ОСНОВЫ БАЗ ДАННЫХ

ОБУЧАЮЩИЙ МАТЕРИАЛ

Негматов Б.М | Консультант по Программированию и управлению БД | 16 апреля 2019 г.

Содержание

1. Основы баз данных

1.1. Понятие базы данных

Несмотря на то, что мы постоянно используем базы данных, для многих остается непонятным, что же это такое на самом деле. И связано это отчасти с тем, что одни и те же термины, относящиеся к базам данных, используются людьми для определения совершенно разных вещей.

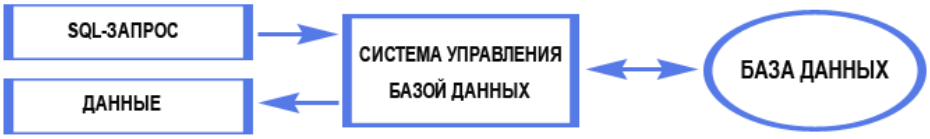
Давайте разберемся с терминами и понятиями баз данных:

База данных - набор сведений, хранящихся некоторым упорядоченным способом. Можно сравнить базу данных со шкафом, в котором хранятся документы. Иными словами, база данных - это хранилище данных. Сами по себе базы данных не представляли бы интереса, если бы не было систем управления базами данных (СУБД).

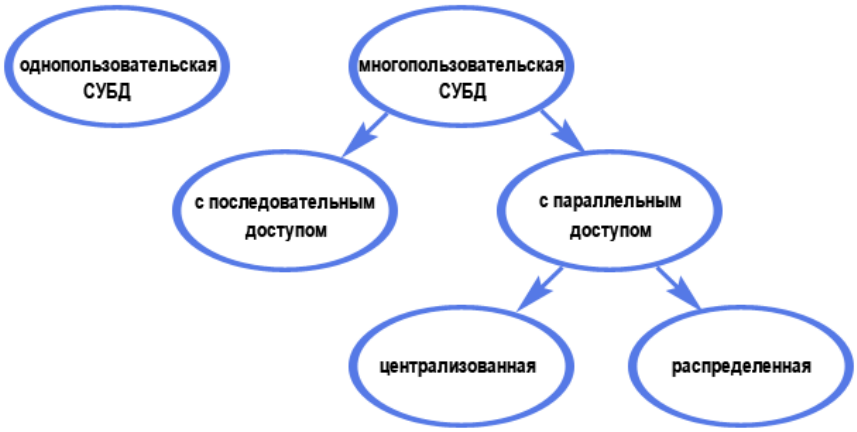
Система управления базами данных - это совокупность языковых и программных средств, которая осуществляет доступ к данным, позволяет их создавать, менять и удалять, обеспечивает безопасность данных и т.д. В общем СУБД - это система, позволяющая создавать базы данных и манипулировать сведениями из них. А осуществляет этот доступ к данным СУБД посредством специального языка - SQL.

SQL - язык структурированных запросов, основной задачей которого является предоставление простого способа считывания и записи информации в базу данных.

Итак, простейшая схема работы с базой данных выглядит примерно так:



По характеру использования СУБД делят на однопользовательские (предназначенные для создания и использования БД на персональном компьютере) и многопользовательские (предназначенные для работы с единой БД нескольких компьютеров, объединенных в локальные сети). Вообще деление по характеру использования можно представить следующей схемой:



Не вдаваясь далее в подробности, отметим, что на сегодняшний день число используемых СУБД исчисляется десятками. Наиболее известные однопользовательские СУБД - Microsoft Visual FoxPro и Access, многопользовательские - MS SQL Server, Oracle и MySQL.

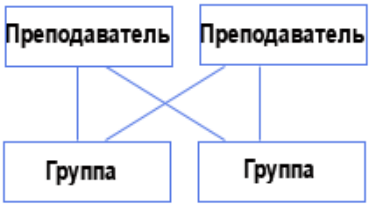
1.2. Структура базы данных

Создавая базу данных, мы стремимся упорядочить информацию по различным признакам для того, чтобы потом извлекать из нее необходимые нам данные в любом сочетании. Сделать это возможно, только если данные структурированы. Структурирование - это набор соглашений о способах представления данных. Понятно, что структурировать информацию можно по-разному. В зависимости от структуры различают иерархическую, сетевую, реляционную, объектно-ориентированную и гибридную модели баз данных. Самой популярной на сегодняшний день является реляционная структура, поэтому об остальных упомянем лишь вскользь.

Иерархическая структура базы данных

Это древовидная структура представления информации. Ее особенность в том, что каждый узел на более низком уровне имеет связь только с одним узлом на более высоком уровне. Посмотрим, например, на фрагмент иерархической структуры базы данных "Институт":

Из структуры понятно, что на одной кафедре может работать несколько преподавателей. Такая связь называется *"один ко многим"* (одна кафедра - много преподавателей). Но если мы попытаемся добавить в эту структуру группы студентов, то нам понадобится связь *"многие ко многим"*:



(один преподаватель может работать со многими группами, а одна группа может учиться у многих преподавателей), а такой связи в иерархической структуре быть не может (т.к. связь может быть только с одним узлом на более высоком уровне). Это основной недостаток подобной структуры базы данных.

Сетевая структура базы данных

По сути, это расширение иерархической структуры. Все то же самое, но существует связь *"многие ко многим"*. Сетевая структура базы данных позволяет нам добавить группы в наш пример. Недостатком сетевой модели является сложность разработки серьезных приложений.

Реляционная структура базы данных

Все данные представлены в виде простых таблиц, разбитых на строки и столбцы, на пересечении которых расположены данные. Подробно об этом мы будем говорить в следующих уроках, здесь же хочется отметить, что эта структура стала настоящим прорывом в развитии баз данных.

Объектно-ориентированные и гибридные базы данных

В объектно-ориентированных базах данных данные хранятся в виде объектов, что очень удобно. Но на сегодняшний день такие БД еще распространенны, т.к. уступают в производительности реляционным.

Гибридные БД совмещают в себе возможности реляционных и объектно-ориентированных, поэтому их часто называют объектно-реляционными. Примером такой СУБД является Oracle, начиная с восьмой версии.

Несомненно, такие БД будут развиваться в будущем, но пока первенство остается за реляционными структурами. Поэтому именно их мы и будем изучать в последующих уроках.

1.3. Реляционные базы данных

Реляционные базы данных, как мы уже знаем, состоят из таблиц. Каждая таблица состоит из столбцов (их называют *полями или атрибутами*) и строк (их называют *записями или кортежами*). Таблицы в реляционных базах данных обладают рядом свойств. Основными являются следующие:

- В таблице не может быть двух одинаковых строк. В математике таблицы, обладающие таким свойством, называют *отношениями* - по-английски relation, отсюда и название - реляционные.
- Столбцы располагаются в определенном порядке, который создается при создании таблицы. В таблице может не быть ни одной строки, но обязательно должен быть хотя бы один столбец.
- У каждого столбца есть уникальное имя (в пределах таблицы), и все значения в одном столбце имеют один тип (число, текст, дата...).
- На пересечении каждого столбца и строки может находиться только атомарное значение (одно значение, не состоящее из группы значений). Таблицы, удовлетворяющие этому условию, называют *нормализованными*.

Все будет понятнее на примере. Предположим, мы захотели создать базу данных для форума. У форума есть зарегистрированные пользователи, которые создают темы и оставляют сообщения в этих темах. Эта информация и должна храниться в базе данных.

Теоретически (на бумаге) мы можем все это расположить в одной таблице, например, так:

Имя	E-mail	Пароль	Созданные темы	Созданные сообщения

Но это противоречит свойству атомарности (одно значение в одной ячейке), а в столбцах Темы и Сообщения у нас предполагается неограниченное количество значений. Значит, нашу таблицу надо разбить на три: Пользователи, Темы и Сообщения.

Пользователи

Имя	E-mail	Пароль

Темы

Наименование	Автор

Сообщения

Текст	Автор

Наша таблица Пользователи удовлетворяет всем условиям. А вот таблицы Темы и Сообщения - нет. Ведь в таблице не может быть двух одинаковых строк, а где гарантия, что один пользователь не оставит два одинаковых сообщения, например:

Сообщения

Текст	Автор
Думаю надо сделать так…	Кирилл
Согласен	Вася
А еще можно сделать так…	Семен
Согласен	Вася

Кроме того, мы знаем, что каждое сообщение обязательно относится к какой-либо теме. А как это можно узнать из наших таблиц? Никак. Для решения этих проблем, в реляционных базах данных существуют *ключи*.

Первичный ключ (сокращенно РК - primary key) - столбец, значения которого во всех строках различны. Первичные ключи могут быть логическими (естественными) и суррогатными (искусственными). Так, для нашей таблицы Пользователи первичным ключом может стать столбец e-mail (ведь теоретически не может быть двух пользователей с одинаковым e-mail). На практике лучше использовать суррогатные ключи, т.к. их применение позволяет абстрагировать ключи от реальных данных. Кроме того, первичные ключи менять нельзя, а что если у пользователя сменится e-mail?

Суррогатный ключ представляет собой дополнительное поле в базе данных. Как правило, это порядковый номер записи (хотя вы можете задавать их на свое усмотрение, контролируя, чтобы они были уникальны). Давайте внесем поля первичных ключей в наши таблицы:

Пользователи

id пользователя	Имя	E-mail	Пароль
1	Кирилл	kirill@mail.ru	Gh345fgh
2	Вася	vasy@rambler.ru	As3bh7
3	Семен	semen@yandex.ru	gk4bb6

Темы

id темы	Наименование	Автор
1	О рыбалке	Кирилл
2	Велосипеды	Вася
3	Ночные клубы	Семен
4	О рыбалке	Вася

Сообщения

id сообшения	Текст	Автор
1	Думаю надо сделать так…	Кирилл
2	Согласен	Вася
3	А еще можно сделать так…	Семен
4	Согласен	Вася

Теперь каждая запись в наших таблицах уникальна. Нам осталось установить соответствие между темами и сообщениями в них. Делается это также при помощи первичных ключей. В таблицу сообщения мы добавим еще одно поле:

Сообщения

id сообщения	Текст	Автор	id темы
1	Думаю надо сделать так…	Кирилл	1
2	Согласен	Вася	4
3	А еще можно сделать так…	Семен	1
4	Согласен	Вася	1

Теперь понятно, что сообщение с id=2 принадлежит теме "О рыбалке" (id темы = 4), созданной Васей, а остальные сообщения принадлежать теме "О рыбалке" (id темы = 1), созданной Кириллом. Такое поле называется *внешний ключ* (сокращенно FK - foreign key). Каждое значение этого поля соответствует какому-либо первичному ключу из таблицы "Темы". Так устанавливается однозначное соответствие между сообщениями и темами, к которым они относятся.

Последний нюанс. Предположим, у нас добавился новый пользователь, и зовут его тоже Вася:

Пользователи

id пользователя	Имя	E-mail	Пароль
1	Кирилл	kirill@mail.ru	*******
2	Вася	vasy@rambler.ru	*******
3	Семен	semen@yandex.ru	*******
4	Вася	vasy@mail.ru	*******

Как мы узнаем, какой именно Вася оставил сообщения? Для этого поля автор в таблицах "Темы" и "Сообщения" мы сделаем также внешними ключами:

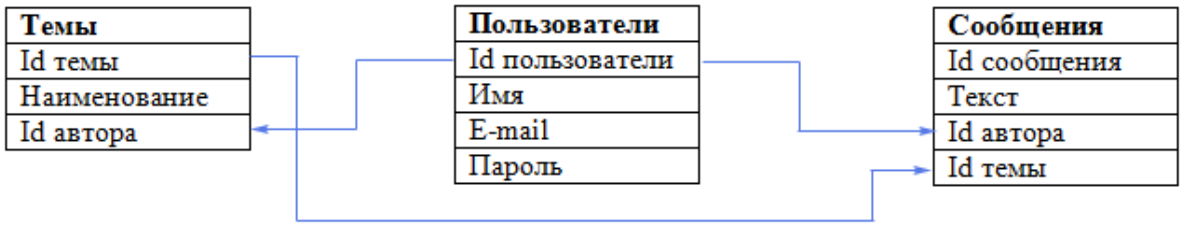
Темы

id темы	Наименование	id автора
1	О рыбалке	1
2	Велосипеды	2
3	Ночные клубы	3
4	О рыбалке	1
5	К кому обратиться	4

Сообщения

id сообщения	Текст	id автора	id темы
1	Думаю надо сделать так…	1	1
2	Согласен	2	4
3	А еще можно сделать так…	3	1
4	Согласен	2	1

Наша база данных готова. Схематично ее можно представить так:

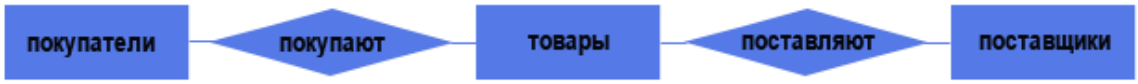


В нашей маленькой базе данных всего три таблички, а если бы их было 10 или 100? Понятно, что сразу невозможно представить все таблицы, поля и связи, которые нам могут понадобиться. Именно поэтому проектирование базы данных начинается с ее концептуальной модели, которую мы и рассмотрим в следующем уроке.

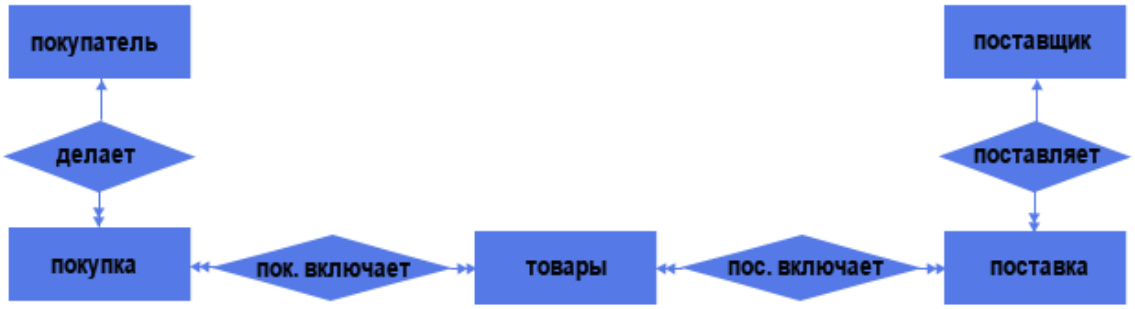
1.4. Концептуальная модель базы данных

Концептуальная модель - это отражение предметной области, для которой разрабатывается база данных. Не вдаваясь в теорию, отметим, что это некая диаграмма с принятыми обозначениями элементов. Так, все объекты, обозначающие вещи, обозначаются в виде прямоугольника. Атрибуты, характеризующие объект - в виде овала, а связи между объектами - ромбами. Мощность связи обозначаются стрелками (в направлении, где мощность равна многим - двойная стрелка, а со стороны, где она равна единице - одинарная).

Давайте в качестве примера рассмотрим интернет-магазин. У магазина есть товары, которые поставляются поставщиками и покупаются покупатели. Это можно представить тремя объектами и двумя связями:

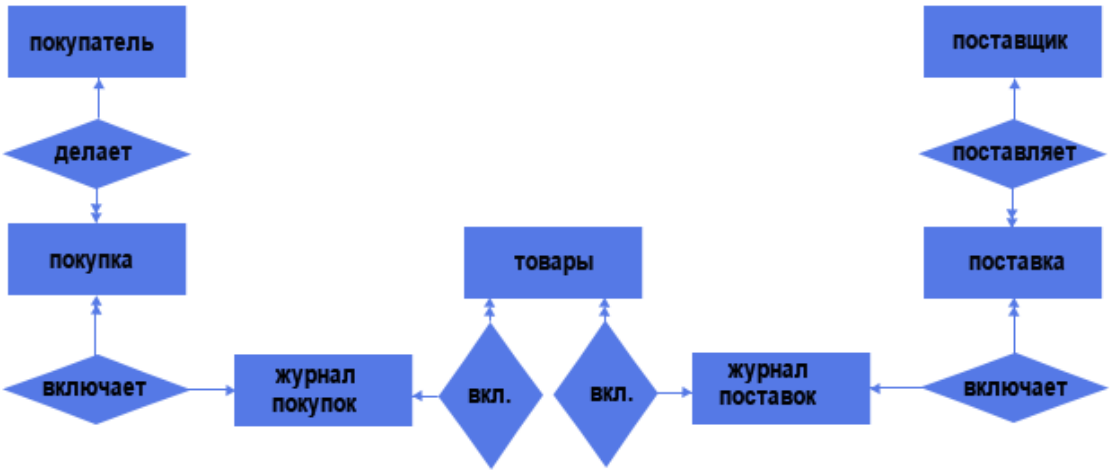


Но как поставщик поставляет товары? Он делает поставку, которая подтверждается документом. Аналогично и покупатель делает покупку, которая также может подтверждаться документом. Таким образом, поставка и покупка могут рассматриваться, как самостоятельные объекты:



Теперь у нас пять объектов и четыре связи. Две связи "один ко многим" (один поставщик может осуществить несколько поставок, но каждая поставка осуществляется только одним поставщиком, аналогично и для связи Покупатель - Покупка) и две связи "многие ко многим" (каждая поставка может содержать несколько товаров, а один и тот же товар может содержаться в нескольких поставках, аналогично и для связи Покупка - Товар).

Но связи "многие ко многим" недопустимы в реляционной модели, поэтому каждую такую связь надо заменить на две связи "один ко многим". Делается это добавлением промежуточного объекта:



Таким образом, у нас появилось еще два объекта - журнал покупок и журнал поставок, со связями "один ко многим" (один журнал поставок может включать несколько поставок, но каждая поставка может входить только в один журнал, аналогично и для остальных).

Каждый объект нашего магазина имеет свои атрибуты:

Вот собственно мы и создали концептуальную модель базы данных магазин, вернее ее части, ведь в магазине еще есть сотрудники, склады, доставка товаров и т.д.

Вообще, если предметная область обширная, то ее полезно разбить на несколько локальных предметных областей (наша концептуальная модель отражает именно локальную предметную область). Объем локальной области выбирается таким образом, чтобы в нее входило не более 6-7 объектов. После создания моделей каждой выделенной предметной области производится объединение локальных концептуальных моделей в одну общую, как правило, довольно сложную схему.

Для наших учебных целей, мы ограничимся созданной моделью. Теперь нашу концептуальную модель надо преобразовать в реляционную модель данных, т.е. в уже известные нам таблицы, поля, ключи и т.д. Этим мы и займемся на следующем уроке.

1.5. Преобразование модели в реляционную

Преобразование концептуальной модели в реляционную состоит в следующем:

- Построить набор предварительных таблиц и указать первичные ключи.
- Провести процесс нормализации.

Первый пункт мы рассматривали в третьем уроке, со вторым мы пока не знакомы, но ознакомимся на практике.

Итак, нам надо построить набор таблиц. Сделать это несложно, т.к. таблицы - это наши объекты, а поля таблиц - атрибуты объектов. Набор предварительных таблиц, исходя из нашей концептуальной модели, выглядит так:

Покупатель
Id покупателя (PK)
ФИО
E-mail

Поставщик
Id поставщика (PK)
Наименование
Адрес

Покупка
Id покупки (PK)
Id покупателя (FK)
Дата

Поставка
Id поставки (PK)
Id поставщика (FK)
Дата

Товар
Id товара (PK)
Наименование
Цена

Журнал покупок	
Id покупки (FK)	} PK
Id товара (FK)	
Количество	

Журнал поставок	
Id поставки (FK)	} PK
Id товара (FK)	
Количество	

Таким образом, у нас определены таблицы, поля, первичные ключи (PK) и связи (FK). Обратите внимание, в таблицах Журнал поставок и Журнал покупок первичные ключи - составные, т.е. состоят из двух полей. Теоретически бывают таблицы, в которых все поля являются одним составным ключом.

Переходим ко второму пункту, а именно к нормализации отношений (таблиц). *Нормализация* - это пошаговый, обратимый процесс замены исходной схемы другой схемой, в которой таблицы имеют более простую и логичную структуру. Для чего это нужно?

Во-первых, для устранения избыточности данных. Например, в нашем примере для форума (из третьего урока), мы оставили бы вот такую таблицу:

Id сообщения	Тема	Текст	Автор
1	О рыбалке	Думаю надо сделать так…	Кирилл
2	О рыбалке	Согласен	Вася
3	О рыбалке	А еще можно сделать так…	Семен
4	Велосипеды	Согласен	Вася

В поле Темы часто повторяются одни и те же названия. Помимо того, что для их хранения потребуются дополнительные ресурсы памяти, при дублировании информации очень несложно допустить ошибку при вводе значений атрибута, в результате чего БД перейдет в несогласованное состояние.

Кроме того, при работе с такими таблицами могут возникнуть так называемые аномалии обновления. Например, если мы удалим из этой таблицы четвертое сообщение, то вместе с ним пропадет и информация о теме. Такая ситуация представляет собой аномалию удаления. Если мы решим поменять название темы, то нам придется просмотреть все строки и в каждой заменить старую тему на новую. Это так называемая аномалия модификации. Существуют и другие виды аномалий.

Далеко не всегда эти недостатки можно учесть сразу. Для их устранения и применяется процесс нормализации. Он включает ряд правил, используемых для проверки всех таблиц базы данных. Различают:

- 1НФ - первая нормальная форма
- 2НФ - вторая нормальная форма
- 3НФ - третья нормальная форма

Каждая нормальная форма налагает определенные ограничения на данные. Каждая нормальная форма более высокого уровня предполагает, что анализируемая таблица уже находится в нормальной форме на уровень ниже рассматриваемой. В ходе нормализации схема базы данных становится все более строгой, а ее таблицы все менее подвержены различного рода аномалиям.

Для реляционных баз данных необходимо, чтобы ее таблицы находились в 1НФ. Нормальные формы более высоких уровней могут использоваться разработчиками по своему усмотрению. Однако грамотный специалист стремится к тому, чтобы довести уровень нормализации базы данных хотя бы до 3НФ, тем самым исключив избыточность данных и аномалии обновления.

Первая нормальная форма

Таблица находится в первой нормальной форме, если все ее поля имеют простые (атомарные) значения. Само понятие атомарности определить достаточно трудно. Значение, атомарное в одном случае, может быть неатомарным в другом. Общий принцип здесь такой: значение не атомарно, если оно используется по частям. Понятнее будет на примере.

В нашей таблице Поставщики есть поле Адрес. Если наш магазин работает только с поставщиками из одного города, то значения поля Адрес можно считать атомарными, а саму таблицу - приведенной к 1НФ.

Но что если наши поставщики находятся в разных городах? Тогда, посылая машину за товарами в определенный город, мы должны быть уверенны, что она заберет товары у всех поставщиков, находящихся в этом городе. Т.е. нам могут понадобиться сведения о поставщикам, находящихся в определенном городе. В этом случае, значения в поле Адрес уже не являются атомарными (т.к. мы используем часть адреса), и для приведения таблицы к 1НФ нам надо выделить еще одно поле - Город:

Поставщик
Id поставщика (PK)
Наименование
Город
Адрес

Таким образом надо проанализировать все таблицы нашей базы данных. Так, в таблице Покупатель есть поле ФИО. Если мы собираемся, например, поздравлять наших покупателей с именинами (которые, как известно, завися от имени), то это поле пришлось бы разбить на три: Фамилию, Имя и Отчество. Наш магазин этого делать не собирается, поэтому поле ФИО можно считать атомарным, а таблицу - приведенной к 1НФ.

Для запросов нашего магазина все остальные таблицы приведены к 1НФ.

Вторая нормальная форма

Эта форма применяется к таблицам с составными ключами. Таблица, у которой первичный ключ включает только одно поле, всегда находится во 2НФ.

Таблица находится во второй нормальной форме, если она находится в первой нормальной форме, а каждое неключевое поле функционально полно зависит от составного ключа.

В нашей базе данных две таблицы имеют составной ключ - Журнал покупок и Журнал поставок. Значение поля Количество зависит, как от Поставки (Покупки), так и от Товара. Значит, наши таблицы находятся во 2НФ.

Но предположим, что на этапе концептуального моделирования нашей базы данных, мы не выделили объекты Поставка и Покупка. Тогда наши таблицы могли бы выглядеть так:

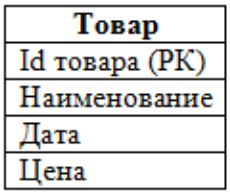
Посмотрим теперь на таблицу Журнал поставок: поле Количество зависит от Наименования товара и от Даты поставки, но не зависит от того, кто поставил товар (поле Поставщика). Т.е. таблица не находится во 2НФ. Если бы на этапе концептуального моделирования нашей базы данных, мы не выделили объекты Поставка и Покупка, нам бы пришлось это делать сейчас. Но мы их выделили, поэтому все наши таблицы находятся во 2НФ.

Третья нормальная форма

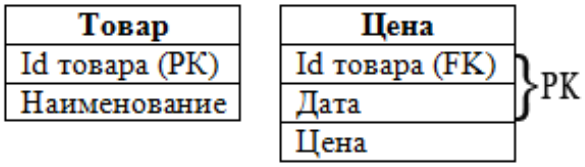
Таблица находится в третьей нормальной форме, если она находится во второй нормальной форме, и каждое неключевое поле нетранзитивно зависит от первичного ключа.

Транзитивная зависимость наблюдается в том случае, если одно из двух неключевых полей зависит от первичного ключа, а другое зависит от первого неключевого поля. На примере будет понятнее.

Посмотрим на нашу таблицу Товар. В ней есть поле Цена, но цены, как известно, имеют свойство меняться. Если мы будем их менять прямо здесь, то будет пропадать вся информация о предыдущих ценах. Чтобы не терять эту информацию, надо добавить поле Дата (когда изменилась цена). Тогда наша таблица будет выглядеть так:



Даже не прибегая к 3НФ видно, что такая таблица будет содержать избыточную информацию. Но посмотрим на ее поля: поля Наименование и Дата зависят от id товара, а поле Цена зависит также и от Даты. Т.е. таблица не находится в 3НФ. Для устранения транзитивной зависимости необходимо провести "расщепление" объекта на два:



Все остальные таблицы нашей базы данных находятся в 3НФ. Кстати, в таблице Товар можно было и не вводить поле id товара, а сделать первичным ключом поле Наименование, но как уже говорилось в третьем уроке суррогатные ключи все-таки предпочтительнее.

Подведем итог. Схема нашей базы данных после нормализации несколько изменилась и выглядит теперь так:

Таким образом, мы преобразовали нашу концептуальную модель в реляционную. Дальше необходимо эту модель реализовать в конкретной СУБД. Для этого нам понадобится сама СУБД и знание языка SQL. Этим вопросам будут посвящены Уроки SQL.

А пока подведем итог уроков "Основы БД". Проектирование БД процесс, как правило, трудоемкий и небыстрый. Ведь нужно очень хорошо изучить предметную область, чтобы учесть все нюансы, пожелания и требования пользователей. Затем всю собранную информацию изобразить в виде объектов, атрибутов и связей. Причем сделать это надо наиболее рационально.

Вообще, среди разработчиков наблюдаются различные взгляды на процесс проектирования БД. Одни игнорируют всякую теорию и руководствуются только своим опытом и здравым смыслом. Другие считают этот процесс искусством, отводя главную роль интуиции. Но в любом случае, знания не бывают лишними. И если вы к интуиции и здравому смыслу добавите теорию, то результат будет гораздо лучше.

Да, база данных - это всего лишь хранилище данных, но от того насколько грамотно вы организуете это хранилище, будет зависеть работа вашего приложения, использующего данные. Помните об этом и не пренебрегайте теорией.

В заключение хотелось бы напомнить, зачем вообще вам нужно уметь проектировать базы данных. Предположим, вы решили организовать у себя на сайте регистрацию пользователей с тем, чтобы обеспечить им доступ к закрытым материалам сайта.

Для реализации этого вопроса вам потребуется создать БД, которая будет хранить информацию о пользователях, их логинах и паролях. А также сделать html-формы регистрации и входа в закрытый раздел.

Когда пользователь регистрируется, эти данные программными средствами (например, с помощью языка PHP) заносятся в созданную вами БД. Когда пользователь вводит логин и пароль в форме входа в закрытый раздел, к базе данных отправляется запрос (на языке SQL), есть ли пользователь с такими данными. И если ответ положительный, то пользователю посылается запрашиваемая страница (разумеется тоже с помощью программы на PHP).

Таким образом, чтобы реализовывать такие приложения вам необходимо уметь создавать БД, строить запросы на языке SQL к БД и знать какой-нибудь язык программирования, применимый для разработки динамических web-страниц (например, PHP).

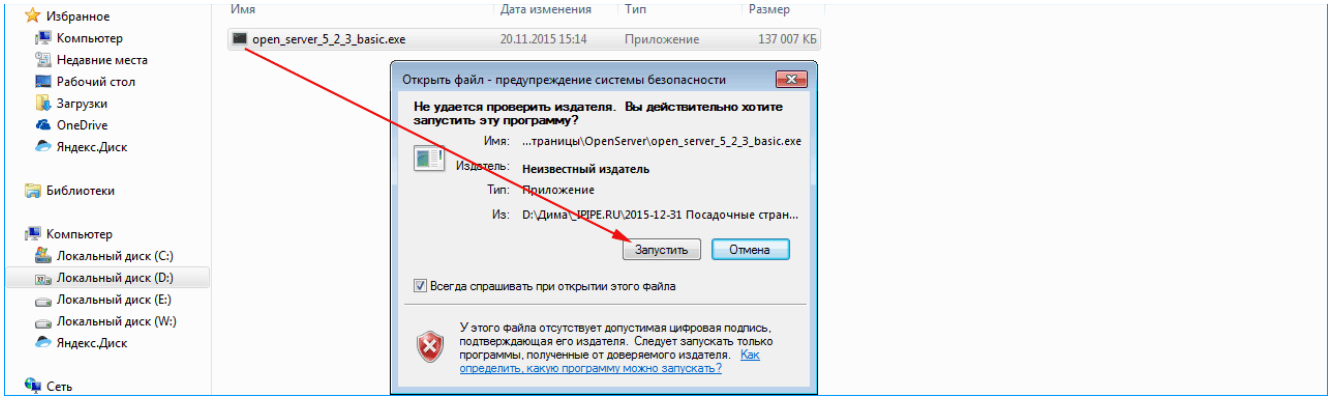
В принципе, вы можете сначала изучить язык программирования, а потом изучать БД и SQL. Но на этом сайте обучение построено в следующем порядке: БД - SQL - PHP. Сделано это потому, что без баз данных ничего интересного на PHP сделать не удастся. Поэтому до встречи в Уроках SQL.

Перед тем, как переходить к уроках SQL, вам потребуется установить сервер MySQL. В принципе вы можете установить только его, но для полноценной дальнейшей работы вам понадобится интерфейс phpMyAdmin, а для работы с ним - модуль PHP и локальный сервер. Поэтому настоятельно рекомендую установить сразу и сервер Apache, и модуль PHP, и MySQL. Можно также установить пакет Openserver в котором все включено.

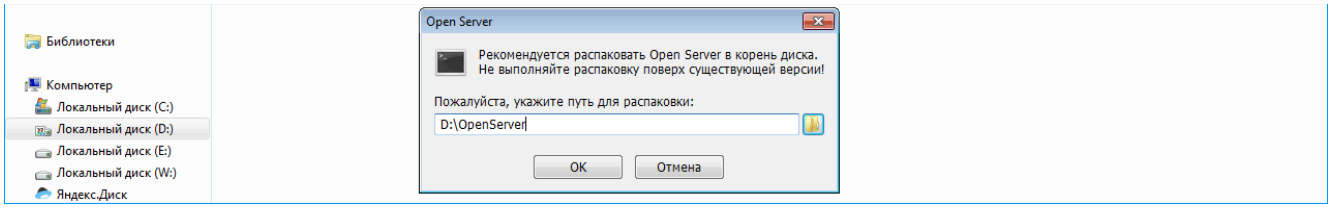
2. Предварительные установки (Apache, PHP , MySQL, phpMyAdmin,)

Устанавливаем OpenServer.

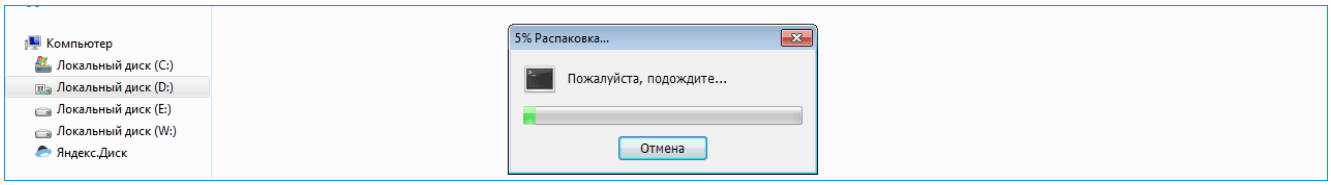
Переходим в папку со скачанным установщиком, запускаем файл:



Выбираем папку, в которую будет распакована программа, нажимаем «ОК»:



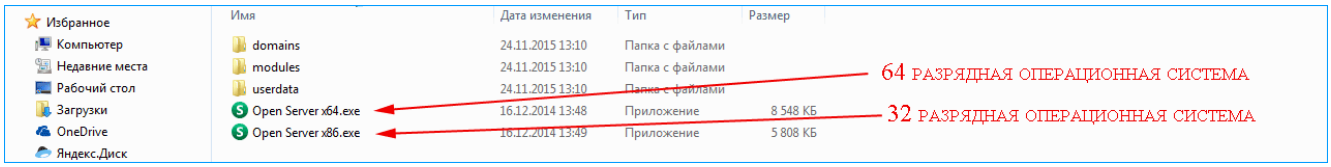
Ждем, пока распакуется архив:



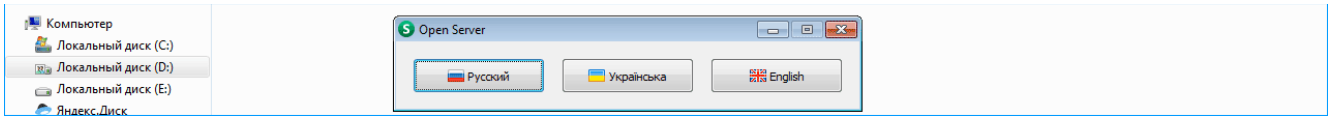
Теперь программу можно запускать.

Настройка и запуск OpenServer.

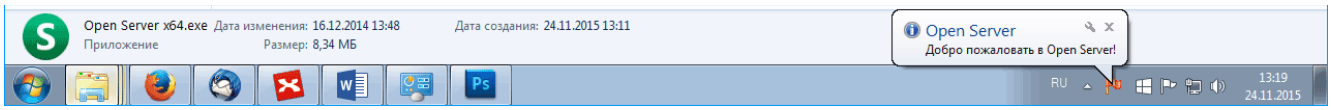
После завершения процесса установки, можно перейти в заданную папку и запустить программу. **В зависимости от разрядности операционной системы, выберите либо «Open Server x64.exe», либо «Open Server x86.exe».**



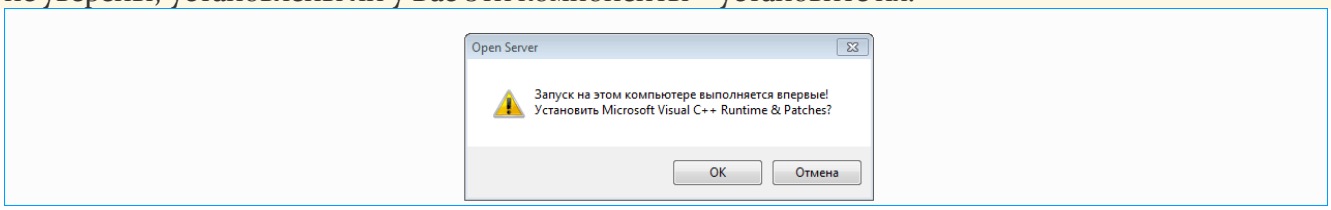
После запуска выберите желаемый язык:



В трее появится иконка OpenServer с уведомлением об успешном запуске:

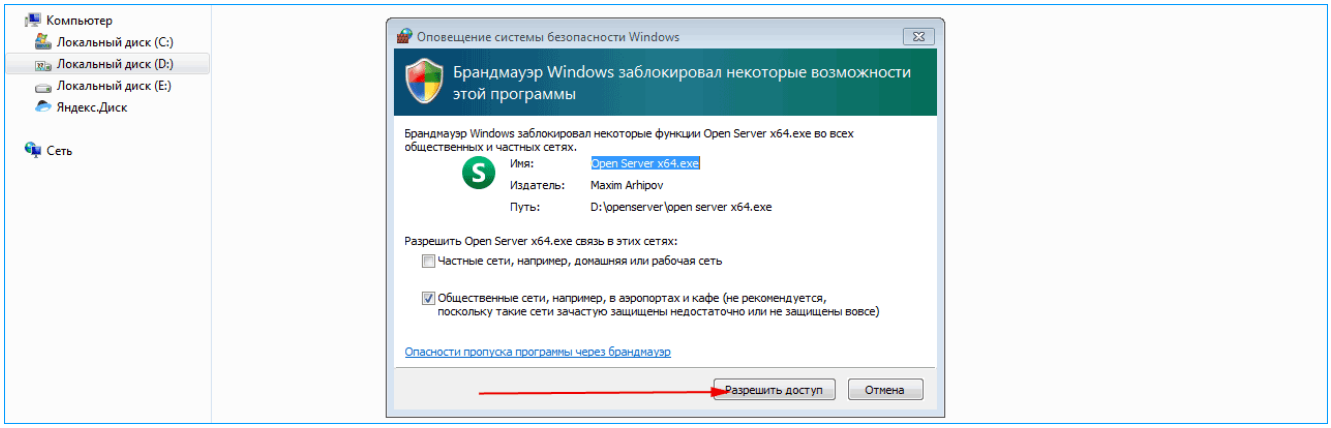


Если программа запускается впервые, Вам могут предложить установить патчи для Microsoft Visual C++. Для базовой работы с программой это можно не делать. В случае, если Вы не уверены, установлены ли у Вас эти компоненты – установите их:



Брандмауэр Windows и OpenServer.

Если доступ к программе блокирует брандмауэр - проверьте, откуда был скачан дистрибутив. Если программа была скачана с официального сайта – опасаться нечего, разрешаем доступ.

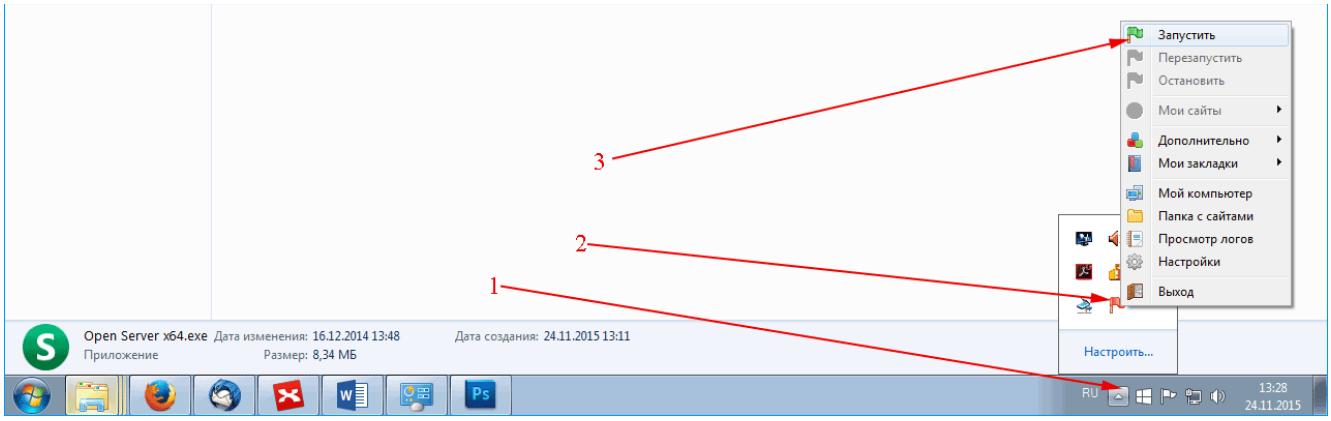


Проверка работы Опен Сервер после установки.

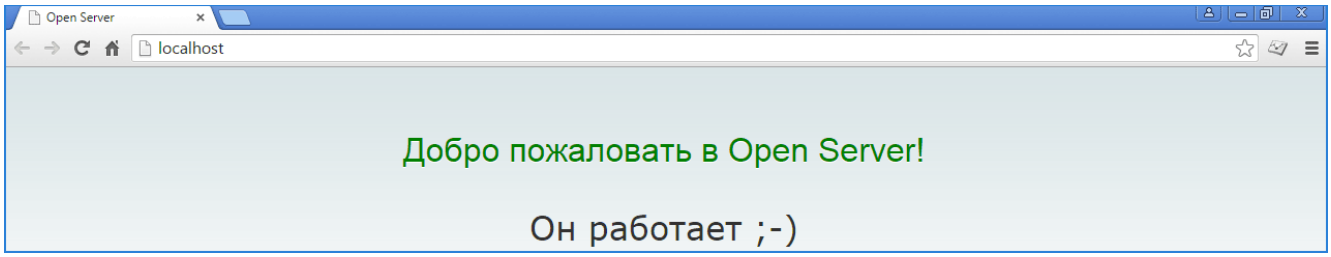
После установки всех необходимых компонентов – нужно запустить программу. Для этого в трее нажимаем на ее иконку и выбираем «Запустить».

OpenServer может не запуститься одновременно со скайпом, т.к. обе программы используют одни и те же порты. **Поэтому перед запуском сервера выключите скайп.** Вы сможете изменить порты, используемые по умолчанию в одной из программ позднее.

Запускаем:



В браузере набираем «localhost» - если программа была установлена корректно, увидим следующее сообщение:

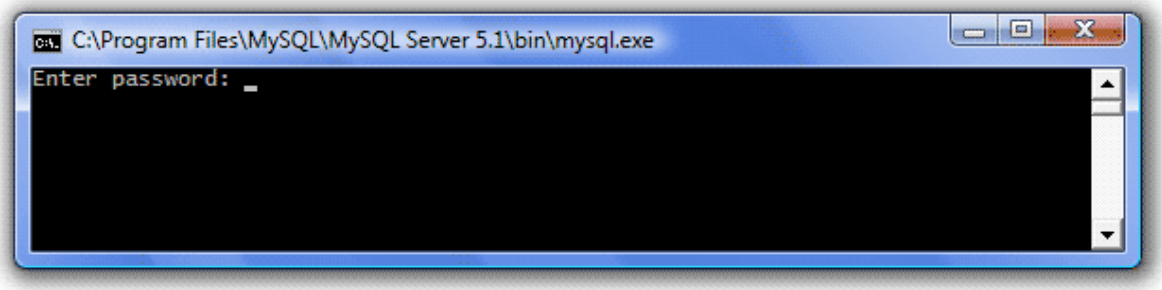


На этом установка закончена, теперь Вы можете использовать любые компоненты программы.

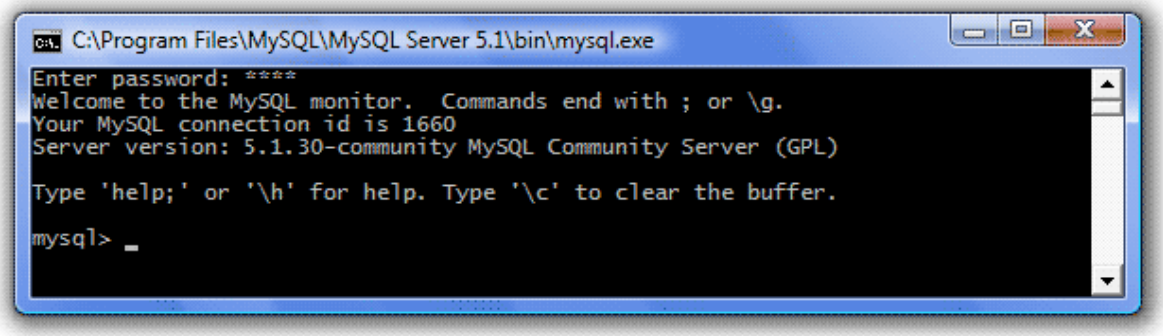
3. Создание базы данных и таблиц

Итак, вы установили MySQL, и мы начинаем осваивать язык SQL. В предыдущем уроке по основам баз данных, мы создали концептуальную модель маленькой БД для форума. Пришло время реализовать ее в СУБД MySQL.

Для этого прежде всего надо запустить сервер MySQL. Идем в системное меню Пуск - Программы - MySQL - MySQL Server 5.1 - MySQL Command Line Client. Откроется окно, предлагающее ввести пароль.



Нажимаем Enter на клавиатуре, если вы не указывали пароль при настройке сервера или указываем пароль, если вы его задавали. Ждем приглашения mysql>.

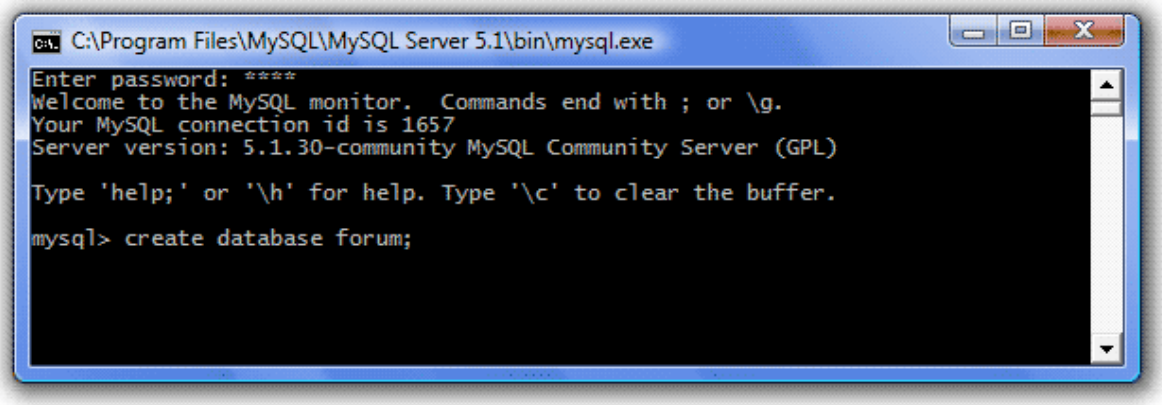


Нам надо создать базу данных, которую мы назовем forum. Для этого в SQL существует оператор *create database*. Создание базы данных имеет следующий синтаксис:

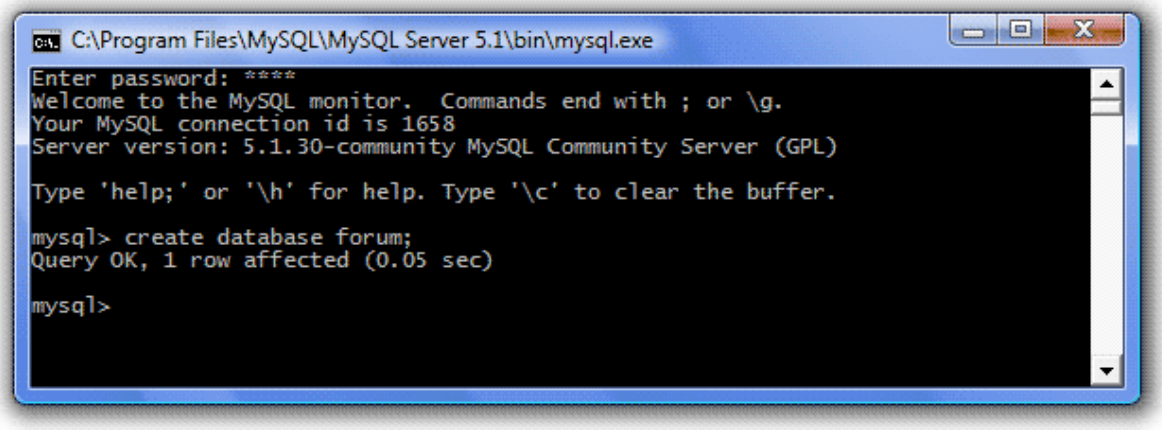
```
create database имя_базы_данных;
```

Максимальная длина имени БД составляет 64 знака и может включать буквы, цифры, символ "_" и символ "$". Имя может начинаться с цифры, но не должно полностью состоять из цифр. Любой запрос к БД заканчивается точкой с запятой (этот символ называется разделителем - delimiter). Получив запрос, сервер выполняет его и в случае успеха выдает сообщение "Query OK ..."

Итак, создадим БД forum:



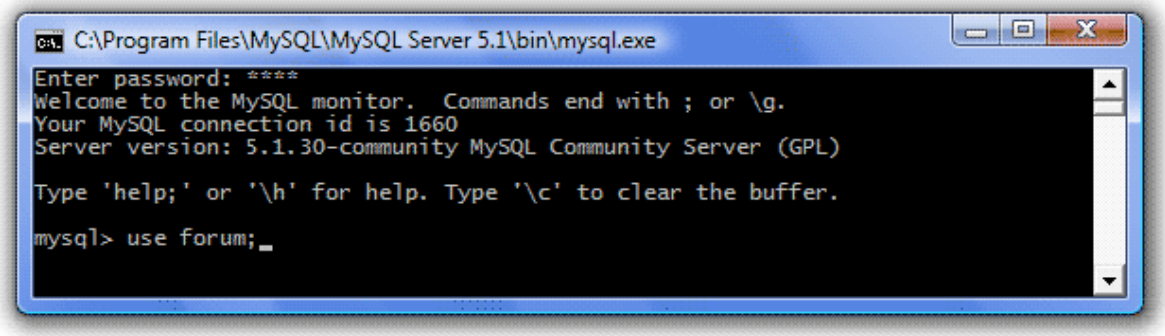
Нажимаем Enter и видим ответ "Query OK ...", означающий, что БД была создана:



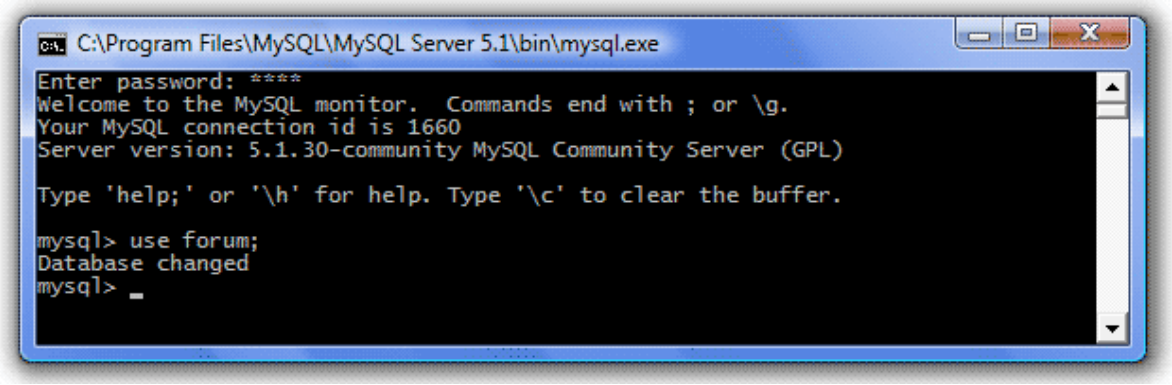
Вот так все просто. Теперь в этой базе данных нам надо создать 3 таблицы: темы, пользователи и сообщения. Но перед тем, как это делать, нам надо указать серверу в какую именно БД мы создаем таблицы, т.е. надо выбрать БД для работы. Для этого используется оператор *use*. Синтаксис выбора БД для работы следующий:

```
use имя_базы_данных;
```

Итак, выберем для работы нашу БД forum:



Нажимаем Enter и видим ответ "Database changed" - база данных выбрана.

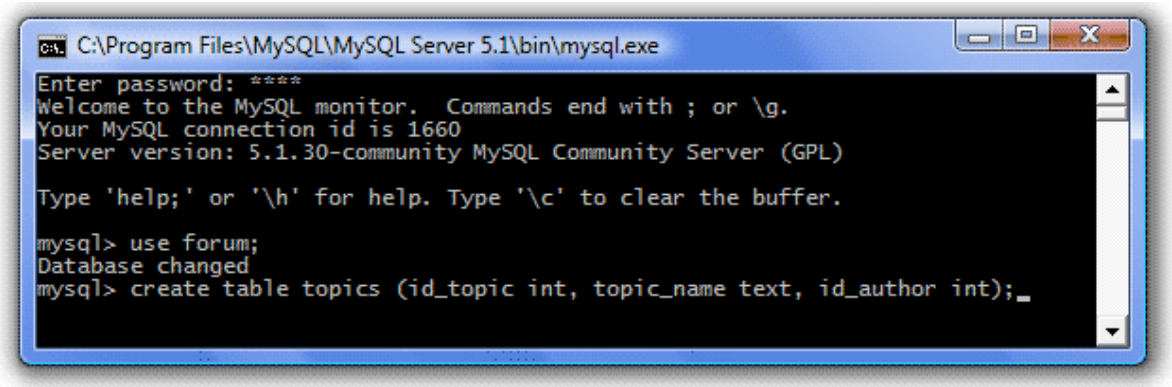


Выбирать БД необходимо в каждом сеансе работы с MySQL.

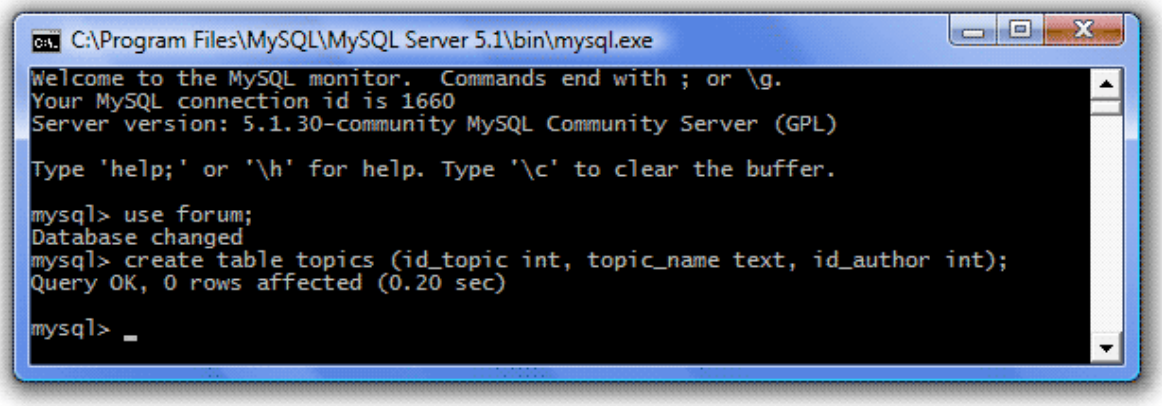
Для создания таблиц в SQL существует оператор *create table*. Создание базы данных имеет следующий синтаксис:

```
create table имя_таблицы (имя_первого_столбца тип, имя_второго_столбца тип, ..., имя_последнего_столбца тип );
```

Требования к именам таблиц и столбцов такие же, как и для имен БД. К каждому столбцу привязан определенный тип данных, который ограничивает характер информации, которую можно хранить в столбце (например, предотвращает ввод букв в числовое поле). MySQL поддерживает несколько типов данных: числовые, строковые, календарные и специальный тип NULL, обозначающий отсутствие информации. Подробно о типах данных мы будем говорить в следующем уроке, а пока вернемся к нашим таблицам. В них у нас всего два типа данных - целочисленные значения (int) и строки (text). Итак, создадим первую таблицу - Темы:

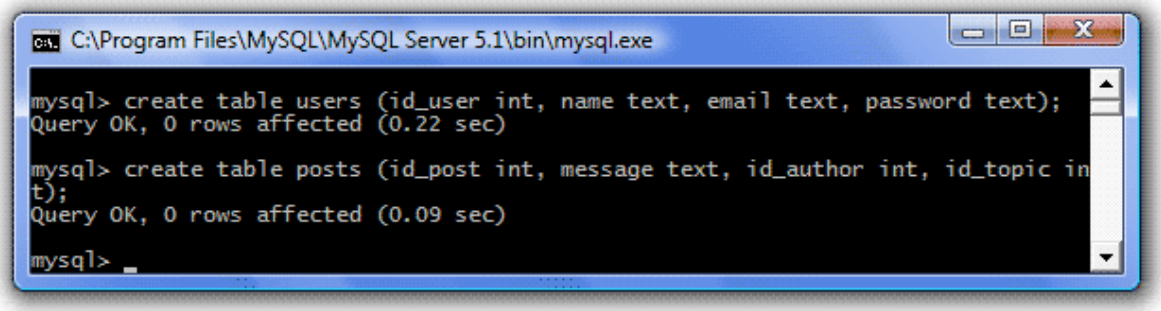


Нажимаем Enter - таблица создана:



Итак, мы создали таблицу topics (темы) с тремя столбцами:
id_topic int - id темы (целочисленное значение),
topic_name text - имя темы (строка),
id_author int - id автора (целочисленное значение).

Аналогичным образом создадим оставшиеся две таблицы - users (пользователи) и posts (сообщения):



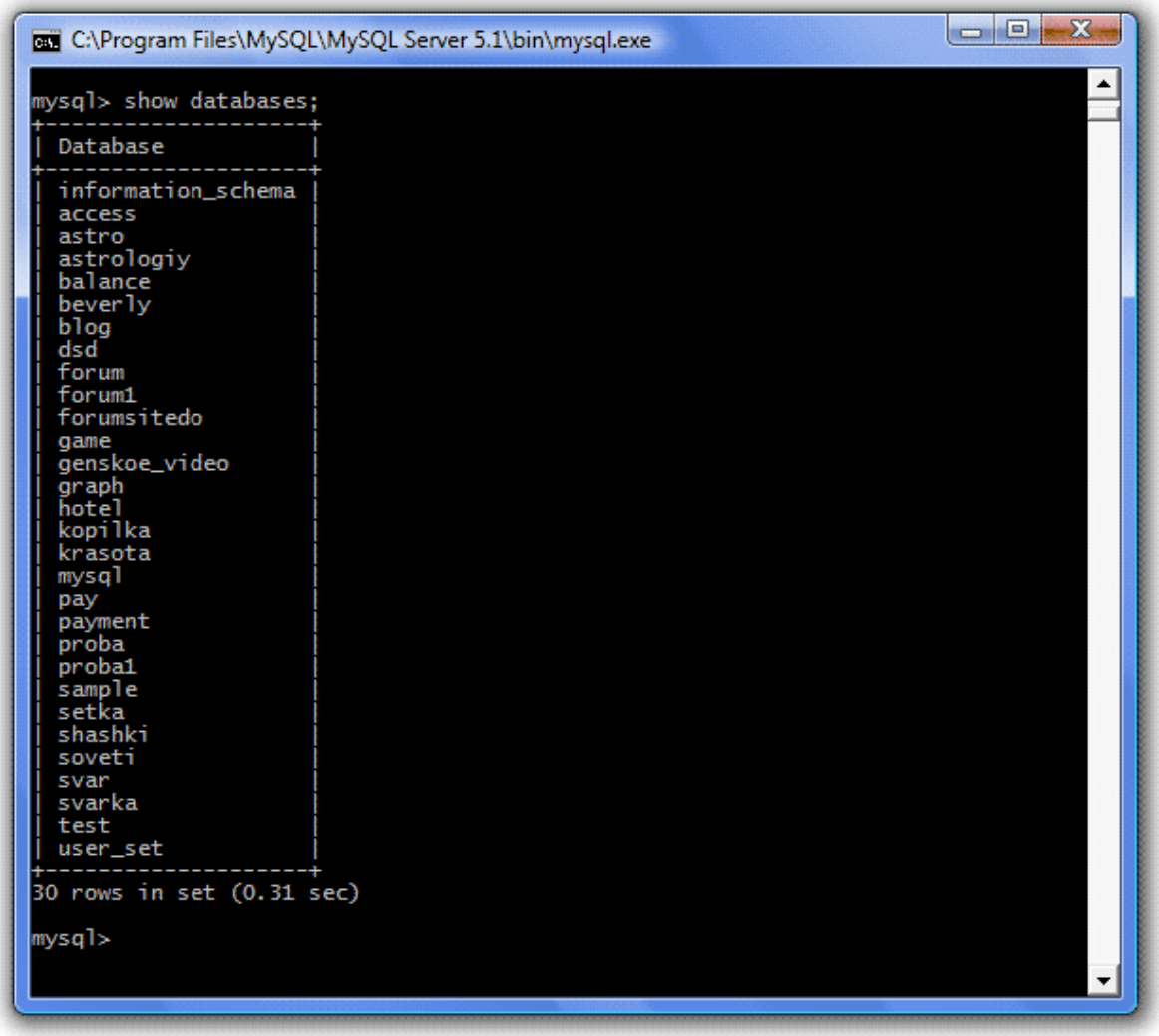
Итак, мы создали БД forum и в ней три таблицы. Сейчас мы об этом помним, но если наша БД будет очень большой, то удержать в голове названия всех таблиц и столбцов просто невозможно. Поэтому надо иметь возможность посмотреть, какие БД у нас существуют, какие таблицы в них присутствуют, и какие столбцы эти таблицы содержат. Для этого в SQL существует несколько операторов:

show databases - показать все имеющиеся БД,

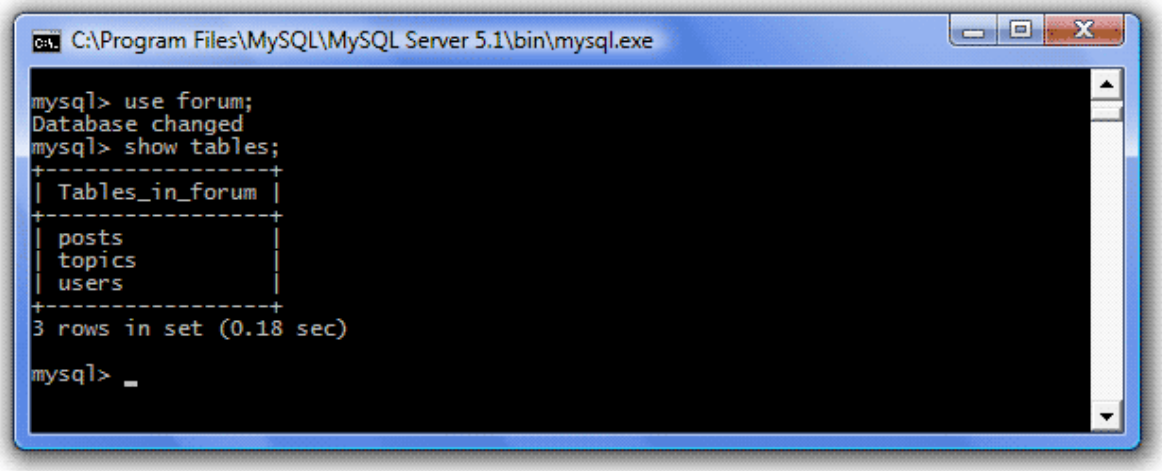
show tables - показать список таблиц текущей БД (предварительно ее надо выбрать с помощью оператора *use*),

describe имя_таблицы - показать описание столбцов указанной таблицы.

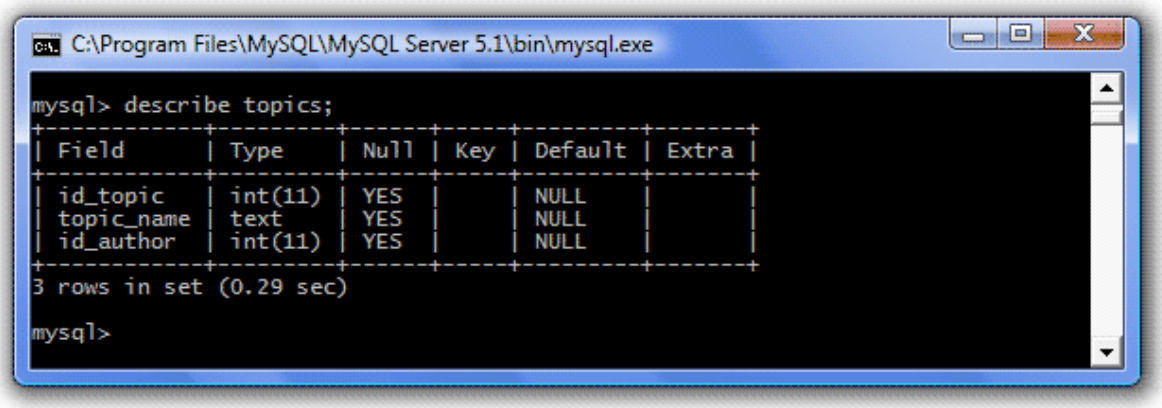
Давайте попробуем. Смотрим все имеющиеся базы данных (у вас она пока одна - forum, у меня 30, и все они перечислены в столбик):



Теперь посмотрим список таблиц БД forum (для этого ее предварительно надо выбрать), не забываем после каждого запроса нажимать Enter:



В ответе видим названия наших трех таблиц. Теперь посмотрим описание столбцов, например, таблицы topics:

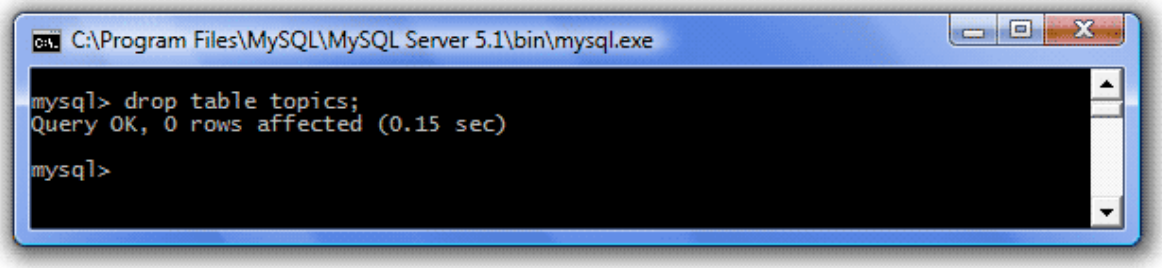


Первые два столбца нам знакомы - это имя и тип данных, значения остальных нам еще предстоит узнать. Но прежде мы все-таки узнаем какие типы данных бывают, какие и когда следует использовать.

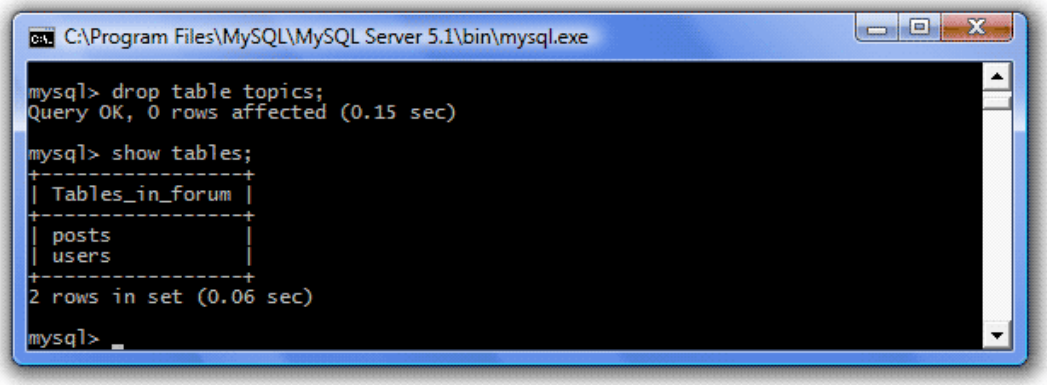
А сегодня мы рассмотрим последний оператор - *drop*, он позволяет удалять таблицы и БД. Например, давайте удалим таблицу topics. Так как мы два шага назад выбирали БД forum для работы, то сейчас ее выбирать не надо, можно просто написать:

```
drop table имя_таблицы;
```

и нажать Enter.



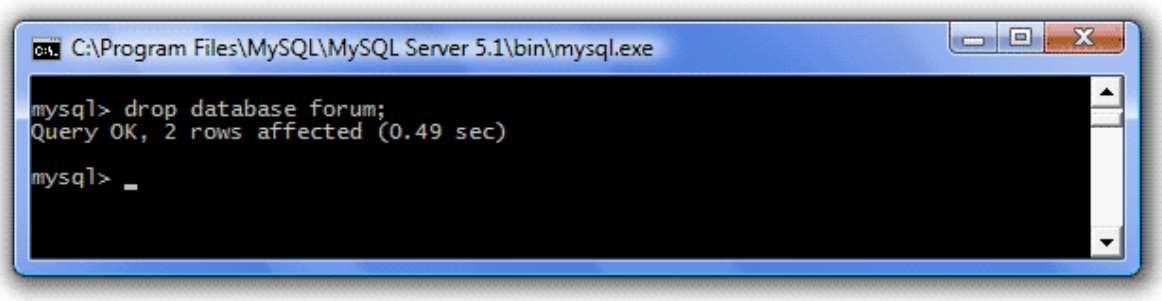
Теперь снова посмотрим список таблиц нашей БД:



Наша таблица действительно удалена. Теперь давайте удалим и саму БД forum (удаляйте, не жалейте, ее все равно придется переделывать). Для этого напишем:

```
drop database имя_базы данных;
```

и нажмем Enter.



3.1. Типы данных

Этот урок носит больше теоретический характер, но пропустить его нельзя. В дальнейшем вы сможете возвращаться к нему, как к справочному уроку, сейчас же просто ознакомьтесь. В прошлом уроке говорилось, что MySQL поддерживает числовые, строковые, календарные данные и данные типа NULL. Рассмотрим их по очереди.

Числовые типы данных

Тип данных	Объем памяти	Диапазон	Описание
TINYINT (M)	1 байт	от -128 до 127 или от 0 до 255	Целое число. Может быть объявлено положительным с помощью ключевого слова UNSIGNED, тогда элементам столбца нельзя будет присвоить отрицательное значение. Необязательный параметр M - количество отводимых под число символов. Необязательный атрибут ZEROFILL позволяет свободные позиции по умолчанию заполнить нулями. *Примеры:* TINYINT - хранит любое число в диапазоне от -128 до 127. TINYINT UNSIGNED - хранит любое число в диапазоне от 0 до 255. TINYINT (2) - предполагается, что значения будут двузначными, но по факту будет хранить

			и трехзначные. TINYINT (3) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 002.
SMALLINT (M)	2 байта	от -32768 до 32767 или от 0 до 65535	Аналогично предыдущему, но с большим диапазоном. *Примеры:* SMALLINT - хранит любое число в диапазоне от -32768 до 32767. SMALLINT UNSIGNED - хранит любое число в диапазоне от 0 до 65535. SMALLINT (4) - предполагается, что значения будут четырехзначные, но по факту будет хранить и пятизначные. SMALLINT (4) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 0002.
MEDIUMINT (M)	3 байта	от -8388608 до 8388608 или от 0 до 16777215	Аналогично предыдущему, но с большим диапазоном. *Примеры:* MEDIUMINT - хранит любое число в диапазоне от -8388608 до 8388608. MEDIUMINT UNSIGNED - хранит любое число в диапазоне от 0 до 16777215. MEDIUMINT (4) - предполагается, что значения будут четырехзначные, но по факту будет хранить и семизначные. MEDIUMINT (5) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 00002.

INT (M) или INTEGER (M)	4 байта	от -2147683648 до 2147683648 или от 0 до 4294967295	Аналогично предыдущему, но с большим диапазоном. *Примеры:* INT - хранит любое число в диапазоне от -2147683648 до 2147683648. INT UNSIGNED - хранит любое число в диапазоне от 0 до 4294967295. INT (4) - предполагается, что значения будут четырехзначные, но по факту будет хранить максимально возможные. INT (5) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 00002.
BIGINT (M)	8 байта	от -2^{63} до $2^{63}-1$ или от 0 до 2^{64}	Аналогично предыдущему, но с большим диапазоном. *Примеры:* BIGINT - хранит любое число в диапазоне от -2^{63}до $2^{63}-1$. BIGINT UNSIGNED - хранит любое число в диапазоне от 0 до 2^{64}. BIGINT (4) - предполагается, что значения будут четырехзначные, но по факту будет хранить максимально возможные. BIGINT (7) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 0000002.
BOOL или BOOLEAN	1 байт	либо 0, либо 1	Булево значение. 0 - ложь (false), 1 - истина (true).
DECIMAL (M,D) или DEC (M,D) или	M + 2 байта	зависят от параметров M и D	Используются для величин повышенной точности, например, для денежных данных. M - количество отводимых под число символов (максимальное значение - 64). D - количество знаков после запятой (максимальное значение -

NUMERIC (M,D)			30). *Пример:* DECIMAL (5,2) - будет хранить числа от -99,99 до 99,99.
FLOAT (M,D)	4 байта	мин. значение +(-) $1.175494351 * 10^{-39}$ макс. значение +(-) $3.402823466 * 10^{38}$	Вещественное число (с плавающей точкой). Может иметь параметр UNSIGNED, запрещающий отрицательные числа, но диапазон значений от этого не изменится. M - количество отводимых под число символов. D - количество символов дробной части. *Пример:* FLOAT (5,2) - будет хранить числа из 5 символов, 2 из которых будут идти после запятой (например: 46,58).
DOUBLE (M,D)	8 байт	мин. значение +(-) $2.2250738585072015 * 10^{-308}$ макс. значение +(-) $1.797693134862315 * 10^{308}$	Аналогично предыдущему, но с большим диапазоном. *Пример:* DOUBLE - будет хранить большие дробные числа.

Необходимо понимать, чем больше диапазон значений у типа данных, тем больше памяти он занимает. Поэтому, если предполагается, что значения в столбце не будут превышать 100, то используйте тип TINYINT. Если при этом все значения будут положительными, то используйте атрибут UNSIGNED. Правильный выбор типа данных позволяет сэкономить место для хранения этих данных.

Строковые типы данных

Тип данных	Объем памяти	Максимальный размер	Описание
CHAR (M)	M символов	M символов	Позволяет хранить строку фиксированной длины M. Значение M - от 0 до 65535. *Примеры:* CHAR (8) - хранит строки из 8 символов и

			занимает 8 байтов. Например, любое из следующих значений: '', 'Иван','Ирина', 'Сергей' будет занимать по 8 байтов памяти. А при попытке ввести значение 'Александра', оно будет усечено до 'Александ', т.е. до 8 символов.
VARCHAR (M)	L+1 символов	M символов	Позволяет хранить переменные строки длиной L. Значение M - от 0 до 65535. *Примеры:* VARCHAR (3) - хранит строки максимум из 3 символов, но пустая строка '' занимает 1 байт памяти, строка 'a' - 2 байта, строк 'aa' - 3 байта, строка 'aaa' - 4 байта. Значение более 3 символов будет усечено до 3.
BLOB, TEXT	L+2 символов	2^{16}-1 символов	Позволяют хранить большие объемы текста. Причем тип TEXT используется для хранения именно текста, а BLOB - для хранения изображений, звука, электронных документов и т.д.
MEDIUMBLOB, MEDIUMTEXT	L+3 символов	2^{24}-1 символов	Аналогично предыдущему, но с большим размером.
LONGBLOB, LONGTEXT	L+4 символов	2^{32}-1 символов	Аналогично предыдущему, но с большим размером.
ENUM ('value1', 'value2', ...,'valueN')	1 или 2 байта	65535 элементов	Строки этого типа могут принимать только одно из значений указанного множества. *Пример:* ENUM ('да', 'нет') - в столбце с таким типом может храниться только одно из имеющихся значений. Удобно использовать, если предусмотрено, что в столбце должен храниться ответ на вопрос.
SET ('value1', 'value2', ...,'valueN')	до 8 байт	64 элемента	Строки этого типа могут принимать любой или все элементы из значений указанного множества. *Пример:*

			SET ('первый', 'второй') - в столбце с таким типом может храниться одно из перечисленных значений, оба сразу или значение может отсутствовать вовсе.

Календарные типы данных

Тип данных	Объем памяти	Диапазон	Описание
DATE	3 байта	от '1000-01-01' до '9999-12-31'	Предназначен для хранения даты. В качестве первого значения указывается год в формате "YYYY", через дефис - месяц в формате "MM", а затем день в формате "DD". В качестве разделителя может выступать не только дефис, а любой символ отличный от цифры.
TIME	3 байта	от '-838:59:59' до '838:59:59'	Предназначен для хранения времени суток. Значение вводится и хранится в привычном формате - hh:mm:ss, где hh - часы, mm - минуты, ss - секунды. В качестве разделителя может выступать любой символ отличный от цифры.
DATATIME	8 байт	от '1000-01-01 00:00:00' до '9999-12-31 23:59:59'	Предназначен для хранения и даты и времени суток. Значение вводится и хранится в формате - YYYY-MM-DD hh:mm:ss. В качестве разделителей могут выступать любые символы отличные от цифры.
TIMESTAMP	4 байта	от '1970-01-01 00:00:00' до '2037-12-31 23:59:59'	Предназначен для хранения даты и времени суток в виде количества секунд, прошедших с полуночи 1 января 1970 года (начало эпохи UNIX).
YEAR (M)	1 байт	от 1970 до 2069 для M=2 и от 1901 до 2155 для M=4	Предназначен для хранения года. M - задает формат года. Например, YEAR (2) - 70, а YEAR (4) - 1970. Если параметр M не указан, то по умолчанию считается, что он равен 4.

Тип данных NULL

Вообще-то это лишь условно можно назвать типом данных. По сути это скорее указатель возможности отсутствия значения. Например, когда вы регистрируетесь на каком-либо сайте, вам предлагается заполнить форму, в которой присутствуют, как обязательные, так и необязательные поля. Понятно, что регистрация пользователя невозможна без указания логина

и пароля, а вот дату рождения и пол пользователь может указать по желанию. Для того, чтобы хранить такую информацию в БД и используют два значения:

NOT NULL (значение не может отсутствовать) для полей логин и пароль,

NULL (значение может отсутствовать) для полей дата рождения и пол.

По умолчанию всем столбцам присваивается тип NOT NULL, поэтому его можно явно не указывать.

Пример:

create table users (login varchar(20), password varchar(15), sex enum('man', 'woman') NULL, date_birth date NULL);

Таким образом, мы создаем таблицу с 4 столбцами: логин (не более 20 символов) обязательное, пароль (не более 15 символов) обязательное, пол (мужской или женский) не обязательное, дата рождения (тип дата) необязательное.

3.2 Создание таблиц и наполнение их информацией

Итак, мы познакомились с типами данных, теперь будем усовершенствовать таблицы для нашего форума. Сначала разберем их. И начнем с таблицы users (пользователи). В ней у нас 4 столбца:

id_user - целочисленные значения, значит будет тип int, ограничим его 10 символами - int (10).
name - строковое значение varchar, ограничим его 20 символами - varchar(20).
email - строковое значение varchar, ограничим его 50 символами - varchar(50).
password - строковое значение varchar, ограничим его 15 символами - varchar(15).

Все значения полей обязательны для заполнения, значит надо добавить тип NOT NULL.

id_user int (10) NOT NULL
name varchar(20) NOT NULL
email varchar(50) NOT NULL
password varchar(15) NOT NULL

Первый столбец, как вы помните из концептуальной модели нашей БД, является первичным ключом (т.е. его значения уникальны, и они однозначно идентифицируют запись). Следить за уникальностью самостоятельно можно, но не рационально. Для этого в SQL есть специальный атрибут - *AUTO_INCREMENT*, который при обращении к таблице на добавление данных высчитывает максимальное значение этого столбца, полученное значение увеличивает на 1 и заносит его в столбец. Таким образом, в этом столбце автоматически генерируется уникальный номер, а следовательно тип NOT NULL излишен. Итак, присвоим атрибут столбцу с первичным ключом:

id_user int (10) AUTO_INCREMENT
name varchar(20) NOT NULL
email varchar(50) NOT NULL
password varchar(15) NOT NULL

Теперь надо указать, что поле id_user является первичным ключом. Для этого в SQL используется ключевое слово*PRIMARY KEY ()*, в скобочках указывается имя ключевого поля. Внесем

изменения:

id_user int (10) AUTO_INCREMENT
name varchar(20) NOT NULL
email varchar(50) NOT NULL
password varchar(15) NOT NULL
PRIMARY KEY (id_user)

Итак, таблица готова, и ее окончательный вариант выглядит так:

```
create table users (
id_user int (10) AUTO_INCREMENT,
name varchar(20) NOT NULL,
email varchar(50) NOT NULL,
password varchar(15) NOT NULL,
PRIMARY KEY (id_user)
);
```

Теперь разберемся со второй таблицей - topics (темы). Рассуждая аналогично, имеем следующие поля:

id_topic int (10) AUTO_INCREMENT
topic_name varchar(100) NOT NULL
id_author int (10) NOT NULL
PRIMARY KEY (id_topic)

Но в модели нашей БД поле id_author является внешним ключом, т.е. оно может иметь только те значения, которые есть в поле id_user таблицы users. Для того, чтобы указать это в SQL есть ключевое слово *FOREIGN KEY ()*, которое имеет следующий синтаксис:

FOREIGN KEY (имя_столбца_которое_является_внешним_ключом) REFERENCES имя_таблицы_родителя (имя_столбца_родителя);

Укажем, что id_author - внешний ключ:

id_topic int (10) AUTO_INCREMENT
topic_name varchar(100) NOT NULL
id_author int (10) NOT NULL
PRIMARY KEY (id_topic)
FOREIGN KEY (id_author) REFERENCES users (id_user)

Таблица готова, и ее окончательный вариант выглядит так:

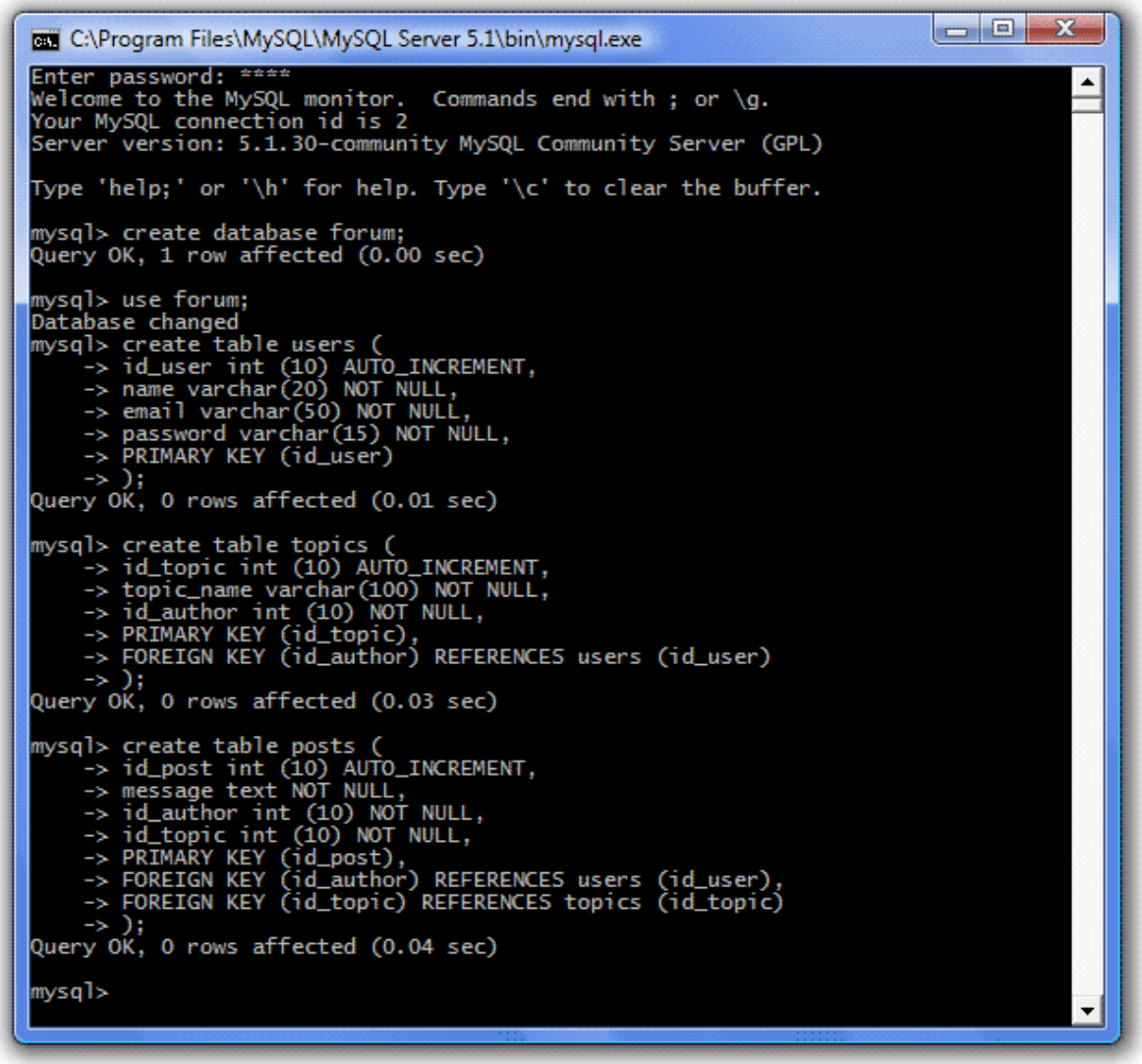
```
create table topics (
id_topic int (10) AUTO_INCREMENT,
topic_name varchar(100) NOT NULL,
id_author int (10) NOT NULL,
PRIMARY KEY (id_topic),
FOREIGN KEY (id_author) REFERENCES users (id_user)
);
```

Осталась последняя таблица - posts (сообщения). Здесь все аналогично, только два внешних ключа:

```
create table  posts (
id_post int (10) AUTO_INCREMENT,
message text NOT NULL,
id_author int (10) NOT NULL,
id_topic int (10) NOT NULL,
PRIMARY KEY (id_post),
FOREIGN KEY (id_author) REFERENCES users (id_user),
FOREIGN KEY (id_topic) REFERENCES topics (id_topic)
);
```

Обратите внимание, внешних ключей у таблицы может быть несколько, а первичный ключ в MySQL может быть только один. В первом уроке мы удалили нашу БД forum, пришло время создать ее вновь.

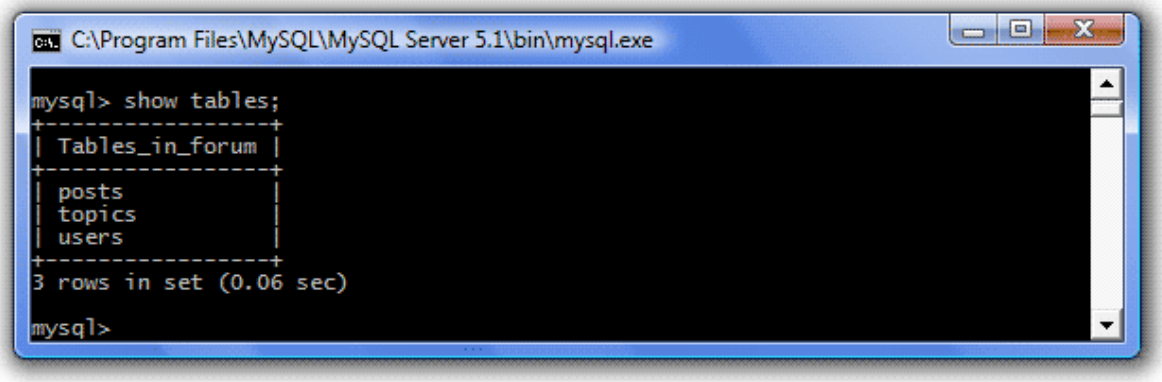
Запускаем сервер MySQL (Пуск - Программы - MySQL - MySQL Server 5.1 - MySQL Command Line Client), вводим пароль, создаем БД forum (create database forum;), выбираем ее для использования (use forum;) и создаем три наших таблицы:



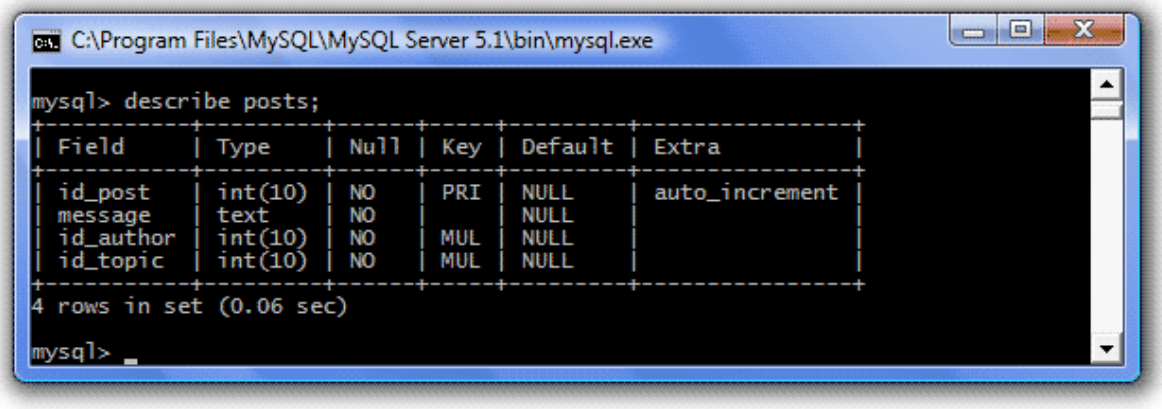
Обратите внимание, одну команду можно писать в несколько строк, используя клавишу Enter (MySQL автоматически подставляет символ новой строки ->), и только после разделителя (точки с запятой) нажатие клавиши Enter приводит к выполнению запроса.

Помните, если вы сделали что-то не так, всегда можно удалить таблицу или всю БД с помощью оператора DROP. Исправлять что-то в командной строке крайне неудобно, поэтому иногда (особенно на начальном этапе) проще писать запросы в каком-нибудь редакторе, например в Блокноте, а затем копировать и вставлять их в черное окошко.

Итак, таблицы созданы, чтобы убедиться в этом вспомним о команде *show tables*:



И, наконец, посмотрим структуру нашей последней таблицы posts:



Теперь становятся понятны значения всех полей структуры, кроме поля DEFAULT. Это поле значений по умолчанию. Мы могли бы для какого-нибудь столбца (или для всех) указать значение по умолчанию. Например, если бы у нас было поле с названием "Женаты\Замужем" и типом ENUM ('да', 'нет'), то было бы разумно сделать одно из значений значением по умолчанию. Синтаксис был бы следующий:

married enum ('да', 'нет') NOT NULL default('да')

Т.е. это ключевое слово пишется через пробел после указания типа данных, а в скобках указывается значение по умолчанию.

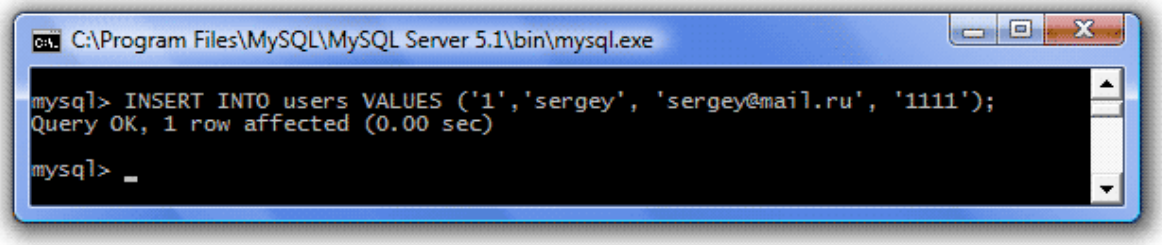
Но вернемся к нашим таблицам. Теперь нам необходимо внести данные в наши таблицы. На сайтах, вы обычно вводите информацию в какие-нибудь html-формы, затем сценарий на каком-либо языке (php, java...) извлекает эти данные из формы и заносит их в БД. Делает он это посредством SQL-запроса на внесение данных в базу. Писать сценарии на php мы пока не умеем, а вот отправлять SQL-запросы на внесение данных сейчас научимся.

Для этого используется оператор *INSERT*. Синтаксис можно использовать двух видов. Первый вариант используется для внесения данных во все поля таблицы:

```
INSERT INTO имя_таблицы VALUES ('значение_первого_столбца','значение_второго_столбца', ..., 'значение_последнего_столбца');
```

Давайте попробуем внести в нашу таблицу users следующие значения:

INSERT INTO users VALUES ('1','sergey', 'sergey@mail.ru', '1111');

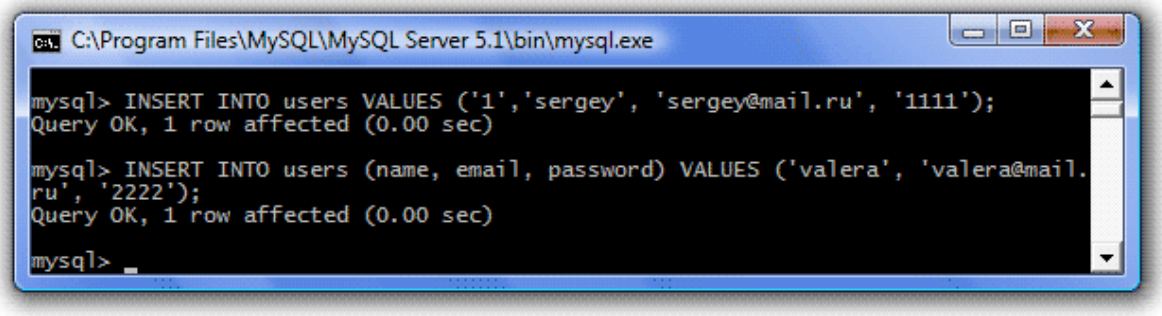


Второй вариант используется для внесения данных в некоторые поля таблицы:

```
INSERT INTO имя_таблицы ('имя_столбца', 'имя_столбца') VALUES ('значение_первого_столбца','значение_второго_столбца');
```

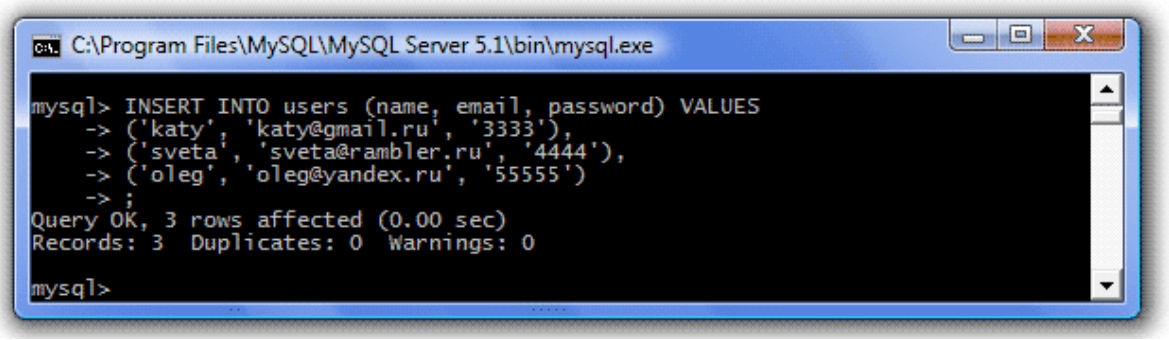
В нашей таблице users все поля обязательны для заполнения, но наше первое поле имеет ключевое слово - AUTO_INCREMENT (т.е. оно заполняется автоматически), поэтому мы можем пропустить этот столбец:

INSERT INTO users (name, email, password) VALUES ('valera', 'valera@mail.ru', '2222');



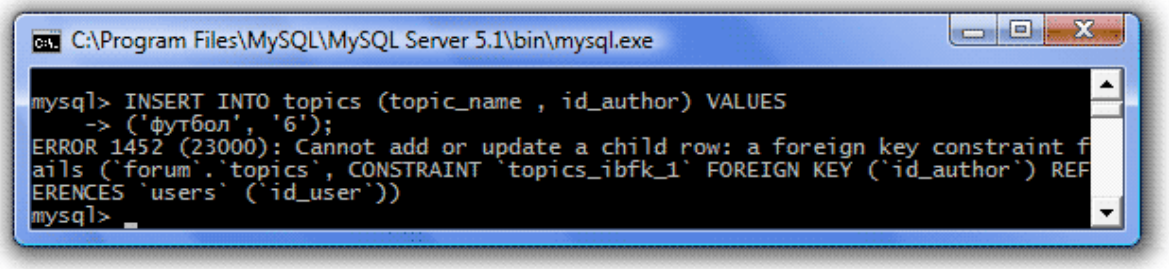
Если бы у нас были поля с типом NULL, т.е. необязательные для заполнения, мы бы тоже могли их проигнорировать. А вот если попытаться оставить пустым поле со значением NOT NULL, то сервер выдаст сообщение об ошибке и не выполнит запрос. Кроме того, при внесении данных сервер проверяет связи между таблицами. Поэтому вам не удастся внести в поле, являющееся внешним ключом, значение, отсутствующее в связанной таблице. В этом вы убедитесь, внося данные в оставшиеся две таблицы.

Но прежде внесем информацию еще о нескольких пользователях. Чтобы добавить сразу несколько строк, надо просто перечислять скобки со значениями через запятую:



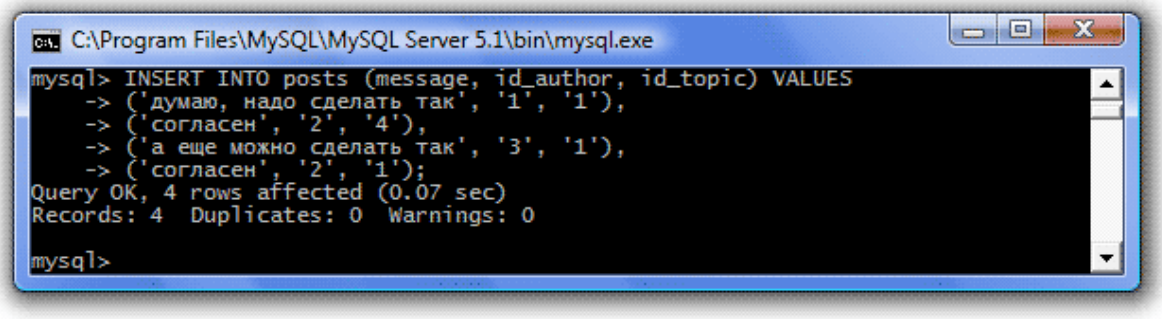
Теперь внесем данные во вторую таблицу - topics (темы). Все тоже самое, но надо помнить, что значения в поле id_author должны присутствовать в таблице users (пользователи):

Теперь давайте попробуем внести еще одну тему, но с id_author, которого в таблице users нет (т.к. мы внесли в таблицу users только 5 пользователей, то id=6 не существует):



Сервер выдает ошибку и говорит, что не может внести такую строку, т.к. в поле, являющемся внешним ключом, стоит значение, отсутствующее в связанной таблице users.

Теперь внесем несколько строк в таблицу posts (сообщения), помня, что в ней у нас 2 внешних ключа, т.е. id_author и id_topic, которые мы будем вносить должны присутствовать в связанных с ними таблицах:



3.3. Выборка данных - оператор SELECT

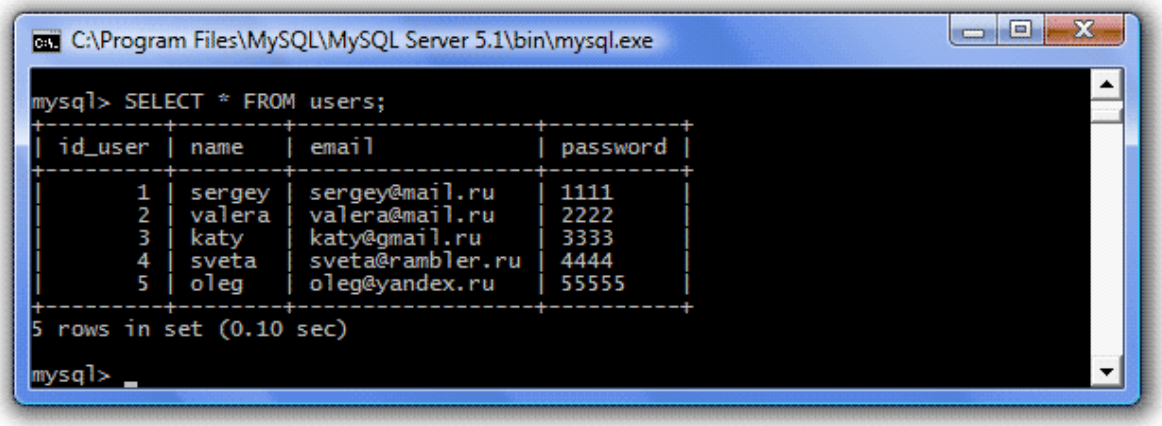
Итак, в нашей БД forum есть три таблицы: users (пользователи), topics (темы) и posts (сообщения). И мы хотим посмотреть, какие данные в них содержатся. Для этого в SQL существует оператор *SELECT*. Синтаксис его использования следующий:

```
SELECT что_выбрать FROM откуда_выбрать;
```

Вместо "что_выбрать" мы должны указать либо имя столбца, значения которого хотим увидеть, либо имена нескольких столбцов через запятую, либо символ звездочки (*), означающий выбор всех столбцов таблицы. Вместо "откуда_выбрать" следует указать имя таблицы.

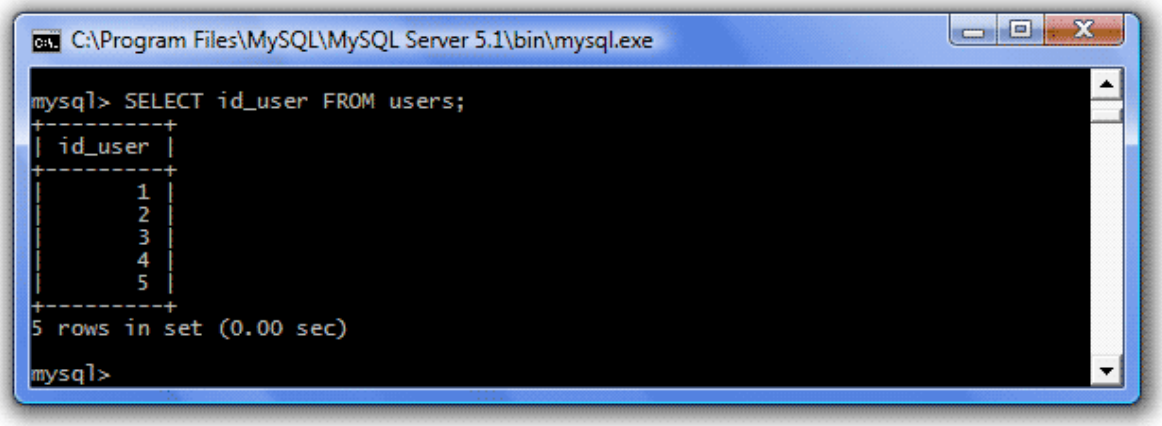
Давайте сначала посмотрим все столбцы из таблицы users:

SELECT * FROM users;



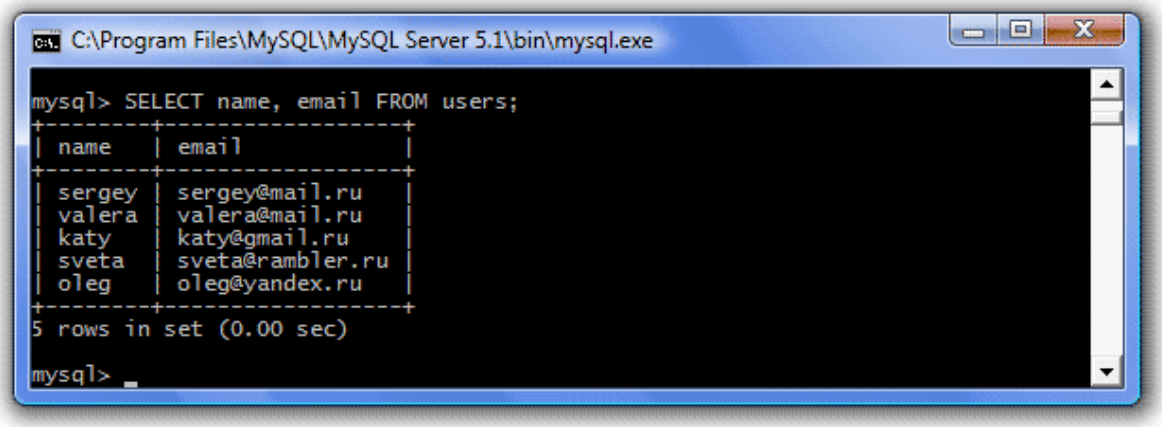
Вот и все наши данные, которые мы вносили в эту таблицу. Но предположим, что мы хотим посмотреть только столбец id_user (например, в прошлом уроке, нам надо было для заполнения таблицы topics (темы) знать, какие id_user есть в таблице users). Для этого в запросе мы укажем имя этого столбца:

SELECT id_user FROM users;



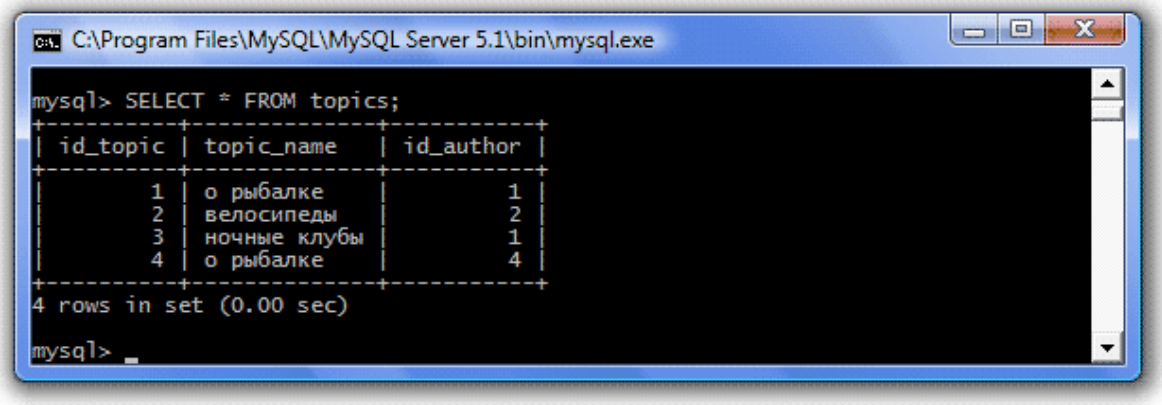
Ну, а если мы захотим посмотреть, например, имена и e-mail наших пользователей, то мы перечислим интересующие столбцы через запятую:

SELECT name, email FROM users;



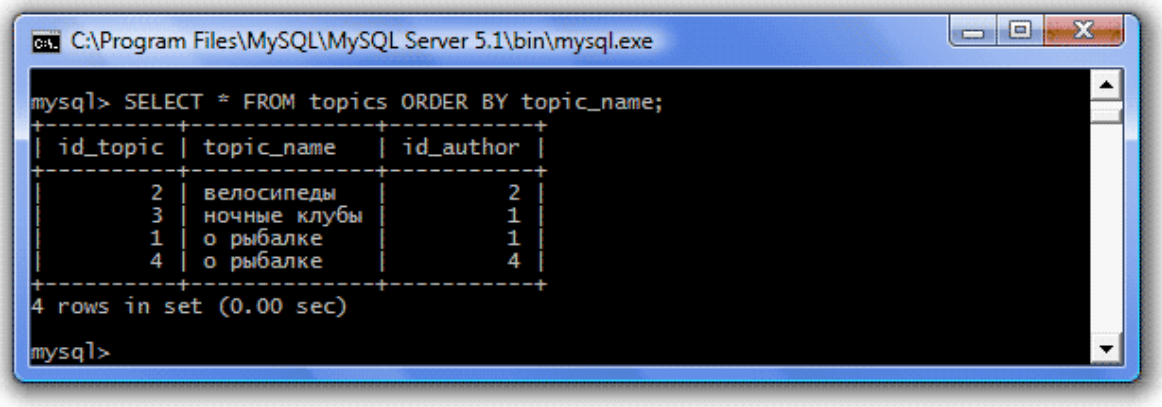
Аналогично, вы можете посмотреть, какие данные содержат и другие наши таблицы. Давайте посмотрим, какие у нас существуют темы:

SELECT * FROM topics;

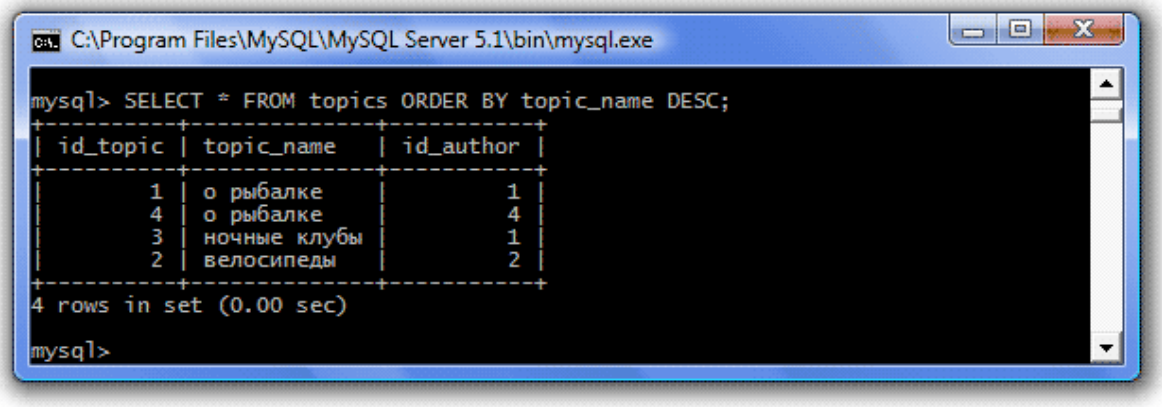


Сейчас у нас всего 4 темы, а если их будет 100? Хотелось бы, чтобы они выводились, например, по алфавиту. Для этого в SQL существует ключевое слово *ORDER BY* после которого указывается имя столбца по которому будет происходить сортировка. Синтаксис следующий:

```
SELECT имя_столбца FROM имя_таблицы ORDER BY имя_столбца_сортировки;
```

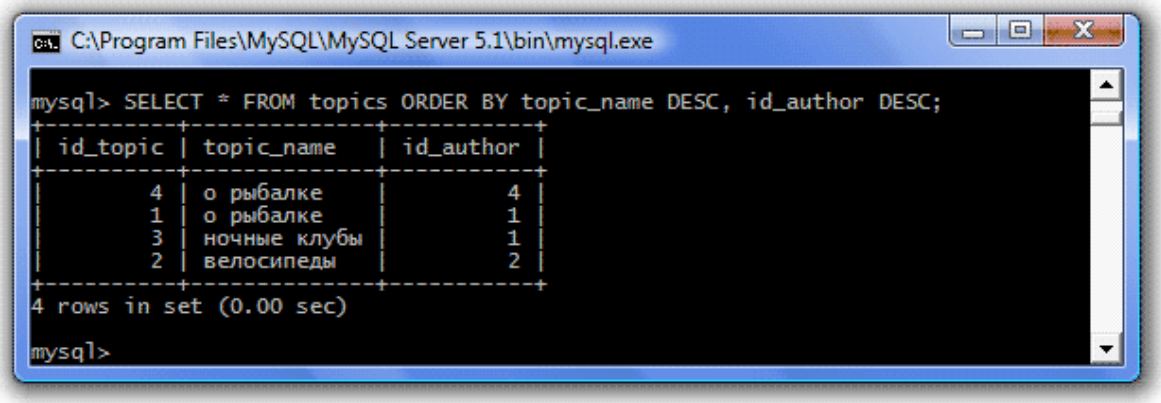


По умолчанию сортировка идет по возрастанию, но это можно изменить, добавив ключевое слово *DESC*



Теперь наши данные отсортированы в порядке по убыванию.

Сортировку можно производить сразу по нескольким столбцам. Например, следующий запрос отсортирует данные по столбцу topic_name, и если в этом столбце будет несколько одинаковых строк, то в столбце id_author будет осуществлена сортировка по убыванию:



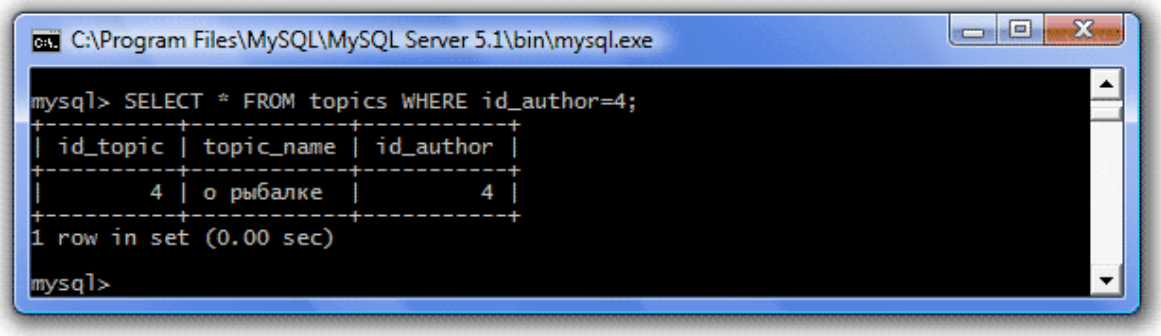
Сравните результат с результатом предыдущего запроса.

Очень часто нам не нужна вся информация из таблицы. Например, мы хотим узнать, какие темы были созданы пользователем sveta (id=4). Для этого в SQL есть ключевое слово *WHERE*, синтаксис у такого запроса следующий:

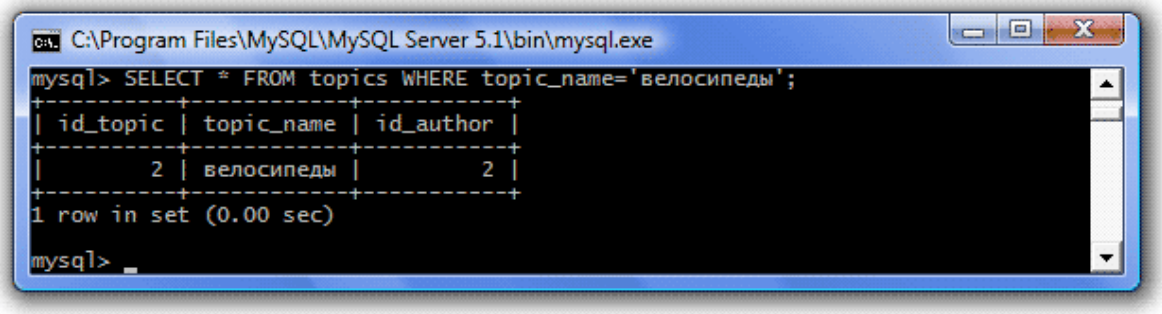
```
SELECT имя_столбца FROM имя_таблицы WHERE условие;
```

Для нашего примера условием является идентификатор пользователя, т.е. нам нужны только те строки, в столбце id_author которых стоит 4 (идентификатор пользователя sveta):

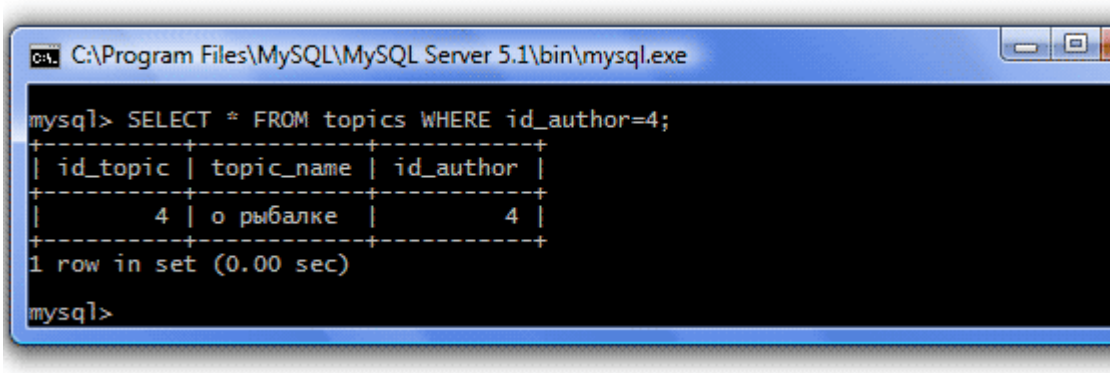
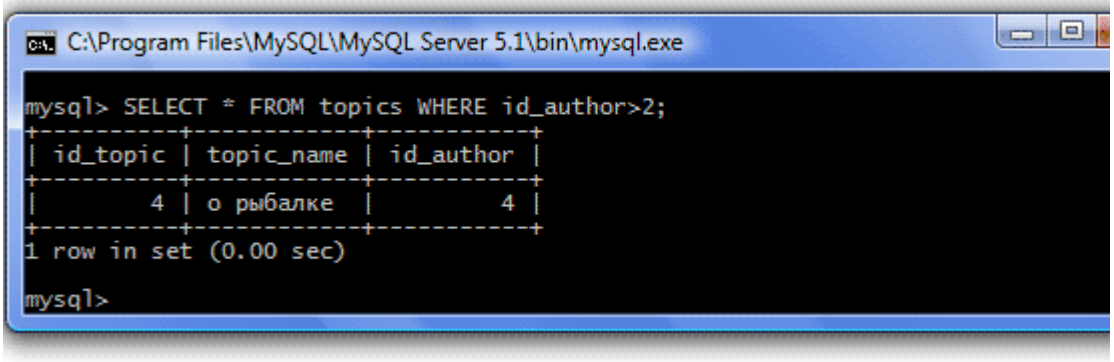
SELECT * FROM topics WHERE id_author=4;

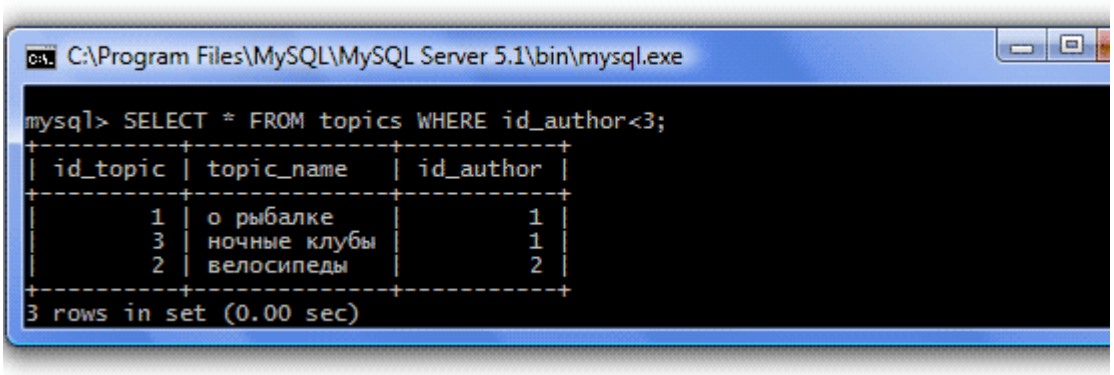
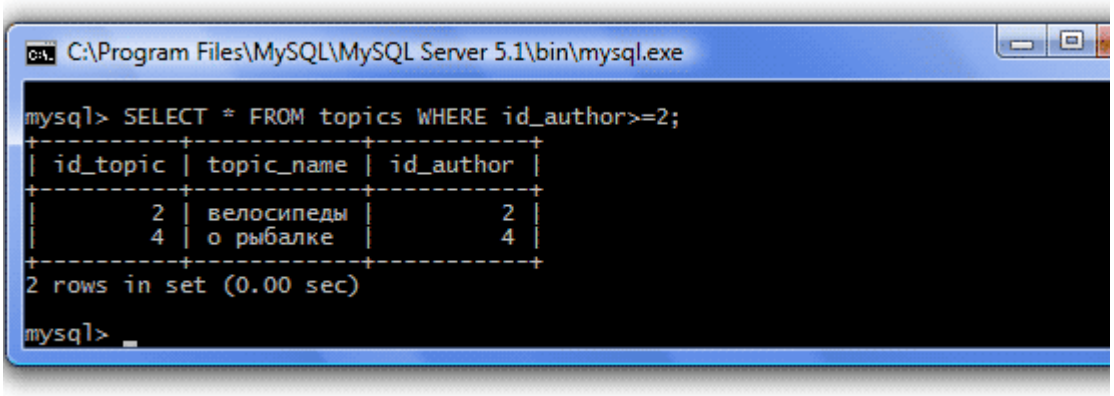


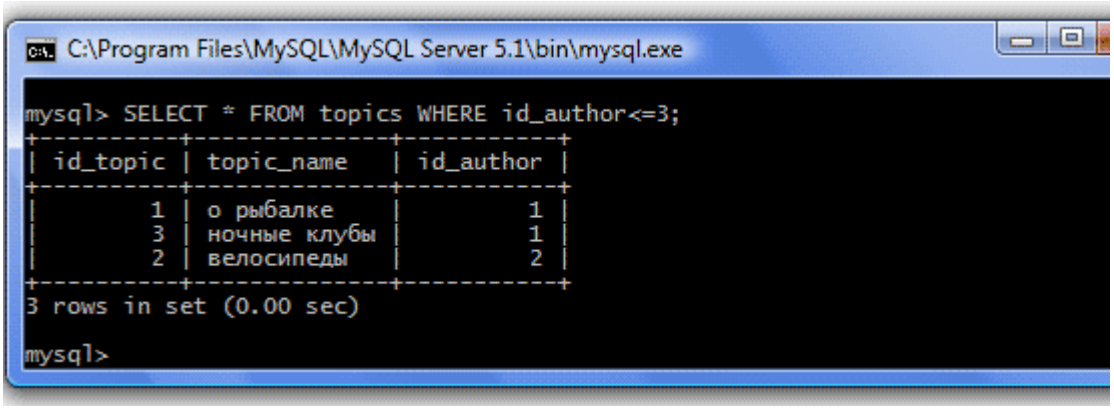
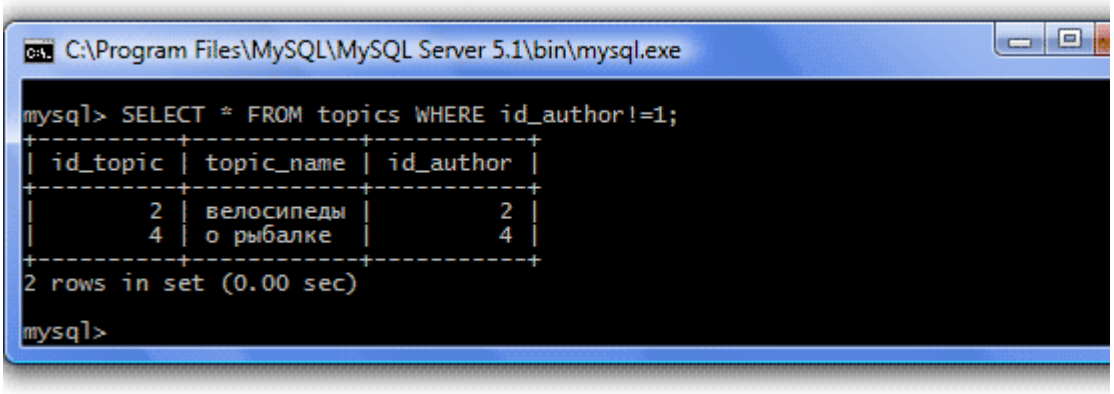
Или мы хотим узнать, кто создал тему "велосипеды":

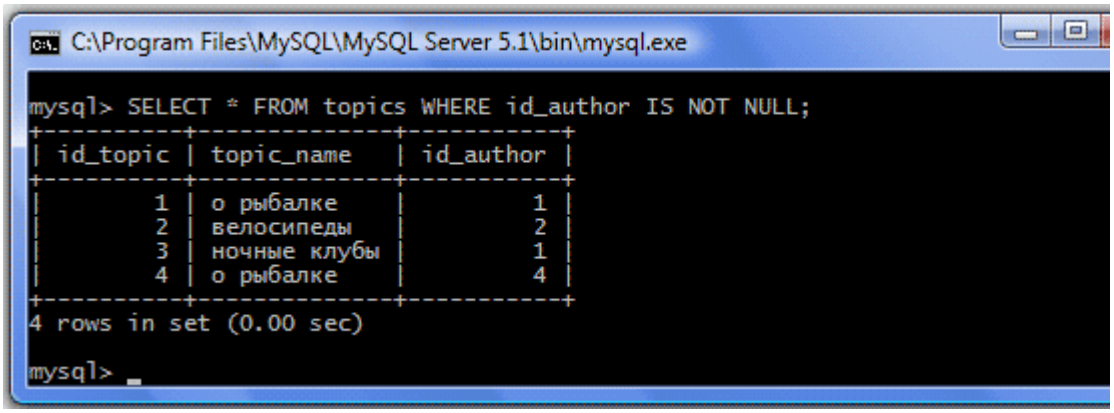
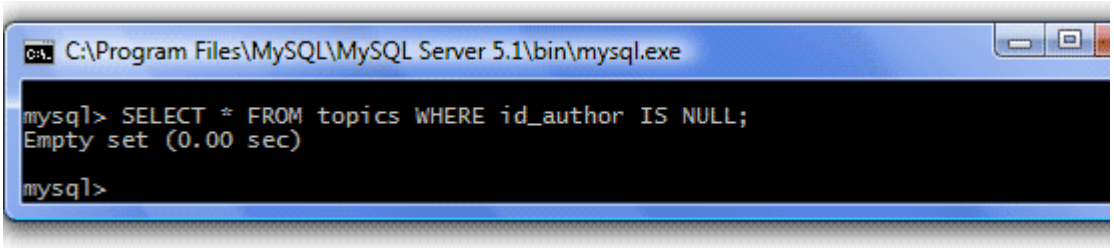


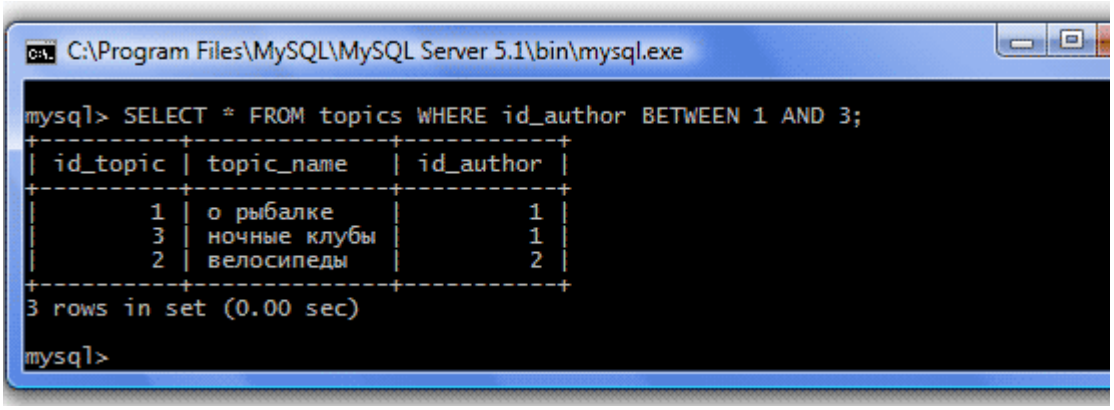
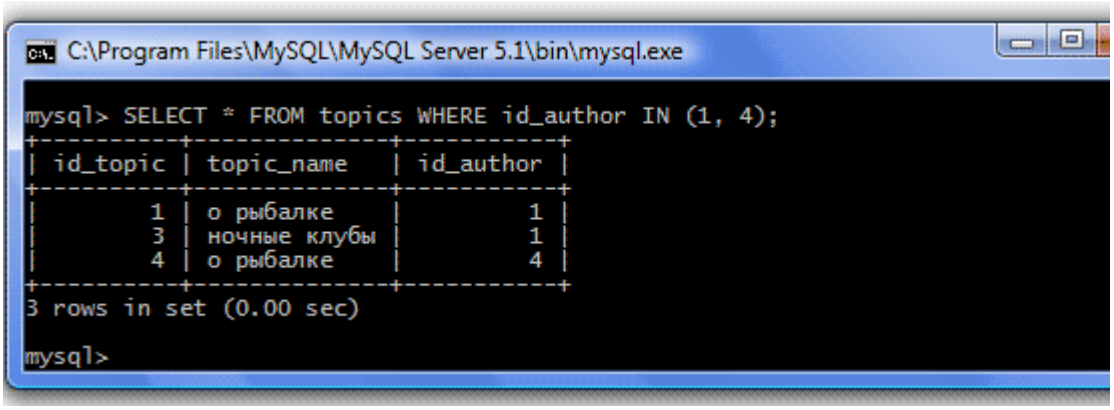
Конечно, было бы удобнее, чтобы вместо id автора, выводилось его имя, но имена хранятся в другой таблице. В последующих уроках мы узнаем, как выбирать данные из нескольких таблиц. А пока узнаем, какие условия можно задавать, используя ключевое слово WHERE.

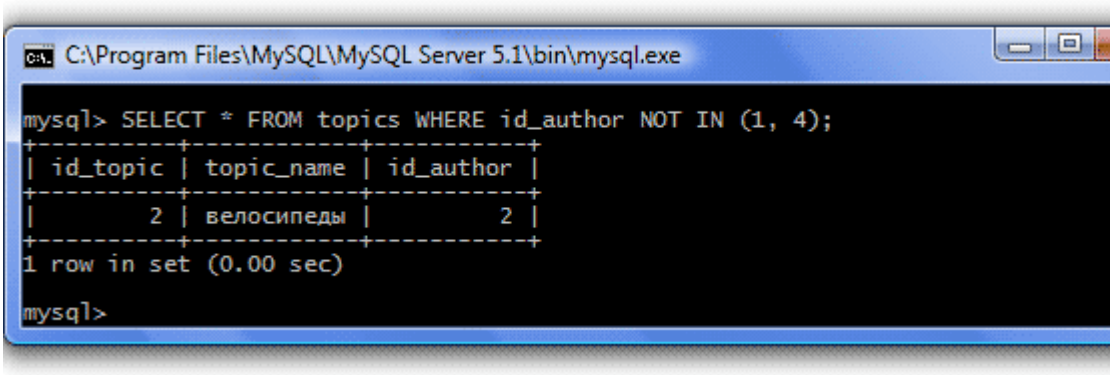
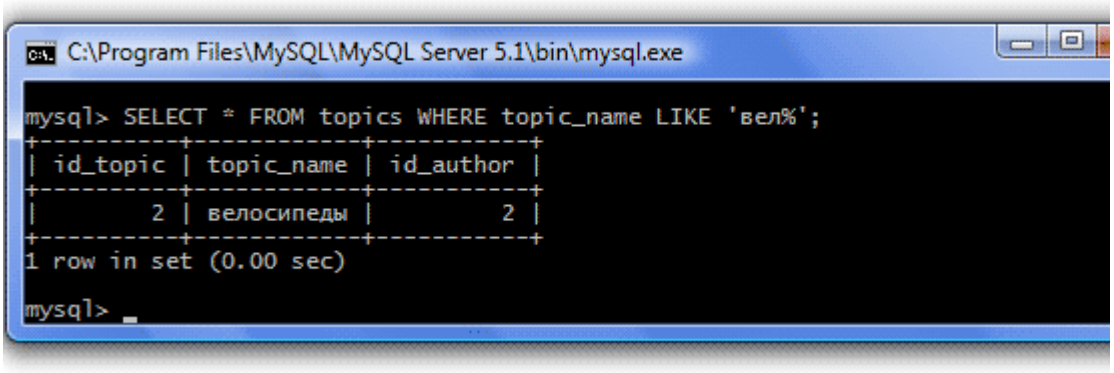
Оператор	Описание
= (равно)	Отбираются значения равные указанному *Пример:* SELECT * FROM topics WHERE id_author=4; *Результат:* 
> (больше)	Отбираются значения больше указанного *Пример:* SELECT * FROM topics WHERE id_author>2; *Результат:* 

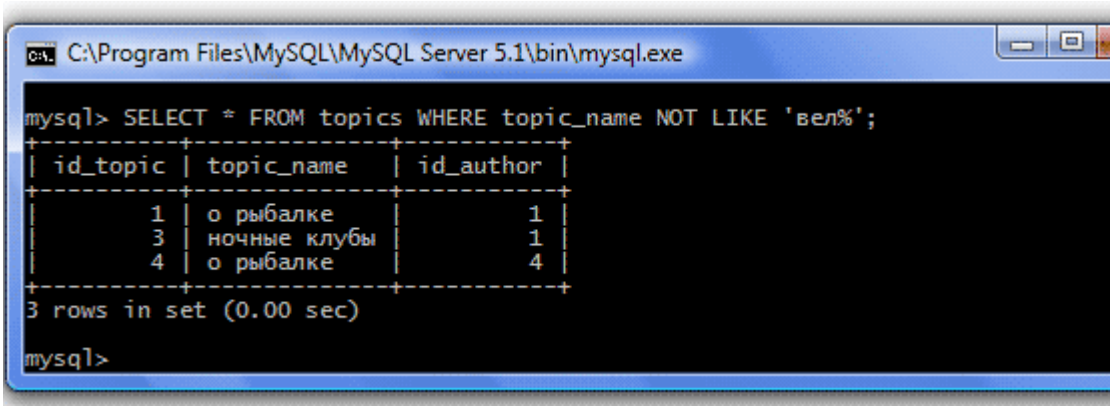
< (меньше)	Отбираются значения меньше указанного *Пример:* SELECT * FROM topics WHERE id_author<3; *Результат:* 
>= (больше или равно)	Отбираются значения большие и равные указанному *Пример:* SELECT * FROM topics WHERE id_author>=2; *Результат:* 
<= (меньше или равно)	Отбираются значения меньшие и равные указанному *Пример:* SELECT * FROM topics WHERE id_author<=3; *Результат:*

	
!= (не равно)	Отбираются значения не равные указанному *Пример:* SELECT * FROM topics WHERE id_author!=1; *Результат:* 
IS NOT NULL	Отбираются строки, имеющие значения в указанном поле *Пример:* SELECT * FROM topics WHERE id_author IS NOT NULL; *Результат:*

	
IS NULL	Отбираются строки, не имеющие значения в указанном поле *Пример:* SELECT * FROM topics WHERE id_author IS NULL; *Результат:*  Empty set - нет таких строк.
BETWEEN (между)	Отбираются значения, находящиеся между указанными *Пример:* SELECT * FROM topics WHERE id_author BETWEEN 1 AND 3; *Результат:*

	
IN (значение содержится)	Отбираются значения, соответствующие указанным *Пример:* SELECT * FROM topics WHERE id_author IN (1, 4); *Результат:* 
NOT IN (значение не содержится)	Отбираются значения, кроме указанных *Пример:* SELECT * FROM topics WHERE id_author NOT IN (1, 4); *Результат:*

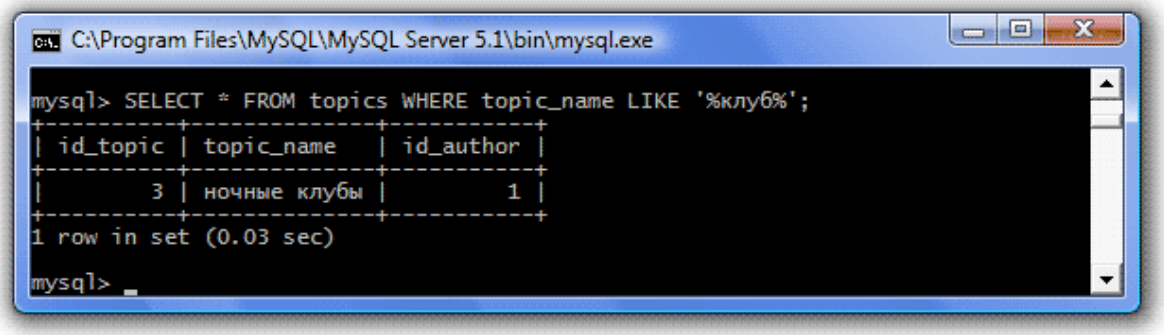
	
LIKE (соответствие)	Отбираются значения, соответствующие образцу *Пример:* SELECT * FROM topics WHERE topic_name LIKE 'вел%'; *Результат:*  Возможные метасимволы оператора LIKE будут рассмотрены ниже.
NOT LIKE (не соответствие)	Отбираются значения, не соответствующие образцу *Пример:* SELECT * FROM topics WHERE topic_name NOT LIKE 'вел%'; *Результат:*



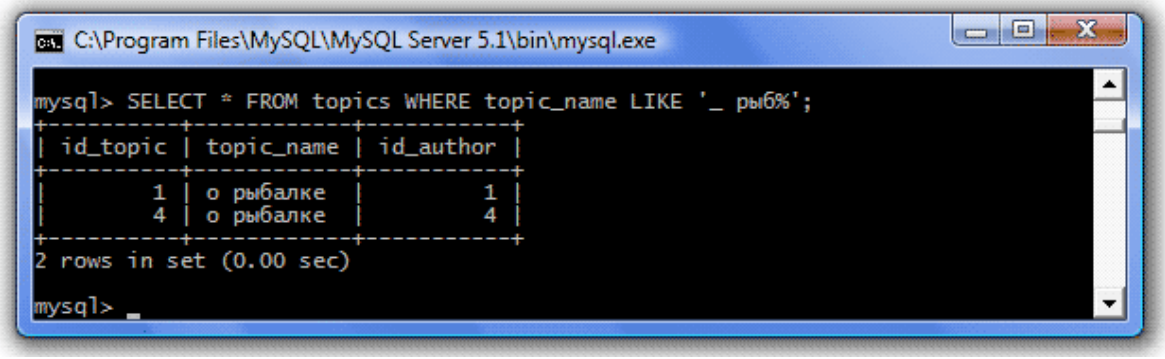
Метасимволы оператора LIKE

Поиск с использованием метасимволов может осуществляться только в текстовых полях.

Самый распространенный метасимвол - %. Он означает любые символы. Например, если нам надо найти слова, начинающиеся с букв "вел", то мы напишем LIKE 'вел%', а если мы хотим найти слова, которые содержат символы "клуб", то мы напишем LIKE '%клуб%'. Например:



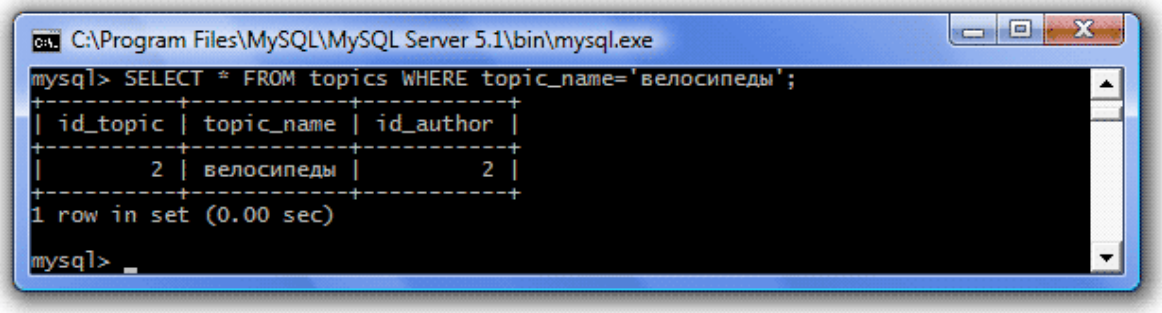
Еще один часто используемый метасимвол - _. В отличие от %, который обозначает несколько или ни одного символа, нижнее подчеркивание обозначает ровно один символ. Например:



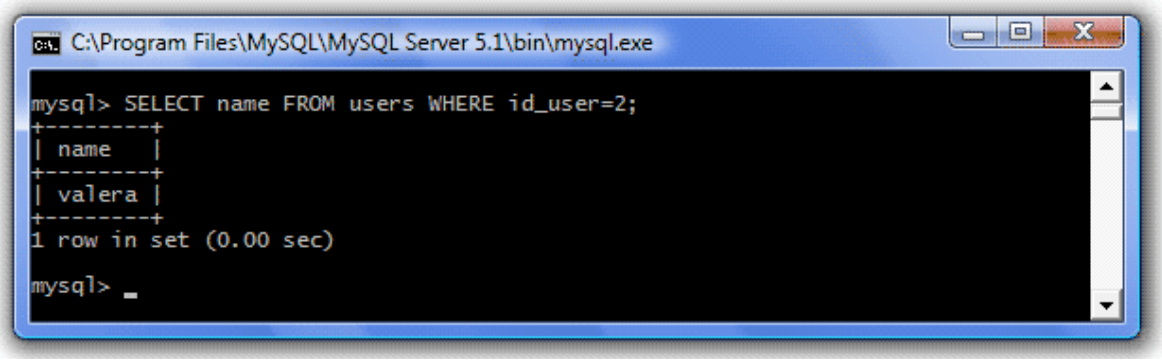
Обратите внимание на пробел между метасимволом и "рыб", если его пропустить, то запрос не сработает, т.к. метасимвол _ обозначает ровно один символ, а пробел - это тоже символ.

Вложенные запросы

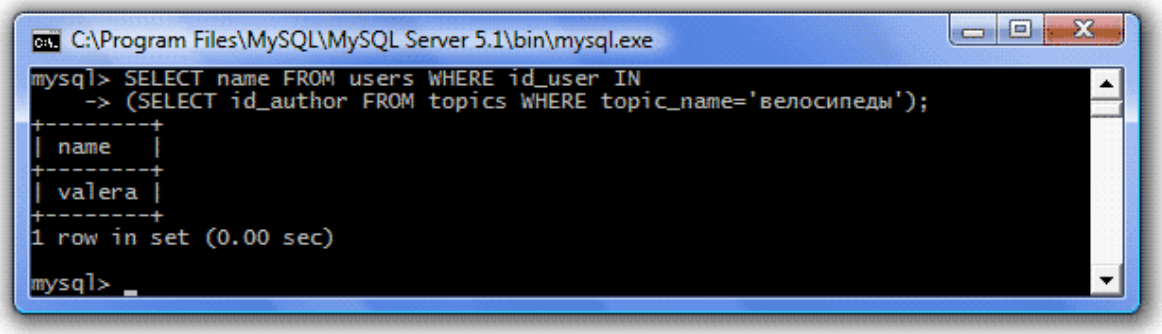
В прошлом уроке мы столкнулись с одним неудобством. Когда мы хотели узнать, кто создал тему "велосипеды", и делали соответствующий запрос:



Вместо имени автора, мы получали его идентификатор. Это и понятно, ведь мы делали запрос к одной таблице - Темы, а имена авторов тем хранятся в другой таблице - Пользователи. Поэтому, узнав идентификатор автора темы, нам надо сделать еще один запрос - к таблице Пользователи, чтобы узнать его имя:



В SQL предусмотрена возможность объединять такие запросы в один путем превращения одного из них в подзапрос (вложенный запрос). Итак, чтобы узнать, кто создал тему "велосипеды", мы сделаем следующий запрос:



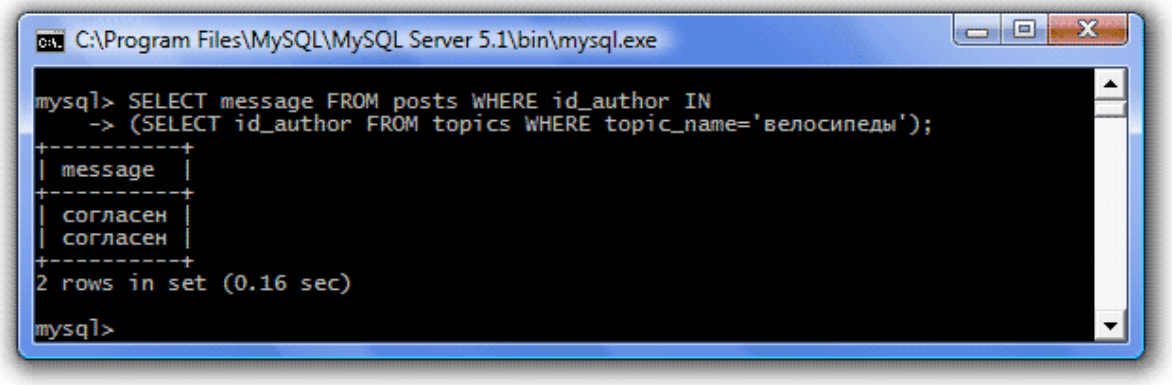
То есть, после ключевого слова *WHERE*, в условие мы записываем еще один запрос. MySQL сначала обрабатывает подзапрос, возвращает id_author=2, и это значение передается в предложение *WHERE* внешнего запроса.

В одном запросе может быть несколько подзапросов, синтаксис у такого запроса следующий:

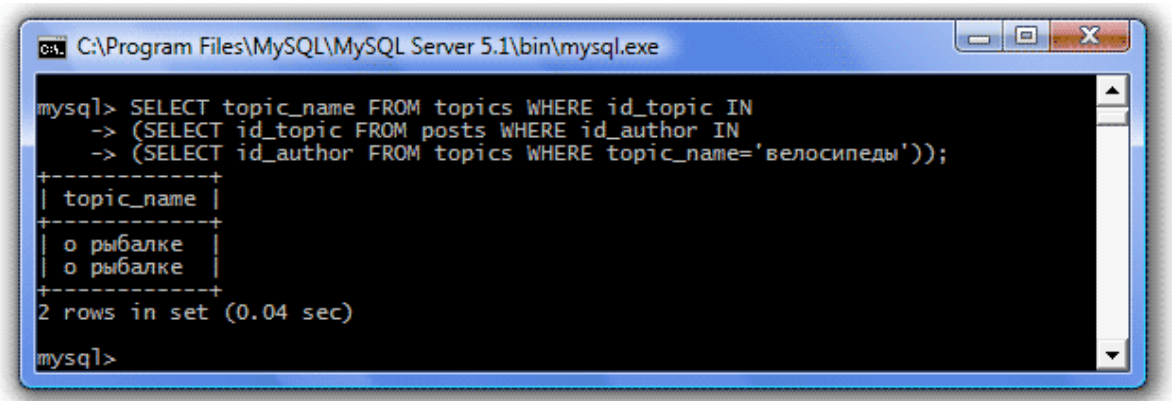
```
SELECT имя_столбца FROM имя_таблицы WHERE часть условия IN
  (SELECT имя_столбца FROM имя_таблицы WHERE часть условия IN
    (SELECT имя_столбца FROM имя_таблицы WHERE условие)
  )
;
```

Обратите внимание, что подзапросы могут выбирать только один столбец, значения которого они будут возвращать внешнему запросу. Попытка выбрать несколько столбцов приведет к ошибке.

Давайте для закрепления составим еще один запрос, узнаем, какие сообщения на форуме оставлял автор темы "велосипеды":



Теперь усложним задачу, узнаем, в каких темах оставлял сообщения автор темы "велосипеды":



Давайте разберемся, как это работает.

- Сначала MySQL выполнит самый глубокий запрос:

  ```
  SELECT id_author FROM topics WHERE topic_name='велосипеды'
  ```

- Полученный результат (id_author=2) передаст во внешний запрос, который примет вид:

  ```
  SELECT id_topic FROM posts WHERE id_author IN (2);
  ```

- Полученный результат (id_topic:4,1) передаст во внешний запрос, который примет вид:

```
SELECT topic_name FROM topics WHERE id_topic IN (4,1);
```

- И выдаст окончательный результат (topic_name: о рыбалке, о рыбалке). Т.е. автор темы "велосипеды" оставлял сообщения в теме "О рыбалке", созданной Сергеем (id=1) и в теме "О рыбалке", созданной Светой (id=4).

Вот собственно и все, что хотелось сказать о вложенных запросах. Хотя, есть два момента, на которые стоит обратить внимание:

- Не рекомендуется создавать запросы со степенью вложения больше трех. Это приводит к увеличению времени выполнения и к сложности восприятия кода.

- Приведенный синтаксис вложенных запросов, скорее наиболее употребительный, но вовсе не единственный. Например, мы могли бы вместо запроса

```
SELECT name FROM users WHERE id_user IN
(SELECT id_author FROM topics WHERE topic_name='велосипеды');
```

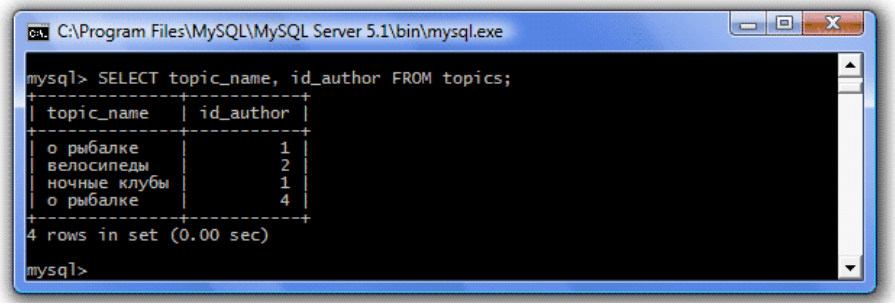
написать

```
SELECT name FROM users WHERE id_user =
(SELECT id_author FROM topics WHERE topic_name='велосипеды');
```

Т.е. мы можем использовать любые операторы, используемые с ключевым словом WHERE

3.4. Объединение таблиц (внутреннее объединение)

Предположим, мы хотим узнать, какие темы, и какими авторами были созданы. Для этого проще всего обратиться к таблице Темы (topics):

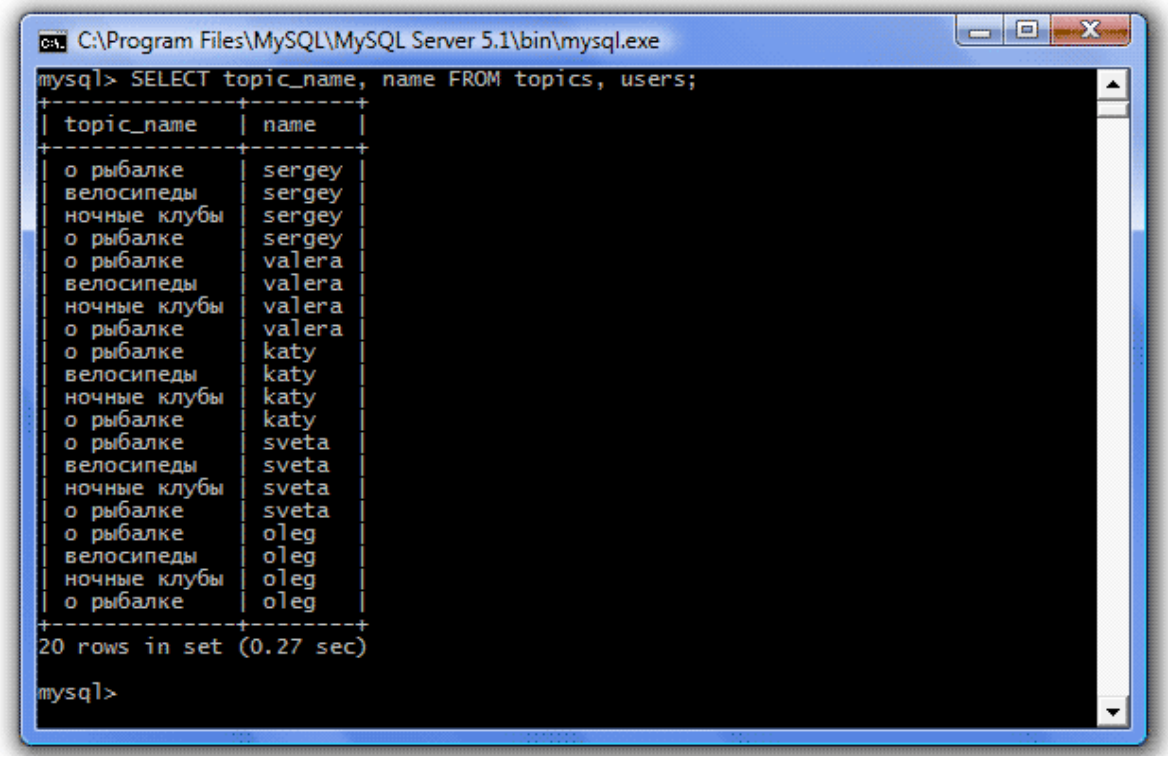


Но, что если нам необходимо, чтобы в ответе на запрос были не идентификаторы авторов, а их имена? Вложенные запросы нам не помогут, т.к. в конечном итоге они выдают данные из одной таблицы. А нам надо получить данные из двух таблиц (Темы и Пользователи) и объединить их в одну. Запросы, которые позволяют это сделать, в SQL называются *Объединениями*.

Синтаксис самого простого объединения следующий:

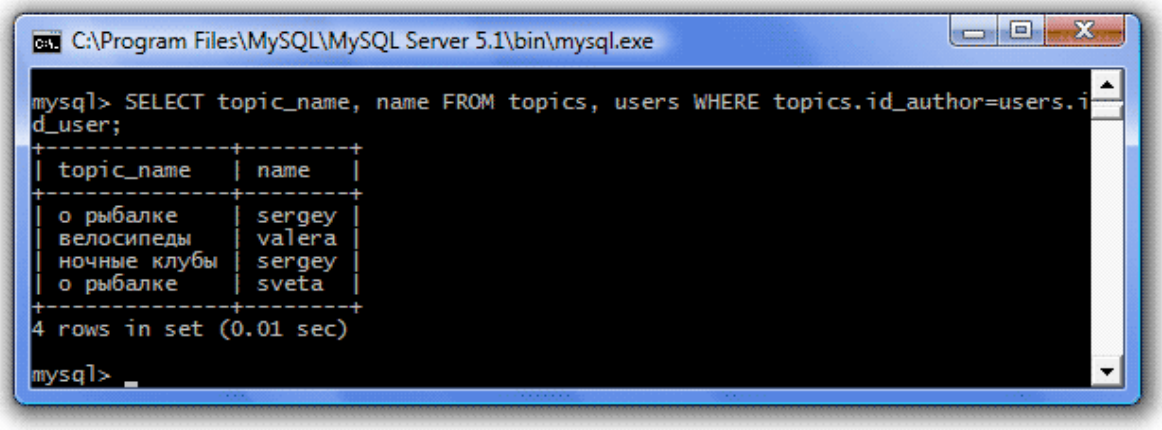
```
SELECT имена_столбцов_таблицы_1, имена_столбцов_таблицы_2 FROM имя_таблицы_1, имя_таблицы_2;
```

Давайте создадим простое объединение:

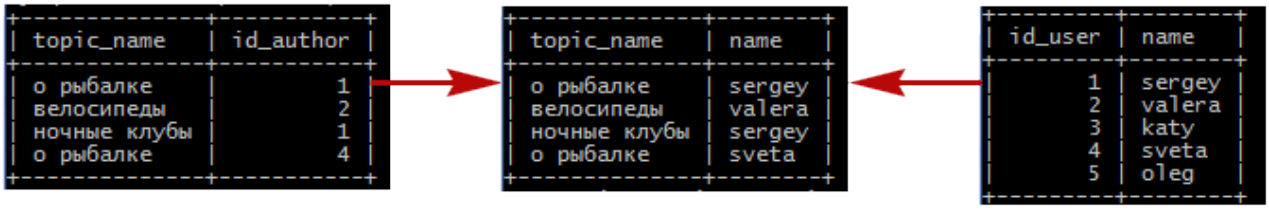


Получилось не совсем то, что мы ожидали. Такое объединение научно называется декартовым произведением, когда каждой строке первой таблицы ставится в соответствие каждая строка второй таблицы. Возможно, бывают случаи, когда такое объединение полезно, но это явно не наш случай.

Чтобы результирующая таблица выглядела так, как мы хотели, необходимо указать условие объединения. Мы связываем наши таблицы по идентификатору автора, это и будет нашим условием. Т.е. мы укажем в запросе, что необходимо выводить только те строки, в которых значения поля id_author таблицы topics совпадают со значениями поля id_user таблицы users:



На схеме будет понятнее:



Т.е. мы в запросе сделали следующее условие: если в обеих таблицах есть одинаковые идентификаторы, то строки с этим идентификатором необходимо объединить в одну результирующую строку.

Обратите внимание на две вещи:

- Если в одной из объединяемых таблиц есть строка с идентификатором, которого нет в другой объединяемой таблице, то в результирующей таблице строки с таким идентификатором не будет. В нашем примере есть пользователь Oleg (id=5), но он не создавал тем, поэтому в результате запроса его нет.

- При указании условия название столбца пишется после названия таблицы, в которой этот столбец находится (через точку). Это сделано во избежание путаницы, ведь столбцы в разных таблицах могут иметь одинаковые названия, и MySQL может не понять, о каких конкретно столбцах идет речь.

Вообще, корректный синтаксис объединения с условием выглядит так:

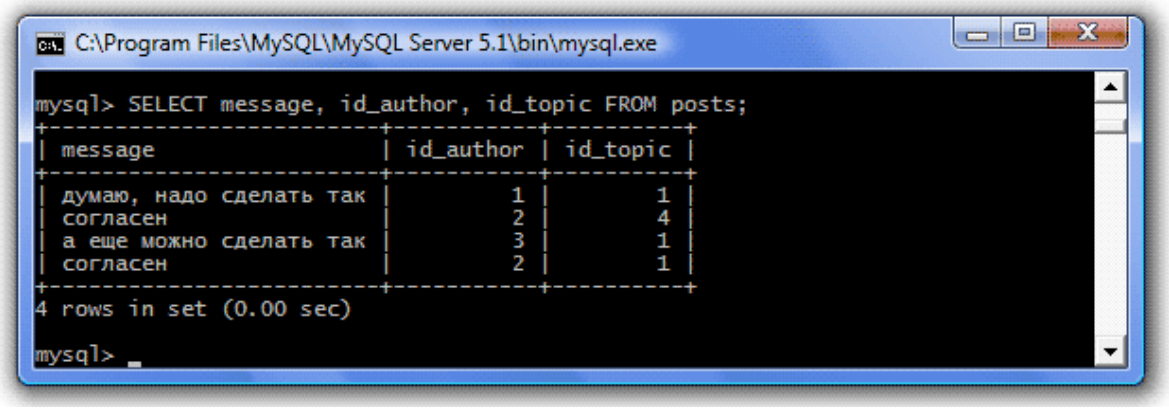
```
SELECT имя_таблицы_1.имя_столбца1_таблицы_1,
    имя_таблицы_1.имя_столбца2_таблицы_1,
    имя_таблицы_2.имя_столбца1_таблицы_2,
    имя_таблицы_2.имя_столбца2_таблицы_2
```

```
FROM
    имя_таблицы_1, имя_таблицы_2
WHERE
    имя_таблицы_1.имя_столбца_по_которому_объединяем =
    имя_таблицы_2.имя_столбца_по_которому_объединяем;
```

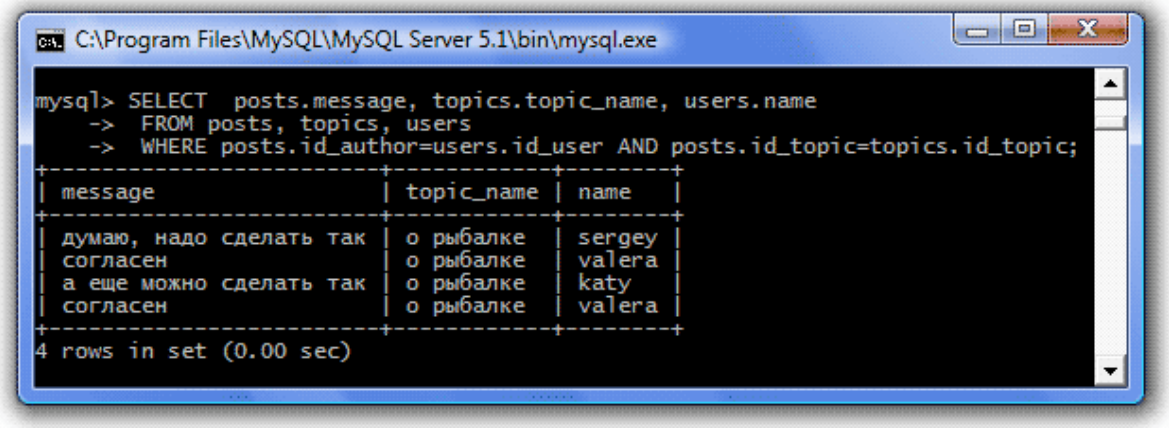
Если имя столбца уникально, то название таблицы можно опустить (как мы делали в примере), но делать это не рекомендуется.

Как вы понимаете, объединения дают возможность выбирать любую информацию из любых таблиц, причем объединяемых таблиц может быть и три, и четыре, да и условие для объединения может быть не одно.

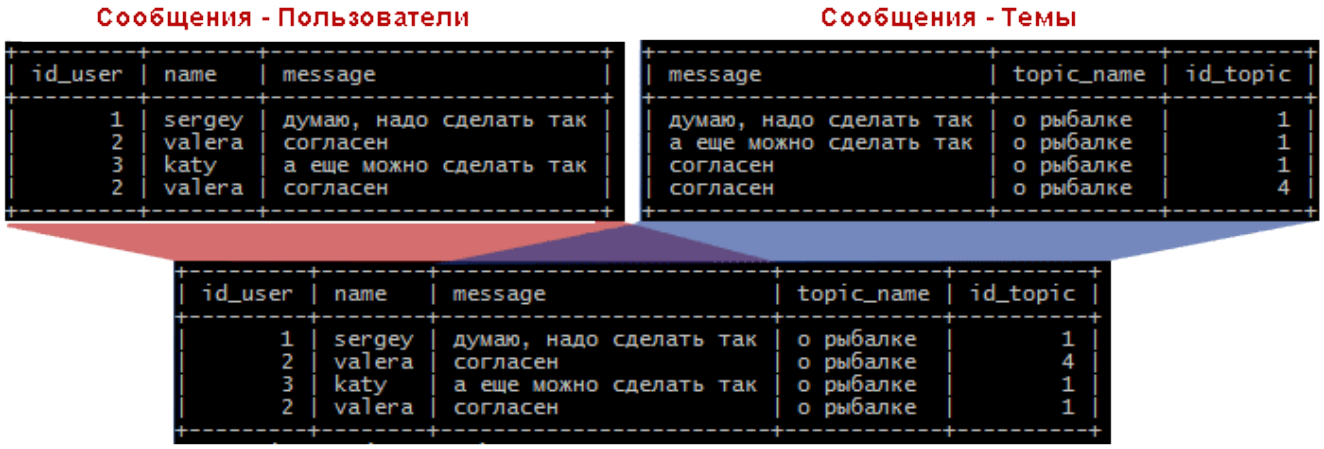
Для примера давайте создадим запрос, который покажет нам все сообщения, к каким темам они относятся и авторов этих сообщений. Конечно, вся эта информация хранится в таблице Сообщения (posts):



Но чтобы вместо идентификаторов отображались имена и названия, нам придется сделать объединение трех таблиц:



Т.е. мы объединили таблицы Сообщения и Пользователи условием posts.id_author=users.id_user, а таблицы Сообщения и Темы - условием posts.id_topic=topics.id_topic

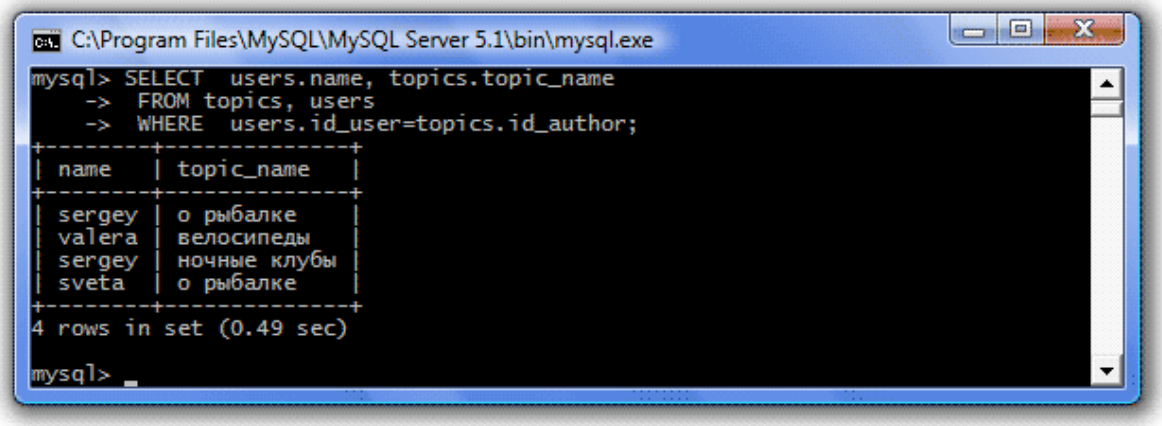


Объединения, которые мы сегодня рассматривали, называются *Внутренними объединениями*. Такие объединения связывают строки одной таблицы со строками другой таблицы (а может еще и третьей таблицы). Но бывают ситуации, когда необходимо, чтобы в результат были включены строки, не имеющие связанных. Например, когда мы создавали запрос, какие темы и какими авторами были созданы, пользователь Oleg в результирующую таблицу не попал, т.к. тем не создавал, а потому и связанной строки в объединяемой таблице не имел.

Поэтому, если нам потребуется составить несколько иной запрос - вывести всех пользователей и темы, которые они создавали, если таковые имеются - то нам придется воспользоваться *Внешним объединением*, позволяющим выводить все строки одной таблицы и имеющиеся связанные с ними строки из другой таблицы.

3.4.1. Объединение таблиц (внешнее объединение)

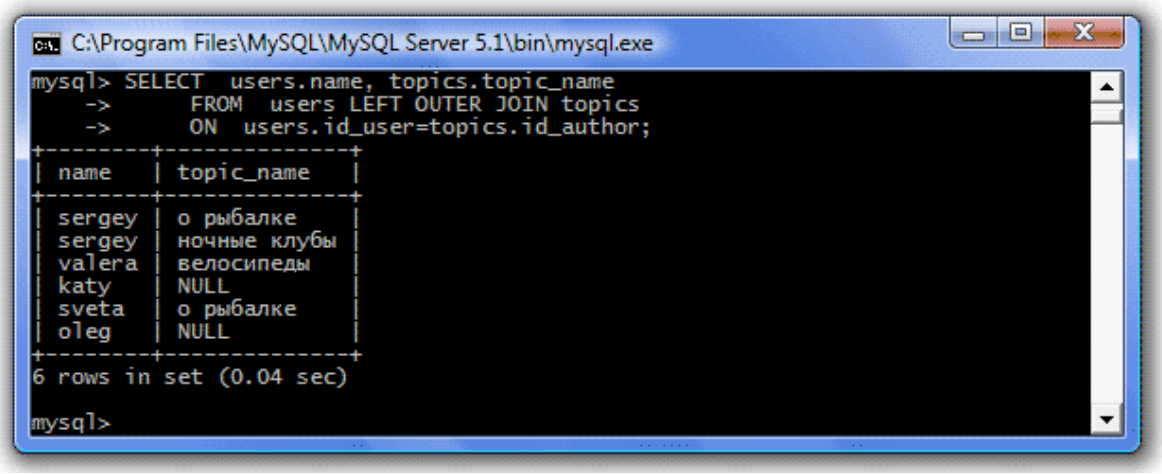
Итак, в продолжение прошлого урока, нам надо вывести всех пользователей и темы, которые они создавали, если таковые имеются. Если мы воспользуемся внутренним объединением, рассмотренным на прошлом уроке, то получим в итоге следующее:



То есть в результирующей таблице есть только те пользователи, которые создавали темы. А нам надо, чтобы выводились все имена. Для этого мы немного изменим запрос:

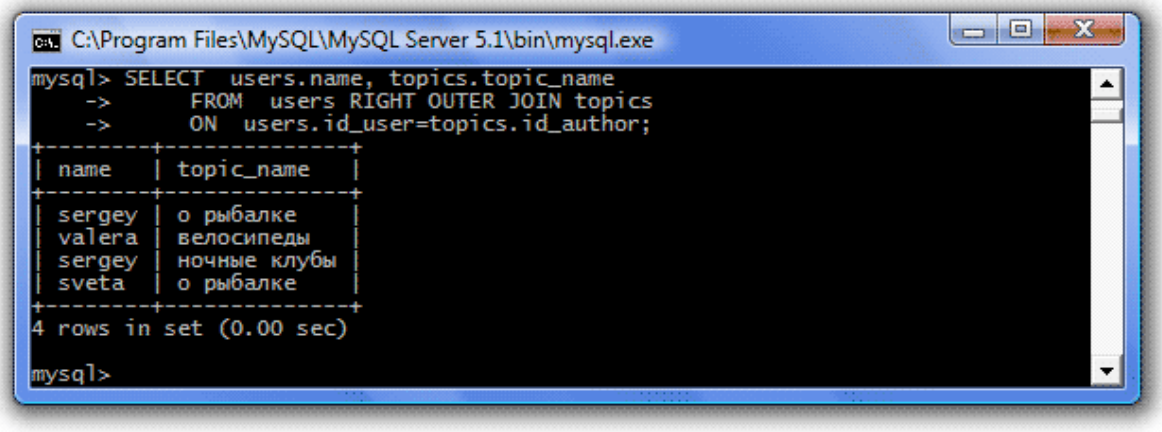
```
SELECT  users.name, topics.topic_name
FROM  users LEFT OUTER JOIN topics
ON  users.id_user=topics.id_author;
```

И получим желаемый результат - все пользователи и темы, ими созданные. Если пользователь не создавал тему, но в соответствующем столбце стоит значение NULL.



Итак, мы добавили в наш запрос ключевое слово - *LEFT OUTER JOIN*, указав тем самым, что из таблицы слева надо взять все строки, и поменяли ключевое слово *WHERE* на *ON*. Кроме ключевого слова *LEFT OUTER JOIN* может быть использовано ключевое слово *RIGHT OUTER JOIN*. Тогда будут выбираться все строки из правой таблицы и имеющиеся связанные с ними из левой таблицы. И наконец, возможно полное внешнее объединение, которое извлечет все строки из обеих таблиц и свяжет между собой те, которые могут быть связаны. Ключевое слово для полного внешнего объединения - *FULL OUTER JOIN*.

Давайте поменяем в нашем запросе левостороннее объединение на правостороннее:



Как видите, теперь у нас есть все темы (все строки из правой таблицы), а вот пользователи только те, которые темы создавали (т.е. из левой таблицы выбираются только те строки, которые связаны с правой таблицей).

К сожалению полное объединение СУБД MySQL не поддерживает.

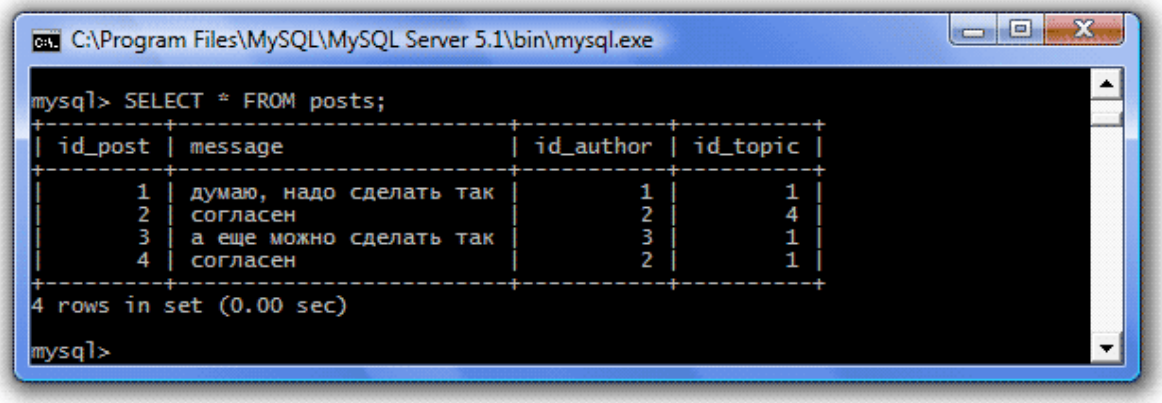
Подведем итог этого короткого урока. Синтаксис для внешнего объединения следующий:

```
SELECT имя_таблицы_1.имя_столбца, имя_таблицы_2.имя_столбца
FROM имя_таблицы_1 ТИП ОБЪЕДИНЕНИЯ имя_таблицы_2
ON условие_объединения;
```

где ТИП ОБЪЕДИНЕНИЯ - либо LEFT OUTER JOIN, либо RIGHT OUTER JOIN

3.5. Группировка записей и функция COUNT()

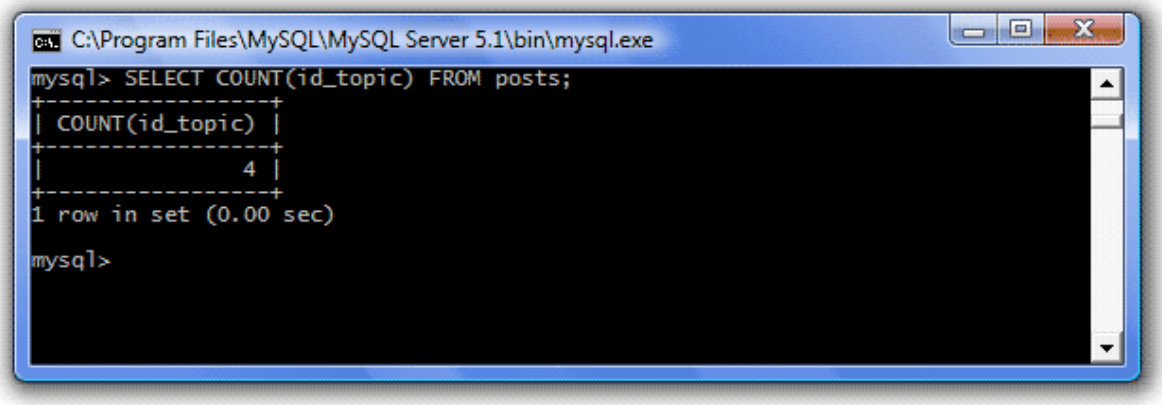
Давайте вспомним, какие сообщения и в каких темах у нас имеются. Для этого можно воспользоваться привычным запросом:



А что, если нам надо лишь узнать сколько сообщений на форуме имеется. Для этого можно воспользоваться встроенной функцией *COUNT()*. Эта функция подсчитывает число строк. Причем, если в качестве аргумента этой функции выступает *, то подсчитываются все строки таблицы. А если в качестве аргумента указывается имя столбца, то подсчитываются только те строки, которые имеют значение в указанном столбце.

В нашем примере оба аргумента дадут одинаковый результат, т.к. все столбцы таблицы имеют тип NOT NULL. Давайте напишем запрос, используя в качестве аргумента столбец id_topic:

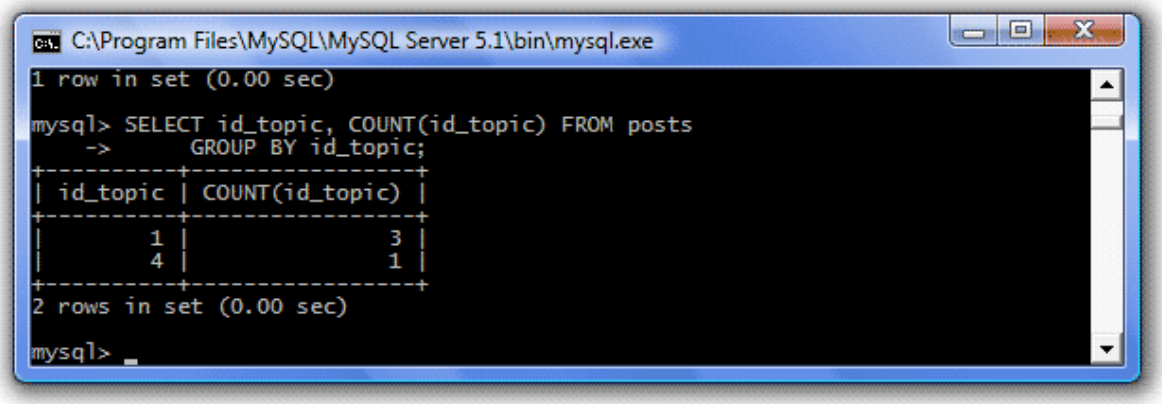
```
SELECT COUNT(id_topic) FROM posts;
```



Итак, в наших темах имеется 4 сообщения. Но что, если мы хотим узнать сколько сообщений имеется в каждой теме. Для этого нам понадобится сгруппировать наши сообщения по темам и вычислить для каждой группы количество сообщений. Для группировки в SQL используется оператор *GROUP BY*. Наш запрос теперь будет выглядеть так:

```
SELECT id_topic, COUNT(id_topic) FROM posts
GROUP BY id_topic;
```

Оператор *GROUP BY* указывает СУБД сгруппировать данные по столбцу id_topic (т.е. каждая тема - отдельная группа) и для каждой группы подсчитать количество строк:

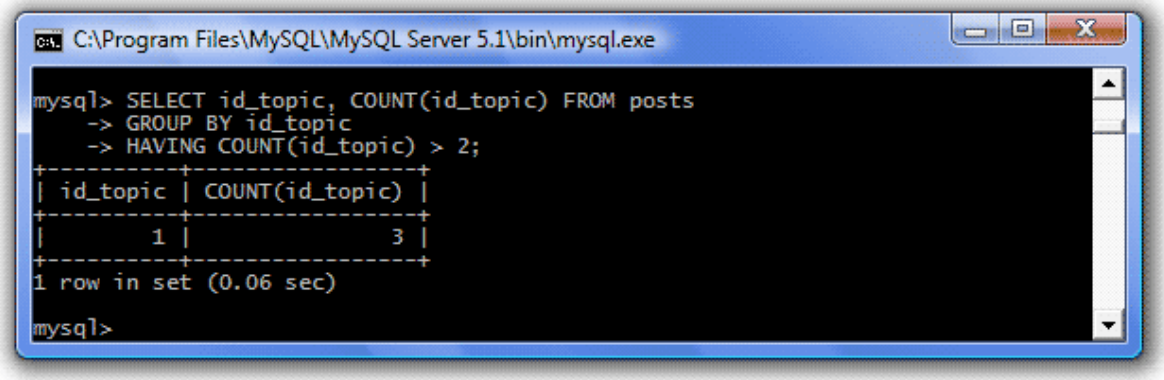


Ну вот, в теме с id=1 у нас 3 сообщения, а с id=4 - одно. Кстати, если бы в поле id_topic были возможны отсутствия значений, то такие строки были бы объединены в отдельную группу со значением NULL.

Предположим, что нас интересуют только те группы, в которых больше двух сообщений. В обычном запросе мы указали бы условие с помощью оператора *WHERE*, но этот оператор умеет работать только со строками, а для групп те же функции выполняет оператор *HAVING*:

```
SELECT id_topic, COUNT(id_topic) FROM posts
GROUP BY id_topic
HAVING COUNT(id_topic) > 2;
```

В результате имеем:



В уроке 4 мы рассматривали, какие условия можно задавать оператором *WHERE*, те же условия можно задавать и оператором *HAVING*, только надо запомнить, что *WHERE* фильтрует строки, а *HAVING* - группы.

Итак, сегодня мы узнали, как создавать группы и как подсчитать количество строк в таблице и в группах. Вообще вместе с оператором *GROUP BY* можно использовать и другие встроенные функции, но их мы будем изучать позже.

3.6. Редактирование, обновление и удаление данных

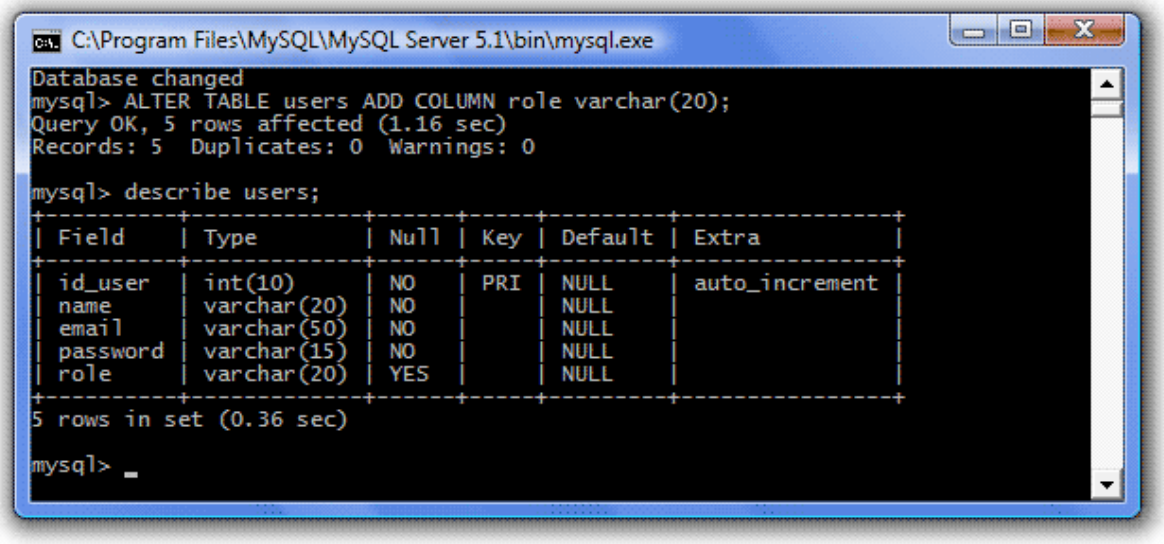
Предположим, мы решили, что нашему форуму нужны модераторы. Для этого в таблицу users надо добавить столбец с ролью пользователя. Для добавления столбцов в таблицу используется оператор *ALTER TABLE - ADD COLUMN*. Его синтаксис следующий:

```
ALTER TABLE имя_таблицы ADD COLUMN имя_столбца тип;
```

Давайте добавим столбец role в таблицу users:

```
ALTER TABLE users ADD COLUMN role varchar(20);
```

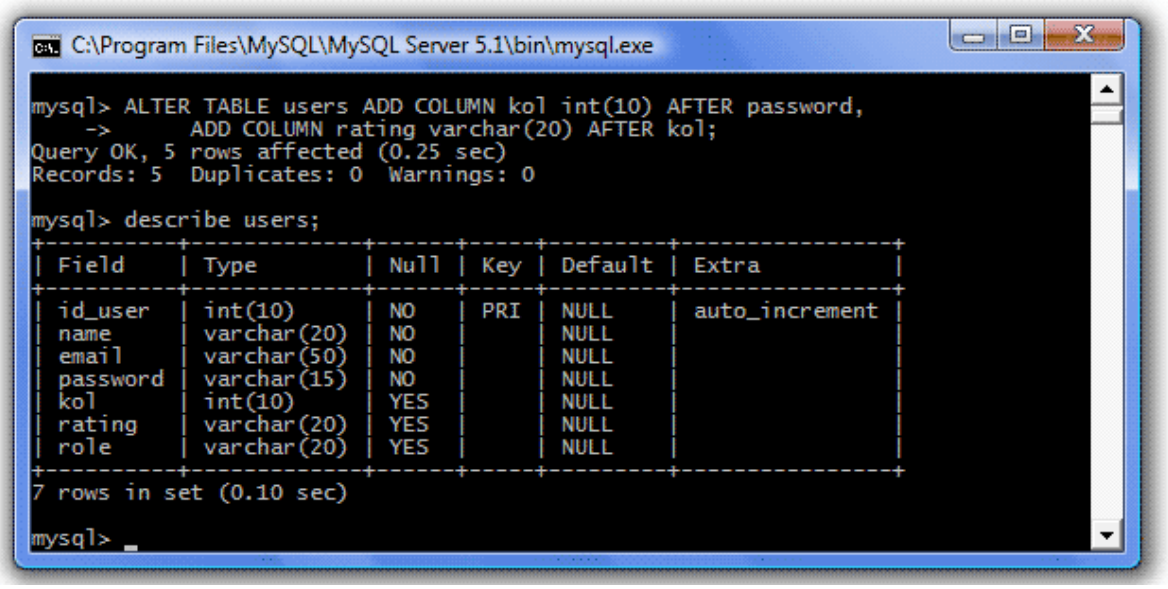
Столбец появился в конце таблицы:



Для того, чтобы указать местоположение столбца используются ключевые слова: *FIRST* - новый столбец будет первым, и *AFTER* - указывает после какого столбца поместить новый.

Давайте добавим еще два столбца: один - kol - количество оставленных сообщений, а другой - rating - рейтинг пользователя. Оба столбца вставим после поля password:

```
ALTER TABLE users ADD COLUMN kol int(10) AFTER password,
ADD COLUMN rating varchar(20) AFTER kol;
```

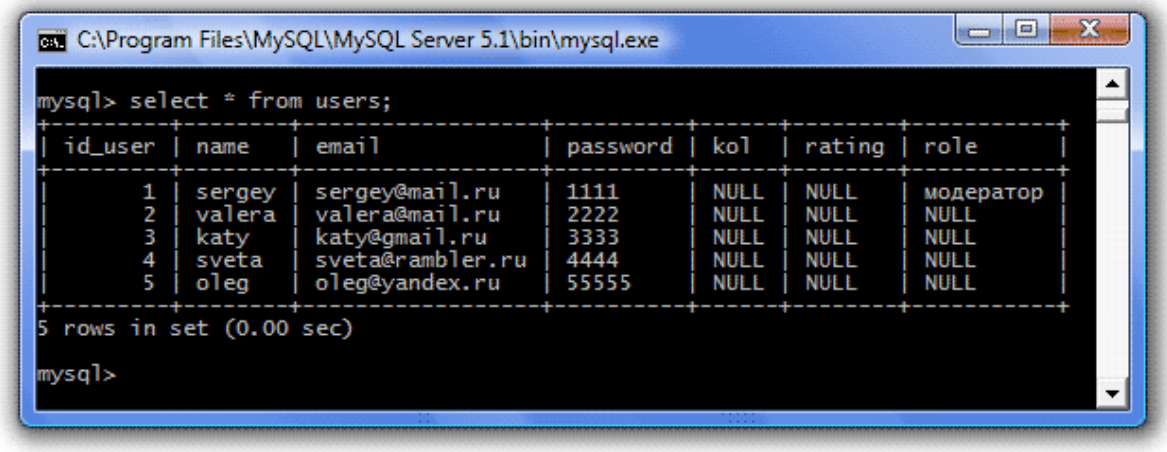


Теперь надо назначить роль модератора какому-нибудь пользователю, пусть это будет sergey с id=1. Для обновления уже существующих данных служит оператор *UPDATE*. Его синтаксис следующий:

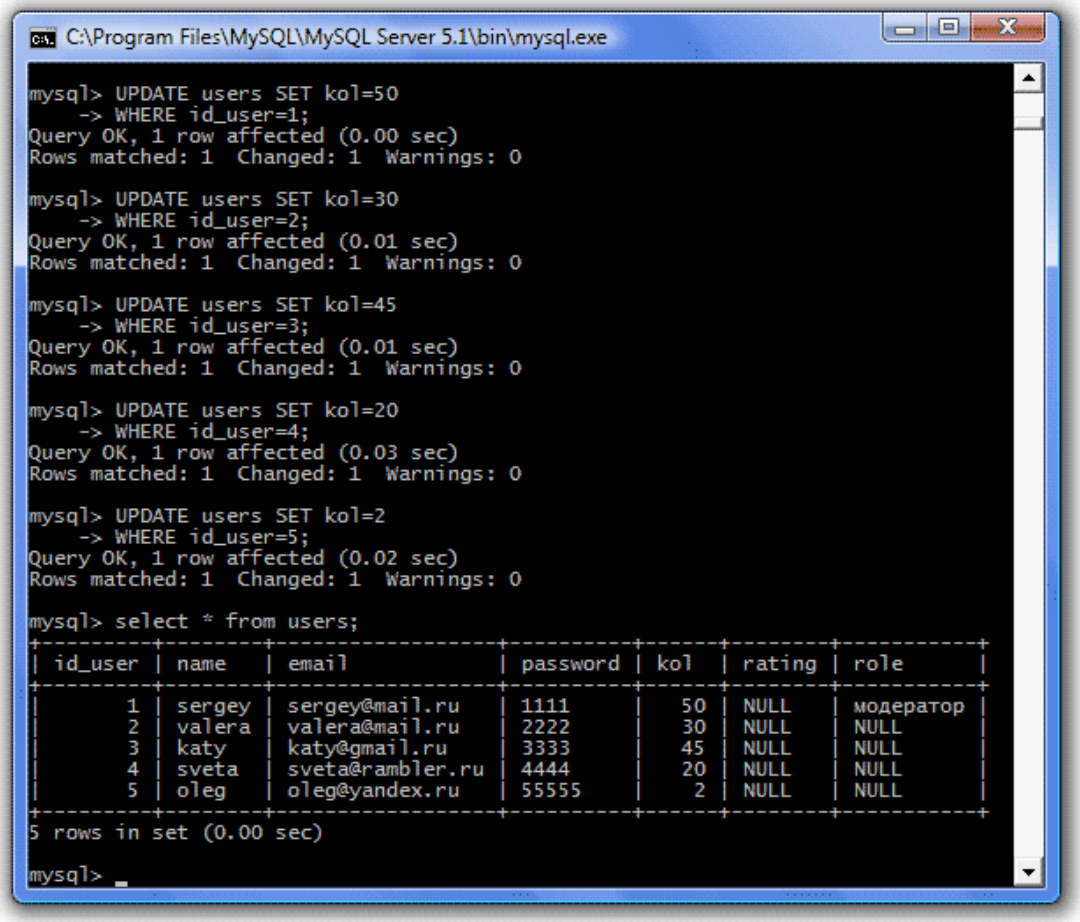
```
UPDATE имя_таблицы SET имя_столбца=значение_столбца
WHERE условие;
```

Давайте сделаем Сергея модератором:

```
UPDATE users SET role='модератор'
WHERE id_user=1;
```

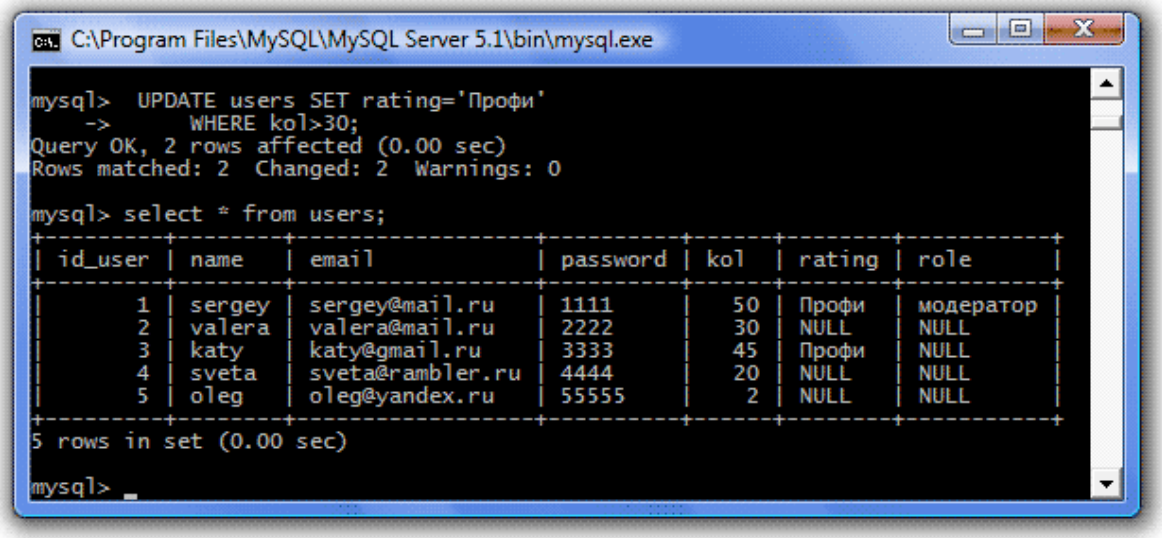


Изменять данные можно и сразу в нескольких строках и во всей таблице. Например, мы решили давать рейтинг в зависимости от количества оставленных пользователем сообщений. Давайте в нашу таблицу сначала внесем значения столбца kol так, как мы уже умеем:



А теперь давайте зададим рейтинг Профи тем, у кого количество сообщений больше 30:

```
UPDATE users SET rating='Профи'
WHERE kol>30;
```



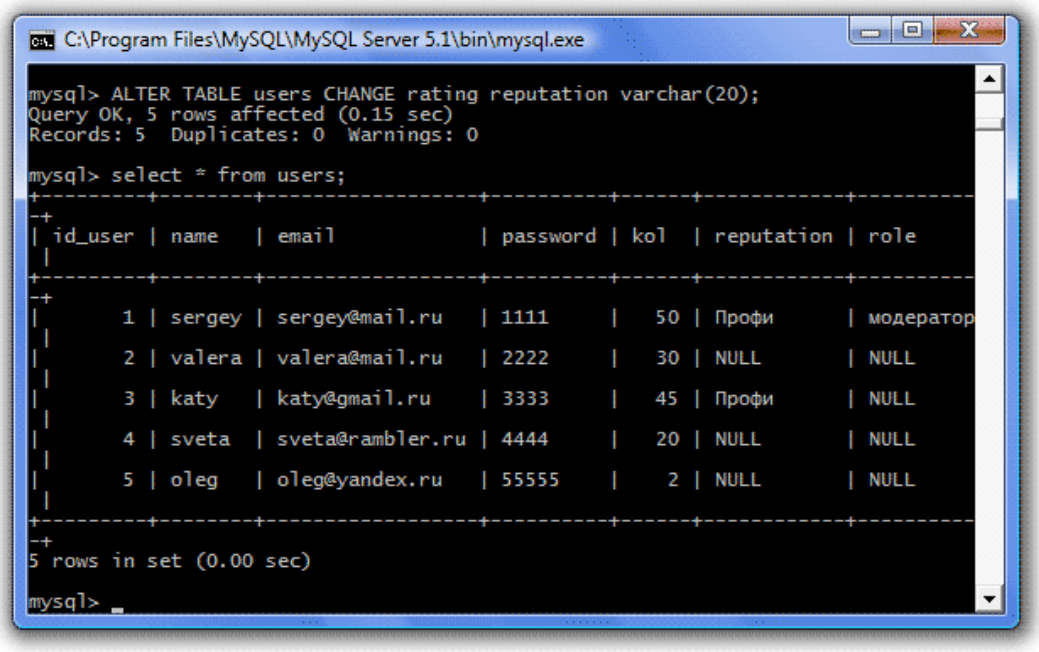
Данные изменились в двух строках, согласно заданному условию. Понятно, что если в запросе опустить условие, то данные будут обновлены во всех строках таблицы.

Предположим, что нам не нравится название Рейтинг у нашего столбца, и мы хотим переименовать столбец в Репутация - reputation. Для изменения имени существующего столбца используется оператор *CHANGE*. Его синтаксис следующий:

```
ALTER TABLE имя_таблицы CHANGE старое_имя_столбца новое_имя_столбца тип;
```

Давайте поменяем rating на reputation:

```
ALTER TABLE users CHANGE rating reputation varchar(20);
```



Обратите внимание, что тип столбца надо указывать даже, если он не меняется. Кстати, если нам понадобится изменить только тип столбца, то мы будем использовать оператор *MODIFY*. Его синтаксис следующий:

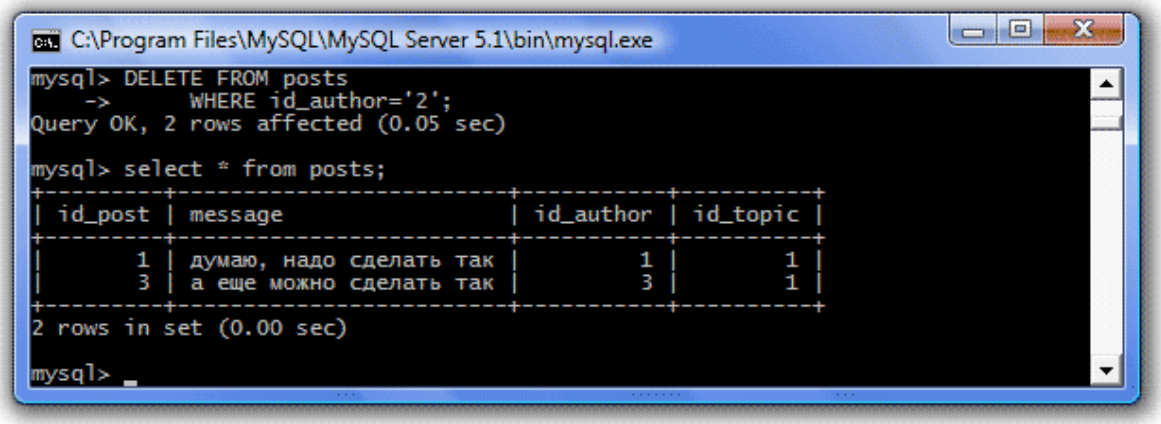
```
ALTER TABLE имя_таблицы MODIFY имя_столбца  новый_тип;
```

Последнее, что мы сегодня рассмотрим - оператор *DELETE*, который позволяет удалять строки из таблицы. Его синтаксис следующий:

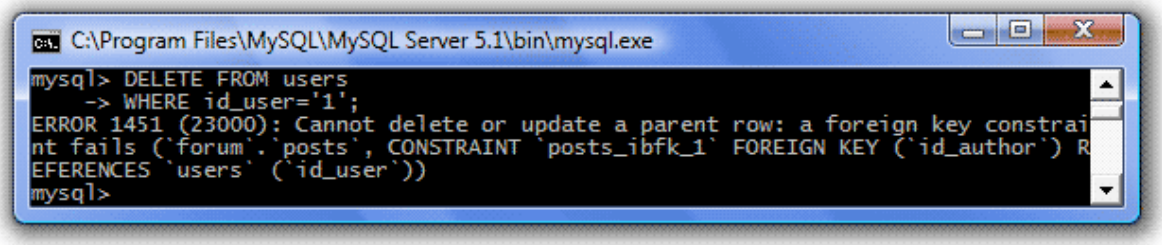
```
DELETE FROM имя_таблицы
WHERE условие;
```

Давайте из таблицы сообщений удалим те записи, которые оставлял пользователь valera (id=2):

```
DELETE FROM posts
WHERE id_author='2';
```



Понятно, если опустить условие, то из таблицы будут удалены все данные. Следует помнить, что данные СУБД даст удалить только в том случае, если они не являются внешними ключами для данных из других таблиц (поддержка целостности БД). Например, если мы захотим удалить из таблицы users пользователя, который оставлял сообщения, то нам это не удастся.



Сначала надо удалить его сообщения, а уж потом и его самого.

Давайте подведем промежуточный итог. Мы умеем создавать таблицы и связывать их между собой, обновлять, редактировать и удалять данные и извлекать данные различным образом. В принципе - это можно назвать базовыми знаниями SQL. Далее мы будем изучать встроенные функции и расширенные возможности MySQL.