СУБД — система управления базами данных

СУБД позволяет вам выполнять поиск по данным в базе, добавление, удаление и другие операции.

Реляционные базы данных используются там, где данные могут быть представлены в табличном виде, и база данных в такой системе представляет собой набор таблиц, связей между ними и операций с данными таблицами

каждая база данных имеет свое собственное имя. Также каждая таблица имеет собственное имя.

Каждому платежу соответствует свой номер, и повторяться они не могут. Подобное значение в базах данных обычно называется первичным ключом. Первичный ключ часто используется для того, чтобы можно было сослаться на конкретную запись в данной таблице.

Первичный ключ может состоять не из одного столбца, а из нескольких.Уникальной является их комбинация.

**CHAR** или **CHARACTER**— строковый тип данных с фиксированной длинной. Это значит, что при хранении в базе все строки в столбце будут занимать один и тот же объем. Это хорошо для быстродействия, но увеличивает объем базы.  
**VARCHAR** — строковые данные переменной длины. При использовании этого типа, каждая строка в столбце занимает столько места, сколько необходимо для хранения положенной в нее информации.  
**DATE** — дата  
**DATETIME** — дата и время  
**TEXT** (либо **BLOB**) — большой объем информации. В MySQL, как и в некоторых других СУБД, типы **CHAR** и **VARCHAR** имеют ограничение — максимум 255 символов в строке. В случаях, если текста должно быть больше, используется тип **TEXT**. Тип **BLOB** используется, если необходимо хранить двоичную информацию.(хранит текст)

Для многих типов данных можно указать их размер, поместив после типа круглые скобки с одним или несколькими числами, задающими размер данных. Так для **DECIMAL** можно указать два числа — общее количество цифр в числе, и количество после запятой: **DECIMAL(9,3)** означает что в числе может быть шесть знаков до, и три знака после запятой. Если не указать размеры, то **DECIMAL**, не смотря на то, что это вещественный тип будет всегда иметь дробную часть нулевой длины.

Для типов **CHAR** **VARCHAR** указывать размер обязательно. Для **VARCHAR** это будет максимальный размер хранимой строки, для **CHAR** это размер всех строк в этой колонке. Причем, как уже было сказано, в MySQL существует ограничение до 255 символов.

**CREATE DATABASE имя\_базы; (нижний регистр/как в програм, без пробелов )**

**USE имя\_базы; - использовать эту БД**

**CREATE TABLE имя\_таблицы(имя\_поля тип, имя\_поля тип.....);**

SHOW databases; - оказывает все БД;

Пример:

**CREATE**

**TABLE**

**expenses(**

**num INT,**

**paydate DATE,**

**receiver VARCHAR(30),**

**value DEC);**

или

**CREATE TABLE receivers(**

**num INT,**

**name VARCHAR(255));**

DESC имя табл - выводит структуру таблицы.(строки и столбцы)

Если необходимо создать первичный ключ, в MySQL, при создании таблицы он указывается в списке полей, обычно через запятую после последнего поля в виде **PRIMARY KEY (имя поля)**. После скобок никаких типов данных, и других настроек, которые могут применяться для обычных полей для создания первичного ключа указывать не надо.

В таблице некоторые ячейки могут быть пустыми. в этом случае в них заносится специальное значение **NULL**.

Если вы хотите запретить пустые значения в столбце, при создании таблицы следует после типа поля указать **NOT NULL**

**CREATE TABLE receivers(**

**num INT,**

**name VARCHAR(255) NOT NULL,**

**PRIMARY KEY(num));**

**receiver INT NOT NULL DEFAULT 1,**

Если по какому-нибудь полю часто будет выполняться поиск его можно сделать индексированным, указав **INDEX(имя\_поля)** в конце списка полей таблицы (аналогично с PRIMARY KEY). Это ускорит операции поиска и выборки данных по данному полю. Индексу можно дать собственное имя написав его между **INDEX** и скобками. Например, если вы решили сделать в таблице получателей имя получателя индексированным:

**CREATE TABLE receivers(**

**num INT,**

**name VARCHAR(255) NOT NULL,**

**PRIMARY KEY(num), INDEX namesindex (name));**

Если вам необходимо запретить повторы значений, т.е. все значения в столбце должны быть уникальными, это можно сделать для индексов. при объявлении следует указать **UNIQUE(имя\_индекса)**, аналогично поместив его в конце списка полей при объявлении таблицы.

**Добавление запис**и в таблицу осуществляется с помощью команды **INSERT INTO.**Существуют две формы использования данного запроса. Рассмотрим первую, более простую:

**INSERT INTO имя\_таблицы VALUES(значение1, значение2, значение3...); - для поля1, поля 2 и т.д - если текст то в ‘’**

Данная форма используется, если количество вносимых в запись данных равно количеству полей таблицы. При этом порядок значений должен соответствовать порядку полей в записи таблицы. Например :

**INSERT INTO expenses VALUES(1,'2011-5-10',1,2000.0);**

Следует помнить, что строковые данные вставляются в запрос в одинарных кавычках. Как видно из примера, для дат также используются кавычки. Причем, обязательно соблюдать последовательность и всегда вводить дату в этом виде:"год-месяц-день".

Второй вариант позволяет добавлять запись с меньшим количеством значений, или с другим порядком следования. Его общий вид:

**INSERT INTO имя\_таблицы (имя\_поля1, имя\_поля2...)**

**VALUES(значение1, значение2...);**

Он отличается тем, что перед **VALUES** добавляются скобки со списком имен полей таблицы, в этом случае количество параметров в обоих скобках должно совпадать, и порядок данный в скобках **VALUES** должен совпадать не с порядком полей в таблице, а с порядком, в котором они указаны в первых скобках. Например, запрос в котором переставлены величина платежа и номер получателя:

**INSERT INTO expenses (num,paydate,value,receiver)**

**VALUES(2,'2011-5-10',94200.0,2);**

Если в подобном запросе перечислены не все поля, которые есть в таблице, недостающие поля получат **NULL** или значение по-умолчанию, если был задан **DEFAULT**. Если значения по умолчанию не было задано, и при этом стоит **NOT NULL**, вы получите синтаксическую ошибку.

После ключевого слова **SELECT** могут следовать не только имена полей, но и выражения с ними. SQL содержит все основные арифметические операции: **+ - / \*** и, например, если вас интересует одна четвертая величины платежа, запрос может иметь вид: **SELECT num,paydate,value/4,receiver FROM expenses;**

**ID INT AUTO\_INCREMENT - сам ставит номер за тебя, забудь про него.**

**ORDER BY** и имя поля, по которому будет выполняться упорядочивание. Например, чтобы получить результат, упорядоченный по сумме денег:

**SELECT \* FROM expenses ORDER BY value;**

По умолчанию сортировка происходит по возрастанию — от меньших к большим. Если необходимо сменить направление, в конце добавляется слово **DESC**, и сортировка будет происходить от большего к меньшему т.е. по убыванию.

**SELECT \* FROM expenses ORDER BY value DESC;**

Запрос к 2 таблицам

**SELECT paydate,value,name FROM expenses,receivers WHERE receiver=receivers.num;**

Очень важным здесь является условие в **WHERE**. Если его не поставить, запрос будет синтаксически правильным, не вызовет никаких сообщений об ошибках, но при этом его результаты могут удивить. Дело в том, что запрос, выполняемый по нескольким таблицам без дополнительных условий, делает комбинации всех возможных строк участвующих в запросе таблиц. И если, мы имеем две таблицы, в одной из которых три строки, а в другой четыре, в результате запроса будет выдано двенадцать строк.

Математические функции:

Запись функции представляет собой: **ИМЯ(аргументы)**

**Например ---SELECT UPPER(name) FROM RECEIVERS;**

|  |  |
| --- | --- |
| **Имя функции** | **Описание** |
| **SIN() COS() TAN()** | Основные тригонометрические функции. На вход получают либо числовое поле, либо выражение с числовым результатом |
| **ROUND()** | Получает из поля округленное значение, по правилам. |
| **FLOOR()** | Получает из поля округленное значение в меньшую сторону. |
| **CEIL()** | Получает из поля округленное значение в большую сторону. |
| **POW(аргумент,аргумент)** | возводит первый аргумент, в степень равную второму аргументу. Аргументы — числовые поля, или выражения с числовым результатом. |
| **ABS()** | Возвращает модуль числа, переданного в качестве аргумента. |
| **SQRT()** | возвращает квадратный корень аргумента |
| **LOWER()** | Преобразует строку в нижний регистр |
| **UPPER()** | Преобразует строку в верхний регистр |
| **LENGTH()** | возвращает длину строки |
| **LTRIM()** | обрезает пробелы в начале строки |
| **RTRIM()** | обрезает пробелы в конце строки |
| **REPLACE()** | Функция поиска и замены. Имеет три аргумента — исходная строка, что ищем, чем заменяем. Все три аргумента — строковые. |
| **SUBSTRING()** | Функция получения части строки. Первый аргумент — исходная строка, второй — позиция, с которой начинаем извлекать фрагмент, третий — длина фрагмента. |

#### **Функции работы с датами**

|  |  |
| --- | --- |
| **Имя функции** | **Описание** |
| **NOW()** | Получение текущей даты и времени из системных часов |
| **DAY()** | Получение номера дня из даты |
| **MONTH()** | Получение номера месяца из даты |
| **YEAR()** | Получение года из даты |

**SELECT \* FROM expenses WHERE MONTH(paydate) = 12;**

**DELETE FROM имя\_таблицы WHERE условие;**

вносить изменения в существующие записи. Для таких случаев используется команда **UPDATE**

**UPDATE имя\_таблицы SET имя\_столбца=значение, имя\_столбца=значение...**

**WHERE условие;**

команда **DELETE** ,если не поставить условие, удаляет все данные из таблицы но не саму таблицу. Чтобы удалить таблицу полностью, необходимо воспользоваться операцией **DROP TABLE** вида:

**DROP TABLE имя\_таблицы;**

Можно удалить не только отдельную таблицу, но и всю базу данных целиком

**DROP DATABASE имя базы;**

Например, вам понадобилось добавить новую колонку в таблицу получателей с адресом получателя. Команда будет выглядеть следующим образом:

**ALTER TABLE receivers ADD address VARCHAR(255);**

После **ALTER TABLE** обычно следует имя таблицы, над которой выполняется операция, дальше имя операции, в нашем случае **ADD** — добавление, и данные для выполняемой операции, в нашем случае описание нового поля таблицы.

Через некоторое время вы решили, что адрес вам не нужен, и необходимо удалить этот столбец, для этого в команде **ALTER TABLE** используется операция **DROP**:

**ALTER TABLE receivers DROP address;**

Эта команда удалит колонку **address** и все данные в ней находящиеся.

Alter table my\_table - изменить таблицу

Add column id int first(second…/after + указать имя столбца, после которого надо вставить) - добавить столбец ID c типом int, поместить на первое место

Add primary key (id) - ключ в конце ставится

Alter table my\_table Drop column имя\_столбца - удалить столбец

-//- rename to новое\_имя\_таблицы - переименование таблицы

-//- change column имя\_столбца новое\_имя\_столбца varchart(15) not null - переименование столбца

-//- modify column имя\_столбца изменяем\_тип; -изменение типа столбца

Правильной последовательностью соблюдения нормальных форм обычно называют следующую:

1. Первая нормальная форма
2. Вторая нормальная форма
3. Третья нормальная форма
4. Нормальная форма Бойса-Кодда
5. Четвертая нормальная форма
6. Пятая нормальная форма

Как уже было сказано, в большинстве случаев при разработке баз данных следуют первым трем нормальным формам, т.к. полное следование всем нормальным формам вплоть до пятой, приводит к излишнему усложнению структуры базы данных, усложнению запросов к этой базе, часто к существенной потере производительности, а иногда и росту объема базы.

Первая нормальная форма требует, чтобы каждый элемент таблицы имел только одно значение.

**PRIMARY KEY(имя поля)** — Это будет означать что указанная в скобках колонка будет первичным ключом.

Но первичный ключ может состоять из нескольких колонок. Для этого в скобках через запятую можно указать несколько названий полей.

Одно из главных требований к содержимому первичного ключа: значения не должны повторяться. Если первичный ключ состоит из нескольких колонок, значения в каждой из этих колонок повторяться могут. Не может повторяться полное сочетание значений этих колонок в данной строке.

Вторая нормальная форма требует, чтобы все поля зависели от первичного ключа, а не от его какой-то части.

Третья нормальная форма требует, чтобы каждый не ключевой элемент таблицы зависел только от ключа, и ни от каких других элементов таблицы.

Псевдонимы или «алиасы» таблиц позволяют дать таблице, полю или другому элементу запроса новое имя.

В частности, псевдоним можно дать таблице из которой получаем данные. Для этого в блоке **FROM** после имени таблицы через пробел указывается псевдоним.

**SELECT paydate,value,name FROM expences, reseivers rs WHERE receiver=rs.num**

В данном запросе таблица **receivers** получила псевдоним **rs**, и мы в условии обращаемся к таблице не по полному имени, а по псевдониму.

Можно также давать псевдонимы полям в разделе **SELECT** , как впрочем и другим элементам запроса, для этого после поля пишется **AS** и новое имя. Также это часто используют, если в **SELECT** присутствует не просто поле, а какое-либо выражение с использованием операторов и функций:

**SELECT value/2 AS Половина, paydate ....**

В данном примере будет выведена величина платежа деленная на два, и в заголовке таблицы будет название **Половина**.

Чтобы такие платежи были показаны в единственном экземпляре, в запрос добавляется слово **DISTINCT**

**SELECT DISTINCT value**

Чтобы вывести не все записи результата, а только часть следует воспользоваться ключевым словом **LIMIT** после которого указывается номер строки, с которой начинается вывод и количество выводимых строк.

**SELECT paydate,value,name FROM expenses, receivers rs WHERE receiver=rs.num LIMIT 0,5**

В результате, будет показано только 5 записей начиная с нулевой, т.е. с начала таблицы.

Оператор **LIKE** используется следующим образом: **строка LIKE шаблон**, строкой может быть любое поле хранящее текстовую информацию, или результат функции работающей со строками. Шаблон — представляет собой текст, со специальными знаками. Большинство СУБД воспринимают два таких знака:  
**%** — означает любое количество любых символов;  
**\_** — означает один любой символ.

Например, если нам необходимо распечатать список получателей, которые являются гипермаркетами, мы можем написать следующий запрос:

**SELECT num,name FROM receivers WHERE name LIKE 'Гипермаркет%';**

Такой запрос выдаст список всех получателей, названия которых начинаются с **Гипермаркет**

Если в шаблоне нужны сами символы **% \_**, то их следует экранировать, то есть записывать в виде: **\% \\_**

Но есть группа функций, характерных именно для баз данных — агрегатные функции. Они работают не с одним значением, а со множеством, получая их из разных строк таблицы.

Типичной агрегатной функцией является **COUNT()**, функция осуществляющая подсчет строк. Например:

**SELECT COUNT(\*) FROM expenses**

В качестве аргумента можно указать имя поля (например **COUNT(paydate)**), в этом случае будет подсчитано количество не пустых значений(т.е. в которых содержится не **NULL**).

Еще один вариант использования: **COUNT(DISTINCT имя\_поля)** считает кол-во уникальных значений в указанном столбце. Например, вам нужно узнать, скольким получателям вы делали платежи в рассматриваемой нами базе расходов.

Функция **MIN(имя\_поля)** ищет минимальное значение, функция **MAX(имя\_поля)** ищет максимальное значение.

Кроме полей для функций **MIN** и **MAX** аргументами могут служить выражения. Например **MINT(SQRT(VALUE))** будет искать минимальное значение квадратного корня из платежа.

Чтобы просуммировать значения в столбце, используется агрегатная функция **SUM(имя\_поля)**. Например, вам необходимо узнать, общую сумму платежей за 2012 год. Запрос будет иметь вид:

**SELECT SUM(value) FROM expenses WHERE YEAR(paydate) = 2012;**

функция для среднего значения

**SELECT AVG(value) FROM expenses;**

При использовании **AVG** важно помнить особенности его взаимодействия с пустыми полями. Пустые поля(то есть содержащие **NULL**) игнорируются.

Группировка выполняется с помощью ключевых слов **GROUP BY имя\_поля** в конце запроса, и позволяет объединить несколько строк, у которых одинаковое значение в одном поле.

**SELECT COUNT(\*),name FROM expenses, receivers rs WHERE receiver=rs.num GROUP BY name**

Здесь сначала GROUP BY name сгруппирует все строки с одинаковыми именами. А затем, для каждой из этих групп, COUNT(\*) подсчитает количество строк в группе. В результате будет выдана таблица в первой колонке которой стоит количество платежей получателя, а во второй — имя этого получателя.

вложенным запросом или подзапросом.

**SELECT \* FROM expenses WHERE value = (SELECT MAX(value) FROM expenses);**

**~~SELECT \* FROM expenses WHERE value = MAX(value);~~**

Подзапросы могут использоваться не только в WHERE, они могут находится практически в любой части основного запроса, в том числе и в разделе SELECT. Например:

**SELECT name, (SELECT count(\*) FROM expenses WHERE receiver=r.num)**

**FROM receivers r**

Что важно, в данном запросе для таблицы **receivers** обязательно использовать псевдоним **r** и именно к нему обращаться в подзапросе.

Иногда, в условии необходимо какое-то поле сравнить на равенство не с одним значением, а с несколькими. Для этого можно воспользоваться логическими операторами, но в SQL есть специальный оператор IN. Его использование в этом случае выглядит так:

**имяПоля IN (список\_значений)**

**SELECT \* FROM expenses WHERE paydate IN (SELECT paydate FROM expenses WHERE value =**

**(SELECT MAX(value) FROM expenses));**

**ELECT MAX(sumValue) FROM (SELECT paydate, SUM(VALUE) AS sumValue**

**FROM expenses GROUP BY paydate) AS sums;**

Если вы делаете подзапрос во FROM, обязательно следует дать ему псевдоним, иначе получите синтаксическую ошибку.

Update таблица set поле= case when поле=значение

Then значение....

Else значение (не совпадает с никаким значением)

End;

Primary Key - (он же - первичный ключ) - предоставляет ссылку для связи с другими таблицами и создаёт ограничение уникальности для столбца, на котором задаётся.

Первичный ключ имеет пару особенностей, которые необходимо знать для его успешного применения:

* Primary Key может быть исключительно один. Несколько первичных ключей сделать невозможно, т.к. он, своего рода, уникален.
* Первичный ключ может быть простым и составным, со всеми вытекающими последствиями)

primary key(CustomerPhone, CustomerEmail) -- пример составного первичного ключа.

Как видим, применение составного первичного ключа гарантирует уникальность только пары значений (!!!), тобишь правдивость будет подтверждена только если оба значения в одно время будут уникальными.

Foreign Key нужен для того, что бы предоставить определённую ссылку для связи между двумя таблицами.

**CREATE** **TABLE** orders (

    order\_id **INT** AUTO\_INCREMENT **PRIMARY** **KEY**,

    customer\_id **INT**,

    amount **DOUBLE**,

**FOREIGN** **KEY** (customer\_id) **REFERENCES** customers(customer\_id)

Или через alter после создания таблицы

);

Join вширь, union в длину.