

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

**✅**Если необходимо **получить информацию об индексах** таблицы, можно воспользоваться **оператором SHOW INDEX**, синтаксис которого имеет следующий вид:

SHOW INDEX FROM <имя таблицы>;

**✅ 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.

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

CREATE VIEW <имя представления> AS

<извлекающий запрос>

**✅ Представления**, как и таблицы, должны иметь **уникальные имена**.

**✅** **Расширенный** **оператор CREATE OR REPLACE VIEW** создает представление только в том случае, если оно еще не существует. Если же представление с создаваемым именем существует, оно будет заменено новым.

**✅ Для изменения запроса,**которыйиспользуется в представлении, используется **оператор ALTER VIEW**, синтаксис которого имеет следующий вид:

ALTER VIEW <имя представления> AS

<новый извлекающий запрос>

**✅ Для удаления**представления используется **оператор DROP VIEW**, синтаксис которого имеет следующий вид:

DROP VIEW <имя представления>

**✅ Определить названия полей** виртуальной таблицы можно не только в самом запросе с помощью псевдонимов, но и путем указания этих имен в скобках через запятую после названия представления.

**✅ 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> AS (

<извлекающий запрос, определяющий содержимое CTE>

)

<основной запрос, который может обращаться к CTE>

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

**✅**Одним из основных способов **использования CTE** является создание подзапросов, которые используются в рамках основного запроса несколько раз, что позволяет избежать повторного написания одних и тех же подзапросов.

**✅**После оператора WITH допустимо определить не только одно CTE, но и **несколько**, для этого достаточно перечислить их определения через запятую.

**✅**CTE является **обычной таблицей**, поэтому с ней допустимо выполнять любые необходимые операции: извлекать данные, фильтровать и сортировать, объединять с другими таблицами, представлениями или CTE.

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

**✅**Определить **названия полей** CTE можно не только в самом запросе с помощью псевдонимов, но и путем указания этих имен в скобках через запятую после названия CTE.

**✅ Рекурсивное обобщенное табличное выражение**или **рекурсивное CTE** — это CTE, содержимое которого последовательно формируется на основе самого себя. Тело рекурсивного CTE, в отличие от обычного CTE, включает два извлекающих запроса. Первый запрос используется для однозначного определения содержимого первой записи CTE, второй — для определения правила формирования всех остальных записей.

**✅**Запрос, определяющий первую запись CTE, обычно называют **нерекурсивным**, а запрос, описывающий процесс получения всех остальных записей, — **рекурсивным**.

**✅ 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>

* Для создания рекурсивного CTE перед его именем необходимо указать ключевое слово RECURSIVE.
* Затем предоставить два извлекающих запроса, связав их парой **ключевых слов:**либо **UNION ALL**, либо **UNION DISTINCT**.
* 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.

**✅**Если при определении **рекурсивного** CTE пропустить ключевое слово RECURSIVE, произойдет **ошибка**.

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

**✅**Определить **названия полей** рекурсивного CTE можно не только в самом запросе с помощью псевдонимов, но и путем указания этих имен в скобках через запятую после названия CTE.

**✅ Переменная** в программировании — это именованная область памяти, которая хранит определенные данные и позволяет неоднократно к ним обращаться. С помощью переменных можно сохранять значения, полученные во время промежуточных вычислений, выполнять операции с этими значениями и передавать их между различными частями программы.

**✅**В SQL выделяют два типа переменных: **системные** и **пользовательские**. **Системные** переменные используются для хранения технической информации и для настройки различных аспектов работы СУБД. **Пользовательские** переменные, в отличие от системных, определяются программистом, а не СУБД, и используются при составлении запросов для определенных целей.

**✅ Создание** пользовательской переменной и присвоение ей значения выполняется с помощью **операторов SET** и **:=**. Сначала следует оператор SET, а затем между оператором := указываются имя переменной и ее значение. Перед именем пользовательской переменной необходимо располагать **знак @**. Его необходимо использовать при создании любой пользовательской переменной и при каждом обращении к ней.

**✅ Значением** пользовательской переменной необязательно должно быть явно определенное значение, это может быть результат некоторого вычисления или возвращаемое значение функции.

**✅**Пользовательские переменные **нечувствительны** к регистру.

**✅**Внутри одного **блока SET** можно определить как **одну** пользовательскую переменную, так и **несколько,**перечислив определение каждой переменной через запятую.

**✅**Пользовательские переменные могут участвовать в математических выражениях, операциях сравнения, передаваться функциям в качестве аргументов и использоваться в контексте запроса как любые другие значения.

**✅ Значением** пользовательской переменной может являться не только вычисляемое значение, но и результат выполнения **запроса**. При этом запрос должен быть заключен в **круглые скобки**. Кроме того, запрос, определяющий значение пользовательской переменной, всегда должен возвращать **одиночное значение** (одно поле и одна запись), в противном случае произойдет ошибка.

**✅ Определить** пользовательскую переменную можно прямо внутри извлекающего запроса в блоке оператора SELECT без использования ключевого слова SET. Внутри одного блока SELECT можно определить как одну пользовательскую переменную, так и несколько.

**✅ Пользовательские** переменные нельзя использовать в блоке оператора LIMIT для ограничения количества извлекаемых записей.

**✅ При определении** пользовательской переменной с помощью ключевого слова SET допустимо использовать **оператор =**.

**✅ Системные** переменные делятся на два вида: **глобальные** и **сессионные**.

**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, изменить их они не смогут.

**Сессионные** переменные создаются во время каждого подключения к базе данных, и их значения могут изменять пользователи. Сессионные переменные создаются для каждого подключения отдельно, и если один пользователь изменит значение сессионной переменной, то у других пользователей оно не изменится.

**✅**Функции, которые не являются встроенными, а определены пользователем, называются **пользовательскими**. В зависимости от поведения, пользовательские функции делятся на два вида: **детерминированные** и **недетерминированные**.

**✅ Детерминированной** называется функция, котораяпри одних и тех же входных данных возвращает один и тот же результат.

**✅ Недетерминированной**называется функция, котораяпри одних и тех же входных данных может вернуть разные результаты.

**✅**Общий шаблон для **создания** **пользовательской функции** имеет следующий вид:

DELIMITER //

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

RETURNS <тип возвращаемого значения>

<вид функции: детерминированная или недетерминированная>

BEGIN

<тело функции>;

RETURN <возвращаемое значение>;

END //

DELIMITER ;

* **Ключевое слово DELIMITER** используется для изменение разделителя на двойной слеш.
* Далее следует **оператор CREATE FUNCTION**, после которого указывается имя функции.
* Затем в скобках через запятую выполняется определение каждого **параметра функции**, которое включает его имя и тип данных.
* 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**.
* Внутрь последующего **блока BEGIN END** помещается **тело функции**, которое и определяет ее поведение.
* Следом за телом функции внутри блока BEGIN END располагаются оператор RETURN и возвращаемое значение функции.
* Завершается создание функции обновленным разделителем // и возвратом к стандартному разделителю в виде символа ;.

**✅** **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>;

Для локальной переменной 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.

**✅**Функции, создаваемые с помощью **оператора CREATE FUNCTION**, являются **скалярными**, то есть всегда возвращают единственное значение.

**✅ Для удаления** пользовательской функции нужно воспользоваться **оператором DROP FUNCTION**, а затем указать имя удаляемой функции.

**✅**Пользовательские функции **нечувствительны** к регистру.

**✅**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** условной конструкции IF-ELSEIF-ELSE выглядит следующим образом:

IF <первое проверяемое условие> THEN

<одна или несколько операций>;

ELSEIF <второе проверяемое условие> THEN

<одна или несколько операций>;

ELSEIF <третье проверяемое условие> THEN

<одна или несколько операций>;

...

ELSEIF <n-е проверяемое условие> THEN

<одна или несколько операций>;

ELSE <одна или несколько операций>;

END IF;

* Конструкция начинается с **ключевого слова IF**, после которого указывается первое проверяемое условие.
* Следом располагается **ключевое слово THEN**, а затем одна или несколько операций, которые должны быть выполнены, если условие окажется истинным.
* После блока с первым условием следуют блоки со вторым, третьим и всеми последующими условиями. Блок с каждым дополнительным условием начинается с**ключевого слова ELSEIF**, после которого указывается само условие. Следом за условием располагается **ключевое слово THEN**, а затем одна или несколько операций, которые должны быть выполнены, если условие окажется истинным.
* После блока с последним дополнительным условием указывается **ключевое слово ELSE**, а затем одна или несколько операций, которые должны быть выполнены, если ни одно рассмотренное условие не окажется истинным.
* Завершается условная конструкция **парой ключевых слов END IF**.
* Блоки **ELSEIF**, а также блок **ELSE** являются необязательными и при необходимости могут быть опущены.

**✅ Циклы** позволяют повторять выполнение ряда операций до тех пор, пока истинно определенное условие.

**✅ Синтаксис** **цикла WHILE** выглядит следующим образом:

WHILE <проверяемое условие> DO

<тело цикла>;

END WHILE;

* Определение цикла начинается с к**лючевого слова WHILE**, после которого указывается некоторое условие.
* Затем следуют **ключевое слово DO** и непосредственно тело цикла. Тело цикла представляет собой одну или несколько операций, выполнение которых и будет повторяться.
* Завершается определение цикла **парой ключевых слов END WHILE**.

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

**✅ Процедуры**заключают в себе определенный набор операций, который выполняется при их вызове. Процедуры могут принимать различные аргументы, которые могут быть использованы в их теле. Процедуры **не имеют** возвращаемого значения. В контексте баз данных процедуры обычно называют **хранимыми процедурами**.

**✅**Общий шаблон для **создания хранимой процедуры** имеет следующий вид:

DELIMITER //

CREATE PROCEDURE <имя процедуры>(<1-й параметр>, <2-й параметр>, ...)

BEGIN

<тело процедуры>;

END //

DELIMITER ;

* Создание процедуры начинается с **ключевого слова DELIMITER**, с помощью которого выполняется изменение разделителя на двойной слеш (//).
* Далее следует **оператор CREATE PROCEDURE**, после которого указывается имя процедуры.
* Затем в скобках через запятую выполняется определение каждого параметра процедуры.
* Внутрь последующего **блока BEGIN END** помещается тело процедуры, которое и определяет ее поведение.
* Завершается создание процедуры обновленным разделителем // и возвратом к стандартному разделителю в виде символа ;.

Процедуры поддерживают несколько типов параметров: **входные** и **выходные**. Для определения параметра как входного используется **ключевое слово IN**, выходного — **OUT**. Параметры процедуры по умолчанию являются входными, поэтому ключевое слово IN указывать необязательно.

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

**✅**Несмотря на то что у процедуры отсутствует возвращаемое значение, получить результаты выполненных в ее теле операций или вычислений можно. Реализуется это с помощью **выходных** параметров. При вызове процедуры в качестве значения выходного параметра передается не конкретное значение, а **пользовательская переменная**, и если процедура в своем теле присвоит выходному параметру какое-либо значение, такое же значение примет и переменная, указанная при вызове процедуры.

**✅**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**.

**✅**При создании хранимой процедуры ее можно определить как **детерминированную** или **недетерминированную**, однако, в отличие от пользовательских функций, делать это **необязательно**. Пояснять, почему процедура является недетерминированной, также необязательно.

**✅ Для удаления** хранимой процедуры нужно воспользоваться **оператором DROP PROCEDURE**, а затем указать имя удаляемой процедуры.

**✅**Хранимые процедуры **нечувствительны** к регистру.

**✅ Результирующий набор**представляет собой таблицу, которая формируется извлекающим запросом в результате выполнения следующих операций (некоторые являются опциональными): соединение (JOIN), извлечение (FROM), фильтрация (WHERE), группировка (GROUP BY) и фильтрация групп (HAVING).

**✅ Оконная функция** — это функция, которая выполняет вычисления на основе определенного набора записей и возвращает одиночное значение. Набор записей, с которым работает оконная функция, называют **окном**. По умолчанию содержимое окна полностью совпадает с результирующим набором запроса, в рамках которого определено окно.

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

WINDOW <название окна> AS (<спецификация окна>)

**✅ Оператор WINDOW** в запросе располагается **после** оператораGROUP BY (или HAVING) и **до** оператора ORDER BY.

**✅ Спецификация окна**, заключаемая в круглые скобки, может включать три базовых элемента: **секционирование**(разбиение на секции), **упорядочивание**и **определение границ окна**.

**✅ Упорядочивание** окна использует синтаксис оператора ORDER BY.

**✅ Секционирование** выполняется с помощью оператора PARTITION BY, после которого перечисляются все поля, по которым должно быть произведено разбиение на секции. Разбиение окна на секции с помощью оператора PARTITION BY выполняется **без учета регистра**.

**✅ Определяемое окно** можно секционировать и упорядочивать **одновременно**, однако в данном случае оператор ORDER BY будет определять порядок следования записей в рамках секции, а не всего окна. Если определение окна включает как секционирование, так и упорядочивание, то оператор PARTITION BY должен располагаться до оператора ORDER BY.

**✅**Оконные функции обычно подразделяют на три основные группы: **ранжирующие функции, агрегатные функции** и **функции смещения**.

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

**✅ Определить окно** можно без использования оператора WINDOW. Для этого достаточно предоставить спецификацию окна, заключенную в круглые скобки, сразу после **ключевого слова OVER**.

**✅**С помощью оператора WINDOW можно определить как одно окно, так и **несколько**. Во втором случае достаточно перечислить определения всех окон через запятую.

**✅ 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.

**✅ Функции RANK()** аналогична функции DENSE\_RANK(). Отличие в том, что функция DENSE\_RANK() выполняет ранжирование без пропусков, в то время как функция RANK() при повторении рангов следующий ранг отбрасывает.

**✅ 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.

**✅ Ранжирующие функции** практически всегда используются **только с упорядоченными окнами**. Более того, в некоторых СУБД ни одну из ранжирующих функций нельзя применить к окну, если оно не упорядочено.

**✅ Оконные функции** применимы только в блоках **операторов SELECT** и **ORDER BY**.

**✅ Агрегатные оконные функции** представляют собой те же самые агрегатные функции группировки, но применяемые не к группам, а к окнам.

**✅ Функция AVG()** вычисляет среднее арифметическое числовых значений, хранящихся в определенном поле окна.

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

**✅ 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, хранящееся в определенном поле окна, **функция MAX()** — максимальное.

**✅**Спецификация **определения границ** **окна** позволяет 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 точками, включая сами границы.

**✅** **Определить границы окна** можно с помощью **оператора ROWS**, синтаксис использования которого имеет следующий вид:

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-я запись перед текущей
* n FOLLOWING — n-я 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-я запись перед текущей
* n FOLLOWING — n-я запись после текущей
* UNBOUNDED FOLLOWING — самая последняя запись окна (или секции окна, если было применено секционирование)

**✅ Определить границы окна** также можно с помощью **оператора RANGE**, синтаксис использования которого имеет следующий вид:

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

Как можно заметить, применяется оператор RANGE практически так же, как и оператор ROWS. Более того, для указания граничных точек используются те же ключевые слова CURRENT ROW, PRECEDING и FOLLOWING. 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**.

**✅**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).

**✅ 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.

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

**✅**Функции FIRST\_VALUE(), LAST\_VALUE() и NTH\_VALUE(), называемые **функциями смещения**, практически всегда используются **только с упорядоченными окнами**.

**✅ Функция LAG()** используется для получения значения, которое содержится в определенном поле записи окна, отстающей от текущей на n. Например, запись окна, отстающая от текущей на 1, считается предыдущей, на 2 — предпредыдущей и так далее.

**✅ Функция LEAD()** используется для получения значения, которое содержится в определенном поле записи окна, опережающей текущую на n. Например, запись окна, опережающая текущую на 1, считается следующей, на 2 — следующей за следующей и так далее.

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

**✅ Функция REGEXP\_LIKE()** используется для того, чтобы узнать, есть ли совпадение с шаблоном регулярного выражения в строке. Она 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.

**✅ 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)
* occurrence — 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.

**✅ 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)
* occurrence — 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.

**✅ 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)
* occurrence — 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.