**mysql -uroot -p12345**

**CREATE DATABASE ListExpenses;**

**USE ListExpenses; CREATE TABLE expenses(num INT, paydate DATE, receiver INT,value DEC, PRIMARY KEY (num));**

USE ListExpenses;CREATE TABLE receivers(num INT, name VARCHAR(255));

**INSERT INTO expenses VALUES(10,'2011-9-13',1,3000.0);**

**INSERT INTO** receivers **VALUES(4,'Butterfly');**

**SELECT \* FROM expenses ORDER** **BY volume WHERE DAY(paydate) = 31 ;**

**SELECT \* FROM expenses,receivers WHERE receiver=receivers.num ORDER** **BY paydate;**

**Основные типы данных**

Возможные типы полей базы данных могут существенно отличаться в разных СУБД, но существует общий набор, который встречается практически везде:  
  
**INT** или **INTEGER** — целочисленные данные;  
**DEC** или **DECIMAL** — десятичные дробные величины;  
**FLOAT** — еще один тип для десятичных дробных величин;  
**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.

Несмотря на то что существует стандарт SQL, названия и наборы используемых данных могут очень сильно отличаться в разных СУБД.

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)**. Это означает, что в числе может быть шесть знаков до и три знака после запятой. Если не указать размеры, то **DECIMAL**, несмотря 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.

**Создание таблиц**

После создания базы необходимо создать таблицы для хранения информации. Для этого используется команда:

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

Количество полей, перечисленных через запятую, может быть любым. Например:

**CREATE TABLE expenses(num INT, paydate DATE, receiver INT,value DEC);**

Следует обратить внимание на то, что при использовании консоли запрос может быть многострочным. Необязательно, чтобы запрос был в одну строку. Таким образом, была создана таблица расходов. Команда для создания таблицы получателей будет иметь вид:

**CREATE TABLE receivers(num INT, name VARCHAR(255));**

Напоминаем, что некоторые типы, такие, как **CHAR** и **VARCHAR**, должны иметь в скобках дополнительный параметр, указывающий их размер. Указание правильного размера поля таблицы может значительно сэкономить занимаемую ею память.

Если при создании таблицы необходимо создать первичный ключ в MySQL, 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.

**Уникальность значений, начальные значения и пустые поля**

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**

Команда для создания таблицы получателей платежей, в результате может принять вид:

**CREATE TABLE receivers(num INT,name VARCHAR(255) NOT NULL,**

**PRIMARY KEY(num));**

Таким образом, создается таблица, в которой поле **num** является первичным ключом и поле **name** не может быть пустым.

Далее мы будем рассматривать заполнение таблицы новыми записями. И если при добавлении записи не были указаны данные для поля, у вас есть возможность задать значение по умолчанию, которое будет использоваться в этом случае. Для этого используется ключевое слово **DEFAULT**, после которого ставится нужное значение.

Приведенные настройки ставятся после описания типа данных в таблице и могут свободно комбинироваться:

**CREATE TABLE**

**expenses(**

**num INT,**

**paydate DATE DEFAULT DATE(),**

**receiver INT NOT NULL DEFAULT 1,**

**value DEC(10,2) NOT NULL,**

**PRIMARY KEY(num));**

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 данных в таблицу

Добавление записи 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...);**

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.

**Знакомство с оператором SELECT.**

Наиболее часто выполняемой задачей является получение или т.н. выборка из базы данных. Она выполняется с помощью команды **SELECT**. В простейшем варианте данная операция выглядит следующим образом:

**SELECT имена\_полей FROM имя таблицы;**

для нашей таблицы расходов:

**SELECT num,paydate,value,receiver FROM expenses;**

Если выбираются все поля таблицы, их можно заменить знаком звездочки:

**SELECT \* FROM expenses;**

В результате такого запроса пользователь получит всю таблицу **expenses** целиком. Поля в таблице будут расставлены в том же порядке,в котором объявлялись при создании таблицы.

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;**

**Получение данных по условию**

Только что мы увидели, как получить содержимое всех строк таблицы.

Но куда чаще появляется необходимость получить часть таблицы, которая удовлетворяет определенному условию. Например, нас интересуют только расходы от **20000**и выше. В этом случае к запросу добавляется условие с помощью ключевого слова **WHERE**. Например:

**SELECT \* FROM expenses WHERE value >= 20000;**

В качестве операторов условия могут использоваться такие же операторы, как в Си-подобных языках программирования (Java, C#, JavaScript, PHP), за двумя исключениями: равенство обозначается одинарным знаком равно, а неравенство — **<>**. Также в выражениях могут использоваться логические операторы, которые записываются: **and** для логического **и**, и **or** для логического **или**. Например, нас интересуют только расходы в гипермаркете «Корона», которые выше 100000. В этом случае запрос можно записать так:

**SELECT \* FROM expenses WHERE receiver = 2 and value > 100000;**

Как и в других языках программирования, в условиях допустимо использование круглых скобок.

В условиях также при необходимости можно писать выражения со значениями полей.

Условие можно ставить по любому из полей таблицы, в т.ч. по тем, которые отсутствуют в списке после **SELECT**, т.е. которые не передаются пользователю.

орядке хранятся строки в таблицах базы. СУБД в ответ на ваш запрос может их выдать в любом порядке.

Но очень часто полученный результат нужно упорядочить. Для этого следует к запросу добавить ключевые слова **ORDER BY** и имя поля, по которому будет выполняться упорядочивание. Например, чтобы получить результат, упорядоченный по сумме денег:

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

Если необходима сортировка сразу по нескольким полям, их можно перечислить через запятую. В этом случае, если в нескольких строках значения в первом поле будут одинаковые, между собой эти строки будут упорядочиваться по второму и т.д. Например, сортировка по дате и номеру платежа:

**SELECT \* FROM expenses ORDER BY paydate,num;**

Важно понимать, что упорядочивание никак не влияет на порядок строк в базе. Т.е. это не сортировка базы данных. Упорядочивание влияет только на то, в каком порядке пользователь получает записи.

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

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

**Запрос по нескольким таблицам одновременно**

Запрос может выполняться не только к одной, но и к нескольким таблицам одновременно. В этом случае после ключевого слова **FROM** следует указать названия таблиц через запятую.

Например, необходимо получить данные о расходах не с кодами, а с именами получателей платежей. Т.к. данные о платежах и именах получателей находятся в разных таблицах, это будет запрос по двум таблицам. Запрос в этой ситуации будет выглядеть так :

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

В данном случае мы запросили поля даты платежа, величины и имени получателя из таблиц платежей и получателей.

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*.

Условие **WHERE** в данном случае служит для связки этих двух таблиц, чтобы в результирующем запросе оставались только необходимые нам строки. Это строки, где номер получателя из одной таблицы, равен номеру получателя из другой.

Также следует обратить внимание на обращение к полю **receivers.num**. Оно записано таким образом, потому что поля **num** есть в обеих таблицах. Это позволяет избежать путаницы и указать, поле из какой таблицы нас в данном случае интересует.

Составляя запрос, вы можете указать имя таблицы каждый раз при обращении к полю: **SELECT expenses.num, expenses.value,expenses.receiver FROM expenses;**. Но обязательным такая запись является только для запросов, где могут быть совпадающие имена полей.

**Дополнительная информация. Функции.**

Как уже было сказано, в запросах можно использовать не только имена полей, но и выражения с этими полями с использованием основных операторов и функций. MySQL предлагает достаточно обширный набор функций, с некоторыми из них мы сейчас познакомимся.

Запись функции представляет собой: **ИМЯ(аргументы)**, после имени обязательно следуют круглые скобки, в которых функция может получить исходные данные для своей работы: аргументы. Если их несколько, они перечисляются через запятую. Аргументами могут служить поля таблицы, если их тип совпадает с требуемым, или выражения.

Стандарты SQL, в частности один из базовых стандартов SQL92, не содержат многих видов функций, например, математических. Поэтому разработчики СУБД создают собственные наборы таких функций, которые могут существенно друг от друга отличаться. Здесь мы постараемся привести наиболее часто используемые функции, которые одинаково работают в большинстве разных СУБД.

Математические функции:

|  |  |
| --- | --- |
| **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. |

### Строковые функции

Функции для работы со строками:

|  |  |
| --- | --- |
| **Имя функции** | **Описание** |
| **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. |

Пример использования строковых функций. Например, мы хотим из нашей базы расходов распечатать всех получателей так, чтобы их имена были выведены верхним регистром. Запрос будет иметь вид:

**SELECT num,UPPER(name) FROM RECEIVERS;**

#### 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 |

Пример использования функций работы с датой. Допустим, нам необходимо вывести платежи, которые были сделаны в декабре (любого года). Запрос будет иметь вид:

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

**select** \* **from** Customers

**where** City **IN** ('London', 'Berlin')

**select** \* **from** Customers

**where** City **NOT** **IN** ('Madrid', 'Berlin','Bern')

Фильтрация по нескольким условиям с применением AND (выполняются все условия) или OR (выполняется хотя бы одно условие) и нескольким значениям:

**select** \* **from** Customers

**where** Country = 'Germany' **AND** City **not** **in** ('Berlin', 'Aachen') **AND** CustomerID > 15

**select** \* **from** Customers

**where** City **in** ('London', 'Berlin') **OR** CustomerID > 4

При использовании GROUP BY обязательно:

1. перечень столбцов, по которым делается разрез, был одинаковым внутри SELECT и внутри GROUP BY,
2. агрегатные функции (SUM, AVG, COUNT, MAX, MIN) должны быть также указаны внутри SELECT с указанием столбца, к которому такая функция применяется.

Группировка количества клиентов по городу:

**select** City, count(CustomerID) **from** Customers

**GROUP** **BY** City

Группировка количества клиентов по стране и городу:

**select** Country, City, count(CustomerID) **from** Customers

**GROUP** **BY** Country, City

Группировка продаж по ID товара с разными агрегатными функциями: количество заказов с данным товаром и количество проданных штук товара:

**select** ProductID, COUNT(OrderID), SUM(Quantity) **from** OrderDetails

**GROUP** **BY** ProductID

Группировка продаж с фильтрацией исходной таблицы. В данном случае на выходе будет таблица с количеством клиентов по городам Германии:

**select** City, count(CustomerID) **from** Customers

**WHERE** Country = 'Germany'

**GROUP** **BY** City

Переименование столбца с агрегацией с помощью оператора AS. По умолчанию название столбца с агрегацией равно примененной агрегатной функции, что далее может быть не очень удобно для восприятия.

**select** City, count(CustomerID) **AS** Number\_of\_clients **from** Customers

**group** **by** City

#### **HAVING**

HAVING — необязательный элемент запроса, который отвечает за фильтрацию на уровне сгруппированных данных (по сути, WHERE, но только на уровень выше).  
  
Фильтрация агрегированной таблицы с количеством клиентов по городам, в данном случае оставляем в выгрузке только те города, в которых не менее 5 клиентов:

**select** City, count(CustomerID) **from** Customers

**group** **by** City

**HAVING** count(CustomerID) >= 5

В случае с переименованным столбцом внутри HAVING можно указать как и саму агрегирующую конструкцию count(CustomerID), так и новое название столбца number\_of\_clients:

**select** City, count(CustomerID) **as** number\_of\_clients **from** Customers

**group** **by** City

**HAVING** number\_of\_clients >= 5

Пример запроса, содержащего WHERE и HAVING. В данном запросе сначала фильтруется исходная таблица по пользователям, рассчитывается количество клиентов по городам и остаются только те города, где количество клиентов не менее 5:

**select** City, count(CustomerID) **as** number\_of\_clients **from** Customers

**WHERE** CustomerName **not** **in** ('Around the Horn','Drachenblut Delikatessend')

**group** **by** City

**HAVING** number\_of\_clients >= 5

#### **ORDER BY**

ORDER BY — необязательный элемент запроса, который отвечает за сортировку таблицы.  
  
Простой пример сортировки по одному столбцу. В данном запросе осуществляется сортировка по городу, который указал клиент:

**select** \* **from** Customers

**ORDER** **BY** City

Осуществлять сортировку можно и по нескольким столбцам, в этом случае сортировка происходит по порядку указанных столбцов:

**select** \* **from** Customers

**ORDER** **BY** Country, City

По умолчанию сортировка происходит по возрастанию для чисел и в алфавитном порядке для текстовых значений. Если нужна обратная сортировка, то в конструкции ORDER BY после названия столбца надо добавить DESC:

**select** \* **from** Customers

**order** **by** CustomerID **DESC**

Обратная сортировка по одному столбцу и сортировка по умолчанию по второму:

**select** \* **from** Customers

**order** **by** Country **DESC**, City

#### **JOIN**

JOIN — необязательный элемент, используется для объединения таблиц по ключу, который присутствует в обеих таблицах. Перед ключом ставится оператор ON.  
  
Запрос, в котором соединяем таблицы Order и Customer по ключу CustomerID, при этом перед названиям столбца ключа добавляется название таблицы через точку:

**select** \* **from** Orders

**JOIN** Customers **ON** Orders.CustomerID = Customers.CustomerID

Нередко может возникать ситуация, когда надо промэппить одну таблицу значениями из другой. В зависимости от задачи, могут использоваться разные типы присоединений. INNER JOIN — пересечение, RIGHT/LEFT JOIN для мэппинга одной таблицы знаениями из другой,

**select** \* **from** Orders

**join** Customers **on** Orders.CustomerID = Customers.CustomerID

**where** Customers.CustomerID >10

Внутри всего запроса JOIN встраивается после элемента from до элемента where, пример запроса: